原 Kubernetes系列05：深入掌握Service

2017年04月21日 18:27:42 levy_cui 阅读数：11918 更多

Service是kubernetes最核心的概念，通过创建Service，可以为一组具有相同功能的容器应用提供一个统一的入口地址，并且将请求进行负载分发到后端的各个容器应用上。

本节对Service的使用进行说明，包括Service的负载均衡、外网访问、DNS服务的搭建、Ingress7层路由机制等

1.Service定义详解
yaml格式的Service定义文件的完整内容：

```
apiVersion: v1
kind: Service
matadata:
  name: string
  namespace: string
  labels:
    - name: string
  annotations:
    - name: string
spec:
  selector: []
  type: string
  clusterIP: string
  sessionAffinity: string
  ports:
```

```
- name: string
  protocol: string
  port: int
  targetPort: int
  nodePort: int
status:
  loadBalancer:
    ingress:
      ip: string
      hostname: string
```

表 2.17　对 Service 的定义文件模板的各属性的说明

属性名称	取值类型	是否必选	取值说明
version	string	Required	v1
kind	string	Required	Service
metadata	object	Required	元数据
metadata.name	string	Required	Service 名称，需符合 RFC 1035 规范
metadata.namespace	string	Required	命名空间，不指定系统时将使用名为“default”的命名空间
metadata.labels[]	list		自定义标签属性列表
metadata.annotation[]	list		自定义注解属性列表
spec	object	Required	详细描述
spec.selector[]	list	Required	Label Selector 配置，将选择具有指定 Label 标签的 Pod 作为管理范围
spec.type	string	Required	Service 的类型，指定 Service 的访问方式，默认为 ClusterIP。 ClusterIP：虚拟的服务 IP 地址，该地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置的 iptables 规则进行转发。 NodePort：使用宿主机的端口，使能够访问各 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务。 LoadBalancer：使用外接负载均衡器完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，并同时定义 nodePort 和 clusterIP，用于公有云环境
spec.clusterIP	string		虚拟服务 IP 地址，当 type=ClusterIP 时，如果不指定，则系

			统进行自动分配，也可以手工指定：当 type=LoadBalancer 时，则需要指定

spec.sessionAffinity	string		是否支持 Session，可选值为 ClientIP，默认为空。ClientIP：表示将同一个客户端（根据客户端的 IP 地址决定）的访问请求都转发到同一个后端 Pod
spec.ports[]	list		Service 需要暴露的端口列表
spec.ports[].name	string		端口名称
spec.ports[].protocol	string		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.ports[].port	int		服务监听的端口号
spec.ports[].targetPort	int		需要转发到后端 Pod 的端口号
spec.ports[].nodePort	int		当 spec.type=NodePort 时，指定映射到物理机的端口号
Status	object		当 spec.type=LoadBalancer 时，设置外部负载均衡器的地址，用于公有云环境
status.loadBalancer	object		外部负载均衡器
status.loadBalancer.ingress	object		外部负载均衡器
status.loadBalancer.ingress.ip	string		外部负载均衡器的 IP 地址
status.loadBalancer.ingress.hostname	string		外部负载均衡器的主机名

对外提供服务的应用程序需要通过某种机制来实现，对于容器应用最简单的方式就是通过TCP/IP机制及监听IP和端口来实现。例如，我们定义一个提供web服务的RC，由两个tomcat容器副本组成，每个容器通过containerPort设置提供服务号为8080

文件webapp-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: webapp
spec:
  replicas: 2
  template:
    metadata:
```

```
    name: webapp
    labels:
      app: webapp
  spec:
    containers:
    - name: webapp
      image: tomcat
      ports:
      - containerPort: 80
```

创建RC：

```
#kubectl create -f webapp-rc.yaml
```

获取Pod的IP地址：

```
#kubectl get pods -l app=webapp -o yaml|grep podIP
 podIP:172.17.1.4
 podIP:172.17.1.5
```

访问这两个Pod提供的Tomcat服务

```
#curl 172.17.1.4:8080
```

直接通过Pod的IP地址和端口号访问容器应用是不可靠的，例如Podu所在的Node发生故障，Pod将被重新调度到另一台Node进行启动，Pod的IP地址将发生变化，更重要的是，如果容器应用本身是分布式的部署方式，通过多个实例共同提供服务，就需要在这些实例的前端设置一个负载均衡器来实现请求的分发。kubernetes中的Service就是设计出来用于解决这些问题的核心组件。

为了让客户端应用能够访问到两个Tomcat Pod 实例，需要创建一个Service来提供服务，通过kubectl expose命令来创建：

