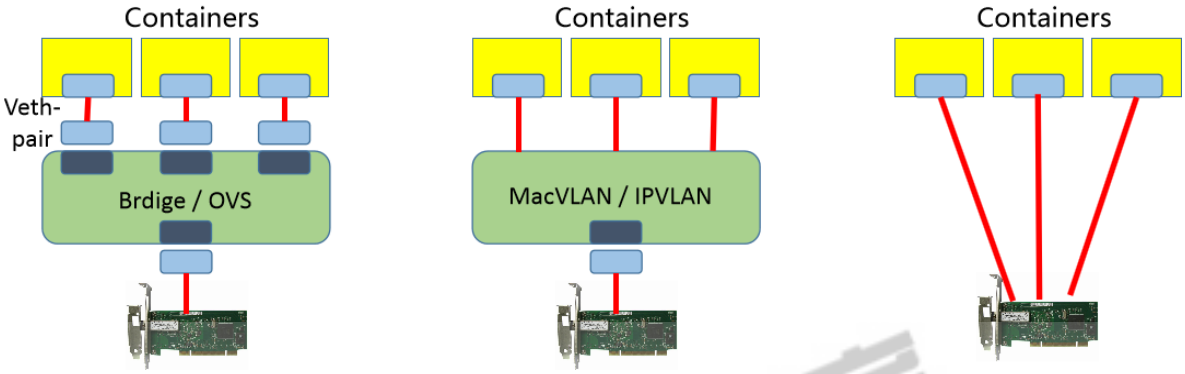
网络机制

网络基础

基础知识

pod接入网络的具体实现



1 虚拟网桥：

brdige，用纯软件的方式实现一个虚拟网络，用一个虚拟网卡接入到我们虚拟网桥上去。这样就能保证每一个容器和每一个pod都能有一个专用的网络接口，从而实现每一主机组件有网络接口。每一对网卡一半留在pod之上一半留在宿主机之上并接入到网桥中。甚至能接入到真实的物理网桥上能顾实现物理桥接的方式

2 多路复用：

MacVLAN，基于mac的方式去创建vlan，为每一个虚拟接口配置一个独有的mac地址，使得一个物理网卡能承载多个容器去使用。这样子他们就直接使用物理网卡并直接使用物理网卡中的MacVLAN机制进行跨节点之间进行通信了。

需要借助于内核级的VLAN模块来实现。

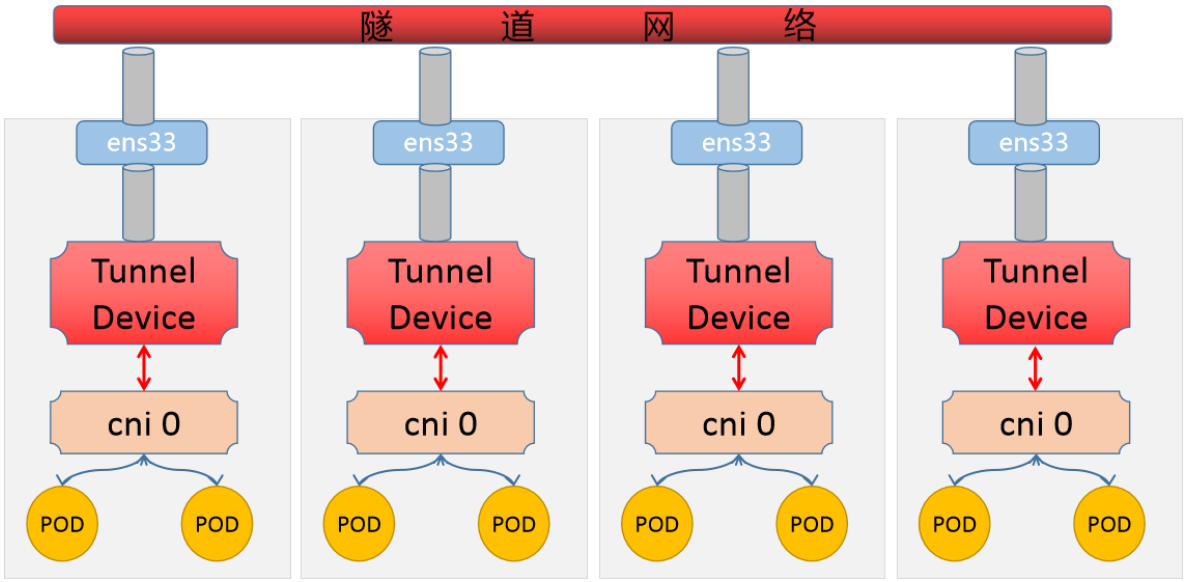
3 硬件交换：

使用支持单根IOV（SR-IOV）的方式，一个网卡支持直接在物理机虚拟出多个接口来，所以我们称为单根的网络连接方式，现在市面上的很多网卡都已经支持"单根IOV的虚拟化"了。它是创建虚拟设备的一种很高性能的方式，一个网卡能够虚拟出在硬件级多个网卡来。然后让每个容器使用一个网卡

相比来说性能肯定是硬件交换的方式效果更好，不过很多情况下我们用户期望去创建二层或三层的一些逻辑网络子网这就需要借助于叠加的网络协议来实现，所以会发现在多种解决方案中第一种叫使用虚拟网桥确实我们能够实现更为强大的控制能力的解决方案，但是这种控制确实实现的功能强大但多一点，他对网络传输来讲有额外的性能开销，毕竟他叫使用隧道网络，或者我们把它称之为叠加网络，要多封装IP守护或多封装mac守护，不过一般来讲我们使用这种叠加网络时控制平面目前而言还没有什么好的标准化，那么用起来彼此之间有可能不兼容，另外如果我们要使用VXLAN这种技术可能会引入更高的开销，这种方式给了用户更大的腾挪的空间

实现思路

对于任何一种第三方解决方案来说，如果它要实现k8s集群内部多节点间的pod通信都要从三个方面来实现：
1 构建一个网络
2 将pod接入到这个网络中
3 实时维护所有节点上的路由信息，实现隧道的通信



1 所有节点的内核都启用了VXLAN的功能模块，每个节点都有一个唯一的编号
节点内部的pod的跨节点通信需要借助于VXLAN内部的路由机制或隧道转发机制实现通信
每个cni0上维护了各个节点所在的隧道网段的路由列表
2 node上的pod发出请求到达cni0，根据内核的路由列表判断对端网段的节点在哪里
然后经由 隧道设备 对数据包进行封装标识，接下来对端节点的隧道设备解封标识数据包，
当前数据包一看当前节点的路由表发现有自身的ip地址，这直接交给本地的pod
3 多个节点上的路由表信息维护，就是各种网络解决方案的工作位置

注意：
这些解决方案，可以完成所有的步骤，也可以完成部分的功能，借助于其他方案实现完整的方案
但是需要注意的是：不要同时部署多个插件来做同一件事情，因为对于CNI来说，只会有一个生效。

常见插件

根据我们刚才对pod通信的回顾，多节点内的pod通信，k8s是通过CNI接口来实现网络通信的。CNI基本思想：创建容器时，先创建好网络名称空间，然后调用CNI插件配置这个网络，而后启动容器内的进程

CNI插件类别：main、meta、ipam
main，实现某种特定的网络功能，如loopback、bridge、macvlan、ipvlan
meta，自身不提供任何网络实现，而是调用其他插件，如flanne
ipam，仅用于分配IP地址，不提供网络实现

常见的CNI解决方案有：
Flannel
提供叠加网络，基于linux TUN/TAP，使用UDP封装IP报文来创建叠加网络，并借助etcd维护网络分配情况
Calico
基于BGP的三层网络，支持网络策略实现网络的访问控制。在每台机器上运行一个vRouter，利用内核转发数据包，并借助iptables实现防火墙等功能
Canal
由Flannel和Calico联合发布的一个统一网络插件，支持网络策略
Weave Net
多主机容器的网络方案，支持去中心化的控制平面，数据平面上，通过UDP封装实现L2 Overlay
Contiv
思科方案，直接提供多租户网络，支持L2(VLAN)、L3(BGP)、Overlay(VXLAN)
kube-router
K8s网络一体化解决方案，可取代kube-proxy实现基于ipvs的Service，支持网络策略、完美兼容BGP的高级特性

这些方案中，最值得关注的是：flannel、calico、kube-route

小结

flannel

原理介绍

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

集群节点的网络分配

```
root@master1:~# cat /etc/kubernetes/manifests/kube-controller-manager.yaml
apiVersion: v1
...
spec:
  containers:
  - command:
    - kube-controller-manager
    - --allocate-node-cidrs=true
    ...
    - --cluster-cidr=10.244.0.0/16
```

配置解析：

allocate-node-cidrs属性表示，每增加一个新的节点，都从cluster-cidr子网中切分一个新的子网网段分配给对应的节点上。

这些相关的网络状态属性信息，会经过 kube-apiserver 存储到etcd中。

注意：

我们在创建k8s集群的时候，当我们创建好集群主节点后，然后就开始安装flannel服务了，否则多个节点间无法正常通信，每个节点都会处于 NotReady 状态。

CNI接口

使用CNI插件编排网络，Pod初始化或删除时，kubelet会调用默认CNI插件，创建虚拟设备接口附加到相关的底层网络，设置IP、路由并映射到Pod对象网络名称空间.

kubelet在/etc/cni/net.d目录查找cni json配置文件，基于type属性到/opt/cni/bin中查找相关插件的二进制文件，然后调用相应插件设置网络

flannel现状

k8s节点上所有节点都运行了flannel容器

```
root@master1:~# kubectl get pod -n kube-system | grep flannel
kube-flannel-ds-68qs2              1/1     Running   1 (4d ago)      4d21h
kube-flannel-ds-gjv48              1/1     Running   5 (23h ago)     4d22h
kube-flannel-ds-zh5qv              1/1     Running   1 (4d ago)      4d22h
```

注意

由于flannel是以pod的样式存在，flannel 启动后就相当于在当前节点上启动了一个守护进程。该守护进程：

- 该进程会负责当前节点上的 所有报文封装解封等动作
- 通过 kube-apiserver 组件从集群的etcd服务中，获取每个节点的网络信息，并生成本地路由表信息。
- 启动要给本地的flannel.1网卡，配置相关的ip地址。

kube-apiserver为了更方便后续 flannel与etcd 直接的交流，单独分配一个url用于flannel和etcd的交流

-- 在二进制部署集群中可以看到效果。

每个节点上都生成了一个网卡 flannel.1

```
root@master1:~# ifconfig flannel.1
flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.0.0  netmask 255.255.255.255  broadcast 10.244.0.0

root@node1:~# ifconfig flannel.1
flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.1.0  netmask 255.255.255.255  broadcast 10.244.1.0

root@node2:~# ifconfig flannel.1
flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.244.2.0  netmask 255.255.255.255  broadcast 10.244.2.0
```

注意：

flannel.1 后面的.1 就是 vxlan的网络标识。便于隧道正常通信。

网段的分配原理

集群的 kube-controller-manager 负责控制每个节点的网段分配

集群的 etcd 负责存储所有节点的网络配置存储

集群的 flannel 负责各个节点的路由表定制及其数据包的拆分和封装

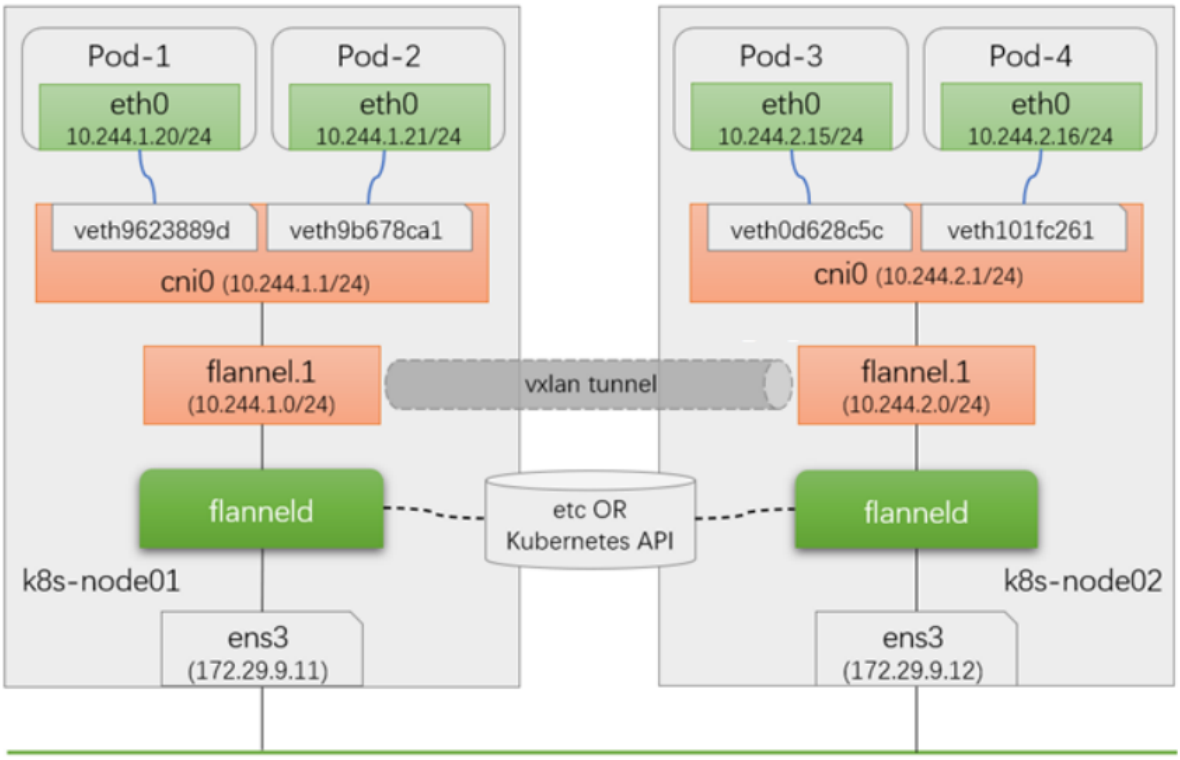
-- 所以flannel各个节点是平等的，仅负责数据平面的操作。网络功能相对来说比较简单。

