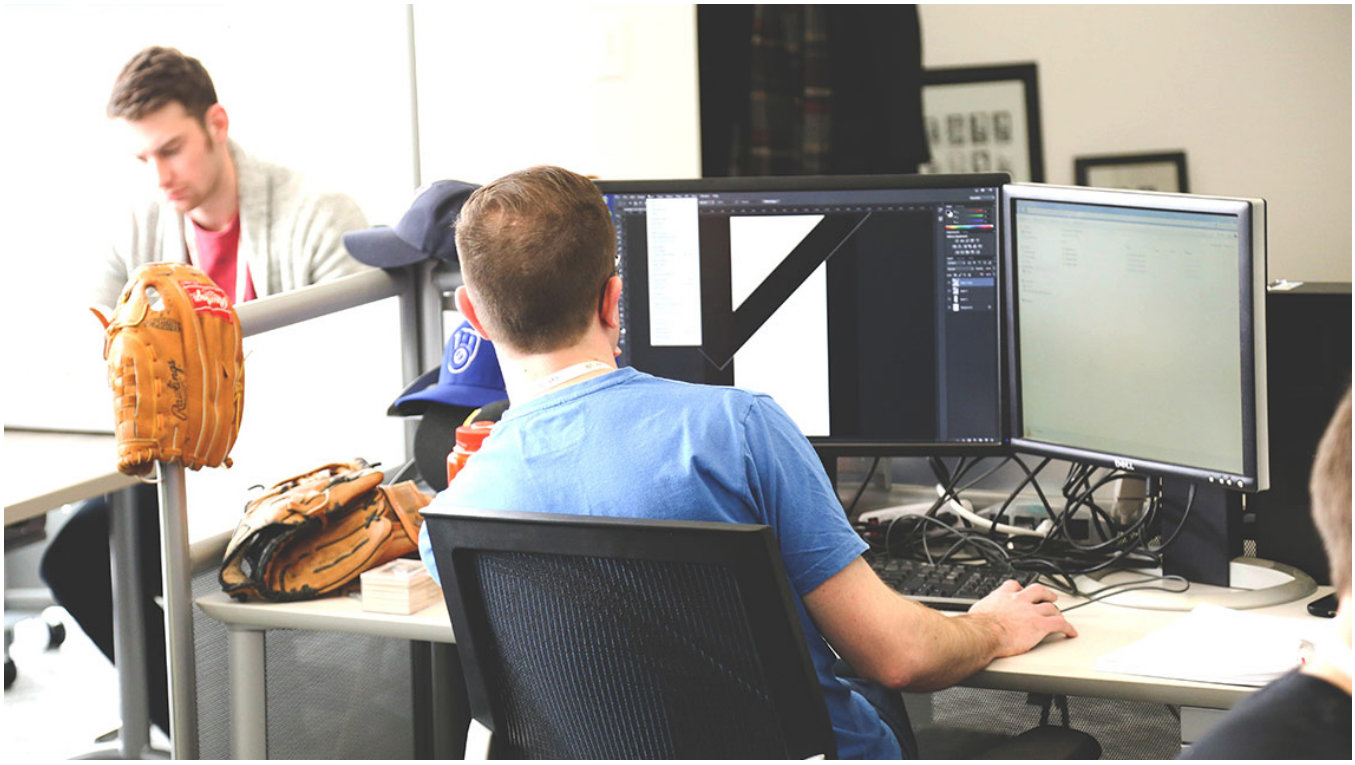
31 | 容器存储实践：CSI插件编写指南

2018-11-02 张磊



31 | 容器存储实践：CSI插件编写指南
朗读人：张磊 14'15'' | 6.53M

你好，我是张磊。今天我和你分享的主题是：容器存储实践之 CSI 插件编写指南。

在上一篇文章中，我已经为你详细讲解了 CSI 插件机制的设计原理。今天我将继续和你一起实践一个 CSI 插件的编写过程。

为了能够覆盖到 CSI 插件的所有功能，我这一次选择了 DigitalOcean 的块存储（Block Storage）服务，来作为实践对象。

DigitalOcean 是业界知名的“最简”公有云服务，即：它只提供虚拟机、存储、网络等为数不多的几个基础功能，其他功能一概不管。而这，恰恰就使得 DigitalOcean 成了我们在公有云上实践 Kubernetes 的最佳选择。

