**加我微信可进群学习交流：**

**微信号：**

luckylucky421302

也可通过扫描下面二维码添加



**课程更新的知识点会通过微信公众号免费分享给大家，可以关注我的公众号**

****

--pod-network-cidr=10.244.0.0/16 （pod网段）

--service-cidr=10.96.0.0/12 （service网段）

**1、Service概念：**

**Kubernetes中的** [**Pod**](https://kubernetes.io/docs/user-guide/pods)**是有生命周期的**，它们可以被创建，也可以被销毁，然而一旦被销毁pod生命就永远结束，这个pod就不存在了，通过[ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)能够动态地创建和销毁Pod（例如，需要进行扩缩容，或者执行[滚动升级](https://kubernetes.io/docs/user-guide/kubectl/v1.7/" \l "rolling-update)），每个Pod都会获取它自己的 IP 地址，这些IP地址并不是一直处于稳定的状态，可能随时改变。 这会导致一个问题：在 Kubernetes 集群中，如果一组Pod（称为 backend）为其它Pod （称为 frontend）提供服务，那么那些 frontend 该如何发现这些backend？

**关于Service**

Kubernetes Service定义了这样一种抽象：逻辑上的一组Pod，一种可以访问它们的策略-通常称为微服务。 这一组Pod能够被Service访问到，通常是通过[Label Selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/" \l "label-selectors)实现的。举个例子，考虑一个图片处理程序 backend，它运行了3个副本。这些副本是可互换的 —— frontend 不需要关心它们调用了哪个 backend 副本。 然而组成这一组 backend 程序的 Pod 实际上可能会发生变化，frontend 客户端不应该也没必要知道，而且也不需要跟踪这一组 backend 的状态。 Service 定义的抽象能够解耦这种关联。

例如想要用nginx反向代理tomcat，那么tomcat如果是通过pod部署的，pod的ip可能会随时变化，那么我们就需要在所有这些部署tomcat的pod前面加上一个固定接入层service，我们nginx反向代理只需要写service地址，就会代理到后端的pod，那么pod就算ip怎么变化，通过service都可以找到，对 Kubernetes 集群中的应用，Kubernetes 提供了简单的Endpoints API，只要Service中的一组Pod发生变更，应用程序就会被更新。 对非 Kubernetes 集群中的应用，Kubernetes 提供了基于 VIP 的网桥的方式访问Service，再由Service重定向到backend Pod。

**简单归纳一下**

service是一个固定接入层，客户端可以通过访问service来访问到service关联的后端pod，这个service工作依赖于在kubernetes集群之上部署的一个附件，就是kubernetes的dns服务（不同kubernetes版本的dns默认使用的也是不一样的，1.11之前的版本使用的是kubeDNs，较新的版本使用的是coredns），service的名称解析是依赖于dns附件的，因此在部署完k8s之后需要在部署dns附件，kubernetes要想给客户端提供网络功能，需要依赖第三方的网络插件（flannel，calico等）。

在kubernetes集群中有三类ip地址：

node network（节点网络），pod network（pod 网络），这两种网络地址是我们实实在在配置的，其中节点网络地址是配置在节点接口之上，而pod网络地址是配置在pod资源之上的，因此这些地址都是配置在某些设备之上的，这些设备可能是硬件，也可能是软件模拟的，cluster network（集群地址，也成为service network），这个地址是虚拟的地址（virtual ip），没有配置在某个接口上，只是出现在service的规则当中。

每个K8s节点上都有一个工作的组件叫做kube-proxy，kube-proxy这个组件将始终监视着apiserver中有关service资源的变动信息，需要跟master之上的apiserver交互，随时连接到apiserver上获取任何一个与service资源相关的资源变动状态，这种是通过kubernetes中固有的一种请求方法watch（监视）来实现的，一旦有service资源的内容发生变动（如创建，删除），kube-proxy都会将它转化成当前节点之上的能够实现service资源调度，把我们请求调度到后端特定的pod资源之上的规则，这个规则可能是iptables，也可能是ipvs，取决于service的实现方式

**service实现方式**

**第一种：iptables**

客户端ip请求时直接请求service的ip，这个请求报文被本地内核空间中的service规则所截取，进而直接调度给相关的pod，这个方式是直接工作在内核空间，由iptables规则直接实现

**第二种：ipvs**

客户端请求到达内核空间之后直接由ipvs规则来调度到相关的pod资源

1.11之前的版本使用的是iptables

1.11+版本使用的是ipvs，ipvs如果没有被激活就会自动降级为iptables

如果某个服务背后的pod资源发生改变，比如service的标签选择器适用的pod又多了一个，这个pod使用的信息会立即反应在apiserver上，kube-proxy能监听到这个service的变化，将其立即转为service规则（如iptables规则）

