# Docker扁平化网络设计与实现

开发背景

众所周知，Docker容器跨主机互访一直是一个问题，Docker官方为了避免网络上带来的诸多麻烦，故将跨主机网络开了比较大的口子，而由用户自己去实现。

目前Docker跨主机的网络实现方案也有很多种, 主要包括端口映射，ovs, fannel等。但是这些方案都无法满足我们的需求，端口映射服务内的内网IP会映射成外网的IP，这样会给开发带来困惑，因为他们往往在跨网络交互时是不需要内网IP的，而ovs与fannel则是在基础网络协议上又包装了一层自定义协议，这样当网络流量大时，却又无端的增加了网络负载，最后我们采取了自主研发扁平化网络插件，也就是说让所有的容器统统在大二层上互通。

网络模式

目前， 基于Docker的网络模式有很多种，接下来就简单的对他们进行一下介绍。

Docker 原生四种网络模式：

1. Bridge模式

该模式为Docker的默认网络模式，Docker daemon 会在宿主机上建立一个默认的网桥docker0, 相信大家对docker0非常熟悉，但是在跨容器通信当中它却没有派上用场，因为默认的docker0的地址都是内网地址，而且启动后容器虽然也桥接在docker0上，但是容器的默认网关却依然无法设置，这就是docker原生默认网络的一个弊端。当然了，他却实现了在当先宿主机的网络隔离，拥有自己的namespace, 网卡和IP，具体的桥接原理我将在后面继续说明。

1. Host 模式

该模式其实就是和当前宿主机共享网络空间，而Docker本身并没有进行网络隔离，说的通俗点，也就是说容器其实都是和宿主机拥有相同的IP, 而如何具体区分各个容器的呢？那就是通过端口映射，在启动Docker容器的时候来指定-p参数来进行设置端口映射。虽然这种方式在某种程度上也可以达到跨宿主机容器访问的目的，但是，却丧失了Docker网络隔离的意义，而且端口映射同样给微服务迁移带来一些麻烦，无法像非虚拟环境那样的平滑迁移，而是要考虑到很多端口转换的问题。

1. Container 模式

顾名思义，此模式会共享另一个容器的网络命名空间，但是会限制在一台宿主机上，依然无法实现容器间跨主机通信的功能。

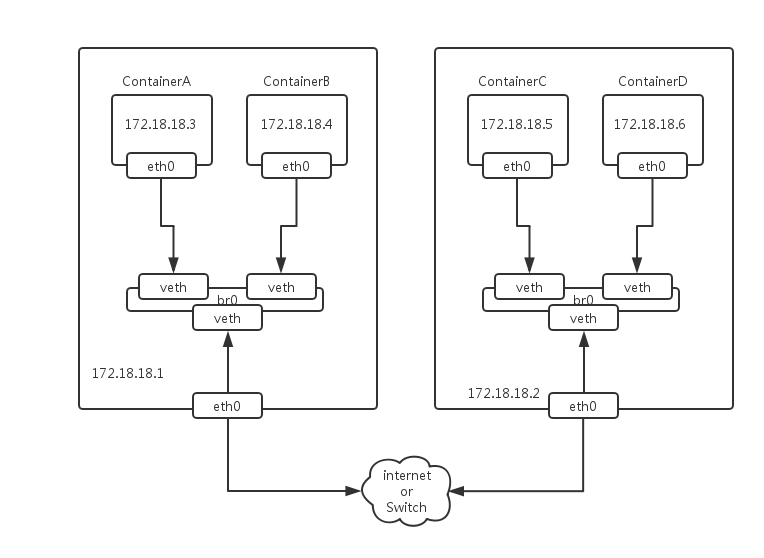
1. None 模式

该模式是容器拥有自己的网络命名空间，自己的网路栈，自己的网卡，不和外界有任何瓜葛，容器网络完全独立，换句话说就是容器不需要网络功能，这种模式适用于容器包含写数据到磁盘卷的一些任务。这种模式依然无法实现我们的跨宿主机容器网络通信的功能。

Docker Overlay网络模式：

目前Overlay网络模式主要是由隧道和路由两种方式实现，一种是对基础网络协议进行封包，另一种是配置更复杂的路由配置实现容器间跨主机的网络通信。其实，以上两种或多或少的都会给我们网络的实现带来了复杂性以及性能上的损耗，因为当我们拥有庞大的业务集群以后，这些复杂度和性能损耗都是不能忽视的。

