Kubernetes Resource Quota

Raúl Estrada

Octubre 2020

ResourceQuota

By default, pods in Kubernetes are resource-unbounded. Then the running pods might use up all the compute or storage resources in a cluster. ResourceQuota is a resource object that allows us to restrict the resource consumption that a namespace could use. By setting up the resource limit, we could reduce the noisy neighbor symptom. The team working for `project1` won't use up all the resources in the physical cluster.

Then we can ensure the quality of service for other teams working in other projects which share the same physical cluster. There are three kinds of resource quotas supported in Kubernetes 1.7. Each kind includes different resource names, (https://kubernetes.io/docs/concepts/policy/resource-quotas):

- Compute resource quota (CPU, memory)
- Storage resource quota (requested storage, Persistent Volume Claims)
- Object count quotas (pods, RCs, ConfigMaps, services, LoadBalancers)

Created resources won't be affected by newly created resource quotas. If the resource creation request exceeds the specified ResourceQuota, the resources won't be able to start up.

Create a ResourceQuota for a namespace

Now, let's learn the syntax of `ResourceQuota` . Below is one example:

Copy

```
# cat 8-1-2_resource_quota.yml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: project1-resource-quota
spec:
  hard:# the limits of the sum of memory request
    requests.cpu: "1"               # the limits of the sum
    of requested CPU
    requests.memory: 1Gi            # the limits of the sum
    of requested memory
    limits.cpu: "2"           # the limits of total CPU
    limits
    limits.memory: 2Gi        # the limits of total memory
    limit
    requests.storage: 64Gi    # the limits of sum of
    storage requests across PV claims
    pods: "4"                 # the limits of pod number
```

The template is like other objects, just this kind becomes `ResourceQuota`. The quota we specified is valid across the pods which are in a succeeded or failed state (that is, non-terminal state). There are several resource constraints that are supported. In the preceding example, we demonstrate how to set compute ResourceQuota, storage ResourceQuota and object CountQuota. Any time, we could still use the `kubectl` command to check the quota we set: `kubectl describe resourcequota <resource_quota_name>`.

Right now let's modify our existing nginx Deployment by the command `kubectl edit deployment nginx` , changing replica from `2` to `4` and save. Let's list the state now.

Copy

```
# kubectl describe deployment nginx
Replicas:        4 desired | 2 updated | 2 total | 2 available | 2 unavailable
Conditions:
  Type                 Status      Reason
  ----                 ------      ------
  Available            False MinimumReplicasUnavailable
  ReplicaFailure  True  FailedCreate
```

It indicates some pods failed on creation. If we check the corresponding ReplicaSet, we could find out the reason:

Copy

```
# kubectl describe rs nginx-3599227048
...
Error creating: pods "nginx-3599227048-" is forbidden: failed quota: project1-resource-quota: must specify
```

Since we've specified the request limits on memory and CPU, Kubernetes doesn't know the default request limits on the newly desired three pods. We could see the original two pods are still up and running, since the resource quota doesn't apply to existing resources. We now then use `kubectl edit deployment nginx` to modify container specs as follows:

```yaml
spec:
  containers:
  - image: nginx:1.12.0
    imagePullPolicy: IfNotPresent
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    resources:
      limits:
        memory: "300Mi"
        cpu: "300m"
      requests:
        memory: "150Mi"
        cpu: "100m"
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
```

Here, we specify the requests and limits for CPU and memory in the pod spec. It indicates the pod can't exceed the specified quota, otherwise it will be unable to start:

Copy

```
// check the deployment state
# kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     4         3         2            3           2d
```

Available pods become four instead of two, but still not equal to our desired four. What went wrong? If we take a step back and check our resource quota, we can find we've used all the quota of pods. Since Deployments use the rolling update deployment mechanism by default, it'll require pod numbers larger than four, which is exact object limit we set earlier:

Copy

```
# kubectl describe resourcequota project1-resource-quota
Name:             project1-resource-quota
Namespace:        project1
Resource          Used  Hard
--------          ----  ----
limits.cpu        900m  4
limits.memory     900Mi 4Gi
pods              4     4
requests.cpu      300m  4
requests.memory   450Mi 16Gi
requests.storage  0     64Gi
```

After modifying the pods quota from `4` to `8` by `kubectl edit resourcequota project1-resource-quota` command, the Deployment has sufficient resource to launch the pods. Once the `Used` quota exceeds the `Hard` quota, the request will be rejected by the ResourceQuota admission controller, otherwise, the resource quota usage will be updated to ensure sufficient resource allocation.

> **Note**
>
> Since ResourceQuota won't affect already created resources, sometimes we might need to tweak the failed resources, such as deleting an empty change set of RS or scale up and down Deployment, in order to let Kubernetes create new pods or RS which will soak the latest quota limits.

Request pods with default compute resource limits

We could also specify default resource requests and limits for a namespace. Default setting will be used if we don't specify the requests and limits during pod creation. The trick is using `LimitRange` resource object. A `LimitRange` object contains a set of `defaultRequest` (request) and `default` (limits).

> **Note**
>
> LimitRange is controlled by the LimitRanger admission controller plugin. Be sure you enable it if you launch a self-hosted solution. For more information, check out the admission controller section in this chapter.

