[1. Namespaces资源配额](https://note.youdao.com/ynoteshare1/iframe.html#1012-1576331347317)

[2. 应用资源限制](https://note.youdao.com/ynoteshare1/iframe.html#1786-1576332495515)

[2.1 内存限制](https://note.youdao.com/ynoteshare1/iframe.html#8070-1576332751074)

[2.2 CPU限制](https://note.youdao.com/ynoteshare1/iframe.html#4065-1576333033542)

[3. Limit Range](https://note.youdao.com/ynoteshare1/iframe.html#8531-1576382069971)

[4. Network Policies](https://note.youdao.com/ynoteshare1/iframe.html#8678-1576333294259)

[4.1 相同namespace的NetworkPolicy的隔离性](https://note.youdao.com/ynoteshare1/iframe.html#3632-1576333302236)

[4.2 不同namespace pod的隔离性](https://note.youdao.com/ynoteshare1/iframe.html#8010-1576333349247)

[5. 基于角色的访问权限配置](https://note.youdao.com/ynoteshare1/iframe.html#8682-1576341163710)

[5.1 创建用户](https://note.youdao.com/ynoteshare1/iframe.html#6965-1576384327939)

[5.2. Role和RoleBinding创建](https://note.youdao.com/ynoteshare1/iframe.html#9360-1576341210594)

[5.3. ClusterRole和ClusterRoleBinding创建](https://note.youdao.com/ynoteshare1/iframe.html#9457-1576341210621)

**1. Namespaces资源配额**

<https://kubernetes.io/docs/concepts/policy/resource-quotas/>

# 创建namespaces

kubectl create namespace compute

# 创建计算资源配额

cat << EOF > compute-resources.yaml

apiVersion: v1

kind: ResourceQuota

metadata:

name: compute

namespace: compute

spec:

hard:

cpu: 5

memory: 2Gi

pods: "10"

services: "2"

requests.cpu: 500m

limits.memory: 1Gi

persistentvolumeclaims: "2"

services.nodeports: "2"

configmaps: "2"

secrets: "2"

EOF

kubectl create -f compute-resources.yaml

# 查看配额的最新信息

kubectl -n compute describe resourcequotas compute

**2. 应用资源限制**

**2.1 内存限制**

cat << EOF > stress-mem.yaml

apiVersion: v1

kind: Pod

metadata:

name: memory-demo

spec:

containers:

- name: memory-demo-2-ctr

image: polinux/stress

imagePullPolicy: IfNotPresent

resources:

requests:

memory: "50Mi"

limits:

memory: "100Mi"

command: ["stress"]

args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]

EOF

kubectl apply -f stress-mem.yaml

### 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

[root@master01 ~]# kubectl get pod -n mem-example

NAME READY STATUS RESTARTS AGE

memory-demo 1/1 Running 0 7m13s

memory-demo-2 0/1 OOMKilled 2 20s

**2.2 CPU限制**

cat << EOF > stress-cpu.yaml

apiVersion: v1 kind: Pod metadata: name: cpu-demo spec: containers: - name: cpu-demo-ctr image: vish/stress resources: limits: cpu: "1" requests: cpu: "0.5" args: - -cpus - "2"

EOF

kubectl create -f stress-cpu.yaml

# 查看cpu使用

root@master01:~/yaml# kubectl top pod cpu-demo

NAME CPU(cores) MEMORY(bytes)

cpu-demo 1001m 0Mi

**3. Limit Range**

前面已经说过了，LimitRange是用来给namespace内的pod设置一个默认的request和limit值，所以这里我们简单介绍几个场景

先创建namespace

kubectl create namespace default-cpu-example

使用以下yaml文件创建limitRange，指定一个默认的cpu request和cpu limit

创建个名为cpu-limit-range.yaml的文件，填入以下内容

apiVersion: v1 kind: LimitRange metadata: name: cpu-limit-range spec: limits: - default: cpu: 1 defaultRequest: cpu: 0.5 type: Container

执行

应用到default-cpu-example这个namespace

kubectl apply -f 1.yaml --namespace=default-cpu-example

使用以下yaml文件创建pod

创建名为default-cpu-demo.yaml的yaml文件，输入以下内容

apiVersion: v1 kind: Pod metadata: name: default-cpu-demo spec: containers: - name: default-cpu-demo-ctr image: nginx

