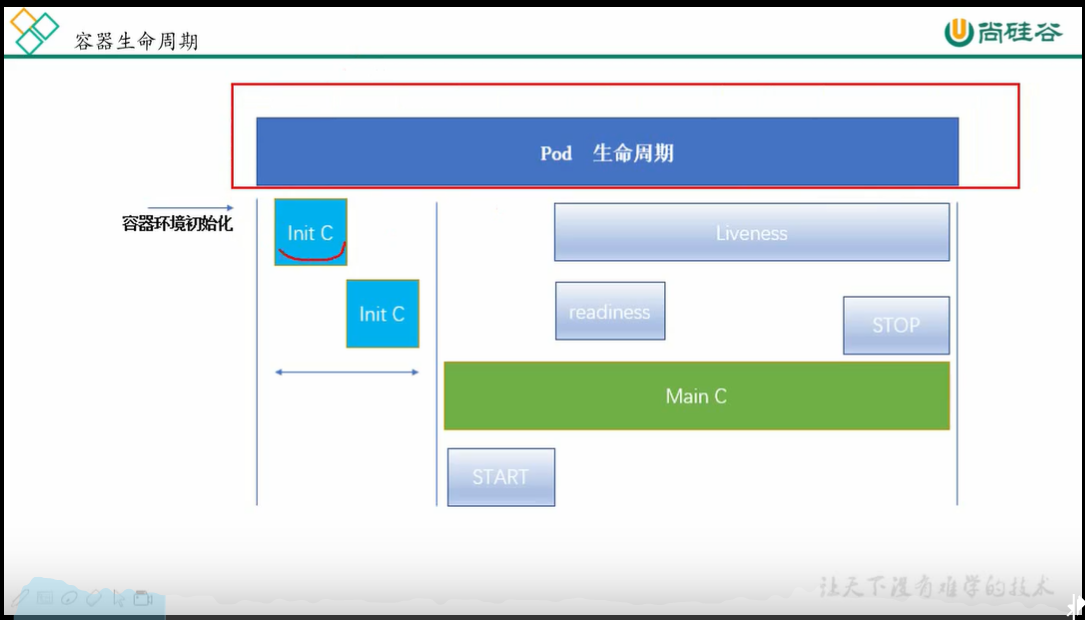
https://www.bilibili.com/video/BV1w4411y7Go?p=31

Pod生命周期



多个initc，不能并行

Readness：就绪检测，判断服务是否可用，是则把状态改为running。不仅仅是pod起来了

Liveness：生存检测，发现容器内部已经不能对外提供正常访问

Kubectl —> kubeapi—etct—>kublet——>cri

Cri完成初始化，首先会创建一个pause基础容器，然后进行多个/0个 container初始化。

运行main c，main c刚运行的时候可以在start阶段执行命令，结束时也可以在stop执行命令。Readness检测成功，pod才会显示为running/ready。Liveness检测到不能正常工作执行对应的删除/重启命令。

initC

初始化容器，Pod能够具有多个容器，应用运行在容器中。但是pod可以有一个或多个在应用容器之前启动的init容器。

Init容器总是运行到成功完成为止。Initc退出pod不会结束，mainc退出pod结束，init成功正常退出后才会有mainc，如果initc没有正常退出pod要进行处理。

每个init容器都必须在上一个成功之后才启动。

每个mainc都会有initc，可以控制启动顺序

Mainc配置：

[root@k8s-master ~]# cat init-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: busybox

command: ['sh', '-c', 'echo The app is running && sleep 360']

initContainers:

- name: init-myservice

image: busybox

command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']

- name: init-mydb

image: busybox

command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']

initc配置

[root@k8s-master ~]# cat myservice.yaml

kind: Service

apiVersion: v1

metadata:

name: myservice

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

initc配置

[root@k8s-master ~]# cat mydb.yaml

kind: Service

apiVersion: v1

metadata:

name: mydb

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9377

kubectl create –f init-pod.yaml 创建mainc，但是不会启动

kubectl create –f myservice.yaml和kubectl create –f mydb.yaml创建了两个mainc依赖的initc后，mainc才会启动。

