k8s CSI

作者：肖晓

日期：*2023/12/26*

前置知识

attach & mount

1. Attach：将远程磁盘挂载到本地，成为一个主机上的一个块设备，通过`lsblk`命令可以查看到。

   VolumeAttachment： K8S 集群中记载的 pv 和某个 Node 的挂载关系。可以执行 `kubectl get volumeattachment |grep pv-name` 进行查看

   > *Attach* 这一步，由*`kube-controller-manager`*中的*`Volume Controller`*负责

2. Mount：本地有了新的块设备后，先将其格式化为某种文件系统格式后，就可以进行mount操作了。

   "将 volume 挂载到 pod 里的过程涉及到 kubelet。整个流程简单地说是，对应节点上的 **kubelet** 在创建 **pod** 的过程中，会调用 **CSI Node** 插件，执行 **mount** 操作。"

   > *Mount* 这一步，由*kubelet*中的*`VolumeManagerReconciler`*这个控制循环负责，它是一个独立于*kubelet*主循环的*goroutine*。

volume

https://kubernetes.io/zh-cn/docs/concepts/storage/volumes/

存储卷，是**pod**中能够被多个容器访问的共享目录。

临时卷类型的生命周期与 Pod 相同， 但持久卷可以比 Pod 的存活期长。当**Pod**不存在的时候，**K8S**也会销毁临时卷，不会销毁持久卷。在容器重启期间，Pod中任何类型的卷都不会丢失

临时卷：emptyDir、configMap、 downwardAPI、 secret 作为 本地临时存储 提供的。它们由各个节点上的 kubelet 管理。

卷不能挂载到其他卷之上（不过存在一种使用 subPath 的相关机制）

k8s的volume被定义在Pod上，然后被pod中的多个容器挂载到容器内部。使用：

在**Pod**上声明一个**Volume**，在容器中引用该**volume**并挂载**mount**到容器的目录

```
template:
   metadata:
     name: mynginx
     labels:
       app: mynginx
   spec:
     containers:
       - name: mynginx
         image: nginx
         imagePullPolicy: IfNotPresent
         volumeMounts:
           - mountPath: "/data"
             name: data
     restartPolicy: Always
     volumes:
       - name: data
         emptyDir: {}
```

k8s中的volume还支持容器配置文件集中化定义和管理 configmap资源

k8s提供的volume类型

1. **emptyDir**

   在 Pod 被指派到某节点时此卷会被创建。`emptyDir` 卷最初是空的，k8s自动分配一个目录。 **emptyDir Volume** 的生命周期与 **Pod** 一致：当 **Pod** 因为某些原因被从节点上删除时，**`emptyDir`** 卷中的数据也会被永久删除。容器崩溃期间 `emptyDir` 卷中的数据是安全的。

   `emptyDir` 主要用于那些需要在同一个 **Pod** 中的不同容器之间共享数据的场景

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: shared-data-pod
   spec:
     containers:
     - name: container-1
       image: nginx
       volumeMounts:
       - name: shared-volume
   ```

```
      mountPath: /data
  - name: container-2
    image: busybox
    volumeMounts:
    - name: shared-volume
      mountPath: /data
  volumes:
    - name: shared-volume
      emptyDir:
        sizeLimit: 500Mi
```

2. **hostPath**

将主机节点文件系统上的文件或目录挂载到**Pod** 中。通常用于以下场景：

- 运行一个需要访问 Docker 内部机制的容器；可使用 **`hostPath`** 挂载 **`/var/lib/docker`** 路径。
- 在容器中运行 cAdvisor（**Kubelet** 内置的容器资源收集工具） 时，以 `hostPath` 方式挂载 `/sys`。
- 允许 Pod 指定给定的 `hostPath` 在运行 Pod 之前是否应该存在，是否应该创建以及应该以什么方式存在。

存在的问题：

- HostPath 卷可能会暴露特权系统凭据（例如 Kubelet）或特权 API（例如容器运行时套接字），可用于容器逃逸或攻击集群的其他部分。
- 具有相同配置（例如基于同一 PodTemplate 创建）的多个 Pod 会由于节点上文件的不同而在不同节点上有不同的行为。可能会导致可移植性差，因为 Pod 只能在具有相同文件路径结构的主机上正常运行。
- 下层主机上创建的文件或目录只能由 root 用户写入。 你需要在特权容器中以 root 身份运行进程，或者修改主机上的文件权限以便容器能够写入 `hostPath` 卷。

```
  volumes:
  - name: test-volume
    hostPath:
      # 宿主机上目录位置
      path: /etc/kubernetes
      # 此字段为可选
      type: Directory  #在给定路径上必须存在的目录。 DirectoryOrCreate不存在则创建空目录
```

3. **local**

`local`卷所代表的是某个被挂载的本地存储设备，例如磁盘、分区或者目录。`local`卷只能作为静态创建的持久卷，不支持动态配置

与hostPath相比，**local**卷是可持久化并且可移植的，无需手动将pod调度到结点，系统通过查看PersistentVolume的节点亲和性配置，将 Pod 调度到具有特定标签的节点上，以便充分利用本地存储

如果结点变得不健康，那么local卷也将变得不可被Pod访问，使用它的Pod将不能运行，使用local卷的应用程序必须能够容忍这种可用性的降低，以及因底层磁盘的耐用性特征而带来的潜在的数据丢失风险。

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage #可以通过 StorageClass 进行动态分配 PV。
  local:
    path: /path/to/local/storage
   nodeAffinity: #节点亲和性配置
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage-class
provisioner: kubernetes.io/no-provisioner #表示这是一个静态存储类，不会动态地创建 PV。
volumeBindingMode: WaitForFirstConsumer
```

4. **configMap**

configMap 提供了向 Pod 注入配置数据的方法。ConfigMap对象中存储的数据可以被解析，然后被Pod中运行的容器化应用使用。

ConfigMap 提供了一种在不修改容器镜像的情况下更新配置的方式，使得 Pod 中的配置可以灵活地配置和更新。

在使用ConfigMap之前首先要创建它，并且ConfigMap总是以readOnly的模式挂载，容器以subPath卷挂载方式使用ConfigMap时，将无法接收ConfigMap的更新

```
#从 ConfigMap 中选择特定的键值对。
volumes:
   - name: config-vol
     configMap:
       name: log-config
        items:
          - key: log_level
            path: log_level
```

```
#Pod 中将创建一个卷，其中包含了 aiedge-auth-configmap ConfigMap 中的所有键值对。文件的默认权限模式被设置为 420volumes:
  - name: aiedge-auth-config-volume
    configMap:
      defaultMode: 420
      name: aiedge-auth-configmap
```

5. **secret**

secret 卷用来给 Pod 传递敏感信息，例如密码。你可以将 Secret 存储在 Kubernetes API 服务器上，然后以文件的形式挂载到 Pod 中，无需直接与 Kubernetes 耦合。secret 卷由 tmpfs（基于 RAM 的文件系统）提供存储，因此它们永远不会被写入非易失性（持久化的）存储器。

```
volumes:
  - name: aiedge-auth-config-volume
    configMap:
      defaultMode: 420
      name: aiedge-auth-configmap
   - name: jwt-prvkey-volume
    secret:
      defaultMode: 420
      secretName: jwt-prvkey-secret
```

```
#存储在 log-config文件中log_level 条目中的所有内容都被挂载到 Pod
volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level
```

下面为外部存储：

6. **NFS**

   `nfs` 卷能将 NFS (网络文件系统) 挂载到你的 Pod 中。 不像 `emptyDir` 那样会在删除 Pod 的同时也会被删除，`nfs` 卷的内容在删除 Pod 时会被保存，卷只是被卸载。 这意味着 `nfs` 卷可以被预先填充数据，并且这些数据可以在 Pod 之间共享。

```
volumes:
 - name: test-volume
   nfs:
     server: my-nfs-server.example.com
     path: /my-nfs-volume
     readOnly: true
```

7. **persistentVolumeClaim**

```
    volumes:
     - name: site-data
       persistentVolumeClaim:
         claimName: my-lamp-site-data
```

具体如下

8. **CSI**

   CSI（Container Storage Interface 容器存储接口）为容器编排系统定义标准接口，以将任意存储系统暴露给它们的容器工作负载

   一旦在K8S集群上部署了CSI兼容卷驱动程序，用户就可以使用CSI卷类型来接挂、挂载CSI驱动所提供的卷，csi卷可以在Pod中以三种方式使用：

   - 通过**PVC**对象引用
   - 使用一般性的临时卷
   - 使用CSI临时卷（驱动支持的情况下）

9. ...

挂载

volumeMounts字段

使用subPath：可用于指定所引用的卷内的子路径，而不是其根路径。

```
containers:
    - name: mysql
      image: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "rootpasswd"
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php:7.0-apache
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

PersistentVolume

https://kubernetes.io/zh-cn/docs/concepts/storage/persistent-volumes/

将存储如何供应的细节从其如何被使用中抽象出来。为了实现这点，我们引入了两个新的 API 资源：PersistentVolume 和 PersistentVolumeClaim。

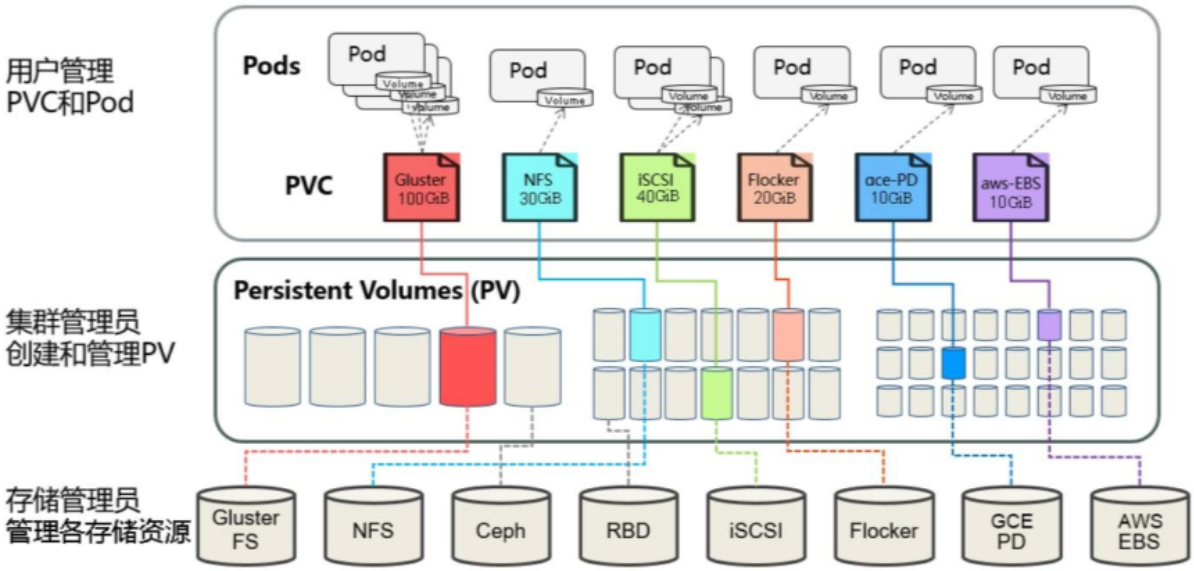
持久卷（**PersistentVolume**，**PV**） 是集群中的一块存储，可以由管理员事先制备， 或者使用存储类（**Storage Class**）来动态制备。 持久卷是集群资源，而不是像volume定义在pod上。

和普通**volumen**一样也是使用卷插件来实现的，只是它们拥有独立于任何使用 **PV** 的 **Pod** 的生命周期。 支持插件：csi、hostPath、nfs、local

制备方式：静态制备或动态制备

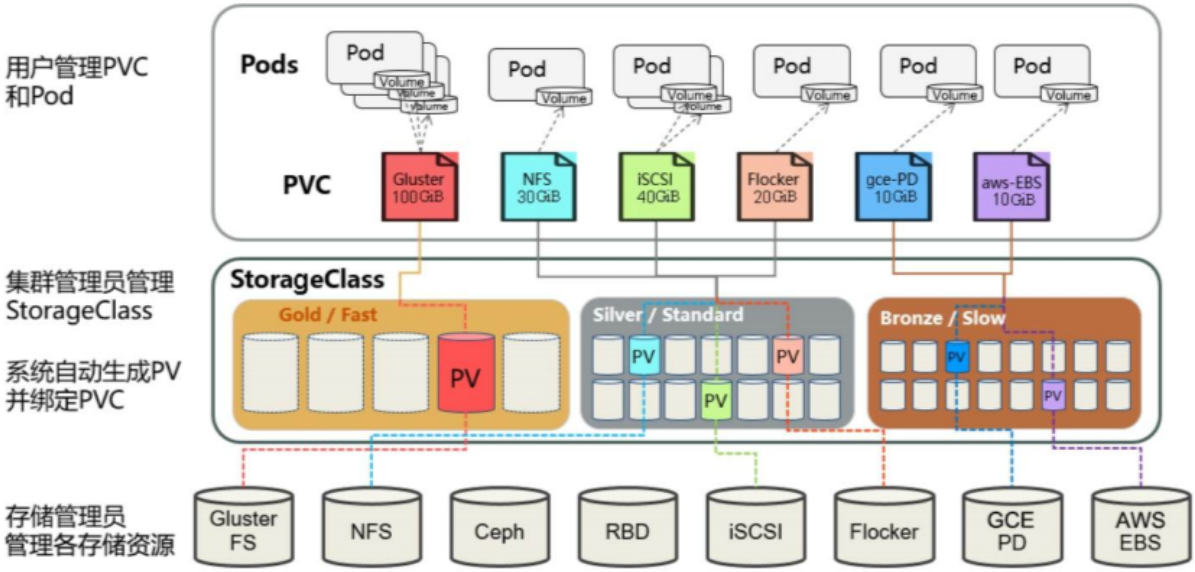
- 静态制备

  集群管理员创建若干 PV 卷。这些卷对象带有真实存储的细节信息，并且对集群 用户可用（可见）。PV 卷对象存在于 Kubernetes API 中，可供用户消费（使用）



- 动态制备

  无须手工创建**PV**，基于 `StorageClass` 来实现的：PVC 申领必须请求某个 存储类，同时集群管理员必须 已经创建并配置了该类，这样动态供应卷的动作才会发生。 如果 **PVC** 指定存储类为 `""`，则相当于为自身禁止使用动态供应的卷（见**csi-driver-nfs**的部署步骤的静态方式）。如下 StorageClass



支持的插件

PV 持久卷是用插件的形式来实现的。Kubernetes 目前支持以下插件：

https://kubernetes.io/zh-cn/docs/concepts/storage/persistent-volumes/

PV 持久卷是用插件的形式来实现的。Kubernetes 目前支持以下插件：

- `csi` - 容器存储接口 (CSI)
- `fc` - Fibre Channel (FC) 存储
- `hostPath` - HostPath 卷 （仅供单节点测试使用；不适用于多节点集群；请尝试使用 `local` 卷作为替代）
- `iscsi` - iSCSI (SCSI over IP) 存储
- `local` - 节点上挂载的本地存储设备
- `nfs` - 网络文件系统 (NFS) 存储

以下的持久卷已被弃用。这意味着当前仍是支持的，但是 Kubernetes 将来的发行版会将其移除。

- `azureFile` - Azure File （于 v1.21 **弃用**）
- `flexVolume` - FlexVolume （于 v1.23 **弃用**）
- `portworxVolume` - Portworx 卷 （于 v1.25 **弃用**）
- `vsphereVolume` - vSphere VMDK 卷 （于 v1.19 **弃用**）
- `cephfs` - CephFS 卷 （于 v1.28 **弃用**）
- `rbd` - Rados Block Device (RBD) 卷 （于 v1.28 **弃用**）

字段说明

https://kubernetes.io/zh-cn/docs/reference/kubernetes-api/config-and-storage-resources/persistent-volume-v1/

status状态：(生命周期)

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

下面是`spec`

capacity：

capacity 描述持久卷的资源和容量

accessModes属性：

- **ReadWriteOnce**：读写权限，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

volumeMode存储卷模式：

volumeMode: Filesystem / Block

volumeMode 定义一个卷是带着已格式化的文件系统来使用还是保持在原始块状态来使用。默认`Filesystem`

回收策略：`persistentVolumeReclaimPolicy`

定义当从PVC释放PV时会发生什么。 有效的选项为 Retain（保留 PV 的数据。）、 Delete（立即删除 PV 的数据。）和 Recycle（使 PV 重新变为"空闲"状态等待下一次绑定。已弃用）。 只有NFS和HostPath支持Recycle策略

- **保留（Retain)**

  手动创建 PersistentVolumes 所用的默认值。回收策略Retain使得用户可以手动回收资源，当**PVC**被删除的时候，**PV**仍然存在，对应的数据卷会被视为**Released**，卷仍属于当前用户，不会被其他PVC使用，只能被用户手动回收

  删除**PV**后，此**PV**的数据依然留存于外部的存储中。手工清理存储系统上依然留存的数据，可以再次创建、或者直接将其重新创建为PV

- **删除（Delete)**

  动态制备 PersistentVolumes 所用的默认值。对于支持Delete的卷插件，在**PVC**被删除后会直接移除**PV**对象、同时移除的还有**PV**相关的外部存储系统上的存储资产(asset)、支持这种操作的存储系统有AWS EBS、GCE PD、Azure Disk或Cinder。

节点亲和性：nodeAffinity

