k8s基础应用

查看帮助

```
[root@master ~]# kubectl -h
```

类型	命令	描述
基础命令	create	通过文件名或标准输入创建资源
	expose	将一个资源公开为一个新的Service
	run	在集群中运行一个特定的镜像
	set	在对象上设置特定的功能
	get	显示一个或多个资源
	explain	文档参考资料
	edit	使用默认的编辑器编辑一个资源。
	delete	通过文件名、标准输入、资源名称或标签选择器来删除资源。
部署命令	rollout	管理资源的发布
	rolling-update	对给定的复制控制器滚动更新
	scale	扩容或缩容Pod数量，Deployment、ReplicaSet、RC或Job
	autoscale	创建一个自动选择扩容或缩容并设置Pod数量
集群管理命令	certificate	修改证书资源
	cluster-info	显示集群信息
	top	显示资源（CPU/Memory/Storage）使用。需要Heapster运行
	cordon	标记节点不可调度
	uncordon	标记节点可调度
	drain	驱逐节点上的应用，准备下线维护
	taint	修改节点taint标记

类型	命令	描述
故障诊断和调试命令	describe	显示特定资源或资源组的详细信息
	logs	在一个Pod中打印一个容器日志。如果Pod只有一个容器，容器名称是可选的
	attach	附加到一个运行的容器
	exec	执行命令到容器
	port-forward	转发一个或多个本地端口到一个pod
	proxy	运行一个proxy到Kubernetes API server
	cp	拷贝文件或目录到容器中
	auth	检查授权
高级命令	apply	通过文件名或标准输入对资源应用配置
	patch	使用补丁修改、更新资源的字段
	replace	通过文件名或标准输入替换一个资源
	convert	不同的API版本之间转换配置文件
设置命令	label	更新资源上的标签
	annotate	更新资源上的注释
	completion	用于实现kubectl工具自动补全
其他命令	api-versions	打印受支持的API版本
	config	修改kubeconfig文件（用于访问API，比如配置认证信息）
	help	所有命令帮助
	plugin	运行一个命令行插件
	version	打印客户端和服务版本信息

一、集群与节点信息

查看集群信息

```
[root@master ~]# kubectl cluster-info
Kubernetes master is running at
https://192.168.122.11:6443
KubeDNS is running at
https://192.168.122.11:6443/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use
'kubectl cluster-info dump'.
```

查看节点信息

```
[root@master ~]# kubectl get nodes
NAME     STATUS   ROLES    AGE    VERSION
master   Ready    master   118m   v1.15.1
node1    Ready    <none>   33m    v1.15.1
node2    Ready    <none>   31m    v1.15.1
```

查看节点详细信息

```
[root@master ~]# kubectl get nodes -o wide
NAME     STATUS   ROLES    AGE    VERSION   INTERNAL-IP
   EXTERNAL-IP   OS-IMAGE                KERNEL-VERSION
       CONTAINER-RUNTIME
master   Ready    master   119m   v1.15.1   192.168.122.11
  <none>        CentOS Linux 7 (Core)   3.10.0-
957.el7.x86_64   docker://18.9.8
node1    Ready    <none>   34m    v1.15.1   192.168.122.12
  <none>        CentOS Linux 7 (Core)   3.10.0-
957.el7.x86_64   docker://18.9.8
node2    Ready    <none>   32m    v1.15.1   192.168.122.13
  <none>        CentOS Linux 7 (Core)   3.10.0-
957.el7.x86_64   docker://18.9.8
```

描述节点详细信息

```
[root@master ~]# # kubectl describe node master
Name:               master
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=master
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=
Annotations:        flannel.alpha.coreos.com/backend-data:
{"VtepMAC":"fe:14:f4:0f:9b:55"}
                    flannel.alpha.coreos.com/backend-type:
vxlan
                    flannel.alpha.coreos.com/kube-subnet-
manager: true
```

```
                    flannel.alpha.coreos.com/public-ip:
192.168.122.11
                    kubeadm.alpha.kubernetes.io/cri-
socket: /var/run/dockershim.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-
managed-attach-detach: true
CreationTimestamp:  Fri, 15 Nov 2019 17:21:19 +0800
Taints:             node-
role.kubernetes.io/master:NoSchedule
Unschedulable:      false
Conditions:
  Type             Status  LastHeartbeatTime
  LastTransitionTime                Reason
      Message
  ----             ------  -----------------
  ------------------                ------
      -------
  MemoryPressure   False   Sun, 17 Nov 2019 09:27:31 +0800
  Fri, 15 Nov 2019 17:21:14 +0800
KubeletHasSufficientMemory   kubelet has sufficient memory
available
  DiskPressure     False   Sun, 17 Nov 2019 09:27:31 +0800
  Fri, 15 Nov 2019 17:21:14 +0800
KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Sun, 17 Nov 2019 09:27:31 +0800
  Fri, 15 Nov 2019 17:21:14 +0800
KubeletHasSufficientPID      kubelet has sufficient PID
available
  Ready            True    Sun, 17 Nov 2019 09:27:31 +0800
  Fri, 15 Nov 2019 17:30:39 +0800   KubeletReady
      kubelet is posting ready status
Addresses:
  InternalIP:  192.168.122.11
  Hostname:    master
Capacity:
 cpu:                4
 ephemeral-storage:  49999000Ki
 hugepages-2Mi:      0
 memory:             4045060Ki
 pods:               110
```

```
Allocatable:
 cpu:                4
 ephemeral-storage:  46079078324
 hugepages-2Mi:      0
 memory:             3942660Ki
 pods:               110
System Info:
 Machine ID:
a1ca5c651b2c4e8d947972ca21c0c5a0
 System UUID:                2FD28984-5F6B-4ED0-98F9-
749B196DE12F
 Boot ID:                    67b912ce-bb57-493f-9021-
a0376fb26cc0
 Kernel Version:             3.10.0-957.el7.x86_64
 OS Image:                   CentOS Linux 7 (Core)
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://18.9.8
 Kubelet Version:            v1.15.1
 Kube-Proxy Version:         v1.15.1
PodCIDR:                     10.3.0.0/24
Non-terminated Pods:         (8 in total)
  Namespace                  Name
     CPU Requests  CPU Limits  Memory Requests  Memory
Limits  AGE
  ---------                  ----
     ------------  ----------  ---------------  ----------
---  ---
  kube-system                coredns-bccdc95cf-c8fzd
    100m (2%)     0 (0%)      70Mi (1%)        170Mi (4%)
    40h
  kube-system                coredns-bccdc95cf-t4h2x
    100m (2%)     0 (0%)      70Mi (1%)        170Mi (4%)
    40h
  kube-system                etcd-master
    0 (0%)        0 (0%)      0 (0%)           0 (0%)
    40h
  kube-system                kube-apiserver-master
    250m (6%)     0 (0%)      0 (0%)           0 (0%)
    40h
```

```
  kube-system                kube-controller-manager-
master    200m (5%)     0 (0%)      0 (0%)           0
(0%)         40h
  kube-system                kube-flannel-ds-amd64-dlp8f
    100m (2%)     100m (2%)    50Mi (1%)        50Mi (1%)
   39h
  kube-system                kube-proxy-7bhzh
     0 (0%)        0 (0%)      0 (0%)           0 (0%)
    40h
  kube-system                kube-scheduler-master
    100m (2%)     0 (0%)      0 (0%)           0 (0%)
    40h
Allocated resources:
  (Total limits may be over 100 percent, i.e.,
overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                850m (21%)  100m (2%)
  memory             190Mi (4%)  390Mi (10%)
  ephemeral-storage  0 (0%)      0 (0%)
Events:
  Type    Reason                   Age                From
               Message
  ----    ------                   ----               ----
               -------
  Normal  Starting                 20m
 kubelet, master     Starting kubelet.
  Normal  NodeAllocatableEnforced  20m
 kubelet, master     Updated Node Allocatable limit across
pods
  Normal  NodeHasSufficientMemory  20m (x8 over 20m)
 kubelet, master     Node master status is now:
NodeHasSufficientMemory
  Normal  NodeHasNoDiskPressure    20m (x8 over 20m)
 kubelet, master     Node master status is now:
NodeHasNoDiskPressure
  Normal  NodeHasSufficientPID     20m (x7 over 20m)
 kubelet, master     Node master status is now:
NodeHasSufficientPID
  Normal  Starting                 19m
 kube-proxy, master  Starting kube-proxy.
```

node节点管理集群

在node节点上管理时会报如下错误

```
[root@node1 ~]# kubectl get nodes
The connection to the server localhost:8080 was refused -
did you specify the right host or port?
```

只要把master上的管理文件`/etc/kubernetes/admin.conf`拷贝到node节点的`$HOME/.kube/config`就可以让node节点也可以实现kubectl命令管理

1, 在node节点的用户家目录创建`.kube`目录

```
[root@node1 ~]# mkdir /root/.kube
```

2, 在master节点做如下操作

```
[root@master ~]# scp /etc/kubernetes/admin.conf
node1:/root/.kube/config
```

3, 在node节点验证

```
[root@node1 ~]# kubectl get nodes
NAME     STATUS   ROLES    AGE     VERSION
master   Ready    master   2d23h   v1.15.1
node1    Ready    node     2d22h   v1.15.1
node2    Ready    node     2d22h   v1.15.1
```

节点标签

查看节点标签信息

```
[root@master ~]# kubectl get node --show-labels
NAME     STATUS   ROLES    AGE    VERSION   LABELS
master   Ready    master   121m   v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=master,kub
ernetes.io/os=linux,node-role.kubernetes.io/master=
node1    Ready    <none>   36m    v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=node1,kube
rnetes.io/os=linux
node2    Ready    <none>   34m    v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=node2,kube
rnetes.io/os=linux
```

设置节点标签信息

为node2和node3加上role的标签信息

```
[root@master ~]# kubectl label node node1 node-
role.kubernetes.io/node=

[root@master ~]# kubectl label node node2 node-
role.kubernetes.io/node=
```

查看验证

```
[root@master ~]# kubectl get nodes
NAME     STATUS   ROLES    AGE    VERSION
master   Ready    master   150m   v1.15.1
node1    Ready    node     50m    v1.15.1
node2    Ready    node     41m    v1.15.1
```

```
[root@master ~]# kubectl get nodes --show-labels
NAME     STATUS   ROLES    AGE    VERSION   LABELS
master   Ready    master   128m   v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=master,kub
ernetes.io/os=linux,node-role.kubernetes.io/master=
node1    Ready    node     43m    v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=node1,kube
rnetes.io/os=linux,node-role.kubernetes.io/node=
node2    Ready    node     41m    v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=node2,kube
rnetes.io/os=linux,node-role.kubernetes.io/node=
```

多维度标签

也可以加其它的多维度标签,用于不同的需要区分的场景

如把node2标签为华南区,A机房,测试环境,游戏业务

```
[root@master ~]# kubectl label node node2 region=huanai
zone=A env=test bussiness=game
```

```
[root@master ~]# kubectl get nodes node2 --show-labels
NAME    STATUS   ROLES    AGE   VERSION   LABELS
node2   Ready    node     42m   v1.15.1
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
bussiness=game,env=test,kubernetes.io/arch=amd64,kubernete
s.io/hostname=node2,kubernetes.io/os=linux,node-
role.kubernetes.io/node=,region=huanai,zone=A
```

显示节点的相应用标签

```
[root@master ~]# kubectl get nodes -L region,zone
NAME     STATUS   ROLES    AGE   VERSION   REGION   ZONE
master   Ready    master   18h   v1.15.1
node1    Ready    node     16h   v1.15.1
node2    Ready    node     16h   v1.15.1   huanai   A
```

查找`region=huanai`的节点

```
[root@master ~]# kubectl get nodes -l region=huanai
NAME    STATUS   ROLES   AGE   VERSION
node2   Ready    node    16h   v1.15.1
```

标签的修改

```
[root@master ~]# kubectl label node node2 bussiness=ad --
overwrite=true
node/node1 labeled

加上--overwrite=true覆盖原标签的value进行修改操作
```

取消标签信息

使用key加一个减号的写法来取消标签

```
[root@master ~]# kubectl label node node2 region- zone-
env- bussiness-
node/node2 labeled
```

```
[root@master ~]# kubectl get nodes --show-labels |grep
node2
node2   Ready    node     16d   v1.15.0
beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,
kubernetes.io/arch=amd64,kubernetes.io/hostname=node2,kube
rnetes.io/os=linux,node-role.kubernetes.io/node=
```

标签选择器

打完标签后的node,pod,deployment等资源都可以在标签选择器里进行匹配

标签选择器主要有2类:

- 等值关系: =, !=
- 集合关系: KEY in {VALUE1, VALUE2.....}

```
[root@master ~]# kubectl get node -l "bussiness in
(game,ad)"
```

总之:标签是为了更好的进行资源对象的相关选择与匹配

二、namespace

Namespace是对一组资源和对象的抽象集合.

Namespace常用来隔离不同的用户,如Kubernetes自带的服务一般运行在kube-system namespace中.