在pod启动过程中，initc会按顺序在网络和数据卷初始化之后启动，网络和数据卷初始化在pause中，所以pause是第一个启动的。

Pod重启，所有的init容器必须重新执行，所以initc必须是幂等的

对init容器spec的修改被限制在容器image字段，修改其他字段都不会生效，更改init容器的image字段等价于重启该pod

Init和app的名称必须是唯一的，不然会抛出错误。

同一组initc的端口可以一致。因为是顺序启动

资源清单——探针

有kubelet对容器执行的定期诊断。

探测程序：

Execaction：容器内执行指令，0认为成功

Tcpsocketaction：对指定端口上的容器的ip地址进行tcp检查，端口打开则认为成功。

Httpgetactoin：相应码200～400认为成功

成功/失败/未知（不会采取任何操作，挂死）

探测方案：

Livenessprobe：

存活探测。跟随整个生命周期。

Readnessprobe

就绪探测。

探针-就绪检测

Readinessprobe-httpget

[root@k8s-master ~]# cat read.yaml

apiVersion: v1

kind: Pod

metadata:

name: readiness-httpget-pod

namespace: default

spec:

containers:

- name: readiness-httpget-container

image: hub.atguigu.com/library/myapp:v1

imagePullPolicy: IfNotPresent

readinessProbe:

httpGet:

port: 80

path: /index1.html

initialDelaySeconds: 1

periodSeconds: 3

进入容器

kubectl exec readiness-httpget-pod -it -- /bin/sh

检测探针——存活检测

livenessProbe-exec

apiVersion: v1

kind: Pod

metadata:

name: liveness-exec-pod

namespace: default

spec:

containers:

- name: liveness-exec-container

image: busybox

imagePullPolicy: IfNotPresent

command: ["/bin/sh","-c","touch /tmp/live; sleep 60; rm -rf /tmp/live; sleep 3600"]

livenessProbe:

exec:

command: ["test","-e","/tmp/live"]

initialDelaySeconds: 1

periodSeconds: 3

1min后会restart，pod里运行了一个容器，容器启动时会创建一个文件，60s后删除，liveness每3秒检查发现不存在了，就会kill容器，pod重启，如此反复

LivenessProbe-http-get

apiVersion: v1

kind: Pod

metadata:

name: liveness-httpget-pod

namespace: default

spec:

containers:

- name: liveness-httpget-container

image: hub.atguigu.com/library/myapp:v1

imagePullPolicy: IfNotPresent

ports:

- name: http

containerPort: 80

livenessProbe:

httpGet:

port: http

path: /index.html

initialDelaySeconds: 1

periodSeconds: 3

timeoutSeconds: 10

livenesstcpprobe：

apiVersion: v1

kind: Pod

metadata:

name: probe-tcp

namespace: default

spec:

containers:

- name: nginx

image: hub.atguigu.com/library/myapp:v1

livenessProbe:

initialDelaySeconds: 5

timeoutSeconds: 1

tcpSocket:

port: 8080

periodSeconds: 3

start & stop

启动动作：

kind: Pod

metadata:

name: lifecycle-demo

spec:

containers:

- name: lifecycle-demo-container

image: hub.atguigu.com/library/myapp:v1

lifecycle:

postStart:

exec:

command: ["/bin/sh","-c","echo hello from the postStart handler > /usr/share/message"]

preStop:

exec:

command: ["/bin/sh","-c","echo hello from the preStop handler > /usr/share/message"]

状态：

挂起(Pending)：pod已被kubenetes系统接受，但是有一个或多个容器镜像尚未创建，等待时间包括调度pod的时间和通过网路下载镜像的时间。

运行中（running）：pod已经绑定到了一个节点，pod里所有的容器都已被创建，至少有一个容器正在运行或者正处于启动或重启状态。

成功（succeeded）：pod中所有的容器都成功终止，不会再重启。

失败（failed）：pod中所有容器都已终止了，并且至少有一个容器是因为失败终止，即容器以非0的状态退出或被系统终止。

未知（unknown）：因为某些原因无法取得pod的状态，通常是因为pod与所在主机通信失败。

# 资源控制器

Pod的分类

自主式pod

Pod退出后此类型的pod不会被创建。

控制器管理的pod

在控制器的生命周期里始终维持pod的副本数。

什么是控制器：

Kubernetes中内建了很多controller，相当于一个状态机，用来控制pod的具体状态和行为。

控制器类型：

Rc和rs

用来维持pod的副本数。Rs已替代rx，rs支持selector

apiVersion: extensions/v1beta1

kind: ReplicaSet

metadata:

name: frontend

spec:

replicas: 3

selector:

matchLabels:

tier: frontend

template:

metadata:

labels:

tier: frontend

spec:

containers:

- name: myapp

image: hub.atguigu.com/library/myapp:v1

env:

- name: GET\_HOSTS\_FROM

value: dns

以labels为基础进行副本数目的监控。如果改了标签，就会创建新的。

查看标签：kubectl get pods --show-labesl

kubectl label pod frontend-5nd7z tier=1 --overwrite=true

kubectl delete rs –all 也不会删除改了标签的pod

Deployment

为pod和rs提供了声明式定义（declarative）方法

声明式：侧重于定义想要什么。告诉计算机，计算机实现。

Deployment apply

命令式：侧重于如何实现，把实现过程一步步写下来。

Rs create

1. 定义deployment创建rs(多)，rs创建pod(多)
2. 滚动回滚
3. 扩容缩容
4. 暂停继续

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: hub.atguigu.com/library/myap:v1

ports:

- containerPort: 80

kubectl apply -f nginx-deployment.yaml --record

扩容：

kubectl scale deployment nginx-deployment --replicas=10

扩容不会创建新的rs

更新镜像

kubectl set image deployment/nginx-deployment nginx=wangyanglinux/myapp:v2

回滚

kubectl rollout undo deployment/nginx-deployment

kubectl rollout undo deployment/nginx-deployment --to-revision=1 回滚到指定版本

更新策略

1. 保证升级时只有一定数量的pod是down的，默认的会确保至少有比期望的pod数量少一个是up的状态，即最多一个不可用。
2. 同时也确保只创建出超过期望数量的一定数量的pod，默认的会确保最多比期望的pod数量多一个pod是up的。

新的版本会由1-1变为25%-25%

Rollover

假设创建了一个有5个v1版本应用的副本的deployment，创建到三个的时候，升级到v2，此时会立即杀死已创建的三个然后重新创建而不会等到创建完v1的5个后再升级。

查看回滚状态：

kubectl rollout status deployments nginx-deployment

kubectl rollout history deployments nginx-deployment

deployment.extensions/nginx-deployment

REVISION CHANGE-CAUSE

1 kubectl apply --filename=nginx-deployment.yaml --record=true

3 kubectl apply --filename=nginx-deployment.yaml --record=true

4 kubectl apply --filename=nginx-deployment.yaml --record=true

kubectl rollout undo deployment/nginx-deployment --to-revision=1 回滚到指定版本

echo $? 查看退出码，为0表示成功。

清理策略

可以通过.spec.revisionHistoryLimit指定。默认保存全部。

Daemonset

确保全部或一些node上运行一个pod的副本上。

当有node加入集群时，会为他们新增一个node，当有node从集群移除会被回收。

删除daemonset会删除它所创建的所有pod。

典型用法:

1. 运行集群存储daemon，例如在每个node上运行gluster/ceph
2. 每个node上运行日志收集daemon
3. 在每个node上运行监控。

[root@k8s-master ~]# cat daemon.yaml

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: daemonset-example

labels:

app: daemonset

spec:

selector:

matchLabels:

name: daemonset-example

template:

metadata:

labels:

name: daemonset-example

spec:

containers:

- name: daemonset-example

image: wangyanglinux/myapp:v3

kubectl create -f daemon.yaml

默认情况下主节点不会参加调度任务。

Kubect delete pod podname 后会再次在该node新建一个

Job

负责批处理任务，即仅执行一次的任务，保证批处理任务的一个或多个pod成功结束。

apiVersion: batch/v1

kind: Job

metadata:

name: pi

spec:

template:

metadata:

name: pi

spec:

containers:

- name: pi

image: perl

command: ["perl","-Mbignum=bpi","-wle","print (bpi(2000))"]

restartPolicy: Never

kubectl create –f job.yaml

kubectl get pod

pi-6f4hj 0/1 Completed 0 5m39s

状态变成了completed，完成了。

查看日志

Kubectl logs pi-6f4hj

Cronjob

在特定的时间循环创建job，管理基于时间的job

只支持Never和OnFailure两种策略

单个pod时，默认pod成后运行后job即退出

Job

spec.completions 标志job结束需要成功运行的pod个数，默认为1

spec.parallelism 标志并行运行的pod个数

spec.activeDeadlineSeconds：标志失败pod的重试最大时间，超过这个时间不会继续重试。

Cronjob

周期性的在给定时间点运行。定期的创建job

Spec.schedule：调度，必需字段，格式同crontab，\*/1 \* \* \* \* 1min

Spec.jobtemplate：job模板，格式同job

Spec.startingdeadlineseconds: 启动job的期限

Spec.concurrenncyPolicy: 并发策略，指定了如何处理被cron job创建的job的并发执行。Allow-默认-允许并发运行job/forbid-禁止并发运行job，前一个未结束直接跳过下一个/replace-取消当前job，用一个新的来替换。只能应用于同一个cronjob创建的job

