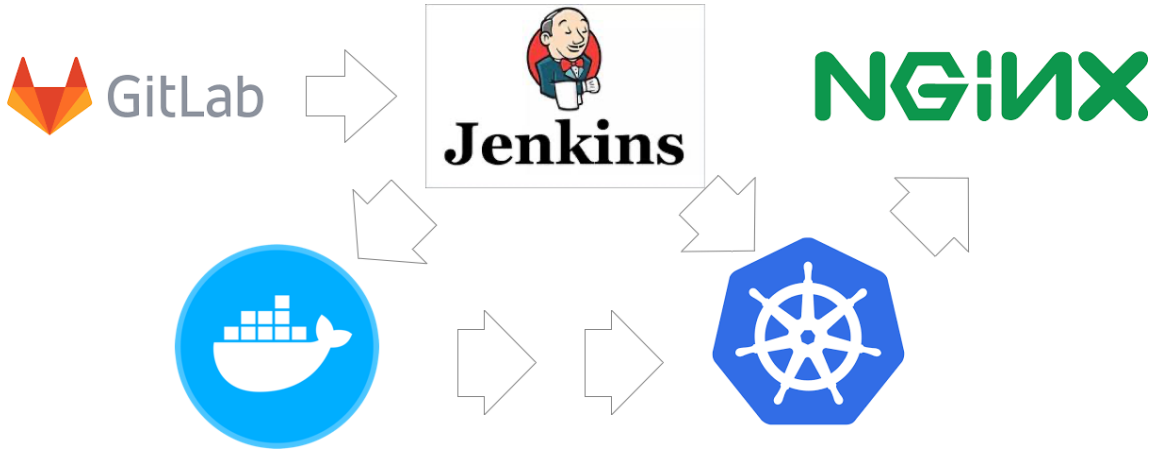
讲师(老司机)

容器虚拟化技术和自动化部署



k8s运维篇-常用软件安装

dashboard

Dashboard简介

Dashboard 是基于网页的 Kubernetes 用户界面。您可以使用 Dashboard 将容器应用部署到 Kubernetes 集群中，也可以对容器应用排错，还能管理集群本身及其附属资源。您可以使用 Dashboard 获取运行在集群中的应用的概览信息，也可以创建或者修改 Kubernetes 资源（如 Deployment，Job，DaemonSet 等等）。例如，可以对 Deployment 实现弹性伸缩、发起滚动升级、重启 Pod 或者使用向导创建新的应用。

官网地址

```
https://github.com/kubernetes/dashboard

下载配置文件
https://github.com/kubernetes/dashboard/blob/v2.0.3/aio/deploy/recommended.yaml
```

安装镜像

```
kubernetsui下边的镜像不需要科学上网

docker pull kubernetesui/dashboard:v2.0.3
docker pull kubernetesui/metrics-scraper:v1.0.4
```

修改配置文件

控制器部分

```
179行左右

    containers:
        - name: kubernetes-dashboard
          image: kubernetesui/dashboard:v2.0.3
          imagePullPolicy: IfNotPresent

262行左右。新增下载策略
    containers:
        - name: dashboard-metrics-scraper
          image: kubernetesui/metrics-scraper:v1.0.4
          imagePullPolicy: IfNotPresent
```

service部分

默认Dashboard只能集群内部访问，修改Service为NodePort类型，暴露到外部访问。找到Services配置。在配置文件上边。增加type:NodePort和 nodePort:30100端口

```yaml
kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  ports:
    - port: 443
      targetPort: 8443
      nodePort: 30100
  type: NodePort
  selector:
    k8s-app: kubernetes-dashboard
```

管理 Service Accounts

这是一篇针对service accounts（服务账户）的集群管理员指南。 它呈现了 User Guide to Service Accounts中的信息。

对授权和用户账户的支持已在规划中，当前并不完备，为了更好地描述 service accounts，有时这些不完善的特性也会被提及。

用户账户与服务账户

Kubernetes 区分用户账户和服务账户的概念主要基于以下原因：

- User(用户账户)是针对人而言的。 service accounts(服务账户)是针对运行在 pod 中的进程而言的。
- 用户账户是全局性的。 其名称在集群各 namespace 中都是全局唯一的，未来的用户资源不会做 namespace 隔离， 服务账户是 namespace 隔离的。
- 通常情况下，集群的用户账户可能会从企业数据库进行同步，其创建需要特殊权限，并且涉及到复杂的业务流程。 服务账户创建的目的是为了更轻量，允许集群用户为了具体的任务创建服务账户 (即权限最小化原则)。
- 对人员和服务账户审计所考虑的因素可能不同。
- 针对复杂系统的配置可能包含系统组件相关的各种服务账户的定义。 因为服务账户可以定制化地创建，并且有 namespace 级别的名称，这种配置是很轻量的。