另外一种插件 calico相对于flannel来说，多了一个控制节点，来管控所有的网络节点的服务进程。

原理解析

基本原理

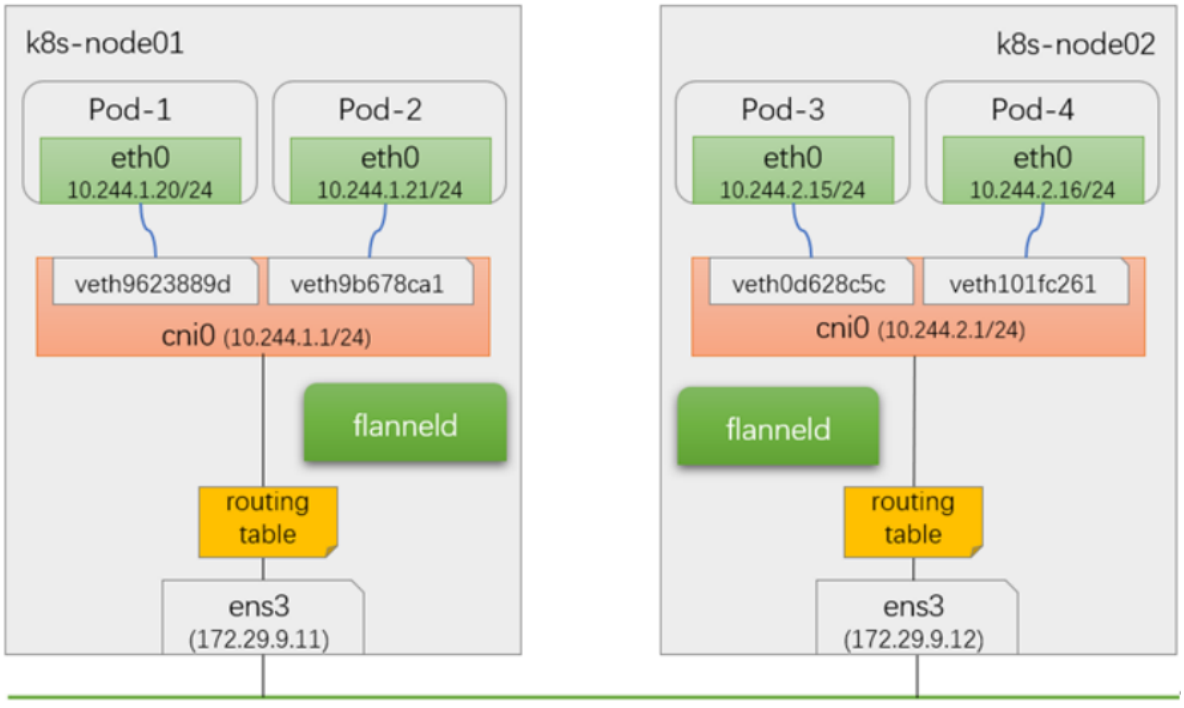
flannel默认就是以VXLAN的隧道模型来实现了正常的跨节点的网络通信。



1 节点上的pod通过虚拟网卡对，连接到cni0的虚拟网络交换机上
　　当有外部网络通信的时候，借助于 flannel.1网卡向外发出数据包
2 经过 flannel.1 网卡的数据包，借助于flanneld实现数据包的封装和解封
　　最后送给宿主机的物理接口，发送出去
3 对于pod来说，它以为是通过 flannel.x -> vxlan tunnel -> flannel.x 实现数据通信
　　因为它们的隧道标识都是".1"，所以认为是一个vxlan，直接路由过去了，没有意识到底层的通信机制。
注意：
　　由于这种方式，是对数据报文进行了多次的封装，降低了当个数据包的有效载荷。所以效率降低了

host-gw模式

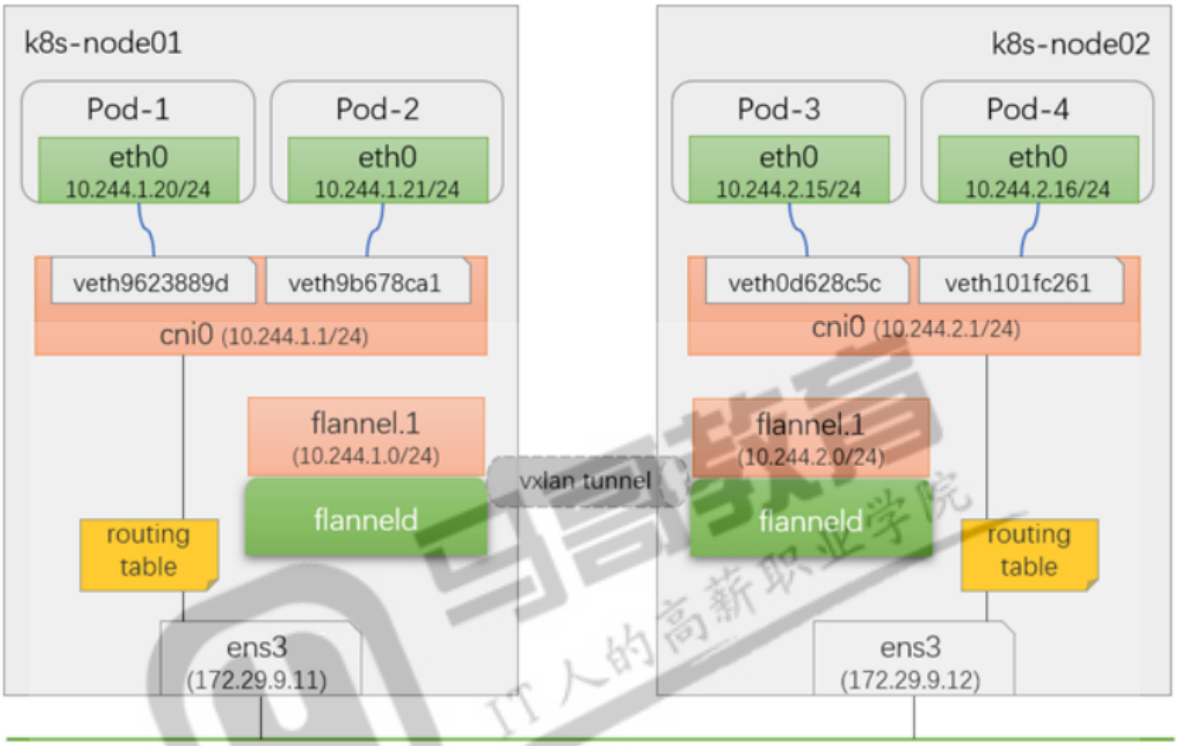
根据我们当前的实践来说，所有集群中的主机节点处于同一个可以直接通信的二层网络，本来就可以直接连通，那么还做二层的数据包封装，pod通信效率会非常差，所以flannel就出现了host-gw的通信模式



1 节点上的pod通过虚拟网卡对，连接到cni0的虚拟网络交换机上。
2 pod向外通信的时候，到达CNI0的时候，不再直接交给flannel.1由flanneld来进行打包处理了。
3 cni0直接借助于内核中的路由表，通过宿主机的网卡交给同网段的其他主机节点
4 对端节点查看内核中的路由表，发现目标就是当前节点，所以交给对应的cni0，进而找到对应的pod。

直连路由模式

根据我们对 host-gw模式的了解，我们发现，这种常见有一个限制，集群主机的节点必须处于同一个二层网络中。一旦出现节点无法正常通信，那么 host-gw模式就不能正常使用了。
所以，为了平衡上面的两种场景，就出现了第三种模式： VxLAN Directrouting模式



1 pod向外通信的时候，到达CNI0的时候，不再直接交给flannel.1由flanneld来进行打包处理了。
2 如果两个pod不是处于同一网段，那么还是通过源始的方式进行正常的隧道封装通信。
3 如果两个pod是处于同一网段内。
cni0直接借助于内核中的路由表，通过宿主机的网卡交给同网段的其他主机节点
对端节点查看内核中的路由表，发现目标就是当前节点，所以交给对应的cni0，进而找到对应的pod

注意：
这种Directrouting就大大提高了同网段间的跨节点的pod数据传输效率
所以，我们可以知道，flannel机制，不要求所有的节点都处于同一个二层网络中。

flannel模型

模型	解析
vxlan	pod与Pod经由隧道封装后通信，各节点彼此间能通信就行，不要求在同一个二层网络
vxlan directrouting	位于同一个二层网络上的、但不同节点上的Pod间通信，无须隧道封装；但非同一个二层网络上的节点上的Pod间通信，仍须隧道封装
host-gw	Pod与Pod不经隧道封装而直接通信，要求各节点位于同一个二层网络

原理解析

学习目标

这一节，我们从 方案解析、测试效果、小结 三个方面来学习。

方案解析

方案实现

```
我们曾经说过，一个合格的网络解决方案要实现多个功能：
    1 构建一个网络
    2 将pod接入到这个网络中
    3 实时维护所有节点上的路由信息，实现隧道的通信
但是我们不要求，每个解决方案都能实现所有的步骤功能，对于flannel来说，它实现的仅仅是第一步 --
构建虚拟网络的功能，而其他的步骤是需要借助于其他功能来实现。
```

```
比如要接入到虚拟网络的功能，我们在部署flannel的时候，有一个配置文件，在这个配置文件中的
configmap中就定义了虚拟网络的接入功能。
root@master1:~/mykubernetes/flannel# cat kube-flannel.yml
kind: ConfigMap
...
data:
  cni-conf.json: |                          cni插件的功能配置
    {
      "name": "cbr0",
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "flannel",                基于flannel实现网络通信
          "delegate": {
            "hairpinMode": true,
            "isDefaultGateway": true
          }
        },
        {
          "type": "portmap",                来实现端口映射的功能
          "capabilities": {
            "portMappings": true
          }
        }
      ]
    }
  net-conf.json: |                          flannel的网址分配
    {
      "Network": "10.244.0.0/16",
      "Backend": {
        "Type": "vxlan"                     来新节点的时候，基于vxlan从network中获取子网
      }
    }
```

每个节点上都有一个路由表

```
root@node1:~# route -n
```

```
root@master1:~/mykubernetes/flannel# route -n
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         10.0.0.2        0.0.0.0         UG    100    0        0 ens33
10.0.0.0        0.0.0.0         255.255.255.0   U     100    0        0 ens33
10.244.0.0      0.0.0.0         255.255.255.0   U     0      0        0 cni0
10.244.1.0      10.244.1.0      255.255.255.0   UG    0      0        0 flannel.1
10.244.2.0      10.244.2.0      255.255.255.0   UG    0      0        0 flannel.1
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

```
root@node1:~# route -n
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         10.0.0.2        0.0.0.0         UG    100    0        0 ens33
10.0.0.0        0.0.0.0         255.255.255.0   U     100    0        0 ens33
10.244.0.0      10.244.0.0      255.255.255.0   UG    0      0        0 flannel.1
10.244.1.0      0.0.0.0         255.255.255.0   U     0      0        0 cni0
10.244.2.0      10.244.2.0      255.255.255.0   UG    0      0        0 flannel.1
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

```
root@node2:~# route -n
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         10.0.0.2        0.0.0.0         UG    100    0        0 ens33
10.0.0.0        0.0.0.0         255.255.255.0   U     100    0        0 ens33
10.244.0.0      10.244.0.0      255.255.255.0   UG    0      0        0 flannel.1
10.244.1.0      10.244.1.0      255.255.255.0   UG    0      0        0 flannel.1
10.244.2.0      0.0.0.0         255.255.255.0   U     0      0        0 cni0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

结果显示：

如果数据包的目标是当前节点，这直接通过cni来进行处理

如果数据包的目标是其他节点，这根据路由配置，交给对应节点上的flannel.1网卡来进行处理

然后交给配套的flanneld对数据包进行封装

flannel.1子网内处理

当flannel.1接收到子网请求的时候，首先有flanneld进行数据包的解封。会从etcd中获取相关的网络细节配置。

为了避免每次请求都去etcd中获取相关的数据信息，所以会在第一次查询后，会在本地生成一个历史的查询记录

