

## 37 | 找到容器不容易：Service、DNS与服务发现

# 37 | 找到容器不容易：Service、DNS与服务发现

张磊 2018-11-16



□

10:56

讲述：张磊 大小：5.01M

你好，我是张磊。今天我和你分享的主题是：找到容器不容易之 Service、DNS与服务发现。

在前面的文章中，我们已经多次使用到了 Service 这个 Kubernetes 里重要的服务对象。而 Kubernetes 之所以需要 Service，一方面是因为 Pod 的 IP 不是固定的，另一方面则是因为一组 Pod 实例之间总会有负载均衡的需求。

一个最典型的 Service 定义，如下所示：

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:  
  name: hostnames  
  
spec:  
  selector:  
    app: hostnames  
  
  ports:  
    - name: default  
      protocol: TCP  
      port: 80  
      targetPort: 9376
```

#### □复制代码

这个 Service 的例子，相信你不会陌生。其中，我使用了 selector 字段来声明这个 Service 只代理携带了 app=hostnames 标签的 Pod。并且，这个 Service 的 80 端口，代理的是 Pod 的 9376 端口。

然后，我们的应用的 Deployment，如下所示：

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: hostnames  
  
spec:  
  selector:  
    matchLabels:  
      app: hostnames  
  replicas: 3  
  template:  
    metadata:  
      labels:  
        app: hostnames  
    spec:  
      containers:  
        - name: hostnames  
          image: k8s.gcr.io/serve_hostname
```

ports:

- containerPort: 9376

protocol: TCP

□复制代码

这个应用的作用，就是每次访问 9376 端口时，返回它自己的 hostname。

而被 selector 选中的 Pod，就称为 Service 的 Endpoints，你可以使用 `kubectl get ep` 命令看到它们，如下所示：

```
$ kubectl get endpoints hostnames
```

```
NAME ENDPOINTS
```

```
hostnames 10.244.0.5:9376,10.244.0.6:9376,10.244.0.7:9376
```

□复制代码

需要注意的是，只有处于 Running 状态，且 readinessProbe 检查通过的 Pod，才会出现在 Service 的 Endpoints 列表里。并且，当某一个 Pod 出现问题时，Kubernetes 会自动把它从 Service 里摘除掉。

而此时，通过该 Service 的 VIP 地址 10.0.1.175，你就可以访问到它所代理的 Pod 了：

```
$ kubectl get svc hostnames
```

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

```
hostnames ClusterIP 10.0.1.175 <none> 80/TCP 5s
```

```
$ curl 10.0.1.175:80
```

```
hostnames-0uton
```

```
$ curl 10.0.1.175:80
```

```
hostnames-yp2kp
```

```
$ curl 10.0.1.175:80
```

```
hostnames-bvc05
```

□复制代码

这个 VIP 地址是 Kubernetes 自动为 Service 分配的。而像上面这样，通过三次连续不断地访问 Service 的 VIP 地址和代理端口 80，它就为我们依次返回了三个 Pod 的 hostname。这也正印证了 Service 提供的是 Round Robin 方式的负载均衡。对于这种方式，我们称为：ClusterIP 模式的 Service。

你可能一直比较好奇，Kubernetes 里的 Service 究竟是如何工作的呢？

实际上，**Service 是由 kube-proxy 组件，加上 iptables 来共同实现的。**

举个例子，对于我们前面创建的名叫 hostnames 的 Service 来说，一旦它被提交给 Kubernetes，那么 kube-proxy 就可以通过 Service 的 Informer 感知到这样一个 Service 对象的添加。而作为对这个事件的响应，它就会在宿主机上创建这样一条 iptables 规则（你可以通过 iptables-save 看到它），如下所示：

```
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment
"default/hostnames: cluster IP" -m tcp --dport 80 -j KUBE-SVC-
NWV5X2332I4OT4T3
```

□复制代码

可以看到，这条 iptables 规则的含义是：凡是目的地址是 10.0.1.175、目的端口是 80 的 IP 包，都应该跳转到另外一条名叫 KUBE-SVC-NWV5X2332I4OT4T3

的 iptables 链进行处理。

而我们前面已经看到，10.0.1.175 正是这个 Service 的 VIP。所以这一条规则，就为这个 Service 设置了一个固定的入口地址。并且，由于 10.0.1.175 只是一条 iptables 规则上的配置，并没有真正的网络设备，所以你 ping 这个地址，是不会有响应的。

那么，我们即将跳转到的 KUBE-SVC-NWV5X2332I4OT4T3 规则，又有什么作用呢？

实际上，它是一组规则的集合，如下所示：

```
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment
"default/hostnames:" -m statistic --mode random --probability
0.33332999982 -j KUBE-SEP-WNBA2IHDGP2BOBGZ

-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment
"default/hostnames:" -m statistic --mode random --probability
0.50000000000 -j KUBE-SEP-X3P2623AGDH6CDF3

-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment
"default/hostnames:" -j KUBE-SEP-57KPRZ3JQVENLNBR
```

