# K8s

## k8s基本架构

master节点：kube-apiserver、controller manager、scheduler。

**kube-apiserver**：监听8080或（HTTPS）6443端口，提供了k8s各种资源的增删改查等功能。是集群与etcd交互的唯一入口。

**controller manager**：不是一个组件，而是由多个控制器组成：

1. replication controller
2. node controller

node controller会管理node的信息，定期将收到的节点信息和本地保存的nodeStatusMap中保存的节点信息做比较。如果节点状态为非“就绪”，那么将会删除该节点信息。

1. resourceQuota controller

资源配额管理，k8s提供了三个层面的资源配额管理：

1. 容器级别，对于CPU和内存进行限制。
2. Pod级别，对于Pod内所有容器可用的资源进行限制。
3. namespaces级别，包括对Pod数量，RC数量，Service数量等进行限制。
4. namespace controller
5. service controller
6. endpoint controller

endpoint就是service下podip和targetPort的集合。

**sheduler**：为controller manager创建的pod分配到具体的node节点上。

主要分两步：1）预选过程，从所有node中选择符合条件的node节点。2）优选过程，计算每个候选node的积分，积分高的被选中。

默认加载的预选策略：

1. PodFitsPorts

判断Pod的端口在node上是否已占用，已占用返回false。

1. PodFitsResources
2. NoDiskConflict
3. MatchNodeSelector
4. PodFitsHost

判断Pod的sepc.nodeName所指定的名称与node名称是否相同，不同返回false。

可用的预选策略：

1. NoDiskConflict
2. PodSelectorMatches

判断node是否拥有Pod标签选择器所指定的标签。如果Pod没有标签选择器，那返回true。否则就判断Pod标签和node标签是否匹配，如不匹配则返回false。

1. CheckNodeLabelPresence
2. CheckServiceAffinity

优选策略：

1. LeastRequestedPrority

从备选节点中选择资源消耗最少的节点（CPU和内存）。

1. CaclulateNodeLabelPriority
2. BalancedResourceAllocation

从备选节点中选择各项资源使用最平均的节点。

node节点：kubelet、kube-proxy、docker。

存储：etcd。

## 服务发现

1、环境变量：podA里的应用需要访问serviceB里的服务，通过环境变量将serviceB的ip和port注入podA。这要求serviceB必须在podA之前被创建。

2、dns：kubeadm自带coredns。可以让podA通过serviceB的名称来访问serviceB，[service-name].[namespace].svc.cluster.local就是serviceB的域名。

## 服务发布

三种方式：

1、NodePort：在宿主机上创建一个端口，可以通过宿主机ip+NodePort的方式访问集群内部服务。

NodePort的范围是通过--service-node-port-range配置项来指定的，默认值为（30000-32767）。

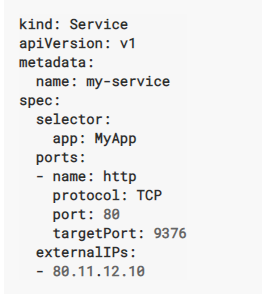
还可以通过kube-proxy的--nodeport-addresses选项来决定可以被用作NodePort绑定的ip，比如127.0.0.0/8这种。这个选项默认是[]，相当于NodePort会被绑定在0.0.0.0的ip上。

2、LoadBalancer：通过云厂商提供的LoadBalancer来访问集群内部服务。

3、ExternalName：通过服务名称来访问集群内部服务。

通过在yaml文件中配置externalName来给指定service的名称。

Externel IPs：如果有外部ip能够联通一个或者多个node节点的话，就可以使用这类ip作为service访问的入口。



比如这种就可以通过80.11.12.10:80来访问service。

## Ingress

Ingress提供了集群外部通过域名访问内部服务的能力，还提供了负载均衡。Ingress只针对HTTP和HTTPS协议有用，对于其他协议还是需要用到NodePort或者LoadBalancer。

使用Ingress之前需要安装Ingress controller。

Ingress本身也需要创建一个service来提供给外部集群来访问。根据http请求的host字段，找到对应的servicename和端口，然后将请求转发到对应的service里。

