存储

介绍

介绍

核心概念

核心概念

Persistent Volume (PV)

Persistent Volume Claim (PVC)

Generic Ephemeral Volumes

emptyDir

hostPath

ConfigMap

Secret

StorageClass

Container Storage Interface (CSI)

Persistent Volume

动态 Persistent Volume 与静态 Persistent

Persistent Volume 的生命周期

访问模式和

Kubernetes 中

Kubernetes 中

存储特性：快照

结论

操作指南

创建 CephFS 文件存储类型存储类

部署卷插件

创建存储类

创建 CephRBD 块存储类

部署卷插件

创建存储类

创建 TopoLV

背景信息

部署卷插件

创建存储类

部署卷快照组件

后续步骤

创建 PV

前提条件

创建 NFS 共享存储类

创建 PVC

前提条件

示例 PersistentVolumeClaim：

通过 Web 控制台创建 Persistent Volume C

通过 CLI 创建 Persistent Volume Claim

操作

通过 Web 控制台扩展 PersistentVolumeCl

通过 CLI 扩展 Persistent Volume Claim 存

其他资源

使用卷快照

前提条件

示例 VolumeSnapshot 自定义资源（CR）

使用 Web 控制台创建卷快照

使用 CLI 创建卷快照

从卷快照创建持久卷声明

附加资源

克隆 PVC

前提条件

ACP 存储支持

创建克隆后的 P

验证克隆

实用指南

通用临时卷

使用 emptyDir

从旧版插件迁

故障排除

节点恢复后移除污点

对象存储

介绍

限制

核心概念

概述

核心资源

安装

前提条件

安装 COSI

操作指南

实用指南

介绍

Kubernetes 提供了一种灵活且可扩展的存储机制，用于管理容器化环境中的数据持久化。通过抽象存储资源，如 Volumes、PersistentVolumes 和 PersistentVolumeClaims，Kubernetes 实现了应用与底层存储系统的解耦，支持动态配置、自动挂载以及跨节点的数据持久化。

其主要特性包括支持多种后端存储系统（例如本地磁盘、NFS、云存储服务）、动态配置、访问模式控制（如读写权限）以及生命周期管理，满足有状态应用的存储需求。对于需要高可用性、数据持久化和多租户隔离的企业级工作负载，Kubernetes 存储是不可或缺的基础能力。

Kubernetes 存储面向开发人员、运维工程师和平台团队，帮助他们高效且安全地管理容器化工作负载中的数据。

核心概念

核心概念

Kubernetes 存储围绕三个关键概念展开：**PersistentVolume (PV)**、**PersistentVolumeClaim (PVC)** 和 **StorageClass**。它们定义了在集群中如何请求、分配和配置存储。**CSI**（Container Storage Interface）驱动程序通常负责存储的实际供给和挂载。

目录

Persistent Volume (PV)

PersistentVolume (PV) 是集群中一块已经被供给的存储（可以由管理员静态供给，也可以通过 StorageClass 动态供给）。它表示底层存储——例如云提供商上的磁盘或网络附加文件系统——并被视为集群中的一种资源，类似于节点。

Persistent Volume Claim (PVC)

PersistentVolumeClaim (PVC) 是对存储的请求。用户定义所需存储的容量以及访问模式（例如读写）。如果存在合适的 PV，或者可以通过 StorageClass 动态供给，则 PVC 会与该 PV “绑定”。一旦绑定，Pod 就可以引用该 PVC 来持久化或共享数据。

Generic Ephemeral Volumes

Kubernetes 的 Generic Ephemeral Volumes 是 Kubernetes 中引入的一项特性，允许你在 Pod 生命周期内使用由 CSI 驱动的 `temporary` 卷，类似于 `emptyDir`，但功能更强大，并且允许你挂载任何类型的 CSI 卷（支持快照、扩缩容等）。

有关使用详情，请参见 Generic ephemeral volumes。

emptyDir

1. emptyDir 是一种空目录类型的临时存储卷。
2. 当 Pod 被调度到某个节点时会创建该卷，存储位于该节点的本地文件系统上（默认情况下是节点磁盘）。
3. 当 Pod 被删除时，emptyDir 中的数据也会被清除。