服务账户的自动化

三个独立组件协作完成服务账户相关的自动化 :

- 服务账户准入控制器（Service account admission controller）
- Token 控制器（Token controller）
- 服务账户控制器（Service account controller）

服务账户准入控制器

对 pod 的改动通过一个被称为 Admission Controller 的插件来实现。它是 apiserver 的一部分。 当 pod 被创建或更新时，它会同步地修改 pod。 当该插件处于激活状态 (在大多数发行版中都是默认的)，当 pod 被创建或更新时它会进行以下动作：

1. 如果该 pod 没有 ServiceAccount 设置，将其 ServiceAccount 设为 default。
2. 保证 pod 所关联的 ServiceAccount 存在，否则拒绝该 pod。
3. 如果 pod 不包含 `ImagePullSecrets` 设置，那么 将 ServiceAccount 中的 ImagePullSecrets 信息添加到 pod 中。
4. 将一个包含用于 API 访问的 token 的 volume 添加到 pod 中。
5. 将挂载于 `/var/run/secrets/kubernetes.io/serviceaccount` 的 volumeSource 添加到 pod 下的每个容器中。

Token 管理器

Token 管理器是 controller-manager 的一部分。 以异步的形式工作：

- 检测服务账户的创建，并且创建相应的 Secret 以支持 API 访问。
- 检测服务账户的删除，并且删除所有相应的服务账户 Token Secret。
- 检测 Secret 的增加，保证相应的服务账户存在，如有需要，为 Secret 增加 token。
- 检测 Secret 的删除，如有需要，从相应的服务账户中移除引用。

你需要通过 `--service-account-private-key-file` 参数项传入一个服务账户私钥文件至 Token 管理器。 私钥用于为生成的服务账户 token 签名。 同样地，你需要通过 `--service-account-key-file` 参数将对应的公钥传入 kube-apiserver。 公钥用于认证过程中的 token 校验。

服务账户管理器

服务账户管理器管理各命名空间下的服务账户，并且保证每个活跃的命名空间下存在一个名为 "default" 的服务账户

RBAC

使用 RBAC 鉴权。基于角色（Role）的访问控制（RBAC）是一种基于企业中用户的角色来调节控制对计算机或网络资源的访问方法。`RBAC` 使用 `rbac.authorization.k8s.io` API 组 来驱动鉴权操作，允许管理员通过 Kubernetes API 动态配置策略。

在 1.8 版本中，RBAC 模式是稳定的并通过 `rbac.authorization.k8s.io/v1` API 提供支持。

要启用 RBAC，在启动 API 服务器时添加 `--authorization-mode=RBAC` 参数。

Role 和 ClusterRole

在 RBAC API 中，一个角色包含一组相关权限的规则。权限是纯粹累加的（不存在拒绝某操作的规则）。 角色可以用 Role 来定义到某个命名空间上， 或者用 ClusterRole 来定义到整个集群作用域。

一个 Role 只可以用来对某一命名空间中的资源赋予访问权限。 下面的 Role 示例定义到名称为 "default" 的命名空间，可以用来授予对该命名空间中的 Pods 的读取权限：

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" 指定核心 API 组
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRole 可以授予的权限和 Role 相同， 但是因为 ClusterRole 属于集群范围，所以它也可以授予以下访问权限：

集群范围资源 （比如 nodes）
非资源端点（比如 `/healthz`）
跨命名空间访问的有名字空间作用域的资源（如 Pods），比如运行命令 `kubectl get pods --all-namespaces` 时需要此能力
下面的 `ClusterRole` 示例可用来对某特定命名空间下的 Secrets 的读取操作授权， 或者跨所有命名空间执行授权（取决于它是如何绑定的）：

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # 此处的 "namespace" 被省略掉是因为 ClusterRoles 是没有命名空间的。
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RoleBinding 和 ClusterRoleBinding

角色绑定（`RoleBinding`）是将角色中定义的权限赋予一个或者一组用户。 它包含若干主体（用户，组和服务账户）的列表和对这些主体所获得的角色的引用。 可以使用 `RoleBinding` 在指定的命名空间中执行授权， 或者在集群范围的命名空间使用 `ClusterRoleBinding` 来执行授权。

