Volume

Volume 是 Pod 中能够被多个容器访问的共享目录。Kubernetes 的 Volume 概念、用途和目的与 Docker 的 Volume 比较类似，但两者不能等价。首先，Kubernetes 中的 Volume 定义在 Pod 上，然后被一个 Pod 里的多个容器挂载到具体的文件目录下；其次，Kubernetes 中的 Volume 与 Pod 的生命周期相同，但与容器的声明周期不相干关，当容器中止或者启动时，Volume 中的数据也不会丢失。最后，Kubernetes 支持多种类型的 Volume，例如 GlusterFS、Ceph 等先进的分布式文件存储。

Volume 的使用也比较简单，在大多数情况下，先在 Pod 上声明一个 Volume，然后在容器里引用该 Volume 并 Mount 到容器里的某个目录上。举例来说，要给之前的 Tomcat Pod 增加一个名字为 datavol 的 Volume，并且 Mount 到容器的 /mydata-data 目录上，则只要对 Pod 的定义文件做如下修改即可：

```
[root@vlnx251101 volume]# vim test-volume.yaml
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
```

```
    containers:
    - name: tomcat-demo
      image: tomcat
      volumeMounts:
        - mountPath: /mydata-data
          name: datavol
      imagePullPolicy: IfNotPresent
    volumes:
    - name: datavol
      emptyDir: {}
```

除了可以让一个 Pod 里的多个容器共享文件，让容器的数据写到宿主机的磁盘上或者写文件到网络存储中，Kubernetes 的 Volume 还扩展出了一种非常有实用价值的功能，即容器配置文件集中化定义与管理，这是通过ConfigMap这个新的资源对象来实现的。

EmptyDir

一个 EmptyDir Volume 是在 Pod 分配到 Node 时创建的。从它的名称就可以看出，它的初始化内容为空，并且无须指定宿主机上对应的目录文件，因为这是 Kubernetes 自动分配的一个目录，当 Pod 从 Node 上移除时，emptyDir 中的数据也会被永久删除。emptyDir 的一些用途如下。

- 临时空间，例如用于某些应用程序运行时所需的临时目录，且无须永久保留。（/var/lib/kubelet/pods/81ada8ab-64cd-11e9-adfc-000c29526d85/volumes/kubernetes.io~empty-dir/datavol/）
- 长时间任务的中间过程 CheckPoint 的临时保存目录。
- 一个容器需要从另一个容器中获取数据的目录（多容器共享目录）。

目前，用户无法控制 emptyDir 使用的介质种类。如果 kubelet 的配置是使用硬盘，那么所有 emptyDir 都将创建在该硬盘上。Pod 在将来可以设置 emptyDir 是位于硬盘、固态硬盘上还是基于内存的 tmpfs 上，上面的例子采用了 emptyDir 类的 Volume。

HostPath

hostPath 为在 Pod 上挂载宿主机上的文件或目录，它通常可以用于以下几方面。

- 容器应用程序生成的日志文件需要永久保存时，可以使用宿主机的高速文件系统进行存储。
- 需要访问宿主机上 Docker 引擎内部数据结构的容器应用时，可以通过定义 hostPath 为宿主机 /var/lib/docker 目录，使容器内部应用可以直接访问 Docker 的文件系统。

在使用这种类型的 Volume，需要注意以下几点。

- 在不同的 Node 上具有相同配置的 Pod 可能会因为宿主机上的目录和文件不同而导致对 Volume 上目录和文件的访问结果不一致。
- 如果使用了资源配额管理，则 Kubernetes 无法将 hostPath 在宿主机上使用的资源纳入管理。