有关使用详情，请参见 Using an emptyDir。

hostPath

在 Kubernetes 中，hostPath 卷是一种特殊的卷类型，它会将主机节点文件系统中的文件或目录直接映射到 Pod 的容器中。

- 它允许 Pod 访问主机节点上的文件或目录。
- 适用于：
  - 访问主机级资源
  - 调试

- 使用节点上已存在的数据

ConfigMap

Kubernetes 中的 ConfigMap 是一种 API 对象，用于以键值对的形式存储非敏感配置信息。它可以将配置与应用代码解耦，从而使你的应用更具可移植性，也更易于管理。

Secret

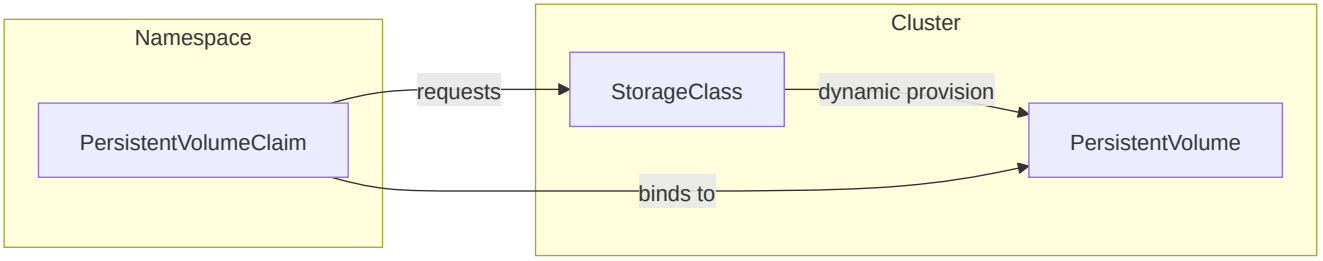
在 Kubernetes 中，Secret 是一种用于存储敏感数据的 API 对象，例如：

- 密码
- OAuth 令牌
- SSH 密钥
- TLS 证书
- 数据库凭据

Secret 通过避免将这些数据直接存储在 Pod 规范或容器镜像中，来帮助保护这些数据。

StorageClass

StorageClass 描述了卷应如何动态供给。它映射到特定的 provisioner（通常是 CSI 驱动程序），并且可以包含存储层级、性能特征或其他后端配置等参数。通过创建多个 StorageClass，你可以向开发人员提供不同类型的存储。



图示：PVC、PV 和 StorageClass 之间的关系。

Container Storage Interface (CSI)

Container Storage Interface (CSI) 是 Kubernetes 用于与存储驱动程序集成的标准 API。它允许第三方存储提供商构建 out-of-tree 插件，这意味着你可以在不修改 Kubernetes 本身的情况下安装或更新存储驱动程序。

一个 CSI **driver** 通常包含两个组件：

1. **Controller** 组件：在集群中运行（通常作为 Deployment），负责高层操作，例如创建或删除卷。对于网络存储，它还可能负责将卷挂载到节点和从节点卸载卷。
2. **Node** 组件：在每个节点上运行（通常作为 DaemonSet），负责在该特定节点上挂载和卸载卷。它会与 kubelet 通信，以确保 Pod 能够访问该卷。

当用户创建一个引用了使用 CSI 驱动程序的 StorageClass 的 PVC 时，CSI 驱动程序会观察到该请求，并相应地供给存储（如果需要动态供给）。当存储创建完成后，驱动程序会通知 Kubernetes，Kubernetes 随即创建对应的 PV 并将其绑定到 PVC。只要 Pod 使用该 PVC，驱动程序的节点组件就会负责卷挂载，从而使存储在容器内可用。

通过利用 **PV**、**PVC**、**StorageClass** 和 **CSI**，Kubernetes 实现了一种强大、声明式的存储管理方式。管理员可以定义一个或多个 StorageClass 来表示不同的存储后端或性能层级，而开发人员只需使用 PVC 请求存储——无需关心底层基础设施。

