# 目录

# 通过Nginx-ingress实现灰度发布

在对服务进行版本发布或版本升级场景中，常会用到灰度发布、蓝绿发布等发布方式。本文将介绍如何在金山云容器服务中通过Nginx-ingress服务实现应用的灰度发布。

## 前提条件

集群内完成nginx-ingress-controller的部署，并通过金山云的负载均衡服务暴露到集群外。部署方式可参考Nginx-ingress使用。

## Ingress配置策略

Ingress controller支持通过配置ingress annotations实现不同场景下的灰度发布和测试。Nginx annotations支持以下四种灰度发布（Canary）规则：

- nginx.ingress.kubernetes.io/canary-by-header：基于 Request Header 的流量切分，适用于灰度发布以及 A/B 测试。当 Request Header 设置为 always时，请求将会被一直发送到 Canary 版本；当 Request Header 设置为 never时，请求不会被发送到 Canary 入口；对于任何其他 Header 值，将忽略 Header，并通过优先级将请求与其他 Canary 规则进行优先级的比较。
- nginx.ingress.kubernetes.io/canary-by-header-value：要匹配的 Request Header 的值，用于通知 Ingress 将请求路由到 Canary Ingress 中指定的服务。当 Request Header 设置为此值时，它将被路由到 Canary 入口。该规则允许用户自定义 Request Header 的值，必须与上一个 annotation（即：canary-by-header）一起使用。
- nginx.ingress.kubernetes.io/canary-weight：基于服务权重的流量切分，适用于蓝绿部署，权重范围 0 - 100 按百分比将请求路由到 Canary Ingress 中指定的服务。权重为 0 意味着该金丝雀规则不会向 Canary 入口的服务发送任何请求。权重为 100 意味着所有请求都将被发送到 Canary 入口。
- nginx.ingress.kubernetes.io/canary-by-cookie：基于 Cookie 的流量切分，适用于灰度发布与 A/B 测试。用于通知 Ingress 将请求路由到 Canary Ingress 中指定的服务的 cookie。当 cookie 值设置为 always时，它将被路由到 Canary 入口；当 cookie 值设置为 never时，请求不会被发送到 Canary 入口；对于任何其他值，将忽略 cookie 并将请求与其他金丝雀规则进行优先级的比较。

  注意：当同时使用多个Canary规则时，按如下优先顺序进行排序： canary-by-header -> canary-by-cookie -> canary-weight

## 部署示例

以下示例中将会部署helloworld服务的v1和v2版本，将v2版本作为canary版本，在ingress中设置canary规则，实现基于服务权重的流量切分。

### 创建测试应用

helloworld-v1.yaml如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-world
      version: v1
  template:
    metadata:
      labels:
        app: hello-world
        version: v1
    spec:
      containers:
      - name: hello-world
        image: hub.kce.ksyun.com/kingsoft/hello-world:v1
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello-world
  name: hello-world-svc
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

```
  selector:
    app: hello-world
    version: v1
  type: ClusterIP
```

helloworld-v2.yaml如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-world
      version: v2
  template:
    metadata:
      labels:
        app: hello-world
        version: v2
    spec:
      containers:
      - name: hello-world
        image: hub.kce.ksyun.com/kingsoft/hello-world:v2
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello-world
  name: hello-world-svc-v2
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: hello-world
    version: v2
  type: ClusterIP
```

创建并验证v1,v2版本服务的部署情况：

```
[root@vm10-0-11-201 ~]# kubectl apply -f helloworld-v2.yaml
deployment.apps/hello-world-v2 created
service/hello-world-svc-v2 created
[root@vm10-0-11-201 ~]# kubectl get deploy
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
hello-world      1/1     1            1           15s
hello-world-v2   1/1     1            1           9s
[root@vm10-0-11-201 ~]# kubectl get svc
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
hello-world-svc      ClusterIP   10.254.200.211   <none>        80/TCP    19s
hello-world-svc-v2   ClusterIP   10.254.14.130    <none>        80/TCP    13s
kubernetes           ClusterIP   10.254.0.1       <none>        443/TCP   7d
[root@vm10-0-11-201 ~]# curl 10.254.200.211
Hello World v1!
[root@vm10-0-11-201 ~]# curl 10.254.14.130
Hello World v2!
```

## 配置Ingress

### 基于服务权重进行流量切分

对v1版本服务进行Ingress配置，创建helloworld-ingress.yaml：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: hello.world.test
    http:
      paths:
      - backend:
          serviceName: hello-world-svc
```

```
          servicePort: 80
```

