简介

将运行在一组 Pods 上的应用程序公开为网络服务的抽象方法。这里的网络服务其实就是我们经常提起的微服务。

准备一组pod

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy
  namespace: default
  labels:
    name: myapp-deploy
spec:
  replicas: 5
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      restartPolicy: Always
      containers:
      - name: myhello
        image: xlhmzch/hello:curl
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        command: ["./app"]
        args: ["--param1=k8s-p1", "--param2=k8s-p2"]
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
        lifecycle:
          postStart:
            exec:
              command: ["/bin/sh", "-c","echo post start command exec >>
/tmp/data"]
          preStop:
```

```
            exec:
              command: ["/bin/sh", "-c","echo pre stop command exec >>
/tmp/data"]
        startupProbe:
          exec:
            command: ["/bin/sh", "-c","statuscode=`curl -o /dev/null -s -w %
{http_code} http://localhost/healthz`; [ $$statuscode -le 400 ] || exit 1"]
          # 指定初始化时间，即探针在容器启动之后的延迟时长
          initialDelaySeconds: 5
          # 探测周期，即每隔多长时间探测一次
          periodSeconds: 5
          # 最小连续失败次数，即连续失败多少次表示探测失败
          failureThreshold: 3
          # 最小连续成功次数，即连续探测成功多少次表示探测成功，liveness 和 startup 必须为1
          successThreshold: 1
          # 探测超时时间
          timeoutSeconds: 1
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
      - name: myredis                    #容器的名称
        image: redis       #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 6379
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        livenessProbe:
          tcpSocket:
            port: 6379
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
```

```
 kubectl apply -f myapp-deployment.yaml
 # 创建一个redis pod
kubectl run redis --image=redis
# 创建一个带有curl工具的pod
kubectl run curl --image=radial/busyboxplus:curl -it
```

service定义

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc # service 名称
spec:
  type: ClusterIP # service发布类型
  selector: # 标签选择器
    app: myapp
  ports:    # 指定端口数组可以指定多个
    - protocol: TCP # 协议
      name: http # 为端口指定一个端口名称
      port: 8080 # service 端口
      targetPort: 80 # 容器端口
    - protocol: TCP
      name: https
      port: 443
      targetPort: 80
    - protocol: TCP
      name: redis-tcp
      port: 6379
      targetPort: 6379
```

type 表示service发布类型:
ClusterIP： 自动分配一个仅clusterIP内部可以访问的虚拟ip地址
NodePort：在ClusterIP基础上为Service在每台机器上绑定一个端口，这样就可以通过节点IP:NodePort来访问服务
ExternalName：将服务映射到 DNS 名称，把集群外部的服务引入到集群内部来使用
LoadBalancer：使用云厂商提供的外部负载均衡器，来自外部负载均衡器的流量将直接重定向到后端 Pod 上。即service将内部的pod信息发布到了云厂商的负载均衡器，外部负载均衡直接将流量定向到了内部pod，不再需要内部负载均衡。

Service使用

service四种发布类型

ClusterIP

1. 定义

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc # service 名称
```

```
spec:
  type: ClusterIP # service发布类型
  selector: # 标签选择器
    app: myapp
  ports:    # 指定端口数组可以指定多个
    - protocol: TCP # 协议
      name: http # 为端口指定一个端口名称
      port: 8080 # service 端口
      targetPort: 80 # 容器端口
    - protocol: TCP
      name: https
      port: 443
      targetPort: 80
    - protocol: TCP
      name: redis-tcp
      port: 6379
      targetPort: 6379
```

2. 应用及查看service

```
# 应用service资源
kubectl apply -f myapp-svc.yaml
# 查看svc
kubectl get -f myapp-svc.yaml
# 查看svc详情
kubectl describe -f myapp-svc.yaml
```

3. 访问service

```
# 宿主机访问ServiceIP:PORT
curl http://IP:8080/ping
curl http://IP:443/ping

# 容器里面通过DNS访问进入交互模式
kubectl exec -it curl -- sh
# 通过servicename.default访问
curl http://myapp-svc.default:8080/ping
# 通过完整的服务域名访问
curl http://myapp-svc.default.svc.cluster.local:8080/ping


