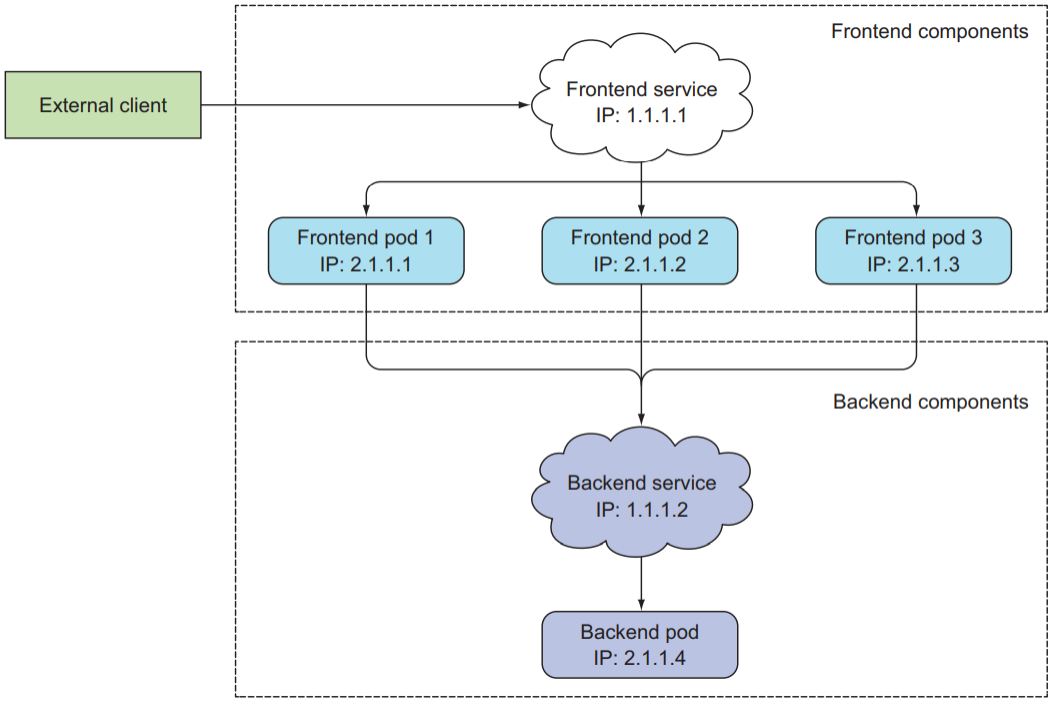
24.Service

概念

Kubernetes Service是一种资源对象，我们可以利用它对一组提供相同服务的pod创建一个统一的、固定的访问入口。每个service在存续期间都有一个不会改变的IP地址和端口。客户端可以与这个IP和端口建立连接，然后这些连接会被路由到这个service后面的其中一个pod上。通过这种方式，service的客户端就不需要知道提供该服务的每个pod的地址，这样这些pod在集群中就可以随时被创建或者移除。
假定我们现在有一个前端的web服务器和一个后端的数据库服务器。可以有多个pod充当前端服务器，但是只能有一个pod充当后端数据库。要想这样的一个系统正常运转，我们需要解决如下两个问题：

- 外部客户端需要连接到前端web服务器的pod，不需要关心web服务器的数量
- 前端web服务器pod需要连接到后端数据库。由于数据库运行在一个pod中，随着时间的推移，它可能在集群中不断地变换所在的节点，导致IP地址不断变化。我们肯定不希望数据库服务器被移动后就重新配置前端pod。

通过为前端pod创建一个service以及将其配置成从集群外部可访问，我们可以对外暴露一个单一的、不变的IP地址，外部客户端可以通过这个IP地址连接到pod。同样地，通过为后端pod创建一个service，我们可以为后端pod创建一个固定的地址。即使pod的IP地址变了，service的地址也不会变化。除此之外，通过创建service，前端pod能够轻易地通过环境变量或者DNS根据名称找到后端服务。如下图显示了两个service、支持这些service的两组pod以及它们之间的相互依赖关系：