在上面的例子中使用宿主机的 /opt/volume 目录定义了一个 hostPath 类型的 Volume：

```
...
volumes:
- name: datavol
  hostPath:
    path: "/opt/volume/"
```

gcePersistentDisk

使用这种类型的 Volume 表示使用谷歌公有云提供的永久磁盘（Persistent Disk，PD）存放 Volume 的数据，它与 emptyDir 不同，PD 上的内容会被永久保存，当 Pod 被删除时，PD 只是被卸载（Unmount），但不会被删除。需要注意的是，需要先创建一个永久磁盘（PD），才能使用 gcePersistentDisk。

使用 gcePersistentDisk 有以下一些限制条件。

- Node（运行 kubelet 的节点）需要是 GCE 虚拟机。
- 这些虚拟机需要与 PD 存在相同的 GCE 项目和 Zone 中。

通过 gcloud 命令可以创建一个 PD：

```
gcloud compute disks create --size=50GB --zone=us-centrall-a my-data-disk
```

定义 gcePersistentDisk 类型的 Volume 的示例如下：

```
...
- name: test-volume
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

awsElasticBlockStore

与 GCE 类似，该类型的 Volume 使用亚马逊公有云提供的 EBS Volume 存储数据，需要先创建一个 EBS Volume 才能使用 awsElasticBlockStore。

使用 awsElasticBlockStore 的一些限制条件如下：

- Node（运行 kubelet 的节点）需要是 AWS EC2 实例。
- 这些 AWS EC2 实例需要与 EBS volume 存在相同的 region 和 zvailability-zone 中。

- EBS 只支持单个 EC2 实例 mount 一个 volume。

通过 aws ec2 命令可以创建一个 EBS Volume：

```
aws ec2 create-volume --availability-zone eu-west-1a --size 10 --
volume-type gp2
```

定义 awsElasticBlockStore 类型的 Volume 示例如下：

```
...
volumes:
- name: test-volume
  awsElasticBlockStore:
    volumeID: aws://<availability-zone>/<volume-id>
    fsType: ext4
```

NFS

使用 NFS 网络文件系统提供的共享存储数据时，需要在系统中部署一个NFS Server。定义 NFS 类型的 Volume 示例如下：

```
...
volumes:
- name: nfs
  nfs:
    server: nfs-server.localhost # 自己 NFS 服务的地址
    path: "/"
```

其他类型的 Volume

- iscsi：使用 iSCSI 存储设备上的目录挂载到 Pod 中。
- flocker：使用 Flocker 来管理存储卷。

- glusterfs：使用开源 GlusterFS 网络文件系统的目录挂载到 Pod 中。
- rdb：使用 rbd 块设备存储（Rados Block Device）挂载到 Pod 中。
- gitRepo：通过挂载一个空目录，并从 GIT 库 clone 一个 git repository 以供 Pod 使用。
- secret：一个 secret volume 用于为 Pod 提供加密的信息，可以将定义在 Kubernetes 的 secret 直接挂载为文件让 Pod 访问。secret volume 是通过 tmfs（内存文件系统）实现的，所以这种类型的 volume 总是不会持久化。

Persistent Volume

Volume 是定义在 Pod 上的，属于“计算资源”的一部分，而实际上，“网络存储”是相对独立于“计算资源”而存在的一种实体资源。比如在使用虚拟机的情况下，通常会先定义一个网络存储，然后从中划出一个“网盘”并挂载到虚拟机上的。Persistent Volume（简称 PV）和与之相关联的 Persistent Volume Claim（简称 PVC）也起到了类似的作用。
PV 可以理解成 Kubernetes 集群中的某个网络存储中相应的一块存储，它与 Volume 很类似，但有以下区别。

- PV 只能是网络存储，不属于任何 Node，但可以在每个 Node 上访问。
- PV 并不是定义在 Pod 上的，而是独立于 Pod 之外定义。
- PV 目前支持的类型包括：gcePersistenDisk、AWSElasticBlockStore、AzureFile、AzoreDisk、FC(Fibre Channel)、Flocker、NFS、iSCSI、RBD（Rados Block Device）、CephFS、Cinder、GlusterFS、VsphereVolume、Quobyte Volumes、VMware Photon、Portworx Volumes、ScaleIO Volumes 和 HostPath（仅供单机模式）。

下面给出了 NFS 类型 PV 的一个 yaml 定义文件，声明了需要 5Gi 的存储空间：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /somepath
    server: 172.17.0.2
```