Below is an example where we set `cpu.request` as `250m` and `limits` as `500m` , `memory.request` as `256Mi` and `limits` as `512Mi` :

Copy

```
# cat 8-1-3_limit_range.yml
apiVersion: v1
kind: LimitRange
metadata:
  name: project1-limit-range
spec:
  limits:
 - default:
     cpu: 0.5
     memory: 512Mi
    defaultRequest:
     cpu: 0.25
     memory: 256Mi
    type: Container
```

Copy

```
// create limit range
# kubectl create -f 8-1-3_limit_range.yml
limitrange "project1-limit-range" created
```

When we launch pods inside this namespace, we don't need to specify the `cpu` and `memory` requests and `limits` anytime, even if we have a total limitation set inside ResourceQuota.

> **Note**
>
> The unit of CPU is core, which is an absolute quantity. It can be an AWS vCPU, a GCP core or a hyperthread on a machine with hyperthreading processor equipped. The unit of memory is a byte. Kubernetes uses the first alphabet or power-of-two equivalents. For example, 256M would be written as 256,000,000, 256 M or 244 Mi.

Additionally, we can set minimum and maximum CPU and memory values for a pod in LimitRange. It acts differently as default values. Default values are only used if a pod spec doesn't contain any requests and limits. The minimum and maximum constraint is used for verifying if a pod requests too much resource. The syntax is `spec.limits[].min` and `spec.limits[].max` . If the request exceeds the minimum and maximum values, forbidden will be thrown from the server.

Copy

```
limits:
   - max:
      cpu: 1
      memory: 1Gi
     min:
      cpu: 0.25
      memory: 128Mi
    type: Container
```

> **Note**
>
> Quality of service for pods: There are three QoS classes for pods in Kubernetes: Guaranteed, Burstable and BestEffort. It's tied together with the namespace and resource management concept we learned above. We also learned QoS in Chapter 4, **Working with Storage and Resources**. Please refer to the last section **Kubernetes Resource Management** in Chapter 4, **Working with Storage and Resources** for recap.

Delete a namespace

Just like any other resources, deleting a namespace is `kubectl delete namespace <namespace_name>` . Please be aware that if a namespace is deleted, all the resources associated with that namespace will be evicted.

Kubeconfig

Kubeconfig is a file that you can use to switch multiple clusters by switching context. We can use `kubectl config view` to view the setting. The following is an example of a minikube cluster in a `kubeconfig` file.

Copy

```
# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /Users/k8s/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /Users/k8s/.minikube/apiserver.crt
    client-key: /Users/k8s/.minikube/apiserver.key
```

Just like what we learned previously. We could use `kubectl config use-context` to switch the cluster to manipulate. We could also use `kubectl config --kubeconfig=<config file name>` to specify which `kubeconfig` file we'd like to use. Only the specified file will be used. We could also specify `kubeconfig` files by the environment variable `$KUBECONFIG`. In this way, config files could be merged. For example, the following command will merge `kubeconfig-file1` and `kubeconfig-file2`:

Copy

```
# export KUBECONFIG=$KUBECONFIG: kubeconfig-file1: kubeconfig-file2
```

You might find we didn't do any specific setting previously. Then where does the output of `kubectl config view` come from? By default, it exists under `$HOME/.kube/config`. This file will be loaded if none of the preceding are set.

Service account

Unlike normal users, **service account** is used by processes inside a pod to contact the Kubernetes API server. By default, a Kubernetes cluster creates different service accounts for different purposes. In GKE, there are bunch of service accounts that have been created:

Copy

```
// list service account across all namespaces
# kubectl get serviceaccount --all-namespaces
NAMESPACE     NAME                         SECRETS   AGE
default       default                      1         5d
kube-public   default                      1         5d
kube-system   namespace-controller         1         5d
kube-system   resourcequota-controller     1         5d
kube-system   service-account-controller   1         5d
kube-system   service-controller           1         5d
project1      default                      1         2h
...
```

Kubernetes will create a default service account in each namespace, which will be used if no service account is specified in pod spec during pod creation. Let's take a look at how the default service account acts for our `project1` namespace:

Copy

```
# kubectl describe serviceaccount/default
Name:       default
Namespace:  project1
Labels:             <none>
Annotations:        <none>
Image pull secrets:       <none>
Mountable secrets:        default-token-nsqls
Tokens:                   default-token-nsqls
```

We could see a service account is basically using mountable secrets as a token. Let's dig into what contents are inside the token:

Copy

```
// describe the secret, the name is default-token-nsqls here
# kubectl describe secret default-token-nsqls
Name:        default-token-nsqls
Namespace:   project1
Annotations:  kubernetes.io/service-account.name=default
              kubernetes.io/service-account.uid=5e46cc5e-
              8b52-11e7-a832-42010af00267
Type: kubernetes.io/service-account-token
Data
====
ca.crt:      # the public CA of api server. Base64 encoded.
namespace:   # the name space associated with this service account. Base64 encoded
token:       # bearer token. Base64 encoded
```

The secret will be automatically mounted to the directory `/var/run/secrets/kubernetes.io/serviceaccount` . When the pod accesses the API server, the API server will check the cert and token to do the authentication. The concept of a service account will be with us in the following sections.