```
root@node2:~# ip neigh | grep flannel
10.244.1.0 dev flannel.1 lladdr 76:64:2c:29:12:07 PERMANENT
35.232.111.17 dev flannel.1  FAILED
35.224.170.84 dev flannel.1  FAILED
10.244.0.0 dev flannel.1 lladdr f6:0c:38:3a:f7:a9 PERMANENT
34.122.121.32 dev flannel.1  FAILED
```

注意：

这些条目记录，都是由flanneld来进行自动维护的，一旦节点丢弃或者关闭后，这里面的信息会自动更新的。

flannel的转发逻辑，对于节点上的数据包转发来说，它都在内核的fdb表中

- 如果涉及到转发的会有相关的目标
- 如果不涉及转发的会直接有对应的记录信息

```
root@node2:~# bridge fdb show flannel.1 | grep flannel.1
f6:0c:38:3a:f7:a9 dev flannel.1 dst 10.0.0.12 self permanent
76:64:2c:29:12:07 dev flannel.1 dst 10.0.0.15 self permanent
52:d3:56:06:5f:3f dev flannel.1 dst 10.0.0.12 self permanent
2a:a6:38:e3:ba:4a dev flannel.1 dst 10.0.0.12 self permanent
8a:b5:4c:82:6c:a4 dev flannel.1 dst 10.0.0.12 self permanent
```

前面的mac地址是本地对应dst ip地址的信息。

测试效果

- 默认效果

开启测试容器

```
root@master1:~# kubectl run pod-client --image="10.0.0.19:80/mykubernetes/admin-box:v0.1" -it --rm --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@pod-test # ifconfig
eth0      Link encap:Ethernet  HWaddr 9A:91:AF:24:CB:10
          inet addr:10.244.2.18  Bcast:10.244.2.255  Mask:255.255.255.0
          ...
```

查看节点部署效果

```
root@master1:~# kubectl get pod -o wide
NAME       READY   STATUS    RESTARTS   AGE     IP            NODE    NOMINATED NODE   READINESS GATES
pod-test   1/1     Running   0          4m30s   10.244.2.18   node2
```

开启一个测试容器

```
root@master1:~# cat 01-network-pod-test.yml
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
spec:
  nodeName: node1
  containers:
  - name: pod-test
    image: 10.0.0.19:80/mykubernetes/pod_test:v0.1

应用容器
kubectl apply -f 01-network-pod-test.yml

测试效果
root@master1:~# kubectl get pod -o wide
NAME         READY   STATUS    RESTARTS   AGE     IP             NODE    ...
pod-client   1/1     Running   0          2m25s   10.244.2.18    node2   ...
pod-test     1/1     Running   0          7s      10.244.1.56    node1   ...
```

在node1节点上启动一个数据包的抓取工具

注意：Flannel默认使用8285端口作为UDP封装报文的端口，VxLan使用8472端口

```
root@node1:~# tcpdump -i ens33 -en host 10.0.0.16  and udp port 8472
```

注意：

抓取从 10.0.0.16 主机过来的数据

在测试环境中发起ping测试

```
root@pod-client # ping -c 1 10.244.1.56
```

在node1上查看数据包抓取效果

```
root@node1:~# tcpdump -i ens33 -en host 10.0.0.16  and udp port 8472
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
12:12:28.374454 00:50:56:21:d2:7a > 00:50:56:2b:5e:38, ethertype IPv4 (0x0800), length 148: 10.0.0.16.38146 > 10.0.0.15
.8472: OTV, flags [I] (0x08), overlay 0, instance 1
6e:3d:65:7f:52:85 > 76:64:2c:29:12:07, ethertype IPv4 (0x0800), length 98: 10.244.2.18 > 10.244.1.56: ICMP echo request
, id 6912, seq 0, length 64
12:12:28.374905 00:50:56:2b:5e:38 > 00:50:56:21:d2:7a, ethertype IPv4 (0x0800), length 148: 10.0.0.15.46001 > 10.0.0.16
.8472: OTV, flags [I] (0x08), overlay 0, instance 1
76:64:2c:29:12:07 > 6e:3d:65:7f:52:85, ethertype IPv4 (0x0800), length 98: 10.244.1.56 > 10.244.2.18: ICMP echo reply,
id 6912, seq 0, length 64
```

结果显示：

这里面每一条数据，都包括了两层ip数据包

- 其他效果

更改flannel模式为 直连路由模式

开启flannel的 直连路由 模型。

```
# vim kube-flannel.yml
...
  net-conf.json: |
    {
      "Network": "10.244.0.0/16",
      "Backend": {
        "Type": "vxlan",
        "DirectRouting": true
      }
    }
```

重启pod

```
kubectl apply -f kube-flannel.yml
kubectl delete pod -n kube-system -l app=flannel
```

查看路由效果

```
root@master1:~/mykubernetes/flannel# route -n
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         10.0.0.2        0.0.0.0         UG    100    0        0 ens33
10.0.0.0        0.0.0.0         255.255.255.0   U     100    0        0 ens33
10.244.0.0      0.0.0.0         255.255.255.0   U     0      0        0 cni0
10.244.1.0      10.0.0.15       255.255.255.0   UG    0      0        0 ens33
10.244.2.0      10.0.0.16       255.255.255.0   UG    0      0        0 ens33
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

结果显示：

所有的路由转发，都不再使用flannel了，直接进行路由转发了

这是因为我们没有涉及到跨网段的主机节点

在node1上继续tcpdump抓包

```
root@node1:~# tcpdump -i ens33 -nn host 10.244.2.18 and tcp port 80
```

在测试容器里面使用curl来进行访问

```
root@pod-client # curl 10.244.1.56
kubernetes pod-test v0.1!! ClientIP: 10.244.2.18, ServerName: pod-test,
ServerIP: 10.244.1.56!
```

查看抓包效果

```
root@node1:~# tcpdump -i ens33 -nn host 10.244.2.18 and tcp port 80
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
12:38:35.527167 IP 10.244.2.18.33598 > 10.244.1.56.80: Flags [S], seq 3536504389, win 64860, options [mss 1410,sackOK,T
S val 2436082677 ecr 0,nop,wscale 7], length 0
12:38:35.527543 IP 10.244.1.56.80 > 10.244.2.18.33598: Flags [S.], seq 4150405849, ack 3536504390, win 64308, options [
mss 1410,sackOK,TS val 3526770141 ecr 2436082677,nop,wscale 7], length 0
12:38:35.527795 IP 10.244.2.18.33598 > 10.244.1.56.80: Flags [.], ack 1, win 507, options [nop,nop,TS val 2436082677 ec
r 3526770141], length 0
12:38:35.529647 IP 10.244.2.18.33598 > 10.244.1.56.80: Flags [P.], seq 1:76, ack 1, win 507, options [nop,nop,TS val 24
36082679 ecr 3526770141], length 75: HTTP: GET / HTTP/1.1
12:38:35.529725 IP 10.244.1.56.80 > 10.244.2.18.33598: Flags [.], ack 76, win 502, options [nop,nop,TS val 3526770144 e
cr 2436082679], length 0
12:38:35.531272 IP 10.244.1.56.80 > 10.244.2.18.33598: Flags [P.], seq 1:18, ack 76, win 502, options [nop,nop,TS val 3
526770145 ecr 2436082679], length 17: HTTP: HTTP/1.0 200 OK
12:38:35.531512 IP 10.244.1.56.80 > 10.244.2.18.33598: Flags [FP.], seq 18:249, ack 76, win 502, options [nop,nop,TS va
l 3526770145 ecr 2436082679], length 231: HTTP
12:38:35.531552 IP 10.244.2.18.33598 > 10.244.1.56.80: Flags [.], ack 18, win 507, options [nop,nop,TS val 2436082681 e
cr 3526770145], length 0
12:38:35.532911 IP 10.244.2.18.33598 > 10.244.1.56.80: Flags [F.], seq 76, ack 250, win 506, options [nop,nop,TS val 24
36082682 ecr 3526770145], length 0
12:38:35.532999 IP 10.244.1.56.80 > 10.244.2.18.33598: Flags [.], ack 77, win 502, options [nop,nop,TS val 3526770147 e
cr 2436082682], length 0
```

更改flannel模式为 host-gw模式

开启flannel的 直连路由 模型。

```
# vim kube-flannel.yml
...
  net-conf.json: |
    {
      "Network": "10.244.0.0/16",
      "Backend": {
        "Type": "host-gw"
      }
    }
```

重启pod

```
kubectl apply -f kube-flannel.yml
kubectl delete pod -n kube-system -l app=flannel
```