对于同一种资源的不同版本,就可以直接使用label来划分即可,不需要使用namespace来区分

> 常见的 pod, service, replication controller 和 deployment 等都是属于某一个 namespace 的（默认是 default）
>
> 而 nodes, persistent volume，namespace 等资源则不属于任何 namespace。

查看namespace信息

```
[root@master ~]# kubectl get namespaces       # namespaces
可以简写为namespace或ns
NAME              STATUS   AGE
default           Active   130m          # 所有未指定
Namespace的对象都会被分配在default命名空间
kube-node-lease   Active   130m          # 节点资源
kube-public       Active   130m          # 此命名空间下的资源
可以被所有人访问
kube-system       Active   130m          # 所有由Kubernetes
系统创建的资源都处于这个命名空间
```

创建namespace

命令创建

```
[root@master ~]# kubectl create namespace namespace1
namespace/namespace1 created

[root@master ~]# kubectl get namespaces
NAME              STATUS   AGE
default           Active   131m
kube-node-lease   Active   131m
kube-public       Active   131m
kube-system       Active   131m
namespace1        Active   2s
```

可以通过`# kubectl edit namespace 命名空间名`来编辑或查看它的YAML语法

后面学的资源几乎都可以使用`kubectl edit 资源类型 资源名`编辑它的YAML语法或者使用`kubectl get 资源类型 资源名 -o yaml`来查看

YAML文件创建

```
[root@master ~]# vim create_namespace.yml
apiVersion: v1                              # api版本号
kind: Namespace                             # 类型为namespace
metadata:                                   # 定义namespace的元数据属性
  name: namespace2                          # 定义name属性为namespace2

[root@master ~]# kubectl apply -f create_namespace.yml
namespace/namespace2 created
```

```
[root@master ~]# kubectl get namespaces
NAME              STATUS   AGE
default           Active   132m
kube-node-lease   Active   133m
kube-public       Active   133m
kube-system       Active   133m
namespace1        Active   103s
namespace2        Active   17s
```

删除namespace

注意:

- 删除一个namespace会自动删除所有属于该namespace的资源(类似mysql> drop database xxx;会删除库里的所有表一样，请慎重操作)
- default,kube-system,kube-public命名空间不可删除

命令删除

```
[root@master ~]# kubectl delete namespace namespace1
namespace "namespace1" deleted
```

YAML文件删除

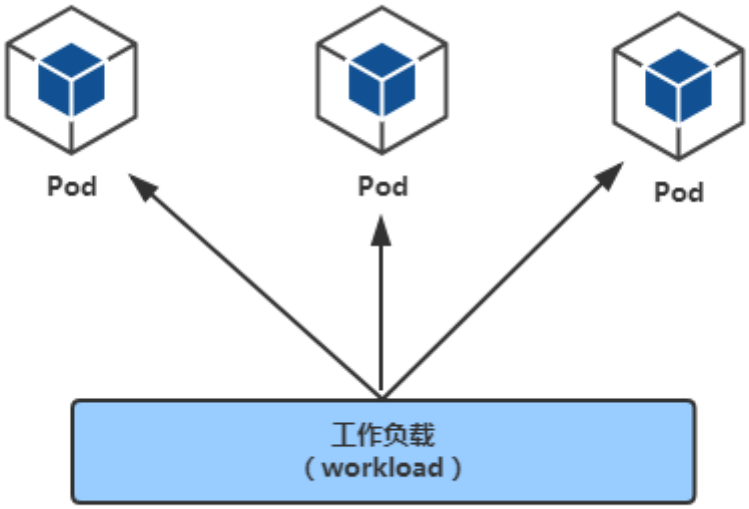
```
[root@master ~]# kubectl delete -f create_namespace.yml
namespace "namespace2" deleted
```

```
[root@master ~]# kubectl get namespaces
NAME              STATUS   AGE
default           Active   1d
kube-node-lease   Active   1d
kube-public       Active   1d
kube-system       Active   1d
```

三、工作负载(workloads)

workloads分为pod与controllers

- pod通过控制器实现应用的运行，如何伸缩，升级等
- controllers 在集群中管理pod
- pod与控制器之间通过label-selector相关联，是唯一的关联方式



比如在pod的YAML里指定

```
  labels:
    app: nginx
```

在控制器的YAML里指定

```
  labels:
    app: nginx
```

四、pod

pod简介

- Pod是Kubernetes最小的管理单位,一个Pod可以封装**一个容器或多个容器**
- 一个Pod里的多个容器可以共享存储和网络, 可以看作一个逻辑的主机

- 多个容器共享同一个network namespace，由此在一个Pod里的多个容器共享Pod的IP和端口namespace，所以一个Pod内的多个容器之间可以通过localhost来进行通信,所需要注意的是不同容器要注意不要有端口冲突即可。不同的Pod有不同的IP,不同Pod内的多个容器之前通信，不可以使用IPC（如果没有特殊指定的话）通信，通常情况下使用Pod的IP进行通信。
- 一个Pod里的多个容器可以共享存储卷，这个存储卷会被定义为Pod的一部分，并且可以挂载到该Pod里的所有容器的文件系统上。

pod分类

pod可分为:

- **无控制器管理的自主式pod** 没有副本控制器控制，删除自主式pod后不会重新创建
- **控制器管理的pod** 控制器会按照定义的策略控制pod的数量，发现pod数量少了，会立即自动建立出来新的pod；一旦发现pod多了，也会自动杀死多余的Pod。

pod的YAML格式

先看一个yaml格式的pod定义文件解释

```
# yaml格式的pod定义文件完整内容:
apiVersion: v1       #必选，api版本号，例如v1
kind: Pod            #必选，Pod
metadata:            #必选，元数据
  name: string       #必选，Pod名称
  namespace: string    #Pod所属的命名空间,默认在default的
namespace
  labels:            # 自定义标签
    - name: string     #自定义标签名字
  annotations:        #自定义注释列表
    - name: string
```

```
spec:          #必选，Pod中容器的详细定义(期望)
  containers:       #必选，Pod中容器列表
  - name: string      #必选，容器名称
    image: string     #必选，容器的镜像名称
    imagePullPolicy: [Always | Never | IfNotPresent] #获取镜像的策略 Alawys表示下载镜像 IfnotPresent表示优先使用本地镜像，否则下载镜像，Nerver表示仅使用本地镜像
    command: [string]     #容器的启动命令列表，如不指定，使用打包时使用的启动命令
    args: [string]      #容器的启动命令参数列表
    workingDir: string      #容器的工作目录
    volumeMounts:     #挂载到容器内部的存储卷配置
    - name: string      #引用pod定义的共享存储卷的名称，需用volumes[]部分定义的的卷名
      mountPath: string     #存储卷在容器内mount的绝对路径，应少于512字符
      readOnly: boolean     #是否为只读模式
    ports:        #需要暴露的端口库号列表
    - name: string      #端口号名称
      containerPort: int    #容器需要监听的端口号
      hostPort: int     #容器所在主机需要监听的端口号，默认与Container相同
      protocol: string      #端口协议，支持TCP和UDP，默认TCP
    env:        #容器运行前需设置的环境变量列表
    - name: string      #环境变量名称
      value: string     #环境变量的值
    resources:        #资源限制和请求的设置
      limits:       #资源限制的设置
        cpu: string     #Cpu的限制，单位为core数，将用于docker run --cpu-shares参数
        memory: string      #内存限制，单位可以为Mib/Gib，将用于docker run --memory参数
      requests:       #资源请求的设置
        cpu: string     #Cpu请求，容器启动的初始可用数量
        memory: string      #内存清求，容器启动的初始可用数量
    livenessProbe:      #对Pod内个容器健康检查的设置，当探测无响应几次后将自动重启该容器，检查方法有exec、httpGet和tcpSocket，对一个容器只需设置其中一种方法即可
      exec:       #对Pod容器内检查方式设置为exec方式
        command: [string]   #exec方式需要制定的命令或脚本
```

```
      httpGet:        #对Pod内个容器健康检查方法设置为HttpGet，需要制定Path、port
        path: string
        port: number
        host: string
        scheme: string
        HttpHeaders:
        - name: string
          value: string
      tcpSocket:      #对Pod内个容器健康检查方式设置为tcpSocket方式
         port: number
       initialDelaySeconds: 0  #容器启动完成后首次探测的时间，单位为秒
       timeoutSeconds: 0   #对容器健康检查探测等待响应的超时时间，单位秒，默认1秒
       periodSeconds: 0    #对容器监控检查的定期探测时间设置，单位秒，默认10秒一次
       successThreshold: 0
       failureThreshold: 0
       securityContext:
         privileged:false
    restartPolicy: [Always | Never | OnFailure] # Pod的重启策略，Always表示一旦不管以何种方式终止运行，kubelet都将重启，OnFailure表示只有Pod以非0退出码退出才重启，Nerver表示不再重启该Pod
    nodeSelector: obeject  # 设置NodeSelector表示将该Pod调度到包含这个label的node上，以key：value的格式指定
    imagePullSecrets:    #Pull镜像时使用的secret名称，以key：secretkey格式指定
    - name: string
    hostNetwork:false       #是否使用主机网络模式，默认为false，如果设置为true，表示使用宿主机网络
    volumes:        #在该pod上定义共享存储卷列表
    - name: string      #共享存储卷名称 （volumes类型有很多种）
      emptyDir: {}      #类型为emtyDir的存储卷，与Pod同生命周期的一个临时目录。为空值
      hostPath: string      #类型为hostPath的存储卷，表示挂载Pod所在宿主机的目录
        path: string      #Pod所在宿主机的目录，将被用于同期中mount的目录
```

```
      secret:       #类型为secret的存储卷，挂载集群与定义的
secret对象到容器内部
        scretname: string
        items:
        - key: string
          path: string
      configMap:     #类型为configMap的存储卷，挂载预定义的
configMap对象到容器内部
        name: string
        items:
        - key: string
          path: string
```

YAML格式查找帮助方法

查看api版本

```
[root@master ~]# kubectl api-versions
```

查看资源写法

```
[root@master ~]# kubectl explain namespace

[root@master ~]# kubectl explain pod
[root@master ~]# kubectl explain pod.spec
[root@master ~]# kubectl explain pod.spec.containers
```

创建自主式pod

1, 准备yaml文件

```
[root@master ~]# vim pod1.yaml
apiVersion: v1                      # api版本(不同版本语法有少量差异),这里为v1.
kind: Pod                           # 资源类型为Pod
metadata:
  name: memory-demo                 # 自定义pod的名称
spec:
  containers:                       # 定义pod里包含的容器
  - name: demo                      # 自定义pod中的容器名
    image: polinux/stress           # 启动容器的镜像名
    command: ["stress"]             # 自定义启动容器时要执行的命令(类似dockerfile里的CMD)
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"] # 自定义启动容器执行命令的参数

# polinux/stress这个镜像用于压力测试,在启动容器时传命令与参数就是相当于分配容器运行时需要的压力
```

说明: 镜像拉取策略　imagePullPolicy

- Always : 不管本地有没有镜像，都要从仓库中下载镜像
- Never : 从来不从仓库下载镜像, 只用本地镜像,本地没有就算了
- IfNotPresent: 如果本地存在就直接使用, 不存在才从仓库下载

默认的策略是：

- 当镜像标签版本是latest，默认策略就是Always
- 如果指定特定版本默认拉取策略就是IfNotPresent。

2, 通过yaml文件创建pod

```
[root@master ~]# kubectl apply -f pod1.yaml
pod/memory-demo created
```

查看pod信息

```
[root@master ~]# kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
memory-demo   1/1     Running   0          25s
```

查看pod详细信息

```
[root@master ~]# kubectl get pod -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP
  NODE    NOMINATED NODE   READINESS GATES
memory-demo   1/1     Running   0          10m   10.3.1.2
  node1   <none>           <none>
```

描述pod详细信息

```
[root@master ~]# kubectl describe pod memory-demo
```

pod的标签

- 为pod设置label,用于控制器通过label与pod关联

1. 查看pod的标签

```
[root@master ~]# kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE   LABELS
memory-demo   1/1     Running   0          30s   <none>
```

2. 打标签,再查看

```
[root@master ~]# kubectl label pod memory-demo
region=huanai zone=A env=test bussiness=game
pod/memory-demo labeled

[root@master ~]# kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE     LABELS
memory-demo   1/1     Running   0          2m29s
bussiness=game,env=test,region=huanai,zone=A
```

3. 通过等值关系标签查询

```
[root@master ~]# kubectl get pods -l zone=A
NAME          READY   STATUS    RESTARTS   AGE
memory-demo   1/1     Running   0          10m
```

4. 通过集合关系标签查询

```
[root@master ~]# kubectl get pods -l "zone in (A,B,C)"
NAME          READY   STATUS    RESTARTS   AGE
memory-demo   1/1     Running   0          11m
```

5. 删除标签后再验证

```
[root@master ~]# kubectl label pod memory-demo region-
zone- env- bussiness-
pod/memory-demo labeled

[root@master ~]# kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE   LABELS
memory-demo   1/1     Running   0          17m   <none>
```

小结:

- pod的label与node的label操作方式几乎相同
- node的label用于pod调度到指定label的node节点
- pod的label用于controller关联控制的pod