比较重要的是 PV 的 accessModes 属性，目前有以下类型。

- ReadWriteOnce：读写权限，并且只能被单个 Node 挂载。
- ReadOnlyMany：只读权限，允许被多个 Node 挂载。
- ReadWriteMany：读写权限，允许被多个 Node 挂载。

如果某个 Pod 想申请某种类型的 PV，则首先需要定义一个 PersistentVolumeClaim（PVC）对象：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
```

```
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

然后，在 Pod 的 Volume 定义中引用上述 PVC 即可：

```
...
volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

然后，PV 是有状态的对象，它有以下几种状态。

- Available：空闲状态。
- Bound：已经绑定到某个 PVC 上。
- Released：对应的 PVC 已经删除，但资源还没有被集群收回。
- Failed：PV 自动回收失败。

```
[root@vlnx251101 volume]# vim test-volume.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv001
  labels:
    pv: pv001
spec:
  capacity:
```

```
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /opt/nfsshare
    server: 192.168.251.103
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      pv: pv001
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    tier: frontend
  template:
```

```
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: tomcat-demo
        image: tomcat
        volumeMounts:
          - mountPath: /mydata-data
            name: mypv
        imagePullPolicy: IfNotPresent
      volumes:
      - name: mypv
        persistentVolumeClaim:
          claimName: myclaim
```

```
[root@vlnx251101 volume]# kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS    CLAIM              STORAGECLASS   REASON    AGE
pv001     1Gi        RWO            Retain           Bound
    default/myclaim                                 1m
```

```
[root@vlnx251101 volume]# kubectl get pvc
NAME      STATUS    VOLUME    CAPACITY   ACCESS MODES
STORAGECLASS   AGE
myclaim   Bound     pv001     1Gi        RWO
                                1m
```

```
[root@vlnx251101 volume]# kubectl get pod
NAME               READY     STATUS    RESTARTS   AGE
frontend-dsdrb     1/1       Running   0          2m


[root@vlnx251101 volume]# kubectl exec -it frontend-dsdrb
-- /bin/bash

root@frontend-dsdrb:/usr/local/tomcat# ls /mydata-data/
a.txt
root@frontend-dsdrb:/usr/local/tomcat# cat /mydata-
data/a.txt
103
```

ConfigMap

其实ConfigMap功能在Kubernetes1.2版本的时候就有了，许多应用程序会从配置文件、命令行参数或环境变量中读取配置信息。这些配置信息需要与docker image解耦，你总不能每修改一个配置就重做一个image吧？ConfigMap API给我们提供了向容器中注入配置信息的机制，ConfigMap可以被用来保存单个属性，也可以用来保存整个配置文件或者JSON二进制的对象。

ConfigMap概览

ConfigMap API资源用来保存**key-value pair**配置数据，这个数据可以在**pods**里使用，或者被用来为像**controller**一样的系统组件存储配置数据。虽然ConfigMap跟Secrets类似，但是ConfigMap更方便的处理不含敏感信息的字符串。 注意：ConfigMaps不是属性配置文件的替代品。ConfigMaps只是作为多个properties文件的引用。你可以把它理解为Linux系统中的/etc目录，专门用来存储配置文件的目录。下面举个例子，使用ConfigMap配置来创建Kuberntes Volumes，ConfigMap中的每个data项都会成为一个新文件。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-config
  namespace: default
data:
  example.property.1: hello
  example.property.2: world
  example.property.file: |
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

data一栏包括了配置数据，ConfigMap可以被用来保存单个属性，也可以用来保存一个配置文件。 配置数据可以通过很多种方式在Pods里被使用。ConfigMaps可以被用来：

1. 设置环境变量的值
2. 在容器里设置命令行参数
3. 在数据卷里面创建config文件

用户和系统组件两者都可以在ConfigMap里面存储配置数据。

其实不用看下面的文章，直接从`kubectl create configmap -h`的帮助信息中就可以对ConfigMap究竟如何创建略知一二了。