□复制代码

可以看到，这一组规则，实际上是一组随机模式（-mode random）的 iptables 链。

而随机转发的目的地，分别是 KUBE-SEP-WNBA2IHDGP2BOBGZ、KUBE-SEP-X3P2623AGDH6CDF3 和 KUBE-SEP-57KPRZ3JQVENLNBR。

而这三条链指向的最终目的地，其实就是这个 Service 代理的三个 Pod。所以这一组规则，就是 Service 实现负载均衡的位置。

需要注意的是，iptables 规则的匹配是从上到下逐条进行的，所以为了保证上述三条规则每条被选中的概率都相同，我们应该将它们的 probability 字段的值分别设置为 1/3 (0.333...)、1/2 和 1。

这么设置的原理很简单：第一条规则被选中的概率就是 1/3；而如果第一条规则没有被选中，那么这时候就只剩下两条规则了，所以第二条规则的 probability 就必须设置为 1/2；类似地，最后一条就必须设置为 1。

你可以想一下，如果把这三条规则的 probability 字段的值都设置成 1/3，最终每条规则被选中的概率会变成多少。

通过查看上述三条链的明细，我们就很容易理解 Service 进行转发的具体原理了，如下所示：

```
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0x00004000/0x00004000
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination 10.244.3.6:9376
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0x00004000/0x00004000
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination 10.244.1.7:9376
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "default/hostnames:" -j MARK --set-xmark 0x00004000/0x00004000
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT --to-destination 10.244.2.3:9376
```

#### □复制代码

可以看到，这三条链，其实是三条 DNAT 规则。但在 DNAT 规则之前，iptables 对流入的 IP 包还设置了一个“标志”（--set-xmark）。这个“标志”的作用，我会在下一篇文章再为你讲解。

而 DNAT 规则的作用，就是在 PREROUTING 检查点之前，也就是在路由之前，将流入 IP 包的目的地地址和端口，改成--to-destination 所指定的新的目的地地址和端口。可以看到，这个目的地地址和端口，正是被代理 Pod 的 IP 地址和端口。

这样，访问 Service VIP 的 IP 包经过上述 iptables 处理之后，就已经变成了访问具体某一个后端 Pod 的 IP 包了。不难理解，这些 Endpoints 对应的 iptables 规则，正是 kube-proxy 通过监听 Pod 的变化事件，在宿主机上生成并维护的。

以上，就是 Service 最基本的工作原理。

此外，你可能已经听说过，Kubernetes 的 kube-proxy 还支持一种叫作 IPVS 的模式。这又是怎么回事儿呢？

其实，通过上面的讲解，你可以看到，kube-proxy 通过 iptables 处理 Service 的过程，其实需要在宿主机上设置相当多的 iptables 规则。而且，kube-proxy 还需要在控制循环里不断地刷新这些规则来确保它们始终是正确的。

不难想到，当你的宿主机上有大量 Pod 的时候，成百上千条 iptables 规则不断地被刷新，会大量占用该宿主机的 CPU 资源，甚至会让宿主机“卡”在这个过程中。所以说，**一直以来，基于 iptables 的 Service 实现，都是制约 Kubernetes 项目承载更多量级的 Pod 的主要障碍。**

而 IPVS 模式的 Service，就是解决这个问题一个行之有效的方法。

IPVS 模式的工作原理，其实跟 iptables 模式类似。当我们创建了前面的 Service 之后，kube-proxy 首先会在宿主机上创建一个虚拟网卡（叫作：kube-ipvs0），并为它分配 Service VIP 作为 IP 地址，如下所示：

```
# ip addr
...
73: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state
DOWN qlen 1000
link/ether 1a:ce:f5:5f:c1:4d brd ff:ff:ff:ff:ff:ff
inet 10.0.1.175/32 scope global kube-ipvs0
valid_lft forever preferred_lft forever
```

□复制代码

而接下来，kube-proxy 就会通过 Linux 的 IPVS 模块，为这个 IP 地址设置三个 IPVS 虚拟主机，并设置这三个虚拟主机之间使用轮询模式 (rr) 来作为负载均衡策略。我们可以通过 ipvsadm 查看到这个设置，如下所示：

```
# ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
-> RemoteAddress:Port Forward Weight ActiveConn InActConn
TCP 10.102.128.4:80 rr
-> 10.244.3.6:9376 Masq 1 0 0
-> 10.244.1.7:9376 Masq 1 0 0
```