我们这次编写的 CSI 插件的功能，就是：让我们运行在 DigitalOcean 上的 Kubernetes 集群能够使用它的块存储服务，作为容器的持久化存储。

> 备注：在 DigitalOcean 上部署一个 Kubernetes 集群的过程，也很简单。你只需要先在 DigitalOcean 上创建几个虚拟机，然后按照我们在第 11 篇文章《从 0 到 1：搭建一个完整的 Kubernetes 集群》中从 0 到 1 的步骤直接部署即可。

而有了 CSI 插件之后，持久化存储的用法就非常简单了，你只需要创建一个如下所示的 StorageClass 对象即可：

```
复制代码
1 kind: StorageClass
2 apiVersion: storage.k8s.io/v1
3 metadata:
4   name: do-block-storage
5   namespace: kube-system
6   annotations:
7     storageclass.kubernetes.io/is-default-class: "true"
8 provisioner: com.digitalocean.csi.dobs
```

有了这个 StorageClass，External Provisoner 就会为集群中新出现的 PVC 自动创建出 PV，然后调用 CSI 插件创建出这个 PV 对应的 Volume，这正是 CSI 体系中 Dynamic Provisioning 的实现方式。

> 备注：`storageclass.kubernetes.io/is-default-class: "true"`的意思，是使用这个 StorageClass 作为默认的持久化存储提供者。

不难看到，这个 StorageClass 里唯一引人注意的，是 provisioner=com.digitalocean.csi.dobs 这个字段。显然，这个字段告诉了 Kubernetes，请使用名叫 com.digitalocean.csi.dobs 的 CSI 插件来为我处理这个 StorageClass 相关的所有操作。

那么，Kubernetes 又是如何知道一个 CSI 插件的名字的呢？

这就需要从 CSI 插件的第一个服务 CSI Identity 说起了。

其实，一个 CSI 插件的代码结构非常简单，如下所示：

```
复制代码
1 tree $GOPATH/src/github.com/digitalocean/csi-digitalocean/driver
2 $GOPATH/src/github.com/digitalocean/csi-digitalocean/driver
3 ├── controller.go
4 ├── driver.go
5 ├── identity.go
6 ├── mounter.go
7 └── node.go
```

其中，CSI Identity 服务的实现，就定义在了 driver 目录下的 identity.go 文件里。

当然，为了能够让 Kubernetes 访问到 CSI Identity 服务，我们需要先在 driver.go 文件里，定义一个标准的 gRPC Server，如下所示：

```
// Run starts the CSI plugin by communication over the given endpoint
func (d *Driver) Run() error {
 ...

 listener, err := net.Listen(u.Scheme, addr)
 ...

 d.srv = grpc.NewServer(grpc.UnaryInterceptor(errHandler))
 csi.RegisterIdentityServer(d.srv, d)
 csi.RegisterControllerServer(d.srv, d)
 csi.RegisterNodeServer(d.srv, d)

 d.ready = true // we're now ready to go!
 ...
 return d.srv.Serve(listener)
}
```

可以看到，只要把编写好的 gRPC Server 注册给 CSI，它就可以响应来自 External Components 的 CSI 请求了。

CSI Identity 服务中，最重要的接口是 GetPluginInfo，它返回的就是这个插件的名字和版本号，如下所示：

> 备注：CSI 各个服务的接口我在上一篇文章中已经介绍过，你也可以在这里找到它的 protoc 文件。

```
func (d *Driver) GetPluginInfo(ctx context.Context, req *csi.GetPluginInfoRequest) (*csi.GetPlug
 resp := &csi.GetPluginInfoResponse{
  Name:          driverName,
  VendorVersion: version,
 }
 ...
}
```