一个 `RoleBinding` 可以引用同一的命名空间中的 `Role` 。 下面的例子 `RoleBinding` 将 `pod-reader` 角色授予在 "default" 命名空间中的用户 "jane"； 这样，用户 "jane" 就具有了读取 "default" 命名空间中 pods 的权限。

roleRef 里的内容决定了实际创建绑定的方法。kind 可以是 `Role` 或 `ClusterRole`， name 将引用你要指定的 `Role` 或 `ClusterRole` 的名称。在下面的例子中，角色绑定使用 roleRef 将用户 "jane" 绑定到前文创建的角色 `Role`，其名称是 `pod-reader`。

```
apiVersion: rbac.authorization.k8s.io/v1
# 此角色绑定使得用户 "jane" 能够读取 "default" 命名空间中的 Pods
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # 这里的名称必须与你想要绑定的 Role 或 ClusterRole 名称一致
  apiGroup: rbac.authorization.k8s.io
```

`RoleBinding` 也可以引用 `ClusterRole`，对 `ClusterRole` 所定义的、位于 `RoleBinding` 命名空间内的资源授权。 这可以允许管理者在 整个集群中定义一组通用的角色，然后在多个命名空间中重用它们。

Dashboard新增用户

可以选择使用资源文件方式或者命令行方式为dashboard新建具有管理集群角色的用户。

使用资源文件方式新增用户

在配置文件下边增加用户及给用户授予集群管理员角色

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: dashboard-admin
  namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: dashboard-admin-cluster-role
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
```

```
    name: dashboard-admin
    namespace: kubernetes-dashboard
```

使用命令行方式新增用户

创建service account并绑定默认cluster-admin管理员集群角色：

```
创建用户
kubectl create serviceaccount dashboard-admin -n kube-system

用户授权
kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin --serviceaccount=kube-system:dashboard-admin


kubectl delete  clusterrolebinding dashboard-admin -n kube-system
删除用户
kubectl delete sa dashboard-admin -n kube-system

```

部署dashboard

部署完dashboard服务，可以选在使用token认证方式登录或者kubeconfig认证方式登录dashboard。

```

kubectl apply -f .

kubectl get pods -n kubernetes-dashboard -o wide


kubectl get svc -n kubernetes-dashboard

kubectl delete -f .
```

token认证方式

分步查看token信息

```
1.根据命名空间找到我们创建的用户
kubectl get sa -n kubernetes-dashboard

2.查看我们创建用户的详细信息。找到token属性对应的secret值
kubectl describe sa dashboard-admin -n kubernetes-dashboard
kubectl describe secrets dashboard-admin-token-9pl4b -n kubernetes-dashboard

3.或者是根据命名空间查找secrets。获得dashboard-admin用户的secret。
kubectl get secrets -n kubernetes-dashboard
kubectl describe secrets dashboard-admin-token-9pl4b -n kubernetes-dashboard
```

快速查看token信息

```
kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret | grep dashboard-admin | awk '{print $1}')
```

浏览器访问

```
注意：是https方式访问
https://192.168.198.156:30100/
```

kubeConfig认证方式

了解即可。

```
以下命令可以一起执行。可以更改dashboard-admin.conf的生成目录。关键点还是要首先或者dashboard-admin用户的secret值。

DASH_TOCKEN=$(kubectl get secret -n kubernetes-dashboard dashboard-admin-token-jdvkg -o jsonpath={.data.token}|base64 -d)

kubectl config set-cluster kubernetes --server=192.168.198.156:6443 --kubeconfig=/root/dashboard-admin.conf

kubectl config set-credentials dashboard-admin --token=$DASH_TOCKEN --kubeconfig=/root/dashboard-admin.conf

kubectl config set-context dashboard-admin@kubernetes --cluster=kubernetes --user=dashboard-admin --kubeconfig=/root/dashboard-admin.conf

kubectl config use-context dashboard-admin@kubernetes --kubeconfig=/root/dashboard-admin.conf