-> 10.244.2.3:9376 Masq 1 0 0

#### □复制代码

可以看到，这三个 IPVS 虚拟主机的 IP 地址和端口，对应的正是三个被代理的 Pod。

这时候，任何发往 10.102.128.4:80 的请求，就都会被 IPVS 模块转发到某一个后端 Pod 上了。

而相比于 iptables，IPVS 在内核中的实现其实也是基于 Netfilter 的 NAT 模式，所以在转发这一层上，理论上 IPVS 并没有显著的性能提升。但是，IPVS 并不需要在宿主机上为每个 Pod 设置 iptables 规则，而是把对这些“规则”的处理放到了内核态，从而极大地降低了维护这些规则的代价。这也正印证了我在前面提到过的，“将重要操作放入内核态”是提高性能的重要手段。

**备注：**这里你可以再回顾下第 33 篇文章 [《深入解析容器跨主机网络》](#) 中的相关内容。

不过需要注意的是，IPVS 模块只负责上述的负载均衡和代理功能。而一个完整的 Service 流程正常工作所需要的包过滤、SNAT 等操作，还是要靠 iptables 来实现。只不过，这些辅助性的 iptables 规则数量有限，也不会随着 Pod 数量的增加而增加。

所以，在大规模集群里，我非常建议你为 kube-proxy 设置 `-proxy-mode=ipvs` 来开启这个功能。它为 Kubernetes 集群规模带来的提升，还是非常巨大的。

**此外，我在前面的文章中还介绍过 Service 与 DNS 的关系。**

在 Kubernetes 中，Service 和 Pod 都会被分配对应的 DNS A 记录（从域名解析 IP 的记录）。

对于 ClusterIP 模式的 Service 来说（比如我们上面的例子），它的 A 记录的格式是：..svc.cluster.local。当你访问这条 A 记录的时候，它解析到的就是该 Service 的 VIP 地址。

而对于指定了 `clusterIP=None` 的 Headless Service 来说，它的 A 记录的格式也是：..svc.cluster.local。但是，当你访问这条 A 记录的时候，它返回的是所有被代理的 Pod 的 IP 地址的集合。当然，如果你的客户端没办法解析这个集合的话，它可能会只会拿到第一个 Pod 的 IP 地址。

此外，对于 ClusterIP 模式的 Service 来说，它代理的 Pod 被自动分配的 A 记录的格式是：..pod.cluster.local。这条记录指向 Pod 的 IP 地址。

而对 Headless Service 来说，它代理的 Pod 被自动分配的 A 记录的格式是：...svc.cluster.local。这条记录也指向 Pod 的 IP 地址。

但如果你为 Pod 指定了 Headless Service，并且 Pod 本身声明了 hostname 和 subdomain 字段，那么这时候 Pod 的 A 记录就会变成：<pod 的 hostname>...svc.cluster.local，比如：

```
apiVersion: v1
kind: Service
metadata:
  name: default-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
    - name: foo
      port: 1234
      targetPort: 1234
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
labels:
  name: busybox
spec:
  hostname: busybox-1
  subdomain: default-subdomain
  containers:
    - image: busybox
      command:
        - sleep
        - "3600"
```



name: busybox

□复制代码

在上面这个 Service 和 Pod 被创建之后，你就可以通过 `busybox-1.default-subdomain.default.svc.cluster.local` 解析到这个 Pod 的 IP 地址了。

需要注意的是，在 Kubernetes 里，`/etc/hosts` 文件是单独挂载的，这也是为什么 kubelet 能够对 hostname 进行修改并且 Pod 重建后依然有效的原因。这跟 Docker 的 Init 层是一个原理。

## 总结

在这篇文章里，我为你详细讲解了 Service 的工作原理。实际上，Service 机制，以及 Kubernetes 里的 DNS 插件，都是在帮助你解决同样一个问题，即：如何找到我的某一个容器？

这个问题在平台级项目中，往往就被称作服务发现，即：当我的一个服务（Pod）的 IP 地址是不固定的且没办法提前获知时，我该如何通过一个固定的方式访问到这个 Pod 呢？

而我在这里讲解的、ClusterIP 模式的 Service 为你提供的，就是一个 Pod 的稳定的 IP 地址，即 VIP。并且，这里 Pod 和 Service 的关系是可以通过 Label 确定的。

而 Headless Service 为你提供的，则是一个 Pod 的稳定的 DNS 名字，并且，这个名字是可以通过 Pod 名字和 Service 名字拼接出来的。

在实际的场景里，你应该根据自己的具体需求进行合理选择。

## 思考题

请问，Kubernetes 的 Service 的负载均衡策略，在 iptables 和 ipvs 模式下，都有哪几种？具体工作模式是怎样的？