```
 affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
            - "node-1"  # 替换为希望 Pod 调度到的节点名
```

storageClassName

storageClassName 是这个持久卷所属于的 StorageClass 的名称。 空值意味着此卷不属于任何 **StorageClass**。未设置storageClassName的PV卷没有类设定，会被绑定到那些没有制定StorageClass的PVC

存储后端字段

根据PV支持的插件 `cephfs:（已经被弃用） nfs:`

```
#eg:cephfs
apiVersion: v1
kind: PersistentVolume
metadata:
  name: cephfs-pv
```

```yaml
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany  # 适用于多个 Pod 同时读写的情况
  persistentVolumeReclaimPolicy: Retain  # 可选，根据需求选择 Retain 或
Delete
  storageClassName: cephfs-storage  # 替换为你实际的 StorageClass 名称
  cephfs:
    monitors:
      - "ceph-mon-1:6789"
      - "ceph-mon-2:6789"
      - "ceph-mon-3:6789"
    user: "your-ceph-username"  # 替换为 CephFS 用户名
    secretRef:
      name: ceph-secret  # 替换为你存储 Ceph 密钥的 Kubernetes Secret 名称
    path: "/your/cephfs/path"  # 替换为 CephFS 的具体路径
```

```yaml
#eg:nfs
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  storageClassName: nfs-csi
  nfs:
    path: /tmp  #必需
    server: 192.168.20.235 #必需
```

CSI表示由一个外部 CSI 卷驱动管理的存储

- **csi.driver** (string)，必需

  driver 是此卷所使用的驱动的名称。必需。

- **csi.volumeHandle** (string)，必需

  volumeHandle 是 CSI 卷插件的 CreateVolume 所返回的唯一卷名称，用于在所有后续调用中引用此卷。必需。

- **csi.controllerExpandSecretRef** (SecretReference)

  controllerExpandSecretRef 是对包含敏感信息的 Secret 对象的引用， 该 Secret 会被传递到 CSI 驱动以完成 CSI

- **csi.readOnly** (boolean)

  传递到 ControllerPublishVolumeRequest 的 readOnly 值。默认为 false（读/写）。

- **csi.volumeAttributes** (map[string]string)

  要发布的卷的 volumeAttributes。查找对应的文档

  关于nfs-driver的参数：https://github.com/kubernetes-csi/csi-driver-nfs/blob/master/docs/driver-parameters.md

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-csi-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany  # 适用于多个 Pod 同时读写的情况
  persistentVolumeReclaimPolicy: Retain  # 可选，根据需求选择 Retain 或 Delete
  storageClassName: nfs-csi-storage  # 不为空""，因此可以根据storageClassName相同来和pvc绑定
  csi:
    driver: nfs.csi.k8s.io
    volumeHandle: 192.168.20.235/home/aiedge/csiTest/sharepv  # 格式：{nfs-服务器地址}/{子目录名称}/{共享名称}，确保该值对于集群中的每个共享都是唯一的
    volumeAttributes:
      server: 192.168.20.235
      share: /home/aiedge/csiTest/
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
            - "node-1"  # 替换为希望 Pod 调度到的节点名

---
# StorageClass
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi-storage
provisioner: nfs.csi.k8s.io
---
# PersistentVolumeClaim
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-csi-pvc
spec:
  storageClassName: nfs-csi-storage
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

persistentVolumeClaim

持久卷申领（**persistentVolumeClaim**，**PVC**） 用来将持久卷PV挂载到 Pod 中，表达的是用户对存储的请求。 **Pod** 会耗用节点资源，而 **PVC** 会耗用 **PV** 资源。 PVC是用户在不知道特定云环境细节的情况下“申领”持久存储（例如 iSCSI 卷）的一种方法。**pvc**可以向**pv**申请指定大小的存储资源并设置访问模式

资源请求、访问模式、存储卷模式 与PV定义类似

绑定： 如果系统中没有满足PVC要求的PV，PVC会一直处于pending **pod**可以挂载**PVC**，就能持续独占使用。多个**pod**可以挂载同一个**PVC**

字段说明

https://kubernetes.io/zh-cn/docs/reference/kubernetes-api/config-and-storage-resources/persistent-volume-claim-v1/

PersistentVolumeClaimSpec

- **accessModes** ([]string)

  accessModes 包含卷应具备的预期访问模式。（同上面pv）

- **selector** (LabelSelector)

  selector 是在绑定时对卷进行选择所执行的标签查询。

- **resources** (ResourceRequirements)

  会根据这个需求来选择适当的 PersistentVolume（PV）来绑定PVC，满足 PVC 的存储需求。

  resources.requests表示卷应拥有的最小资源。如果针对容器省略 requests，则在显式指定的情况下默认为 limits，否则为具体实现所定义的值。请求不能超过限制。

  resources.limits描述允许的最大计算资源量。

- **volumeName** (string)

volumeName 是对此申领所对应的 **PersistentVolume** 的绑定引用。

- **storageClassName** (string)

  storageClassName 是此申领所要求的 **StorageClass** 名称

- **volumeMode** (string)

  volumeMode 定义申领需要哪种类别的卷。当申领规约中未包含此字段时，意味着取值为 Filesystem。

绑定PersistentVolume & persistentVolumeClaim

> - *PersistentVolume（PV）*：对存储资源创建和使用的抽象，使得存储作为集群中的资源管理
> - *PersistentVolumeClaim（PVC）*：让用户不需要关心具体的*Volume*实现细节

PVC 申领与 **PV** 卷之间的绑定是一种一对一的映射，实现上使用 ClaimRef 来记述 PV 卷与 PVC 申领间的双向绑定关系。

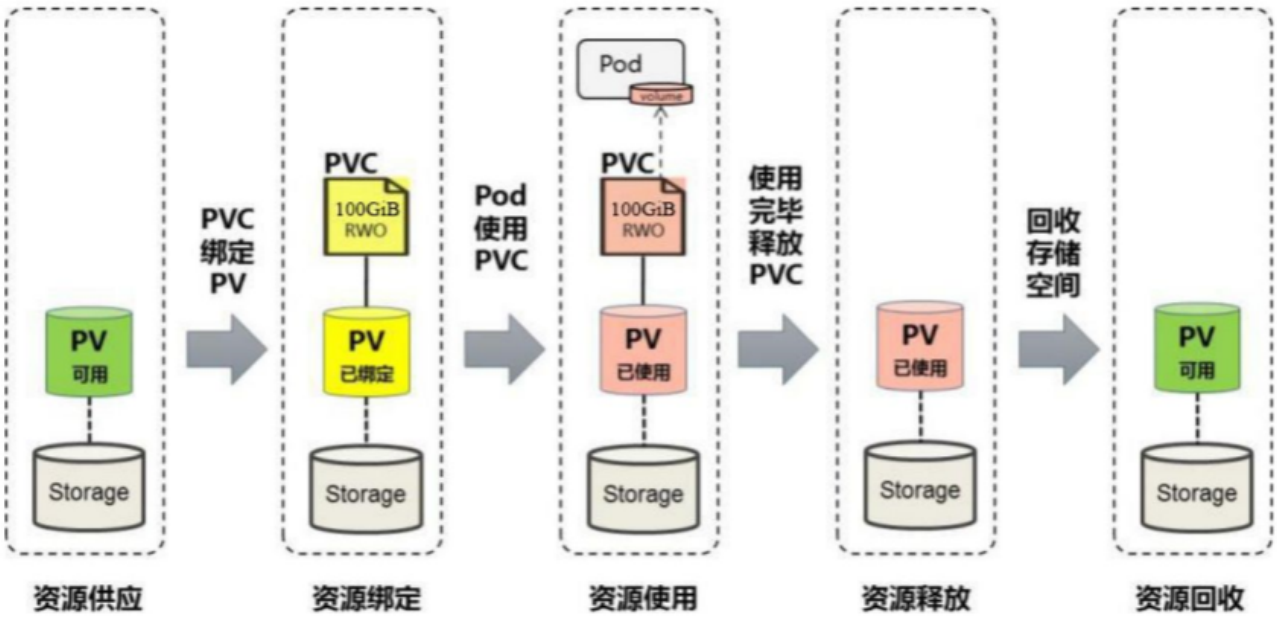
当**Pod** 将 **PVC** 申领当做存储卷来使用。集群会检视 **PVC** 申领，找到所绑定的卷，并 为 **Pod** 挂载该卷。

根据什么进行绑定？

- PV和PVC的storageClassName值相同
- 对卷进行选择所执行的标签查询
- 查找pv中满足 满足PVC中resources需求的
- pv pvc的volumeMode满足要求

静态制备卷的卷绑定矩阵：

PV volumeMode	PVC volumeMode	Result
未指定	未指定	绑定
未指定	Block	不绑定
未指定	Filesystem	绑定
Block	未指定	不绑定
Block	Block	绑定
Block	Filesystem	不绑定
Filesystem	Filesystem	绑定
Filesystem	Block	不绑定
Filesystem	未指定	绑定



StorageClass

集群管理员需要能够提供不同性质的 PersistentVolume， 并且这些 PV 卷之间的差别不仅限于卷大小和访问模式，同时又不能将卷是如何实现的这些细节暴露给用户。 为了满足这类需求，就有了存储类（**StorageClass**） 资源。

SC 为管理员提供了一种动态提供存储卷的“类”模板，**SC** 中的 **.Spec** 中详细定义了存储卷 **PV** 的不同服务质量级别、备份策略等等；

StorageClass定义哪一个驱动将被使用和哪些参数将被传递给驱动。

一方面减少了用户对存储资源细节的关注；另一方面减轻了手工管理**PV**的工作，由系统自动完成**PV**的创建和绑定。

字段说明

https://kubernetes.io/zh-cn/docs/reference/kubernetes-api/config-and-storage-resources/storage-class-v1/

- **provisioner**：存储资源的提供者（后端存储驱动） 必需
- **parameters**：就是生成出来的PV的参数。不同的provisioner有不同的参数设置

