Kubernetes 毕业论文

|  |
| --- |
|  |
|  |
|  |  |
|  | Kubernetes是master worker的工作模式，master上有apiserver组件，etcd master组件以及scheduler等组件，woker上则是有kubelet组件（负责创建容器通过cri 创建容器, csi 创建storage创建存储相关,cni创建网络相关）, cri 这边是可以是调用docker来实现，也可以是containerd。  K8s架构图如下图所示： |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  | **apiserver是处理请求的一个restapi** |
|  | 进入apiserver首先是要认证，audit审计记录log，然后是鉴权，然后是admission准入，分为mutating(加入参数生成新的request)，validating，对生成的新的请求进行校验，以上几步都可以有hook来进行额外的操作， |
|  |
|  | apiserver是挡在etcd之前的它会存在缓存，以减少etcd的压力 |
|  | **基于x509创建访问apiserver的user。** |
|  | 1 生成对应的key |
|  | openssl genrsa -out myuser.key 2048 |
|  | 2. 生成csr certificate sign request |
|  | openssl req -new -key myuser.key -out myuser.csr |
|  | 3. cat myuser.csr | base64 | tr -d "\n" |
|  | 查看csr中的request 填入下方k8s csr对象中 |
|  | apiVersion: certificates.k8s.io/v1 |
|  | kind: CertificateSigningRequest |
|  | metadata: |
|  | name: myuser |
|  | spec: |
|  | request: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQzJ6Q0NBY01DQVFBdR4Q3pBSkJnTlZCQWdNQWtwVE1RCkdjcFlKV2Ivc2VQV25acm8rRlhuMXZHUUUyQlRlUjRpTWZINjNaRUhQSXhwaXlGc2gycUlGNUJPcU4xeVJRa3kKdE5tZmh4QlJiWFJ1Vm9LYnFJY3QKLS0tLS1FTkQgQ0VSVElGSUNBVEUgUkVRVUVTVC0tLS0tCg== |
|  | signerName: kubernetes.io/kube-apiserver-client |
|  | expirationSeconds: 86400 # one day |
|  | usages: |
|  | - client auth |
|  |  |
|  | 创建csr对象 |
|  | kubectl certificate approve myuser |
|  | 创建role rolebinding 关联myuser |
|  | kubectl create role developer --verb=create --verb=get --verb=list --verb=update --verb=delete --resource=pods |
|  | kubectl create rolebinding developer-binding-myuser --role=developer --user=myuser |
|  | kubectl get pods --username=myuser |

**基于webhook(githubtoken)操作flow**

1. **编写webhook的service,监听在比如30000端口**
2. **定义config.json文件,定义监听的server 的访问接口**

{

"kind": "Config",

"apiVersion": "v1",

"preferences": {},

"clusters": [

{

"name": "github-authn",

"cluster": {

"server": "http://192.168.34.2:3000/authenticate"

}

}

],

"users": [

{

"name": "authn-apiserver",

"user": {

"token": "secret"

}

}

],

"contexts": [

{

"name": "webhook",

"context": {

"cluster": "github-authn",

"user": "authn-apiserver"

}

}

],

"current-context": "webhook"}

mkdir -p /etc/config

cp webhook-config.json /etc/config

1. **更新/etc/Kubernetes/manifests/kube-apiserver.yaml 加一行—authentication-token-webhook-config-file=/etc/config/ webhook-config.json,在volumes中定义hostpath: path : /etc/config type: DirectoryOrCreate name: webhook-config, VoulmeMount挂载该文件。**
2. **在/etc/.kube/config 下方加一些user信息**

**Name: myuser123**

**User:**

**Token: github中生成的token**

**5． Kubectl get po –user myuser123**

**etcd**

etcd是基于raft选举安全协议的存储数据库。

Etcd的集群中的node个数推荐或者说必须是奇数个，因为etcd遵循raft协议，需要选主 若为比如4个节点，当有两个节点follower(election time)选举时间到了成为candicate，它们会各自投自己一票，然后还剩两个，如果是一人一票，2比2则要重新投票，所以一般是奇数个etcd节点（http://thesecretlivesofdata.com/raft/）