```
#kubectl expose rc webapp
```

查看新创建的Service可以看到系统为它分配了一个虚拟的IP地址（clusterIP），而Service所需的端口号则从Pod中的containerPort复制而来：

```
#kubectl get svc
NAME |Cluster-IP |External-IP |Port(s) |AGE
webapp |169.169.235.79 |<none> |8080/TCP |3s
```

接下来，我们就可以通过Service的IP地址和Service的端口号访问该Service了：
#curl 169.169.235.79:8080
这里，对Service地址169.169.235.79:8080的访问被自动负载分发到了后端两个Pod之一
除了使用kubectl expose命令创建Service，我们也可以通过配置文件定义Service，再通过kubectl

create命令进行创建。我们可以设置一个Service：
文件webapp-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
  - port: 8081
    targetPort: 8080
  selector:
    app: webapp
```

Service定义中的关键字段是ports和selector。本例ports定义部分指定了Service所需的虚拟端口号为8081，由于与Pod容器端口号8080不一样，所以需要在通过targetPort来指定后端Pod的端口。

selector定义部分设置的是后端Pod所拥有的label: app=webapp
创建该Service并查看器ClusterIP地址：
#kubectl create -f webapp-svc.yaml
#kubectl get svc
#curl 169.169.28.190:8081

目前kubernetes提供了两种负载分发策略：RoundRobin和SessionAffinity
RoundRobin：轮询模式，即轮询将请求转发到后端的各个Pod上
SessionAffinity：基于客户端IP地址进行会话保持的模式，第一次客户端访问后端某个Pod，之后的请求都转发到这个Pod上
默认是RoundRobin模式

在某些场景中，开发人员希望自己控制负载均衡的策略，不使用Service提供的默认负载，kubernetes通过Headless Service的概念来实现。不给Service设置ClusterIP（无入口IP地址）

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
  clusterIP: None
  selector:
    app: nginx
```

有时候，一个容器应用提供多个端口服务：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
  - port: 8080
    targetPort: 8080
    name: web
  - port: 8005
    targetPort: 8005
    name: management
  selector:
```

```
    app: webapp
```
另一个例子是两个端口使用了不同的4层协议，即TCp或UDP
```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

3. 集群外部访问Pod或Service
为了让外部客户端可以访问这些服务，可以将Pod或者Service的端口号映射到宿主主机，使得客户端应用能够通过物理机访问容器应用。

1）将容器应用的端口号映射到物理机
（1）通过设置容器级别的hostPort，将容器应用的端口号映射到物理机上：
文件pod-hostport.yaml
```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  containers:
  - name: webapp
    image: tomcat
    ports:
    - containerPort: 8080
      hostPort:8081
```

通过kubectl create创建这个Pod：

```
#kubectl create -f pod-hostport.yaml
```

通过物理机的IP地址和8081端口号访问Pod内的容器服务：

```
#curl 192.168.18.3:8081
```

（2）通过设置Pod级别的hostNetwork=true，该Pod中所有容器的端口号都将被直接映射到物理机上，设置hostWork=true是需要注意，在容器的ports定义部分如果不指定hostPort，则默认hostPort等于containerPort，如果指定了hostPort，则hostPort必须等于containerPort的值。

文件pod-hostnetwork.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  hostNetwork: true
  containers:
```

```
  - name: webapp
    image: tomcat
    imagePullPolicy: Never
    ports:
    - containerPort: 8080
```

创建这个Pod：

```
#kubectl create -f pod-hostnetwork.yaml
```

通过物理机的IP地址和8080端口访问Pod的容器服务

```
#curl 192.168.18.4:8080
```

2）将Service的端口号映射到物理机

（1）通过设置nodePort映射到物理机，同时设置Service的类型为NodePort：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    nodePort: 8081
  selector:
    app: webapp
```

创建这个Service：