# 访问redis容器
# 进入交互
kubectl exec -it redis -- bash
# 通过集群IP访问
redis-cli -h <clusterIP> -p 6379
# 通过域名访问
redis-cli -h myapp-svc.default -p 6379
redis-cli -h myapp-svc.default.svc.cluster.local -p 6379
```

NodePort

1. 定义

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-nodeport-svc # service 名称
spec:
  type: NodePort # service发布类型
  selector: # 标签选择器
    app: myapp
  ports:    # 指定端口数组可以指定多个
    - protocol: TCP # 协议
      name: http # 为端口指定一个端口名称
      port: 8080 # service 端口
      targetPort: 80 # 容器端口
      nodePort: 31490 # 节点端口，取值范围（30000-32767）
    - protocol: TCP
      name: https
      port: 443
      targetPort: 80
      nodePort: 30108
    - protocol: TCP
      name: redis-tcp
      port: 6379
      targetPort: 6379
      nodePort: 31542
```

2. 应用及查看service

```
# 应用service资源
kubectl apply -f myapp-nodeport-svc.yaml
# 查看svc
kubectl get -f myapp-nodeport-svc.yaml
# 查看svc详情
kubectl describe -f myapp-nodeport-svc.yaml
```

3. 访问service

```
# 通过service ip:port 可在集群内部node节点访问服务，可在集群内部其他容器内访问服务
# 通过DNS域名，可在集群内部其他容器访问服务
# 通过集群内任意节点IP:nodePort 可在集群内部容器、节点以及集群外部访问
curl http://<clusterIP>:8080/ping
curl http://<nodeIP1>:31490/ping
curl http://<nodeIP2>:31490/ping

kubectl exec -it curl -- sh
curl http://<clusterIP>:8080/ping
curl http://myapp-nodport-svc.default.svc.cluster.local:8080
curl http://<nodeIP1>:31490/ping
curl http://<nodeIP2>:31490/ping
```

ExternalName

1. 定义

```
apiVersion: v1
kind: Service
metadata:
  name: externalname-svc
spec:
  type: ExternalName
  externalName: www.0voice.com
```

2. 应用及查看service

```
 # 应用service资源
 kubectl apply -f externalname-svc.yaml
 # 查看svc
 kubectl get -f externalname-svc.yaml
 # 查看svc详情
 kubectl describe -f externalname-svc.yaml
```

3. 访问

```
# 进入集群curl pod
kubectl exec -it curl -- sh
# 当主机查找externalname-svc.default.svc.cluster.local时，集群 DNS 服务返回 CNAME 记录，
# 其值为 www.0voice.com
ping externalname-svc.default.svc.cluster.local
curl -k  https://externalname-svc.default.svc.cluster.local
```

注意：ExternalName类型，使用 CNAME 重定向。无法实现端口映射。相当于将集群内的服务域名映射到外部域名，与协议和端口号无关。

LoadBalancer

在使用支持外部负载均衡器的云提供商的服务时，设置 type 的值为 "LoadBalancer"， 将为 Service 提供负载均衡器，即云提供商的服务会自动的为type值为“LoadBalancer”的service创建负载均衡。负载均衡是异步创建的，部分云提供商支持在service定义的时候指定负载均衡IP地址（loadBalancerIP）。
使用云厂商提供的外部负载均衡器，来自外部负载均衡器的流量将直接重定向到后端 Pod 上。不再需要内部负载均衡。之所以通过service来定义，是因为可以通过service找到目标pod。

1. 定义

```
apiVersion: v1
kind: Service
metadata:
  name: my-loadbalancer-svc
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
  type: LoadBalancer
```

```
nick@k8s-master1:~/work$ kubectl get -f my-loadbalancer-svc.yaml
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
my-loadbalancer-svc   LoadBalancer   10.96.127.240   192.168.1.241   8080:30938/TCP   15m
```