对v2版本的canary规则进行配置，创建weight-ingress.yaml：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-weight: "50"
  name: helloworld-weight
spec:
  rules:
  - host: hello.world.test
    http:
      paths:
      - backend:
          serviceName: hello-world-svc-v2
          servicePort: 80
```

创建Ingress规则：

```
[root@vm10-0-11-201 ~]# kubectl apply -f helloworld-ingress.yaml
ingress.extensions/hello-world created
[root@vm10-0-11-201 ~]# kubectl apply -f weight-ingress.yaml
ingress.extensions/helloworld-weight created
[root@vm10-0-11-201 ~]# kubectl get ingress
NAME                CLASS    HOSTS              ADDRESS   PORTS   AGE
hello-world         <none>   hello.world.test             80      41s
helloworld-weight   <none>   hello.world.test             80      27s
```

### 验证访问情况

通过以下命令获取EXTERNAL-IP及访问服务：

```
[root@vm10-0-11-201 ~]# kubectl get svc -n ingress-nginx
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                      AGE
nginx-ingress   LoadBalancer   10.254.28.54   120.92.xx.xx   80:31741/TCP,443:32754/TCP   3h19m
[root@vm10-0-11-201 ~]# for i in $(seq 1 10); do curl -H "Host: hello.world.test"  http://120.92.xx.xx; done;
Hello World v2!
Hello World v2!
Hello World v1!
Hello World v2!
Hello World v2!
Hello World v1!
Hello World v1!
Hello World v1!
Hello World v2!
Hello World v2!
```

多次访问能发现约50%的流量会被分发到v2版本服务中。

# 在KCE集群中对Pod进行带宽限速

金山云容器服务原生支持对Pod进行带宽限速。本文档讲介绍如何在KCE集群设置Pod带宽限速。

### 备注

- 对于创建时间在2021-3-16之后的集群，默认支持对Pod带宽进行限速；对于创建时间在2021-3-16之前的集群，如需要使用Pod带宽限速的能力，请联系您的商务申请
- 集群网络插件Flannel和Canal均支持Pod带宽限速的能力

### 使用方式

新建Pod，通过annotations的方式设置Pod出入带宽

- kubernetes.io/egress-bandwidth:定义Pod的出向带宽，如240M，这里单位是Mbit，
- kubernetes.io/ingress-bandwidth：定义Pod的入向带宽，如400M，这里单位是Mbit

Yaml示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
```

```
    kubernetes.io/egress-bandwidth: 240M
    kubernetes.io/ingress-bandwidth: 400M
  name: iperf3-4
  namespace: default
spec:
  containers:
  - args:
    - iperf3
    - -s
    image: networkstatic/iperf3
    imagePullPolicy: Always
    name: iperf3-4
    resources: {}
  dnsPolicy: ClusterFirst
```

### 验证

你可以通过以下两种方式验证：

- 登陆pod所在的节点，执行：

  ```
  tc qdisc show
  ```

  返回如下示例代表设置成功

  ```
  qdisc tbf 1: dev vethbab31be6 root refcnt 2 rate 400Mbit burst 256Mb lat 25.0ms
  qdisc ingress ffff: dev vethbab31be6 parent ffff:fff1 ----------------
  qdisc tbf 1: dev bwp91fff0f8f685 root refcnt 2 rate 240Mbit burst 256Mb lat 25.0ms
  ```