Ingress不要求应用容器一定要是按照deployment方式部署的，单个pod也是可行的。

Ingress搭建完成之后外部需要通过域名来访问还需要具备将域名转换成master节点ip的能力，单机测试可以通过修改host，网络的话就是通过dns。

Ingress需要跟service在相同的命名空间，不同的命名空间，ingress无法实现外部访问内部服务。

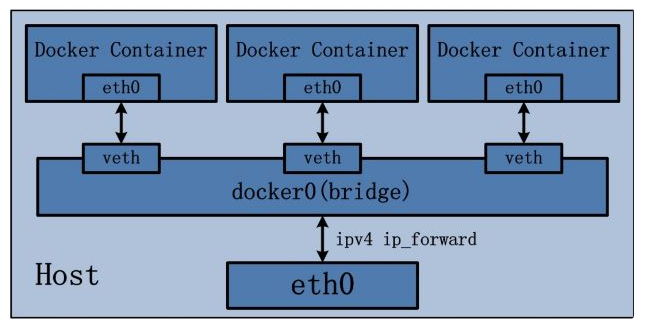
## Service的负载均衡

service本身是一个逻辑上的概念，是依赖各个node上的kube-proxy来将请求转发到service对应的各个pod上。请求转发到pod采用的是轮询的方式。可以通过service配置来实现请求定向转发到某一个pod上。

## 网络模型

Docker网络模型

Docker默认是采用brige网络模式，同时还提供了host、none、overlay等其他网络模式。

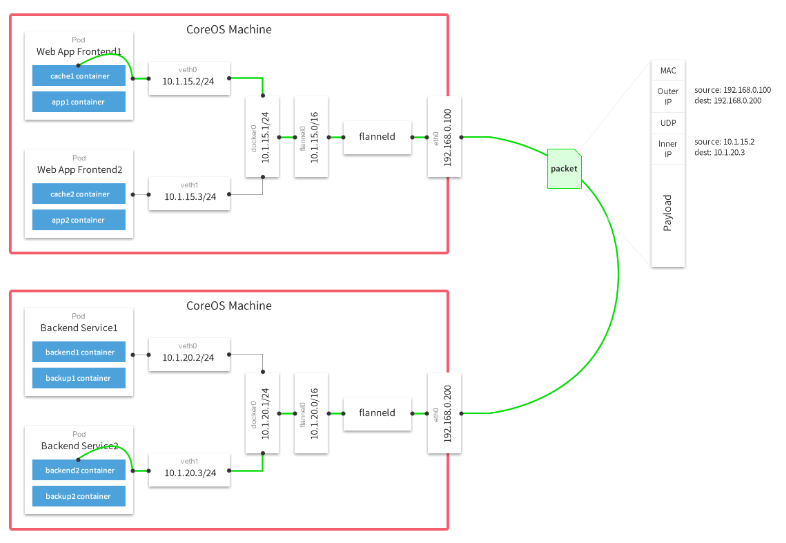


docker0网桥是在安装docker的时候就会生成。veth pair设备对是在启动容器的时候生成的。

一端在容器内部作为eth0网口，另一端在docker0上。作用就是将容器内发送到eth0网口的数据传递到docker0网桥上，再由网桥从宿主机的eth0接口发送出去。