查看路由效果

```
root@master1:~/mykubernetes/flannel# route -n
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         10.0.0.2        0.0.0.0         UG    100    0        0 ens33
10.0.0.0        0.0.0.0         255.255.255.0   U     100    0        0 ens33
10.244.0.0      0.0.0.0         255.255.255.0   U     0      0        0 cni0
10.244.1.0      10.0.0.15       255.255.255.0   UG    0      0        0 ens33
10.244.2.0      10.0.0.16       255.255.255.0   UG    0      0        0 ens33
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

结果显示：

所有的路由转发，都不再使用flannel了，直接进行路由转发了

如果我们是第一次安装flannel的时候，使用这种模式，flannel.1网卡就不会生成了

在node1上继续tcpdump抓包
root@node1:~# tcpdump -i ens33 -nn host 10.244.2.18 and tcp port 80

在测试容器里面使用curl来进行访问
root@pod-client # curl 10.244.1.56
kubernetes pod-test v0.1!! ClientIP: 10.244.2.18, ServerName: pod-test,
ServerIP: 10.244.1.56!

由于这种模式下，与刚才的直连路由的方式一致，所以抓包效果是一样的。

小结

calico

网络模型

学习目标

这一节，我们从 基础知识、模型简介、小结 三个方面来学习。

基础知识

简介

Calico是一个开源的虚拟化网络方案,用于为云原生应用实现互联及策略控制.相较于 Flannel 来说,Calico 的优势是对网络策略(network policy),它允许用户动态定义 ACL 规则控制进出容器的数据报文,实现为 Pod 间的通信按需施加安全策略.不仅如此,Calico 还可以整合进大多数具备编排能力的环境,可以为 虚机和容器提供多主机间通信的功能。
Calico 本身是一个三层的虚拟网络方案,它将每个节点都当作路由器,将每个节点的容器都当作是节点路由器的一个终端并为其分配一个 IP 地址,各节点路由器通过 BGP(Border Gateway Protocol)学习生成路由规则,从而将不同节点上的容器连接起来.因此,Calico 方案其实是一个纯三层的解决方案,通过每个节点协议栈的三层（网络层）确保容器之间的连通性,这摆脱了 flannel host-gw 类型的所有节点必须位于同一二层网络的限制,从而极大地扩展了网络规模和网络边界。

官方地址：https://www.tigera.io/project-calico/
最新版本：v3.20.3 (20211006)

网络模型

Calico为了实现更广层次的虚拟网络的应用场景，它支持多种网络模型来满足需求。
underlay network - BGP
overlay network - IPIP、VXLAN

设计思想

Calico不使用隧道或者NAT来实现转发，而是巧妙的把所有二三层流量转换成三层流量，并通过host上路由配置完成跨host转发。

模型简介

BGP

BGP(Border Gateway Protocol – 边界网关协议) ，这是一种三层虚拟网络解决方案，也是Calico广为人知的一种网络模型。

它是互联网上一个核心的去中心化自治路由协议。

- 每个工作节点都是一个网络主机边缘节点。

  由一个虚拟路由(vrouter)和一系列其他节点组成的一个自治系统(AS)。

- 各个节点上的vrouter基于BGP协议，通过互相学习的方式，动态生成路由规则，实现各节点之间的网络互通性。

  BGP不使用传统的内部网关协议(IGP)的指标，而使用基于路径、网络策略或规则集来决定路由规则

  它属于矢量路由协议。

这也是很多人从flannel切换到Calico之上的一个重要的考虑因素，VGP是一种纯粹的三层路由解决方案，并没有叠加，而是一种承载网络。

它有点类似于flannel的host-gw模型，但是与host-gw的区别在于，host-gw是由flanneld通过kube-apiserver来操作etct内部的各种网络配置，进而实时更新到当前节点上的路由表中。

BGP方案中，各节点上的vRouter通过BGP协议学习生成路由表，基于路径、网络策略或规则集来动态生成路由规则

BGP模型根据主机容量来划分，可以分层两种网络模型：

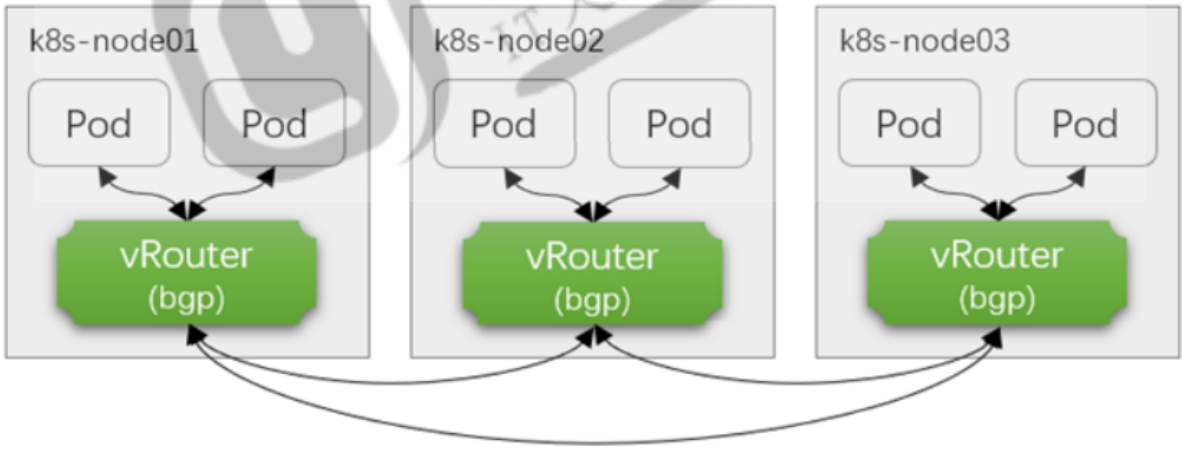
小规模网络：BGP peer，以一对多(N-1)的方式，彼此更新网络信息，主机量多的时候，性能不好。

因为每个节点都要进行(N-1)的网络信息更新，形成了mesh(网格)的效果

大规模网络：BGP Reflector，他以中央反射器的方式收集所有节点主机的信息，然后将收集后的信息，发布给所有节点，从而减轻集群中的路由同步的报文效率。但是我们需要对中央BGP反射器进行冗余处理。

简单来说，BGP就是通过动态生成的路由表的方式，以类似flannel的host-gw方式，来完成pod之间报文的直接路由，通信效率较高。

它要求所有的节点处于一个二层网络中，同时所有的主机也需要支持BGP协议。



注意：如果节点是跨网段的话，会因为目标地址找不到，而无法发送数据包

IPIP

通过把一个IP数据包又套在一个IP包里，即把IP层封装到IP层的一个 tunnel，实现跨网段效果

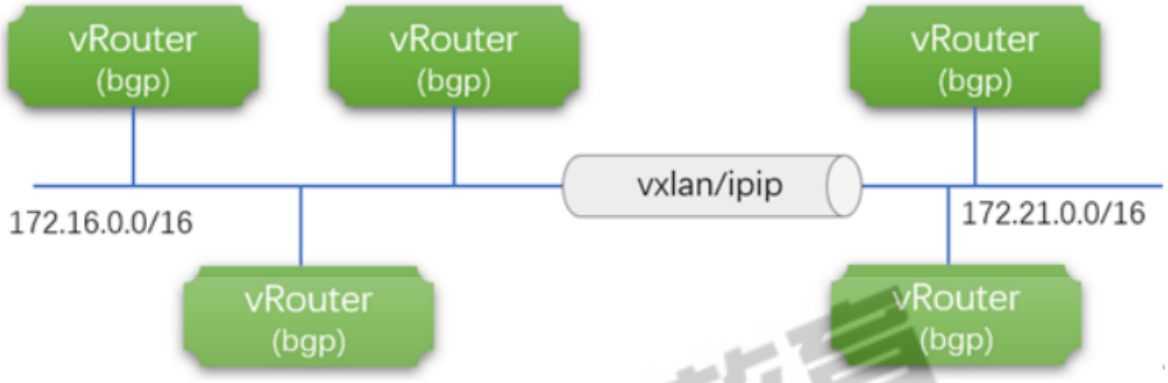
VXLAN

把数据包封装在UDP中，并使用物理网络的IP/MAC作为outer-header进行封装标识，然后在物理IP网上传输，"根据标识"到达目的地后由隧道终结点解封并将数据发送给目标虚拟机。

BGP要求所有的节点处于一个二层网络中，同时所有的主机也需要支持BGP协议。但是一旦我们要构建大规模的网络集群，比如是一个B类网站，放了很多的节点主机，这个时候，由于网络报文无法被隔离，所以一旦安全措施做的不到位的话，就会导致广播风暴，对我们产生很大的影响。所以，正常情况下，我们一般不推荐在一个子网内创建大量的节点主机。所以一旦跨子网，BGP就不支持了。

所以VXLAN就实现了这样一种类似于 flannel的 VXLAN with directrouting的机制，演变出了一种 Vxlan With BGP的效果。

- 如果是同一网段，就使用BGP
- 如果不是同一网段，就用VXLAN机制



小结

注意：

calico默认的配置清单中，使用的是 IPIP方式，因为它默认节点中不支持BGP协议

工作模型

这一节，我们从 软件架构、组件解析、小结 三个方面来学习。

软件架构

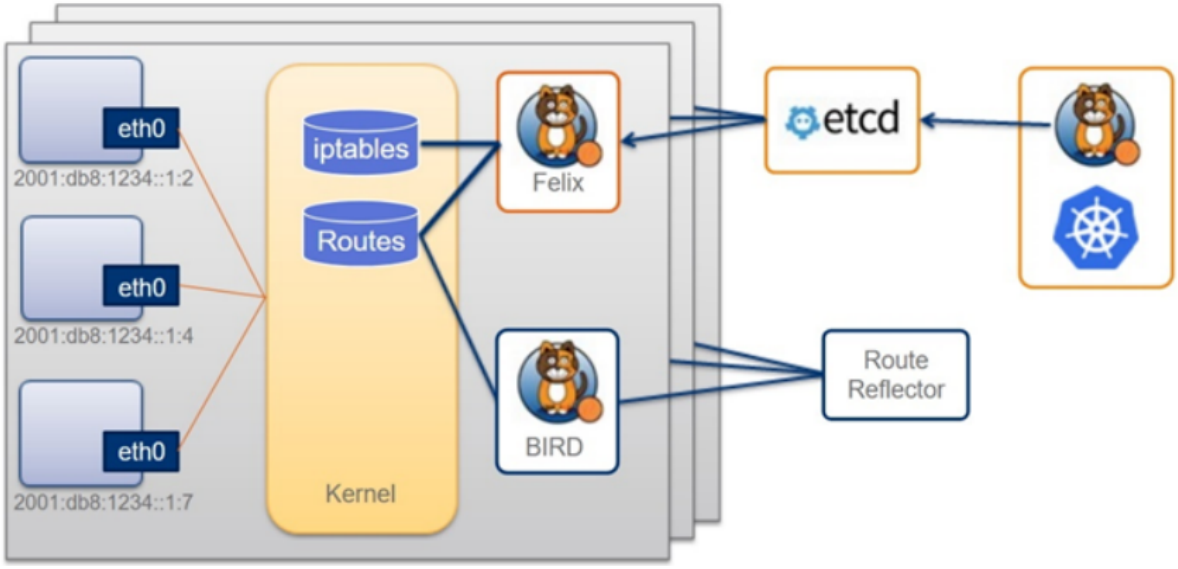
前提

如果我们要使用calico的时候，必须要提前将 flannel 功能禁用，然后再安装calico。因为对于k8s来说，cni配置的是谁，他就用谁。

安装方式

calico支持的模式很多，但是默认情况下，calico不知道我们的节点是否支持 BGP，所以，默认情况下，我们在安装calico的时候，它采用的模式是 IPIP模式。如果我们非要使用BGP模型的话，需要自己修改配置清单文件或者选择适合我们自己的BGP配置。
参考资料： https://docs.projectcalico.org/getting-started/kubernetes/quickstart

软件架构



注意：以下三点不是calico的组成部分。

1 每个黑色的背景就是一个边缘自治节点，里面有很多工作中的pod对象
2 每个pod对象不像flannel通过一对虚拟网卡连接到cni0，而是直接连接到内核中。
3 借助于内核中的iptables 和 routers(路由表)来完成转发的功能

关键组件

组件	解析
Felix	每个节点都有，负责配置路由、ACL、向etcd宣告状态等
BIRD	每个节点都有，负责把 Felix 写入Kernel的路由信息 分发到整个 Calico网络，确保 workload 连通
etcd	存储calico自己的状态数据，可以结合kube-apiserver来工作
Route Reflector	路由反射器，用于集中式的动态生成所有主机的路由表，非必须选项
Calico编排系统插件	实现更广范围的虚拟网络解决方案。

参考资料：https://docs.projectcalico.org/reference/architecture/overview

组件解析

总体

由于Calico是一种纯三层的实现，因此可以避免与二层方案相关的数据包封装的操作，中间没有任何的NAT，没有任何的overlay，所以它的转发效率可能是所有方案中最高的，因为它的包直接走原生TCP/IP的协议栈，它的隔离也因为这个栈而变得好做。因为TCP/IP的协议栈提供了一整套的防火墙的规则，所以它可以通过IPTABLES的规则达到比较复杂的隔离逻辑。

Calico网络模型主要工作组件 - Felix

运行在每一台host的agent进程，主要负责网络接口管理和监听、路由规划、ARP管理、ACL管理和同步、状态报告等。Felix会监听Etcd中心的存储，从它获取事件，比如说用户在这台机器上加了一个IP，或者是创建了一个容器等，用户创建Pod后，Felix负责将其网卡、IP、MAC都设置好，然后在内核的路由表里面写一条，注明这个IP应该到这张网卡。同样，用户如果制定了隔离策略，Felix同样会将该策略创建到ACL中，以实现隔离。

注意：这里的策略规则是通过内核的iptables方式实现的

Calico网络模型主要工作组件 - 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，可以与Kubernetes共用。简单来说：ETCD是所有calico节点的通信总线，也是calico对接到到其他编排系统的通信总线。

官方推荐；

- < 50节点,可以结合 kube-apiserver 来实现数据的存储
- > 50节点,推荐使用独立的ETCD集群来进行处理。

参考资料：https://docs.projectcalico.org/getting-started/kubernetes/self-managed-onprem/onpremises

Calico网络模型主要工作组件 - BIRD

Calico为每一台host部署一个BGP Client，使用BIRD实现，BIRD是一个单独的持续发展的项目，实现了众多动态路由协议比如：BGP、OSPF、RIP等。在Calico的角色是监听Host上由Felix注入的路由信息，然后通过BGP协议广播告诉剩余Host节点，从而实现网络互通。BIRD是一个标准的路由程序，它会从内核里面获取哪一些IP的路由发生了变化，然后通过标准BGP的路由协议扩散到整个其他的宿主机上，让外界都知道这个IP在这里，你们路由的时候得到这里来。

Calico网络模型主要工作组件 - Route Reflector

在大型网络规模中，如果仅仅使用BGP Client形成mesh全网互联的方案就会导致规模限制，因为所有节点之间两两互连，需要N^2个连接，为了解决这个规模问题，可以采用BGP的Route Reflector的方法，使所有BGP Client仅与特定RR节点互连并做路由同步，从而大大减少连接数。

小结

软件部署

这一节，我们从 准备工作、简单实践、小结 三个方面来学习。

准备工作

部署解析

对于calico在k8s集群上的部署来说，为了完成上面四个组件的部署，这里会涉及到两个部署组件

组件名	组件作用
calico-node	需要部署到所有集群节点上的代理守护进程，提供封装好的Felix和BIRD
calico-kube-controller	专用于k8s上对calico所有节点管理的中央控制器。负责calico与k8s集群的协同及calico核心功能实现。

部署思路

```
calico的官方对于k8s集群上的环境部署，提供了三种方式：
    小于50节点、大于50节点、专属的etcd节点等多种方式。

我们这里采用第一种，小于50节点的场景。

参考资料：
    https://docs.projectcalico.org/getting-started/kubernetes/self-managed-onprem/onpremises
```

部署步骤

```
1 获取资源配置文件
    从calico官网获取相关的配置信息
2 定制CIDR配置
    定制calico自身对于pod网段的配置信息，并且清理无关的网络其他插件
3 定制pod的manifest文件分配网络配置
    默认的k8s集群在启动的时候，会有一个cidr的配置，有可能与calico进行冲突，那么我们需要修改一下
4 应用资源配置文件
```