下一任的leader的任期比上一任的要长。

数据同步是以心跳包的形式，比如etcd leader接受到了数据，它会将数据以心跳包的形式发送到follower上。  
每一个key都有ttl，如果不续约就会过期。

etcdctl 是需要认证的一般使用格式如下

etcdctl --endpoints+https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/health-client.crt --key=/etc/health-client.key snapshot save /opt/data.db

可以使用别名

etcdctl --endpoints=https://[127.0.0.1]:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/healthcheck-client.crt --key=/etc/kubernetes/pki/etcd/healthcheck-client.key member list -w table

查看所有key值

etcdctl get / --prefix --keys-only

查看集群节点状态

etcdctl endpoint status -w table

etcdctl endpoint health -w table

设置key值

etcdctl put wuyong 18

取key值

etcdctl get wuyong

**etcd需要定期备份，可以通过cronjob实现**

etcdctl --endpoints+https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/health-client.crt --key=/etc/health-client.key snapshot save /opt/data.db

apiVersion: batch/v1

kind: CronJob

metadata:

name: hello

spec: #分时日月周

schedule: "\* \* \* \* \*"

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: busybox:1.28

imagePullPolicy: IfNotPresent

command:

- /bin/sh

- -c

- date; etcdctl --endpoints+https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/health-client.crt --key=/etc/health-client.key snapshot save /opt/data.db

restartPolicy: OnFailure

etcdctl restore /opt/data.db

**etcd 优化**

减少网络延迟，etcd集群尽量同地域部署，避免主从数据传输超时，导致重新选举。

减少磁盘io，选择高性能的ssd

为了避免磁盘存储空间不足导致数据写不进去，需要对历史数据进行周期性压缩，启动参数中指定“—auto-compaction”

Etcd是以日志的形式保存数据，无论数据创建还是修改，它都将追加到文件日志，因此日志大小会随数据修改次数而增长，因此要设置文件合适大小。

# K8s中的资源 （ns pod deployment statefulset job

# cronjob service ingress）

首先是namespace对象，它实现了对象的隔离 service deployment statefulset （有序一般搭配headless service使用）以及pod都是隶属于namespace下的资源对象。

删除namespace对象需要先删除namespace下的子资源，同时ns存在一把类似锁的对象finalizer对象。

**Namespace对象**

apiVersion: v1

kind: Namespace

metadata:

creationTimestamp: "2022-07-23T10:26:48Z"

labels:

kubernetes.io/metadata.name: default

name: default

resourceVersion: "203"

uid: 2a16a86c-2b02-4a6d-8b1f-609ed1c22661

spec:

finalizers: #该对象不会物理删除，而是给一个标志位

- kubernetes

status: #状态表示ns活着active

phase: Active

**Pod 对象**

**Pod是k8s管理的最小资源单位**

pod是容器的组合，共享pid network uts namespace。

pod资源的yaml定义如下

root@master:~# kubectl get pod nginx-test -oyaml

apiVersion: v1

kind: Pod

metadata:

annotations: #用于定义metadta之外的属性，也可以定义被prometheus monitor的annotation

cni.projectcalico.org/containerID: ab15a4d20958a83fa29a0a583a7030d1c7edf31532d3f99cbe544ca03d72e165

cni.projectcalico.org/podIP: 192.168.58.91/32

cni.projectcalico.org/podIPs: 192.168.58.91/32

labels: #pod 会有label，会跟同label的deployment bind以及node

run: nginx-test

name: nginx-test

namespace: default #默认属于default namespace

resourceVersion: "624171"

uid: 0f181789-3928-4d7c-8416-6078ca7b38e5

spec:

containers:

- image: nginx

imagePullPolicy: Always #总是从远程拉取

name: nginx-test

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

volumeMounts:

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount #挂载sa

name: kube-api-access-7mhl8

readOnly: true

dnsPolicy: ClusterFirst #默认是ClusterFisrt 会在pod的/etc/resolve.conf，作为dns查找的配置

enableServiceLinks: true

nodeName: harbor #bind在harbor这个node上是schedule调度上去，然后写到etcd并更新pod nodename，本来是空的先创建pod，然后bind

priority: 0 #权重设计到驱逐，权重越高越后驱逐

restartPolicy: Always #pod中容器不论如何停止都将自动重启

schedulerName: default-scheduler

serviceAccount: default

serviceAccountName: default

terminationGracePeriodSeconds: 30 #优雅终止等待容器30秒终止所有进程，30秒后未终止则由kubelet发送SIGKILL信号终止

tolerations: #容忍针对污点的容忍机制

- effect: NoExecute

key: node.kubernetes.io/not-ready

operator: Exists

tolerationSeconds: 300

- effect: NoExecute

key: node.kubernetes.io/unreachable

operator: Exists

tolerationSeconds: 300

volumes:

- name: kube-api-access-7mhl8

projected:

defaultMode: 420

sources:

- serviceAccountToken:

expirationSeconds: 3607

path: token

- configMap:

items:

- key: ca.crt

path: ca.crt

name: kube-root-ca.crt

- downwardAPI:

items:

- fieldRef:

apiVersion: v1

fieldPath: metadata.namespace

path: namespace

status:

conditions:

- lastProbeTime: null

status: "True"

type: Initialized

- lastProbeTime: null

type: Ready

type: ContainersReady

- lastProbeTime: null

lastTransitionTime: "2022-09-24T09:08:16Z"

status: "True"

type: PodScheduled

containerStatuses:

- image: nginx

imageID: ""

lastState: {}

name: nginx-test

ready: false

restartCount: 0

started: false

state:

waiting:

reason: ContainerCreating

hostIP: 192.168.172.133

phase: Pending

qosClass: BestEffort # BestEffort就是不知道用多少，用多少给多少，这种pod的优先级比较低，相对gauarantee

**pod探活**

livenessProbe是探测容器是否健康不健康则会重新建容器

readinessprobe是检测是否处于kubernetes service可接收流量的状态

startupprobe 探测是否启动完成

apiVersion: v1

kind: Pod

metadata: # pod 的源数据信息，可以写多个

name: nginx-busybox # pod 的名字

spec:

containers:

- name: nginx # 容器的名字

image: nginx:alpine # 镜像的名字

ports:

- containerPort: 80

livenessProbe:

httpGet:

path: /

port: 80

httpHeaders:

- name: X-Custom-Header

value: Awesome

initialDelaySeconds: 3

periodSeconds: 3

**pod的暴露端口操作**

kubectl expose pod nginx --port 80

**把pod暴露出去**

**进入容器查看**

kubectl exec -it podname – sh

**创建configmap**

kubectl create cm game-demo --from-literal=play\_init=test\_init

把configmap挂载到container中

**Deployment**

是无状态应用的集合，可以删除用新的会替代，statefulset则不行。

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

run: centos

name: nginx-controller

spec:

replicas: 1

selector:

matchLabels:

run: centos

template:

metadata:

labels:

run: centos

spec:

containers:

- name: centos

image: alpine

command: ["sleep", "3600"]

imagePullPolicy: IfNotPresent

env:

- name: PLAY\_INI

valueFrom:

configMapKeyRef:

name: game-demo

key: play\_init

- name: PLAY\_B

valueFrom:

configMapKeyRef:

name: game-demo

key: play\_init

volumeMounts:

- name: config

mountPath: "/config"

readOnly: true

volumes:

- name: config

configMap:

name: game-demo

**statefulset，是有状态应用的集合**

**它一般用于分布式的或者主从有状态的应用，比如mysql主从，存在一定顺序，一般与headlessservice一起使用，通过core-dns存储的A记录，可以直接访问到对应的podIP**

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

serviceName: "nginx"

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: registry.k8s.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi

**Job 对象**

**是单次作业，只需要执行一次的操作，比如初始化，或者计算逻辑**

apiVersion: batch/v1

kind: Job

metadata：

name: pi

spec:

template:

spec:

containers:

- name: p

image: perl:5.34.0

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

restartPolicy: Never

backoffLimit: 4

**crontab 对象**

**是定时任务,执行定时任务，比如etcd的备份**

apiVersion: batch/v1

kind: CronJob

metadata:

name: hello

spec: #分时日月周

schedule: "\* \* \* \* \*"

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: busybox:1.28

imagePullPolicy: IfNotPresent

command:

- /bin/sh

- -c

- date; echo Hello from the Kubernetes cluster

restartPolicy: OnFailure

**Service对象**

Service对像由kube-proxy通过ipvs iptables实现对pod的负载均衡的策略。

默认策略是roundrobin 同比例分发，第一个pod是1/3第二个是1/2，第三个是100%

svc 与 pod 表面看着是label相同而关联的，其实是通过**endpoint进行关联的**，endpoint controller 会监听pod对象以及svc对象，如果创建了svc对象且包含了selector，那么ep controller对象会创建一个同名的endpoint对象，按照selector去寻找是否有同样label的pod，看pod是否就绪，pod不就绪在notreadaddress ，就绪了addresses里，一个service对应多个pod，多对多的关系，关系由endpoint这个中间表来维护svc与pod关系，selector找到的所有的podip存放在address中的，ready的是存在address中，Notready的存在notreadaddress不就绪的pod不接受流量

note：endpoint对象

是关联pod跟svc的一个对象

Service yaml如下

apiVersion: v1

kind: Service

metadata:

name: my-nginx

spec:

selector:

app: nginx

type: NodePort

ports:

- protocol: TCP

port: 80

**service Topology就近访问**  
k8s提供通用标签标记所处的物理位置，解决就近访问，增加访问速度

apiVersion: v1

kind: Service

metadata:

name: prefer-nodelocal

spec:

ports:

- port: 80

protocol: TCP

name: http

selector:

app: nginx

topologyKeys:

- "kubernetes.io/hostname" #找本机有没有对应ep

- "topology.kubernetes.io/zone"#没有到zone里找

- "topology.kubernetes.io/region"#到region中找

- "\*"#更大区域

**Ingress对象**

**使用ingress原因**

由于service只支持4层的内核态的负载均衡五元组（源ip port 目标IP port 以及协议），service不足：当服务较多时需要较多的虚ip，且有些需要对请求的header进行处理的东东（这个ingress不支持）这个service无法进行有效处理，所以呢需要这个ingress来帮忙实现七层的负载均衡。

当我们需要实现更高的请求要求时我们需要ingress 来实现7层的负载均衡，当然也是ingress controller来监听ingress的资源变化根据相应的rule的规则，生成比如nginx的配置文件从而实现类似nginx的负载均衡，同时也是支持安全协议 tls https，转到host，然后根据不同的path 可以分配到不同的backend service