部署pod

kubectl apply -f default-cpu-demo.yaml --namespace=default-cpu-example

查看pod yaml配置

kubectl get pod default-cpu-demo --output=yaml --namespace=default-cpu-example

可以看见Resource这里自动配置了limit和request，这个就是继承LimitRange

resources: limits: cpu: "1" requests: cpu: 500m

场景二：只配置pod Resource的limit不配置request 使用以下yaml文件创建pod

default-cpu-demo-2.yaml

apiVersion: v1 kind: Pod metadata: name: default-cpu-demo-2 spec: containers: - name: default-cpu-demo-2-ctr image: nginx resources: limits: cpu: "1"

部署pod

kubectl apply -f default-cpu-demo-2.yaml --namespace=default-cpu-example

查看pod yaml配置

kubectl get pod default-cpu-demo-2 --output=yaml --namespace=default-cpu-example

需要注意的是这里request值没有继承LimitRange配置的值0.5而是直接根limit相等。

resources: limits: cpu: "1" requests: cpu: "1"

场景三：指定容器请求值，不指定容器限额值 使用以下yaml文件创建pod test.yaml

apiVersion: v1 kind: Pod metadata: name: default-cpu-demo-3 spec: containers: - name: default-cpu-demo-3-ctr image: nginx resources: requests: cpu: "0.75"

部署pod

kubectl apply -f test.yaml --namespace=default-cpu-example

查看pod yaml配置

kubectl get pod default-cpu-demo-2 --output=yaml --namespace=default-cpu-example

kubectl get pod default-cpu-demo-2 --output=yaml --namespace=default-cpu-example

需要注意的是这里request值没有继承LimitRange配置的值，而是直接是我们在pod中配置的值，limit继承的是LimitRange的值

resources: limits: cpu: "1" requests: cpu: 750m

总结：

* 如果没有在pod内设置request和limit默认就继承在namespace中配置的LimitRange。
* 如果在pod只配置了Resource的limit没配置request，这时request值不会继承LimitRange配置的值而是直接根pod中配置limit相等。
* 如果在pod中配置了request没有配置limit，这时request值以pod中配置的为准，limit值以namespace中的LimitRange为主。

**4. Network Policies**

**4.1 相同namespace的NetworkPolicy的隔离性**

# 创建一个namespace

kubectl create namespace policy-demo

# 创建pod

kubectl run --namespace=policy-demo nginx --image=nginx --labels=app=nginx

# 创建service

kubectl expose --namespace=policy-demo pod nginx --port=80

# 测试访问正常

kubectl -n policy-demo run test --image=busybox:1.28 --labels=app=busybox --command -- sleep 3600000 kubectl -n policy-demo exec -it test -- wget -q nginx -O -

结果：访问正常

# 创建NetworkPolicy规则

cat <<EOF> default-deny.yaml kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata: name: default-deny namespace: policy-demo spec: podSelector: matchLabels: {} EOF

kubectl create -f default-deny.yaml

此规则表示拒绝pod连接policy-demo namespace下的pod

测试

kubectl -n policy-demo exec -it test-6778df478f-gnkp8 -- wget -q nginx -O -

wget: Download time out

可以看见被拒绝访问了

#添加允许规则

cat <<EOF> access-nginx.yaml kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata: name: access-nginx namespace: policy-demo spec: podSelector: matchLabels: app: nginx ingress: - from: - podSelector: matchLabels: app: busybox EOF

kubectl create -f access-nginx.yaml 这条规则意思，允许，label为 app:busyboxy的pod访问policy-demo namespace下label为app:nginx的pod

# 在次测试,可以访问成功

kubectl -n policy-demo exec -it test-6bdbf87b87-cx5r7 -- wget -q nginx -O -\ 结果：访问成功

# 创建一个不是app: busybox的pod去测试访问 kubectl -n policy-demo run demo --image=busybox:1.28 --labels=app=test --command -- sleep 3600000 kubectl -n policy-demo exec -it demo-6778df478f-gnkp8 -- wget -q nginx -O - 结果：wget：Download time out

