# 浅谈K8S cni和网络方案

叁叁肆 2018-11-30 16:38



目前不论是个人还是企业，在使用k8s时，都会采用CNI作为集群网络方案实现的规范。
在早先的k8s版本中，kubelet代码里提供了networkPlugin,networkPlugin是一组接口，实现了pod的网络配置、解除、获取，当时kubelet的代码中有个一个docker_manager，负责容器的创建和销毁，亦会负责容器网络的操作。而如今我们可以看到基本上kubelet的启动参数中，networkPlugin的值都会设置为cni。

## cni插件的使用方式

使用CNI插件时，需要做三个配置：

- kubelet启动参数中networkPlugin设置为cni
- 在/etc/cni/net.d中增加cni的配置文件，配置文件中可以指定需要使用的cni组件及参数
- 将需要用到的cni组件（二进制可执行文件）放到/opt/cni/bin目录下

所有的cni组件都支持两个命令：add和del。即配置网络和解除网络配置。
cni插件的配置文件是一个json文件，不同版本的接口、以及不同的cni组件,有着不同的配置内容结构，目前比较通用的接口版本是0.3.1的版本。
在配置文件中我们可以填入多个cni组件，当这些cni组件的配置以数组形式记录时，kubelet会对所有的组件进行按序链式调用，所有组件调用成功后，视为网络配置完成，过程中任何一步出现error，都会进行回滚的del操作。以保证操作流上的原子性。

## 几种基本的cni插件

cni插件按照代码中的存放目录可以分为三种：ipam、main、meta。

- ipam cni用于管理ip和相关网络数据，配置网卡、ip、路由等。
- main cni用于进行网络配置，比如创建网桥，vethpair、macvlan等。

叁叁肆
这个世界会好吗
454篇博客

最新博客

2019年微服务5大趋势，你pick哪个?
LinkedBlockingQueue源码解析（3）
LinkedBlockingQueue源码解析（2）
LinkedBlockingQueue源码解析（1）
企业项目开发--本地缓存guava cache（2）
企业项目开发--本地缓存guava cache（1）
AtomicInteger源码解析
Google guava cache源码解析1--构建缓存器（3）
Google guava cache源码解析1--构建缓存器（2）
Google guava cache源码解析1--构建缓存器（1）

最新资源下载

不谈虚的，给传统企业一份代码级中台落地实践

## ipam类CNI

ipam类型的cni插件，在执行add命令时会分配一个IP给调用者。执行del命令时会将调用者指定的ip放回ip池。社区开源的ipam有host-local、dhcp。

### host-local

我们可以通过host-local的配置文件的数据结构来搞懂这个组件是如何管理ip的。

```
type IPAMConfig struct {
    *Range
    Name       string
    Type       string         `json:"type"`
    Routes     []*types.Route `json:"routes"`//交付的ip对应的路由
    DataDir    string         `json:"dataDir"`//本地ip池的数据库目录
    ResolvConf string         `json:"resolvConf"`//交付的ip对应的dns
    Ranges     []RangeSet     `json:"ranges"`//交付的ip所属的网段，网关信息
    IPArgs     []net.IP       `json:"-"` // Requested IPs from CNI_ARGS and args
}

#配置文件范例:
{
    "cniVersion": "0.3.1",
    "name": "mynet",
    "type": "ipvlan",
    "master": "foo0",
    "ipam": {
        "type": "host-local",
        "resolvConf": "/home/here.resolv",
        "dataDir": "/home/cni/network",
        "ranges": [
            [
                {
                    "subnet": "10.1.2.0/24",
                    "rangeStart": "10.1.2.9",
                    "rangeEnd": "10.1.2.20",
                    "gateway": "10.1.2.30"
                },
                {
                    "subnet": "10.1.4.0/24"
                }
```

微服务框架在多个行业的架构设计与落地实践
微服务系统设计实践
微服务实践痛点与解决方案
网易云：技术驱动·产品为王-汪源
新形势下的互联网安全攻与防_CNCERT_20180413
安卓App安全过检实践 – 网易云 朱星星
UGC产品如何规避运营风险

编辑推荐

一网打尽！2018网络安全事件最全的盘点


Facebook内部报告：争取青少年用户的鸡贼小技巧


企业项目开发--maven父子模块（1）


【译文】东京的外国工程师

```
                "rangeEnd": "11.1.2.20",
                "gateway": "11.1.2.30"
            }]
        ]
    }
}
```