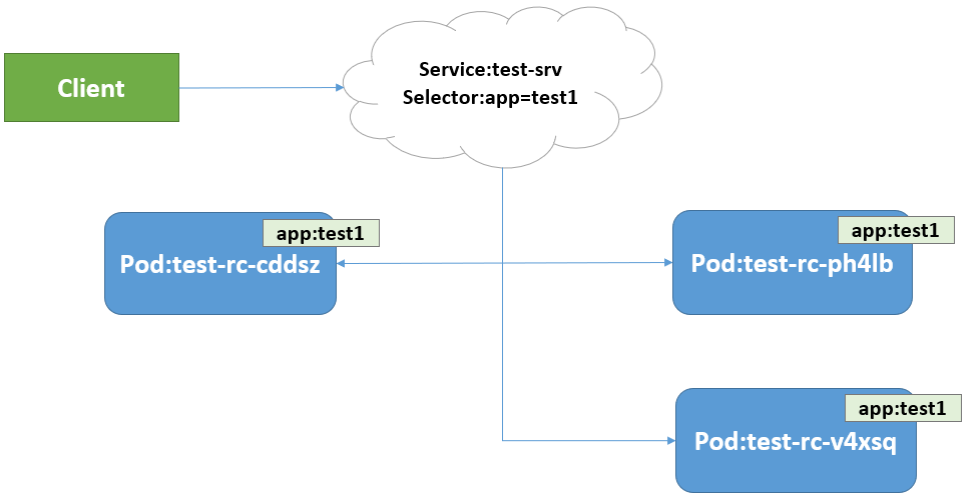


上图中，内部和外部客户端通过service连接到pod。

创建service

服务的后端可以不止一个pod。连接到该服务的请求可以通过负载均衡的方式分发到服务后面的pod上。但是要如何准确地定义哪些pod是服务的一部分哪些不是呢?

在之前的章节中我们学习过标签选择器，知道怎样在ReplicationController以及其他pod控制器中指定哪些pod属于相同的集合。service也使用与此相同的机制，如下图:



标签选择器用于决定哪些pod属于相同的服务。

在之前的章节中，我们创建了一个ReplicationController，它会运行包含Node.js应用的三个pod实例。本节中我们再次创建这个ReplicationController，然后验证是否有三个pod实例启动并运行。在此之后，我们再为这三个pod创建一个service。

```
[david@dhr-demo ~]$ kubectl create -f test-rc.yaml
replicationcontroller/test-rc created
[david@dhr-demo ~]$ kubectl get pod
NAME            READY   STATUS              RESTARTS   AGE
test-rc-8n2b8   0/1     ContainerCreating   0          4s
test-rc-bxlvx   0/1     ContainerCreating   0          4s
test-rc-xmll8   0/1     ContainerCreating   0          4s
[david@dhr-demo ~]$ kubectl get pod
NAME            READY   STATUS    RESTARTS   AGE
test-rc-8n2b8   1/1     Running   0          6s
test-rc-bxlvx   1/1     Running   0          6s
test-rc-xmll8   1/1     Running   0          6s
```

我们通过YAML文件来手动创建一个service。

vim test-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: test-svc
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: test1
~
```

apiVersion: v1

kind: Service

metadata:

  name: test-svc

spec:

  ports:

  - port: 80

    targetPort: 8080

  selector:

    app: test1

上面的文件定义了一个名为test-svc的service，它会在端口80上接收连接，并将每个连接路由到匹配标签选择器app=test1的所有pod中的某个pod的8080端口。

然后执行：

kubectl create -f test-svc.yaml

```
[david@dhr-demo ~]$ kubectl create -f test-svc.yaml
service/test-svc created
[david@dhr-demo ~]$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   32d
test-svc     ClusterIP   10.106.143.212   <none>        80/TCP    17s
[david@dhr-demo ~]$
```

从上图可以看到分配给test-svc服务的IP地址是10.106.143.212。由于这是一个集群IP，所以只能在集群内部访问。服务的主要目的是将一组pod暴露给集群中的其他pod，但是我们通常也希望将服务暴露给外部，这点在后面会讲到。

从集群内部访问服务

现在我们先从集群内部来使用刚刚创建的服务并了解服务的功能。

可以通过如下几种方式从集群内部向服务发送请求：

- 创建一个pod，它将请求发送到服务的集群IP并记录响应。然后可以通过pod的日志来查看服务的具体响应信息。
- 可以通过ssh远程登录到其中一个Kubernetes节点上，然后使用curl命令。
- 可以在某个已经存在的pod中通过kubectl exce命令来执行curl命令。

我们采用第三种方式并演示如何在已有的pod中运行命令。

kubectl exec命令使我们能够在已有pod的容器中远程运行任意命令。这非常有助于我们查看容器的内容、状态以及环境。

从现有的pod中选取某一个pod:

```
[david@dhr-demo ~]$ kubectl get pod
NAME            READY   STATUS    RESTARTS   AGE
test-rc-8n2b8   1/1     Running   0          23h
test-rc-bxlvx   1/1     Running   0          23h
test-rc-xmll8   1/1     Running   0          23h
[david@dhr-demo ~]$
```

kubectl exec test-rc-8n2b8 -- curl -s http://10.106.143.212