```
#nfs
parameters:
  server: 192.168.20.235        #必需
  share: /home/aiedge/csiTest   #必需
```

- **reclaimPolicy** ：控制此存储类动态制备的 PersistentVolume 的 reclaimPolicy。默认为 Delete。
- **volumeBindingMode**：卷绑定和动态制备应该发生在什么时候，当未设置时，默认使用 `Immediate` 模式，一旦创建了 PersistentVolumeClaim 也就完成了卷绑定和动态制备。

StorageClass交给系统自动完成PV的动态创建以及与PVC的绑定

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: 192.168.20.235
  share: /home/aiedge/csiTest
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

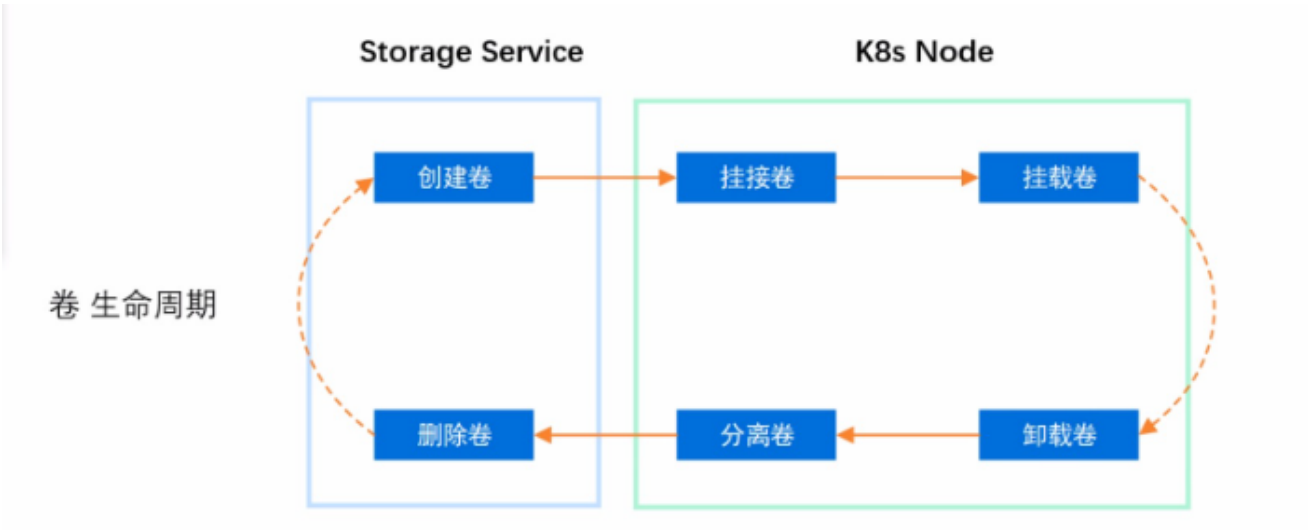
```yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: nfs-csi
```

卷快照

`VolumeSnapshotContent` 和 `VolumeSnapshot` 这两个 API 资源用于给用户和管理员创建卷快照。与 `PersistentVolume` 和 `PersistentVolumeClaim` 这两个 API 资源用于给用户和管理员制备卷类似。

- `VolumeSnapshotContent` 是从一个卷获取的一种快照，该卷由管理员在集群中进行制备。 就像持久卷（PersistentVolume）是集群的资源一样，它也是集群中的资源。
- `VolumeSnapshot` 是用户对于卷的快照的请求。它类似于持久卷声明（PersistentVolumeClaim）。
- `VolumeSnapshotClass` 允许指定属于 `VolumeSnapshot` 的不同属性。

存储卷的生命周期