从上面的配置我们可以清楚：

- host-local组件通过在配置文件中指定的subnet进行网络划分
- host-local在本地通过指定目录（默认为/var/lib/cni/networks）记录当前的ip pool数据
- host-local将IP分配并告知调用者时，还可以告知dns、路由等配置信息。这些信息通过配置文件和对应的resolv文件记录。

host-local的应用范围比较广，kubenet、bridge、ptp、ipvlan等cni network插件都被用来和host-local配合进行ip管理。

### dhcp

社区的cni组件中就包含了dhcp这个ipam，但并没有提供一个可以参考的案例，翻看了相关的源码，大致逻辑是：

- 向dhcp申请ip时，dhcp会使用rpc访问本地的socket（/run/cni/dhcp.sock）申请一个ip的租约。然后将IP告知调用者。
- 向dhcp删除IP时，dhcp同样通过rpc请求，解除该IP的租约。

## main（network）类CNI

main类型的cni组件做的都是一些核心功能，比如配置网桥、配置各种虚拟化的网络接口（veth、macvlan、ipvlan等）。这里我们着重讲使用率较高的bridge和ptp。

### bridge

brige模式，即网桥模式。在node上创建一个linux bridge，并通过vethpair的方式在容器中设置网卡和IP。只要为容器配置一个二层可达的网关：比如给网桥配置IP，并设置为容器ip的网关。容器的网络就能建立起来。
如下是bridge的配置项数据结构:

```
type NetConf struct {
    types.NetConf
    BrName       string `json:"bridge"` //网桥名
    IsGW         bool   `json:"isGateway"`  //是否将网桥配置为网关
    IsDefaultGW  bool   `json:"isDefaultGateway"` //
    ForceAddress bool   `json:"forceAddress"`//如果网桥已存在且已配置了其他IP，通过此参数决定是否将其他ip除去
    IPMasq       bool   `json:"ipMasq"`//如果true，配置私有网段到外部网段的masquerade规则
```

手滑把库给删了，跑路前应该做的事。。。

我们关注其中的一部分字段，结合代码可以大致整理出bridge组件的工作内容。首先是ADD命令：

- 执行ADD命令时，brdige组件创建一个指定名字的网桥，如果网桥已经存在，就使用已有的网桥；
- 创建vethpair，将node端的veth设备连接到网桥上；
- 从ipam获取一个给容器使用的ip数据，并根据返回的数据计算出容器对应的网关；
- 进入容器网络名字空间，修改容器中网卡名和网卡ip，以及配置路由，并进行arp广播（注意我们只为vethpair的容器端配置ip，node端是没有ip的）；
- 如果IsGW=true，将网桥配置为网关，具体方法是：将第三步计算得到的网关IP配置到网桥上，同时根据需要将网桥上其他ip删除。最后开启网桥的ip_forward内核参数；
- 如果IPMasq=true，使用iptables增加容器私有网网段到外部网段的masquerade规则，这样容器内部访问外部网络时会进行snat，在很多情况下配置了这条路由后容器内部才能访问外网。（这里代码中会做exist检查，防止生成重复的iptables规则）；
- 配置结束，整理当前网桥的信息，并返回给调用者。

其次是DEL命令：

- 根据命令执行的参数，确认要删除的容器ip，调用ipam的del命令，将IP还回IP pool;
- 进入容器的网络名字空间，根据容器IP将对应的网卡删除；
- 如果IPMasq=true，在node上删除创建网络时配置的几条iptables规则。

### ptp

ptp其实是bridge的简化版。但是它做的网络配置其实看上去倒是更复杂了点。并且有一些配置在自测过程中发现并没有太大用处。它只创建vethpair，但是会同时给容器端和node端都配置一个ip。容器端配置的是容器IP，node端配置的是容器IP的网关（/32），同时，容器里做了一些特殊配置的路由，以满足让容器发出的arp请求能被vethpair的node端响应。实现内外的二层连通。

ptp的网络配置步骤如下：

- 从ipam获取IP，根据ip类型（ipv4或ipv6）配置响应的内核ip_forward参数；
- 创建一对vethpair；一端放到容器中；
- 进入容器的网络namespace，配置容器端的网卡，修改网卡名，配置IP，并配置一些路由。假如容器ip是10.18.192.37/20，所属网段是10.18.192.0/20，网关是10.18.192.1，我们这里将进行这样的配置：
    - 配置IP后，内核会自动生成一条路由，形如：`10.18.192.0/20 dev eth0 scope link`，我们将它删掉：`ip r d ****`