```
注意事项：
    对于calico来说，它自己会生成自己的路由表，如果路由表中存在响应的记录，默认情况下会直接使用，而不是覆盖掉当前主机的路由表
    所以如果我们在部署calico之前，曾经使用过flannel，尤其是flannel的host-gw模式的话，一定要注意，在使用calico之前，将之前所有的路由表信息清空，否则无法看到calico的tunl的封装效果
```

简单实践

- 环境部署

1 获取文件

```
curl https://docs.projectcalico.org/manifests/calico.yaml -O
cp calico.yaml calico-base.yaml
```

2 配置CIDR

```
# vim calico.yaml
---- 官网推荐的修改内容
3878             - name: CALICO_IPV4POOL_CIDR
3879               value: "10.244.0.0/16"
---- 方便我们的后续实验，新增调小子网段的分配
3880             - name: CALICO_IPV4POOL_BLOCK_SIZE
3881               value: "24"
```

配置解析：

开放默认注释的CALICO_IPV4POOL_CIDR变量，然后定制我们当前的pod的网段范围即可

原则上来说，我们修改官方提示的属性即可

3 定制pod的manifest文件分配网络配置

注意：这里我们直接让calico使用kube-controller-manager为节点分配的网段信息

```
# vim calico.yaml
---- 修改下面的内容
  34           "ipam": {
  35               "type": "calico-ipam"
  36           },
---- 修改后的内容
  34           "ipam": {
  35               "type": "host-local",
  36               "subnet": "usePodCidr"
  37           },
---- 定制calico使用k8s集群节点的地址
3882             - name: USE_POD_CIDR
3883               value: "true"
```

配置解析：

Calico默认并不会从Node.Spec.PodCIDR中分配地址，但可通过USE_POD_CIDR变量并结合host-local这一IPAM插件以强制从PodCIDR中分配地址

4 定制默认的docker镜像

```
查看默认的镜像文件
# grep docker.io calico.yaml
          image: docker.io/calico/cni:v3.20.2
          image: docker.io/calico/cni:v3.20.2
          image: docker.io/calico/pod2daemon-flexvol:v3.20.2
          image: docker.io/calico/node:v3.20.2
          image: docker.io/calico/kube-controllers:v3.20.2

修改为定制的镜像
sed -i 's#docker.io/calico#10.0.0.19:80/google_containers#g' calico.yaml

查看效果
# grep google calico.yaml
          image: 10.0.0.19:80/google_containers/cni:v3.20.2
          image: 10.0.0.19:80/google_containers/cni:v3.20.2
          image: 10.0.0.19:80/google_containers/pod2daemon-flexvol:v3.20.2
          image: 10.0.0.19:80/google_containers/node:v3.20.2
          image: 10.0.0.19:80/google_containers/kube-controllers:v3.20.2
```

5 应用资源配置文件

清理之前的flannel插件

```
kubectl delete -f kube-flannel.yml
kubectl get pod -n kube-system | grep flannel
```

这个时候，先清除旧网卡，然后最好重启一下主机，直接清空所有的路由表信息

```
ifconfig flannel.1
reboot
```

重启后，查看网络效果

```
root@node2:~# ifconfig | grep flags
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
root@node2:~# ip route list
default via 10.0.0.2 dev ens33 proto static metric 100
10.0.0.0/24 dev ens33 proto kernel scope link src 10.0.0.16 metric 100
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdow
```

```
root@master1:~# ifconfig | grep flags
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
root@master1:~# ip route list
default via 10.0.0.2 dev ens33 proto static metric 100
10.0.0.0/24 dev ens33 proto kernel scope link src 10.0.0.12 metric 100
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
```

使用自定义的镜像

应用calico插件

```
kubectl apply -f calico.yaml
```

在calico-node部署的时候，会启动多个进程

```
kubectl get pod -n kube-system | egrep 'NAME|calico'
```

```
root@master1:~/calico# kubectl get pod -n kube-system | egrep 'NAME|calico'
NAME                                       READY   STATUS     RESTARTS   AGE
calico-kube-controllers-6d97d7c96f-jw2tf   1/1     Running    0          2m35s
calico-node-67ww6                          0/1     Init:1/3   0          2m36s
calico-node-fb8kl                          0/1     Pending    0          2m36s
calico-node-gbgqh                          0/1     Init:2/3   0          2m36s
calico-node-scv9d                          0/1     Init:1/3   0          2m35s
calico-node-v7hp5                          1/1     Running    0          2m35s
```

环境部署完毕后，查看效果

```
kubectl get pod -n kube-system | egrep 'NAME|calico'
```

```
root@master1:~/calico# kubectl get pod -n kube-system | egrep 'NAME|calico'
NAME                                       READY   STATUS    RESTARTS   AGE
calico-kube-controllers-6d97d7c96f-jw2tf   1/1     Running   0          5m27s
calico-node-67ww6                          1/1     Running   0          5m28s
calico-node-fb8kl                          0/1     Pending   0          5m28s
calico-node-gbgqh                          1/1     Running   0          5m28s
calico-node-scv9d                          1/1     Running   0          5m27s
calico-node-v7hp5                          1/1     Running   0          5m27s
```

注意：这里的有一个一致处于Pending，是因为我为了节省主机资源，主动关闭了master3节点

```
root@master1:~/calico# ps aux | egrep 'NAME|calico'
root       62528  0.1  2.5 1436396 50440 ?       Sl   10:35   0:00 calico-node -confd
root       62535  2.6  2.7 1658104 55264 ?       Sl   10:35   0:06 calico-node -felix
root       62537  0.0  2.2 1288932 44748 ?       Sl   10:35   0:00 calico-node -monitor-token
root       62538  0.4  2.4 1141468 49992 ?       Sl   10:35   0:01 calico-node -monitor-addresses
root       62539  0.0  2.1 1214944 43592 ?       Sl   10:35   0:00 calico-node -allocate-tunnel-addrs
root       62650  0.0  0.0   1844  1292 ?        S    10:35   0:00 bird -R -s /var/run/calico/bird.ctl -d -c /etc/calico/confd/config/bird.cfg
root       62651  0.0  0.0   1728     4 ?        S    10:35   0:00 bird6 -R -s /var/run/calico/bird6.ctl -d -c /etc/calico/confd/config/bird6.cfg
root       66496  0.0  0.0  17688   672 pts/0    S+   10:39   0:00 grep -E --color=auto NAME|calico
```

每个calico节点上都有多个进程，分别来处理不同的功能

测试效果

创建一个deployment

```
kubectl create deployment pod-deployment --
image=10.0.0.19:80/mykubernetes/pod_test:v0.1 --replicas=3

查看pod
root@master1:~/calico# kubectl get pod
NAME                              READY   STATUS    RESTARTS   AGE
pod-deployment-554dff674-267gq    1/1     Running   0          48s
pod-deployment-554dff674-c8cjs    1/1     Running   0          48s
pod-deployment-554dff674-pxrwb    1/1     Running   0          48s

暴露一个service
kubectl expose deployment pod-deployment --port=80 --target-port=80

确认效果
kubectl get service
curl 10.108.138.97
```

```
root@master1:~/calico# kubectl get service
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP   24d
pod-deployment   ClusterIP   10.108.138.97   <none>        80/TCP    20s
root@master1:~/calico# curl 10.108.138.97
kubernetes pod-test v0.1!! ClientIP: 10.244.0.0, ServerName: pod-deployment-554dff674-pxrwb, ServerIP: 10.244.6.3!
root@master1:~/calico# curl 10.108.138.97
kubernetes pod-test v0.1!! ClientIP: 10.244.0.0, ServerName: pod-deployment-554dff674-267gq, ServerIP: 10.244.6.2!
root@master1:~/calico# curl 10.108.138.97
kubernetes pod-test v0.1!! ClientIP: 10.244.0.0, ServerName: pod-deployment-554dff674-c8cjs, ServerIP: 10.244.5.2!
```

- 网络解析

calico部署完毕后，会生成一系列的自定义配置属性信息

```
自动生成一个api版本信息
root@master1:~/calico# kubectl api-versions  | grep crd
crd.projectcalico.org/v1

该api版本信息中有大量的配置属性
kubectl api-resources  | grep crd.pro
```

```
root@master1:~/calico# kubectl api-resources  | grep crd.pro
bgpconfigurations                 crd.projectcalico.org/v1       false        BGPConfiguration
bgppeers                          crd.projectcalico.org/v1       false        BGPPeer
blockaffinities                   crd.projectcalico.org/v1       false        BlockAffinity
clusterinformations               crd.projectcalico.org/v1       false        ClusterInformation
felixconfigurations               crd.projectcalico.org/v1       false        FelixConfiguration
globalnetworkpolicies             crd.projectcalico.org/v1       false        GlobalNetworkPolicy
globalnetworksets                 crd.projectcalico.org/v1       false        GlobalNetworkSet
hostendpoints                     crd.projectcalico.org/v1       false        HostEndpoint
ipamblocks                        crd.projectcalico.org/v1       false        IPAMBlock
ipamconfigs                       crd.projectcalico.org/v1       false        IPAMConfig
ipamhandles                       crd.projectcalico.org/v1       false        IPAMHandle
ippools                           crd.projectcalico.org/v1       false        IPPool
kubecontrollersconfigurations     crd.projectcalico.org/v1       false        KubeControllersConfiguration
networkpolicies                   crd.projectcalico.org/v1       true         NetworkPolicy
networksets                       crd.projectcalico.org/v1       true         NetworkSet
```

```
这里面的 ippools 里面包含了calico相关的网络属性信息
root@master1:~/calico# kubectl get ippools
NAME                  AGE
default-ipv4-ippool   37m

查看这里配置的calico相关的信息
```

```
root@master1:~/calico# kubectl get ippools default-ipv4-ippool -o yaml
apiVersion: crd.projectcalico.org/v1
kind: IPPool
metadata:
  annotations:
    projectcalico.org/metadata: '{"uid":"3687f82b-838c-41b3-b84c-6a60fdb791b3","creationTimestamp":"2021-10-09T02:33:58Z"}'
  creationTimestamp: "2021-10-09T02:33:58Z"
  generation: 1
  name: default-ipv4-ippool
  resourceVersion: "1287699"
  uid: 272764fd-12c0-4b40-8186-191e9e85d0a6
spec:
  blockSize: 24
  cidr: 10.244.0.0/16
  ipipMode: Always
  natOutgoing: true
  nodeSelector: all()
  vxlanMode: Never
```

结果显式：

这里的calico采用的模型就是 ipip模型，分配的网段是使我们定制的 cidr网段，而且子网段也是我们定制的 24位掩码。

环境创建完毕后，会生成一个tunl0的网卡，所有的流量会走这个tunl0网卡

```
root@master1:~/calico# ifconfig | grep flags
califc9d1d28412: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1480
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
tunl0: flags=193<UP,RUNNING,NOARP>  mtu 1480
root@master1:~/calico# ip route list
default via 10.0.0.2 dev ens33 proto static metric 100
10.0.0.0/24 dev ens33 proto kernel scope link src 10.0.0.12 metric 100
blackhole 10.244.0.0/24 proto bird
10.244.0.2 dev califc9d1d28412 scope link
10.244.1.0/24 via 10.0.0.13 dev tunl0 proto bird onlink
10.244.5.0/24 via 10.0.0.15 dev tunl0 proto bird onlink
10.244.6.0/24 via 10.0.0.16 dev tunl0 proto bird onlink
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
```

```
root@node2:~# ifconfig | grep flags
cali29fc9c5aee4: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1480
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
tunl0: flags=193<UP,RUNNING,NOARP>  mtu 1480
root@node2:~# ip route list
default via 10.0.0.2 dev ens33 proto static metric 100
10.0.0.0/24 dev ens33 proto kernel scope link src 10.0.0.16 metric 100
10.244.0.0/24 via 10.0.0.12 dev tunl0 proto bird onlink
10.244.1.0/24 via 10.0.0.13 dev tunl0 proto bird onlink
10.244.5.0/24 via 10.0.0.15 dev tunl0 proto bird onlink
blackhole 10.244.6.0/24 proto bird
10.244.6.2 dev cali29fc9c5aee4 scope link
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
```

由于在calico的默认网络模型是 IPIP，所以我们在进行数据包测试的时候，可以通过直接抓取宿主机数据包，来发现双层ip效果

```
root@master1:~/calico# kubectl get pod -o wide
NAME                                 READY   STATUS    RESTARTS   AGE   IP           NODE    NOMINATED NODE   READINESS GATES
pod-deployment-76dd67889b-45l8h   1/1     Running   0          66s   10.244.6.3   node2   <none>           <none>
pod-deployment-76dd67889b-jqgwr   1/1     Running   0          66s   10.244.6.4   node2   <none>           <none>
pod-deployment-76dd67889b-zn5hp   1/1     Running   0          66s   10.244.5.4   node1   <none>           <none>
root@master1:~/calico# ping -c 1 10.244.6.3
PING 10.244.6.3 (10.244.6.3) 56(84) bytes of data.
64 bytes from 10.244.6.3: icmp_seq=1 ttl=63 time=0.493 ms
```