删除pod

```
[root@master ~]# kubectl delete pod memory-demo
pod "memory-demo" deleted
```

pod资源限制

准备2个不同限制方式的创建pod的yaml文件

```
[root@master ~]# vim pod2.yml
apiVersion: v1
```

```
kind: Namespace
metadata:
  name: namespace1
---
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: namespace1
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    command: ["stress"]                    # 启动容器时执行的命
令
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang",
"1"]  # 产生1个进程分配150M内存1秒后释放
```

```
[root@master ~]# vim pod3.yml
apiVersion: v1
kind: Namespace
metadata:
  name: namespace1
---
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
  namespace: namespace1
spec:
  containers:
  - name: memory-demo-ctr2
    image: polinux/stress
```

```
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "150Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang",
"1"]
```

```
[root@master ~]# kubectl apply -f pod2.yml
namespace/namespace1 created
pod/memory-demo created

[root@master ~]# kubectl apply -f pod3.yml
namespace/namespace1 unchanged
pod/memory-demo-2 created
```

```
[root@master ~]# kubectl get namespace  |grep namespace1
namespace1        Active   2m28s

[root@master ~]# kubectl get pod -n namespace1
NAME            READY   STATUS      RESTARTS   AGE
memory-demo     1/1     Running     0          3m37s
memory-demo-2   0/1     OOMKilled   5          3m13s
```

查看会发现memory-demo-2这个pod状态变为OOMKilled，因为它是内存不足所以显示Container被杀死

说明: 一旦pod中的容器挂了，我们就把容器重启. 策略包括如下：

- Always：表示容器挂了总是重启，这是默认策略
- OnFailures：表容器状态为错误时才重启，也就是容器正常终止时才重启
- Never：表示容器挂了不予重启

- 对于Always这种策略，容器只要挂了，就会立即重启，这样是很耗费资源的。所以Always重启策略是这么做的：第一次容器挂了立即重启，如果再挂了就要延时10s重启，第三次挂了就等20s重启..... 依次类推

测试完后删除

```
[root@master ~]# kubectl delete ns namespace1
```

一个pod包含多个容器

1, 准备yml文件

```
[root@master ~]# vim pod4.yml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
spec:
  containers:
  - name: memory-demo-ctr-1
    image: polinux/stress
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang",
"1"]

  - name: memory-demo-ctr-2
    image: polinux/stress
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        memory: "200Mi"
      requests:
```

```
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang",
"1"]
```

2, 应用yml文件创建pod

```
[root@master ~]# kubectl apply -f pod4.yml
```

3, 查看pod在哪个节点

```
[root@master ~]# kubectl get pods  -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP
    NODE    NOMINATED NODE   READINESS GATES
memory-demo   2/2     Running   0          4m32s
10.3.2.3   node2   <none>           <none>

可以看到有2个容器,运行在node2节点
```

4,在node2上验证,确实产生了2个容器

```
[root@node2 ~]# docker ps -a |grep stress
7f2ba28dc7bb        68478b32266c
              "stress --vm 1 --vm-…"   5 minutes ago
Up 5 minutes                                    k8s_memory-
demo-ctr-2_memory-demo_default_86c31332-d8df-40ee-b332-
f285ffe0a7df_0

9e45276b3e3a        68478b32266c
              "stress --vm 1 --vm-…"   5 minutes ago
Up 5 minutes                                    k8s_memory-
demo-ctr-1_memory-demo_default_86c31332-d8df-40ee-b332-
f285ffe0a7df_0
```

对pod里的容器进行操作

命令帮助

```
[root@master ~]# kubectl exec -h
```

不用交互直接执行命令

格式为: `kubectl exec pod名 -c 容器名 -- 命令`

注意:

- -c 容器名为可选项,如果是1个pod中1个容器,则不用指定;
- 如果是1个pod中多个容器,不指定默认为第1个。

```
[root@master ~]# kubectl exec  memory-demo -c memory-demo-ctr-1 -- touch /111
```

不指定容器名,则默认为pod里的第1个容器

```
[root@master ~]# kubectl exec  memory-demo  -- touch /222
```

和容器交互操作

和docker exec几乎一样

```
[root@master ~]# kubectl exec -it memory-demo -c memory-demo-ctr-1 /bin/bash
bash-4.3# touch /333
bash-4.3# ls
111    333    dev    home   media  proc   run    srv
 tmp    var
222    bin    etc    lib    mnt    root   sbin   sys
 usr
bash-4.3# exit
exit
```

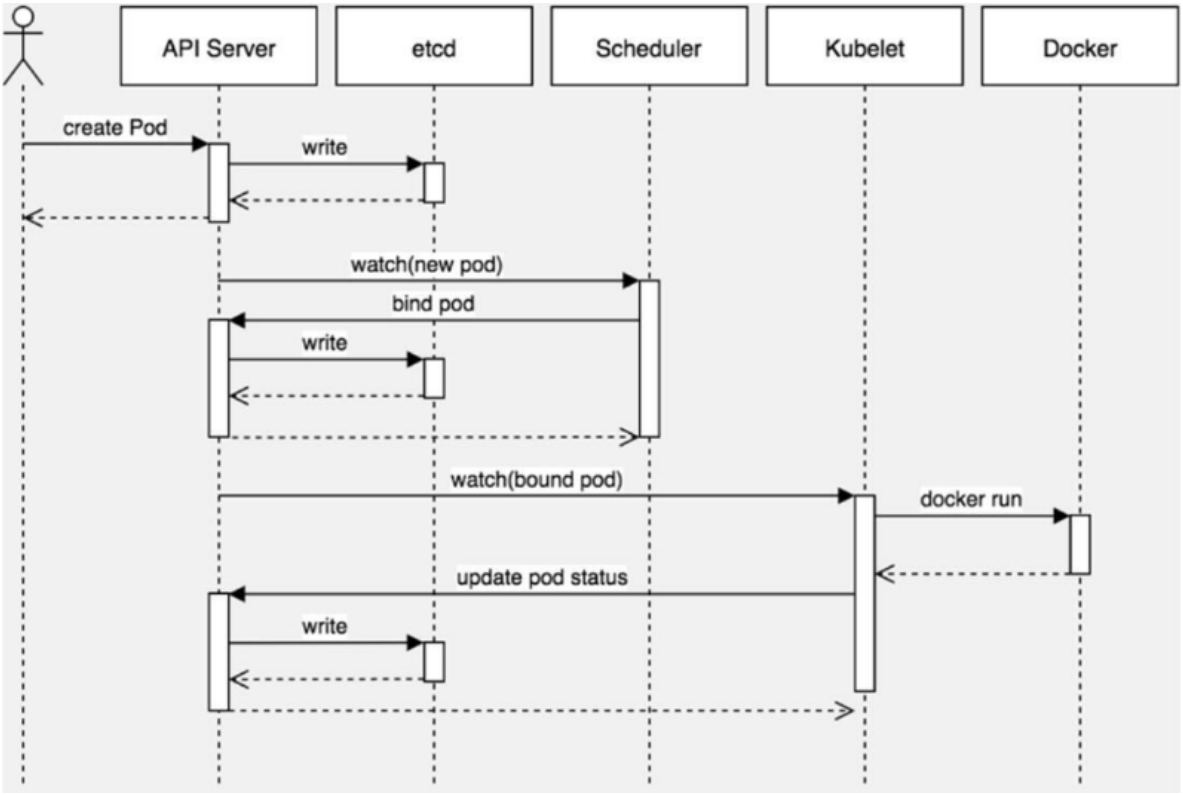
删除pod

```
[root@master ~]# kubectl delete pod memory-demo
pod "memory-demo" deleted
```

五、pod调度

kubernetes会通过算法将pod调度到node节点上运行

pod调度流程



调度约束方法

我们为了实现容器主机资源平衡使用, 可以使用约束把pod调度到指定的node节点

- nodeName 用于将pod调度到指定的node名称上
- nodeSelector 用于将pod调度到匹配Label的node上

案例1: nodeName

1, 编写YAML文件

```
[root@master ~]# vim pod-nodename.yml
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodename
spec:
  nodeName: node1                          # 通过nodeName调度到node1节点
  containers:
  - name: nginx
    image: nginx:1.15-alpine
```

2, 应用YAML文件创建pod

```
[root@master ~]# kubectl apply -f pod-nodename.yml
pod/pod-nodename created
```

3, 验证

```
[root@master ~]# kubectl describe pod pod-nodename |tail -6
Events:
  Type    Reason   Age   From             Message
  ----    ------   ----  ----             -------
  Normal  Pulled   40s   kubelet, node1  Container image "nginx:1.15" already present on machine
  Normal  Created  40s   kubelet, node1  Created container nginx
  Normal  Started  39s   kubelet, node1  Started container nginx

倒数第3行没有使用scheduler,而是直接给node1了,说明nodeName约束生效
```

案例2: nodeSelector

1, 为node2打标签

```
[root@master ~]# kubectl label nodes node2 bussiness=game
node/node2 not labeled
```

2, 编写YAML文件

```
[root@master ~]# vim pod-nodeselector.yml
apiVersion: v1
kind: Pod
metadata:
  name: pod-nodeselect
spec:
  nodeSelector:                          # nodeSelector节点选择器
    bussiness: game                      # 指定调度到标签为bussiness=game的节点
  containers:
  - name: nginx
    image: nginx:1.15-alpine
```

3, 应用YAML文件创建pod

```
[root@master ~]# kubectl apply -f pod-nodeselector.yml
pod/pod-nodeselect created
```

4, 验证

```
[root@master ~]# kubectl describe pod pod-nodeselect |tail -6
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  10s   default-scheduler  Successfully assigned default/pod-nodeselect to node2
  Normal  Pulled     8s    kubelet, node2     Container image "nginx:1.15" already present on machine
  Normal  Created    8s    kubelet, node2     Created container nginx
  Normal  Started    7s    kubelet, node2     Started container nginx

仍然经过了scheduler,但确实分配到了node2上
```

有兴趣可以再删除后再创建,重复几次验证

六、pod的生命周期

- 有些pod(比如跑httpd服务),正常情况下会一直运行中,但如果手动删除它,此pod会终止
- 也有些pod(比如执行计算任务)，任务计算完后就会自动终止

上面两种场景中,pod从创建到终止的过程就是pod的生命周期。

容器启动

1. pod中的容器在创建前,有初始化容器(init container)来进行初始化环境
2. 初化完后,主容器(main container)开始启动
3. 主容器启动后,有一个**post start**的操作(启动后的触发型操作,或者叫启动后钩子)
4. post start后,就开始做健康检查
    - 第一个健康检查叫存活状态检查(liveness probe)，用来检查主容器存活状态的
    - 第二个健康检查叫准备就绪检查(readyness probe)，用来检查主容器是否启动就绪

容器终止

1. 可以在容器终止前设置**pre stop**操作(终止前的触发型操作,或者叫终止前钩子)
2. 当出现特殊情况不能正常销毁pod时,大概等待30秒会强制终止
3. 终止容器后还可能会重启容器(视容器重启策略而定)。

回顾容器重启策略

- Always：表示容器挂了总是重启，这是默认策略
- OnFailures：表容器状态为错误时才重启，也就是容器正常终止时才重启
- Never：表示容器挂了不予重启
- 对于Always这种策略，容器只要挂了，就会立即重启，这样是很耗费资源的。所以Always重启策略是这么做的：第一次容器挂了立即重启，如果再挂了就要延时10s重启，第三次挂了就等20s重启..... 依次类推

HealthCheck健康检查

当Pod启动时，容器可能会因为某种错误(服务未启动或端口不正确)而无法访问等。

Health Check方式

kubelet拥有两个检测器，它们分别对应不同的触发器(根据触发器的结构执行进一步的动作)

方式	说明
Liveness Probe(存活状态探测)	检查后不健康，重启pod
readiness Probe(就绪型探测)	检查后不健康，将容器设置为Notready;如果使用service来访问,流量不会转发给此种状态的pod

Probe探测方式

方式	说明
Exec	执行命令
HTTPGet	http请求某一个URL路径
TCP	tcp连接某一个端口

案例1: liveness-exec

1, 准备YAML文件

```
[root@master ~]# vim pod-liveness-exec.yml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
  namespace: default
spec:
  containers:
  - name: liveness
    image: busybox
    imagePullPolicy: IfNotPresent
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy;
sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5                # pod启动延迟5
秒后探测
      periodSeconds: 5                      # 每5秒探测1次
```

2, 应用YAML文件

```
[root@master ~]# kubectl apply -f pod-liveness-exec.yml
```

3, 通过下面的命令观察

```
[root@master ~]# kubectl describe pod liveness-exec
......
Events:
  Type     Reason     Age   From               Message
  ----     ------     ----  ----               -------
  Normal   Scheduled  40s   default-scheduler
 Successfully assigned default/liveness-exec to node1
  Normal   Pulled     38s   kubelet, node1     Container
image "busybox" already present on machine
  Normal   Created    37s   kubelet, node1     Created
container liveness
  Normal   Started    37s   kubelet, node1     Started
container liveness
  Warning  Unhealthy  3s    kubelet, node1     Liveness
probe failed: cat: can't open '/tmp/healthy': No such file
or directory

看到40s前被调度以node1节点,3s前健康检查出问题
```