```
Examples:
  # Create a new configmap named my-config based on folder
bar
  kubectl create configmap my-config --from-
file=path/to/bar

  # Create a new configmap named my-config with specified
keys instead of file basenames on disk
  kubectl create configmap my-config --from-
file=key1=/path/to/bar/file1.txt --from-
file=key2=/path/to/bar/file2.txt

  # Create a new configmap named my-config with
key1=config1 and key2=config2
  kubectl create configmap my-config --from-
literal=key1=config1 --from-literal=key2=config2
```

创建ConfigMaps

可以使用该命令，用给定值、文件或目录来创建ConfigMap。

```
kubectl create configmap
```

使用目录创建

比如我们已经有个了包含一些配置文件，其中包含了我们想要设置的ConfigMap的值：

```
[root@vlnx251101 configmap]# pwd
/test/configmap


[root@vlnx251101 configmap]# cat game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30


[root@vlnx251101 configmap]# cat ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

使用下面的命令可以创建一个包含目录中所有文件的ConfigMap。

```
[root@vlnx251101 configmap]# kubectl create configmap
game-config --from-file=/test/configmap
configmap/game-config created
```

–from-file指定在目录下的所有文件都会被用在ConfigMap里面创建一个键值对，键的名字就是文件名，值就是文件的内容。

让我们来看一下这个命令创建的ConfigMap：

```
[root@vlnx251101 configmap]# kubectl describe configmap
game-config
Name:         game-config
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
Events:  <none>
```

我们可以看到那两个key是从kubectl指定的目录中的文件名。这些key的内容可能会很大，所以在kubectl describe的输出中，只能够看到键的名字和他们的大小。 如果想要看到键的值的话，可以使用kubectl get以yaml格式输出配置。

```
[root@vlnx251101 configmap]# kubectl get configmap game-
config -o yaml
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2018-08-12T06:46:09Z
  name: game-config
  namespace: default
  resourceVersion: "137843"
  selfLink: /api/v1/namespaces/default/configmaps/game-
config
  uid: 64f68593-9dfb-11e8-b407-000c29526d85
```

使用文件创建

刚才**使用目录创建**的时候我们`--from-file`指定的是一个目录，只要指定为一个文件就可以从单个文件中创建`ConfigMap`。

```
[root@vlnx251101 configmap]# kubectl create configmap game-config-2 --from-file=/test/configmap/game.properties
configmap/game-config-2 created


[root@vlnx251101 configmap]# kubectl get configmaps game-config-2 -o yaml
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2018-08-12T06:51:38Z
  name: game-config-2
  namespace: default
  resourceVersion: "138308"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: 292383c8-9dfc-11e8-b407-000c29526d85
```

`--from-file`这个参数可以使用多次，你可以使用两次分别指定上个实例中的那两个配置文件，效果就跟指定整个目录是一样的。

使用字面值创建

使用文字值创建，利用`--from-literal`参数传递配置信息，该参数可以使用多次，格式如下；

```
[root@vlnx251101 configmap]# kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm
configmap/special-config created


[root@vlnx251101 configmap]# kubectl get configmaps special-config -o yaml
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2018-08-12T06:52:58Z
  name: special-config
  namespace: default
  resourceVersion: "138424"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
```

```
  uid: 591fcbae-9dfc-11e8-b407-000c29526d85
```

Pod中使用ConfigMap

使用ConfigMap来替代环境变量

ConfigMap可以被用来填入环境变量。看下下面的ConfigMap。

```
[root@vlnx251101 configmap]# vim configmap-test.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO

[root@vlnx251101 configmap]# kubectl create -f configmap-
test.yaml
configmap/special-config created
```

```
configmap/env-config created
```