Spec.suspend：挂起。

Spec.successfuljobshistorylimit 和 spec.failedjobhistorylimit 成功失败的副本历史

创建cronjob的操作应该是幂等的。

Statefulset

1. 稳定的持久化存储，pod重新调度后能访问到相同的持久化数据，基于pvc实现
2. 稳定的网络标志，即pod重新调度后podName和hostname不变
3. 有序部署，有序扩展。基于init container实现。
4. 有序收缩，有序删除

Horizontal pod autoscaling

管理控制器；基于指标实现自动扩容缩容。

无状态服务部署在rs/deployment

以node为节点 daemonset

批处理任务 job

有状态服务 statefulset

# Service — svc

通过label selector匹配一组pod对外服务，每个svc是一个微服务。一个svc管理多个pod，轮询访问。后端pod死亡后，创建出的新的pod会被svc监控到。默认只提供四层负载。

类型

ClusterIp：

默认类型，自动分配一个仅cluster内部可以访问到的虚拟ip。只能被内部访问

Nodeport：

在clusterip基础上为service在每台机器上绑定一个端口，通过nodeip:nodeport 来访问该服务。对外暴露服务。

Loadbalance：

在nodeport基础上，借助cloud provider创建一个外部负载均衡器，并将请求转发到nodeip:nodeport

Externalname：

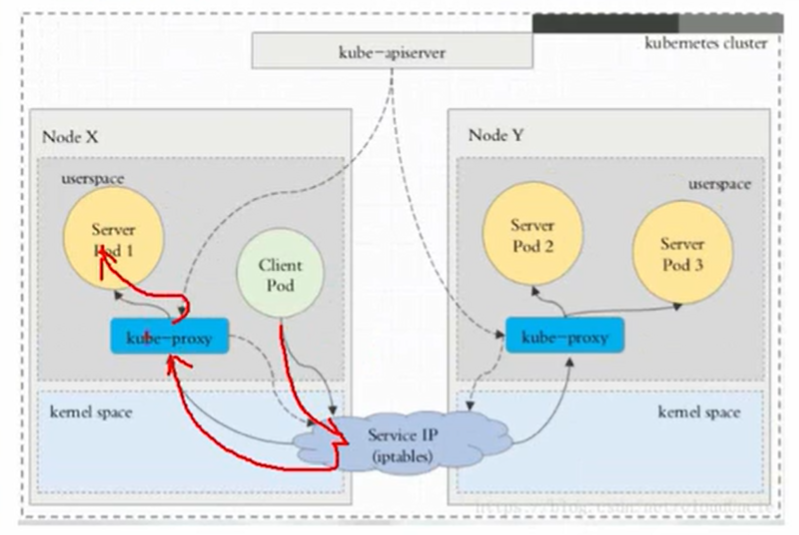
把集群外部的服务引入到集群内部来，在集群内部直接使用，没有任何类型代理被创建。

Vip和service代理

在kubernetes集群中，每个node运行一个kube-proxy进程，kube-proxy负责为service实现了一种vip（虚拟ip）的形式，而不是externalname的形式。

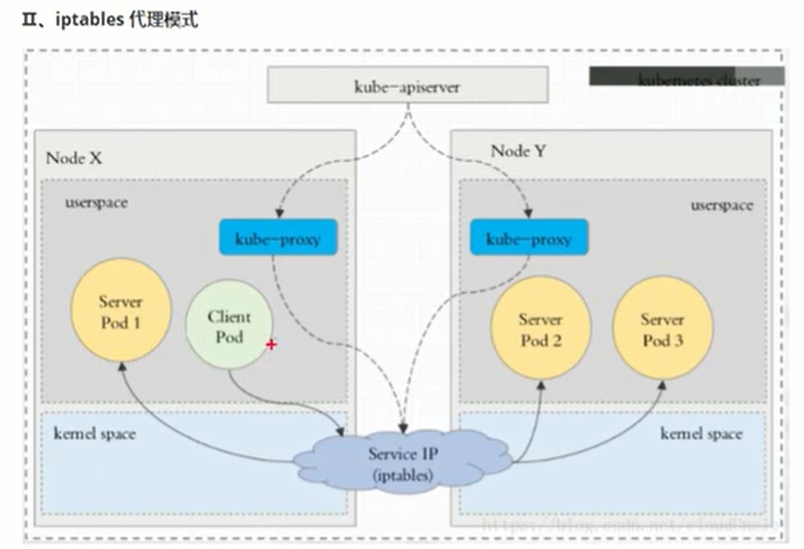
代理模式的分类

1. Userspace代理模式



1. Iptables代理模式

Apiserver用户通过kubectl命令访问



1. Ipvs代理模式

最新版本默认使用ipvs。Ipvs内核模块