其中，driverName 的值，正是"com.digitalocean.csi.dobs"。所以说，Kubernetes 正是通过 GetPluginInfo 的返回值，来找到你在 StorageClass 里声明要使用的 CSI 插件的。

> 备注：CSI 要求插件的名字遵守“反向 DNS”格式。

另外一个GetPluginCapabilities 接口也很重要。这个接口返回的是这个 CSI 插件的“能力”。

比如，当你编写的 CSI 插件不准备实现“Provision 阶段”和“Attach 阶段”（比如，一个最简单的 NFS 存储插件就不需要这两个阶段）时，你就可以通过这个接口返回：本插件不提供 CSI Controller 服务，即：没有 csi.PluginCapability_Service_CONTROLLER_SERVICE 这个“能力”。这样，Kubernetes 就知道这个信息了。

最后，CSI Identity 服务还提供了一个 Probe 接口。Kubernetes 会调用它来检查这个 CSI 插件是否正常工作。

一般情况下，我建议你在编写插件时给它设置一个 Ready 标志，当插件的 gRPC Server 停止的时候，把这个 Ready 标志设置为 false。或者，你可以在这里访问一下插件的端口，类似于健康检查的做法。

> 备注：关于健康检查的问题，你可以再回顾一下第 15 篇文章《深入解析 Pod 对象（二）：使用进阶》中的相关内容。

然后，我们要开始编写 CSI 插件的第二个服务，即 CSI Controller 服务了。它的代码实现，在 controller.go 文件里。

在上一篇文章中我已经为你讲解过，这个服务主要实现的就是 Volume 管理流程中的“Provision 阶段”和“Attach 阶段”。

“Provision 阶段”对应的接口，是 CreateVolume 和 DeleteVolume，它们的调用者是 External Provisoner。以 CreateVolume 为例，它的主要逻辑如下所示：

```
func (d *Driver) CreateVolume(ctx context.Context, req *csi.CreateVolumeRequest) (*csi.CreateVol
 ...

 volumeReq := &godo.VolumeCreateRequest{
  Region:        d.region,
  Name:          volumeName,
  Description:   createdByDO,
  SizeGigaBytes: size / GB,
 }

 ...

 vol, _, err := d.doClient.Storage.CreateVolume(ctx, volumeReq)

 ...

 resp := &csi.CreateVolumeResponse{
  Volume: &csi.Volume{
   Id:            vol.ID,
   CapacityBytes: size,
   AccessibleTopology: []*csi.Topology{
    {
```

```
      Segments: map[string]string{
       "region": d.region,
      },
     },
    },
   },
  }

  return resp, nil
 }
```

可以看到，对于 DigitalOcean 这样的公有云来说，CreateVolume 需要做的操作，就是调用 DigitalOcean 块存储服务的 API，创建出一个存储卷（d.doClient.Storage.CreateVolume）。如果你使用的是其他类型的块存储（比如 Cinder、Ceph RBD 等），对应的操作也是类似地调用创建存储卷的 API。

而“Attach 阶段”对应的接口是 ControllerPublishVolume 和 ControllerUnpublishVolume，它们的调用者是 External Attacher。以 ControllerPublishVolume 为例，它的逻辑如下所示：

复制代码

```
func (d *Driver) ControllerPublishVolume(ctx context.Context, req *csi.ControllerPublishVolumeRe
 ...

  dropletID, err := strconv.Atoi(req.NodeId)

  // check if volume exist before trying to attach it
  _, resp, err := d.doClient.Storage.GetVolume(ctx, req.VolumeId)

 ...

  // check if droplet exist before trying to attach the volume to the droplet
  _, resp, err = d.doClient.Droplets.Get(ctx, dropletID)

 ...

  action, resp, err := d.doClient.StorageActions.Attach(ctx, req.VolumeId, dropletID)

 ...

 if action != nil {
  ll.Info("waiting until volume is attached")
 if err := d.waitAction(ctx, req.VolumeId, action.ID); err != nil {
  return nil, err
  }
  }

  ll.Info("volume is attached")
```

```
  return &csi.ControllerPublishVolumeResponse{}, nil
}
```