没有选择符的Service

没有选择符的service需要手动配置endpoints，但是endpoints的配置，可以配置集群外部的服务或者是集群内其他命名空间下的服务。

1. 准备素材

```
# 选择一台非集群节点的机器，192.168.239.149
docker run -d -p 80:80 --name myhello xlhmzch/hello:1.0.0

# 当前集群运行pod，并取得其IP地址：10.244.2.69
kubectl run myhello-noselector --image=xlhmzch/hello:1.0.0
```

2. 定义

```
apiVersion: v1
kind: Service
metadata:
  name: notselector-svc
spec:
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

由于此服务没有选择算符，因此不会自动创建相应的 Endpoint 对象。 可以通过手动添加 Endpoint 对象，将服务手动映射到运行该服务的网络地址和端口：

```
apiVersion: v1
kind: Endpoints
metadata:
  # 这里的 name 要与 Service 的名字相同
  name: notselector-svc
subsets:
  - addresses:
      - ip: 192.168.239.149
    ports:
```

```
    - port: 80
  - addresses:
    - ip: 10.244.2.69
    ports:
    - port: 80
```

3. 应用资源

```
# 创建并查看service
kubectl apply -f notselector-svc.yaml
kubectl describe -f notselector-svc.yaml

# 创建并查看endPoints
kubectl apply -f notselector-points.yaml
kubectl describe -f notselector-points.yaml
```

4. 创建endpoint前后，service 对比

```
nick@k8s-master1:~/work$ kubectl describe svc notselector-svc
Name:              notselector-svc
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          <none>
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.111.182.150
IPs:               10.111.182.150
Port:              <unset>  8080/TCP
TargetPort:        80/TCP
Endpoints:         <none>
Session Affinity:  None
Events:            <none>
nick@k8s-master1:~/work$ kubectl describe svc notselector-svc
Name:              notselector-svc
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          <none>
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.111.182.150
IPs:               10.111.182.150
Port:              <unset>  8080/TCP
TargetPort:        80/TCP
Endpoints:         192.168.239.149:80,10.244.2.69:80
Session Affinity:  None
Events:            <none>
```

5. 访问service

```
# 在nod节点上访问service
curl http://10.111.182.150:8080/ping
```

无头Service

无头 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。
无头service不能通过集群IP访问，因为没有集群IP
无头service 不能通过dns名称访问，因为没有负载均衡等操作

无头service 的DNS记录是返回服务下所有endpoint，正常service是返回ClusterIP

有选择符的service

1. 定义

```
apiversion: v1
kind: Service
metadata:
  name: myapp-headless-svc # service 名称
spec:
  clusterIP: None
  type: ClusterIP # service发布类型
  selector: # 标签选择器
    app: myapp
  ports:    # 指定端口数组可以指定多个
    - protocol: TCP # 协议
      name: http # 为端口指定一个端口名称
      port: 8080 # service 端口
      targetPort: 80 # 容器端口
```

2. 创建或应用

```
# 创建服务
kubectl apply -f myapp-headless-svc.yaml

# 查看服务详情
kubectl describe  -f myapp-headless-svc.yaml

# 查看DNS记录
# 进入curl  pod
kubectl exec -it curl -- sh
# 查询DNS 记录 serviceName.namespace
nslookup  myapp-headless-svc.default.svc.cluster.local
```

无选择符的Service

1. 准备素材

```
# 选择一台非集群节点的机器，192.168.239.149
docker run -d -p 80:80 --name myhello xlhmzch/hello:1.0.0

# 当前集群运行pod，并取得其IP地址：10.244.2.69
kubectl run myhello-noselector --image=xlhmzch/hello:1.0.0
```

2. 定义

```
apiVersion: v1
kind: Service
metadata:
  name: notselector-headless-svc
spec:
  clusterIP: None
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