```
#kubectl create -f webapp-svc-nodeport.yaml
```

通过物理机的IP和端口访问：

```
#curl 192.168.18.3:8081
```

如果访问不通，查看下物理机的防火墙设置

同样，对该Service的访问也将被负载分发到后端多个Pod上

（2）通过设置LoadBalancer映射到云服务商提供的LoadBalancer地址。这种用法仅用于在公有云服务提供商的云平台上设置Service的场景。status.loadBalancer.ingress.ip设置的146.148.47.155为云服务商提供的负载均衡器的IP地址。对该Service的访问请求将会通过LoadBalancer转发到后端的Pod上，负载分发的实现方式依赖云服务商提供的LoadBalancer的实现机制。

```
apiVersion: v1
kind: Service
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:
    ingree:
    - ip: 146.148.47.155
```

4. DNS服务搭建指南

为了能够通过服务的名字在集群内部进行服务的相互访问，需要创建一个虚拟的DNS服务来完成服务名到ClusterIP的解析。

kubernetes提供的虚拟DNS服务名为skydns，由四个组件组成。

1）etcd：NDS存储

2）kube2sky：将kubernetes Master中的Service（服务）注册到etcd

3）skyDNS：提供NDS域名解析服务

4）healthz：提供对skydns服务的健康检查功能

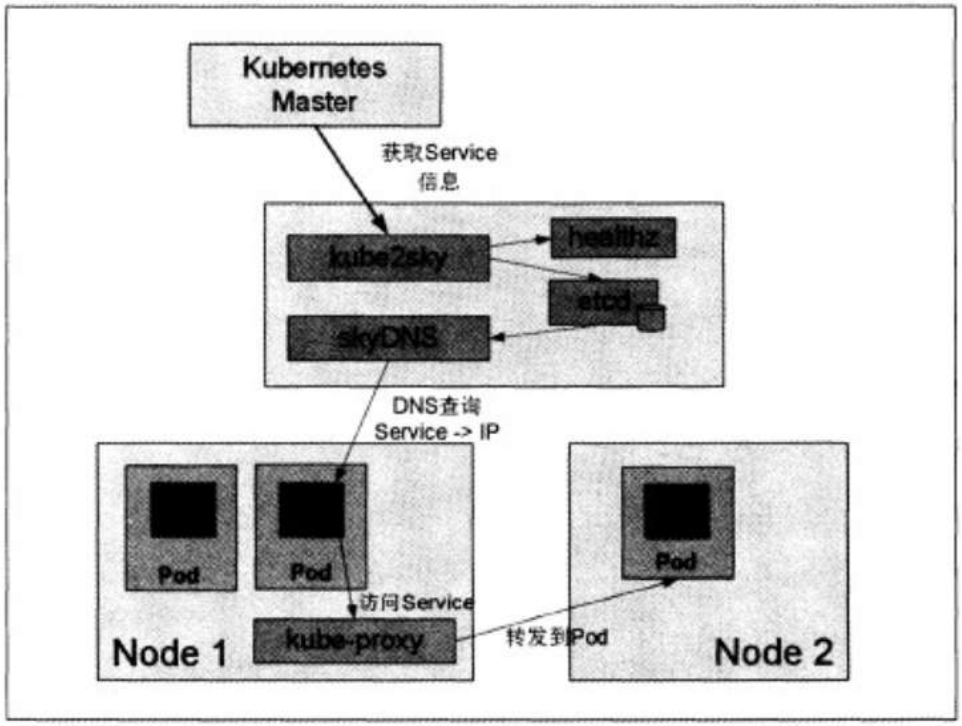

图 2.17　Kubernetes DNS 服务的总体架构

1）skydns配置文件说明

skydns服务由一个RC和一个Service的定义组成，分别由配置文件skydns-rc.yaml和skydns-svc.yaml定义skydns的RC配置文件skydns-rc.yaml的内容如下，包含4个容器的定义：

文件skydns-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v11
  namespace: kube-system
```

```
  labels:
    k8s-app: kube-dns
    version: v11
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v11
  template:
    metadata:
      labels:
        k8s-app: kube-dns
        version: v11
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: etcd
        image: gcr.io/google_containers/etcd-amd64:2.2.1
        resources:
          limits:
            cpu: 100m
            memory: 50Mi
          requests:
            cpu: 100m
            memory: 50Mi
        command:
        - /usr/local/bin/etcd
        - -data-dir
        - /tmp/data
        - -listten-client-urls
```

```
    - http://127.0.0.1:2379,http://127.0.0.1:4001
    - -advertise-client-urls
    - http://127.0.0.1:2379,http://127.0.0.1:4001
    - -initial-cluster-token
    - skydns-etcd
    volumeMounts:
    - name: etcd-storage
      mountPath: /tmp/data
  - name: kube2sky
    image: gcr.io/google_containers/kube2sky-amd64:1.15
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 60
      timeoutSeconds: 5
      successThreshold: 1
      failureThreshold: 5
    readlinessProbe:
      httpGet:
        path: /readiness
        port: 8081
        scheme: HTTP
      initialDelaySeconds: 30
      timeoutSeconds: 5
```

```
  args:
  - --kube-master-url=http://192.168.18.3:8080
  - --domain=cluster.local
- name: skydns
  image: gcr.io/google_containers/skydns:2015-10-13-8c72f8c
  resources:
    limits:
      cpu: 100m
      memory: 50Mi
    requests:
      cpu: 100m
      memory: 50Mi
  args:
  - -machines=http://127.0.0.1:4001
  - -addr=0.0.0.0:53
  - -ns-rostate=false
  - -domain=cluster.local
  ports:
  - containerPort: 53
    name: dns
    protocol: UDP
  - containerPort: 53
    name: dns-tcp
    protocol: TCP
  - name: healthz
    image:gcr.io/google_containers/exechealthz:1.0
    resources:
      limits:
        cpu: 10m
        memory: 20Mi
      requests:
```

```
            cpu: 10m
            memory: 20Mi
        args:
        - -cmd=nslookup kubernetes.default.svc.cluster.local 127.0.0.1 > /dev/null
        - -port=8080
        ports:
        - containerPort: 8080
          protocol: TCP
      volumes:
      - name: etcd-storage
        emptyDir: {}
      dnsPolicy: Default  #Don't use cluster DNS.
```