可以看到，对于 DigitalOcean 来说，ControllerPublishVolume 在“Attach 阶段”需要做的工作，是调用 DigitalOcean 的 API，将我们前面创建的存储卷，挂载到指定的虚拟机上（d.doClient.StorageActions.Attach）。

其中，存储卷由请求中的 VolumeId 来指定。而虚拟机，也就是将要运行 Pod 的宿主机，则由请求中的 NodeId 来指定。这些参数，都是 External Attacher 在发起请求时需要设置的。

我在上一篇文章中已经为你介绍过，External Attacher 的工作原理，是监听（Watch）了一种名叫 VolumeAttachment 的 API 对象。这种 API 对象的主要字段如下所示：

复制代码

```
// VolumeAttachmentSpec is the specification of a VolumeAttachment request.
type VolumeAttachmentSpec struct {
 // Attacher indicates the name of the volume driver that MUST handle this
 // request. This is the name returned by GetPluginName().
 Attacher string

 // Source represents the volume that should be attached.
 Source VolumeAttachmentSource

 // The node that the volume should be attached to.
 NodeName string
}
```

而这个对象的生命周期，正是由 AttachDetachController 负责管理的（这里，你可以再回顾一下第 28 篇文章《PV、PVC、StorageClass，这些到底在说啥？》中的相关内容）。

这个控制循环的职责，是不断检查 Pod 所对应的 PV，在它所绑定的宿主机上的挂载情况，从而决定是否需要对这个 PV 进行 Attach（或者 Dettach）操作。

而这个 Attach 操作，在 CSI 体系里，就是创建出上面这样一个 VolumeAttachment 对象。可以看到，Attach 操作所需的 PV 的名字（Source）、宿主机的名字（NodeName）、存储插件的名字（Attacher），都是这个 VolumeAttachment 对象的一部分。

而当 External Attacher 监听到这样的一个对象出现之后，就可以立即使用 VolumeAttachment 里的这些字段，封装成一个 gRPC 请求调用 CSI Controller 的 ControllerPublishVolume 方法。

最后，我们就可以编写 CSI Node 服务了。

CSI Node 服务对应的，是 Volume 管理流程里的“Mount 阶段”。它的代码实现，在 node.go 文件里。

我在上一篇文章里曾经提到过，kubelet 的 VolumeManagerReconciler 控制循环会直接调用 CSI Node 服务来完成 Volume 的“Mount 阶段”。

不过，在具体的实现中，这个“Mount 阶段”的处理其实被细分成了 NodeStageVolume 和 NodePublishVolume 这两个接口。

这里的原因其实也很容易理解：我在前面第 28 篇文章《PV、PVC、StorageClass，这些到底在说啥？》中曾经介绍过，对于磁盘以及块设备来说，它们被 Attach 到宿主机上之后，就成为了宿主机上的一个待用存储设备。而到了“Mount 阶段”，我们首先需要格式化这个设备，然后才能把它挂载到 Volume 对应的宿主机目录上。

在 kubelet 的 VolumeManagerReconciler 控制循环中，这两步操作分别叫作MountDevice 和 SetUp。

其中，MountDevice 操作，就是直接调用了 CSI Node 服务里的 NodeStageVolume 接口。顾名思义，这个接口的作用，就是格式化 Volume 在宿主机上对应的存储设备，然后挂载到一个临时目录（Staging 目录）上。

对于 DigitalOcean 来说，它对 NodeStageVolume 接口的实现如下所示：

```
复制代码
1  func (d *Driver) NodeStageVolume(ctx context.Context, req *csi.NodeStageVolumeRequest) (*csi.Noc
2   ...
3
4   vol, resp, err := d.doClient.Storage.GetVolume(ctx, req.VolumeId)
5
6   ...
7
8   source := getDiskSource(vol.Name)
9   target := req.StagingTargetPath
10
11  ...
12
13  if !formatted {
14   ll.Info("formatting the volume for staging")
15   if err := d.mounter.Format(source, fsType); err != nil {
16    return nil, status.Error(codes.Internal, err.Error())
17   }
18  } else {
19   ll.Info("source device is already formatted")
20  }
21
22 ...
23
```

```
 if !mounted {
  if err := d.mounter.Mount(source, target, fsType, options...); err != nil {
   return nil, status.Error(codes.Internal, err.Error())
  }
 } else {
  ll.Info("source device is already mounted to the target path")
 }

 ...
 return &csi.NodeStageVolumeResponse{}, nil
}
```