4, 过几分钟再观察

```
[root@master ~]# kubectl describe pod liveness-exec
......
Events:
  Type     Reason     Age                  From
    Message
  ----     ------     ----                 ----
    -------
  Normal   Scheduled  3m42s                default-
scheduler  Successfully assigned default/liveness-exec to
node1
  Normal   Pulled     70s (x3 over 3m40s)  kubelet, node1
    Container image "busybox" already present on machine
  Normal   Created    70s (x3 over 3m39s)  kubelet, node1
    Created container liveness
  Normal   Started    69s (x3 over 3m39s)  kubelet, node1
    Started container liveness
  Warning  Unhealthy  26s (x9 over 3m5s)   kubelet, node1
    Liveness probe failed: cat: can't open '/tmp/healthy':
No such file or directory
  Normal   Killing    26s (x3 over 2m55s)  kubelet, node1
    Container liveness failed liveness probe, will be
restarted
```

```
[root@master ~]# kubectl get pod
NAME            READY   STATUS    RESTARTS   AGE
liveness-exec   1/1     Running   3          4m12s
```

看到重启3次,慢慢地重启间隔时间会越来越长

案例2: liveness-httpget

1, 编写YMAL文件

```
[root@master ~]# vim pod-liveness-httpget.yml
apiVersion: v1
kind: Pod
metadata:
```

```
  name: liveness-httpget
  namespace: default
spec:
  containers:
  - name: liveness
    image: nginx:1.15-alpine
    imagePullPolicy: IfNotPresent
    ports:                                  # 指定容器端口，这一
段不写也行，端口由镜像决定
    - name: http                            # 自定义名称，不需要
与下面的port: http对应
      containerPort: 80                     # 类似dockerfile里
的expose 80
    livenessProbe:
      httpGet:                              # 使用httpGet方式
        port: http                          # http协议,也可以直
接写80端口
        path: /index.html                   # 探测家目录下的
index.html
      initialDelaySeconds: 3                # 延迟3秒开始探测
      periodSeconds: 5                      # 每隔5s钟探测一次
```

2, 应用YAML文件

```
[root@master ~]# kubectl apply -f pod-liveness-httpget.yml
```

3, 验证查看

```
[root@master ~]# kubectl get pods
NAME                READY   STATUS             RESTARTS
AGE
liveness-httpget    1/1     Running            0
 9s
```

4, 交互删除nginx里的主页文件

```
[root@master ~]# kubectl exec -it liveness-httpget -- rm -
rf /usr/share/nginx/html/index.html
```

5, 验证查看会发现

```
[root@master ~]# kubectl describe pod liveness-http |tail
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  8m17s                default-scheduler  Successfully assigned default/liveness-httpget to node2
  Warning  Unhealthy  5m1s (x3 over 5m11s) kubelet, node2     Liveness probe failed: HTTP probe failed with statuscode: 404
  Normal   Killing    5m1s                 kubelet, node2     Container liveness failed liveness probe, will be restarted
  Normal   Pulled     5m (x2 over 8m15s)   kubelet, node2     Container image "nginx:1.15" already present on machine
  Normal   Created    5m (x2 over 8m14s)   kubelet, node2     Created container liveness
  Normal   Started    5m (x2 over 8m14s)   kubelet, node2     Started container liveness
```

探测到主页不存在,404报错
重启pod后不会再报错,因为
主页被初始化了

```
[root@master ~]# kubectl get pod
NAME                 READY   STATUS    RESTARTS   AGE
liveness-httpget     1/1     Running   1          11m
```

只restart一次

案例3: liveness-tcp

1, 编写YAML文件

```
[root@master ~]# vim pod-liveness-tcp.yml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp
  namespace: default
spec:
  containers:
  - name: liveness
    image: nginx:1.15-alpine
    imagePullPolicy: IfNotPresent
    ports:
    - name: http
      containerPort: 80
    livenessProbe:
      tcpSocket:                    # 使用tcp连接方式
        port: 80                    # 连接80端口进行探测
      initialDelaySeconds: 3
      periodSeconds: 5
```

2, 应用YAML文件创建pod

```
[root@master ~]# kubectl apply -f pod-liveness-tcp.yml
pod/liveness-tcp created
```

3, 查看验证

```
[root@master ~]# kubectl get pod
NAME                READY   STATUS              RESTARTS
AGE
liveness-tcp        1/1     Running             0
 14s
```

4, 交互关闭nginx

```
[root@master ~]# kubectl exec -it liveness-tcp --
/usr/sbin/nginx -s stop
```

5, 再次验证查看

```
[root@master ~]# kubectl describe pod liveness-tcp |tail -8
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  5m59s                default-scheduler  Successfully assigned default/liveness-tcp to node1
  Normal   Pulled     5m9s (x2 over 5m57s)  kubelet, node1     Container image "nginx:1.15" already present on machine
  Normal   Created    5m9s (x2 over 5m56s)  kubelet, node1     Created container liveness
  Warning  Unhealthy  5m9s                 kubelet, node1     Liveness probe failed: dial tcp 10.3.1.42:80: connect: connection refused
  Normal   Started    5m8s (x2 over 5m55s)  kubelet, node1     Started container liveness
```

```
[root@master ~]# kubectl get pod
NAME                READY   STATUS    RESTARTS   AGE
liveness-tcp        1/1     Running   1          5m13s

也只重启1次,重启后重新初始化了
```

案例4: readiness

1, 编写YAML文件

```
[root@master ~]# vim pod-readiness-httpget.yml
apiVersion: v1
kind: Pod
metadata:
  name: readiness-httpget
  namespace: default
spec:
```

```
  containers:
  - name: readiness
    image: nginx:1.15-alpine
    imagePullPolicy: IfNotPresent
    ports:
    - name: http
      containerPort: 80
    readinessProbe:                          # 这里由liveness换
成了readiness
      httpGet:
        port: http
        path: /index.html
      initialDelaySeconds: 3
      periodSeconds: 5
```

2, 应用YAML文件

```
[root@master ~]# kubectl apply -f pod-readiness-
httpget.yml
pod/readiness-httpget created
```

3, 验证查看

```
[root@master ~]# kubectl get pod
NAME                 READY   STATUS              RESTARTS
AGE
readiness-httpget    1/1     Running             0
 10s
```

4, 交互删除nginx主页

```
[root@master ~]# kubectl exec -it readiness-httpget -- rm
-rf /usr/share/nginx/html/index.html
```

5, 再次验证

```
[root@master ~]# kubectl describe pod readiness-httpget |tail -8
Events:
  Type     Reason     Age    From               Message
  ----     ------     ----   ----               -------
  Normal   Scheduled  2m42s  default-scheduler  Successfully assigned default/readiness-httpget to node2
  Normal   Pulled     2m40s  kubelet, node2     Container image "nginx:1.15" already present on machine
  Normal   Created    2m39s  kubelet, node2     Created container readiness
  Normal   Started    2m39s  kubelet, node2     Started container readiness
  Warning  Unhealthy  4s     kubelet, node2     Readiness probe failed: HTTP probe failed with statuscode: 404
```

```
[root@master ~]# kubectl get pod
NAME                 READY   STATUS    RESTARTS   AGE
readiness-httpget    0/1     Running   0          2m49s

READY状态为0/1
```

6, 交互创建nginx主页文件再验证

```
[root@master ~]# kubectl exec -it readiness-httpget --
touch /usr/share/nginx/html/index.html
```

```
[root@master ~]# kubectl get pod
NAME                 READY   STATUS              RESTARTS
AGE
readiness-httpget    1/1     Running             0
 3m10s

READY状态又为1/1了
```

案例5: readiness+liveness综合

1, 编写YAML文件

```
[root@master ~]# vim pod-readiness-liveiness.yml
apiVersion: v1
kind: Pod
metadata:
  name: readiness-liveness-httpget
  namespace: default
spec:
  containers:
  - name: readiness-liveness
    image: nginx:1.15-alpine
    imagePullPolicy: IfNotPresent
    ports:
    - name: http
      containerPort: 80
    livenessProbe:
      httpGet:
        port: http
```

```
        path: /index.html
      initialDelaySeconds: 1
      periodSeconds: 3
    readinessProbe:
      httpGet:
        port: http
        path: /index.html
      initialDelaySeconds: 5
      periodSeconds: 5
```

2, 应用YAML文件

```
[root@master ~]# kubectl apply -f pod-readiness-
liveiness.yml
pod/readiness-liveness-httpget created
```

3, 验证

```
[root@master ~]# kubectl get pod
NAME                          READY   STATUS
RESTARTS   AGE
readiness-liveness-httpget    0/1     Running            0
          6s

10秒前notready
```

```
[root@master ~]# kubectl get pod
NAME                          READY   STATUS
RESTARTS   AGE
liveness-exec                 0/1     CrashLoopBackOff   25
         80m
liveness-httpget              1/1     Running            1
          59m
liveness-tcp                  1/1     Running            1
          44m
readiness-httpget             1/1     Running            0
          20m
readiness-liveness-httpget    1/1     Running            0
          11s

10秒后ready
```

post-start

1, 编写YAML文件

```
[root@master ~]# vim pod-poststart.yml

apiVersion: v1
kind: Pod
metadata:
  name: poststart
  namespace: default
spec:
  containers:
  - name: poststart
    image: nginx:1.15-alpine
    imagePullPolicy: IfNotPresent
    lifecycle:                                          # 生
命周期事件
      postStart:
        exec:
          command: ["mkdir","-
p","/usr/share/nginx/html/haha"]
```

2, 应用YMAL文件

```
[root@master ~]# kubectl apply -f pod-poststart.yml
```

3, 验证

```
[root@master ~]# kubectl get pods
NAME                         READY   STATUS
RESTARTS   AGE
poststart                    1/1     Running            0
         25s
```

```
[root@master ~]# kubectl exec -it poststart -- ls
/usr/share/nginx/html -l
total 8
-rw-r--r-- 1 root root 494 Apr 16 13:08 50x.html
drwxr-xr-x 2 root root   6 Aug  5 05:33 haha              有
创建此目录
-rw-r--r-- 1 root root 612 Apr 16 13:08 index.html
```

pre-stop

容器终止前执行的命令

1, 编写YAML文件

```
[root@master ~]# vim prestop.yml
apiVersion: v1
kind: Pod
metadata:
  name: prestop
  namespace: default
spec:
  containers:
  - name: prestop
    image: nginx:1.15-alpine
    imagePullPolicy: IfNotPresent
    lifecycle:                                          # 生
命周期事件
      preStop:                                          #
preStop
        exec:
          command: ["/bin/sh","-c","sleep 60000000"]     #
容器终止前sleep 60000000秒
```

2, 应用YAML文件创建pod

```
[root@master ~]# kubectl apply -f prestop.yml
pod/prestop created
```

3, 删除pod验证

```
[root@master ~]# kubectl delete -f prestop.yml
pod "prestop" deleted        会在这一步等待一定的时间(大概30s-60s左右)才能删除,说明验证成功
```

结论: 当出现特殊情况不能正常销毁pod时,大概等待30秒会强制终止

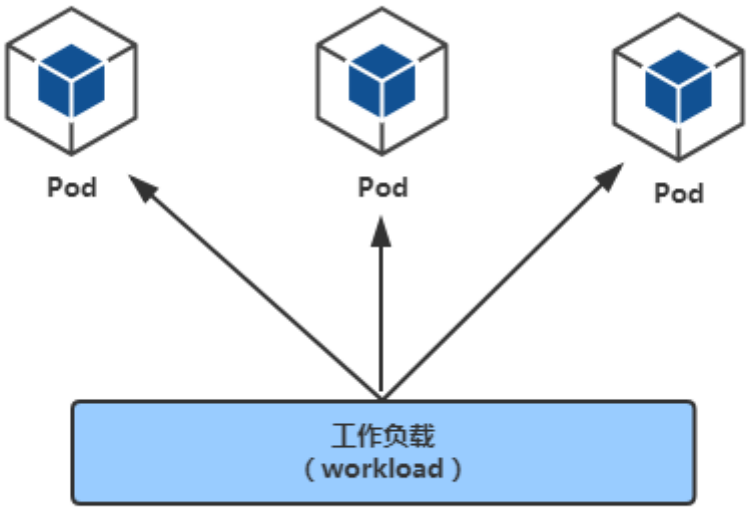
pod故障排除

状态	描述
Pending	pod创建已经提交到Kubernetes。但是，因为某种原因而不能顺利创建。例如下载镜像慢，调度不成功。
Running	pod已经绑定到一个节点，并且已经创建了所有容器。至少有一个容器正在运行中，或正在启动或重新启动。
completed	Pod中的所有容器都已成功终止，不会重新启动。
Failed	Pod的所有容器均已终止，且至少有一个容器已在故障中终止。也就是说，容器要么以非零状态退出，要么被系统终止。
Unknown	由于某种原因apiserver无法获得Pod的状态，通常是由于Master与Pod所在主机kubelet通信时出错。
CrashLoopBackOff	多见于CMD语句错误或者找不到container入口语句导致了快速退出,可以用kubectl logs 查看日志进行排错

- kubectl describe pod pod名
- kubectl logs pod [-c CONTAINER]
- kubectl exec POD [-c CONTAINER] --COMMAND [args...]