**2. 通过Service访问pod**

**（1）创建pod**

**cat pod\_test.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-nginx

spec:

selector:

matchLabels:

run: my-nginx

replicas: 2

template:

metadata:

labels:

run: my-nginx

spec:

containers:

- name: my-nginx

image: nginx

ports:

- containerPort: 80

kubectl apply -f pod\_test.yaml

这使得可以从集群中任何一个节点来访问它。

**检查pod运行情况，可看到该 Pod 正在运行：**

kubectl get pods -l run=my-nginx -o wide

**检查 Pod 的 IP 地址：**

kubectl get pods -l run=my-nginx -o yaml | grep podIP

显示如下：

podIP: 10.244.3.40

cni.projectcalico.org/podIP: 10.244.3.42/32

podIP: 10.244.3.42

应该能够通过 ssh 登录到集群中的任何一个节点上，使用 curl 也能调通所有 IP 地址。 需要注意的是，容器不会使用该节点上的 80 端口，也不会使用任何特定的NAT规则去路由流量到Pod上。 这意味着可以在同一个节点上运行多个 Pod，使用相同的容器端口，并且可以从集群中任何其他的Pod或节点上使用IP的方式访问到它们。像 Docker一样，端口能够被发布到主机节点的接口上，但是出于网络模型的原因应该从根本上减少这种用法。

**（2）创建service**

我们所有Pod在一个扁平的、集群范围的地址空间中运行 Nginx 服务，可以直接连接到这些 Pod，但如果某个节点死掉了会发生什么呢？ Pod 会终止，Deployment 将创建新的 Pod，且使用不同的 IP。这正是 Service 要解决的问题。Kubernetes Service 从逻辑上定义了运行在集群中的一组 Pod，这些 Pod 提供了相同的功能。 当每个 Service 创建时，会被分配一个唯一的 IP 地址（也称为 clusterIP）。 这个 IP 地址与一个 Service 的生命周期绑定在一起，当 Service 存在的时候它也不会改变。 可以配置 Pod 使它与 Service 进行通信，Pod 知道与 Service 通信将被自动地负载均衡到该 Service 中的某些 Pod 上。

**创建一个service**

**cat service.yaml**

apiVersion: v1

kind: Service

metadata:

name: my-nginx

labels:

run: my-nginx

spec:

ports:

- port: 80

protocol: TCP

selector:

run: my-nginx

kubectl apply -f service.yaml

上述yaml将创建一个 Service，对应具有标签**run: my-nginx**的Pod，目标TCP端口 80，并且在一个抽象的Service端口（**targetPort**：容器接收流量的端口；**port**：抽象的 Service 端口，可以使任何其它 Pod访问该 Service 的端口）上暴露。 查看 [Service API 对象](https://kubernetes.io/docs/api-reference/v1.15/#service-v1-core)了解 Service 定义支持的字段列表。

**kubectl get svc my-nginx**

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

my-nginx ClusterIP 10.104.137.147 <none> 80/TCP 10m

正如前面所提到的，一个 Service 由一组 backend Pod 组成。这些 Pod 通过endpoints暴露出来。 Service Selector将持续评估，结果被 POST 到一个名称为my-nginx的 Endpoint对象上。 当 Pod 终止后，它会自动从 Endpoint 中移除，新的能够匹配上 Service Selector 的 Pod 将自动地被添加到 Endpoint 中。 检查该 Endpoint，注意到 IP 地址与在第一步创建的 Pod 是相同的。

**kubectl describe svc my-nginx**

Name: my-nginx

Namespace: default

Labels: run=my-nginx

Annotations: kubectl.kubernetes.io/last-applied-configuration: {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":{"run":"my-nginx"},"name":"my-nginx","namespace":"default"},"spe...

Selector: run=my-nginx

Type: ClusterIP

IP: 10.104.137.147

Port: <unset> 80/TCP

TargetPort: 80/TCP

Endpoints: 10.244.3.40:80,10.244.3.42:80,10.244.3.43:80

Session Affinity: None

Events: <none>

**kubectl get ep my-nginx**

NAME ENDPOINTS AGE

my-nginx 10.244.2.5:80,10.244.3.4:80 1m

现在，能够从集群中任意节点上使用 curl 命令请求 Nginx Service <CLUSTER-IP>:<PORT>

curl 10.104.137.147:80 通过访问service ip:port可以路由到后端的pod

service的type类型如果是ClusterIP，那么就只能被k8s集群中的节点或者pod所访问，不能再k8s集群之外的机器或者浏览器请求service地址，如果想要在k8s之外的主机或者浏览器访问到service，需要把ClusterIP类型变成nodePort，可按如下方法修改：