可以看到，在 NodeStageVolume 的实现里，我们首先通过 DigitalOcean 的 API 获取到了这个 Volume 对应的设备路径（getDiskSource）；然后，我们把这个设备格式化成指定的格式（d.mounter.Format）；最后，我们把格式化后的设备挂载到了一个临时的 Staging 目录（StagingTargetPath）下。

而 SetUp 操作则会调用 CSI Node 服务的 NodePublishVolume 接口。有了上述对设备的预处理工作后，它的实现就非常简单了，如下所示：

复制代码

```
func (d *Driver) NodePublishVolume(ctx context.Context, req *csi.NodePublishVolumeRequest) (*csi
 ...
 source := req.StagingTargetPath
 target := req.TargetPath

 mnt := req.VolumeCapability.GetMount()
 options := mnt.MountFlag
    ...

 if !mounted {
  ll.Info("mounting the volume")
  if err := d.mounter.Mount(source, target, fsType, options...); err != nil {
   return nil, status.Error(codes.Internal, err.Error())
  }
 } else {
  ll.Info("volume is already mounted")
 }

 return &csi.NodePublishVolumeResponse{}, nil
}
```

可以看到，在这一步实现中，我们只需要做一步操作，即：将 Staging 目录，绑定挂载到 Volume 对应的宿主机目录上。

由于 Staging 目录，正是 Volume 对应的设备被格式化后挂载在宿主机上的位置，所以当它和 Volume 的宿主机目录绑定挂载之后，这个 Volume 宿主机目录的“持久化”处理也就完成了。

当然，我在前面也曾经提到过，对于文件系统类型的存储服务来说，比如 NFS 和 GlusterFS 等，它们并没有一个对应的磁盘“设备”存在于宿主机上，所以 kubelet 在 VolumeManagerReconciler 控制循环中，会跳过 MountDevice 操作而直接执行 SetUp 操作。所以对于它们来说，也就不需要实现 NodeStageVolume 接口了。

在编写完了 CSI 插件之后，我们就可以把这个插件和 External Components 一起部署起来。

首先，我们需要创建一个 DigitalOcean client 授权需要使用的 Secret 对象，如下所示：

```
apiVersion: v1
kind: Secret
metadata:
  name: digitalocean
  namespace: kube-system
stringData:
  access-token: "a05dd2f26b9b9ac2asdas__REPLACE_ME____123cb5d1ec17513e06da"
```

接下来，我们通过一句指令就可以将 CSI 插件部署起来：

```
$ kubectl apply -f https://raw.githubusercontent.com/digitalocean/csi-digitalocean/master/deploy
```