七、pod控制器

controller用于控制pod



控制器主要分为:

- ReplicationController(相当于ReplicaSet的老版本,现在建议使用Deployments加ReplicaSet替代RC)
- ReplicaSet　　副本集,控制pod扩容,裁减
- Deployments　控制pod升级,回退
- StatefulSets　部署有状态的pod应用
- DaemonSet　运行在所有集群节点(包括master), 比如使用filebeat,node_exporter
- Jobs　　一次性
- Cronjob　　周期性

Deployment&ReplicaSet

Replicaset控制器的功能:

- 支持新的基于集合的selector(以前的rc里没有这种功能)
- 通过改变Pod副本数量实现Pod的扩容和缩容

Deployment控制器的功能:

- Deployment集成了上线部署、滚动升级、创建副本、回滚等功能
- Deployment里包含并使用了ReplicaSet

Deployment用于部署无状态应用

无状态应用的特点:

- 所有pod无差别
- 所有pod中容器运行同一个image
- 所有pod可以运行在集群中任意node上
- 所有pod无启动顺序先后之分
- 随意pod数量扩容或缩容
- 例如简单运行一个静态web程序

命令创建deployment

1, 创建一个名为nginx1的deployment

如果本地没有镜像,会自动去下载.`nginx:1.15-alpine`比较小巧,下载快.也可以指定自己的内网镜像仓库来提升下载速度

```
[root@master ~]# kubectl run nginx1 --image=nginx:1.15-alpine --port=80 --replicas=1 --dry-run=true
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a future version. Use kubectl run --generator=run-pod/v1 or kubectl create instead.
deployment.apps/nginx1 created (dry run)
```

说明:

- --port=80相当于docker里的暴露端口
- --replicas=1指定副本数,默认也为1
- --dry-run=true为干跑模式,相当于不是真的跑,只是先测试一下
- `--generator=deployment/apps.v1` is DEPRECATED的信息不用管,如果加`--generator=run-pod/v1`参数那么创建的就只有pod而没有deployment

```
[root@master ~]# kubectl run nginx1 --image=nginx:1.15-alpine --port=80 --replicas=1
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a future version. Use kubectl run --generator=run-pod/v1 or kubectl create instead.
deployment.apps/nginx1 created                    看这一行就知道是创建了deployment
```

2, 验证

```
[root@master ~]# kubectl get deployment
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
nginx1   1/1     1            1           119s
```

```
[root@master ~]# kubectl get pod -o wide
NAME                      READY   STATUS    RESTARTS   AGE
    IP          NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-cxcjz   1/1     Running   0
 2m15s   10.3.2.4   node2   <none>           <none>
```

3, 描述deployment和pod相关信息

```
[root@master ~]# kubectl describe deployment nginx1

[root@master ~]# kubectl describe pod nginx1-67f79bc94-jh95m
```

YAML文件创建deployment

1, 准备YAML文件

```
[root@master ~]# vim nginx2-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx2                    # deployment名
spec:
```

```
  replicas: 1                     # 副本集,deployment里使用了
replicaset
  selector:
    matchLabels:
      app: nginx              # 匹配的pod标签,表示deployment和
rs控制器控制带有此标签的pod
  template:                       # 代表pod的配置模板
    metadata:
      labels:
        app: nginx                # pod的标签
    spec:
      containers:                 # 以下为pod里的容器定义
      - name: nginx
        image: nginx:1.15-alpine
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
```

2, 应用YAML文件创建deployment

```
[root@master ~]# kubectl apply -f nginx2-deployment.yml
deployment.apps/nginx2 created
```

3, 查看验证

```
[root@master ~]# kubectl get deployment
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
nginx1   1/1     1            1           12m
nginx2   1/1     1            1           24s
```

```
[root@master ~]# kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
nginx1-7d9b8757cf-cxcjz   1/1     Running   0          14m
nginx2-559567f789-8hstz   1/1     Running   0
 2m18s
```

删除deployment

如果使用`kubectl delete deployment nginx2`命令删除deployment,那么里面的pod也会被自动删除

访问deployment

1,查看pod的IP地址

```
[root@master ~]# kubectl get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE
    IP          NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-cxcjz   1/1     Running   0          16m
    10.3.2.4    node2   <none>           <none>
nginx2-559567f789-8hstz   1/1     Running   0
 4m29s   10.3.1.6    node1   <none>           <none>
可以看到nginx1的pod在node2节点,IP为10.3.2.4
nginx2的pod在node1节点,IP为10.3.1.6
```

2, 查看所有集群节点的CNI网卡

```
[root@master ~]# ifconfig cni |head -2
cni0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.3.0.1  netmask 255.255.255.0  broadcast
0.0.0.0
```

```
[root@node1 ~]# ifconfig cni |head -2
cni0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.3.1.1  netmask 255.255.255.0  broadcast
0.0.0.0
```

```
[root@node2 ~]# ifconfig cni |head -2
cni0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.3.2.1  netmask 255.255.255.0  broadcast
0.0.0.0
```

- 可以看到三个集群节点的IP都为`10.3.0.0/16`这个大网段内的子网

3, 在任意集群节点上都可以访问nginx1和nginx2两个pod

```
# curl 10.3.2.4
# curl 10.3.1.6
结果是任意集群节点都可以访问这两个POD,但集群外部是不能访问的
```

删除deployment中的pod

注意: 是删除deployment中的pod,不是自主式pod

1, 删除nginx1的pod

```
[root@master ~]# kubectl delete pod nginx1-7d9b8757cf-
cxcjz
pod "nginx1-7d9b8757cf-cxcjz" deleted
```

2, 再次查看,发现又重新启动了一个pod(**节点由node2转为node1了,IP地址也变化了**)

```
[root@master ~]# kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE
  IP          NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-czcz4   1/1     Running   0          16s
  10.3.1.7    node1   <none>           <none>
nginx2-559567f789-8hstz   1/1     Running   0          16m
  10.3.1.6    node1   <none>           <none>
```

也就是说**pod的IP不是固定的**,比如把整个集群关闭再启动,pod也会自动启动,但是**IP地址还是变化了**

```
[root@master ~]# kubectl get pod -o wide
NAME                     READY   STATUS    RESTARTS   AGE
   IP          NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-czcz4   1/1     Running   1
 7m7s    10.3.1.9    node1   <none>           <none>
nginx2-559567f789-8hstz   1/1     Running   1          23m
   10.3.1.8    node1   <none>           <none>
```

既然IP地址不是固定的,所以需要一个固定的访问endpoint给用户,那么这种方式就是service.

pod版本升级

查看帮助

```
[root@master ~]# kubectl set image -h
```

1, 升级前验证nginx版本

```
[root@master ~]# kubectl describe pod nginx1-7d9b8757cf-czcz4  |grep Image:
    Image:          nginx:1.15-alpine

[root@master ~]# kubectl exec nginx1-7d9b8757cf-czcz4 -- nginx -v
nginx version: nginx/1.15.12
```

2, **升级为1.16版**

```
[root@master ~]# kubectl set image deployment nginx1 nginx1=nginx:1.16-alpine --record
deployment.extensions/nginx1 image updated
```

说明:

- `deployment nginx1`代表名为nginx1的deployment
- `nginx1=nginx:1.16-alpine`前面的nginx1为容器名
- --record 表示会记录

容器名怎么查看

- `kubectl describe pod pod名`查看
- `kubectl edit deployment deployment名`来查看容器名
- `kubectl get deployment deployment名 -o yaml`来查看容器名
- YAML文件里有指定的pod名就按指定的来

3, 验证

```
[root@master ~]# kubectl rollout status deployment nginx1
Waiting for deployment "nginx1" rollout to finish: 1 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 1 old
replicas are pending termination...
deployment "nginx1" successfully rolled out
```

```
[root@master ~]# kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
nginx1-7ffc8cb4fb-tn4ls   1/1     Running   0
 2m20s     更新后,后面的id变了
nginx2-559567f789-8hstz   1/1     Running   1          68m
```

```
[root@master ~]# kubectl describe pod nginx1-7ffc8cb4fb-
tn4ls |grep Image:
    Image:          nginx:1.16-alpine
升级为1.16了

[root@master ~]# kubectl exec nginx1-7ffc8cb4fb-tn4ls --
nginx -v
nginx version: nginx/1.16.0
升级为1.16了
```

练习: 再将nginx1升级为1.17版 (加上--record可以记录)

```
[root@master ~]# kubectl set image deployment nginx1
nginx1=nginx:1.17-alpine --record
```

pod版本回退

1, 查看版本历史信息

```
[root@master ~]# kubectl rollout history deployment nginx1
deployment.extensions/nginx1
REVISION  CHANGE-CAUSE
1         <none>                    第1版为none,其它升级时加了--
record就会有记录,否则也为none
2         kubectl set image deployment nginx1
nginx1=nginx:1.16-alpine --record=true
3         kubectl set image deployment nginx1
nginx1=nginx:1.17-alpine --record=true
```

2, 定义要回退的版本（还需要执行才是真的回退版本)

```
[root@master ~]# kubectl rollout history deployment nginx1
--revision=1
deployment.extensions/nginx1 with revision #1
Pod Template:
  Labels:       pod-template-hash=7d9b8757cf
        run=nginx1
  Containers:
   nginx1:
    Image:      nginx:1.15-alpine                    可以看
到这是要回退的1.15版本
    Port:       80/TCP
    Host Port:  0/TCP
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

3, 执行回退

```
[root@master ~]# kubectl rollout undo deployment nginx1 --
to-revision=1
deployment.extensions/nginx1 rolled back
```

4, 验证

```
[root@master ~]# kubectl rollout history deployment nginx1
deployment.extensions/nginx1
REVISION  CHANGE-CAUSE
2         kubectl set image deployment nginx1
nginx1=nginx:1.16-alpine --record=true
3         kubectl set image deployment nginx1
nginx1=nginx:1.17-alpine --record=true
4         <none>                                回到了1.15版,但
revision的ID变了
```

```
[root@master ~]# kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
nginx1-7d9b8757cf-m7rt4   1/1     Running   0          97s
nginx2-559567f789-8hstz   1/1     Running   1          89m
```

```
[root@master ~]# kubectl describe pod nginx1-7d9b8757cf-
m7rt4 |grep Image:
    Image:          nginx:1.15-alpine                回到了
1.15版

[root@master ~]# kubectl exec nginx1-7d9b8757cf-m7rt4 --
nginx -v
nginx version: nginx/1.15.12                         回到了
1.15版
```

副本扩容

查看帮助

```
[root@master ~]# kubectl scale -h
```

1, 扩容为2个副本

```
[root@master ~]# kubectl scale deployment nginx1 --
replicas=2
deployment.extensions/nginx1 scaled
```

2, 查看

```
[root@master ~]# kubectl get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE
   IP           NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-m7rt4   1/1     Running   0          17m
   10.3.1.10    node1   <none>           <none>
nginx1-7d9b8757cf-v9xdw   1/1     Running   0          11s
   10.3.2.7     node2   <none>           <none>
nginx2-559567f789-8hstz   1/1     Running   1
 104m   10.3.1.8     node1   <none>           <none>
```

可以看到有2个nginx1的pod,在两个node节点上各1个

3, 继续扩容(我们这里只有2个node,但是可以大于node节点数据)

```
[root@master ~]# kubectl scale deployment nginx1 --
replicas=4
deployment.extensions/nginx1 scaled
```

```
[root@master ~]# kubectl get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE
    IP           NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-2mtzx   1/1     Running   0          4s
     10.3.1.11    node1   <none>           <none>
nginx1-7d9b8757cf-j4hmp   1/1     Running   0          4s
     10.3.2.8     node2   <none>           <none>
nginx1-7d9b8757cf-m7rt4   1/1     Running   0          19m
    10.3.1.10    node1   <none>           <none>
nginx1-7d9b8757cf-v9xdw   1/1     Running   0
 2m25s   10.3.2.7     node2   <none>           <none>
nginx2-559567f789-8hstz   1/1     Running   1
 107m    10.3.1.8     node1   <none>           <none>
```

副本裁减

1, 指定副本数为1进行裁减

```
[root@master ~]# kubectl scale deployment nginx1 --
replicas=1
deployment.extensions/nginx1 scaled
```

2, 查看验证

```
[root@master ~]# kubectl get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE
   IP           NODE    NOMINATED NODE   READINESS GATES
nginx1-7d9b8757cf-m7rt4   1/1     Running   0          22m
   10.3.1.10    node1   <none>           <none>
nginx2-559567f789-8hstz   1/1     Running   1
 110m   10.3.1.8    node1   <none>           <none>
```

多副本滚动更新

1, 先扩容多点副本