将生成的dashboard-admin.conf上传到windows系统中。浏览器选择dashboard-admin.conf文件即可用于登录dashboard
```

使用StatefulSet创建Zookeeper集群

这是k8s官方提供的案例。不推荐学员练习。因为要包含4个节点。学生们电脑内存压力大。k8s官网提供镜像国内无法下载。老司机给大家演示一下即可。

运行 ZooKeeper， 一个 CP 分布式系统。本教程展示了在 Kubernetes 上使用 PodDisruptionBudgets 和 PodAntiAffinity 特性运行 Apache Zookeeper。

需要一个至少包含四个节点的集群，每个节点至少 2 CPUs 和 4 GiB 内存。在本教程中你应该使用一个独占的集群，或者保证你造成的干扰不会影响其它租户。

教程目标

在学习本教程后，你将熟悉下列内容。

- 如何使用 StatefulSet 部署一个 ZooKeeper ensemble。
- 如何使用 ConfigMaps 一致性配置 ensemble。
- 如何在 ensemble 中 分布 ZooKeeper 服务的部署。
- 如何在计划维护中使用 PodDisruptionBudgets 确保服务可用性。

安装镜像

```
k8s官方提供进行国内无法下载，使用国内镜像地址下载。需要修改官网默认给的yaml文件，去掉官网镜像，使用我们下载的镜像。
docker pull mirrorgooglecontainers/kubernetes-zookeeper:1.0-3.4.10
```

资源文件清单

```yaml
apiVersion: v1
kind: Service
metadata:
  name: zk-hs
  labels:
    app: zk
spec:
  ports:
    - port: 2888
      name: server
    - port: 3888
      name: leader-election
  clusterIP: None
```

```yaml
  selector:
    app: zk
---
apiVersion: v1
kind: Service
metadata:
  name: zk-cs
  labels:
    app: zk
spec:
  ports:
    - port: 2181
      name: client
  selector:
    app: zk
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  selector:
    matchLabels:
      app: zk
  maxUnavailable: 1
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: zk
spec:
  selector:
    matchLabels:
      app: zk
  serviceName: zk-hs
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: zk
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                      - zk
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: kubernetes-zookeeper
          imagePullPolicy: IfNotPresent
          image: "mirrorgooglecontainers/kubernetes-zookeeper:1.0-3.4.10"
```

```
          resources:
            requests:
              memory: "1Gi"
              cpu: "0.5"
          ports:
            - containerPort: 2181
              name: client
            - containerPort: 2888
              name: server
            - containerPort: 3888
              name: leader-election
          command:
            - sh
            - -c
            - "start-zookeeper \
          --servers=3 \
          --data_dir=/var/lib/zookeeper/data \
          --data_log_dir=/var/lib/zookeeper/data/log \
          --conf_dir=/opt/zookeeper/conf \
          --client_port=2181 \
          --election_port=3888 \
          --server_port=2888 \
          --tick_time=2000 \
          --init_limit=10 \
          --sync_limit=5 \
          --heap=512M \
          --max_client_cnxns=60 \
          --snap_retain_count=3 \
          --purge_interval=12 \
          --max_session_timeout=40000 \
          --min_session_timeout=4000 \
          --log_level=INFO"
          readinessProbe:
            exec:
              command:
                - sh
                - -c
                - "zookeeper-ready 2181"
            initialDelaySeconds: 10
            timeoutSeconds: 5
          livenessProbe:
            exec:
              command:
                - sh
                - -c
                - "zookeeper-ready 2181"
            initialDelaySeconds: 10
            timeoutSeconds: 5
          volumeMounts:
            - name: datadir
              mountPath: /var/lib/zookeeper
      securityContext:
        runAsUser: 1000
        fsGroup: 1000
      volumes:
      - name: datadir
        emptyDir: {}
```

部署zookeeper

```
kubectl apply -f .