- 使用iperf3工具验证

  ```
  iperf3 -c <服务 IP> -i 1
  ```

  结果如下：

  ```
  Server listening on 5201
  -----------------------------------------------------------
  Accepted connection from 10.0.11.85, port 45690
  [  5] local 10.0.11.207 port 5201 connected to 10.0.11.85 port 45692
  [ ID] Interval           Transfer     Bandwidth
  [  5]   0.00-1.00   sec   243 MBytes  2.03 Gbits/sec
  [  5]   1.00-2.00   sec  49.2 MBytes   413 Mbits/sec
  [  5]   2.00-3.00   sec  27.3 MBytes   229 Mbits/sec
  [  5]   3.00-4.00   sec  27.3 MBytes   229 Mbits/sec
  [  5]   4.00-5.00   sec  27.4 MBytes   230 Mbits/sec
  [  5]   5.00-6.00   sec  27.3 MBytes   229 Mbits/sec
  [  5]   6.00-7.00   sec  27.3 MBytes   229 Mbits/sec
  [  5]   7.00-8.00   sec  27.3 MBytes   229 Mbits/sec
  [  5]   8.00-9.00   sec  27.3 MBytes   229 Mbits/sec
  [  5]   9.00-10.00  sec  27.3 MBytes   229 Mbits/sec
  [  5]  10.00-10.06  sec  1.49 MBytes   225 Mbits/sec
  ```

  ---

- [ ID] Interval Transfer Bandwidth [ 5] 0.00-10.06 sec 0.00 Bytes 0.00 bits/sec sender [ 5] 0.00-10.06 sec 512 MBytes 427 Mbits/sec receiver

# 在KCE集群中使用Networkpolicy（网络策略）

### 组件介绍

NetworkPolicy是Kubernetes提供的一种资源，用于定义基于Pod的网络隔离策略。Networkpolicy允许在IP地址或端口层面（第3层或第4层）对特定应用程序的网络流量进行控制。通过设置规则，可以限制Pod之间的通信，确保只有经过授权的Pod可以相互通信，从而增强网络安全性。 Networkpolicy提供3种可以组合使用的规则：

- podSelector：选择附加在Pod上的标签，对带有标签Pod的进出流量实现访问控制。
- namespaceSelector：选择附加在命名空间上的标签，对命名空间下的Pod进出流量实现访问控制。
- ipBlock：选择特定IP地址段，对归属在此IP地址段的Pod进出流量实现访问控制。

### 使用前提

- 集群类型需为KCE独立部署集群，KCE托管集群暂不支持使用Networkpolicy。

  在创建集群时，网络模型需要选择Calico

- Networkpolicy暂不支持通过界面配置，所有需要获取集群Kubeconfig并通过Kubectl链接集群，编写Yaml并下发至集群内

## 约束与限制

- 不支持对IPV6地址网络隔离。

## 在KCE集群中启用Networkpolicy组件

1. 进入目标集群，在组件管理列表种找到Networkpolicy组件。
2. 在组件实例页面点击安装，当组件状态变为运行中代表已经安装成功。
3. 通过Kubectl工具链接集群，下发Networkpolicy配置。

### Networkpolicy配置示例

#### 默认策略

默认情况下，如果命名空间中不存在任何策略，则所有进出该名字空间中 Pod 的流量都被允许。

#### 拒绝所有入站流量

在命名空间（default）中的所有Pod上启用默认的拒绝策略，阻止任何Ingress(入站)流量进入这些Pod。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: default-deny-ingress
namespace: default
spec:
podSelector: {}
policyTypes:
- Ingress
```

#### 允许所有入站流量

在命名空间（default）中的所有Pod允许所有的Ingress（入站）流量。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: allow-all-ingress
namespace: default
spec:
podSelector: {}
ingress:
- {}
policyTypes:
- Ingress
```

#### 拒绝所有出站流量

在命名空间（default）中的所有Pod上启用默认的拒绝策略，限制任何出站流量，阻止这些Pod发起对外部的网络访问。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: default-deny-egress
namespace: default
spec:
podSelector: {}
policyTypes:
- Egress
```

#### 允许所有出站流量

在命名空间（default）中的所有Pod上启用允许所有的Egress流量的策略，允许这些Pod发起对任何目的地的出站网络访问。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: allow-all-egress
```

```
namespace: default
spec:
podSelector: {}
egress:
- {}
policyTypes:
- Egress
```

### 使用podSelector设置访问控制

在命名空间（default）中的具有标签role: db的Pod上启用策略，允许来自具有标签role: frontend的Pod的TCP流量进入这些Pod的6379端口。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: test-network-policy
namespace: default
spec:
podSelector:
matchLabels:
role: db
policyTypes:
- Ingress
ingress:
- from:
- podSelector:
matchLabels:
  role: frontend
ports:
- protocol: TCP
port: 6379
```

### 使用namespaceSelector设置访问控制

在命名空间（default）中具有标签role: db的Pod上启用策略，允许来自具有标签project: myproject的命名空间的TCP流量进入这些Pod的6379端口。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: test-network-policy
namespace: default
spec:
podSelector:
matchLabels:
role: db
policyTypes:
- Ingress
ingress:
- from:
- namespaceSelector:
matchLabels:
  project: myproject
ports:
- protocol: TCP
port: 6379
```