结论：同namespace下可以使用Policy在限制pod与pod之间的访问

清空环境

kubectl delete namespace policy-demo

**4.2 不同namespace pod的隔离性**

创建两个namespace policy-demo、policy-demo2，然后在policy-demo里面创建nginx-pod和对应的service和busybox，在policy-demo2里面创建busybox，两个namespace的busybox去访问policy-demo里面的nginx

kubectl create namespace policy-demo kubectl create namespace policy-demo2

kubectl run --namespace=policy-demo nginx --image=nginx --labels=app=nginx

kubectl expose --namespace=policy-demo pod nginx --port=80

kubectl -n policy-demo2 run test --image=busybox:1.28 --labels=app=busybox --command -- sleep 3600000

# 测试访问正常,还没设置NetworkPolicy时分别从policy-demo和policy-demo2两个namespace去busybox去访问nginx,访问成功。

需要注意的是 policy-demo2去访问要接上namespace名

kubectl -n policy-demo2 exec -it test -- wget -q nginx.policy-demo -O -

# 配置NetworkPolicy

cat <<EOF> default-deny.yaml kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata: name: default-deny namespace: policy-demo spec: podSelector: matchLabels: {} EOF

kubectl create -f default-deny.yaml

配置拒绝所有Policy，此时policy-demo2 namespace的busybox都不能访问了

# 配置允许policy-demo2下的app:busybox标签的POD访问policy-demo namespace下的app:nginx服务

cat <<EOF> namespace-access.yaml kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata: name: namespace-access namespace: policy-demo spec: podSelector: matchLabels: app: nginx ingress: - from: - namespaceSelector: matchLabels: project: policy-demo2 EOF

kubectl create -f namespace-access.yaml

此时policy-demo2下的run:access标签的POD访问policy-demo namespace下的run：nginx服务，但其他标签不可以。

运行个run：access2标签的busybox去访问policy-demo namespace下的run：nginx服务

kubectl -n policy-demo2 exec -it test -- wget -q nginx.policy-demo -O -

结果：wget：Download time out

注意：

- namespaceSelector: matchLabels: project: policy-demo2 是namespaces的标签，想要夸namespaces访问通过需要给namespaces打标签并添加到matchLabels里。

kubectl label namespaces policy-demo2 project=policy-demo2

# 在测试

kubectl -n policy-demo2 exec -it test-6bdbf87b87-6vxkc -- wget -q nginx.policy-demo -O -

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

<style>

body {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</style>

</head>

<body>

<h1>Welcome to nginx!</h1>

<p>If you see this page, the nginx web server is successfully installed and

working. Further configuration is required.</p>

<p>For online documentation and support please refer to

<a href="http://nginx.org/">nginx.org</a>.<br/>

Commercial support is available at

<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>

</body>

</html>

**5. 基于角色的访问权限配置**

**5.1 创建用户**

# 使用root账户拷贝ca证书到指定目录下

mkdir /root/certificate

cd /etc/kubernetes/pki

cp ca.key ca.crt /root/certificate/

cd /root/certificate/

# 生成user密钥文件

(umask 077; openssl genrsa -out damon.key 2048)

# 创建证书签署请求

openssl req -new -key damon.key -out damon.csr -subj "/CN=damon/O=cluster01"

注意： CN=<填写username>

O=<value>与集群名称一致,kubectl config get-contexts 查看

# 基于ca签署证书

openssl x509 -req -in damon.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out damon.crt -days 3650

# 验证证书

openssl x509 -noout -text -in damon.crt

# 配置集群信息

kubectl config set-cluster cluster01 \

--embed-certs=true --certificate-authority=ca.crt \

--server=https://172.24.124.109:6443

# 配置客户端证书和密钥

kubectl config set-credentials damon \

--embed-certs=true \

--client-certificate=damon.crt \

--client-key=damon.key

# 设置上下文

kubectl config set-context damon@cluster01 --cluster=cluster01 --user=damon

# 查看上下文

kubectl config get-contexts

# 切换上下文

kubectl config use-context damon@local

**5.2. Role和RoleBinding创建**

# 创建角色 定义这个角色只能对default这个namespace 执行get、watch、list权限 定义角色

cat << EOF > role.yaml

kind: Role apiVersion: rbac.authorization.k8s.io/v1beta1 metadata: namespace: default name: pod-reader rules: - apiGroups: [""] # 空字符串""表明使用core API group resources: ["pods"] verbs: ["get", "watch", "list"]