```
[root@master ~]# kubectl scale deployment nginx1 --
replicas=16
```

2, 验证

```
[root@master ~]# kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
nginx1-7d9b8757cf-2hd48   1/1     Running   0          61s
nginx1-7d9b8757cf-5m72n   1/1     Running   0          61s
nginx1-7d9b8757cf-5w2xr   1/1     Running   0          61s
nginx1-7d9b8757cf-5wmdh   1/1     Running   0          61s
nginx1-7d9b8757cf-6szjj   1/1     Running   0          61s
nginx1-7d9b8757cf-9dgsw   1/1     Running   0          61s
nginx1-7d9b8757cf-dc7qj   1/1     Running   0          61s
nginx1-7d9b8757cf-l52pr   1/1     Running   0          61s
nginx1-7d9b8757cf-m7rt4   1/1     Running   0          26m
nginx1-7d9b8757cf-mdkj2   1/1     Running   0          61s
nginx1-7d9b8757cf-s79kp   1/1     Running   0          61s
nginx1-7d9b8757cf-shhvk   1/1     Running   0          61s
nginx1-7d9b8757cf-sv8gb   1/1     Running   0          61s
nginx1-7d9b8757cf-xbhf4   1/1     Running   0          61s
nginx1-7d9b8757cf-zgdgd   1/1     Running   0          61s
nginx1-7d9b8757cf-zzljl   1/1     Running   0          61s
nginx2-559567f789-8hstz   1/1     Running   1
 114m
```

3, 滚动更新

```
[root@master ~]# kubectl set image deployment nginx1
nginx1=nginx:1.17-alpine --record
```

4, 验证

```
[root@master ~]# kubectl rollout status deployment nginx1
......
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 12 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 13 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 13 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 13 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 13 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 14 out
of 16 new replicas have been updated...
Waiting for deployment "nginx1" rollout to finish: 4 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 4 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 2 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 2 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 2 old
replicas are pending termination...
```

```
Waiting for deployment "nginx1" rollout to finish: 2 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 1 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 1 old
replicas are pending termination...
Waiting for deployment "nginx1" rollout to finish: 12 of
16 updated replicas are available...
Waiting for deployment "nginx1" rollout to finish: 13 of
16 updated replicas are available...
Waiting for deployment "nginx1" rollout to finish: 14 of
16 updated replicas are available...
Waiting for deployment "nginx1" rollout to finish: 15 of
16 updated replicas are available...
deployment "nginx1" successfully rolled out
```

YAML单独创建replicaset(拓展)

1, 编写YAML文件

```
[root@master ~]# vim rs.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-rs
  namespace: default
spec:                    # replicaset的spec
  replicas: 2            # 副本数
  selector:              # 标签选择器,对应pod的标签
    matchLabels:
      app: nginx         # 匹配的label
  template:
    metadata:
      name: nginx       # pod名
      labels:           # 对应上面定义的标签选择器selector里面
的内容
        app: nginx
    spec:               # pod的spec
```

```
      containers:
      - name: nginx
        image: nginx:1.15-alpine
        ports:
        - name: http
          containerPort: 80
```

2, 应用YAML文件

```
[root@master ~]# kubectl apply -f rs.yml
replicaset.apps/nginx-rs created
```

3, 验证

```
[root@master ~]# kubectl get rs
NAME       DESIRED   CURRENT   READY   AGE
nginx-rs   2         2         2       23s
```

```
[root@master ~]# kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
nginx-rs-6slkh   1/1     Running   0          49s
nginx-rs-f6f2p   1/1     Running   0          49s
```

```
[root@master ~]# kubectl get deployment
No resources found.

找不到deployment,说明创建rs并没有创建deployment
```

八、pod控制器进阶

DaemonSet

- DaemonSet能够让所有（或者特定）的节点运行同一个pod。

- 当节点加入到K8S集群中，pod会被（DaemonSet）调度到该节点上运行，当节点从K8S集群中被移除，被DaemonSet调度的pod会被移除
- 如果删除DaemonSet，所有跟这个DaemonSet相关的pods都会被删除。
- 如果一个DaemonSet的Pod被杀死、停止、或者崩溃，那么DaemonSet将会重新创建一个新的副本在这台计算节点上。
- DaemonSet一般应用于日志收集、监控采集、分布式存储守护进程等

1, 编写YAML文件

```
[root@master ~]# vim nginx-daemonset.yml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
spec:
  selector:
    matchLabels:
      name: nginx-test
  template:
    metadata:
      labels:
        name: nginx-test
    spec:
      tolerations:                        # tolerations代表容忍
      - key: node-role.kubernetes.io/master  # 能容忍的污点key
        effect: NoSchedule   # kubectl explain pod.spec.tolerations查看(能容忍的污点effect)
      containers:
      - name: nginx
        image: nginx:1.15-alpine
        imagePullPolicy: IfNotPresent
        resources:    # resources资源限制是为了防止master节点的资源被占太多(根据实际情况配置)
          limits:
            memory: 100Mi
          requests:
```

```
            memory: 100Mi
```

2, apply应用YAML文件

```
[root@master ~]# kubectl apply -f nginx-daemonset.yml
daemonset.apps/nginx-daemonset created
```

3, 验证

```
[root@master ~]# kubectl get daemonset.apps
NAME               DESIRED   CURRENT   READY   UP-TO-DATE
AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset    3         3         3       3
 3           <none>          117s
```

```
[root@master ~]# kubectl get pods |grep nginx-daemonset
nginx-daemonset-8rqwl      1/1     Running             0
     2m18s
nginx-daemonset-f4dz6      1/1     Running             0
     2m18s
nginx-daemonset-shggq      1/1     Running             0
     2m18s
```

Job

- 对于ReplicaSet而言，它希望pod保持预期数目、持久运行下去，除非用户明确删除，否则这些对象一直存在，它们针对的是耐久性任务，如web服务等。
- 对于非耐久性任务，比如压缩文件，任务完成后，pod需要结束运行，不需要pod继续保持在系统中，这个时候就要用到Job。
- Job负责批量处理短暂的一次性任务 (short lived one-off tasks)，即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

案例1: 计算圆周率2000位

1, 编写YAML文件

```
[root@master ~]# vim job.yml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi          # job名
spec:
  template:
    metadata:
      name: pi      # pod名
    spec:
      containers:
      - name: pi    # 容器名
        image: perl # 此镜像有800多M,可提前导入到所有节点,也可能指定导入到某一节点然后指定调度到此节点
        imagePullPolicy: IfNotPresent
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never   # 执行完后不再重启
```

2, 应用YAML文件创建job

```
[root@master ~]# kubectl apply -f job.yml
job.batch/pi created
```

3, 验证

```
[root@master ~]# kubectl get jobs
NAME   COMPLETIONS   DURATION   AGE
pi     1/1           11s        18s
```

```
[root@master ~]# kubectl get pods
NAME                          READY   STATUS
RESTARTS   AGE
pi-tjq9b                      0/1     Completed          0
        27s

Completed状态,也不再是ready状态
```

```
[root@master ~]# kubectl logs pi-tjq9b
3.14159265358979323846264338327950288419716939937510582097
49445923078164062862089986280348253421170679821480865132 82
30664709384460955058223172535940812848111745028410270193 85
21105559644622948954930381964428810975665933446128475648 23
37867831652712019091456485669234603486104543266482133936 07
26024914127372458700660631558817488152092096282925409171 53
64367892590360011330530548820466521384146951941511609433 05
72703657595919530921861173819326117931051185480744623799 62
74956735188575272489122793818301194912983367336244065664 30
86021394946395224737190702179860943702770539217176293176 75
23846748184676694051320005681271452635608277857713427577 89
60917363717872146844090122495343014654958537105079227968 92
58923542019956112129021960864034418159813629774771309960 51
87072113499999983729780499510597317328160963185950244594 55
34690830264252230825334468503526193118817101000313783875 28
86587533208381420617177669147303598253490428755468731159 56
28638823537875937519577818577805321712268066130019278766 11
19590921642019893809525720106548586327886593615338182796 82
30301952035301852968995773622599413891249721775283479131 51
55748572424541506959508295331168617278558890750983817546 37
46493931925506040092770167113900984882401285836160356370 76
60104710181942955596198946767837449448255379774726847104 04
75346462080466842590694912933136770289891521047521620569 66
02405803815019351125338243003558764024749647326391419927 26
04269922796782354781636009341721641219924586315030286182 97
45557067498385054945885869269956909272107975093029553211 65
34498720275596023648066549911988183479775356636980742654 25
27862551818417574672890977772793800081647060016145249192 17
32172147723501414419735685481613611573525521334757418494 68
43852332390739414333454776241686251898356948556209921922 21
84272550254256887671790494601653466804988627232791786085 78
43838279679766814541009538837863609506800642251252051173 92
98489608412848862694560424196528502221066118630674427862 20
39194945047123713786960956364371917287467764657573962413 89
08658326459958133904780275901
```

案例2: 创建固定次数job

1, 编写YAML文件

```
[root@master ~]# vim job2.yml
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox-job
spec:
  completions: 10
      # 执行job的次数
  parallelism: 1
     # 执行job的并发数
  template:
    metadata:
      name: busybox-job-pod
    spec:
      containers:
      - name: busybox
        image: busybox
        imagePullPolicy: IfNotPresent
        command: ["echo", "hello"]
      restartPolicy: Never
```

2, 应用YAML文件创建job

```
[root@master ~]# kubectl apply -f job2.yml
job.batch/busybox-job created
```

3, 验证

```
[root@master ~]# kubectl get job
NAME          COMPLETIONS   DURATION   AGE
busybox-job   2/10          9s         9s

[root@master ~]# kubectl get job
NAME          COMPLETIONS   DURATION   AGE
busybox-job   3/10          12s        12s

[root@master ~]# kubectl get job
NAME          COMPLETIONS   DURATION   AGE
busybox-job   4/10          15s        15s

[root@master ~]# kubectl get job
```

```
NAME           COMPLETIONS   DURATION   AGE
busybox-job    10/10         34s        48s
```

34秒左右结束

```
[root@master ~]# kubectl get pods
NAME                          READY   STATUS
RESTARTS   AGE
busybox-job-5zn6l             0/1     Completed          0
          34s
busybox-job-cm9kw             0/1     Completed          0
          29s
busybox-job-fmpgt             0/1     Completed          0
          38s
busybox-job-gjjvh             0/1     Completed          0
          45s
busybox-job-krxpd             0/1     Completed          0
          25s
busybox-job-m2vcq             0/1     Completed          0
          41s
busybox-job-ncg78             0/1     Completed          0
          47s
busybox-job-tbzz8             0/1     Completed          0
          51s
busybox-job-vb99r             0/1     Completed          0
          21s
busybox-job-wnch7             0/1     Completed          0
          32s
```

CronJob

- 类似于Linux系统的crontab，在指定的时间周期运行相关的任务

1, 编写YAML文件

```
[root@master ~]# vim cronjob.yml
```

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob1
spec:
  schedule: "* * * * *"                    # 分时日月周
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo hello kubernetes
            imagePullPolicy: IfNotPresent
          restartPolicy: OnFailure
```

2, 应用YAML文件创建cronjob

```
[root@master ~]# kubectl apply -f cronjob.yml
cronjob.batch/cronjob created
```

3, 查看验证

```
[root@master ~]# kubectl get cronjob
NAME       SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE
AGE
cronjob1   * * * * *   False     0        <none>
 5s
```

```
[root@master ~]# kubectl get pods
NAME                           READY   STATUS
RESTARTS   AGE
cronjob-1564993080-qlbgv       0/1     Completed          0
           2m10s
cronjob-1564993140-zbv7f       0/1     Completed          0
           70s
cronjob-1564993200-gx5xz       0/1     Completed          0
           10s
```

看AGE时间,每分钟整点执行一次

九、service

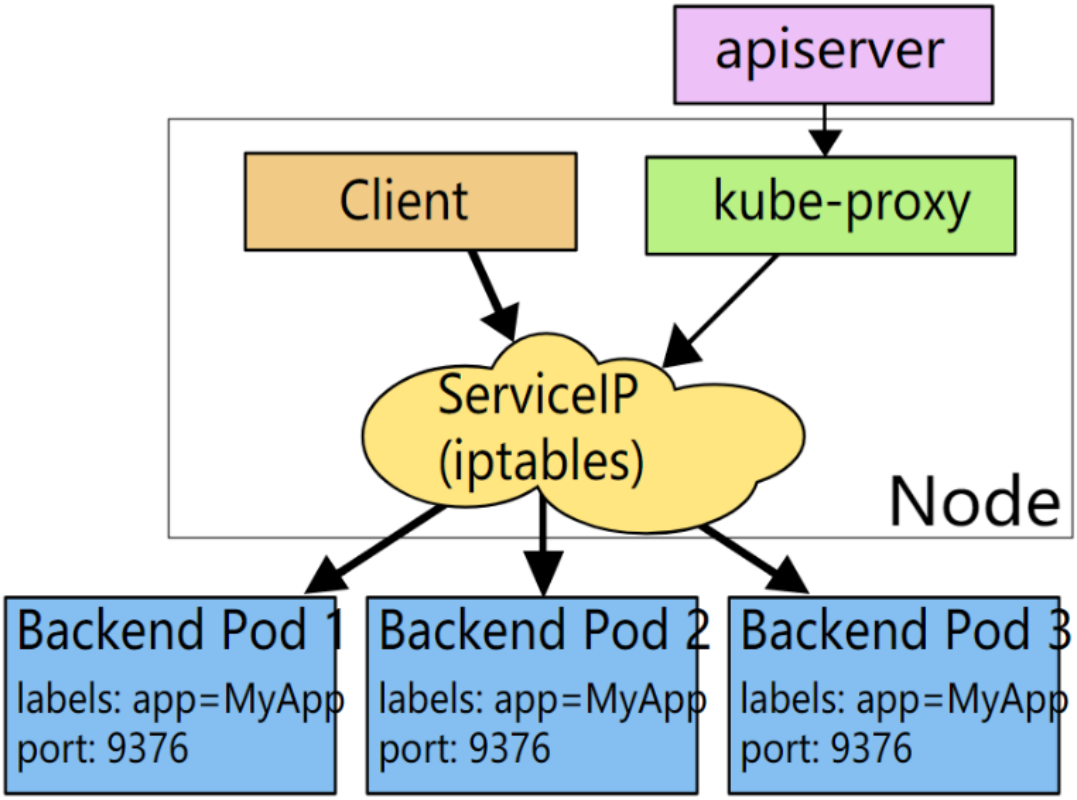
service作用

- 通过service为pod客户端提供访问pod方法，即可客户端访问pod入口
- 通过标签动态感知pod IP地址变化等
- 防止pod失联
- 定义访问pod访问策略
- 通过label-selector相关联
- 通过Service实现Pod的负载均衡（TCP/UDP 4层）
- 底层实现主要通过iptables和IPVS二种网络模式