我们可以在Pod中这样使用ConfigMap：

```
[root@vlnx251101 configmap]# vim dapi-test-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
      envFrom:
        - configMapRef:
            name: env-config
  restartPolicy: Never
```

```
[root@vlnx251101 configmap]# kubectl create -f dapi-test-pod.yaml
pod/dapi-test-pod created

[root@vlnx251101 configmap]# kubectl describe pod  dapi-test-pod

    Environment Variables from:
      env-config  ConfigMap  Optional: false
    Environment:
      SPECIAL_LEVEL_KEY:  <set to the key 'special.how' of config map 'special-config'>   Optional: false
      SPECIAL_TYPE_KEY:   <set to the key 'special.type' of config map 'special-config'>  Optional: false

[root@vlnx251101 configmap]# kubectl logs dapi-test-pod
SPECIAL_TYPE_KEY=charm
SPECIAL_LEVEL_KEY=very
log_level=INFO

[root@vlnx251101 configmap]# kubectl delete -f dapi-test-pod.yaml
```

用ConfigMap设置命令行参数

ConfigMap也可以被使用来设置容器中的命令或者参数值。它使用的是Kubernetes的$(VAR_NAME)替换语法。我们看下下面这个ConfigMap。

为了将ConfigMap中的值注入到命令行的参数里面，我们还要像前面那个例子一样使用环境变量替换语法${VAR_NAME)。（其实这个东西就是给Docker容器设置环境变量，以前我创建镜像的时候经常这么玩，通过docker run的时候指定-e参数修改镜像里的环境变量，然后docker的CMD命令再利用该$(VAR_NAME)通过sed来修改配置文件或者作为命令行启动参数。）

```
[root@vlnx251101 configmap]# vim dapi-test-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo
$(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

```
[root@vlnx251101 configmap]# kubectl create -f dapi-test-pod.yaml
pod/dapi-test-pod created

[root@vlnx251101 configmap]# kubectl describe pod  dapi-test-pod

    Environment:
      SPECIAL_LEVEL_KEY:  <set to the key 'special.how' of config map 'special-config'>  Optional: false
      SPECIAL_TYPE_KEY:   <set to the key 'special.type' of config map 'special-config'>  Optional: false

[root@vlnx251101 configmap]# kubectl logs dapi-test-pod
very charm

[root@vlnx251101 configmap]# kubectl delete -f dapi-test-pod.yaml
```

通过数据卷插件使用ConfigMap

ConfigMap也可以在数据卷里面被使用。还是这个ConfigMap。

在数据卷里面使用这个ConfigMap，有不同的选项。最基本的就是将文件填入数据卷，在这个文件中，键就是文件名，键值就是文件内容：

```
[root@vlnx251101 configmap]# vim dapi-test-pod.yaml
apiVersion: v1
```

```
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat
/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

```
[root@vlnx251101 configmap]# kubectl create -f dapi-test-
pod.yaml
pod/dapi-test-pod created
```

```
[root@vlnx251101 configmap]# kubectl logs dapi-test-pod
very
```

```
[root@vlnx251101 configmap]# kubectl delete -f dapi-test-
pod.yaml
```

运行这个Pod的输出是very。

我们也可以在ConfigMap值被映射的数据卷里控制路径。

```
[root@vlnx251101 configmap]# vim dapi-test-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh","-c","cat
/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key
  restartPolicy: Never

[root@vlnx251101 configmap]# kubectl create -f dapi-test-
pod.yaml
pod/dapi-test-pod created
```

```
[root@vlnx251101 configmap]# kubectl logs dapi-test-pod
very

[root@vlnx251101 configmakubectl delete -f dapi-test-
pod.yaml
pod "dapi-test-pod" deleted
```

运行这个Pod后的结果是very。