```
root@node2:~# tcpdump -i ens33 -nn ip host 10.0.0.16 and host 10.0.0.12
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
12:07:51.202472 IP 10.0.0.12 > 10.0.0.16: IP 10.244.0.1 > 10.244.6.3: ICMP echo request, id 3, seq 1, length 64 (ipip-proto-4)
12:07:51.203031 IP 10.0.0.16 > 10.0.0.12: IP 10.244.6.3 > 10.244.0.1: ICMP echo reply, id 3, seq 1, length 64 (ipip-proto-4)
```

结果显式：

每个数据包都是基于双层ip嵌套的方式来进行传输，而且协议是 ipip-proto-4

结合路由的分发详情，可以看到具体的操作效果。

具体效果：10.244.0.1 -> 10.0.0.12 -> 10.0.0.16 -> 10.244.6.3

小结

BGP网络

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

- 简单介绍

calico本身是一个复杂的系统，复杂到它自己提供一个非常重要的Restful接口，结合calicoctl命令来管理自身的相关属性信息，calicoctl可以直接与etcd进行操作，也可以通过kube-apiserver的方式与etcd来进行操作。默认情况下，它与kube-apiserver通信的认证方式与kubectl的命令使用同一个context。但是我们还是推荐，使用手工定制的一个配置文件。

calicoctl 是运行在集群之外的，用于管理集群功能的一个重要的组件。calicoctl 的安装方式很多，常见的方式有：单主机方式、kubectl命令插件方式、pod方式、主机容器方式。我们需要自己选择一种适合自己的方式

参考资料：https://docs.projectcalico.org/getting-started/clis/calicoctl/install

安装calicoctl

```
这里我们采用单主机命令的方式
cd /usr/local/bin/
curl -o calicoctl -O -L
"https://github.com/projectcalico/calicoctl/releases/download/v3.20.2/calicoctl"
chmod +x calicoctl

查看效果
root@master1:/usr/local/bin# calicoctl --help
Usage:
  calicoctl [options] <command> [<args>...]

参考资料：https://docs.projectcalico.org/getting-started/clis/calicoctl/install
```

关联kube-apiserver

```
手工定制的一个配置文件
# mkdir /etc/calico
# vim /etc/calico/calicoctl.cfg
apiVersion: projectcalico.org/v3
kind: CalicoAPIConfig
metadata:
spec:
  datastoreType: "kubernetes"
  kubeconfig: "/etc/kubernetes/admin.conf"

文件编辑完毕后，就可以直接来测试效果
root@master1:/etc/calico# calicoctl get nodes
NAME
master1
master2
master3
node1
node2
root@master1:/etc/calico# calicoctl get nodes node1 -o yaml
apiVersion: projectcalico.org/v3
kind: Node
metadata:
...
结果显式：
    我们可以通过 calicoctl 来操作kubectl操作的资源了，只不过显式的格式稍微有一点点的区别

参考资料：https://docs.projectcalico.org/getting-started/clis/calicoctl/configure/kdd
```

- 命令操作

命令的基本演示

```
查看ip的管理
calicoctl ipam --help

查看ip的信息
root@master1:/etc/calico# calicoctl ipam show
+----------+---------------+-----------+------------+--------------+
| GROUPING |     CIDR      | IPS TOTAL | IPS IN USE |   IPS FREE   |
+----------+---------------+-----------+------------+--------------+
| IP Pool  | 10.244.0.0/16 |     65536 | 0 (0%)     | 65536 (100%) |
+----------+---------------+-----------+------------+--------------+
```

```
查看信息的显式效果
calicoctl ipam show --help

显式相关的配置属性
root@master1:/etc/calico# calicoctl ipam show --show-configuration
+--------------------+-------+
|      PROPERTY      | VALUE |
+--------------------+-------+
| StrictAffinity     | false |
| AutoAllocateBlocks | true  |
| MaxBlocksPerHost   |     0 |
+--------------------+-------+
```

- kubectl命令插件

对于将calico整合到kubectl里面，方法非常简单，只需要将calicoctl 命令更改一个名字即可

```
定制kubectl 插件子命令
# cd /usr/local/bin/
# cp -p calicoctl kubectl-calico

测试效果
# kubectl calico --help
Usage:
  kubectl-calico [options] <command> [<args>...]

后续的操作基本上都一样了，比如获取网络节点效果
root@master1:~/calico# kubectl calico node status
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-------+----------+-------------+
| PEER ADDRESS |     PEER TYPE     | STATE |  SINCE   |    INFO     |
+--------------+-------------------+-------+----------+-------------+
| 10.0.0.13    | node-to-node mesh | up    | 03:55:44 | Established |
| 10.0.0.15    | node-to-node mesh | up    | 03:57:22 | Established |
| 10.0.0.16    | node-to-node mesh | up    | 03:57:19 | Established |
+--------------+-------------------+-------+----------+-------------+

IPv6 BGP status
```

```
No IPv6 peers found.
```

简单实践

简介

对于现有的calico环境，我们如果需要使用BGP环境，我们可以直接使用一个配置清单来进行修改calico环境即可。我们这里先来演示一下如何使用calicoctl修改配置属性。

获取当前的配置属性

```
root@master1:/usr/local/bin# kubectl calico get ipPools
NAME                  CIDR            SELECTOR
default-ipv4-ippool   10.244.0.0/16   all()

root@master1:/usr/local/bin# kubectl calico get ipPools default-ipv4-ippool -o
yaml
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  creationTimestamp: "2021-10-09T03:55:36Z"
  name: default-ipv4-ippool
  resourceVersion: "1296132"
  uid: ec5078ff-6c7d-4e84-89f4-ad2f36064097
spec:
  blockSize: 24
  cidr: 10.244.0.0/16
  ipipMode: Always
  natOutgoing: true
  nodeSelector: all()
  vxlanMode: Never
```

定制资源配置文件

```
kubectl calico get ipPools default-ipv4-ippool -o yaml > default-ipv4-
ippool.yaml
```

修改配置文件

```
root@master1:~/calico# cat default-ipv4-ippool.yaml
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: default-ipv4-ippool
spec:
  blockSize: 24
  cidr: 10.244.0.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
  nodeSelector: all()
  vxlanMode: Never
```

配置解析：

仅仅将原来的Always 更换成 CrossSubnet(跨节点子网) 模式即可

vxlanMode 的两个值可以实现所谓的 BGP with vxlan的效果

应用资源配置文件

```
root@master1:~/calico# kubectl calico apply -f default-ipv4-ippool.yaml
Successfully applied 1 'IPPool' resource(s)
```

```
root@master1:~/calico# kubectl calico apply -f default-ipv4-ippool.yaml
Successfully applied 1 'IPPool' resource(s)
root@master1:~/calico# ip route list
default via 10.0.0.2 dev ens33 proto static metric 100
10.0.0.0/24 dev ens33 proto kernel scope link src 10.0.0.12 metric 100
blackhole 10.244.0.0/24 proto bird
10.244.0.2 dev califc9d1d28412 scope link
10.244.1.0/24 via 10.0.0.13 dev ens33 proto bird
10.244.5.0/24 via 10.0.0.15 dev ens33 proto bird
10.244.6.0/24 via 10.0.0.16 dev ens33 proto bird
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
```

结果显式：

更新完毕配置后，动态路由的信息就发生改变了，不再完全是 `tunl0` 网卡了，而是变成了通过具体的物理网卡`ens33` 转发出去了。

再次来进行简单的测试

由于这次是直接通过节点进行转发的，所以我们在抓包的时候，直接通过内层ip抓取即可。

```
tcpdump -i ens33 -nn ip host 10.244.6.3
```

```
root@master1:~/calico# kubectl get pod -o wide
NAME                                READY   STATUS    RESTARTS   AGE    IP            NODE    NOMINATED NODE   READINESS GATES
pod-deployment-76dd67889b-45l8h     1/1     Running   0          113m   10.244.6.3    node2   <none>           <none>
pod-deployment-76dd67889b-jqgwr     1/1     Running   0          113m   10.244.6.4    node2   <none>           <none>
pod-deployment-76dd67889b-zn5hp     1/1     Running   0          113m   10.244.5.4    node1   <none>           <none>
root@master1:~/calico# ping -c 1 10.244.6.3
PING 10.244.6.3 (10.244.6.3) 56(84) bytes of data.
64 bytes from 10.244.6.3: icmp_seq=1 ttl=63 time=1.50 ms
root@node2:~# tcpdump -i ens33 -nn ip host 10.244.6.3
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
13:58:41.132067 IP 10.0.0.12 > 10.244.6.3: ICMP echo request, id 5, seq 1, length 64
13:58:41.132202 IP 10.244.6.3 > 10.0.0.12: ICMP echo reply, id 5, seq 1, length 64
```

结果显式：

可以实现正常的通信效果，没有再次使用ipip的方式了。

小结

BGP进阶

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

当前节点的网络效果

```
root@master1:~/calico# kubectl calico node status
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-------+----------+-------------+
| PEER ADDRESS |     PEER TYPE     | STATE |  SINCE   |    INFO     |
+--------------+-------------------+-------+----------+-------------+
| 10.0.0.13    | node-to-node mesh | up    | 03:55:44 | Established |
| 10.0.0.15    | node-to-node mesh | up    | 03:57:22 | Established |
| 10.0.0.16    | node-to-node mesh | up    | 03:57:19 | Established |
+--------------+-------------------+-------+----------+-------------+

IPv6 BGP status
No IPv6 peers found.
```

当前的calico节点的网络状态是 BGP peer 的模型效果，也就是说 每个节点上都要维护 n-1 个路由配置信息，整个集群中需要维护 n*(n-1) 个路由效果，这在节点量非常多的场景中，不是我们想要的。

所以我们需要实现一种 BGP reflecter 的效果。

反射器方案

一个集群中，至少需要一个反射器角色。

如果我们要做 BGP reflecter 效果的话，需要对反射器角色做冗余，如果我们的集群是一个多主集群的话，可以将集群的master节点作为bgp的reflecter角色，从而实现反射器的冗余。

操作步骤

1 定制反射器角色
2 后端节点使用反射器
3 关闭默认的网格效果

简单实践

1 创建反射器角色

定制资源定义文件

```
apiVersion: projectcalico.org/v3
kind: Node
metadata:
  labels:
    route-reflector: true
  name: master1
spec:
  bgp:
    ipv4Address: 10.0.0.12/16
    ipv4IPIPTunnelAddr: 10.244.0.1
    routeReflectorClusterID: 1.1.1.1
```

属性解析；

metadata.labels 是非常重要的，因为它需要被后面的节点来进行获取

metadata.name 的属性，必须是通过 calicoctl get nodes 获取到的节点名称。

spec.bgp.ipv4Address必须是 指定节点的ip地址

spec.bgp.ipv4IPIPTunnelAddr必须是 指定节点上tunl0的地址

spec.bgp.routeReflectorClusterID 是BGP网络中的唯一标识，所以这里的集群标识只要唯一就可以了

应用资源配置文件

```
kubectl calico apply -f 01-calico-reflector-master.yaml
```

2 更改节点的网络模型为 reflecter模型

定制资源定义文件

```
kind: BGPPeer
apiVersion: projectcalico.org/v3
metadata:
  name: bgppeer-demo
spec:
  nodeSelector: all()
  peerSelector: route-reflector=="true"
```

属性解析；
    spec.nodeSelector 指定的所有后端节点
    spec.peerSelector 指定的是反射器的标签，标识所有的后端节点与这个反射器进行数据交流

应用资源配置文件
```
kubectl calico apply -f 02-calico-reflector-bgppeer.yaml
```

查看效果
```
kubectl calico node status
```

```
root@master1:~/calico# kubectl calico node status
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-------+----------+-------------+
| PEER ADDRESS |     PEER TYPE     | STATE |  SINCE   |    INFO     |
+--------------+-------------------+-------+----------+-------------+
| 10.0.0.13    | node-to-node mesh | up    | 03:55:44 | Established |
| 10.0.0.15    | node-to-node mesh | up    | 03:57:22 | Established |
| 10.0.0.16    | node-to-node mesh | up    | 03:57:19 | Established |
| 10.0.0.13    | node specific     | start | 06:28:27 | Idle        |
| 10.0.0.15    | node specific     | start | 06:28:27 | Idle        |
| 10.0.0.16    | node specific     | start | 06:28:27 | Idle        |
+--------------+-------------------+-------+----------+-------------+
```

结果显式：
    这个时候，节点的状态发生了改变

3 关闭默认的网格效果

定制资源定义文件
```
apiVersion: projectcalico.org/v3
kind: BGPConfiguration
metadata:
  name: default
spec:
  logSeverityScreen: Info
  nodeToNodeMeshEnabled: false
  asNumber: 63400
```

属性解析；
    metadata.name 在这里最好是default，因为我们要对BGP默认的网络效果进行关闭
    spec.nodeToNodeMeshEnabled 关闭后端节点的BGP peer默认状态 -- 即点对点通信关闭
    spec.asNumber 指定的是后端节点间使用反射器的时候，我们要在一个标志号内，这里随意写一个

应用资源配置文件
```
kubectl calico apply -f 03-calico-reflector-defaultconfig.yaml
```

查看效果
```
kubectl calico node status
```

```
root@master1:~/calico# kubectl calico node status
Calico process is running.