**注：**

把上面的ClusterIP类型变成NodePort类型，供集群外部访问

在原有的基础上，增加一个字段type，如下所示

**cat service.yaml**

apiVersion: v1

kind: Service

metadata:

name: my-nginx

labels:

run: my-nginx

spec:

ports:

- port: 80

protocol: TCP

type:

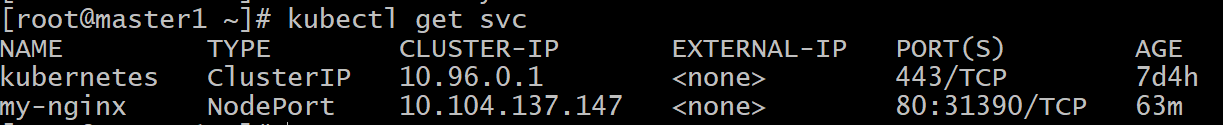
NodePort

selector:

run: my-nginx

kubectl apply -f service.yaml

**kubectl get svc**



上面截图可以发现my-nginx这个service 的TYPE类型变成了NodePort

**流量走向：**

访问物理节点（master节点或者node节点）ip:宿主机映射的端口--->service ip:service的端口----->pod ip:container port

**3.没有selector的Service**

**Servcie 抽象了该如何访问 Kubernetes Pod，但也能够抽象其它类型的backend，例如：**

1）希望在生产环境中使用外部的数据库集群，但测试环境使用自己的数据库。

2）希望服务指向另一个**[Namespace](https://kubernetes.io/docs/user-guide/namespaces)**中或其它集群中的服务。

3）正在将工作负载转移到 Kubernetes 集群

在任何这些场景中，都能够定义没有selector 的**Service** ：

kind: Service

apiVersion: v1

metadata:

name: my-service

spec:

ports:

- protocol: TCP

port: 80

targetPort: 3306

由于这个Service没有selector，就不会创建相关的Endpoints对象。可以手动将Service映射到指定的Endpoints：

kind: Endpoints

apiVersion: v1

metadata:

name: my-service

subsets:

- addresses:

- ip: 1.2.3.4 #这个地址可以是集群外部的地址，如mysql等

ports:

- port: 3306

**注意**：

Endpoint IP 地址不能是 loopback（127.0.0.0/8）、 link-local（169.254.0.0/16）、或者 link-local 多播（224.0.0.0/24）。

访问没有 selector的**Service**，与有selector的**Service**的原理相同。请求将被路由到用户定义的Endpoint（该示例中为**1.2.3.4:3306**）。

**type类型是ExternalName，k8s集群内到外的访问**

ExternalName **Service**是**Service**的特例，它没有selector，也没有定义任何的端口和 Endpoint。 相反地，对于运行在集群外部的服务，它通过返回该外部服务的别名这种方式来提供服务。

kind: Service

apiVersion: v1

metadata:

name: my-service

namespace: prod

spec:

type: ExternalName

externalName: my.database.example.com

当查询主机my-service.prod.svc.CLUSTER时，集群的DNS服务将返回一个值为my.database.example.com的CNAME记录。 访问这个服务的工作方式与其它的相同，唯一不同的是重定向发生在DNS层，而且不会进行代理或转发。 如果后续决定要将数据库迁移到Kubernetes集群中，可以启动对应的 Pod，增加合适的Selector 或 Endpoint，修改Service的type即可。

**Headless Service**

有时不需要或不想要负载均衡，以及单独的 Service IP。 遇到这种情况，可以通过指定 Cluster IP（spec.clusterIP）的值为 "None" 来创建 Headless Service。

这个选项允许开发人员自由寻找他们自己的方式，从而降低与 Kubernetes 系统的耦合性。 应用仍然可以使用一种自注册的模式和适配器，对其它需要发现机制的系统能够很容易地基于这个 API 来构建。对这类 Service 并不会分配 Cluster IP，kube-proxy 不会处理它们，而且平台也不会为它们进行负载均衡和路由。 DNS如何实现自动配置，依赖于Service是否定义了 selector

**配置 Selector**

对定义selector的Headless Service，Endpoint 控制器在 API 中创建了 Endpoints 记录，并且修改 DNS 配置返回 A 记录（地址），通过这个地址直接到达Service的后端Pod上

**不配置 Selector**

对没有定义 selector 的 Headless Service，Endpoint 控制器不会创建Endpoints记录。

**发布服务-服务类型**

对一些应用（如 Frontend）的某些部分，可能希望通过外部（Kubernetes 集群外部）IP 地址暴露 Service。Kubernetes Service Types允许指定一个需要的类型的Service，默认是ClusterIP类型。

**Type的类型如下：**

**1.ClusterIP：**

通过集群的内部IP暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的ServiceType。

**2.NodePort：**

通过每个Node上的IP和静态端口（NodePort）暴露服务。NodePort服务会路由到ClusterIP服务，这个ClusterIP服务会自动创建。通过请求<NodeIP>:<NodePort>，可以从集群的外部访问一个NodePort服务。

