在 IPv6 集群上部署 Cilium

利用 Linux 原生路由在二层和三层网络上组建 Kubernetes 集群.

---

1. Cilium 配置

集群通过 kubeadm bootstrap, 网络插件使用 Cilium.

Cilium 版本是 1.19.0.

```
cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:             OK
 \__/¯¯\__/    Operator:           OK
 /¯¯\__/¯¯\    Envoy DaemonSet:    OK
 \__/¯¯\__/    Hubble Relay:       OK
    \__/       ClusterMesh:        disabled

DaemonSet              cilium                   Desired: 3, Ready: 3/3,
Available: 3/3
DaemonSet              cilium-envoy             Desired: 3, Ready: 3/3,
Available: 3/3
Deployment             cilium-operator          Desired: 1, Ready: 1/1,
Available: 1/1
Deployment             hubble-relay             Desired: 1, Ready: 1/1,
Available: 1/1
```

可以看到, `cilium` DaemonSet 在 3 个节点上运行. 其中两台 Dell R630 与网关在同一个交换机下, 另一台软路由通过 WireGuard 连接到网关. 这个拓扑同时包含二层直连和三层转发路径.

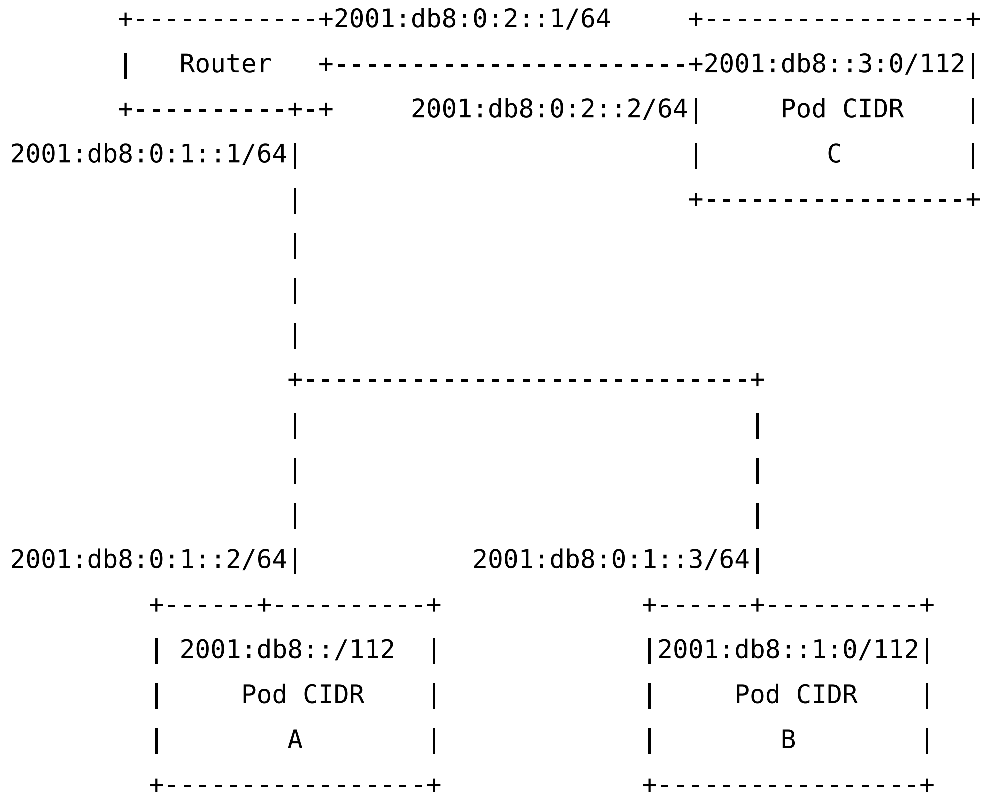

图 1 A, B 和 C 的拓扑

运行

```
cilium connectivity perf
```

集群吞吐可以达到 10 Gbps, 接近该环境中 10G 网络的上限.

此外, Cilium 会为 Pod 配置与宿主机相同的 MTU, Pod 间互联也可以使用巨型帧.

Helm values 中与网络路径相关的配置如下:

```yaml
routingMode: native
ipam:
  mode: kubernetes
ipv4:
  enabled: false
ipv6:
  enabled: true
operator:
  replicas: 1
k8s:
  requireIPv6PodCIDR: true
  requireIPv4PodCIDR: false
kubeProxyReplacement: true
autoDirectNodeRoutes: true
```

```yaml
directRoutingSkipUnreachable: true
ipv6NativeRoutingCIDR: "2001:db8::/32"
enableIPv6Masquerade: false
bpf:
  lbExternalClusterIP: true
hubble:
  enabled: true
  relay:
    enabled: true
```

2. Linux 原生路由

其中和路由路径直接相关的是以下几项:

```yaml
# 原生路由模式, 不使用 VXLAN/Geneve 隧道封装
routingMode: native
# 使用 Cilium eBPF datapath 替代 kube-proxy 处理 Kubernetes Service 转发
kubeProxyReplacement: true
# 如果各节点位于同一个二层网络, 自动在节点之间添加到对端 PodCIDR 的直连路由
autoDirectNodeRoutes: true
# 该范围内的目的地址交给 Linux 路由栈处理, 不需要做 SNAT
ipv6NativeRoutingCIDR: "2001:db8::/32"
# 禁用 Cilium 对 Pod/endpoint 发出的 IPv6 流量做 masquerade
enableIPv6Masquerade: false
# 允许从集群外访问 ClusterIP 类型的 Service
bpf:
  lbExternalClusterIP: true
```