```
CreateVolume +------------+ DeleteVolume
+------------->|  CREATED   +--------------+
|              +---+----+---+              |
|       Controller |    | Controller       v
+++         Publish |    | Unpublish       +++
|X|          Volume |    | Volume          | |
```

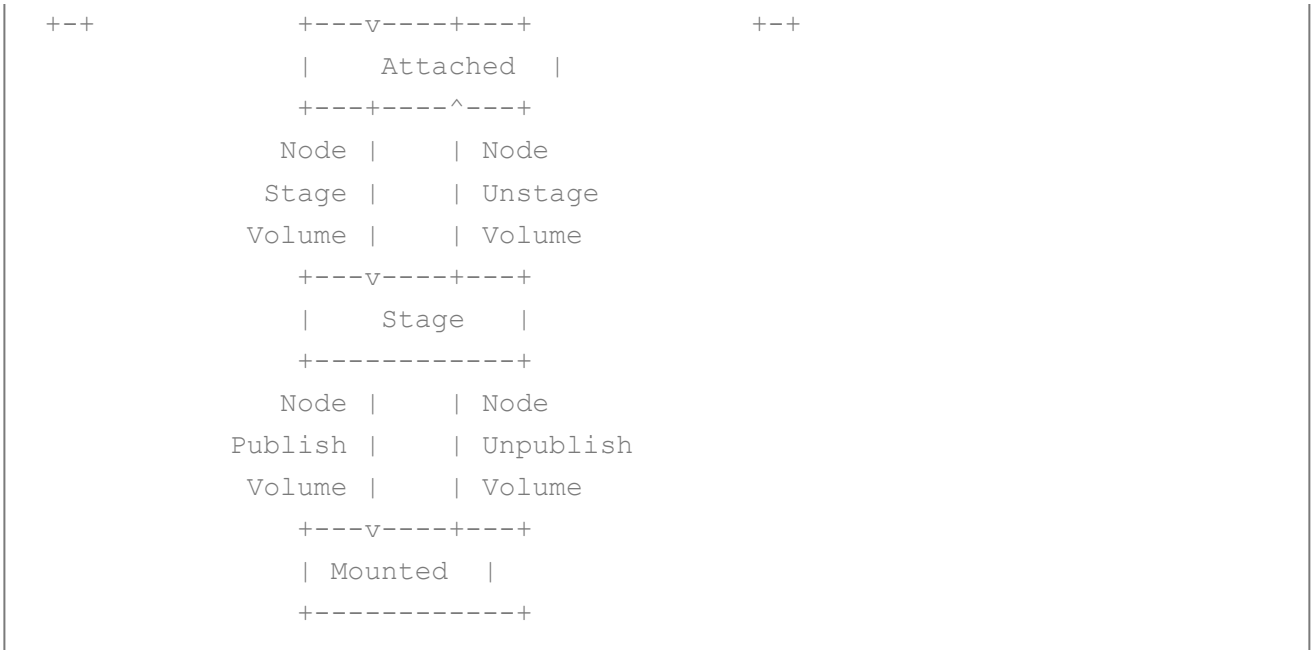


Figure 6: The lifecycle of a dynamically provisioned volume, from creation to destruction, when the Node Plugin advertises the STAGE_UNSTAGE_VOLUME capability.

【注： Pod 使用 PV 的基本流程，需要根据我们的 Volume 类型来决定需要做什么操作】

创建卷-挂接到节点-预处理卷-挂载到**Pod**

CSI

> 参考链接：
>
> *https://kubernetes-csi.github.io/docs/introduction.html*
>
> *https://kingjcy.github.io/post/cloud/paas/base/kubernetes/k8s-store-csi/#provisioning-volumes*
>
> *https://github.com/kubernetes/design-proposals-archive/blob/main/storage/container-storage-interface.md*

背景&说明

Container Storage Interface (CSI)，k8s 1.9引入CSI，v1.13GA。是一种标准，旨在能为容器编排引擎和存储系统间建立一套标准的存储调用接口，实现解耦，通过该接口能为容器编排引擎提供存储服务。

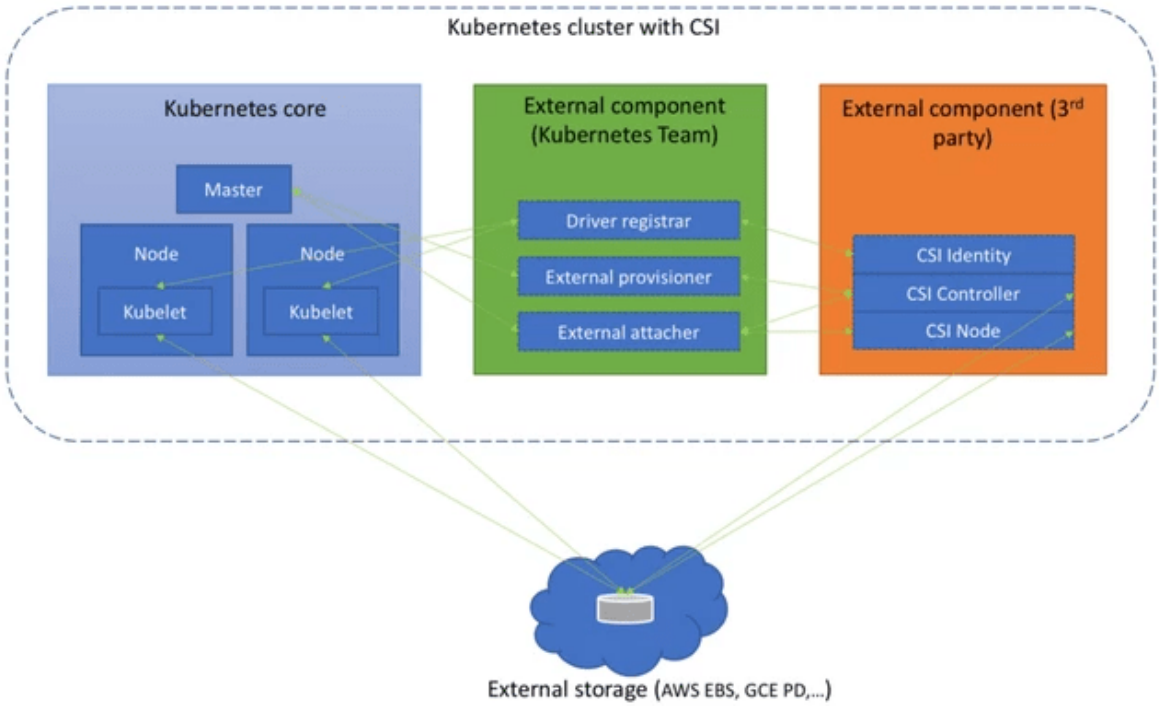
> 背景：
>
> *In-Tree*卷插件：与 *Kubernetes* 的核心组件一同构建、链接、编译和交付的。

> *Out-of-Tree* 卷插件：它们使存储供应商能够创建自定义存储插件，而无需将插件源码添加到 *Kubernetes* 代码仓库。主要是通过 *gRPC* 接口跟 *K8s* 组件交互。
>
> *CSI*使得 *out-of-tree* 的第三方 *CSI* 驱动能够被 *Kubernetes* 使用。 *K8s* 提供了大量的 *SideCar* 组件来配合 *CSI* 插件实现丰富的功能，所用到的组件分为 *SideCar* 组件和第三方需要实现的插件。
>
> 以前，所有卷插件（如上面列出的卷类型）都是“树内（*In-Tree*）”的，数据卷插件的开发与 *Kubernetes* 紧密耦合，更新迭代慢、灵活性比较差。

CSI的目的：

- 定义 Kubernetes API 以与任意第三方 CSI 卷驱动程序交互。
- 定义 Kubernetes 主节点和节点组件与任意第三方 CSI 卷驱动程序安全通信的机制。
- 定义 Kubernetes 主节点和节点组件发现并注册部署在 Kubernetes 上的任意第三方 CSI 卷驱动程序的机制。
- 建议兼容 Kubernetes 的第三方 CSI 卷驱动程序的打包要求。
- 建议在 Kubernetes 集群上部署兼容 Kubernetes 的第三方 CSI 卷驱动程序。

CSI：



- 由**k8s**官方维护的一系列**external**组件负责注册**CSI driver** 或监听**k8s**对象资源，从而发起**csi driver**调用，比如（node-driver-registrar，external-attacher，external-provisioner，external-resizer，external-snapshotter，livenessprobe）
- 各云厂商**or**开发者自行开发的组件（需要实现CSI Identity，CSI Controller，CSI Node 接口）

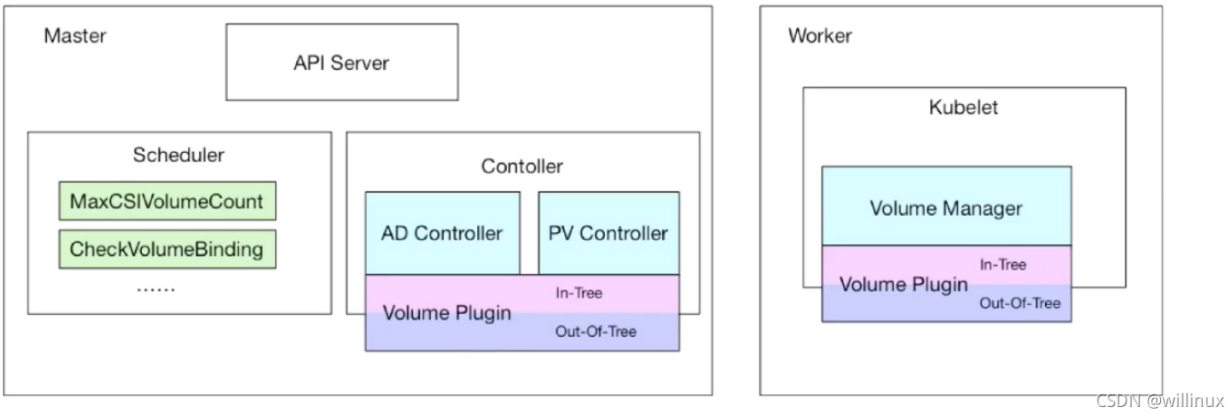
支持的drivers

https://kubernetes-csi.github.io/docs/drivers.html

k8s官方提供：csi-driver-nfs、csi-driver-smb、csi-driver-host-path

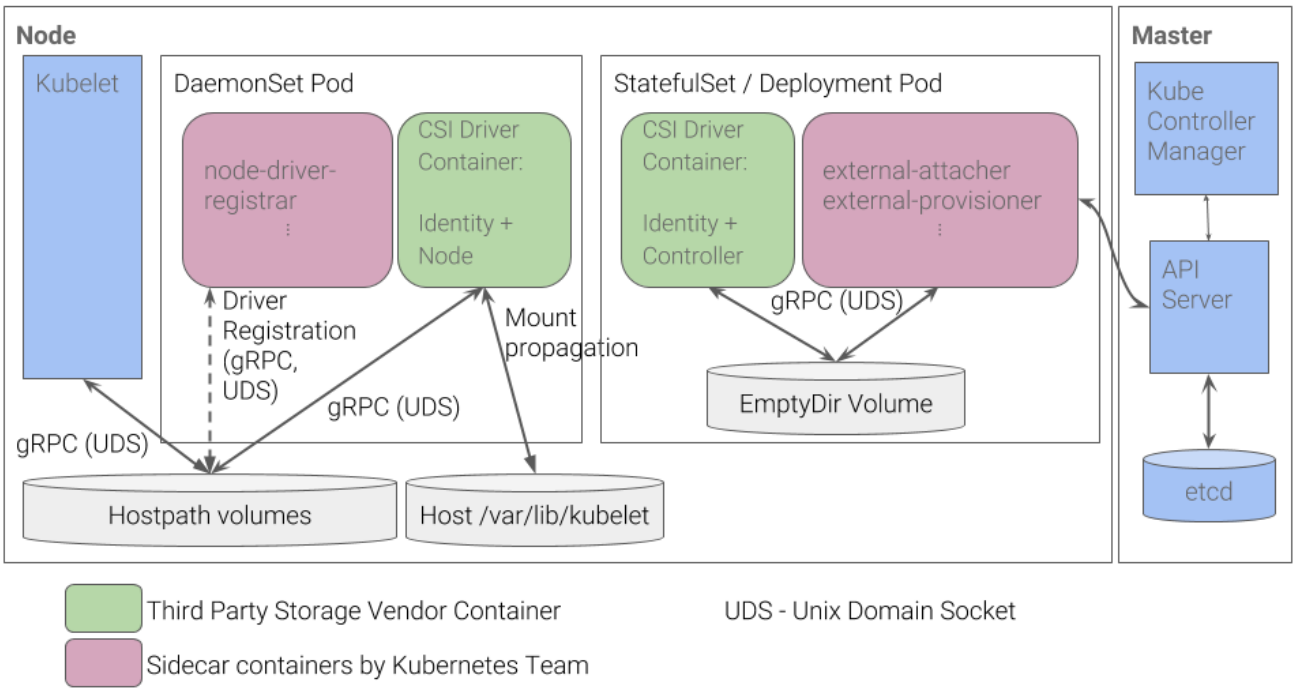
架构

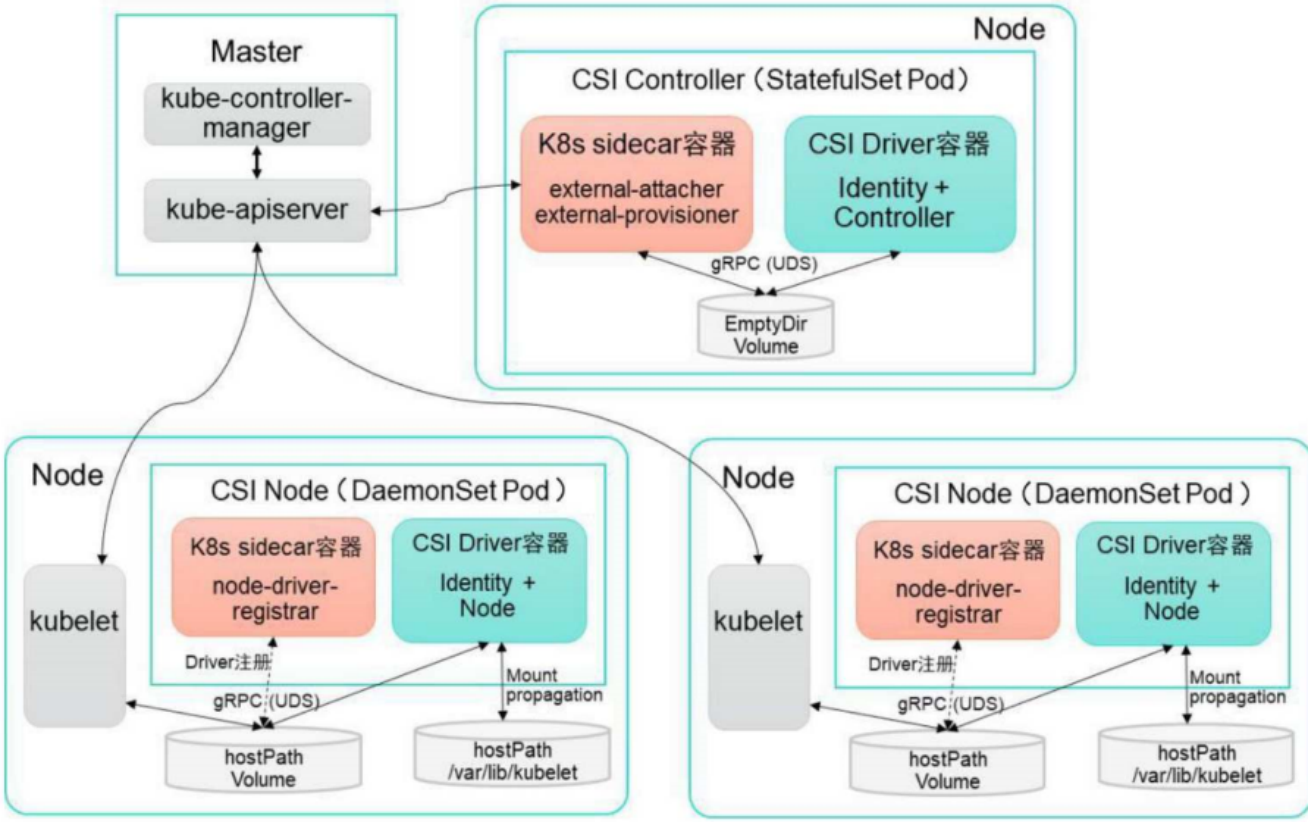
k8s的存储结构图：



- PV Controller：负责 PV/PVC 的绑定，并根据需求进行数据卷的 Provision/Delete 操作
- AD Controller：负责存储设备的 Attach/Detach 操作，将设备挂载到目标节点
- Volume Manager：管理卷的 Mount/Unmount 操作、卷设备的格式化等操作
- Volume Plugin：扩展各种存储类型的卷管理能力，实现第三方存储的各种操作能力和 Kubernetes 存储系统结合

在 Kubernetes 上整合 CSI 插件的整体架构如下图所示：





CSI实现中的组件分为两部分：

- **Controller Server Pod**：以**StatefulSet**或**Deployment**的形式部署。提供存储服务视角对存储资源和存储卷进行管理和操作，关注于存储系统 **API** 调用。设置副本数量为**1**，保证为一种存储插件只运行一个控制器实例，可以在任意一个节点上。

  组成：由实现 CSI 控制器服务的 CSI 驱动程序(CSI Driver)和多个 sidecar 容器组成。sidecar 容器，如 `External Provisioner 和 External Attacher` 等组件,在部署中sidecar里的组件是可选的。这些 sidecar 容器通常与 Kubernetes 对象交互并调用驱动程序的 CSI 控制器服务。<见部署的yaml>

- **Node Server Pod**：以**DaemonSet**的形式部署。关注于主机（**Node**）使用存储卷，对主机（Node）上的Volume进行管理和操作。在每个**Node**上都运行一个**Pod**，以便 Kubelet 可以调用。

  组成：由实现 CSI 节点服务的 CSI 驱动程序(CSI Driver)和节点驱动程序注册器 sidecar 容器(`node-driver-registrar`)组成。

  通信：

  - **node-driver-registrar**容器与**kubelet**通过 Node主机一个`hostPath`目录下的 `unix socket`进行通信。
  - **CSI Driver**容器与**kubelet**通过Node主机另一个`hostPath`目录下的 `unix socket` 进行通信，同时需要将 kubelet 的工作目录（默认为 `/var/lib/kubelet`）挂载给**CSI Driver**容器，用于为Pod进行Volume的管理操作 (包括mount、 umount等）。

SideCar 组件

> 在*Kubernetes*中，*Sidecar*模式可以通过容器组的方式实现，即多个容器共同运行在同一个*Pod*中。这种模式使得不同功能的容器能够协同工作，提供更灵活的部署和管理选项。
>
> 这样做的好处：
>
> - *Reduction of "boilerplate" code.*减少重复性代码
> - *CSI Driver developers do not have to worry about complicated, "Kubernetes specific" code.*
> - *Separation of concerns.*
> - *Code that interacts with the Kubernetes API is isolated from (and in a different container then) the code that implements the CSI interface.* *CSI* 驱动的核心逻辑实现放 *CSI* 接口，而与 *Kubernetes API* 交互的逻辑则可以被放入 *sidecar* 容器中

会使用到的sidecar如下：

♥ external-provisioner

如果CSI插件要实现CREATE__DELETE__VOLUME能力（即动态供应**PV**），则CSI插件需要实现Controller Service的`CreateVolume`、`DeleteVolume`接口，并配合上该sidecar就可以了。监听 **PVC** 对象：这样当watch到指定StorageClass的 PersistentVolumeClaim资源状态变更，会自动地调用这两个接口。

♥ external-attacher

如果CSI插件要实现PUBLISH__UNPUBLISH__VOLUME能力，则CSI插件需要实现Controller Service的`ControllerPublishVolume`、`ControllerUnpublishVolume`接口，并配合上该sidecar就可以了。监听**VolumeAttachment**对象（由**AD**控制器创建的）：这样当watch到VolumeAttachment资源状态变更，会自动地调用这两个接口来实现插拔存储卷（**attach/detach**）。将 volume 附着到 node 上，或从 node 上删除。

因为 K8s 内部的 Attach/Detach Controller 不会直接调用 CSI driver 的接口。

> 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。
>
> 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

external-resizer

监听 PVC 对象的容量发生变更，用来对 volume 进行扩容。

external-snapshotter

卷快照相关。如果CSI插件要实现CREATE_DELETE_SNAPSHOT能力，则CSI插件需要实现Controller Service的`CreateSnapshot`、`DeleteSnapshot`接口，并配合上该sidecar就可以了。这样当watch到指定SnapshotClass的VolumeSnapshot资源状态变更，会自动地调用这两个接口。

external-health-monitor-controller

通过调用 CSI driver Controller 服务的 `ListVolumes` 或者 `ControllerGetVolume` 接口，来检查 **CSI volume** 的健康情况，并上报在 **PVC** 的 **event** 中。

livenessprobe

负责监测 **CSI driver**相关**pod** 的健康情况，并通过 Liveness Probe 机制汇报给 k8s，当监测到 CSI driver 有异常时负责重启 pod。

♥node-driver-registrar

配合上 CSI driver Node 服务的 `NodeGetInfo` 接口。当CSI Node Plugin部署到kubernetes的node节点时，该**sidecar**会自动调用接口获取**CSI**插件信息，并向**kubelet**进行注册。

external-health-monitor-agent

通过调用 CSI driver Node 服务的 `NodeGetVolumeStats` 接口，来检查 **CSI volume** 的健康情况，并上报在 **pod** 的 **event** 中。

CSI Driver容器

第三方存储提供方（即 SP，Storage Provider）需要实现 Controller 、Node、Identity插件。

支持的插件：https://kubernetes-csi.github.io/docs/drivers.html

CSI Identity

用于提供 **CSI driver** 的身份信息，Controller 和 Node 都需要实现。

CSI Controller

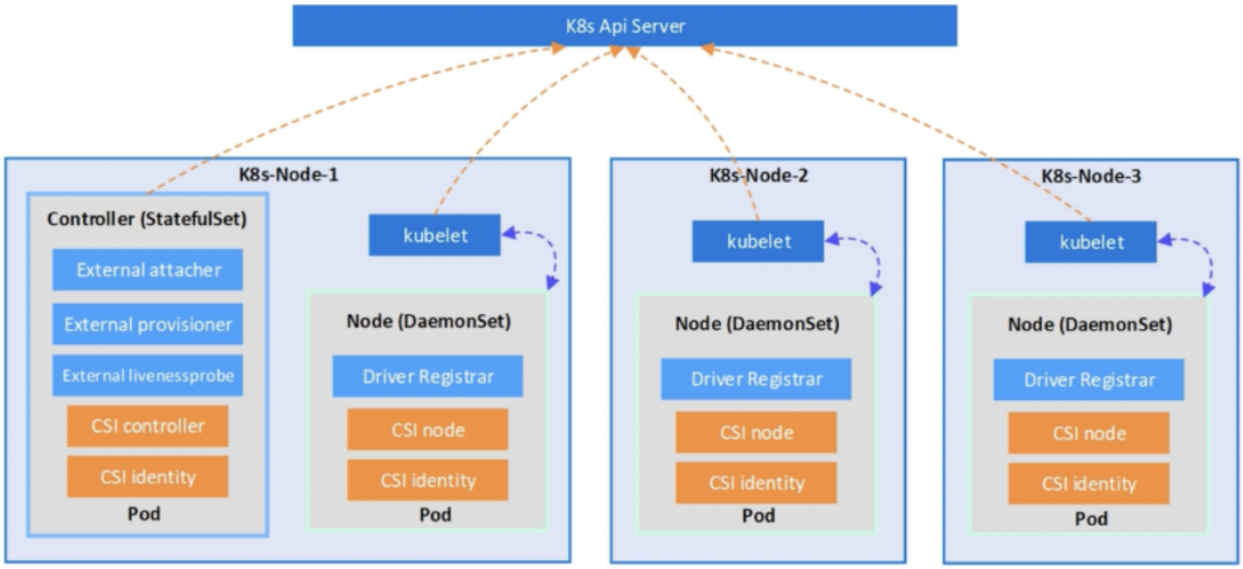
用于实现创建/删除 **volume**（**Provision** ）、**attach/detach volume**（**Attach** ）、**volume** 快照、**volume** 扩缩容等功能，Controller 插件需要实现这组接口。

CSI Node

用于实现 **mount/umount volume**、检查 **volume** 状态等功能

> *NodeStageVolume*实现了多个 *pod* 共享一个 *volume* 的功能，支持先将 *volume* 挂载到一个临时目录，然后通过 `NodePublishVolume` 将其挂载到 *pod* 中

CSI的实际部署



以nfs为例：<见部署yaml>

CSIDriver

需要创建`CSIDriver`资源对象。