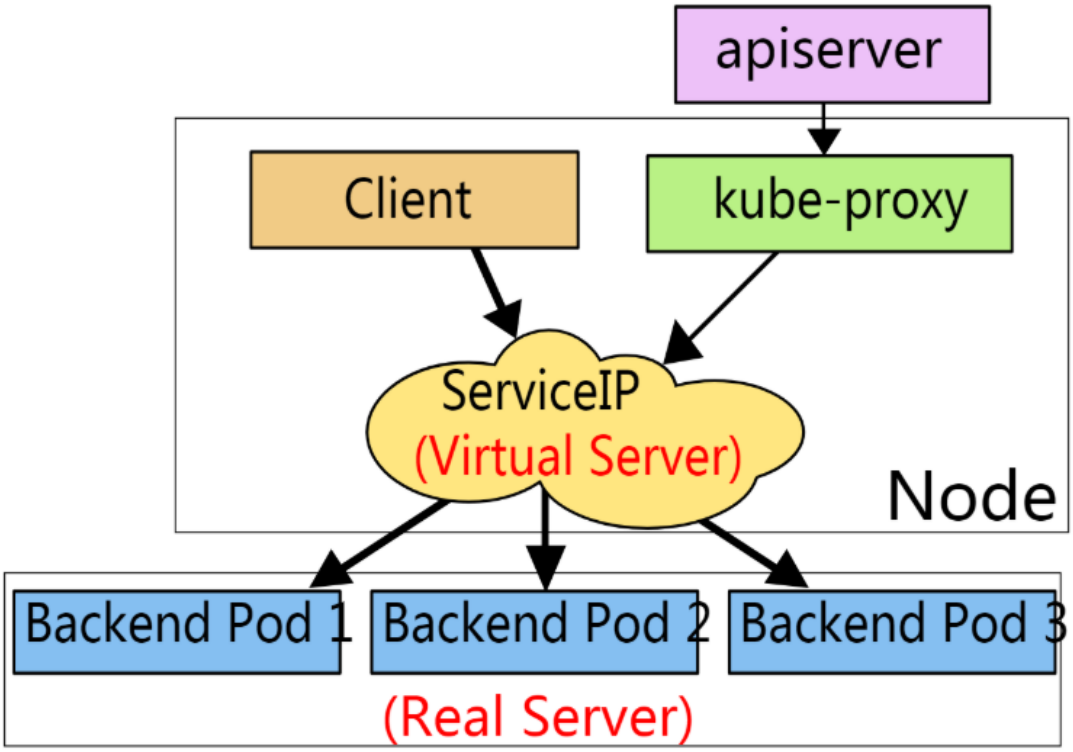
service 底层实现原理

- 底层流量转发与负载均衡实现均可以通过iptables或ipvs实现

iptables实现


apiserver
Client
kube-proxy
ServiceIP
(iptables)
Node
Backend Pod 1
labels: app=MyApp
port: 9376
Backend Pod 2
labels: app=MyApp
port: 9376
Backend Pod 3
labels: app=MyApp
port: 9376

ipvs实现


apiserver
Client
kube-proxy
ServiceIP
(Virtual Server)
Node
Backend Pod 1
Backend Pod 2
Backend Pod 3
(Real Server)

对比

Iptables：

- 灵活，功能强大（可以在数据包不同阶段对包进行操作）
- 规则遍历匹配和更新，呈线性时延

IPVS：

- 工作在内核态，有更好的性能
- 调度算法丰富：rr，wrr，lc，wlc，ip hash...

service类型

service类型分为:

- ClusterIP
  - 默认，分配一个集群内部可以访问的虚拟IP
- NodePort
  - 在每个Node上分配一个端口作为外部访问入口
- LoadBalancer
  - 工作在特定的Cloud Provider上，例如Google Cloud，AWS，OpenStack
- ExternalName
  - 表示把集群外部的服务引入到集群内部中来，即实现了集群内部pod和集群外部的服务进行通信

ClusterIP类型

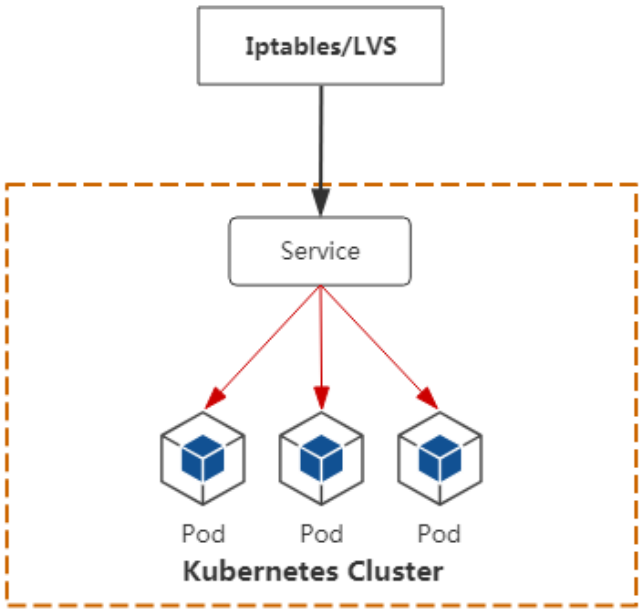
ClusterIP根据是否生成ClusterIP又可分为普通Service和Headless Service两类：

- 普通Service:

为Kubernetes的Service分配一个集群内部可访问的固定虚拟IP(Cluster IP), 实现集群内的访问。

- Headless Service:

  该服务不会分配Cluster IP, 也不通过kube-proxy做反向代理和负载均衡。而是通过DNS提供稳定的网络ID来访问，DNS会将headless service的后端直接解析为podIP列表。



ClusterIP

命令创建方式

1, 查看当前的service

```
[root@master ~]# kubectl get services                    # 或者使用svc简写
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP
PORT(S)    AGE
kubernetes   ClusterIP   10.2.0.1     <none>
 443/TCP    15h

默认只有kubernetes本身自己的services
```

2, 将名为nginx1的deployment映射端口

```
[root@master ~]# kubectl expose deployment nginx1 --
port=80 --target-port=80 --protocol=TCP
service/nginx1 exposed
```

说明:

- 默认是`--type=ClusterIP`,也可以使用`--type="NodePort"`或`--type="ClusterIP"`

3, 验证

```
[root@master ~]# kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)   AGE
kubernetes   ClusterIP   10.2.0.1       <none>
 443/TCP   16h
nginx1       ClusterIP   10.2.211.166   <none>
 80/TCP    9m37s
```

```
[root@master ~]# kubectl get endpoints
NAME         ENDPOINTS
        AGE
kubernetes   192.168.122.11:6443
        16h
nginx1       10.3.1.19:80,10.3.1.20:80,10.3.1.21:80 + 13
more...   10m

因为我的副本数为16,所以这里看到的有16个
```

4, 访问(集群内部任意节点可访问) ,集群外部不可访问

```
# curl 10.2.211.166
```

问题: 一共2个node节点,但有16个pod,那么访问的到底是哪一个呢?

答案: 16个pod会负载均衡，如何验证?

YAML编排方式创建

1,使用前面章节的depolyment产生带有`app=nginx`标签的pod

(过程省略)

2, YAML编写ClusterIP类型service

```
[root@master ~]# cat nginx_service.yml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: default
spec:
  clusterIP: 10.2.11.22      # 这个ip可以不指定，让它自动分配,需要与集群分配的网络对应
  type: ClusterIP            # ClusterIP类型,也是默认类型
  ports:                     # 指定service 端口及容器端口
  - port: 80                 # service ip中的端口
    protocol: TCP
    targetPort: 80           # pod中的端口
  selector:                  # 指定后端pod标签(不是deployment的标签)
     app: nginx              # 可通过kubectl get pod -l app=nginx查看哪些pod在使用此标签
```

3, 应用YAML创建service

```
[root@master ~]# kubectl apply -f nginx_service.yml
service/my-service created
```

4, 验证查看

```
[root@master ~]# kubectl get svc
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)    AGE
kubernetes   ClusterIP   10.2.0.1      <none>
 443/TCP    3d17h
my-service   ClusterIP   10.2.11.22    <none>        80/TCP
   2s     IP对定义的对应了
```

```
[root@master ~]# kubectl get pods -l app=nginx
NAME                                    READY   STATUS
 RESTARTS   AGE
deployment-nginx-6fcfb67547-nv7dn   1/1     Running   0
      122m
deployment-nginx-6fcfb67547-rqrcw   1/1     Running   0
      122m
```

5, 集群内节点访问验证

```
# curl 10.2.11.22
集群内节点都可访问,集群外不可访问
```

6, 两个pod里做成不同的主页方便测试负载均衡

```
[root@master ~]# kubectl exec -it deployment-nginx-
6fcfb67547-nv7dn -- /bin/bash
root@deployment-nginx-6fcfb67547-nv7dn:/# cd
/usr/share/nginx/html/
root@deployment-nginx-6fcfb67547-
nv7dn:/usr/share/nginx/html# echo web1 > index.html
root@deployment-nginx-6fcfb67547-
nv7dn:/usr/share/nginx/html# exit
exit
```

```
[root@master ~]# kubectl exec -it deployment-nginx-
6fcfb67547-rqrcw -- /bin/bash
root@deployment-nginx-6fcfb67547-rqrcw:/# cd
/usr/share/nginx/html/
root@deployment-nginx-6fcfb67547-
rqrcw:/usr/share/nginx/html# echo web2 > index.html
root@deployment-nginx-6fcfb67547-
rqrcw:/usr/share/nginx/html# exit
exit
```

7, 测试

```
# curl 10.2.11.22
多次访问有负载均衡,但是算法看不出来,有点乱
```

sessionAffinity

设置sessionAffinity为Clientip (类似nginx的ip_hash算法,lvs的source hash算法)

```
[root@master ~]# kubectl patch svc my_service -p '{"spec":{"sessionAffinity":"ClientIP"}}'
service/my-service patched
```

测试

```
# curl 10.2.11.22
多次访问,会话粘贴
```

设置回sessionAffinity为None

```
[root@master ~]# kubectl patch svc my-service -p '{"spec":{"sessionAffinity":"None"}}'
service/my-service patched
```

测试

```
# curl 10.2.11.22
多次访问,回到负载均衡
```

修改为ipvs调度方式

从kubernetes1.8版本开始，新增了kube-proxy对ipvs的支持，在kubernetes1.11版本中被纳入了GA.