### 使用ipBlock设置访问控制

在命名空间（default）中具有标签role: db的Pod上启用策略，允许来自CIDR范围为172.17.0.0/16的IP地址的TCP流量进入这些Pod的6379端口，但排除了CIDR范围为172.17.1.0/24的IP地址。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: test-network-policy
namespace: default
spec:
podSelector:
matchLabels:
role: db
policyTypes:
- Ingress
ingress:
- from:
- ipBlock:
cidr: 172.17.0.0/16
except:
```

```
    - 172.17.1.0/24
  ports:
  - protocol: TCP
  port: 6379
```

# 容器集群中使用Volcano调度器

## 组件介绍

Volcano是一个基于Kubernetes的容器批处理平台，旨在为机器学习、深度学习、生物信息学、基因组学等大数据应用提供高性能计算能力。相比于Kubernetes本身的调度和管理能力，Volcano提供了更加专业和高效的任务调度引擎、异构芯片管理和任务运行管理等功能，同时支持CPU、GPU等异构设备的混合调度。Volcano还能够与几乎所有主流计算框架进行无缝对接，如Spark、TensorFlow、PyTorch、Flink、Argo、MindSpore、PaddlePaddle等，为科学计算和大数据分析提供高效、便捷的解决方案。目前金山云容器集群已将Volcano添加至组件管理，欢迎使用。

## 使用流程

1. 登录金山云控制台，进入容器服务
2. 进入集群内，创建名为volcano-system的命名空间（推荐将volcano资源部署在特定的命名空间，不强制）
3. 进入容器服务-Helm应用-Helm3内，安装Volcano
4. 填写部署Volcano Helm chart所需参数，关键参数如下：
   - 所在命名空间：推荐选择volcano-system
   - Chart来源：选择Ksyun Helm Chart
   - Chart名称：选择volcano
   - Chart版本：目前提供Volcano 1.7.0版本
5. 验证组件部署状态，查看volcano-system命名空间下资源状态（也可进入控制台内查看）

### Volcano scheduler配置

点击查看Volcano scheduler配置

### 调度器设置

当集群内安装完Volcano后，如果不指定调度器，集群仍然会使用default scheduler调度Pod。以下提供使用volcano scheduler的方式： 为Pod指定调度器：在pod的yaml中指定spec.scheduerName为volcano

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx2222
labels:
app: nginx
spec:
replicas: 4
selector:
matchLabels:
  app: nginx
template:
metadata:
  labels:
    app: nginx
spec:
  # 指定调度器为volcano
  schedulerName: volcano
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        cpu: 1
        memory: 100Mi
      requests:
        cpu: 1
        memory: 100Mi
    ports:
    - containerPort: 80
```

| volcano支持的Pod注解volcano支持的Pod注解： | Pod注解 | 说明 |
| --- | --- | --- |
| scheduling.volcano.sh/queue-name: "" | 指定负载所在队列，其中为队列名称 | |

| | |
|---|---|
| volcano.sh/preemptable: "true" | 表示作业是否可抢占。开启后，认为该作业可以被抢占。取值范围：true：开启抢占。（默认为开启状态）false：关闭抢占 |

验证效果：

```
kubectl describe po nginx2222-6856d58cd-r2274
```

# 自定义kubelet存储目录的CSI部署

CSI无法自动感知kubelet存储目录的变化，如您变更了kubelet的存储目录，为保证CSI的正常使用，您需要卸载后重装。本文主要介绍自定义kubelet存储目录的CSI部署方法。