```yaml
---
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: nfs.csi.k8s.io
spec:
  attachRequired: false
  volumeLifecycleModes:
    - Persistent
  fsGroupPolicy: File
```

```
aiedge@xx-test-master235:~/csi-driver-nfs$ kubectl get csidriver
NAME             ATTACHREQUIRED   PODINFOONMOUNT   STORAGECAPACITY   TOKENREQUESTS   REQUIRESREPUBLISH   MODES
        AGE
nfs.csi.k8s.io   false            false            false             <unset>         false               Persistent,Ep
hemeral   15d
```

```
aiedge@xx-test-master235:~/csi-driver-nfs$ kubectl describe csidriver/nfs.csi.k8s.io
Name:         nfs.csi.k8s.io
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  storage.k8s.io/v1
Kind:         CSIDriver
Metadata:
  Creation Timestamp:  2023-12-05T01:58:51Z
  Managed Fields:
    API Version:  storage.k8s.io/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .:
          f:kubectl.kubernetes.io/last-applied-configuration:
      f:spec:
        f:attachRequired:
        f:fsGroupPolicy:
        f:podInfoOnMount:
        f:requiresRepublish:
        f:storageCapacity:
        f:volumeLifecycleModes:
          .:
          v:"Ephemeral":
          v:"Persistent":
    Manager:         kubectl-client-side-apply
    Operation:       Update
    Time:            2023-12-05T01:58:51Z
  Resource Version:  5744571
  UID:               5bf8b63d-2600-46a4-8042-a1883b05b19e
Spec:
  Attach Required:     false
  Fs Group Policy:     File
  Pod Info On Mount:   false
  Requires Republish:  false
  Storage Capacity:    false
  Volume Lifecycle Modes:
    Persistent
    Ephemeral
Events:  <none>
```

Controller Pod和Node Pod

```
aiedge@xx-test-master235:~/csi-driver-nfs$ kubectl get po -nkube-system -owide
NAME                                    READY   STATUS    RESTARTS       AGE    IP               NODE
coredns-6d8c4cb4d-rdqln                 1/1     Running   3 (14d ago)    120d   10.244.0.11      xx-test-master235
coredns-6d8c4cb4d-xl485                 1/1     Running   4 (14d ago)    133d   10.244.0.12      xx-test-master235
csi-nfs-controller-78d56d9785-xxdjt     3/3     Running   1 (14d ago)    14d    192.168.20.236   xx-test-node236
csi-nfs-node-249fj                      3/3     Running   0              14d    192.168.20.236   xx-test-node236
csi-nfs-node-lc6qj                      3/3     Running   0              14d    192.168.20.235   xx-test-master235
csi-nfs-node-rcwqs                      3/3     Running   0              14d    192.168.20.239   xx-test-node239
etcd-xx-test-master235                  1/1     Running   13 (14d ago)   133d   192.168.20.235   xx-test-master235
```

CSINode

Kubernetes `CSINode`API 对象：

CSINode 用于将 CSI 驱动程序绑定到节点上，表示节点上的 CSI 驱动程序插件，是节点级别的资源对象。它是集群中的节点信息，在 **Node Driver Registrar** 组件向 Kubelet 注册完毕后，Kubelet 会创建该资源，故不需要显式创建 CSINode 资源。它的作用是每一个新的 CSI Plugin 注册后，都会在 CSINode 列表里添加一个 CSINode 信息。

- CSINode 对象用于告知 Kubernetes 集群该节点上可用的 CSI 驱动程序，以便在调度 Pod 时进行选择和匹配；

- CSINode 对象是节点级别的，每个节点上都需要创建一个对应的 CSINode 对象；