Persistent Volume

PersistentVolume（PV）表示 Kubernetes 集群中与后端存储卷的映射关系，作为 Kubernetes API 资源存在。它是由管理员统一创建和配置的集群资源，负责抽象实际的存储资源，构成集群的存储基础设施。

PersistentVolume 具有独立于 Pod 的生命周期，能够实现 Pod 数据的持久化存储。

管理员可以手动创建静态 PersistentVolume，或基于存储类动态生成 PersistentVolume。开发人员若需要为应用获取存储资源，可通过 PersistentVolumeClaim（PVC）进行申请，PVC 会匹配并绑定合适的 PersistentVolume。

目录

动态 Persistent Volume 与静态 Persistent Volume

平台支持管理员管理两类 PersistentVolume，分别是动态 Persistent Volume 和静态 Persistent Volume。

- 动态 **Persistent Volume**：基于存储类实现。存储类由管理员创建，定义了一种描述存储资源类别的 Kubernetes 资源。当开发人员创建关联存储类的 PersistentVolumeClaim 后，平

台会根据 PersistentVolumeClaim 和存储类中配置的参数动态创建合适的 PersistentVolume，并绑定到该 PersistentVolumeClaim，实现存储资源的动态分配。

- 静态 **Persistent Volume**：由管理员手动创建的 Persistent Volume。目前支持创建 **HostPath** 或 **NFS** 共享存储 类型的静态 Persistent Volume。当开发人员创建不使用存储类的 PersistentVolumeClaim 时，平台会根据 PersistentVolumeClaim 中配置的参数匹配并绑定合适的静态 PersistentVolume。
  - **HostPath**：使用节点主机上的文件目录（不支持本地存储）作为后端存储，例如：`/etc/kubernetes`。通常仅适用于单计算节点集群的测试场景。
  - **NFS** 共享存储：指网络文件系统，是 Persistent Volume 常见的后端存储类型。用户和程序可以像访问本地文件一样访问远程系统上的文件。

Persistent Volume 的生命周期

1. **Provisioning**（配置）：管理员手动创建静态 Persistent Volume。创建后，Persistent Volume 进入 **Available** 状态；或者平台根据关联存储类的 PersistentVolumeClaim 动态创建合适的 Persistent Volume。
2. **Binding**（绑定）：静态 Persistent Volume 匹配并绑定到 PersistentVolumeClaim 后，进入 **Bound** 状态；动态 Persistent Volume 根据请求动态创建，创建成功后也进入 **Bound** 状态。
3. **Using**（使用）：开发人员将 PersistentVolumeClaim 关联到计算组件的容器实例，使用 Persistent Volume 映射的后端存储资源。
4. **Releasing**（释放）：开发人员删除 PersistentVolumeClaim 后，Persistent Volume 被释放。
5. **Reclaiming**（回收）：Persistent Volume 释放后，根据 Persistent Volume 或存储类的回收策略参数对其进行回收操作。

访问模式和卷模式

在 Kubernetes 中，PersistentVolumeClaims（PVC）和 StorageClasses 协同工作，管理存储的供应和工作负载对存储的访问方式。两个关键概念是 访问模式 和 卷模式。本文探讨这些概念，并重点介绍不同存储系统对它们的支持情况。

目录

Kubernetes 中的访问模式

访问模式定义了卷如何被挂载和被 pod 使用。主要的访问模式包括：

- **ReadWriteOnce (RWO)**：卷可以被单个节点以读写方式挂载。
- **ReadOnlyMany (ROX)**：卷可以被多个节点以只读方式挂载。
- **ReadWriteMany (RWX)**：卷可以被多个节点以读写方式挂载。

按存储类划分的访问模式

存储类	支持 **RWO**	支持 **ROX**	支持 **RWX**
CephFS File Storage	是	否	是
CephRBD Block Storage	是	否	否
TopoLVM	是	否	否
NFS Shared Storage	是	否	是