IPv4 BGP status
+--------------+---------------+-------+----------+-------------+
| PEER ADDRESS |   PEER TYPE   | STATE |  SINCE   |    INFO     |
+--------------+---------------+-------+----------+-------------+
| 10.0.0.13    | node specific | up    | 06:34:39 | Established |
| 10.0.0.15    | node specific | up    | 06:34:39 | Established |
| 10.0.0.16    | node specific | up    | 06:34:38 | Established |
+--------------+---------------+-------+----------+-------------+

IPv6 BGP status
No IPv6 peers found.
```

结果显式：
    默认的点对点通信方式就已经丢失了，剩下了反射器模式

小结

网络策略

学习目标

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

- k8s策略

想象中的策略

到现在为止，我们之前操作过的k8s资源对象中，跟策略相关的无非就是
    权限认证 - 权限认证主要是与用户登录和资源使用有关系
    命名空间 - 命名空间可以将我们相关的资源进行隔离，但是我们可以通过 "命名空间.资源对象" 的方式来进行资源通信。

而实际上，k8s中对于资源控制的策略要远远的超出我们之前的理解。

k8s的资源策略体系

在k8s的从策略体系中，策略的内容非常庞大，不过我们可以根据自己的理解，抽取出来最主要的四个方面：：
    1 资源的使用限制 - limit range
    2 资源的应用配额 - resource quota
    3 资源的安全控制 - podsecritypolicy
    4 资源的网络策略 - network policy，主要来管控资源的 通信流量的

注意：

这里的策略也会被k8s集群中的控制器转换成内核的iptables的规则，只不过与service转换的规则不同。
    service会被转换成iptables的net 或 mangle 表的规则
    资源策略中的规则会被转换成 iptables的filter表上的规则

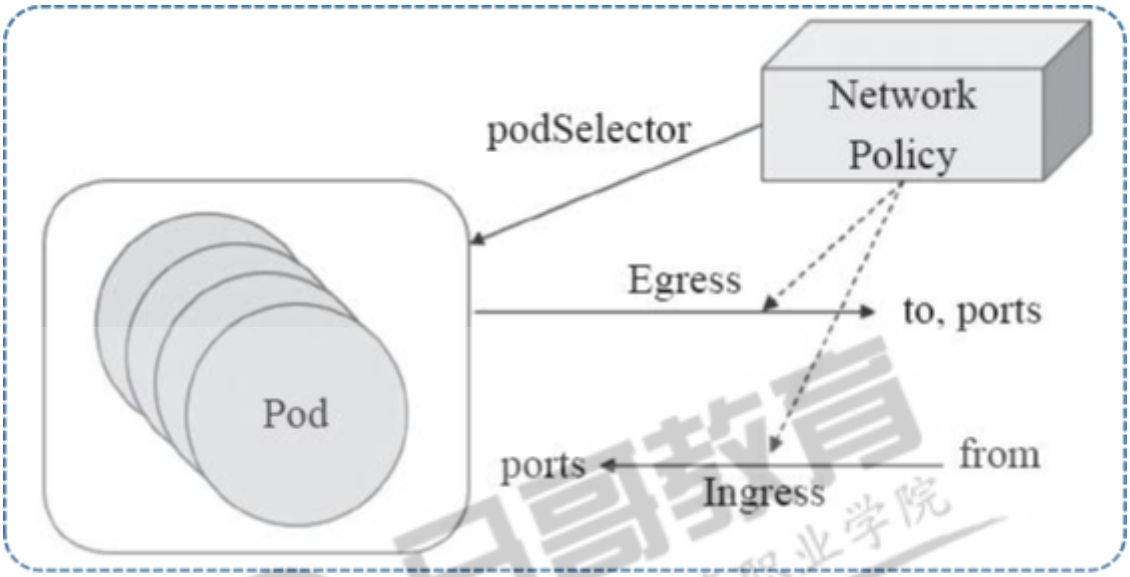
网络策略

为了使用Network Policy，Kubernetes引入了一个新的资源对象Network Policy，供用户设置Pod间网络访问的策略。但仅定义一个网络策略是无法完成实际的网络隔离的，还需要一个策略控制器(Policy Controller)进行策略的实现。

网络策略功能由网络插件实现，支持网络策略的插件有 Calico、Canal、kube-router等，在k8s环境中，专门基于这些策略管理软件的操作对象叫策略控制器,最常见的是networkpolicies。

策略控制器用于监控指定区域创建对象(pod)时所生成的新API端点，并按需为其附加网络策略。

对于Pod对象来说，网络流量分为 流入(Ingress)和流出(Egress)两个方向，每个方向包含允许和禁止两种控制策略，默认情况下，所有的策略都是允许的，应用策略后，所有未经明确允许的流量都将拒绝。



相关术语：

Pod组，NetworkPolicy通过Pod选择器选定的一组Pod

Egress，出站流量，由流量的目标网络端口 to 和端口 ports 定义

Ingress，入站流量，由流量发出的源站点 from 和流量的目标端口定义

Selector: 选择pod对象的一种机制：

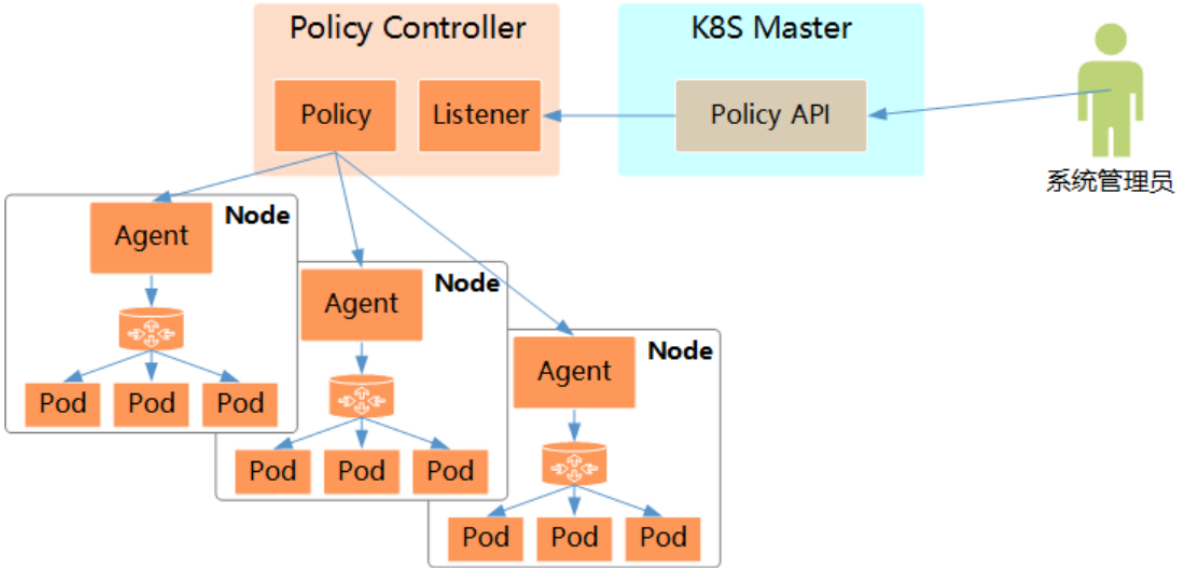
基于命名空间选择pod、基于网段来选择pod、基于pod标签来选择pod

- 配置解析

实践细节

为实现细粒度的容器间网络访问隔离策略，Kubernetes发布Network Policy，目前已升级为networking.k8s.io/v1稳定版本。

Network Policy的主要功能是对Pod间的网络通信进行限制和准入控制，设置方式为将Pod的Label作为查询条件，设置允许访问或禁止访问的客户端Pod列表。目前查询条件可以作用于Pod和Namespace级别。



资源对象属性

```
apiVersion: networking.k8s.io/v1    # 资源隶属的API群组及版本号
kind: NetworkPolicy                 # 资源类型的名称，名称空间级别的资源；
metadata:                           # 资源元数据
    name <string>                   # 资源名称标识
    namespace <string>              # NetworkPolicy是名称空间级别的资源
spec:                               # 期望的状态
    podSelector <Object>            # 当前规则生效的一组目标Pod对象，必选字段；空值表示
当前名称空间中的所有Pod资源
    policyTypes <[]string>          # Ingress表示生效ingress字段；Egress表示生效
egress字段，同时提供表示二者均有效
    ingress <[]Object>              # 入站流量源端点对象列表，白名单，空值表示“所有”
    - from <[]Object>               # 具体的端点对象列表，空值表示所有合法端点
      - ipBlock  <Object>           # IP地址块范围内的端点，不能与另外两个字段同时使用
      - namespaceSelector <Object>  # 匹配的名称空间内的端点
        podSelector <Object>        # 由Pod标签选择器匹配到的端点，空值表示<none>
      ports <[]Object>              # 具体的端口对象列表，空值表示所有合法端口
    egress <[]Object>               # 出站流量目标端点对象列表，白名单，空值表示“所有”
    - to <[]Object>                 # 具体的端点对象列表，空值表示所有合法端点，格式同
ingres.from;
      ports <[]Object>              # 具体的端口对象列表，空值表示所有合法端口

注意：
    入栈和出栈哪个策略生效，由 policyTypes 来决定。
    如果仅配置了podSelector，表明，当前限制仅限于当前的命名空间
```

配置示例

```
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: 10.244.0.0/16
        except:
        - 10.244.1.0/24
    - namespacesSelector:
        matchLabels:
          project: develop
    - podSelector:
        matchLabels:
          arch: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.244.0.0/24
      ports:
      - protocol: TCP
        port: 3306
```

简单实践

准备工作

```
定制资源配置文件
root@master1:~/calico# cat 04-calico-deployment-test.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: develop
  labels:
    kubernetes.io/metadata.name: develop
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: pod-deployment
  name: pod-deployment
  namespace: develop
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pod-deployment
  template:
    metadata:
      labels:
        app: pod-deployment
    spec:
      containers:
      - image: 10.0.0.19:80/mykubernetes/pod_test:v0.1
        name: pod-test-thkxw
```

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: pod-deployment
  name: pod-deployment
  namespace: develop
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: pod-deployment
  type: ClusterIP
```

应用资源配置文件

```
kubectl apply -f 04-calico-deployment-test.yaml
```

查看效果

```
kubectl get all -n develop
```

```
root@master1:~/calico# kubectl get all -n develop -o wide
NAME                                  READY   STATUS    RESTARTS   AGE    IP            NODE    NOMINATED NODE   READINESS GATES
pod/pod-deployment-76dd67889b-7gzhf   1/1     Running   0          3m4s   10.244.6.12   node2   <none>           <none>
pod/pod-deployment-76dd67889b-s4m69   1/1     Running   0          3m3s   10.244.6.11   node2   <none>           <none>
pod/pod-deployment-76dd67889b-wqnvm   1/1     Running   0          3m3s   10.244.5.10   node1   <none>           <none>

NAME                     TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE   SELECTOR
service/pod-deployment   ClusterIP   10.96.80.213   <none>        80/TCP    87s   app=pod-deployment

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE    CONTAINERS       IMAGES                                      SELECTOR
deployment.apps/pod-deployment   3/3     3            3           3m4s   pod-test-thkxw   10.0.0.19:80/mykubernetes/pod_test:v0.1   app=pod-deplo
yment
```

访问效果

```
kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 --rm -it --command -- /bin/sh
```

```
root@master1:~/mykubernetes# kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 --rm -it --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.11, ServerName: pod-deployment-76dd67889b-7gzhf, ServerIP: 10.244.6.12!
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.11, ServerName: pod-deployment-76dd67889b-s4m69, ServerIP: 10.244.6.11!
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.11, ServerName: pod-deployment-76dd67889b-wqnvm, ServerIP: 10.244.5.10!
```

结果显式：

可以正常的访问

流量管控

学习目标

这一节，我们从 默认策略、同ns策略、跨ns策略、小结 三个方面来学习。

默认策略

设置默认的拒绝 pod管控策略

为了安全定制所有的pod都无法正常的访问

```
root@master1:~/calico# cat 05-calico-networkpolicy-denyall.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: deny-all-ingress
  namespace: develop
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]
```