监控zookeeper启动过程
kubectl get pods -o wide -w

kubectl delete -f .
```

statefuleSet

简介

前边我们讲了deployment来管理pod容器的副本数量，如果挂掉之后容器再次启动就可以了，但是如果要是启动的是mysql集群、zookeeper集群、etcd这种集群，里面都有id号，这种有关联的，如果一旦挂掉之后，在启动之后呢，集群id是否会变化呢？答案是肯定会变的。

那有没有另外的一种控制器模式吗？当然k8s会提供的--【statefulset】

那什么场景需要使用StatefulSet呢？官方给出的建议是，如果你部署的应用满足以下一个或多个部署需求，则建议使用StatefulSet。

- 稳定的、唯一的网络标识。
- 稳定的、持久的存储。
- 有序的、优雅的部署和伸缩。
- 有序的、优雅的删除和停止。
- 有序的、自动的滚动更新。

statefulset和deployment的区别：

特性	Deployment	StatefulSet
是否暴露到外网	可以	一般不
请求面向的对象	serviceName	指定pod的域名
灵活性	只能通过service/serviceIp访问到k8s自动转发的pod	可以访问任意一个自定义的pod
易用性	只需要关心Service的信息即可	需要知道要访问的pod启动的名称、headlessService名称
PV/PVC绑定关系的稳定性（多replicas）	（pod挂掉后重启）无法保证初始的绑定关系	可以保证
pod名称稳定性	不稳定，因为是通过template创建，每次为了避免重复都会后缀一个随机数	稳定，每次都一样
启动顺序（多replicas）	随机启动，如果pod宕掉重启，会自动分配一个node重新启动	pod按 app-0、app-1...app-（n-1），如果pod宕掉重启，还会在之前的node上重新启动
停止顺序（多replicas）	随机停止	倒序停止
集群内部服务发现	只能通过service访问到随机的pod	可以打通pod之间的通信（主要是被发现）
性能开销	无需维护pod与node、pod与PVC 等关系	比deployment类型需要维护额外的关系信息

分类

K8s有状态应用部署分为两步：

1. Headless Service： Headless Service 其实和service差不多，只不过定义的这个叫无头服务，它们之间唯一的区别就是将Cluster ip 设置为了none，不会帮你配置ip
2. StatefulSet：需要在pod模板中定义servicename。spec. serviceName。

Headless Service案例-不需要创建演示

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

部署服务

```
kubectl create -f headless-svc.yaml

kubectl get svc

怎么去访问？我们给它定义一个标识。创建完之后会有这个标识符，它会使用这个DNS来解析这个名称，来相互的访问，headless就不会通过ip去访问了，必须通过名称去访问。
```

statefulSet案例-不需要创建演示

```yaml
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: nginx-statefulset
  namespace: default
spec:
  serviceName: my-service
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

总结

- **Pod会被顺序部署和顺序终结**：StatefulSet中的各个 Pod会被顺序地创建出来，每个Pod都有一个唯一的ID，在创建后续 Pod 之前，首先要等前面的 Pod 运行成功并进入到就绪状态。删除会销毁 StatefulSet 中的每个 Pod，并且按照创建顺序的反序来执行，只有在成功终结后面一个之后，才会继续下一个删除操作。
- **Pod具有唯一网络名称**：Pod具有唯一的名称，而且在重启后会保持不变。通过Headless服务，基于主机名，每个 Pod 都有独立的网络地址，这个网域由一个Headless 服务所控制。这样每个Pod会保持稳定的唯一的域名，使得集群就不会将重新创建出的Pod作为新成员。
- **Pod能有稳定的持久存储**：StatefulSet中的每个Pod可以有其自己独立的PersistentVolumeClaim对象。即使Pod被重新调度到其它节点上以后，原有的持久磁盘也会被挂载到该Pod。
- **Pod能被通过Headless服务访问到**：客户端可以通过服务的域名连接到任意Pod。

动态PV

安装镜像

```
由于quay.io仓库部分镜像国内无法下载，所以替换为其他镜像地址
docker pull vbouchaud/nfs-client-provisioner:v3.1.1
```

nfs4服务端配置

```
mkdir -p /nfs/data/
chmod 777 /nfs/data/

yum install -y nfs-utils rpcbind

更改归属组与用户
chown nfsnobody /nfs/data/

vi /etc/exports

/nfs/data *(rw,fsid=0,sync,no_wdelay,insecure_locks,no_root_squash)


为了方便接下来两个实验，提前建立2个共享子目录。
mkdir -p /nfs/data/mariadb
mkdir -p /nfs/data/nginx


systemctl start rpcbind
systemctl start nfs

设置开启启动
systemctl enable rpcbind
systemctl enable nfs
```

nfs的storageClass配置

rbac

nfsdynamic/nfsrbac.yml。每次配置文件，只需要调整ClusterRoleBinding、RoleBinding的namespace值，如果服务是部署在默认的namespace中，配置文件不需要调整。

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: nfs-client-provisioner
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-client-provisioner-runner
```

```yaml
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "update", "patch"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    namespace: default      #替换成要部署NFS Provisioner的namespace
roleRef:
  kind: ClusterRole
  name: nfs-client-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    namespace: default      #替换成要部署NFS Provisioner的namespace
roleRef:
  kind: Role
  name: leader-locking-nfs-client-provisioner
  apiGroup: rbac.authorization.k8s.io
```

storageClass

nfsdynamic/nfsstorage.yml

