В предыдущих статьях мы установили и немного научились работать с кластером kubernetes. При этом я не рассмотрел вопрос с хранением данных, а вынес его в отдельную тему. Сегодня как раз хранение данных в кластере Kubernetes я и рассмотрю. Настраиваются эти хранилища в кластере с помощью абстракции под названием Persistent Volumes.

Содержание:

Цели статьи

1. Рассказать об устройстве внутренностей кластера Kubernetes для работы с дисками.

2. Описать основные абстракции, которые необходимо настроить, чтобы использовать файловые хранилища.
3. На примере NFS показать принцип работы Kubernetes с внешними хранилищами.
4. Рассказать, как в кластере Kubernetes использовать локальные диски нод.

Работа с дисками в Kubernetes

Работа с дисковыми томами в Kubernetes проходит по следующей схеме:

1. Вы описываете типы файловых хранилищ с помощью Storage Classes и Persistent Volumes. Они могут быть совершенно разными от локальных дисков до внешних кластерных систем и дисковых полок.
2. Для подключения диска к поду вы создаете Persistent Volume Claim, в котором описываете потребности пода в доступе к хранилищу — объем, тип и т.д. На основе этого запроса используются либо готовые PV, либо создаются под конкретный запрос автоматически с помощью PV Provisioners.
3. В описании пода добавляете информацию о Persistent Volume Claim, который он будет использовать в своей работе.

Что такое Persistent Volumes

Условно Persistent Volumes можно считать аналогом нод в самом кластере Kubernetes. Допустим, у вас есть несколько разных хранилищ. К примеру, одно быстрое на SSD, а другое медленное на HDD. Вы можете создать 2 Persistent Volumes в соответствии с этим, а затем выделять подам место в этих томах. Кубернетис поддерживает огромное количество подключаемых томов с помощью плагинов.

Вот список наиболее популярных:

- NFS
- iSCSI
- RBD
- CephFS
- Glusterfs
- FC (Fibre Channel)

Полный список можно посмотреть в документации.

Persistent Volume Claim

PersistentVolumeClaim (PVC) есть не что иное как запрос к Persistent Volumes на хранение от пользователя. Это аналог создания Pod на ноде. Поды могут запрашивать определенные ресурсы ноды, то же самое делает и PVC. Основные параметры запроса:

- объем pvc
- тип доступа

Типы доступа у PVC могут быть следующие:

- ReadWriteOnce – том может быть смонтирован на чтение и запись к одному поду.
- ReadOnlyMany – том может быть смонтирован на много подов в режиме только чтения.
- ReadWriteMany – том может быть смонтирован к множеству подов в режиме чтения и записи.

Ограничение на тип доступа может налагаться типом самого хранилища. К примеру, хранилище RBD или iSCSI не поддерживают доступ в режиме ReadWriteMany.

Один PV может использоваться только одним PVC. К примеру, если у вас есть 3 PV по 50, 100 и 150 гб. Приходят 3 PVC каждый по 50 гб. Первому будет отдано PV на 50 гб, второму на 100 гб, третьему на 150 гб, несмотря на то, что второму и третьему было бы достаточно и 50 гб. Но если PV на 50 гб нет, то будет отдано на 100 или 150, так как они тоже удовлетворяют запросу. И больше никто с PV на 150 гб работать не сможет, несмотря на то, что там еще есть свободное место.

Из-за этого нюанса, нужно внимательно следить за доступными томами и запросами к ним. В основном это делается не вручную, а автоматически с помощью **PV Provisioners**. В момент запроса pvc через api кластера автоматически формируется запрос к storage provider. На основе этого запроса хранилище создает необходимый PV и он подключается к поду в соответствии с запросом.

Storage Classes

StorageClass позволяет описать классы хранения, которые предлагают хранилища. Например, они могут отличаться по скорости, по политикам бэкапа, либо какими-то еще произвольными политиками. Каждый StorageClass содержит поля provisioner, parameters и reclaimPolicy, которые используются, чтобы динамически создавать PersistentVolume.

Можно создать дефолтный StorageClass для тех PVC, которые его вообще не указывают. Так же storage class хранит параметры подключения к реальному хранилищу. PVC используют эти параметры для подключения хранилища к подам.

> Важный нюанс. PVC зависим от namespace. Если у вас SC будет с секретом, то этот секрет должен быть в том же namespace, что и PVC, с которого будет запрос на подключение.

Работа с PV в кластере

Проверим, есть ли у нас какие-то PV в кластере.

```
# kubectl get pv
No resources found.
```

Ничего нет. Попробуем создать и применить какой-нибудь PVC и проверим, что произойдет. Для примера создаем pvc.yaml следующего содержания.

```
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: fileshare
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Просим выделить нам 1 гб пространства с типом доступа ReadWriteMany. Применяем yaml.