- 退出到容器外，将vethpair的node端配置一个IP（ip为容器ip的网关，mask=32）；
- 配置外部的路由：访问容器ip的请求都路由到vethpair的node端设备去。
- 如果IPMasq=true，配置iptables
- 获取完整的网卡信息（vethpair的两端），返回给调用者。

与bridge不同主要的不同是：ptp不使用网桥，而是直接使用vethpair+路由配置，这个地方其实有很多其他的路由配置可以选择，一样可以实现网络的连通性，ptp配置的方式只是其中之一。万变不离其宗的是：
**只要容器内网卡发出的arp请求，能被node回复或被node转发并由更上层的设备回复，形成一个二层网络，容器里的数据报文就能被发往node上；然后通过node上的路由，进行三层转发，将数据报文发到正确的地方，就可以实现网络的互联。**
bridge和ptp其实是用了不同方式实现了这个原则中的“二层网络”：

- bridge组件给网桥配置了网关的IP，并给容器配置了到网关的路由。实现二层网络
- ptp组件给vethpair的对端配置了网关的IP，并给容器配置了单独到网关IP的路由，实现二层网络

ptp模式的路由还存在一个问题：没有配置default路由，因此容器不能访问外部网络，要实现也很简单，以上面的例子，在容器里增加一条路由： `default via 10.18.192.1 dev eth0`

### host-device

相比前面两种cni main组件，host-device显得十分简单因为他就只会做两件事情：

- 收到ADD命令时，host-device根据命令参数，将网卡移入到指定的网络namespace（即容器中）。
- 收到DEL命令时，host-device根据命令参数，将网卡从指定的网络namespace移出到root namespace。

细心的你肯定会注意到，在bridge和ptp组件中，就已经有“将vethpair的一端移入到容器的网络namespace”的操作。那这个host-device不是多此一举吗?
并不是。host-device组件有其特定的使用场景。假设集群中的每个node上有多个网卡，其中一个网卡配置了node的IP。而其他网卡都是属于一个网络的，可以用来做容器的网络，我们只需要使用host-device，将其他网卡中的某一个丢到容器里面就行。
host-device模式的使用场景并不多。它的好处是：bridge、ptp等方案中，node上所有容器的网络报文都是通过node上的一块网卡出入的，host-device方案中每个容器独占一个网卡，网络流量不会经过node的网络协议栈，隔离性更强。缺点是：在node上配置数十个网卡，可能并不好管理；另外由于不经过node上的协议栈，所以kube-proxy直接废掉。k8s集群内的负载均衡只能另寻他法了。

### macvlan

后就能互相访问。
macvlan省去了linux bridge，但是配置macvlan后，容器不能访问parent接口的IP。

#### ipvlan

ipvlan与macvlan有点类似，但对于内核要求更高(3.19),ipvlan也会从一个网络接口创建出多个虚拟网络接口，但他们的mac地址是一样的， 只是IP不一样。通过路由可以实现不同虚拟网络接口之间的互联。
使用ipvlan也不需要linux bridge，但容器一样不能访问parent接口的IP。 关于ipvlan的内容可以参考这篇文章
关于macvlan和ipvlan，还可以参考这篇文章

### meta 类CNI

meta组件通常进行一些额外的网络配置（tuning），或者二次调用（flannel）。

#### tuning

用于进行内核网络参数的配置。并将调用者的数据和配置后的内核参数返回给调用者。
有时候我们需要配置一些虚拟网络接口的内核参数，比如：网易云在早期经典网络方案中曾修改vethpair的proxy_arp参数（后面会介绍）。可以通过这个组件进行配置。 另外一些可能会改动的网络参数比如：

- accept_redirects
- send_redirects
- proxy_delay
- accept_local
- arp_filter

可以在这里查看可配置的网络参数和释义。

#### portmap

用于在node上配置iptables规则，进行SNAT,DNAT和端口转发。
portmap组件通常在main组件执行完毕后执行，因为它的执行参数仰赖之前的组件提供

#### flannel

cni plugins中的flannel是开源网络方案flannel的“调用器”。这也是flannel网络方案适配CNI架构的一个产物。为了便于区分，以下我们称cni plugins中的flannel 为 `flanenl cni` 。