```yaml
kind: Deployment
apiVersion: apps/v1
```

```yaml
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    #设置升级策略为删除再创建(默认为滚动更新)
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          #由于quay.io仓库部分镜像国内无法下载，所以替换为其他镜像地址
          image: vbouchaud/nfs-client-provisioner:v3.1.1
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: nfs-client     #--- nfs-provisioner的名称，以后设置的storageclass要和这个保持一致
            - name: NFS_SERVER
              value: 192.168.198.156   #NFS服务器地址，与volumes.nfs.servers保持一致
            - name: NFS_PATH
              value: /nfs/data         #NFS服务共享目录地址，与volumes.nfs.path保持一致
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.198.156    #NFS服务器地址，与spec.containers.env.value保持一致
            path: /nfs/data            #NFS服务器目录，与spec.containers.env.value保持一致

---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" #设置为默认的storageclass
provisioner: nfs-client                                 #动态卷分配者名称，必须和创建的"provisioner"变量中设置的name一致
parameters:
  archiveOnDelete: "true"                               #设置为"false"时删除PVC不会保留数据,"true"则保留数据
mountOptions:
  - hard                                                #指定为硬挂载方式
```

```
  - nfsvers=4                                                    #指定NFS版本，这个需要根据 NFS Server 版本号设置
```

测试pvc

nfsdynamic/nfstestpvc.yml

用于测试nfs动态pv是否成功。

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-pvc
spec:
  storageClassName: nfs-storage  #需要与上面创建的storageclass的名称一致
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
```

部署nfs测试服务

```
kubectl apply -f .

查看storageClass
kubectl get storageclasses.storage.k8s.io || kubectl get sc

查看mariadb服务
kubectl get svc

查看pv pvc

查看statefulSet
kubectl get sts

查看mariadb、storageClass的pods
kubectl get pods
```

删除服务

pv是动态生成，通过查看pv状态，发现pv不会自动回收。

```
删除mariadb服务
kubectl delete -f .
```

```
查看动态nfs的pv状态。发现pv的status状态是：Released
kubectl get pv

编译pv的配置文件
kubectl edit pv pvc-59fb2735-9681-426a-8805-8c94685a07e3

将spec.claimRef属性下的所有内容全部删除
claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: test-pvc
    namespace: default
    resourceVersion: "162046"
    uid: 59fb2735-9681-426a-8805-8c94685a07e3

再次查看pv状态。发现pv的status状态是：Available
kubectl get pv

删除pv
kubectl delete pv pvc-59fb2735-9681-426a-8805-8c94685a07e3

删除共享目录动态pv的目录
rm -rf pvc-59fb2735-9681-426a-8805-8c94685a07e3
```

动态pv案例一

部署3个副本的nginx服务。主要学习 **volumeClaimTemplate** 属性。

statefulset组成

statefulSet的三个组成部分：

- `Headless Service`：名为nginx，用来定义Pod网络标识(DNS domain)。
- `StatefulSet`：定义具体应用，名为Nginx，有三个Pod副本，并为每个Pod定义了一个域名。
- `volumeClaimTemplates`： 存储卷申请模板，创建PVC，指定pvc名称大小，将自动创建pvc，且pvc必须由存储类供应。

为什么需要 headless service 无头服务?

在用Deployment时，每一个Pod名称是没有顺序的，是随机字符串，因此是Pod名称是无序的，但是在statefulset中要求必须是有序 ，每一个pod不能被随意取代，pod重建后pod名称还是一样的。而pod IP是变化的，所以是以Pod名称来识别。pod名称是pod唯一性的标识符，必须持久稳定有效。这时候要用到无头服务，它可以给每个Pod一个唯一的名称 。

为什么需要volumeClaimTemplate?

对于有状态的副本集都会用到持久存储，对于分布式系统来讲，它的最大特点是数据是不一样的，所以各个节点不能使用同一存储卷，每个节点有自己的专用存储，但是如果在Deployment中的Pod template里定义的存储卷，是所有副本集共用一个存储卷，数据是相同的，因为是基于模板来的 ，而statefulset中每个Pod都要自己的专有存储卷，所以statefulset的存储卷就不能再用Pod模板来创建了，于是statefulSet使用volumeClaimTemplate，称为卷申请模板，它会为每个Pod生成不同的pvc，并绑定pv， 从而实现各pod有专用存储。这就是为什么要用volumeClaimTemplate的原因。