这个 CSI 插件的 YAML 文件的主要内容如下所示（其中，非重要的内容已经被略去）：

```
kind: DaemonSet
apiVersion: apps/v1beta2
metadata:
  name: csi-do-node
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: csi-do-node
  template:
    metadata:
      labels:
        app: csi-do-node
        role: csi-do
    spec:
      serviceAccount: csi-do-node-sa

      hostNetwork: true
```

```
      containers:
        - name: driver-registrar
          image: quay.io/k8scsi/driver-registrar:v0.3.0
          ...
        - name: csi-do-plugin
          image: digitalocean/do-csi-plugin:v0.2.0
          args :
            - "--endpoint=$(CSI_ENDPOINT)"
            - "--token=$(DIGITALOCEAN_ACCESS_TOKEN)"
            - "--url=$(DIGITALOCEAN_API_URL)"
          env:
            - name: CSI_ENDPOINT
              value: unix:///csi/csi.sock
            - name: DIGITALOCEAN_API_URL
              value: https://api.digitalocean.com/
            - name: DIGITALOCEAN_ACCESS_TOKEN
              valueFrom:
                secretKeyRef:
                  name: digitalocean
                  key: access-token
          imagePullPolicy: "Always"
          securityContext:
            privileged: true
            capabilities:
              add: ["SYS_ADMIN"]
            allowPrivilegeEscalation: true
          volumeMounts:
            - name: plugin-dir
              mountPath: /csi
            - name: pods-mount-dir
              mountPath: /var/lib/kubelet
              mountPropagation: "Bidirectional"
            - name: device-dir
              mountPath: /dev
      volumes:
        - name: plugin-dir
          hostPath:
            path: /var/lib/kubelet/plugins/com.digitalocean.csi.dobs
            type: DirectoryOrCreate
        - name: pods-mount-dir
          hostPath:
            path: /var/lib/kubelet
            type: Directory
        - name: device-dir
          hostPath:
            path: /dev
---
kind: StatefulSet
apiVersion: apps/v1beta1
metadata:
  name: csi-do-controller
  namespace: kube-system
```

```
spec:
  serviceName: "csi-do"
  replicas: 1
  template:
    metadata:
      labels:
        app: csi-do-controller
        role: csi-do
    spec:
      serviceAccount: csi-do-controller-sa
      containers:
        - name: csi-provisioner
          image: quay.io/k8scsi/csi-provisioner:v0.3.0
          ...
        - name: csi-attacher
          image: quay.io/k8scsi/csi-attacher:v0.3.0
          ...
        - name: csi-do-plugin
          image: digitalocean/do-csi-plugin:v0.2.0
          args :
            - "--endpoint=$(CSI_ENDPOINT)"
            - "--token=$(DIGITALOCEAN_ACCESS_TOKEN)"
            - "--url=$(DIGITALOCEAN_API_URL)"
          env:
            - name: CSI_ENDPOINT
              value: unix:///var/lib/csi/sockets/pluginproxy/csi.sock
            - name: DIGITALOCEAN_API_URL
              value: https://api.digitalocean.com/
            - name: DIGITALOCEAN_ACCESS_TOKEN
              valueFrom:
                secretKeyRef:
                  name: digitalocean
                  key: access-token
          imagePullPolicy: "Always"
          volumeMounts:
            - name: socket-dir
              mountPath: /var/lib/csi/sockets/pluginproxy/
      volumes:
        - name: socket-dir
          emptyDir: {}
```

可以看到，我们编写的 CSI 插件只有一个二进制文件，它的镜像是 digitalocean/do-csi-plugin:v0.2.0。

而我们部署 CSI 插件的常用原则是：

第一，通过 DaemonSet 在每个节点上都启动一个 CSI 插件，来为 kubelet 提供 CSI Node 服务。这是因为，CSI Node 服务需要被 kubelet 直接调用，所以它要和 kubelet “一对一”地部署起来。

此外，在上述 DaemonSet 的定义里面，除了 CSI 插件，我们还以 sidecar 的方式运行着 driver-registrar 这个外部组件。它的作用，是向 kubelet 注册这个 CSI 插件。这个注册过程使用的插件信息，则通过访问同一个 Pod 里的 CSI 插件容器的 Identity 服务获取到。

需要注意的是，由于 CSI 插件运行在一个容器里，那么 CSI Node 服务在“Mount 阶段”执行的挂载操作，实际上是发生在这个容器的 Mount Namespace 里的。可是，我们真正希望执行挂载操作的对象，都是宿主机 /var/lib/kubelet 目录下的文件和目录。

所以，在定义 DaemonSet Pod 的时候，我们需要把宿主机的 /var/lib/kubelet 以 Volume 的方式挂载进 CSI 插件容器的同名目录下，然后设置这个 Volume 的 mountPropagation=Bidirectional，即开启双向挂载传播，从而将容器在这个目录下进行的挂载操作“传播”给宿主机，反之亦然。