skydns的Service配置文件skydns-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service:"true"
    kubernetes.io/name: "kubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
  - name: dns
```

```
    prot: 53
    protocol: UDP
  - name:dns-tcp
    port: 53
    protocol: TCP
```

注意：skydns服务使用的clusterIP需要指定一个固定IP，每个Node的kubelet进程都将使用这个IP地址，不能通过kubernetes自动分配。

另外，这个IP地址需要在kube-apiserver启动参数--service-cluster-ip-range指定IP地址范围在创建skydns容器之前，先修改每个Node上kubelet启动参数。

2）修改每台Node上的kubelet启动参数
加上一下两个参数：
--cluster_dns=169.169.0.100 为NDS服务的ClusterIP地址
--cluster_dns=cluster.local 为DNS服务中设置的域名

然后重启kubelet服务

3）创建skydns RC和Service
#kubectl create -f skydns-rc.yaml
#kubectl create -f skydns-svc.yaml
查看RC、Pod和Service，确保容器启动
#kubectl get rc --namespace=kube-system

#kubectl get pods --namespace=kube-system

#kubectl get service --namespace=kube-system

然后我们为redis-master应用创建一个Service：
文件redis-master-service.yaml
apiVersion: v1

```
kind: Service
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    name: redis-master
```

到此，集群虚拟DNS服务搭建完成，在需要访问redis-master的应用中，仅需要配置上redis-master Service的名字和服务的端口号，就能可以访问，如redis-master:6379

4）通过DNS查找Service

使用一个带有nslookup工具的Pod来验证DNS服务：

文件busybox.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: gcr.io/google_containers/busybox
    conmmand:
      - sleep
      - "3600"
```

运行kubectl create -f busybox.yaml

测试nslookup
#kubectl exec busybox -- nslookup redis-master

如果某个Service属于不同的命名空间，需要带上namespace名字
#kubectl exec busybox -- nslookup kube-dns.kube-system

5）DNS服务的工作原理解析
（1）kube2sky容器应用通过调用kubernetes Master的API获得集群中所有Service的信息，并持续监控新Service的生成，然后写入etcd中。

查看etcd中存储的Service信息：
#kubectl exec kube-dns-v8-5tpm2 -c etcd --namespace=kube-system etcdctl ls

/skydns/local/cluster
（2）根据kubelet启动参数的设置（--cluster_dns）,kubelet会在每个新创建的Pod中设置DNS域名

解析配置文件/etc/resolv.conf文件，在其中增加了一条nameserver配置和一条search配置：
nameserver 169.169.0.100
通过服务器169.169.0.100访问的实际就是skydns在53端口上提供的DNS解析服务。