## 部署流程

1. 登录金山云控制台，进入容器服务。
2. 在左侧导航栏中点击**集群**，进入集群列表页。
3. 选择目标集群进入集群详情页，在左侧导航栏点击**组件管理**，找到csi-driver组件。
4. 点击csi-driver，进入组件实例页面。
5. 点击右侧的删除，将已安装的csi-driver组件实例卸载。
6. 返回集群列表页，在左侧导航栏中点击**Helm应用->Helm 3**。
7. 在上方选择需要操作的集群，点击**新建**。
8. 根据以下提示填写应用配置，配置完成后点击**部署**。
   - 应用名称：填写Helm应用名称，此字段全集群唯一。
   - 所在命名空间：根据实际需求选择。
   - Chart来源：选择Kysun Helm Chart。
   - Chart名称选择csi-driver。
   - Chart版本：选择最新版本。
9. 在自定义变量中点击values.yaml，修改volumes:register与volumes:mount的值，与自定义kubelet中的存储目录保持一致。

说明：

- 不支持一个集群中多个机器的 kubelet参数不同。
- 在部署完成后再次修改kubelet启动参数可能会导致插件失效，此时需要对 PV以及 PV 内的数据做好备份，之后再重新部署csi插件。

# 自建NFS迁移至KFS

KCE支持用户使用自建的NFS，本文将介绍如何将自建的NFS迁移至金山云文件系统（KFS）。

## 步骤一：新建KFS实例

1. 进入文件存储KFS控制台。
2. 单击新建文件系统，进行文件系统的创建，详细操作参考创建文件系统及挂载点。
3. 点击进入已创建的文件系统查看详情。 如上图所示： server：10.x.x.xx share：/cfs-xxxxxxxxxx

## 步骤二：新建StorageClass

创建 StorageClass 的 YAML 文件如下：

```
kind: StorageClass
metadata:
  name: kfsplugin
provisioner: com.ksc.csi.nfsplugin
allowVolumeExpansion: false
parameters:
  server: 10.x.x.xx                ## server地址从步骤1中获取 固定值
  share: /cfs-xxxxxxxxxx           ## 从步骤1中获取 固定值
reclaimPolicy: Retain
volumeBindingMode: Immediate
mountOptions:
  - vers=3
  - nolock
  - proto=tcp
  - noresvport
```

## 步骤三：复制数据

### 1. 缩减应用的副本数量缩减为0

```
kubectl scale deployment <deployment-name> --replicas=0
```

### 2. 按照旧 PVC 的配置创建 PVC，指定新的StorageClass

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "data-new"
  namespace: "default"
spec:
  accessModes:
  - "ReadWriteOnce"
  resources:
    requests:
      storage: "100Mi"
  storageClassName: "kfsplugin"
```

### 3. 拷贝存储数据

作为替换的新的 PV 已创建，创建一个 Deployment 挂载新旧两个 PV，将旧 PV 的数据拷贝到新 PV：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: datacopy
  name: datacopy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: datacopy
  template:
    metadata:
      labels:
        app: datacopy
    spec:
      containers:
        - name: datacopy
          image: busybox
          command:
            - 'sleep'
          args:
            - infinity
          volumeMounts:
            - name: old-pvc
              readOnly: false
              mountPath: /mnt/old
            - name: new-pvc
              readOnly: false
              mountPath: /mnt/new
      volumes:
        - name: old-pvc
          persistentVolumeClaim:
            claimName: xxx-xxx  #需要替换成旧的pvc名称
        - name: new-pvc
          persistentVolumeClaim:
            claimName: data-new
```

① 使用以下命令进入容器后，分别检查旧PV挂载点是否包含应用数据，新PV挂载点是否为空：

```
kubectl exec -it <pod-name> -- sh
```

② 执行以下命令进行数据的复制和权限继承：

```
(cd /mnt/old; tar -cf - .) | (cd /mnt/new; tar -xpf -)
```

说明：此方法在遇到较大数据时较慢。

**验证数据是否复制完成：** ① 检查新的 PV 的挂载点是否包含旧的PV 的数据，在新旧pv的两个挂载路径下通过命令ls -l验证数据的所有权限是否被正确继承。 ② 确认数据复制完成后，需要将 datacopy deployment 的副本缩减为0（若后续不使用可直接删除），这样两个pvc 就和它失去关联，可以执行后续操作：

```
kubectl scale deployment datacopy --replicas=0
```