nfs服务

rbac

nfsnginx/nfsrbac.yml。与前文保持一致。

```yaml
kind: ServiceAccount
apiVersion: v1
metadata:
  name: nfs-client-provisioner
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-client-provisioner-runner
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "update", "patch"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    namespace: default      #替换成要部署NFS Provisioner的namespace
roleRef:
  kind: ClusterRole
  name: nfs-client-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
```

```
    namespace: default      #替换成要部署NFS Provisioner的namespace
roleRef:
  kind: Role
  name: leader-locking-nfs-client-provisioner
  apiGroup: rbac.authorization.k8s.io
```

storageClass

nfsnginx/nfsnginxstorage.yml。与前文介绍类似，注意修改storageClass的名称

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    #设置升级策略为删除再创建(默认为滚动更新)
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          #由于quay.io仓库部分镜像国内无法下载，所以替换为其他镜像地址
          image: vbouchaud/nfs-client-provisioner:v3.1.1
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: nfs-client-nginx  #nfs-provisioner的名称，以后设置的
storageclass要和这个保持一致
            - name: NFS_SERVER
              value: 192.168.198.156     #NFS服务器地址，与volumes.nfs.servers
保持一致
            - name: NFS_PATH
              value: /nginx             #NFS服务共享目录地址，与volumes.nfs.path
保持一致。使用NFS4版本进行多级目录挂载
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.198.156     #NFS服务器地址，与
spec.containers.env.value保持一致
            path: /nginx              #NFS服务器目录，与
spec.containers.env.value保持一致。使用NFS4版本进行多级目录挂载

```

```yaml
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage-nginx
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" #设置为默认的
storageclass
#动态卷分配者名称，必须和创建的"provisioner"变量中设置的name一致
provisioner: nfs-client-nginx
parameters:
  archiveOnDelete: "true"                                    #设置为"false"时删除
PVC不会保留数据,"true"则保留数据
mountOptions:
  - hard                                                     #指定为硬挂载方式
  - nfsvers=4                                                #指定NFS版本，这个需要
根据 NFS Server 版本号设置
```

nginx服务

如果定义多个副本。必须使用volumeClaimTemplate属性。如果定义1个副本。可以使用pod+pvc方式。

nfsnginx/nginxstatefulset.yml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginxdeployment
  labels:
    app: nginxdeployment
spec:
  replicas: 3
  serviceName: nginxsvc
  template:
    metadata:
      name: nginxdeployment
      labels:
        app: nginxdeployment
    spec:
      containers:
        - name: nginxdeployment
          image: nginx:1.17.10-alpine
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          volumeMounts:
            - mountPath: /usr/share/nginx/html/
              name: nginxvolume
      restartPolicy: Always
  volumeClaimTemplates:
    - metadata:
        name: nginxvolume
        annotations:
          volume.beta.kubernetes.io/storage-class: "nfs-storage-nginx"
```

```
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 2Gi
  selector:
    matchLabels:
      app: nginxdeployment
---
apiVersion: v1
kind: Service
metadata:
  name: nginxsvc
spec:
  selector:
    app: nginxdeployment
  ports:
    - port: 8080
  clusterIP: None
```

部署nginx服务

```
kubectl apply -f .

kubectl get pods -o wide

kubectl get pv

kubectl get pvc
```

动态pv案例二

部署mariadb数据库服务。

nfs服务

rbac

nfsmariadb/nfsrbac.yml。与前文保持一致。

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: nfs-client-provisioner
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
```

```yaml
metadata:
  name: nfs-client-provisioner-runner
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "update", "patch"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    namespace: default      #替换成要部署NFS Provisioner的namespace
roleRef:
  kind: ClusterRole
  name: nfs-client-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    namespace: default      #替换成要部署NFS Provisioner的namespace
roleRef:
  kind: Role
  name: leader-locking-nfs-client-provisioner
  apiGroup: rbac.authorization.k8s.io
```

storageClass

nfsmariadb/nfsmariadbstorage.yml。与前文介绍类似，注意修改storageClass的名称

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    #设置升级策略为删除再创建(默认为滚动更新)
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          #由于quay.io仓库部分镜像国内无法下载，所以替换为其他镜像地址
          image: vbouchaud/nfs-client-provisioner:v3.1.1
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: nfs-client-mariadb  #nfs-provisioner的名称，以后设置的storageclass要和这个保持一致
            - name: NFS_SERVER
              value: 192.168.198.156     #NFS服务器地址，与volumes.nfs.servers保持一致
            - name: NFS_PATH
              value: /mariadb            #NFS服务共享目录地址，与volumes.nfs.path保持一致。使用NFS4版本进行多级目录挂载
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.198.156    #NFS服务器地址，与spec.containers.env.value保持一致
            path: /mariadb             #NFS服务器目录，与spec.containers.env.value保持一致。使用NFS4版本进行多级目录挂载

---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-storage-mariadb
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" #设置为默认的storageclass
```

```
#动态卷分配者名称，必须和创建的"provisioner"变量中设置的name一致
provisioner: nfs-client-mariadb
parameters:
  archiveOnDelete: "true"                                    #设置为"false"时删除PVC不会保留数据,"true"则保留数据
mountOptions:
  - hard                                                     #指定为硬挂载方式
  - nfsvers=4                                                #指定NFS版本，这个需要根据 NFS Server 版本号设置
```