配置解析：

这里面的 podSelector 值为空，表示当前命名空间中所有的pod

policyType中指明了 Ingress 和 Egress。但是没有定义任何ingress字段，表示不允许所有pod有流量通过

应用资源配置文件

```
kubectl apply -f 05-calico-networkpolicy-denyall.yaml
```

查看效果

```
root@pod-test # curl 10.96.80.213
curl: (28) Failed to connect to 10.96.80.213 port 80: Operation timed out
```

结果显式：

所有的流量都被拒绝了

设置默认的允许 pod管控策略

为了安全定制所有的pod都正常的访问

```
root@master1:~/calico# cat 06-calico-networkpolicy-denyall.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: develop
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]
  egress:
  - {}
  ingress:
  - {}
```

配置解析：

在下面分别指定了egress和ingress，虽然没有配置，表示所有的pod都不采用默认拒绝策略 -- 全部接收请求

应用资源配置文件

```
kubectl apply -f 06-calico-networkpolicy-denyall.yaml
```

查看效果

```
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.12, ServerName: pod-deployment-76dd67889b-wqnvm, ServerIP: 10.244.5.10!
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.12, ServerName: pod-deployment-76dd67889b-7gzhf, ServerIP: 10.244.6.12!
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.12, ServerName: pod-deployment-76dd67889b-s4m69, ServerIP: 10.244.6.11!
```

结果显式：

所有的pod都可以正常访问了

进一步设置，仅允许当前命名空间所有流量允许访问

```
root@master1:~/calico# cat 07-calico-networkpolicy-denyall.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: develop
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]
  egress:
  - to:
    - podSelector: {}
  ingress:
  - from:
    - podSelector: {}
```

配置解析：

虽然设置了egress和ingress属性，但是下面的podSelector没有选择节点，表示只有当前命名空间所有节点不受限制

应用资源配置文件

```
kubectl apply -f 07-calico-networkpolicy-denyall.yaml
```

查看效果

```
root@master1:~/mykubernetes# kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 -n develop --rm -it --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.15, ServerName: pod-deployment-76dd67889b-wqnvm, ServerIP: 10.244.5.10!
root@pod-test # exit
Session ended, resume using 'kubectl attach pod-test -c pod-test -i -t' command when the pod is running
pod "pod-test" deleted
root@master1:~/mykubernetes# kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 --rm -it --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@pod-test # curl 10.96.80.213
curl: (28) Failed to connect to 10.96.80.213 port 80: Operation timed out
```

结果显示：

只有同名空间的资源可以正常访问，而跨ns的资源是无法被访问

同ns策略

提前准备两个测试的pod

终端1

```
kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 -n develop
--rm -it --command -- /bin/sh
```

终端2

```
kubectl run pod-test1 --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 -n
develop --rm -it --command -- /bin/sh
```

确认效果

```
root@master1:~/calico# kubectl get pod -n develop -o wide --show-labels | egrep
'NAME|test'
NAME       READY   STATUS    RESTARTS   AGE     IP             NODE   ...   LABELS
pod-test   1/1     Running   0          2m22s   10.244.6.13    node2 ...
run=pod-test
pod-test1 1/1      Running   0          2m6s    10.244.5.18    node1 ...
run=pod-test1
```

同一命名空间的pod相互访问控制

定制资源配置文件

```
root@master1:~/calico# cat 08-calico-networkpolicy-denyall.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: develop
spec:
  podSelector: {}
  policyTypes: ["Ingress", "Egress"]
  egress:
  - to:
    - podSelector: {}
  ingress:
  - from:
    - ipBlock:
        cidr: 10.244.0.0/16
        except:
        - 10.244.6.0/24
    - podSelector:
        matchLabels:
          run: pod-test1
    ports:
    - protocol: TCP
      port: 80
```

配置解析：

虽然设置了egress和ingress属性，但是下面的podSelector没有选择节点，表示只有当前命名空间所有节点不受限制

应用资源配置文件

```
kubectl apply -f 08-calico-networkpolicy-denyall.yaml
```

查看效果

```
root@pod-test1 # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.18, ServerName: pod-deployment-76dd67889b-s4m69, ServerIP: 10.244.6.11!
root@pod-test1 # nslookup 10.96.80.213
;; connection timed out; no servers could be reached

root@pod-test # curl 10.96.80.213
curl: (28) Failed to connect to 10.96.80.213 port 80: Operation timed out
```

结果显示：

在 pod-test 中无法执行 curl 10.96.80.213

pod-test1 可以正常访问svc的ip地址，由于出栈的时候没有办法对dns域名进行解析，所以无法curl域名

为了更好的对外进行服务访问，我们在增加一个出栈的规则

定制资源配置文件

```
root@master1:~/calico# cat 09-calico-networkpolicy-egress.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-controller
  namespace: develop
```

```
spec:
  podSelector:
    matchLabels:
      run: pod-test1
  policyTypes: ["Egress"]
  egress:
  - to:
    ports:
    - protocol: UDP
      port: 53
  - to:
    - podSelector:
        matchLabels:
          run: pod-test1
    ports:
    - protocol: TCP
      port: 80
```

应用资源配置文件

```
kubectl apply -f 09-calico-networkpolicy-egress.yaml
```

检查效果

```
root@pod-test1 # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.18, ServerName: pod-deployment-76dd67889b-wqnvm, ServerIP: 10.244.5.10!
root@pod-test1 # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.18, ServerName: pod-deployment-76dd67889b-7gzhf, ServerIP: 10.244.6.12!
root@pod-test1 # nslookup pod-deployment
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   pod-deployment.develop.svc.cluster.local
Address: 10.96.80.213
```

结果显示：

现在的pod不仅仅可以正常的进行服务访问，还可以对外发起nslookup的域名解析

清理所有的空间限制

命名空间

需求

大量的规则，会导致我们无法正常工作，或者艰难工作。

- 多个规则可能导致冲突、重复
- 多个规则的先后执行顺序导致结果不一样

所以我们在定制规则的时候，最好放到一个配置文件中。或者所有的规则都是经过精心管理和模块化管理的时候。而名称空间的控制就属于一种模块化管理方式。

开放一个小口口，只允许一个pod发送信息访问

查看当前的测试pod的命名空间的标签

```
root@master1:~/calico# kubectl get ns --show-labels
NAME                   STATUS   AGE   LABELS
default                Active   24d   kubernetes.io/metadata.name=default
develop                Active   30m   kubernetes.io/metadata.name=develop
kube-node-lease        Active   24d   kubernetes.io/metadata.name=kube-node-
lease
```

```
kube-public            Active   24d   kubernetes.io/metadata.name=kube-public
kube-system            Active   24d   kubernetes.io/metadata.name=kube-system
kubernetes-dashboard   Active   24d   kubernetes.io/metadata.name=kubernetes-
dashboard
```

为了安全定制所有的pod都无法正常的访问

```
root@master1:~/calico# cat 10-calico-networkpolicy-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-controller
  namespace: develop
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - namespaceSelector:
        matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: In
          values: [develop, kube-system, logs]
  - from:
    - namespaceSelector:
        matchExpressions:
        - {key: kubernetes.io/metadata.name, operator: NotIn, values:
["default"]}
```

配置解析：

只允许default命名空间的pod可以访问，其他的地方不允许访问

应用资源配置文件

```
kubectl apply -f 10-calico-networkpolicy-denyall.yaml
```

查看效果

```
root@master1:~/mykubernetes# kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 --rm -it --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.13, ServerName: pod-deployment-76dd67889b-s4m69, ServerIP: 10.244.6.11!
root@pod-test # curl 10.96.80.213
kubernetes pod-test v0.1!! ClientIP: 10.244.5.13, ServerName: pod-deployment-76dd67889b-wqnvm, ServerIP: 10.244.5.10!
root@pod-test # exit
Session ended, resume using 'kubectl attach pod-test -c pod-test -i -t' command when the pod is running
pod "pod-test" deleted
root@master1:~/mykubernetes# kubectl run pod-test --image=10.0.0.19:80/mykubernetes/admin-box:v0.1 -n develop --rm -it --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@pod-test # curl 10.96.80.213
curl: (28) Failed to connect to 10.96.80.213 port 80: Operation timed out
```

结果显式：

只有允许通过的命名空间的pod才可以发出访问，其他的都不允许

小结

其他方案

学习目标

这一节，我们从基础知识、简单实践、小结 三个方面来学习。

基础知识

问题

尽管k8s自己的NetworkPolicy功能上日渐丰富，但k8s自己的NetworkPolicy资源仍然具有相当的局限性，例如它没有明确的拒绝规则、缺乏对选择器高级表达式的支持、不支持应用层规则，以及没有集群范围的网络策略等。

根据我们的实践可知，每个networkpolicy都是以当前命名空间为中心，进行的网络策略控制。如果命名空间过多的话，会导致我们无法正常工作，或者艰难工作。 而这属于k8s网络管控的固有缺陷。

资源对象

我们安装完毕calico之后，会自动生成一些自定义的资源对象，这些资源对象包括NetworkPolicy和GlobalNetworkPolicy等.

```
root@master1:~/mykubernetes/calico# kubectl api-resources  | grep crd.pro
bgpconfigurations                 crd.projectcalico.org/v1               false        BGPConfiguration
bgppeers                          crd.projectcalico.org/v1               false        BGPPeer
blockaffinities                   crd.projectcalico.org/v1               false        BlockAffinity
clusterinformations               crd.projectcalico.org/v1               false        ClusterInformation
felixconfigurations               crd.projectcalico.org/v1               false        FelixConfiguration
globalnetworkpolicies             crd.projectcalico.org/v1               false        GlobalNetworkPolicy
globalnetworksets                 crd.projectcalico.org/v1               false        GlobalNetworkSet
hostendpoints                     crd.projectcalico.org/v1               false        HostEndpoint
ipamblocks                        crd.projectcalico.org/v1               false        IPAMBlock
ipamconfigs                       crd.projectcalico.org/v1               false        IPAMConfig
ipamhandles                       crd.projectcalico.org/v1               false        IPAMHandle
ippools                           crd.projectcalico.org/v1               false        IPPool
kubecontrollersconfigurations     crd.projectcalico.org/v1               false        KubeControllersConfi
guration
networkpolicies                   crd.projectcalico.org/v1               true         NetworkPolicy
networksets                       crd.projectcalico.org/v1               true         NetworkSet
```

其中的NetworkPolicy CRD比Kubernetes NetworkPolicy API提供了更大的功能集，包括支持拒绝规则、规则解析级别以及应用层规则等，但相关的规则需要由calicoctl创建。

以GlobalNetworkPolicy为例，它支持使用selector、serviceAccountSelector或namespaceSelector来选定网络策略的生效范围，默认为all()，即集群上的所有端点。

高级场景：

转发流量、防御DOS

资源	解析
NetworkPolicy	简称 np；是命名空间级别资源。规则应用于与标签选择器匹配的endpoint的集合
GlobalNetworkPolicy	简称 gnp / gnps 与 NetworkPolicy 功能一样，是整个集群级别的资源

简单实践

- GlobalNetworkPolicy

简单示例

定制资源配置文件

```
apiVersion: crd.projectcalico.org/v1
kind: GlobalNetworkPolicy
metadata:
  name: namespaces-default
spec:
  order: 0.0
```

```
  namespaceSelector: name not in {"kube-system","kubernetes-
dashboard","logs","monitoring"}
  types: ["Ingress", "Egress"]
  ingress:
  - action: Allow
    source:
      namespaceSelector: name in {"kube-system","kubernetes-
dashboard","logs","monitoring"}
  egress:
  - action: Allow
```

他的特点在于，在默认规则的基础上，添加了order这个优先级的属性，所以多个同类型的规则同时存在的时候，会根据优先级的方式，进行优先级进行使用。

```
应用配置文件
kubectl apply -f 11-calico-networkpolicy-global.yaml

检查效果
kubectl get globalnetworkpolicy
kubectl describe globalnetworkpolicy namespaces-default
```

测试效果

```
准备工作
kubectl run pod-test --image=10.0.0.19:80/mykubernetes/pod_test:v0.1
# kubectl get pod -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP           NODE
pod-test   1/1     Running   0          28s   10.244.2.2   node2

default空间测试
# kubectl run admin-box-default --image=10.0.0.19:80/mykubernetes/admin-box:v0.1
--rm -it --command -- /bin/sh
If you don't see a command prompt, try pressing enter.
root@admin-box-default # curl 10.244.2.2
长时间无法查看，超时退出

kube-system空间查看
```

收尾措施

```
这个知识点做完后，一定要移除所有的策略，否则就会导致问题
kubectl delete -f networkpolicy-global.yaml
```

小结

马哥教育
IT人的高薪职业学院