docker0本身是无法被外部访问的，如果docker的容器需要被外部访问，就要绑定一个宿主机端口。

Flannel网络模型



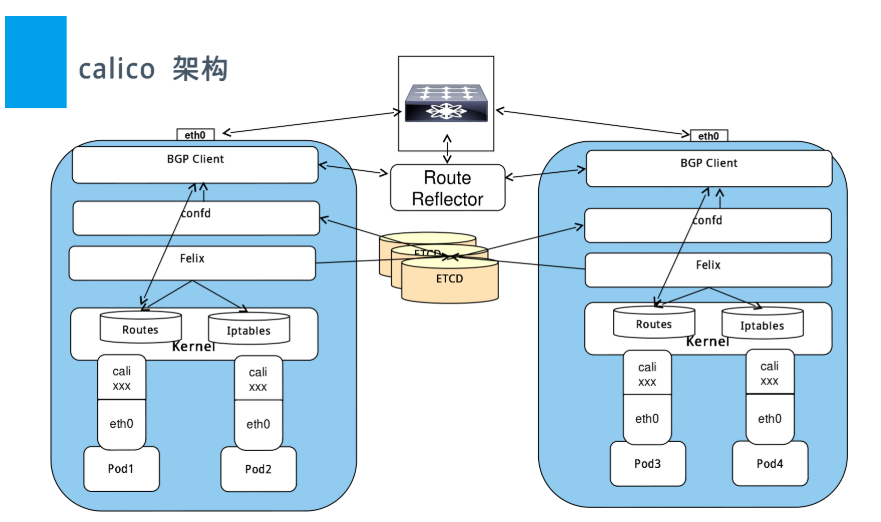
flannel网络插件为集群内每个节点上的docker分配不冲突的IP地址。同时为每个节点及docker创建一个overlay网络，实现数据包的互相传递。

flannel在master和node节点上启动了一个flanneld的服务。简单的说docker0网卡的数据原本是直接发给宿主机的eth0，现在发给flannel1网口，由flanneld服务对数据进行二次封装。flanneld会给数据封装一个二层头部，携带了源容器flannel1网卡的MAC地址和目的容器flannel1网卡的MAC地址。之后将数据交给宿主机eth0网口。目标机器收到数据之后由flanneld的服务解封装置后将真实数据交给对应容器。

同一台宿主机上的Pod之间通过docker0网桥来通信，Pod的ip和docker0网桥在同一个网段的。

不同宿主机上的Pod之间通过flannel通信。

calico网络模型



Felix：负责路由配置和ACLS规则配置的下发，运行在每个node节点上。

BGP Client：负责把Felix写入kernel的信息分发到当前的calico网络中，确保节点间通信有效性。

BGP Route Reflector：大规模部署使用的时候，通过一个或者多个BGP Route Reflector完成集中式路由部署和分发。

calico网络有两种方式：

1. IPIP

将各个Node的路由之间做一个虚拟的通道tuln0，在每个Node上都会创建这个虚拟的接口。

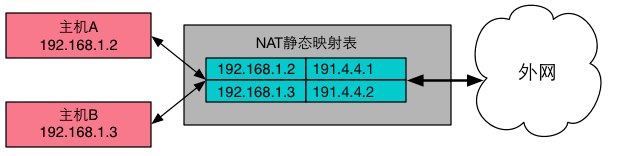
1. BGP

直接使用物理机作为虚拟路由器，不再创建额外的tunnel。

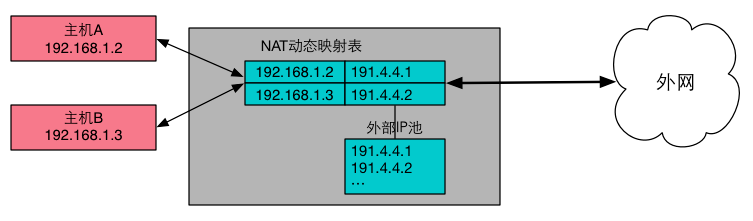
## NAT

NAT有三种模式：

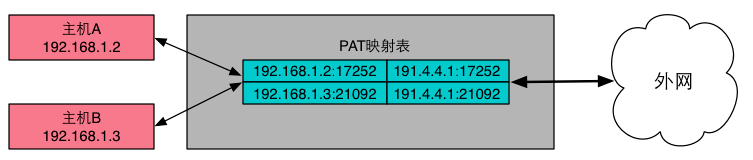
1. 静态NAT：将内部网络的私有IP地址转换为公有IP地址。私有地址和公有地址的对应关系是固定的。