mariadb

pvc

nfsmariadb/mariadbpvc.yml。为后续容灾测试方便。单独创建pvc文件

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  # pvc名称
  name: mariadbpvc
spec:
  # 使用的存储类
  storageClassName: nfs-storage-mariadb
  # 读写权限
  accessModes:
    - ReadWriteMany
  # 定义容量
  resources:
    requests:
      storage: 5Gi
```

statefulset

nfsmariadb/mariadbstatefulset.yml

```
apiVersion: v1
kind: Service
metadata:
  name: mariadbsvc
spec:
  ports:
    - port: 3306
  # 创建service为无头服务，标识容器
  clusterIP: None
  selector:
    app: mariadb-public

---

apiVersion: apps/v1
kind: StatefulSet
# 名称
```

```yaml
metadata:
  name: mariadbsts
spec:
  # 指定service名称
  serviceName: "mariadbsvc"
  replicas: 1
  # 标签选择器
  template:
    metadata:
      labels:
        app: mariadb-public
    spec:
      # 镜像容器编辑
      containers:
        - name: mariadb
          image: mariadb:10.5.2
          env:
            # 创建数据库用户密码
            - name: MYSQL_ROOT_PASSWORD
              value: "admin"
            - name: TZ
              value: Asia/Shanghai
            # 创建数据库
            - name: MYSQL_DATABASE
              value: test
          args:
            - "--character-set-server=utf8mb4"
            - "--collation-server=utf8mb4_unicode_ci"
          # 启用端口
          ports:
            - containerPort: 3306
          # 数据卷
          volumeMounts:
            # 挂在容器目录
            - mountPath: "/var/lib/mysql"
              # 使用来源
              name: mariadb-data
      # 使用数据卷来源
      volumes:
        # 数据卷名称
        - name: mariadb-data
          # 指定数据卷动态供给
          persistentVolumeClaim:
            # pvc动态供给名称
            claimName: mariadbpvc
  selector:
    matchLabels:
      app: mariadb-public
```

部署mariadb服务

```
部署服务
kubectl apply -f .

查看storage
kubectl get storageclasses.storage.k8s.io

查看pv绑定情况
kubectl get pv

查看pvc绑定情况
kubectl get pvc

查看服务
kubectl get svc

查看statefulset
kubectl get sts


查看pod
kubectl get pods
```

测试mariadb

```
查看statefulset的服务名称
kubectl get svc

创建一个临时的pod用于访问statefulset。通过statefulset的服务名进行访问：-hmariadbsvc。
语法规则：--command -- mysql,mysql与--之间有空格。
kubectl run mariadb-test --image=mariadb:10.5.2 --restart=Never -it --rm --command -- mysql -hmariadbsvc -uroot -padmin


命令行方式查看database
show databases;

命令行方式创建database
create database lagou;

进入容器查看database目录
kubectl exec -it mariadbsts-0 sh
cd /var/lib/mysql
ls
exit

查看nfs共享目录，自动创建目录格式为：${namespace}-${pvcName}-${pvName}的文件夹
cd /nfs/data
ls
cd default-mariadbpvc-pvc-26c5785e-5703-4175-bc6a-3f9097d51d98/
ls
```

容灾测试

删除pod测试

```
删除pod进行测试
kubectl get pvc
kubectl delete pod

进入容器查看database目录
kubectl exec -it mariadbsts-0 sh
cd /var/lib/mysql
ls
exit

查看nfs共享目录中database保存情况
cd /nfs/data
ls
cd default-mariadbpvc-pvc-26c5785e-5703-4175-bc6a-3f9097d51d98/
ls

临时客户端查看
kubectl run mariadb-test --image=mariadb:10.5.2 --restart=Never -it --rm --command -- mysql -hmariadbsvc -uroot -padmin
show databases;
exit
```

删除statefuset

```
kubectl delete -f .
```