（3）最后应用程序就能够像访问网站域名一样，仅仅通过服务的名字就能够访问到服务了。

5. Ingress：HTT前面P 7层路由机制
根据前面对Service的使用说明，我们知道Service的表现形式为IP:Port，即工作在TCP/IP层，而对于基于HTTP的服务来说，不同的URL地址经常对应到不同的后端服务或者虚拟服务器，这些应用层的转发机制仅通过kubernetes的Service机制是无法实现的。kubernetes V1.1版本中新增的Ingress将不同URL的访问请求转发到后端不同的Service，实现HTTP层的业务路由机制。在kubernetes集群中，Ingress的实现需要通过Ingress的定义与Ingress Controller的定义结合起来，才能形成完整的HTTP负载分发功能。

- 对 http://mywebsite.com/api 的访问将被路由到后端名为“api”的 Service。
- 对 http://mywebsite.com/web 的访问将被路由到后端名为“web”的 Service。
- 对 http://mywebsite.com/doc 的访问将被路由到后端名为“doc”的 Service。

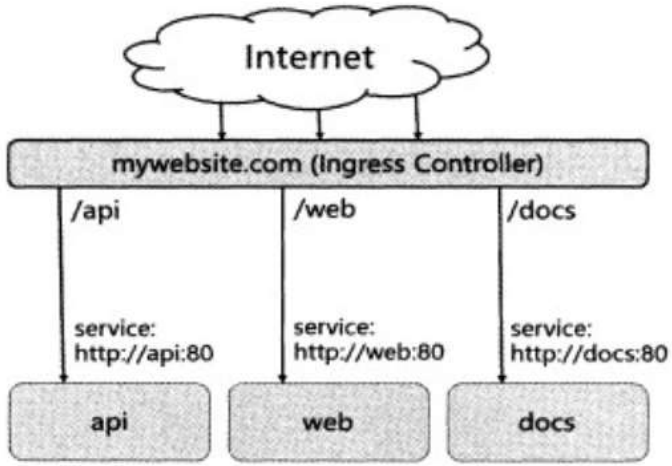

图 2.18　Ingress 示例

1）创建Ingress Controller

使用Nginx来实现一个Ingress Controller，需要实现的基本逻辑如下：

a）监听apiserver，获取全部ingress的定义

b）基于ingress的定义，生成Nginx所需的配置文件/etc/nginx/nginx.conf

c）执行nginx -s relaod命令，重新加载nginx.conf文件，写个脚本。

通过直接下载谷歌提供的nginx-ingress镜像来创建Ingress Controller：

文件nginx-ingress-rc.yaml

```
apiVersion: v1
kind: ReplicationController
matadata:
  name: nginx-ingress
  labels:
    app: nginx-ingress
```

```
spec:
  replicas: 1
  selector:
    app: nginx-ingress
  template:
    metadata:
      labels:
        app: nginx-ingress
    spec:
      containers:
      - image: gcr.io/google_containers/nginx-ingress:0.1
        name: nginx
        ports:
        - containerPort: 80
          hostPort: 80
```

这里，Nginx应用配置设置了hostPort，即它将容器应用监听的80端口号映射到物理机，以使得客户端应用可以通过URL地址“http://物理机IP:80”来访问该Ingress Controller

```
#kubectl create -f nginx-ingress-rc.yaml
#kubectl get pods
```

2)定义Ingress

为mywebsite.com定义Ingress，设置到后端Service的转发规则：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: mywebsite-ingress
spec:
  rules:
  - host: mywebsite.com
    http:
      paths:
```

```
      - path: /web
        backend:
          serviceName: webapp
          servicePort: 80
```

这个Ingress的定义说明对目标http://mywebsite.com/web的访问将被转发到kubernetes的一个Service上 webapp:80

创建该Ingress
#kubectl create -f Ingress.yaml

#kubectl get ingress
NAME |Hosts |Address |Ports |Age
mywebsite-ingress |mywebsite.com |80 |17s

创建后登陆nginx-ingress Pod，查看自动生成的nginx.conf内容

3）访问http://mywebsite.com/web
我们可以通过其他的物理机对其进行访问。通过curl --resolve进行指定
#curl --resolve mywebsite.com:80:192.169.18.3 mywebsite.com/web