```
root@xx-test-master235:/home/aiedge# kubectl get csinode xx-test-node236 -o yaml
apiVersion: storage.k8s.io/v1
kind: CSINode
metadata:
  annotations:
    storage.alpha.kubernetes.io/migrated-plugins: kubernetes.io/aws-ebs,kubernetes.io/azure-disk,kubernetes.io/cinder,kubernetes
/gce-pd
  creationTimestamp: "2023-08-21T09:40:57Z"
  name: xx-test-node236
  ownerReferences:
  - apiVersion: v1
    kind: Node
    name: xx-test-node236
    uid: 81bc11a0-6f3a-4e91-887c-44d5ee6a9f57
  resourceVersion: "5751806"
  uid: d778db91-c952-40b4-a4e2-fd933f665bba
spec:
  drivers:
  - name: nfs.csi.k8s.io
    nodeID: xx-test-node236
    topologyKeys: null
```

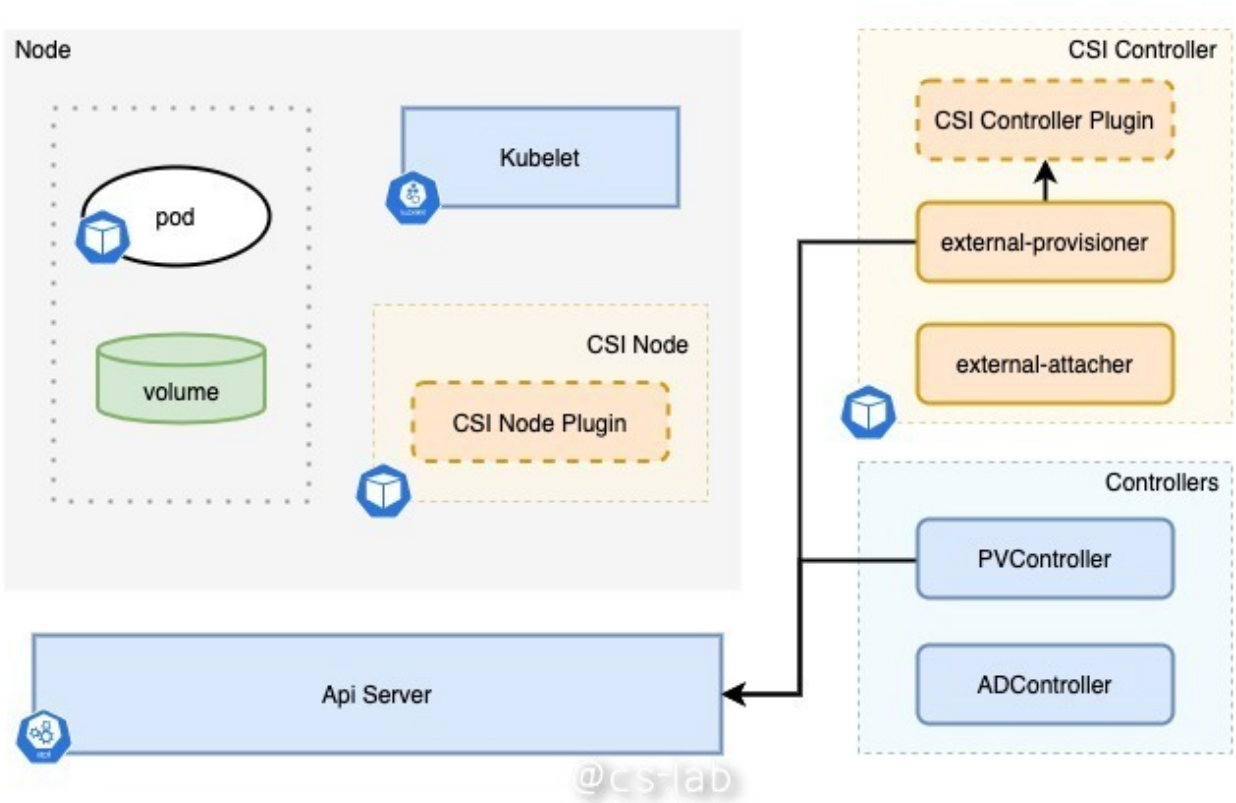

工作流程：

pod 挂载 volume

pod 挂载 **volume** 的整个工作流程：整个流程流程分别三个阶段：**Provision/Delete**、**Attach/Detach**、**Mount/Unmount**，不过不是每个存储方案都会经历这三个阶段，比如 NFS 就没有 Attach/Detach 阶段。

整个过程不仅仅涉及到上面介绍的组件的工作，还涉及 ControllerManager 的 AttachDetachController 组件和 PVController 组件以及 kubelet。

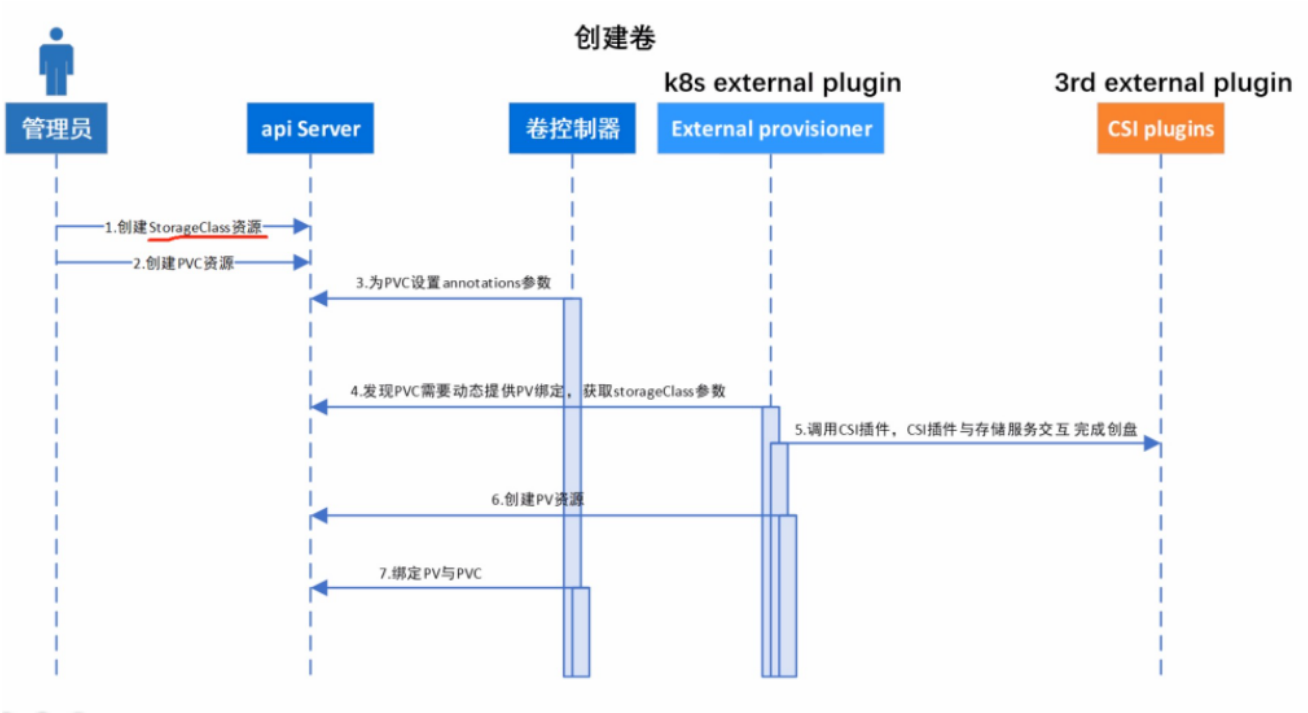
Provision（创盘/删盘）



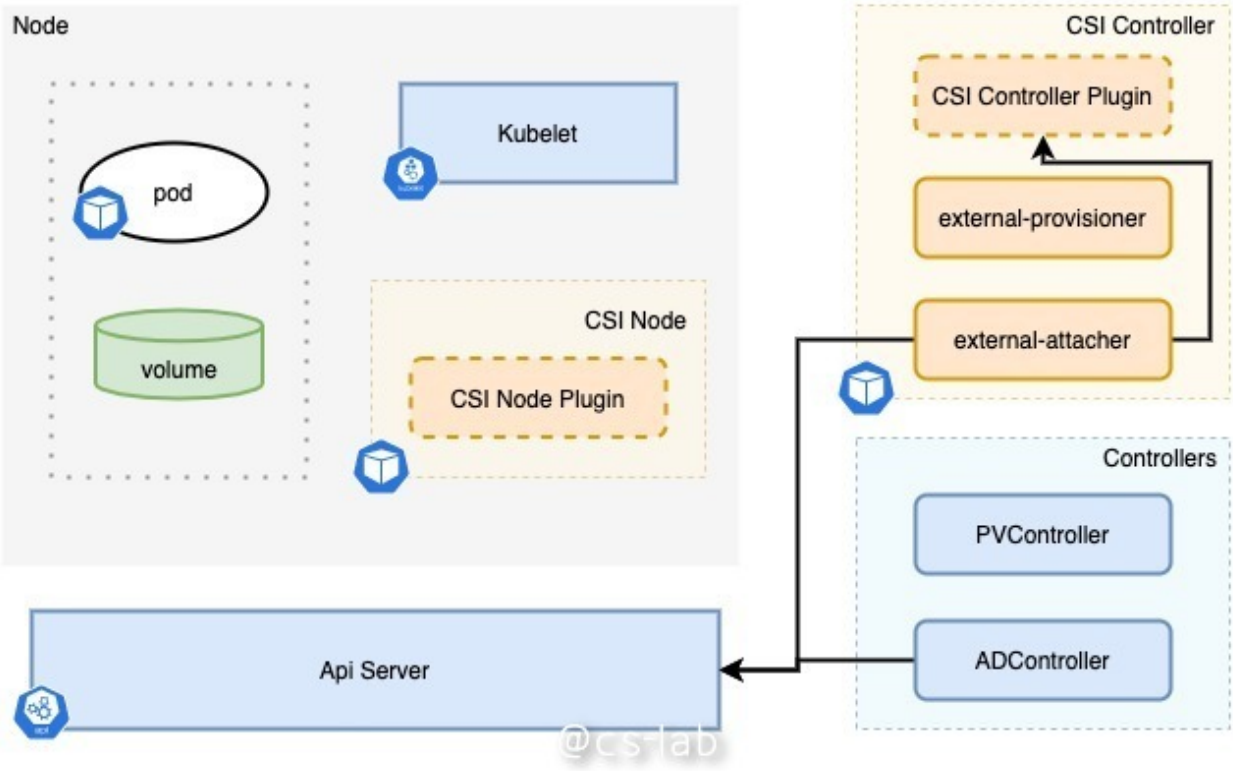
先来看 Provision 阶段，整个过程如上图所示。其中 extenal-provisioner 和 PVController 均 watch PVC 资源。

（动态创建**pv**才会走到**provision**流程，静态并不会）

1. 当 **PVController** watch 到集群中有 PVC 创建（初始处于 Pending 状态）时，会判断当前是否有 in-tree plugin 与之相符，如果没有则判断其存储类型为 out-of-tree 类型，于是给 PVC 打上注解 `volume.beta.kubernetes.io/storage-provisioner={csi driver name}`；
2. 当 extenal-provisioner watch 到 PVC 的注解 csi driver 与自己的 csi driver 一致时，调用 CSI Controller 的 `CreateVolume` 接口；
3. 当 CSI Controller 的 `CreateVolume` 接口返回成功时，extenal-provisioner 会在集群中创建对应的 PV；
4. PVController watch 到集群中有 PV 创建时，将 PV 与 PVC 进行绑定。

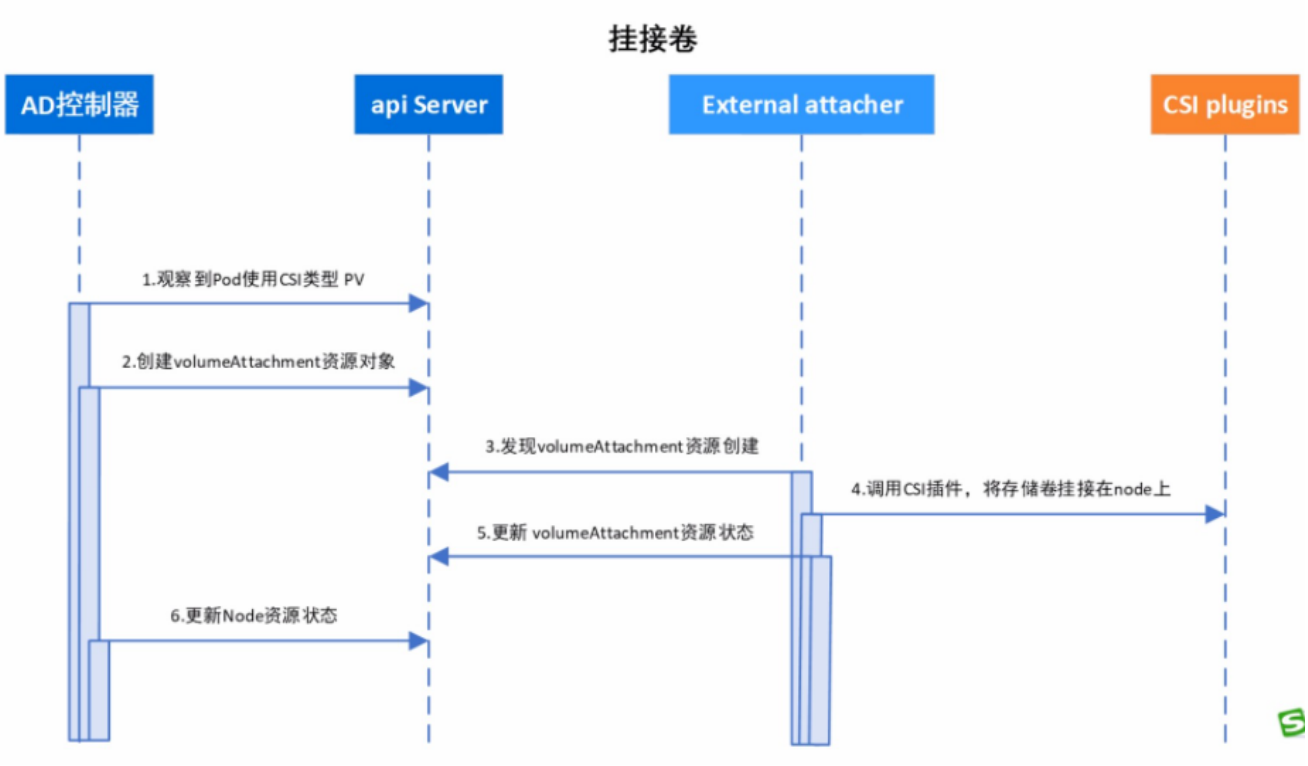


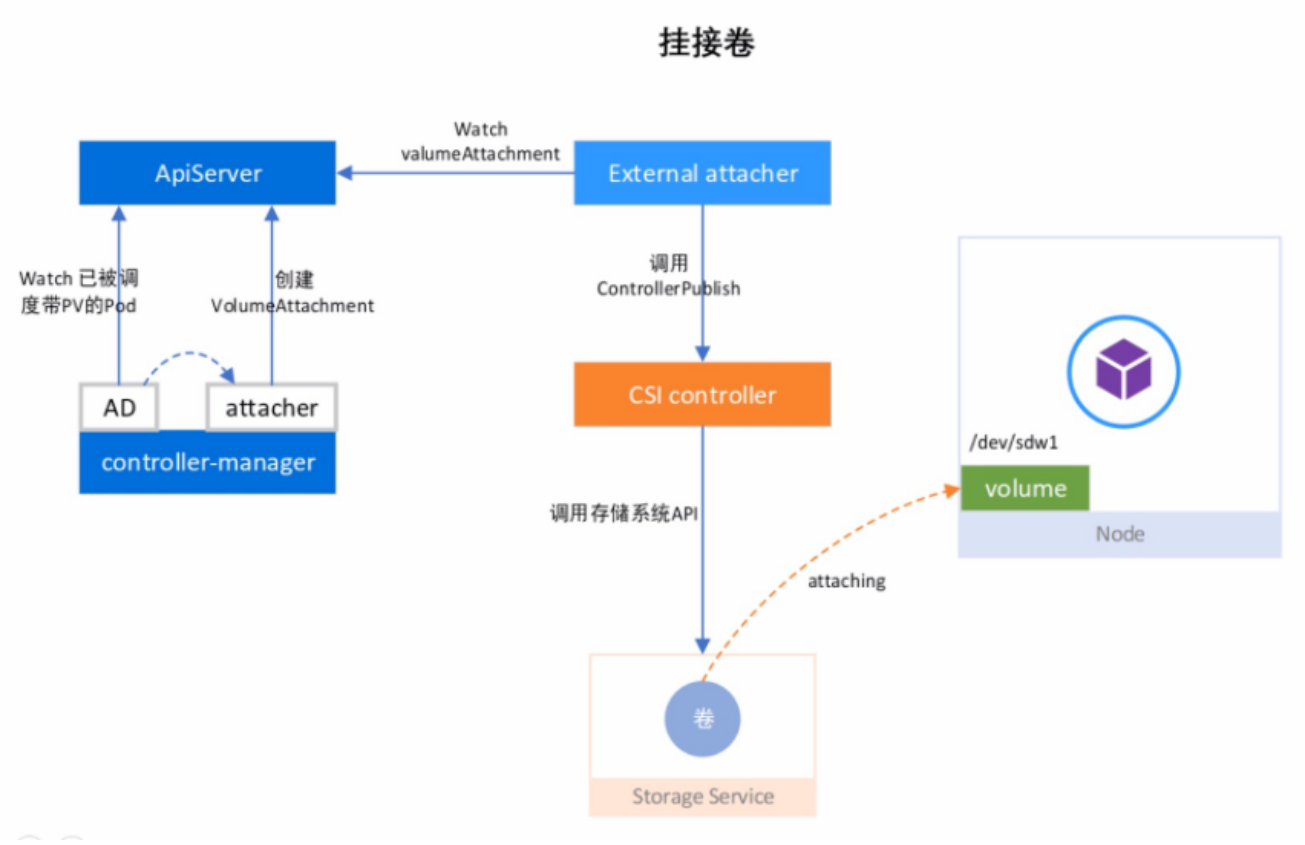
Attach（挂接/摘除）



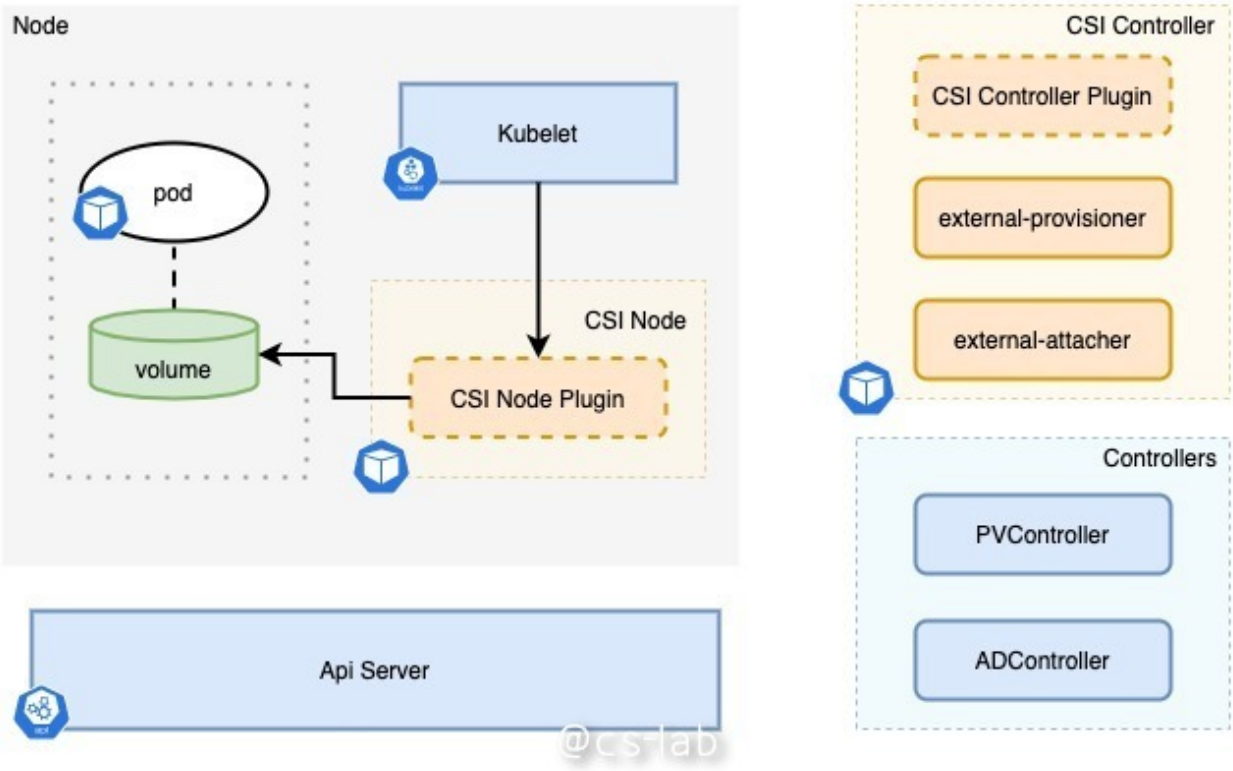
Attach 阶段是指将 volume 挂接到节点上（将存储 **attach**到 **pod** 将会运行的 **node** 上面），整个过程如上图所示。

1. ADController （AttachDetachController）监听到 **pod** 被调度到某节点，并且使用的是 **CSI** 类型的 **PV**，会调用内部的 in-tree CSI 插件的接口，该接口会在集群中创建一个 VolumeAttachment 资源；
2. external-attacher 组件 watch 到有 VolumeAttachment 资源创建出来时，会调用 CSI Controller 的 `ControllerPublishVolume` 接口；
3. 当 CSI Controller 的 `ControllerPublishVolume` 接口调用成功后，external-attacher 将对应的 VolumeAttachment 对象的 Attached 状态设为 true；
4. ADController watch 到 VolumeAttachment 对象的 Attached 状态为 true 时，更新 ADController 内部的状态 ActualStateOfWorld。


挂接卷
AD控制器
api Server
External attacher
CSI plugins
1.观察到Pod使用CSI类型 PV
2.创建volumeAttachment资源对象
3.发现volumeAttachment资源创建
4.调用CSI插件，将存储卷挂接在node上
5.更新 volumeAttachment资源状态
6.更新Node资源状态

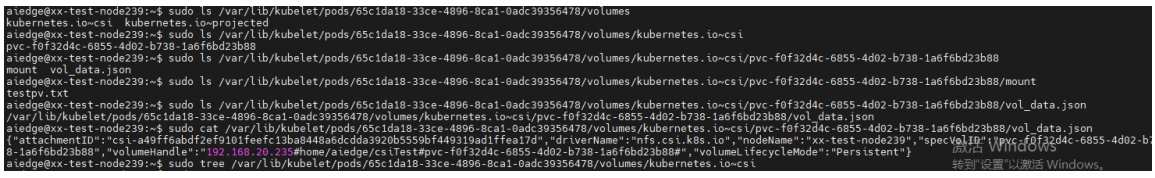

挂接卷
ApiServer
Watch
valumeAttachment
External attacher
Watch 已被调度带PV的Pod
创建
VolumeAttachment
调用
ControllerPublish
AD
attacher
controller-manager
CSI controller
/dev/sdw1
volume
Node
调用存储系统API
attaching
卷
Storage Service

Mount（挂载/卸载）

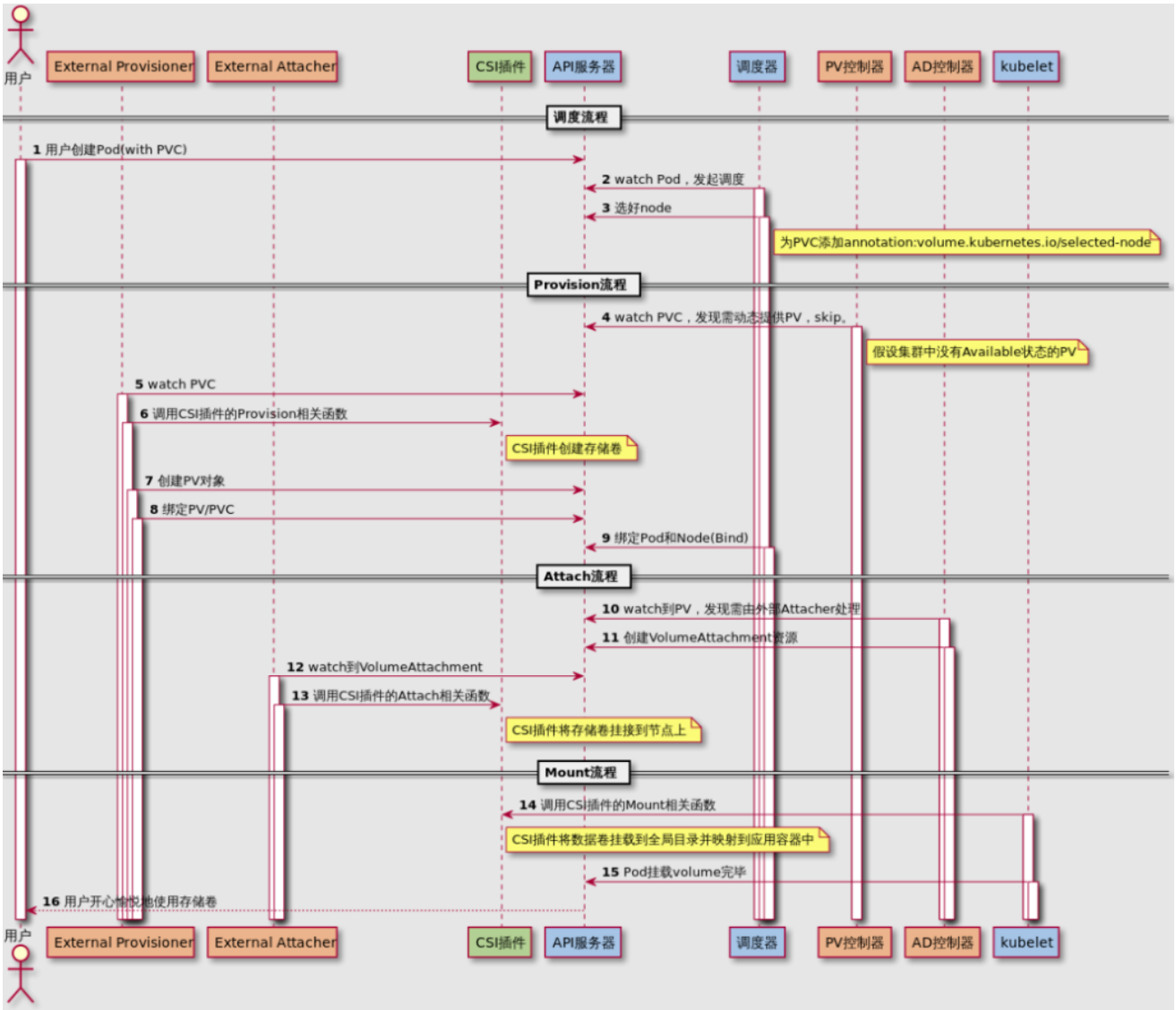


将 volume 挂载到 pod 里的过程涉及到 kubelet。整个流程简单地说是，对应节点上的 **kubelet** 在创建 **pod** 的过程中，会调用 **CSI Node** 插件，执行 **mount** 操作。

kubelet中的volume manager调用csi plugin的NodeStageVolume、NodePublishVolume完成对应的mount操作，kubelet不需要使用grpc交互，直接调用本地二进制文件就可以。

> 1. 在*worker*节点上，*kubelet*（***Volume Manager***）等待设备挂载完成，通过*Volume Plugin*将设备挂载到指定目录：
>
>   *aiedge@xx-test-node239:~$ sudo ls /var/lib/kubelet/pods/65c1da18-33ce-4896-8ca1-0adc39356478/volumes*
>   *kubernetes.io~csi kubernetes.io~projected*
>
> 
>
>
> 2. *kubelet*在被告知挂载目录准备好后，启动*Pod*中的*containers*，用*Docker -v*方式（*bind*）将已经挂载到本地的卷映射到容器中；

总体流程



调度流程

- 创建pod，使用pvc，然后就是正常的调度，选择好node后，对应的pvc添加annotation:volume.kubernetes.io/selected-node

provision流程

attach流程

mount流程

csi-driver-host-path部署

> https://github.com/kubernetes-csi/csi-driver-host-path/blob/master/docs/deploy-1.17-and-later.md
>
> crdyaml：https://github.com/kubernetes-csi/external-snapshotter/blob/v2.0.1/config/crd/snapshot.storage.k8s.io_volumesnapshotclasses.yaml

csi-driver-nfs

https://github.com/kubernetes-csi/csi-driver-nfs/blob/master/docs/install-csi-driver-master.md

http://inksnw.asuscomm.com:3001/post/csi%E6%8C%82%E8%BD%BDnfs/#%E5%88%9B%E5%BB%BAsc%E4%B8%8Epvc

https://blog.csdn.net/fly910905/article/details/120974621

> *kubernetes*中使用*nfs*作为存储卷的几种方式
>
> 1. 在`deployment/statefulset`中直接使用*nfs* 的*volume*
> 2. 创建类型为*nfs*的持久化存储卷，用于为*PersistentVolumClaim*提供存储卷
> 3. 使用*csi-driver-nfs*提供*StorageClass*，使用这个 *StorageClass* 来创建 *PersistentVolumeClaim*（*PVC*）

步骤

第一步：在集群内安装NFS

要在机器192.168.20.235上配置 NFS 服务器端并共享目录`/home/aiedge/csiTest`，需要执行以下步骤：

1. 安装 **NFS** 服务器软件：
   使用以下命令安装 NFS 服务器：

   ```
   sudo apt-get update
   sudo apt-get install nfs-kernel-server
   ```

2. 配置 **NFS** 服务器：
   打开 NFS 服务器的配置文件，通常为 `/etc/exports`。使用文本编辑器（比如 `vi` 或 `nano`）打开配置文件：

   ```
   sudo nano /etc/exports
   ```

   在文件的末尾添加以下行，以共享`/home/aiedge/csiTest`目录：

   ```
   /home/aiedge/csiTest *(rw,sync,no_root_squash)
   ```

   这个配置允许所有主机（`*`）以读写（`rw`）的方式访问共享目录。

   > 内容通常为：