**Ingress api如下**

**apiVersion: networking.k8s.io/v1**

**kind: Ingress**

**metadata:**

**name: minimal-ingress**

**annotations:**

**kubernetes.io/ingress.class: nginx #通过注释时ingress-nginx**

**nginx.ingress.kubernetes.io/rewrite-target: /**

**spec:**

**tls:**

**- hosts:**

**- https-example.foo.com**

**secretName: testsecret-tls**

**rules:**

**- host: https-example.foo.com**

**http:**

**paths:**

**- path: /testpath**

**pathType: Prefix**

**backend:**

**service: #转到service**

**name: test**

**port:**

**number: 80**

**- path: /seller**

**pathType: Prefix**

**backend:**

**service:**

**name: seller**

**port:**

**number: 90**

**istio微服务治理**

**服务网格(service Mesh),微服务网络和应用的交互**

**iostio微服务治理分为两个部分**

**1 数据面 envoy**

**实现类似nginxhaproxy 的负载均衡，通过从istiod拉取到config，或者是istiod推送到config**

**2 istiod 控制平面**

**通过监听k8s到资源的变化，生成对应的config推送给envoy。**

**Envoy是以sidecar的形式运行在pod旁边的一个pod**

**Envoy是单进程多线程**

**主线程负责协调调度子线程过滤转发**

**Envoy的配置**

#声明静态资源

static\_resources:

listeners: #监听器

- address:

socket\_address:

address: 0.0.0.0

port\_value: 8080

filter\_chains: #过滤链和路由

- filters:

- name: envoy.filters.network.http\_connection\_manager

typed\_config:

"@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager

codec\_type: auto

stat\_prefix: ingress\_http

route\_config:

name: local\_route

virtual\_hosts:

- name: backend

domains:

- "\*"

routes: #路由

- match:

prefix: "/service/1"

route:

cluster: service1 #到指定的cluster

- match:

prefix: "/service/2"

route:

cluster: service2

http\_filters:

- name: envoy.filters.http.router

typed\_config: {}

clusters: #设置集群

- name: service1

connect\_timeout: 0.25s

type: strict\_dns

lb\_policy: round\_robin #均分

http2\_protocol\_options: {}

load\_assignment:

cluster\_name: service1

endpoints:

- lb\_endpoints:

- endpoint:

address:

socket\_address:

address: 10.244.1.1

port\_value: 8000

- name: service2

connect\_timeout: 0.25s

type: strict\_dns

lb\_policy: round\_robin

http2\_protocol\_options: {}

load\_assignment:

cluster\_name: service2

endpoints:

- lb\_endpoints:

- endpoint:

address:

socket\_address:

address: service2

port\_value: 8000

admin: #管理节点

access\_log\_path: "/dev/null"

address:

socket\_address:

address: 0.0.0.0

port\_value: 8001

**envoy同时存在动态配置，通过从istiod中拉取配置，可以动态下发的配置主要有：CDS(cluster discovery service)、LDS(listerner discovery service)、EDS(endpoints discovery service)、RDS(route discovery service)、SDS(secret discovery )，其中只有CDS和LDS的发现地址是在dynamic\_resources中指定的。**

**node:**

**id: id\_1**

**cluster: test**

**dynamic\_resources:**

**cds\_config:**

**path: /var/lib/envoy/cds.yaml**

**lds\_config:**

**path: /var/lib/envoy/lds.yaml**

**admin:**

**access\_log\_path: "/dev/null"**

**address:**

**socket\_address:**

**address: 0.0.0.0**

**port\_value: 19000**

**Istio的流量劫持机制**

**自动注入**

**手动注入**

**Istioctl kube-inject -f bookinfo.yaml**

**注入后的结果**

**注入了init-caontainer istio-init**

**注入了sidecar container istio-proxy**

**将应用容器中的流量转发到envoy的15001端口，用istio-proxy用户，uuid为1337，是envoy的用户空间，使用默认的REDIRECT模式重定向流量，将所有的流量重定向到envoy代理。**

**将所有访问9080端口的流量重定向到envoy代理**

**Gateway**

**Gateway为HTTP/TCP流量配置了一个负载均衡，多数情况下是在网格边缘进行操作，用于启用一个服务的入栈流量。**

**1）selector  
必选字段，通过该标签找到执行gateway规则的envoy。  
2）server  
必选字段，表示开放的服务列表，gateway的关键内容信息，是一个数组，每个元素都是server类型。**

apiVersion: networking.istio.io/v1beta1

kind: Gateway

metadata:

name: httpsserver

spec:

selector:

istio: ingressgateway #这个标签是跟istio的pod的标签一致将配置注入到istio内

servers:

- hosts:

- httpsserver.cncamp.io

port:

name: https-default

number: 443

protocol: HTTPS

tls:

mode: SIMPLE

credentialName: cncamp-credential

**Virtualservice**

**类似高级的service，同时能根据权重负载均衡，bind到gateway上，并通过destination 转发流量到service上**

apiVersion: networking.istio.io/v1beta1

kind: VirtualService

metadata:

name: httpsserver

spec:

gateways:

- httpsserver

hosts:

- httpsserver.cncamp.io

http:

- match:

- port: 443

route:

- destination:

host: httpserver.securesvc.svc.cluster.local #到具体的service的域名上

port:

number: 80

**Destinationrule**

**Yaml如下**

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

name: canary

spec:

host: canary

trafficPolicy:

loadBalancer:

simple: RANDOM

subsets:

- name: v1

labels:

version: v1

- name: v2

labels:

version: v2

trafficPolicy:

loadBalancer:

simple: ROUND\_ROBIN

apiVersion: networking.istio.io/v1beta1

kind: VirtualService

metadata:

name: canary

spec:

hosts:

- canary

http:

- match:

- headers:

user:

exact: yong

route:

- destination:

host: canary

subset: v2

weight: 50

### if upstream server response delay is greater than 1s, send timeout error to client

timeout: 10s #10秒未有响应超时

- route:

- destination:

host: canary

subset: v1

weight: 520

retries: #超时重试3次，时间间隔为2秒

attmpts: 3

perTryTimeout: 2s

apiVersion: apps/v1

kind: Deployment

metadata:

name: canary

spec:

replicas: 1

selector:

matchLabels:

app: canary

template:

metadata:

annotations:

prometheus.io/scrape: "true"

prometheus.io/port: "80"

labels:

app: canary

version: v1

spec:

containers:

- name: canary

imagePullPolicy: Always

image: cncamp/httpserver:v1.0-metrics

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: canary

spec:

ports:

- name: http

port: 80

protocol: TCP

targetPort: 80

selector:

app: canary

apiVersion: apps/v1

kind: Deployment

metadata:

name: canary-v2

spec:

replicas: 1

selector:

matchLabels:

app: canary

template:

metadata:

annotations:

prometheus.io/scrape: "true"

prometheus.io/port: "80"

labels:

app: canary

version: v2

spec:

containers:

- name: canary

imagePullPolicy: Always

image: cncamp/httpserver:v2.0-metrics

ports:

- containerPort: 80

**以上是灰度发布的yaml,定义了两个deployment，分别有app：canary的label，同时有version：v1和v2的不同标签，ep会将service 跟deployment中的pod同label关联。**

**Istio的flow，到了virtualservice根据的destination rule分到不同的label的找到对应的pod，从而实现流量的分发到v1跟v2。**

**Cicd**

**Continus integration and continus deployment。**

**Cicd**

**我们这边是与github集成，在setting中设置比如分支的保护，设置webhook，比如是跟jenkins集成，设置master merge到hook，然后merge到master后会触发jenkins到pipeline，pipeline，会集成sonar，来检测代码的ut覆盖，以及代码质量，同时集成fortify这些安全测试，然后对代码进行编译打包生成image，push到harbor仓库，然后就是进入到cd阶段，pipeline中到flow是从harbor中拉取镜像，执行kubernetes到deploy操作，创建gateway，创建virtualservice，创建destinationrule，然后call helm进行应用到部署，也可以通过argocd直接连接github，通过检测到的部署文件实现部署，监控则是通过prometheus（通过kubelet中的cadvisor监控node pod的container中的实际情况设置阈值，达到阈值，通过alertmanager将报警发送出去通过邮件短信等）来实现监控，只需要在annotation中打上对应的labelPrometheus.io/scrape: ‘true’,并通过grafana对接，将结果显示出来。**