```
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.106.143.212
已发送消息至： test-rc-xmll8
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.106.143.212
已发送消息至： test-rc-bxlvx
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.106.143.212
已发送消息至： test-rc-xmll8
[david@dhr-demo ~]$
```

> 上面命令中的（--）代表kubectl命令选项的结束。双破折号之后的所有命令都应该在pod中执行。如果命令中没有使用单破折号（-）指定参数，那么就可以不用指定双破折号，但是在上面的例子中，如果我们不使用折号，-s 选项就会被解释为kubectl exec命令的选项：
> kubectl exec test-rc-8n2b8 curl -s http://10.106.143.212
> kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version.
> Use kubectl exec [POD] -- [COMMAND] instead.

在上面的例子中，我们以独立进程的方式执行curl命令，但是是在pod的主容器中。这与容器中真正的主进程和服务通信并没有什么区别。

配置会话绑定（Session Affinity）

在上面的实验中，我们执行了多次相同的命令，可以看到每次调用后消息发送给了不同的pod，这是因为服务代理通常将每个连接转发给一个随机选择的后端pod。

如果希望某个客户端每次发送的请求都被转发到同一个pod，可以将服务的sessionAffinity属性设置成ClientIP（默认None）：

vim test-svc2.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: test-svc1
spec:
  sessionAffinity: ClientIP
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: test1
```

apiVersion: v1
kind: Service
metadata:
  name: test-svc1
spec:
  sessionAffinity: ClientIP
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: test1

kubectl create -f test-svc2.yaml

```
[david@dhr-demo ~]$ kubectl create -f test-svc2.yaml
service/test-svc1 created
[david@dhr-demo ~]$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   33d
test-svc     ClusterIP   10.106.143.212   <none>        80/TCP    15h
test-svc1    ClusterIP   10.107.168.13    <none>        80/TCP    5s
[david@dhr-demo ~]$
```

基于该YAML文件创建的服务代理会将来自于同一个客户端IP的所有请求重定向到同一个pod。

kubectl exec test-rc-8n2b8 -- curl -s http://10.107.168.13

```
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.107.168.13
已发送消息至:  test-rc-bxlvx
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.107.168.13
已发送消息至:  test-rc-bxlvx
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.107.168.13
已发送消息至:  test-rc-bxlvx
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.107.168.13
已发送消息至:  test-rc-bxlvx
[david@dhr-demo ~]$ kubectl exec test-rc-8n2b8 -- curl -s http://10.107.168.13
已发送消息至:  test-rc-bxlvx
[david@dhr-demo ~]$
```

通过上图可以看到，反复调用5次，每次请求都指向同一个pod。

Kubernetes仅支持两种类型的服务会话绑定：None和ClientIP。可能你会惊讶它居然没有基于cookie的会话绑定选项，不过我们需要知道Kubernetes服务并不是在HTTP层面上工作。服务处理TCP和UDP包，并不关心它的载荷内容。由于cookie是HTTP协议中的一种构造，服务并不知道它们，这就解释了为什么不能基于cookie进行会话绑定。

一个服务暴露多个端口

Kubernetes中service支持多个端口。例如，如果pod监听两个端口：比如HTTP监听8080端口，HTTPS监听8443端口，我们可以使用一个服务将80和443端口分别转发到pod的8080和8443端口，完全没必要创建两个不同的服务。通过一个集群IP，使用一个多端口的服务就能将服务的所有端口全部暴露出来。

需要注意的是，当创建多端口服务时，必须为每个端口指定一个名字。

多端口服务的YAML定义如下：

```
apiVersion: v1
kind: Service
metadata:
  name: test–svc
spec:
  ports:
  – name: http
    port: 80
    targetPort: 8080
  – name: https
    port: 443
    targetPort: 8443
  selector:
    app: test1
```

使用命名端口

在之前的例子中我们都是通过端口号来引用端口本身的，还可以为每个pod的端口取一个名字并在service的spec区段中通过名称来引用端口。这对于不常用的端口号来说就比较容易区分。

例如，假设我们为pod的每个端口定义了一个名称，如下所示：

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: test1
    ports:
    - name: http
      containersPort: 8080
    - name: https
      containersPort: 8443
```

然后在服务的定义文件中就可以通过名称引用端口：

```
apiVersion: v1
kind: Service
spec:
 ports:
 - name: http
   port: 80
   targetPort: http
 - name: https
   port: 443
   targetPort: https
```

使用命名端口最大的好处就是即使更改了pod中的端口号也不用修改服务的spec。如果pod当前将8080用于http，但是如果后面决定将端口号换成80呢?

如果使用了命名端口，我们所需要做的就是在pod的spec中更改端口号（同时保持端口名不变）。当启动新端口的pod时，客户连接就会被转发到相应的端口号上，这取决于pod收到的连接（8080端口在旧pod上，80端口在新pod上）。