```
directory machine1(option11,option12)
```

- ***directory***

  要共享的目录。如 */home/aiedge/csiTest*

- ***machine1***

  *nfs*客户端，就是哪些机器可以访问他，可以是指定*ip*，也可以是一个*ip*段。

  ```
  /data 172.18.11.0/24(rw)
  ```

  ```
  /data 172.18.11.1(rw) 172.18.11.2(rw)
  ```

- ***option*** 重要的权限有如下几种：

  - *ro*：只读，不可写
  - *rw*：有读写权限
  - *no_root_squash/root_squash*：默认情况下(*all_squash*)，客户端上的*root*用户发出的任何请求都会变成服务端的***nobody***用户权限去执行。如果开启了此项`no_root_squash`，客户端上的***root***用户发出的请求等同服务端的***root***用户权限，会有安全隐患，不建议使用*no_root_squash*

    (*"nfsnobody"* 是一个特殊的用户和组，通常用于映射在*NFS*服务器上没有对应用户或组的客户端请求。)
  - *async/sync*： 默认情况下，所有*exportfs*命令都将使用异步，即使用*sync*选项文件先保存在内存中，达到触发条件再发往服务端，性能较好，但存在风险。若使用同步*async*，则实时存到服务端。

  基于安全的最佳配置：

  https://blog.csdn.net/yangshihuz/article/details/104783585

  ```
  # 创建文件共享目录和文件夹权限
  mkdir /nfs_share
  chown nobody:nogroup /nfs_share/
  chmod 750 /nfs_share/

  #配置文件
  vi /etc/exports

  /nfs_share 192.168.20.236(rw,all_squash,sync)
  #客户端所有用户在访问服务端都会以nobody用户访问，因此可以读写

  #配置文件生效
  exportfs -rav
  ```

```
#在192.168.20.236端mount
sudo mount -t nfs
#查看
192.168.20.235:/nfs_share /home/aiedge/mnt
aiedge@xx-test-node236:~$ ll /home/aiedge/mnt
total 8
drwxr-x---  2 nobody nogroup 4096 Dec 20 15:01 ./
drwxr-xr-x 12 aiedge aiedge  4096 Dec 19 14:48 ../
```

3. 重启 **NFS** 服务：
   完成配置后，重启 NFS 服务以使更改生效：

```
sudo systemctl restart nfs-kernel-server    # 对于基于 systemd 的系统
```

4. 验证共享配置：
   使用以下命令验证 NFS 服务器是否正在运行并已正确配置：