Kubernetes 网络通常需要满足以下目标:

1. 节点之间相互通信
2. 同一节点的 Pod 相互通信
3. 不同节点的 Pod 相互通信
4. Pod 和宿主节点通信
5. Pod 和其他节点通信
6. 实现 Service CIDR 转发

对应 图 1, 在原生路由下, 网关配置路由表后, 除 Service CIDR 转发外的目标都可以由 Linux 路由完成.

```
2001:db8:0:1::/64 dev eth0
2001:db8::/112 via 2001:db8:0:1::1 dev eth0
```

```
2001:db8::1:0/112 via 2001:db8:0:1::2 dev eth0

2001:db8:0:2::/64 dev wg0
2001:db8::2:0/112 via 2001:db8:0:2::1 dev wg0
```

Service CIDR 的流量由 Cilium 的 BPF service load balancer 处理. 这不一定表现为 nftables 里的 DNAT 规则: 常见路径是在 socket 层把 Service frontend 改写成某个 backend; 如果 socket load balancer 不适用, 再落到 TC/TCX 的 packet path, 在接口 上的 BPF 程序里做 service lookup 和地址改写.

在以上的配置下, 任何一个节点都可以接受目的地址为 Service CIDR 的包.

2.1. Cilium 在哪里接管

从节点上的 nftables 也能看到这个边界. 这里并不是 kube-proxy 在 `nat` 表里展开一串 `KUBE-SVC-*` 规则的模型. 在这个 IPv6-only 集群里, Kubernetes 侧主要留下 `KUBE-KUBELET-CANARY` 之类的探针链, 数据面入口已经变成 Cilium 自己的链:

```
table ip6 raw {
  chain PREROUTING {
    type filter hook prerouting priority raw; policy accept;
    jump CILIUM_PRE_raw
  }

  chain OUTPUT {
    type filter hook output priority raw; policy accept;
    jump CILIUM_OUTPUT_raw
  }
}

table ip6 mangle {
  chain PREROUTING {
    type filter hook prerouting priority mangle; policy accept;
    jump CILIUM_PRE_mangle
  }
}

table ip6 nat {
  chain POSTROUTING {
    type nat hook postrouting priority srcnat; policy accept;
    jump CILIUM_POST_nat
  }
```

```
  chain CILIUM_POST_nat {
  }
}

table ip6 filter {
  chain INPUT {
    type filter hook input priority filter; policy accept;
    jump CILIUM_INPUT
  }

  chain OUTPUT {
    type filter hook output priority filter; policy accept;
    jump CILIUM_OUTPUT
  }

  chain FORWARD {
    type filter hook forward priority filter; policy accept;
    jump CILIUM_FORWARD
  }
}
```

这些规则说明的是 Cilium 在 netfilter 上留下的辅助路径, 而不是 Service rewrite 本身:

1. 外部进入节点的 IPv6 包会先经过 `PREROUTING`, 并跳到 Cilium 的预处理链.
2. 宿主机自己发出的包会在 `OUTPUT` 跳到 `CILIUM_OUTPUT_raw` / `CILIUM_OUTPUT`.
3. 需要转发的包会在 `FORWARD` 跳到 `CILIUM_FORWARD`. 其中 `cilium_host`, `cilium_net`, `lxc*` 这些接口名说明, Pod veth, Cilium host 设备和节点转发路径 都已经被纳入 Cilium 的数据面.

Service rewrite 的位置需要从 BPF 侧确认. Cilium agent 的状态能看到 kube-proxy replacement 已经启用, native routing 之外的主机侧路径也由 BPF 接管:

```
kubectl -n kube-system exec ds/cilium -- cilium-dbg status --verbose
KubeProxyReplacement:   True
Routing:                Network: Native   Host: BPF
Attach Mode:            TCX
```

已经加载的 BPF 程序中, 和 Service 处理关系最直接的是两类:

```
bpftool prog show
cgroup_sock_addr  name cil_sock6_connect
cgroup_sock_addr  name cil_sock6_sendmsg
```

```
cgroup_sock_addr  name cil_sock6_recvmsg
sched_cls         name cil_from_container
sched_cls         name cil_from_netdev
sched_cls         name cil_to_netdev
```

cil_sock6_connect, cil_sock6_sendmsg, cil_sock6_recvmsg 属于 cgroup socket hook. Pod 或宿主机进程连接 Service IP 时, Cilium 可以在这些 hook 中查询 service map, 选择 backend, 并把 socket 的目标地址改成 backend 地址. 这一步发生在内核创建实际网络包之前, 所以 nftables 里不会出现对应的 DNAT 规则.

cil_from_container, cil_from_netdev, cil_to_netdev 等 sched_cls 程序则挂在 TC/TCX packet path 上. 当流量没有走 socket-LB, 或者需要在接口层继续处理时, 这些程序会用同一套 service map 做查表和转发.

Service frontend 和 backend 的对应关系保存在 Cilium 的 BPF LB map 当中:

```
kubectl -n kube-system exec ds/cilium -- cilium-dbg bpf lb list
SERVICE ADDRESS                     BACKEND ADDRESS
[2001:db8:10:96::a]:53/UDP            [2001:db8:1::10]:53/UDP
[2001:db8:10:96::a]:53/UDP            [2001:db8:1::11]:53/UDP
```

第一列是 Service frontend, 第二列是实际 backend. socket-LB 和 TC/TCX 程序都会使用这类 map 做选择和改写; 区别只在于改写发生在 socket 层还是 packet path 上.