1. 动态NAT：有内网IP需要转换为公网IP的时候，从公网IP池里取出一个IP，和内网IP形成对应关系。当通信结束之后，将公网IP返回IP池。



1. NAPT：端口层面的地址转换，对外使用统一的公网IP。根据不同的端口来做地址转换。



## LVS

三种工作模式：

1. 基于NAT的负载均衡：

用户实际访问的是LVS在公网的ip，也就是vip，LVS会根据NAT机制将其转换为对应的私网ip。这种模式的缺点在于，所有的流量都要经过LVS转发，LVS很容易成为性能瓶颈。

1. 基于TUN的负载均衡：

一般情况下，请求数据大小远小于响应数据的大小，TUN就是通过将请求数据和响应数据流量分离，来解决NAT模式碰到的问题。LVS仅处理请求数据，而响应数据由真实服务器直接返回给用户。这就要求真实服务器能够访问公网。

由于请求是经过LVS转发过的，那么对于真实服务器，要如何知道用户的真实ip和port？这里就用到IP隧道技术，在原始数据上添加新的ip包头，包括新的源ip+port，目的ip+port。

1. 基于DR的负载均衡：

DR模式和TUN模式相似之处在于也是有LVS转发请求，而真实服务器将响应直接发给用户。区别在于要求LVS和真实服务器在一个局域网内，LVS的公网vip在真实服务器上共享。而且真实服务器必须配置在Non-ARP设备上。LVS和真实服务器之间做二层交换。

负载均衡调度算法：

1. 轮询、加权伦旭
2. 最小连接、加权最小连接
3. 局部最小连接

根据目标ip找到最近使用的服务器，如果该服务器没有超载，则将请求转发到该服务器。如果该服务器不存在或者已超载，则根据最小连接原则选出一个可用的服务器。

1. 带复制的局部最小连接

与局部最小连接的区别就是，维护的是一个目标ip到一组服务器的映射关系。每次按照最小连接原则从服务器组中选择一台服务器。

1. 目标地址哈希
2. 原地址哈希
3. 最短期望延迟
4. 最少队列

## PV和PVC

k8s中的volumn可以解决共享存储的问题， 但是只是使用pod中的volumn会带来一个问题，就是当pod数量非常多时候，会引起共享存储难以管理的问题。因此k8s给出了PV和PVC的解决方案。

与pod的volumn不同，PV和PVC的生命周期是独立于pod之外的。PV是外部存储的一块空间，可以是宿主机的存储空间，也可以是网络存储。PVC则是对PV的申请，用户创建PVC的时候指定需要的存储空间大小，访问模式（只读、只写、读写），k8s会为PVC匹配满足其条件的PV。当PVC使用完成之后，可以删除PVC，这样PVC绑定的PV会被k8s系统回收，状态重新变为available，可以重新分配给其他PVC。

k8s也支持动态创建PV，即storageclass。如果当前没有满足PVC条件的PV，k8s会动态创建新的PV。PVC只需要指定storageclass名称，需要申请的容量大小和访问模式即可。

## Headless Service

有些情况下用户不希望使用service提供的负载均衡，就可以通过Headless Service来解决，即Service不会分配cluster IP。访问这个service会得到service所包含的全部Pod列表。

## StatefulSet

k8s的RC/Deployment创建和管理pod一般面向无状态的服务，但像mysql之类的数据库集群都是有状态的。StatefulSet有以下几个特点：

1. 里面的Pod都有稳定的、唯一的网络标识。而RC/Deployment创建的Pod名称和ip都是随机的。比如StatefulSet的名称叫A，那么创建的pod名称就是A-0、A-1依次类推。
2. StatefulSet控制Pod副本的启停顺序是按顺序的。操作第n个pod的时候，前面n-1个pod一定是准备好的状态。
3. StatefulSet里的Pod采用稳定的持久化存储卷。

StatefulSet还需要与Headless Service配合使用，StatefulSet需要声明属于哪个Headless Service。同时StatefulSet还为其控制的每个Pod都创建了一个DNS域名实例$(pod\_name).$(headless service name)。