```
sudo systemctl status nfs-kernel-server    # 检查服务状态
showmount -e 192.168.20.235                # 显示可用的 NFS 共享
```

   如果一切设置正确，应该能够看到输出，表明/home/aiedge/csiTest目录已经共享出去了。

现在，你已经在192.168.20.235机器上配置好了 NFS 服务器端，并共享了/home/aiedge/csiTest目录。其他主机可以使用 NFS 客户端挂载这个共享目录。

```
root@xx-test-node239:/home/aiedge# sudo mkdir -p /mnt/nfs
root@xx-test-node239:/home/aiedge# sudo mount -t nfs
192.168.20.235:/home/aiedge/csiTest /mnt/nfs
```

第二步：配置CSI

安装nfs-csi-driver

这个插件驱动本身只提供了集群中的资源和NFS服务器之间的通信层，使用这个驱动之前需要 Kubernetes 集群 1.14 或更高版本和预先存在的 NFS 服务器。

```
git clone https://github.com/kubernetes-csi/csi-driver-nfs.git
cd csi-driver-nfs
./deploy/install-driver.sh v4.1.0 local #表示用本地yaml部署
```

```
root@xx-test-master235:/home/aiedge/csi-driver-nfs# ./deploy/install-driver.sh v4.1.0 local
use local deploy
Installing NFS CSI driver, version: v4.1.0 ...
serviceaccount/csi-nfs-controller-sa created
serviceaccount/csi-nfs-node-sa created
clusterrole.rbac.authorization.k8s.io/nfs-external-provisioner-role created
clusterrolebinding.rbac.authorization.k8s.io/nfs-csi-provisioner-binding created
csidriver.storage.k8s.io/nfs.csi.k8s.io created
deployment.apps/csi-nfs-controller created
daemonset.apps/csi-nfs-node created
NFS CSI driver installed successfully.
```

实际上，最主要的几个部署yaml：

```
rbac-csi-nfs.yaml
csi-nfs-driverinfo.yaml
csi-nfs-controller.yaml
csi-nfs-node.yaml
```

发现，新增：

```
kube-system    csi-nfs-controller-78d56d9785-nnbcc          0/3
ContainerCreating   0               21s     192.168.20.239   xx-test-
node239     <none>           <none>
kube-system    csi-nfs-node-249fj                           0/3
ContainerCreating   0               20s     192.168.20.236   xx-test-
node236     <none>           <none>
kube-system    csi-nfs-node-lc6qj                           0/3
ContainerCreating   0               20s     192.168.20.235   xx-test-
master235   <none>           <none>
kube-system    csi-nfs-node-rcwqs                           0/3
ContainerCreating   0               20s     192.168.20.239   xx-test-
node239     <none>           <none>
```

```
root@xx-test-master235:/home/aiedge# kubectl get po -n kube-system -Aowide | grep csi
kube-system    csi-nfs-controller-78d56d9785-xxdjt        3/3     Running   1 (33m ago)    48m    192.168.20.236   xx-test-node236
kube-system    csi-nfs-node-249fj                         3/3     Running   0              97m    192.168.20.236   xx-test-node236
kube-system    csi-nfs-node-lc6qj                         3/3     Running   0              97m    192.168.20.235   xx-test-master235
kube-system    csi-nfs-node-rcwqs                         3/3     Running   0              97m    192.168.20.239   xx-test-node239
```

查看csinode信息：

```
root@xx-test-master235:/home/aiedge# kubectl get csinode xx-test-node236 -o yaml
apiVersion: storage.k8s.io/v1
kind: CSINode
metadata:
  annotations:
    storage.alpha.kubernetes.io/migrated-plugins: kubernetes.io/aws-ebs,kubernetes.io/azure-disk,kubernetes.io/cinder,kubernetes
/gce-pd
  creationTimestamp: "2023-08-21T09:40:57Z"
  name: xx-test-node236
  ownerReferences:
  - apiVersion: v1
    kind: Node
    name: xx-test-node236
    uid: 81bc11a0-6f3a-4e91-887c-44d5ee6a9f57
  resourceVersion: "5751806"
  uid: d778db91-c952-40b4-a4e2-fd933f665bba
spec:
  drivers:
  - name: nfs.csi.k8s.io
    nodeID: xx-test-node236
    topologyKeys: null
```


> *csi驱动的名称：nfs.csi.k8s.io*

第三步：创建sc与pvc

关于nfs-driver的参数：https://github.com/kubernetes-csi/csi-driver-nfs/blob/master/docs/driver-parameters.md

动态

使用这个 StorageClass 来创建 PersistentVolumeClaim（PVC）

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: 192.168.20.235
  share: /home/aiedge/csiTest
reclaimPolicy: Delete
volumeBindingMode: Immediate
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: nfs-csi
```

```
root@xx-test-master235:/home/aiedge/csi-driver-nfs# kubectl get sc
NAME      PROVISIONER      RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
nfs-csi   nfs.csi.k8s.io   Delete          Immediate           false                  2m35s
root@xx-test-master235:/home/aiedge/csi-driver-nfs# kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nfs-pvc   Bound    pvc-628f3e62-0529-404e-9cff-077d59eb2b79   10Gi       RWX            nfs-csi        75s
```


======

静态

如果不使用SC，使用静态方式制备

```yaml
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-static
spec:
  capacity:
```

```
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  csi:
    driver: nfs.csi.k8s.io
    readOnly: false
    volumeHandle: unique-volumeid  # #确保它是集群中的唯一 ID
    volumeAttributes:
      server: 192.168.20.235
      share: /home/aiedge/csiTest/
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc-static
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: "" # 此处须显式设置空字符串，否则会被设置为默认的
StorageClass
  volumeName: pv-nfs-static
# 控制平面可以在集群中将 PersistentVolumeClaims 绑定到匹配的
PersistentVolumes。 但是，如果你希望 PVC 绑定到特定 PV，则需要预先绑定它们。
```

> 通过在 *PersistentVolumeClaim* 中指定 *PersistentVolume*，你可以声明该特定 *PV* 与 *PVC* 之间的绑定关系。如果该 *PersistentVolume* 存在且未被通过其 *claimRef* 字段预留给 *PersistentVolumeClaim*，则该 *PersistentVolume* 会和该 *PersistentVolumeClaim* 绑定到一起。
>
> 绑定操作不会考虑某些卷匹配条件是否满足，包括节点亲和性等等。 控制面仍然会检查存储类、 访问模式和所请求的存储尺寸都是合法的。

第四步：可以开始使用PVC

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mynginx
  labels:
    app: mynginx
spec:
  replicas: 1
  template:
```

```
    metadata:
      name: mynginx
      labels:
        app: mynginx
    spec:
      containers:
        - name: mynginx
          image: nginx
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - mountPath: "/data"
              name: data
      restartPolicy: Always
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: nfs-pvc
  selector:
    matchLabels:
      app: mynginx
```

进入Pod查看

```
kubectl exec -it mynginx-7968d6dcd5-gsjwk -- /bin/bash
#查看挂载情况
df -h
```

```
root@mynginx-7968d6dcd5-gsjwk:/# df -h
Filesystem                                                                  Size  Used Avail Use% Mounted on
overlay                                                                      20G  5.6G   14G  29% /
tmpfs                                                                        64M     0   64M   0% /dev
tmpfs                                                                       3.0G     0  3.0G   0% /sys/fs/cgroup
192.168.20.235:/home/aiedge/csiTest/pvc-628f3e62-0529-404e-9cff-077d59eb2b79   20G   11G  9.2G  53% /data
/dev/sda1                                                                    20G  5.6G   14G  29% /etc/hosts
shm                                                                          64M     0   64M   0% /dev/shm
```

在NFS 服务器上/home/aiedge/csiTest/pvc-628f3e62-0529-404e-9cff-077d59eb2b79/新增文件test1.txt

在pod中查看：

```
root@mynginx-7968d6dcd5-gsjwk:/# cd /data
root@mynginx-7968d6dcd5-gsjwk:/data# cat test1.txt
hello
12
```

删除：

clean up NFS CSI driver

```
./deploy/uninstall-driver.sh v4.1.0  local
```

分析：

nfs-controller pod中有以下容器 <csi-nfs-controller.yaml>

1. **csi-provisioner** 容器：
   - 作用：提供 CSI (Container Storage Interface) 功能，负责创建和删除持久化卷。
   - 配置：设置 CSI 地址、领导选举、超时等参数。
2. **csi-snapshotter** 容器（边车（**Sidecar**）辅助容器）：
   - 作用：支持 CSI 快照功能，监视 `VolumeSnapshotContent` 对象，用于创建、删除和恢复卷的快照。
   - 配置：设置 CSI 地址、领导选举、超时等参数。
3. **liveness-probe** 容器：
   - 作用：提供健康检查功能，确保其他容器正常运行。
   - 配置：配置 CSI 地址、探测超时、健康检查端口等参数。
4. **nfs** 容器：
   - 作用：提供 NFS 存储功能，允许容器通过 NFS 协议挂载存储。
   - 配置：设置 NFS 插件参数，包括日志级别、节点 ID、终端点等。
   - **Security Context:** 具有特权，允许执行 SYS_ADMIN 操作，并允许提升权限。
   - **Ports:** 暴露健康检查端口 29652。
   - **Liveness Probe:** 配置 HTTP 健康检查。

nfs-node pod中有以下容器：

1. **liveness-probe** 容器：
   - 作用：提供健康检查功能，确保其他容器正常运行。
   - 配置：配置 CSI 地址、探测超时、健康检查端口等参数。
2. **node-driver-registrar** 容器：
   - 作用：注册 CSI 驱动到 Kubelet，确保 Kubelet 能够识别和调用 CSI 驱动。
   - 配置：配置 CSI 地址、注册路径、Kubelet 节点名称等参数。
3. **nfs** 容器：
   - 作用：提供 NFS 存储功能，允许容器通过 NFS 协议挂载存储。
   - 配置：设置 NFS 插件参数，包括日志级别、节点 ID、终端点等。
   - **Security Context:** 具有特权，允许执行 SYS_ADMIN 操作，并允许提升权限。
   - **Ports:** 暴露健康检查端口 29653。
   - **Liveness Probe:** 配置 HTTP 健康检查。

其他方式

直接挂载 NFS 卷

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
labels:
```

```
  app: nginx
  spec:
  replicas: 3
  selector:
  matchLabels:
  app: nginx
  template:
  metadata:
  labels:
    app: nginx
  spec:
  containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
          - containerPort: 80
          volumeMounts:
          - name: data
            mountPath: /data
        volumes:
        - name: data
          nfs:
            path: /home/aiedge/csiTest
            server: 192.168.20.235
```

使用PV PVC（静态 动态）

使用 PV 和 PVC 的方式可以更好地管理存储资源。将存储资源与应用程序的部署解耦。应用程序的部署配置不需要关心底层存储的具体细节。这种解耦性使得应用程序更易于迁移和维护，因为存储配置可以独立管理。

示例：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    path: /home/aiedge/csiTest
    server: 192.168.20.235
```

创建一个pvc使用这块pv后，pv的状态会变更为Bound

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-nfs
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
```

使用它

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: busybox
  labels:
    app: busybox
spec:
  replicas: 1
  selector:
    matchLabels:
      app: busybox
  template:
    metadata:
      labels:
        app: busybox
    spec:
      containers:
      - name: busybox
        image: busybox
        command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
        volumeMounts:
        - name: data
          mountPath: /data
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: pvc-nfs
```

nfs作为存储卷的已知问题：

- 不保证配置的存储。您可以分配超过 NFS 共享的总大小。共享也可能没有足够的存储空间来实际容纳请求。
- 未实施预配的存储限制。无论供应的大小如何，应用程序都可以扩展以使用所有可用存储。
- 目前不支持任何形式的存储调整大小/扩展操作。您最终将处于错误状态：`Ignoring the PVC: didn't find a plugin capable of expanding the volume; waiting for an external controller to process this PVC.`

```
# 将数据放在特定的配置文件，挂载到容器中
...
containers:
  - env:
    # ... 环境变量配置 ...
    image: registry.aiedge.ndsl-lab.cn/aiedge/aiedge-auth-go:latest
    imagePullPolicy: Always
    name: aiedge-auth
    # ... 其他容器配置 ...
    volumeMounts:
    - mountPath: /app/conf/config.json
      name: aiedge-auth-config-volume
      subPath: config.json
    - mountPath: /app/conf/user.json
      name: aiedge-auth-config-volume
      subPath: user.json
    - mountPath: /app/conf/jwtRS256.key.pub
      name: jwt-pubkey-volume
      subPath: jwtRS256.key.pub
    - mountPath: /app/conf/jwtRS256.key
      name: jwt-prvkey-volume
      subPath: jwtRS256.key

volumes:
  - configMap:
      defaultMode: 420
      name: aiedge-auth-configmap
    name: aiedge-auth-config-volume
  - configMap:
      defaultMode: 420
      name: jwt-pubkey-configmap
    name: jwt-pubkey-volume
  - name: jwt-prvkey-volume
    secret:
      defaultMode: 420
```

```
      secretName: jwt-prvkey-secret
```

参考

http://www.lishuai.fun/2021/08/12/k8s-nfs-pv/#/pv-x2F-pvc-%E4%BD%BF%E7%94%A8%EF%BC%88%E9%9D%99%E6%80%81%E9%85%8D%E7%BD%AE%EF%BC%89

https://mdnice.com/writing/9d2f403d7f4f4c6486198011d4b9a801