如果对flannel网络方案有一定的了解，会知道他在做网络接口配置时，实质干的事情和bridge组件差不多。只不过flannel网络中的bridge会跟flannel0网卡互联，而flannel0网卡上的数据会被封包（udp、vxlan下）或直接转发（host-gw）。

而 `flannel cni` 做的事情就是：

- 执行ADD命令时， `flannel cni` 会从本地文件中读取到flanneld的配置。然后根据命令的参数和文件的配置，生成一个新的cni配置文件（保存在本地，文件名包含容器id以作区分）。新的cni配置文件中会使用其他cni组件，并注入相关的配置信息。之后， `flannel cni` 根据这个新的cni配置文件执行ADD命令。
- 执行DEL命令时， `flannel cni` 从本地根据容器id找到之前创建的cni配置文件，根据该配置文件执行DEL命令。

**也就是说 `flannel cni` 此处是一个flannel网络模型的委托者，falnnel网络模型委托它去调用其他cni组件，进行网络配置。通常调用的是bridge和host-local。**

# 几种常见的网络方案

上述所有的cni组件，能完成的事情就是建立容器到虚拟机上的网络。而要实现跨虚拟机的容器之间的网络，有几种可能的办法：

- 容器的IP就是二层网络里分配的IP，这样容器相当于二层网络里的节点，那么就可以天然互访；
- 容器的IP与node的IP不属于同一个网段，node上配置个到各个网段的路由（指向对应容器网段所部属的node IP），通过路由实现互访[flannel host-gw, calico bgp均是通过此方案实现]；
- 容器的IP与node的IP不属于同一个网段，node上有服务对容器发出的包进行封装，对发给容器的包进行解封。封装后的包通过node所在的网络进行传输。解封后的包通过网桥或路由直接发给容器，即overlay网络。[flannel udp/vxlan，calico ipip，openshift-sdn均通过此方案实现]

## kubenet

了解常用的网络方案前，我们先了解一下kubenet，kubenet其实是k8s代码中内置的一个cni组件。如果我们要使用kubenet，就得在kubelet的启动参数中指定 `networkPlugin` 值为 `kubenet` 而不是 `cni` 。

如果你阅读了kubernetes的源码，你就可以在一个名为kubenet_linux.go的文件中看到kubenet做了什么事情：

- 身为一种networkPlugin，kubenet自然要实现networkPlugin的一些接口。比如SetUpPod，TearDownPod，GetPodNetworkStatus等等，kubelet通过这些接口进行容器网络的创建、解除、查询。
- 身为一个代码中内置的cni，kubenet要主动生成一个cni配置文件（字节流数据），自己按照cni的规矩去读取配置文件，做类似ADD/DEL指令的工作。实现网络的创建、解除。

设计上其实挺蠢萌的。实际上是为了省事。我们可以看下自生成的配置文件：

```
  "type": "bridge",
  "bridge": "%s", //通常这里默认是“cbr0”
  "mtu": %d,    //kubelet的启动参数中可以配置，默认使用机器上的最小mtu
  "addIf": "%s", //配置到容器中的网卡名字
  "isGateway": true,
  "ipMasq": false,
  "hairpinMode": %t,
  "ipam": {
    "type": "host-local",
    "subnet": "%s", //node上容器ip所属子网，通常是kubelet的pod-cidr参数指定
    "gateway": "%s", //通过subnet可以确定gateway
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
```

配置文件中明确了要使用的其他cni组件：bridge、host-local（这里代码中还会调用lo组件，通常lo组件会被k8s代码直接调用，所以不需要写到cni配置文件中）。之后的事情就是执行二进制而已。

为什么我们要学习kubenet？因为kubenet可以让用户以最简单的成本（配置networkPlugin和pod-cidr两个启动kubelet启动参数），配置出一个简单的、虚拟机本地的容器网络。结合上面提到的几种“跨虚拟机的容器之间的网络方案”，就是一个完整的k8s集群网络方案了。

*通常kubenet不适合用于overlay网络方案，因为overlay网络方案定制化要求会比较高。*

*许多企业使用vpc网络时，使用自定义路由实现不同pod-cidr之间的路由，他们的网络方案里就会用到kubenet，比如azure AKS（基础网络）。*

## flannel

关于flannel，上面的文章也提到了一下。网上flannel的文章也是一搜一大把。这里简单介绍下flannel对k8s的支持，以及通用的几个flannel backend（后端网络配置方案）。