#### 4. 迁移PVC

迁移存储的理想状态是使用旧的 PVC，并将其指向新的 PV，这样工作负载的 YAML 配置清单就不需要做任何改变。但 PVC 和 PV 之间的绑定关系是不可更改的，要想让它们解绑，必须先删除旧的 PVC，再创建同名的 PVC，并将旧的 PV 与它绑定。 默认情况下 PV 的回收策略是 Delete，一旦删除 PVC，与之绑定的 PV 和 PV 里的数据都会被删除，因此需要修改回收策略，以便删除 PVC 时 PV 能够保留下来。

##### (1) 修改回收策略

执行命令 kubectl describe pv <pv-name>，分别查看新旧 PV 的回收策略： 修改新旧PV的回收策略为Retain，这样可以确保将新旧PVC删除时，PV不会受到影响。

```
kubectl patch pv <pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

##### (2) 删除新旧PVC

执行以下命令行，删除新旧PVC：

```
kubectl delete pvc <pvc-name>
```

##### (3)绑定pv

在创建最终的 PVC 之前，我们必须要确保新创建的 PVC 能够被绑定到新的 PV 上。通过命令kubectl describe pv <pv-name>可以看到新 PV 目前处于释放状态，不能被最终的 PVC 绑定： 这是因为 PV 在 spec.claimRef中仍然引用了已经被删除的 PVC，通过命令编辑PV，将spec.claimRef中的内容删除：

```
kubectl patch pv <pv-name> -p '{"spec":{"claimRef":null}}'
```

再次查看PV已处于可用状态： 创建与旧 PVC 同名的新PVC，并保证与旧 PVC 的参数相同：

- 新 PVC 的名字和旧 PVC 的名字相同；
- spec.volumeName指向新 PV；
- 新 PVC 的 metadata.annotations 和 metadata.labels和旧 PVC 保存相同，因为这些值可能会影响到应用部署（比如 Helm chart 等）。

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
name:
namespace:
spec:
accessModes:
- "ReadWriteOnce"
resources:
requests:
  storage:
storageClassName: "kfsplugin"
volumeMode: Filesystem
volumeName: "新PV的名字"
```

通过kubectl get pv与kubectl get pvc命令查看pv与pvC的信息，确保新的PVC与PV都为Bound状态。

### 步骤四：恢复工作负载

将工作负载副本恢复到期望数量：

```
kubectl scale deployment <deployment-name> --replicas=<期望数量>
```

在运行正常后可将旧PV删除。

# 单sc支持创建多可用区pv

CSI支持单sc多可用区pv。在有跨可用区节点的集群中，避免了由于pv所在的可用区与被调度的node的可用区不一致导致的pod pending的问题。

## 前提条件

组件csi-driver为2.0.4及以上版本。如需更新，请参考更新 Helm应用。

## 创建SC

可根据具体的需求选择不同的模式绑定pv。

**延迟创建绑定**

在sc的yaml中将volumeBindingMode设置为WaitForFirstConsumer，当集群的node跨zone，CSI driver能感知到pod所在的zone，会优先在对应的zone创建pv。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ssd
parameters:
  chargetype: HourlyInstantSettlement
  type: SSD3.0
provisioner: com.ksc.csi.diskplugin
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

**立刻创建绑定**

在sc的yaml中将volumeBindingMode设置为Immediate，会随机选择最合适的zone的创建pv。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ssd
parameters:
  chargetype: Daily
  type: SSD3.0
provisioner: com.ksc.csi.diskplugin
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

**指定创建pv的zone**

在sc的yaml中可通过allowedTopologies字段指定创建pv的可用区。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: wffc-allow
parameters:
  chargetype: Daily
  type: SSD3.0
#  zone: cn-beijing-6a
provisioner: com.ksc.csi.diskplugin
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
allowedTopologies:
    - key: failure-domain.beta.kubernetes.io/zone
      values:
        - cn-beijing-6a
        - cn-beijing-6b
```

## 创建pvc以及deployment

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ssd
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ssd
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ssd-deployment
  labels:
    app: ssd
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: ssd
  template:
    metadata:
      labels:
        app: ssd
    spec:
      containers:
        - name: nginx
          image: nginx
          imagePullPolicy: Always
          volumeMounts:
            - name: ssd
              mountPath: /usr/share/nginx/html
      volumes:
        - name: ssd
          persistentVolumeClaim:
            claimName: ssd
```