```
# kubectl apply -f pvc.yaml
```

Ждем немного и проверяем статус pvc.

```
# kubectl get pvc
NAME        STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
fileshare   Pending                                                     3m33s
```

Статус pending. Проверяем почему так.

```
# kubectl get pvc
```

```
no persistent volumes available for this claim and no storage class is set
```

Все понятно. Запрос находится в ожидании, так как у нас в кластере нет ни одного PV, который бы удовлетворял запросу. Давайте это исправим и добавим одно хранилище для примера на основе внешнего nfs сервера.

NFS хранилище в качестве Persistence Volume

Для простоты я взял в качестве хранилища PV nfs сервер. Для него не существует встроенного Provisioner, как к примеру, для Ceph. Возможно в будущем я рассмотрю более сложный пример с применением хранилища на основе ceph. Но перед этим надо будет написать статью про установку и работу с ceph.

Создаем yaml файл pv-nfs.yaml с описанием Persistence Volume на основе NFS.

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: my-nfs-share
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    server: 10.1.4.39
    path: /mnt/nfs
```

Reclaim Policy

persistentVolumeReclaimPolicy — что будет происходить с pv после удаления pvc. Могут быть 3 варианта:

1. Retain — pv удален не будет.
2. Recycle — pv будет очищен.
3. Delete — pv будет удален.

Так как у нас нет Provisioner для nfs, удалять автоматически pv не получится. Так что у нас только 2 варианта — либо оставлять данные (retain), либо очищать том (recycle).

Все остальное и так понятно, описывать не буду. Добавляем описанный pv в кластер kubernetes.

```
# kubectl apply -f pv-nfs.yml
```

Проверяем список pv и pvc

```
# kubectl get pv
# kebectl get pvc
```

Мы просили в pvc только 1 Гб хранилища, но в pv было только хранилище с 10 Гб и оно было выдано. Как я и говорил раньше. Так что в случае ручного создания PV и PVC нужно самим следить за размером PV.

Подключаем хранилище к поду

Теперь подключим наш PVC в виде volume к какому-нибудь поду. Опишем его в конфиге pod-with-nfs.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-with-nfs
spec:
  containers:
    - name: app
      image: alpine
      volumeMounts:
      - name: data
        mountPath: /mnt/nfs
      command: ["/bin/sh"]
      args: ["-c", "sleep 500000"]
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: fileshare
```

Применяем.

```
# kubectl apply -f pod-with-nfs.yaml
```

Затем проверяйте статус запущенного пода.

```
# kubectl get pod
NAME           READY   STATUS              RESTARTS   AGE
pod-with-nfs   0/1     ContainerCreating   0          80s
```

У меня он не стартовал. Я решил посмотреть из-за чего.

```
# kubectl describe pod pod-with-nfsУвидел в самом конце в разделе Messages ошибку:
```

```
Events:
  Type     Reason       Age    From                Message
  ----     ------       ----   ----                -------
  Warning  FailedMount  4m10s  kubelet, kub-node-1  MountVolume.SetUp failed for volume "my-nfs-share" : mount failed: exit
status 32
Mounting command: systemd-run
Mounting arguments: --description=Kubernetes transient mount for
/var/lib/kubelet/pods/3918714a-87eb-4e66-81cb-0f311461f47d/volumes/kubernetes.io~nfs/my-nfs-share --scope -- mount -t nfs
10.1.4.39:/mnt/nfs /var/lib/kubelet/pods/3918714a-87eb-4e66-81cb-0f311461f47d/volumes/kubernetes.io~nfs/my-nfs-share
Output: Running scope as unit run-980498.scope.
mount: wrong fs type, bad option, bad superblock on 10.1.4.39:/mnt/nfs,
       missing codepage or helper program, or other error
       (for several filesystems (e.g. nfs, cifs) you might
       need a /sbin/mount. helper program)

       In some cases useful info is found in syslog - try
       dmesg | tail or so.
```

Я сразу понял в чем проблема. На ноде, где стартовал под, не были установлены утилиты для работы с nfs. В моем случае система была Centos и я их установил.

```
# yum install nfs-utils nfs-utils-lib
```

После этого в течении минуты pod запустился. Зайдем в его консоль и посмотрим, подмонтировалась ли nfs шара.

```
# kubectl exec -it pod-with-nfs sh
```

Как видите, все в порядке. Попробуем теперь что-то записать в шару. Снова заходим на под и создаем текстовый файл.