由于此服务没有选择算符，因此不会自动创建相应的 Endpoint 对象。 可以通过手动添加 Endpoint 对象，将服务手动映射到运行该服务的网络地址和端口：

```
apiVersion: v1
kind: Endpoints
metadata:
  # 这里的 name 要与 Service 的名字相同
  name: notselector-headless-svc
subsets:
  - addresses:
      - ip: 192.168.239.149
    ports:
      - port: 80
  - addresses:
      - ip: 10.244.2.69
    ports:
      - port: 80
```

3. 创建资源

```
# 创建服务
kubectl apply -f notselector-headless-svc.yaml

# 查看服务详情
kubectl describe  -f notselector-headless-svc.yaml

# 创建节点
kubectl apply -f notselector-headless-points.yaml

# 查看DNS记录
# 进入curl  pod
kubectl exec -it curl -- sh
# 查询DNS 记录 serviceName.namespace
nslookup  notselector-headless-svc.default.svc.cluster.local
```

服务发现机制

环境变量

必须先有service，才能在pod新建时将service注册到pod的环境变量

```
# 查看 pod的环境变量
kubectl exec myapp-deploy-785b767b97-49vl8  -- printenv | grep SERVICE

# 或者可以直接通过docker inspect 查看容器详情
```

DNS

```
# 进入curl  pod
kubectl exec -it curl -- sh
# 查询DNS 记录 serviceName.namespace
nslookup myapp-svc.default
```

service TLS

1. 自签证书

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /tmp/tls.key -out
/tmp/tls.crt -subj "/CN=myapp-svc.default.svc.cluster.local/O=my-hello"
```

2. 准备secret

```
# 创建secret
kubectl create secret tls myhellotls --cert=/tmp/tls.crt --key=/tmp/tls.key
# 查看secret
kubectl get secrets myhellotls -o yaml
```

3. 部署应用程序

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy
  namespace: default
  labels:
    name: myapp-deploy
spec:
  replicas: 5
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      volumes:
      - name: myhello-tls-volume
        secret:
          secretName: myhellotls
      restartPolicy: Always
      containers:
      - name: myhello
```

```
        image: xlhmzch/hello:curl
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        command: ["./app"]
        args: ["--param1=k8s-p1", "--param2=k8s-p2"]
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
        lifecycle:
          postStart:
            exec:
              command: ["/bin/sh", "-c","echo post start command exec >>
/tmp/data"]
          preStop:
            exec:
              command: ["/bin/sh", "-c","echo pre stop command exec >>
/tmp/data"]
        startupProbe:
          exec:
            command: ["/bin/sh", "-c","statuscode=`curl -o /dev/null -s -w %
{http_code} http://localhost/healthz`; [ $$statuscode -le 400 ] || exit 1"]
          # 指定初始化时间，即探针在容器启动之后的延迟时长
          initialDelaySeconds: 5
          # 探测周期，即每隔多长时间探测一次
          periodSeconds: 5
          # 最小连续失败次数，即连续失败多少次表示探测失败
          failureThreshold: 3
          # 最小连续成功次数，即连续探测成功多少次表示探测成功，liveness 和 startup 必须为1
          successThreshold: 1
          # 探测超时时间
          timeoutSeconds: 1
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
```

```
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
        volumeMounts:
        - mountPath: /app/cert
          name: myhello-tls-volume
      - name: myredis                    #容器的名称
        image: redis       #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 6379
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        livenessProbe:
          tcpSocket:
            port: 6379
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
```

```
kubectl apply -f myapp-deployment.yaml
```

4. service 部署

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc # service 名称
spec:
  type: ClusterIP # service发布类型
  selector:     # 标签选择器
    app: myapp
  ports:        # 指定端口数组可以指定多个
    - protocol: TCP # 协议
      name: http # 为端口指定一个端口名称
      port: 8080 # service 端口
      targetPort: 80 # 容器端口
    - protocol: TCP
      name: https
      port: 443
      targetPort: 443
    - protocol: TCP
      name: redis-tcp
      port: 6379
      targetPort: 6379
```

```
kubect  apply -f myapp-svc.yaml
```

5. http与https访问

```
# http 访问
curl  https://10.106.215.119:8080/ping
# https 访问
curl -k  https://10.106.215.119/ping
```