1, 修改kube-proxy的配置文件

```
[root@master ~]# kubectl edit configmap kube-proxy -n kube-system
     26     iptables:
     27       masqueradeAll: false
     28       masqueradeBit: 14
     29       minSyncPeriod: 0s
     30       syncPeriod: 30s
```

```
    31     ipvs:
    32       excludeCIDRs: null
    33       minSyncPeriod: 0s
    34       scheduler: ""                          # 可以在这里
修改ipvs的算法,默认为rr轮循算法
    35       strictARP: false
    36       syncPeriod: 30s
    37     kind: KubeProxyConfiguration
    38     metricsBindAddress: 127.0.0.1:10249
    39     mode: "ipvs"                                  # 默
认""号里为空,加上ipvs
```

2, 查看kube-system的namespace中kube-proxy有关的pod

```
[root@master ~]# kubectl get pods -n kube-system |grep
kube-proxy
kube-proxy-22x22                          1/1     Running
2          3d20h
kube-proxy-wk77n                          1/1     Running
2          3d19h
kube-proxy-wnrmr                          1/1     Running
2          3d19h
```

3, 验证kube-proxy-xxx的pod中的信息

```
[root@master ~]# kubectl logs kube-proxy-wk77n -n kube-system
W0804 14:38:24.428800       1 server_others.go:249] Flag proxy-mode="" unknown, assuming iptables proxy
I0804 14:38:24.464425       1 server_others.go:143] Using iptables Proxier.
I0804 14:38:24.465154       1 server.go:534] Version: v1.15.1
I0804 14:38:24.483729       1 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_max' to 131072
I0804 14:38:24.483795       1 conntrack.go:52] Setting nf_conntrack_max to 131072
I0804 14:38:24.491317       1 conntrack.go:83] Setting conntrack hashsize to 32768
I0804 14:38:24.497818       1 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_tcp_timeout_established' to 86400
I0804 14:38:24.497919       1 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_tcp_timeout_close_wait' to 3600
I0804 14:38:24.498127       1 config.go:96] Starting endpoints config controller
I0804 14:38:24.498184       1 controller_utils.go:1029] Waiting for caches to sync for endpoints config controller
I0804 14:38:24.498226       1 config.go:187] Starting service config controller
I0804 14:38:24.498253       1 controller_utils.go:1029] Waiting for caches to sync for service config controller
I0804 14:38:24.598449       1 controller_utils.go:1036] Caches are synced for endpoints config controller
I0804 14:38:24.598457       1 controller_utils.go:1036] Caches are synced for service config controller
```

4, 删除kube-proxy-xxx的所有pod，让它重新拉取新的kube-proxy-xxx的pod

```
[root@master ~]# kubectl delete pod kube-proxy-22x22 -n
kube-system
pod "kube-proxy-22x22" deleted
[root@master ~]# kubectl delete pod kube-proxy-wk77n -n
kube-system
pod "kube-proxy-wk77n" deleted
[root@master ~]# kubectl delete pod kube-proxy-wnrmr -n
kube-system
pod "kube-proxy-wnrmr" deleted
```

```
[root@master ~]# kubectl get pods -n kube-system |grep
kube-proxy
kube-proxy-224rc                          1/1     Running
0          22s
kube-proxy-8rrth                          1/1     Running
0          35s
kube-proxy-n2f68                          1/1     Running
0          6s
```

5, 随意查看其中1个或3个kube-proxy-xxx的pod,验证是否为IPVS方式了

```
[root@master ~]# kubectl logs kube-proxy-224rc -n kube-system
I0805 12:32:15.786246       1 server_others.go:170] Using ipvs Proxier.
W0805 12:32:15.786587       1 proxier.go:401] IPVS scheduler not specified, use rr by default
I0805 12:32:15.786900       1 server.go:534] Version: v1.15.1
I0805 12:32:15.809095       1 conntrack.go:52] Setting nf_conntrack_max to 131072
I0805 12:32:15.810246       1 config.go:187] Starting service config controller
I0805 12:32:15.810329       1 controller_utils.go:1029] Waiting for caches to sync for service config controller
I0805 12:32:15.810368       1 config.go:96] Starting endpoints config controller
I0805 12:32:15.810391       1 controller_utils.go:1029] Waiting for caches to sync for endpoints config controller
I0805 12:32:15.910617       1 controller_utils.go:1036] Caches are synced for service config controller
I0805 12:32:15.910845       1 controller_utils.go:1036] Caches are synced for endpoints config controller
```

6, 安装ipvsadm查看规则

```
[root@master ~]# yum install ipvsadm -y

[root@master ~]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight
ActiveConn InActConn
......
TCP  10.2.11.22:80 rr
  -> 10.3.1.61:80                 Masq    1      0
 0
  -> 10.3.2.67:80                 Masq    1      0
 0
......
```

7, 再次验证,就是标准的rr算法了

```
[root@master ~]# curl 10.2.11.22
多次访问,rr轮循
```

思考: 改算法

headless service

普通的ClusterIP service是service name解析为cluster ip,然后cluster ip对应到后面的pod ip

而无头service是指service name 直接解析为后面的pod ip

1, 编写YAML文件

```
[root@master ~]# vim headless-service.yml
apiVersion: v1
kind: Service
metadata:
  name: headless-service
```

```
  namespace: default
spec:
  clusterIP: None                          # None就代表是无头
service
  type: ClusterIP                          # ClusterIP类型,也
是默认类型
  ports:                                   # 指定service 端口
及容器端口
  - port: 80                               # service ip中的端
口
    protocol: TCP
    targetPort: 80                         # pod中的端口
  selector:                                # 指定后端pod标签
     app: nginx                  # 可通过kubectl get pod -l
app=nginx查看哪些pod在使用此标签
```

2, 应用YAML文件创建无头服务

```
[root@master ~]# kubectl apply -f headless-service.yml
service/headless-service created
```

3, 验证

```
[root@master ~]# kubectl get svc
NAME               TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)   AGE
headless-service   ClusterIP   None          <none>
 80/TCP    47s
kubernetes         ClusterIP   10.2.0.1      <none>
 443/TCP   3d21h
my-service         ClusterIP   10.2.11.22    <none>
 80/TCP    47m

可以看到headless-service没有CLUSTER-IP,用None表示
```

DNS

DNS服务监视Kubernetes API,为每一个Service创建DNS记录用于域名解析

headless service需要DNS来解决访问问题

DNS记录格式为: ..svc.cluster.local

1, 查看kube-dns服务的IP

```
[root@master ~]# kubectl get svc -n kube-system
NAME             TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)                  AGE
kube-dns         ClusterIP   10.2.0.10     <none>
 53/UDP,53/TCP,9153/TCP   3d21h
查看到coreDNS的服务地址是10.2.0.10
```

2, 能过DNS服务地址查找无头服务的dns解析

```
[root@master ~]# dig -t A headless-
service.default.svc.cluster.local. @10.2.0.10

; <<>> DiG 9.9.4-RedHat-9.9.4-72.el7 <<>> -t A headless-
service.default.svc.cluster.local. @10.2.0.10
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27991
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0,
ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;headless-service.default.svc.cluster.local. IN A

;; ANSWER SECTION:
headless-service.default.svc.cluster.local. 30 IN A
10.3.2.67       注意这里
headless-service.default.svc.cluster.local. 30 IN A
10.3.1.61       注意这里

;; Query time: 9 msec
;; SERVER: 10.2.0.10#53(10.2.0.10)
```

```
;; WHEN: Mon Aug 05 20:57:51 CST 2019
;; MSG SIZE  rcvd: 187
```

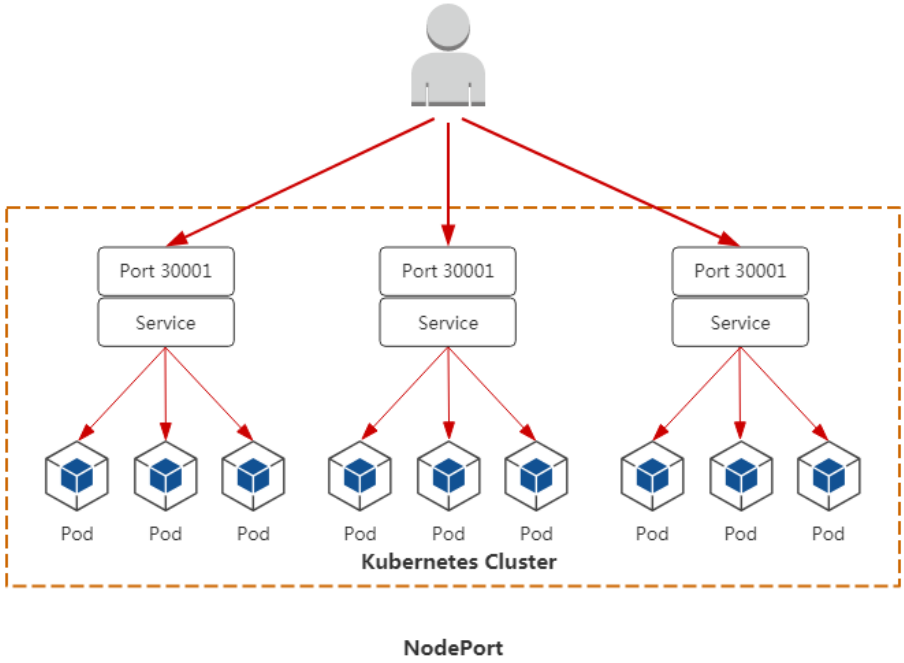
3, 验证pod的IP

```
[root@master ~]# kubectl get pods -o wide
NAME                                     READY   STATUS
 RESTARTS   AGE   IP           NODE    NOMINATED NODE
READINESS GATES
deployment-nginx-6fcfb67547-nv7dn   1/1     Running   0
       65m   10.3.2.67    node2   <none>           <none>
deployment-nginx-6fcfb67547-rqrcw   1/1     Running   0
       65m   10.3.1.61    node1   <none>           <none>
```

可以看到pod的IP与上面dns解析的IP是一致的

NodePort类型

集群外访问：用户->域名->负载均衡器(后端服务器)->NodeIP:Port (service IP) ->Pod IP：端口



NodePort

将nginx1这个service的TYPE由ClusterIP改为NodePort

```
[root@master ~]# kubectl edit service nginx1

apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2019-07-02T08:26:22Z"
  labels:
    run: nginx1
  name: nginx1
  namespace: default
  resourceVersion: "46629"
  selfLink: /api/v1/namespaces/default/services/nginx1
  uid: e4428cbc-85f1-4f0b-aeb3-fbf94926e0c6
spec:
  clusterIP: 10.2.211.166
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 32334
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx1
  sessionAffinity: None
  type: NodePort                    这里由ClusterIP改为NodePort
后保存退出,注意大小写
status:
  loadBalancer: {}
```

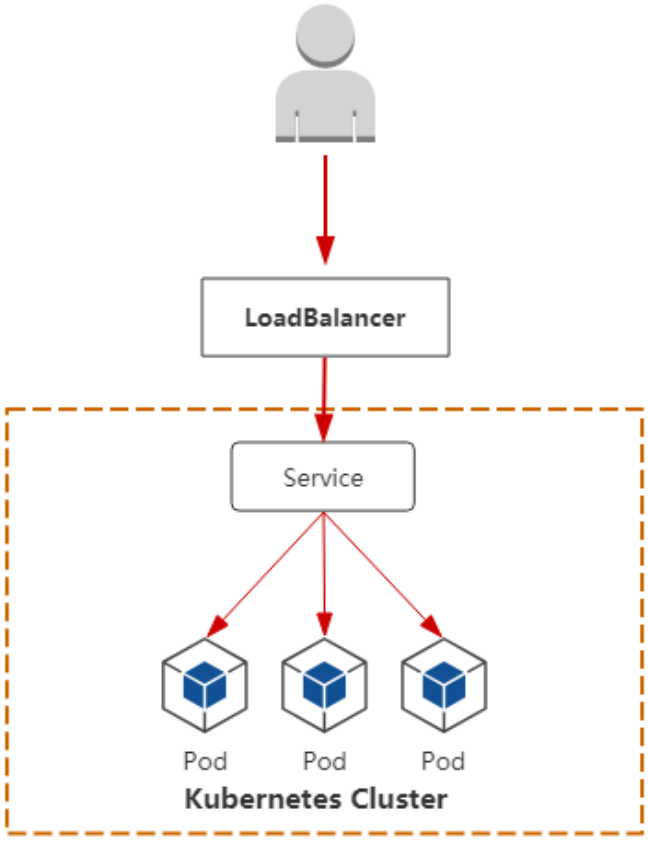
```
[root@master ~]# kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)        AGE
kubernetes   ClusterIP   10.2.0.1       <none>
 443/TCP         17h
nginx1       NodePort    10.2.211.166   <none>
 80:32334/TCP   34m

注意: nginx1后面的TYPE为NodePort了,向外网暴露的端口为32334
```

在集群外部使用`http://任意节点IP:32334`来访问就可以了

```
[root@master ~]# kubectl patch service nginx1 -p '{"spec":
{"type":"NodePort"}}'
```

loadbalancer类型

集群外访问：用户->域名->云服务提供端提供LB->NodeIP:Port(service IP)->Pod IP：端口



LoadBalancer

ExternalName

话不多话,直接做了再下结论

1, 编写YAML文件

```
[root@master ~]# vim externelname.yml

apiVersion: v1
kind: Service
metadata:
  name: my-service                          #  对应的服务是my-
service
  namespace: default
spec:
  type: ExternalName
  externalName: www.itjiangshi.com      # 对应的外部域名为
www.itjiangshi.com
```

2, 应用YAML文件

```
[root@master ~]# kubectl apply -f externelname.yml
service/my-service configured
```

3, 查看my-service的dns解析

```
[root@master ~]# dig -t A my-
service.default.svc.cluster.local. @10.2.0.10

; <<>> DiG 9.9.4-RedHat-9.9.4-72.el7 <<>> -t A my-
service.default.svc.cluster.local. @10.2.0.10
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43624
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;my-service.default.svc.cluster.local. IN A

;; ANSWER SECTION:
```

```
my-service.default.svc.cluster.local. 30 IN CNAME
www.itjiangshi.com.    注意这里

;; Query time: 2001 msec
;; SERVER: 10.2.0.10#53(10.2.0.10)
;; WHEN: Mon Aug 05 21:23:38 CST 2019
;; MSG SIZE  rcvd: 133

从上面看到把外部域名做了一个别名过来
```

结论:

- 这就是把集群外部的服务引入到集群内部中来，实现了集群内部pod和集群外部的服务进行通信
- ExternalName 类型的服务适用于外部服务使用域名的方式，缺点是不能指定端口
- 还有一点要注意: 集群内的Pod会继承Node上的DNS解析规则。所以只要Node可以访问的服务，Pod中也可以访问到, 这就实现了集群内服务访问集群外服务