---

kind: RoleBinding apiVersion: rbac.authorization.k8s.io/v1beta1 metadata: name: read-pods namespace: default subjects: - kind: User name: damon apiGroup: rbac.authorization.k8s.io roleRef: kind: Role name: pod-reader apiGroup: rbac.authorization.k8s.io

EOF

# 使用admin创建

kubectl apply -f role.yaml

# 检查

kubectl get role

kubectl get rolebinding

kubectl get pod

kubectl get nodes

get pod可以但get node不行，因为我们刚刚配置role只有pod权限

删除pod看看

kubectl delete pod/http-app-844765cb6c-nfp7l Error from server (Forbidden): pods "http-app-844765cb6c-nfp7l" is forbidden: User "test-cka" cannot delete pods in the namespace "default"

你会发现也删不掉，因为我们role里面配置的权限是watch和list和get

kubectl get pod -n kube-system No resources found. Error from server (Forbidden): pods is forbidden: User "test-cka"" cannot list pods in the namespace "kube-system" [root@master pki]#

可以看见damon这个用户只能访问default这个namespace的pod资源，其他的namespace都访问不了，同样namespace的其他资源也访问不了

**5.3. ClusterRole和ClusterRoleBinding创建**

cat << EOF > cluster\_role.yaml

kind: ClusterRole apiVersion: rbac.authorization.k8s.io/v1 metadata: name: secret-reader rules: - apiGroups: [""] resources: ["nodes"] verbs: ["get", "watch", "list"]

---

kind: ClusterRoleBinding apiVersion: rbac.authorization.k8s.io/v1 metadata: name: read-secrets-global subjects: - kind: User name: damon apiGroup: rbac.authorization.k8s.io roleRef: kind: ClusterRole name: secret-reader apiGroup: rbac.authorization.k8s.io

EOF

# 创建

kubectl apply -f cluster\_role.yaml

**6. Helm使用**

# 下载Helm

wget https://docs.rancher.cn/download/helm/helm-v3.0.3-linux-amd64.tar.gz

mv linux-amd64/helm /usr/local/bin/

# 查看helm版本

helm version

# 添加Helm Chart Repository

helm repo add stable <https://kubernetes-charts.storage.googleapis.com/>

# 查找mariadb

helm search repo mariadb

# 安装mariadb主从

helm install mariadb --set "master.persistence.enabled=false","slave.persistence.enabled=false" stable/mariadb

# 查看已部署的release列表

helm ls

######## 测试部署的应用

# 获取mysql连接密码

ROOT\_PASSWORD=$(kubectl get secret --namespace default mariadb -o jsonpath="{.data.mariadb-root-password}" | base64 --decode) echo $ROOT\_PASSWORD

# 连接进slave

kubectl exec -it mariadb-slave-0 bash

执行

mysql -u root -pxxxx ///xxx为密码

查看主从连接关系

show slave status\G; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Slave\_IO\_State: Waiting for master to send event Master\_Host: mariadb Master\_User: replicator Master\_Port: 3306 Connect\_Retry: 10 Master\_Log\_File: mysql-bin.000002 Read\_Master\_Log\_Pos: 342 Relay\_Log\_File: mysql-relay-bin.000004 Relay\_Log\_Pos: 641 Relay\_Master\_Log\_File: mysql-bin.000002 Slave\_IO\_Running: Yes Slave\_SQL\_Running: Yes Replicate\_Do\_DB: Replicate\_Ignore\_DB:

# 使用helm升级应用

helm upgrade mariadb stable/mariadb --set "slave.replicas=2","master.persistence.enabled=false","slave.persistence.enabled=false"

更新slave数为2，检查

kubectl get pod NAME READY STATUS RESTARTS AGE mariadb-master-0 1/1 Running 0 14m mariadb-slave-0 1/1 Running 0 14m mariadb-slave-1 1/1 Running 8 14m

# 查看release的历史纪录

helm history mariadb

# helm 回滚

helm rollback mariadb

# 卸载release

helm uninstall mariadb

docker ps -a

docker cp

kubectl drain node02 --ignore-daemonsets --delete-local-data --force

/etc/kubernates/pki