### flannel for kubernetes

flannel在对kubernets进行支持时，flanneld启动参数中会增加 `--kube-subnet-mgr` 参数，flanneld会初始化一个kubernetes client，获取本地node的pod-cidr，这个pod-cidr将会作为flannel为node本地容器规划的ip网段。记录到/run/flannel/subnet.env。（flannel_cni组件会读取这个文件并写入到net-conf.json中，供cni使用）。

发出，所以flanneld可以捕获到容器发出的报文，进行封装。udp方案下会给报文包装一个udp的头部，vxlan下会给报文包装一个vxlan协议的头部（配置了相同VNI的node，就能进行互联）。 目前flannel社区还提供了更多实验性的封装协议选择，比如ipip，但仍旧将vxlan作为默认的backend。

### host-gw

flannel的三层路由方案。每个node节点上都会记录其他节点容器ip段的路由，通过路由，node A上的容器发给node B上的容器的数据，就能在node A上进行转发。

### alloc

类似kubenet，只分配子网，不做其他任何事情。

### 支持云厂商的vpc

flannel支持了aliVPC、gce、aws等云厂商的vpc网络。原理都是一样的，就是当flanneld在某云厂商的机器上运行时，根据机器自身的vpc网络IP，和flanneld分配在该机器上的subnet，调用云厂商的api创建对应的自定义路由。

## calico

calico是基于BGP路由实现的容器集群网络方案，对于使用者来说，基础的calico使用体验可能和flannel host-gw是基本一样的：node节点上做好对容器arp的响应。然后通过node上的路由将容器发出的包转发到对端容器所在node的IP。对端节点上再将包转发给对端容器。
ipip模式则如同flannel ipip模式。对报文封装一个ipip头部，头部中使用node ip。发送到对端容器所在node的IP，对端的网络组件再解包，并转发给容器。
不同之处在于flannel方案下路由都是通过代码逻辑进行配置。而calico则在每个节点建立bgp peer，bgp peer彼此之间会进行路由的共享和学习，所以自动生成并维护了路由。

# 一些大厂的容器服务网络方案

### 阿里云

通过上文flannel aliVPC模式可见一斑。阿里云中kubernetes服务里，k8s集群通常使用自定义路由的方案+flannel_cni组件，这个方案易于部署和管理，同时将容器IP和nodeIP区分，用户可以自定义集群网络范围。
*（比较奇怪的是这里flanenl的backend配置成alloc而非aliVPC，在集群中另外部署了一个controller进行自定义路由的配置）*

云执行。

### azure

azure最近开放了kubernetes服务AKS，AKS支持两种网络方案:基础和高级。
基础网络方案与阿里云的自定义路由方案如出一辙。基础网络中k8s集群使用的网络组件是kubenet,简单的做了网络划分和本地的网络接口配置，自定义路由由其vpc实现。
高级网络方案中，node上的网络接口会创建并绑定多个（默认三十个）fixedIP，主FixedIP作为node IP，其余fixedIP则用于容器IP。 通过azure SDN的支持，不同node之间的容器网络变成一个大二层，他们可以直接互联。高级网络方案中，k8s集群使用azure开源的cni组件：azure-container-networking。这个cni组件包括了ipam和main两部分
azure cni的ipam负责将本地网络接口上绑定着的空闲的fixedIP配置给容器使用。一旦空闲的fixedIP耗尽，除非手动给网卡创建新的fixedIP，否则容器无法创建成功。
azure cni的main组件在node上创建了一个bridge，将node的网卡连接到网桥上，并将node网卡IP设置到网桥上，容器网卡均由vethpair实现，vethpair的node端也是连在网桥上。由此构成node的网络：网桥上的IP作为容器网络的网关，容器网络通过网桥与其他节点形成一个大二层的网络。





< 上一篇 webpack常用配置详解下篇 下一篇 springboot源码解析-run() >

漫画：深入浅出 ES 模块（上篇）

Maven Document翻译——Maven入门指南（下篇）

Android通知栏介绍与适配总结（上篇）

当我们在谈论multidex65535时，我们在谈论什么

GFS论文（2003）小结

值得细读！如何系统有效地提升Android代码的安全性?

Promise之你看得懂的Promise

微服务架构：引领数字化转型的基石