```
# kubectl exec -it pod-with-nfs sh
# echo "test text" >> /mnt/nfs/test.txt
```

Теперь запустим еще один под и подключим ему этот же pvc. Для этого просто немного изменим предыдущий под, обозвав его pod-with-nfs2.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-with-nfs2
spec:
  containers:
    - name: app
      image: alpine
      volumeMounts:
      - name: data
        mountPath: /mnt/nfs
      command: ["/bin/sh"]
      args: ["-c", "sleep 500000"]
  volumes:
  - name: data
    persistentVolumeClaim:
```

```
      claimName: fileshare
```

Запускаем под и заходим в него.

```
# kubectl apply -f pod-with-nfs2.yaml
# kubectl exec -it pod-with-nfs2 sh
# cat /mnt/nfs/test.txt
test text
```

Все в порядке, файл на месте. С внешними хранилищем в Kubernetes закончили. Расскажу дальше, как можно работать с локальными дисками нод, если вы хотите их так же пустить в работу.

Local Persistent Volume — хранилище Kubernetes на локальных дисках

Локальные тома, как и nfs, не имеют встроенного провизионера, так что нарезать их можно только вручную, создавая PV. Есть внешний provisioner — https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner, но лично я его не проверял.

Создаем SC *sc-local.yaml*.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Создаем вручную PV *pv-local-node-1.yaml*, который будет располагаться на kub-node-1 в */mnt/local-storage*. Эту директорию необходимо вручную создать на сервере.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-local-node-1
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /mnt/local-storage
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - kub-node-1
```

Создаем PVC *pvc-local.yaml* для запроса сторейджа, который передадим поду.

```
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-volume
spec:
  storageClassName: "local-storage"
```

```
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

И в завершении создадим тестовый POD *pod-with-pvc-local.yaml* для проверки работы local storage.

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-with-pvc-local
spec:
  containers:
    - name: app
      image: alpine
      volumeMounts:
      - name: data
        mountPath: /mnt
      command: ["/bin/sh"]
      args: ["-c", "sleep 500000"]
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: local-volume
```

Применяем все вышеперечисленное в строго определенном порядке:

1. SC
2. PV
3. PVC
4. POD

```
# kubectl apply -f sc-local.yaml
# kubectl apply -f pv-local-node-1.yaml
# kubectl apply -f pvc-local.yaml
# kubectl apply -f pod-with-pvc-local.yaml
```

После этого посмотрите статус всех запущенных абстракций.

Проверим, что Local Persistent Volume правильно работает. Зайдем в под и создадим тестовый файл.

```
# kubectl exec -it pod-with-pvc-local sh
# echo "local srorage test" >> /mnt/local.txt
```

Теперь идем на сервер kub-node-1 и проверяем файл.

```
[root@kub-node-1 local-storage]# cat /mnt/local-storage/local.txt
local srorage test
```

Все в порядке. Файл создан.

Если у вас возникли какие-то проблемы с POD, то в PVC будет ошибка:

```
waiting for first consumer to be created before binding
```

И в это же время в поде:

```
0/6 nodes are available: 1 node(s) didn't find available persistent volumes to bind, 5 node(s) had taints that the pod
didn't tolerate.
```

Возникли ошибки из-за того, что в описании PV я ошибся в названии сервера, где будет доступен local storage. В итоге pod запускался, проверял pvc, а pvc смотрел на pv и видел, что адрес ноды с pv не соответствует имени ноды, где будет запущен pod. В итоге все висело в ожидании разрешения этих несоответсвий.

С локальным хранилищем разобрались, можно использовать в работе.

Заключение

> Не понравилась статья и хочешь научить меня администрировать? Пожалуйста, я люблю учиться. Комментарии в твоем распоряжении. Расскажи, как сделать правильно!

На наглядных примерах я показал, как можно работать с данными в кластере Kubernetes. Можно использовать как отказоустойчивые внешние хранилища для размещения постоянных данных. А можно сырые диски на серверах и использовать кластер для работы какой-то распределенной базы данных или сервиса, которые сами следят за количеством копий данных и распределяют их по нодам.

В последнем случае можно использовать самое обычное железо для кластера Кубернетиса, без рейд контроллеров и дублирования дорогих и быстрых SSD дисков. Необходимо только следить за количеством реплик приложения на нодах, чтобы их количество было достаточно для стабильной работы системы.

Два этих подхода можно комбинировать. Внешние хранилища для холодных данных и внутренние SSD диски для кэшей.

- Реальные ситуации и задачи;
- Лучшее из международных программ.

Помогла статья? Есть возможность отблагодарить автора