**3.LoadBalancer：**

使用云提供商的负载局衡器，可以向外部暴露服务。外部的负载均衡器可以路由到NodePort服务和ClusterIP 服务。

**4.ExternalName：**

通过返回CNAME和它的值，可以将服务映射到externalName字段的内容（例如， foo.bar.example.com）。 没有任何类型代理被创建，这只有 Kubernetes 1.7 或更高版本的kube-dns才支持。

**NodePort 类型**

如果设置type的值为 "NodePort"，Kubernetes master 将从给定的配置范围内（默认：30000-32767）分配端口，每个Node将从该端口（每个 Node 上的同一端口）代理到Service。该端口将通过Service的spec.ports[\*].nodePort字段被指定。如果需要指定的端口号，可以配置nodePort的值，系统将分配这个端口，否则调用 API 将会失败（比如，需要关心端口冲突的可能性）。这可以让开发人员自由地安装他们自己的负载均衡器，并配置 Kubernetes 不能完全支持的环境参数，或者直接暴露一个或多个 Node 的IP 地址。需要注意的是，Service 将能通过<NodeIP>:spec.ports[\*].nodePort和 spec.clusterIp:spec.ports[\*].port 而对外可见。

**LoadBalancer 类型**

使用支持外部负载均衡器的云提供商的服务，设置type的值为 "LoadBalancer"，将为Service提供负载均衡器。 负载均衡器是异步创建的，关于被提供的负载均衡器的信息将会通过Service的status.loadBalancer字段被发布出去。

kind: Service

apiVersion: v1

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

nodePort: 30061

clusterIP: 10.0.171.239

loadBalancerIP: 78.11.24.19

type: LoadBalancer

status:

loadBalancer:

ingress:

- ip: 146.148.47.155

来自外部负载均衡器的流量将直接打到 backend Pod 上，不过实际它们是如何工作的，这要依赖于云提供商。 在这些情况下，将根据用户设置的 loadBalancerIP 来创建负载均衡器。 某些云提供商允许设置 loadBalancerIP。如果没有设置 loadBalancerIP，将会给负载均衡器指派一个临时 IP。 如果设置了 loadBalancerIP，但云提供商并不支持这种特性，那么设置的 loadBalancerIP值将会被忽略掉。

**AWS 内部负载均衡器**

在混合云环境中，有时从虚拟私有云（VPC）环境中的服务路由流量是非常有必要的。 可以通过在 Service 中增加 annotation 来实现，如下所示：

[...]

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0

[...]

在水平分割的 DNS 环境中，需要两个 Service 来将外部和内部的流量路由到 Endpoint 上。

**AWS SSL 支持**

对运行在 AWS 上部分支持 SSL 的集群，从 1.3 版本开始，可以为 LoadBalancer 类型的 Service 增加两个 annotation：

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012

第一个 annotation 指定了使用的证书。它可以是第三方发行商发行的证书，这个证书或者被上传到 IAM，或者由 AWS 的证书管理器创建。

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-backend-protocol: (https|http|ssl|tcp)

第二个 annotation 指定了 **Pod** 使用的协议。 对于 HTTPS 和 SSL，ELB 将期望该 **Pod** 基于加密的连接来认证自身。HTTP 和 HTTPS 将选择7层代理：ELB 将中断与用户的连接，当转发请求时，会解析 Header 信息并添加上用户的 IP 地址（**Pod** 将只能在连接的另一端看到该 IP 地址）。

TCP 和 SSL 将选择4层代理：ELB 将转发流量，并不修改 Header 信息。

**外部 IP**

如果外部的 IP 路由到集群中一个或多个 Node 上，Kubernetes **Service** 会被暴露给这些 **externalIPs**。 通过外部 IP（作为目的 IP 地址）进入到集群，打到 **Service**的端口上的流量，将会被路由到 **Service** 的 Endpoint 上。 **externalIPs** 不会被 Kubernetes 管理，它属于集群管理员的职责范畴。

根据 **Service** 的规定，**externalIPs** 可以同任意的的一个 **ServiceType** 来一起指定。 在上面的例子中，**my-service** 可以在 80.11.12.10:80（外部 IP:端口）上被客户端访问。

**kind: Service**

**apiVersion: v1**

**metadata:**

**name: my-service**

**spec:**

**selector:**

**app: MyApp**

**ports:**

**- name: http**

**protocol: TCP**

**port: 80**

**targetPort: 3306**

**externalIPs:**

**- 80.11.12.10**

[https://kubernetes.io/zh/docs/concepts/services-networking/service/https://kubernetes.io/zh/docs/concepts/services-networking/service/](https://kubernetes.io/zh/docs/concepts/services-networking/service/)