第二，通过 StatefulSet 在任意一个节点上再启动一个 CSI 插件，为 External Components 提供 CSI Controller 服务。所以，作为 CSI Controller 服务的调用者，External Provisioner 和 External Attacher 这两个外部组件，就需要以 sidecar 的方式和这次部署的 CSI 插件定义在同一个 Pod 里。

你可能会好奇，为什么我们会用 StatefulSet 而不是 Deployment 来运行这个 CSI 插件呢。

这是因为，由于 StatefulSet 需要确保应用拓扑状态的稳定性，所以它对 Pod 的更新，是严格保证顺序的，即：只有在前一个 Pod 停止并删除之后，它才会创建并启动下一个 Pod。

而像我们上面这样将 StatefulSet 的 replicas 设置为 1 的话，StatefulSet 就会确保 Pod 被删除重建的时候，永远有且只有一个 CSI 插件的 Pod 运行在集群中。这对 CSI 插件的正确性来说，至关重要。

而在今天这篇文章一开始，我们就已经定义了这个 CSI 插件对应的 StorageClass（即：do-block-storage），所以你接下来只需要定义一个声明使用这个 StorageClass 的 PVC 即可，如下所示：

复制代码

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: do-block-storage
```

当你把上述 PVC 提交给 Kubernetes 之后，你就可以在 Pod 里声明使用这个 csi-pvc 来作为持久化存储了。这一部分使用 PV 和 PVC 的内容，我就不再赘述了。

总结

在今天这篇文章中，我以一个 DigitalOcean 的 CSI 插件为例，和你分享了编写 CSI 插件的具体流程。

基于这些讲述，你现在应该已经对 Kubernetes 持久化存储体系有了一个更加全面和深入的认识。

举个例子，对于一个部署了 CSI 存储插件的 Kubernetes 集群来说：

当用户创建了一个 PVC 之后，你前面部署的 StatefulSet 里的 External Provisioner 容器，就会监听到这个 PVC 的诞生，然后调用同一个 Pod 里的 CSI 插件的 CSI Controller 服务的 CreateVolume 方法，为你创建出对应的 PV。

这时候，运行在 Kubernetes Master 节点上的 Volume Controller，就会通过 PersistentVolumeController 控制循环，发现这对新创建出来的 PV 和 PVC，并且看到它们声明的是同一个 StorageClass。所以，它会把这一对 PV 和 PVC 绑定起来，使 PVC 进入 Bound 状态。

然后，用户创建了一个声明使用上述 PVC 的 Pod，并且这个 Pod 被调度器调度到了宿主机 A 上。这时候，Volume Controller 的 AttachDetachController 控制循环就会发现，上述 PVC 对应的 Volume，需要被 Attach 到宿主机 A 上。所以，AttachDetachController 会创建一个 VolumeAttachment 对象，这个对象携带了宿主机 A 和待处理的 Volume 的名字。

这样，StatefulSet 里的 External Attacher 容器，就会监听到这个 VolumeAttachment 对象的诞生。于是，它就会使用这个对象里的宿主机和 Volume 名字，调用同一个 Pod 里的 CSI 插件的 CSI Controller 服务的 ControllerPublishVolume 方法，完成“Attach 阶段”。

上述过程完成后，运行在宿主机 A 上的 kubelet，就会通过 VolumeManagerReconciler 控制循环，发现当前宿主机上有一个 Volume 对应的存储设备（比如磁盘）已经被 Attach 到了某个设备目录下。于是 kubelet 就会调用同一台宿主机上的 CSI 插件的 CSI Node 服务的 NodeStageVolume 和 NodePublishVolume 方法，完成这个 Volume 的“Mount 阶段”。

至此，一个完整的持久化 Volume 的创建和挂载流程就结束了。

思考题

请你根据编写 FlexVolume 和 CSI 插件的流程，分析一下什么时候该使用 FlexVolume，什么时候应该使用 CSI?

感谢你的收听，欢迎你给我留言，也欢迎分享给更多的朋友一起阅读。

版权归极客邦科技所有，未经许可不得转载

写留言

通过留言可与作者互动