插件原理



1. 创建Dokcer自定义网络

docker network create

--opt=com.docker.network.bridge.enable\_icc=true

--opt=com.docker.network.bridge.enable\_ip\_masquerade=false

--opt=com.docker.network.bridge.host\_binding\_ipv4=0.0.0.0

--opt=com.docker.network.bridge.name=br0

--opt=com.docker.network.driver.mtu=1500

--ipam-driver=talkingdata

--subnet=容器IP的子网范围， 例：172.18.0.0/17

--gateway=br0网桥使用的IP,也就是宿主机的地址， 例：172.18.0.5

--aux-address=DefaultGatewayIPv4=容器使用的网关地址

mynet

我们首先需要创建一个br0自定义网桥，这个网桥并不是通过系统命令手动建立的原始Linux网桥，而是通过Docker的cerate network命令来建立的自定义网桥，这样避免了一个很重要的问题就是我们可以通过设置DefaultGatewayIPv4参数来设置容器的默认路由，这个解决了原始Linux自建网桥不能解决的问题. 用Docker创建网络时我们可以通过设置subnet参数来设置子网IP范围，默认我们可以把整个网段给这个子网，后面可以用ipam driver（地址管理插件）来进行控制。还有一个参数gateway是用来设置br0自定义网桥地址的，其实也就是你这台宿主机的地址啦。

1. IPAM

这个驱动是专门管理Docker 容器IP的, Docker 每次启停与删除容器都会调用这个驱动提供的IP管理接口，然后IP接口会对存储IP地址的Etcd有一个增删改查的操作。此插件运行时会起一个Unix Socket, 然后会在docker/run/plugins 目录下生成一个.sock文件，Docker daemon之后会和这个sock 文件进行沟通去调用我们之前实现好的几个接口进行IP管理，以此来达到IP管理的目的，防止IP冲突。

1. 桥接

通过Docker命令去创建一个自定义的网络起名为“mynet”，同时会产生一个网桥br0，之后通过更改网络配置文件（在/etc/sysconfig/network-scripts/下ifcfg-br0、ifcfg-默认网络接口名）将默认网络接口桥接到br0上，重启网络后，桥接网络就会生效。Docker默认在每次启动容器时都会将容器内的默认网卡桥接到br0上，而且宿主机的物理网卡也同样桥接到了br0上了。其实桥接的原理就好像是一台交换机，Docker 容器和宿主机物理网络接口都是服务器，通过veth pair这个网络设备像一根网线插到交换机上。至此，所有的容器网络已经在同一个网络上可以通信了，每一个Docker容器就好比是一台独立的虚拟机，拥有和宿主机同一网段的IP，可以实现跨主机访问了。

1. ETCD

我们可以设置1，3，5，7个节点为Etcd集群去集中管理Dcoker集群的IP，而且我们也将宿主机的地址进行了统一的管理，这样做同样也是为了避免IP的使用冲突导致线上资源不可用。我们会通过自己开发的工具进行IP初始化，也就是说会传进来一个IP范围，然后工具会将所有的IP存进Etcd中，每一个网络ID就是一个Etcd目录，目录下就会分成已分配与未分配的IP地址池。Etcd本身会提供Go语言的API来访问Etcd, 目前Etcd还是相当稳定的，没有出现过什么问题。

最后

我们的Docker集群是采用Swarm进行管理的，Swarm相对K8S来说要简单的很多，但是在编排功能上也会有所欠缺，不过目前等新版本的dokcer 1.12发布以后，Swarm集成到Docker内部，而且增加了许多的功能后，我想这个问题就会迎刃而解。管理Swarm的是采用第三方的管理图形界面软件shipyard，这款软件本身并不会兼容自定义网络，而且在设置每个容器使用的Cpu核数时又会有BUG，我们对此开源软件进行了二次开发，解决了这些问题。现如今我们已经成功的在上面运行了yarn集群，网络性能也是没有什么问题的。

参考

<https://docs.docker.com/engine/extend/plugins>  
<https://github.com/docker/libnetwork/blob/master/docs/ipam.md>  
<https://github.com/docker/go-plugins-helpers>