如上所示，基于文件的存储系统如 **CephFS** 和 **NFS** 支持多节点的并发写入或读取操作，适合共享访问场景。另一方面，基于块的存储系统如 **CephRBD** 和 **TopoLVM** 则提供对单个节点的独占访问。

Kubernetes 中的卷模式

卷模式定义了数据如何暴露给 pod：

- **Filesystem**：卷以文件系统的形式挂载到 pod 中。
- **Block**：卷以原始块设备的形式呈现。

按存储类划分的卷模式

存储类	类型	支持的卷模式
CephFS File Storage	文件存储	Filesystem
CephRBD Block Storage	块存储	Filesystem, Block
TopoLVM	块存储	Filesystem, Block
NFS Shared Storage	文件存储	Filesystem

基于块的存储系统如 **CephRBD** 和 **TopoLVM** 同时支持文件系统和原始块访问，满足不同应用需求的灵活性。相比之下，文件存储系统如 **CephFS** 和 **NFS** 仅支持文件系统模式。

存储特性：快照和扩容

Kubernetes 还支持高级功能，如卷快照和 PVC 的动态扩容，具体取决于所使用的存储类。

存储类	卷快照	扩容
CephFS File Storage	支持	支持
CephRBD Block Storage	支持	支持
TopoLVM	支持	支持
NFS Shared Storage	不支持	不支持

只有使用 StorageClass 动态供应的 PVC 才支持卷快照。此功能对于备份和克隆环境非常有用。

结论

在 Kubernetes 中配置存储时，理解 PVC 的 访问模式 和 卷模式 以及其背后的 **StorageClasses** 对于为工作负载选择合适的解决方案至关重要。文件存储解决方案如 CephFS 和 NFS 适合共享访问场景，而块存储如 CephRBD 和 TopoLVM 则在高性能单节点部署中表现出色。此外，快照和扩容等功能的支持能够极大地增强存储的灵活性和数据管理策略。

操作指南

创建 CephFS 文件存储类型存储类

CephFS 文件存储是 Ceph 内置的文件存储系统，它为平台提供了基于 Container Storage Interface (CSI) 的存储访问方式，提供安全、可靠、可扩展的共享文件存储服务，适用于文件共享和数据备份等场景。在继续之前，您必须先创建 CephFS 文件存储类。

在 Persistent Volume Claim (PVC) 中绑定存储类后，平台会根据该持久卷声明自动在节点上为业务应用创建持久卷。

目录

部署卷插件

在 分布式存储 页面点击 部署 后，选择 在内部模式下部署 或 在外部模式下部署。

创建存储类

1. 进入 管理员。
2. 在左侧导航栏中，点击 存储管理 > 存储类。
3. 点击 创建存储类。

注意：以下内容以表单形式提供示例；您也可以选择使用 YAML 创建。

4. 选择 **CephFS** 文件存储，然后点击 下一步。
5. 根据以下说明配置相关参数。

参数	描述
回收策略	持久卷的回收策略。 - Delete：删除持久卷声明时，绑定的持久卷也会被删除。 - Retain：即使删除持久卷声明，绑定的持久卷也会保留。
访问模式	当前存储支持的所有访问模式。后续声明持久卷时，只能选择其中一种模式。 - ReadWriteOnce (RWO)：可由单个节点以读写方式挂载。 - ReadWriteMany (RWX)：可由多个节点以读写方式挂载。
分配项目	分配可使用此类存储的项目。 如果当前没有项目需要使用此类存储，则可先留空，后续再更新。

提示：以下参数需要在分布式存储中设置，并将直接应用到此处。

- 存储集群：当前集群中的内置 Ceph 存储集群。
- 存储池：存储集群中用于数据存储的逻辑分区。

6. 点击 创建。

创建 CephRBD 块存储类

CephRBD 块存储是平台内置的 Ceph 块存储，提供基于 Container Storage Interface (CSI) 的存储访问方式，可提供高 IOPS 和低延迟的存储服务，适用于数据库和虚拟化等场景。使用前，需要先创建 CephRBD 块存储类。

当 Persistent Volume Claim (PVC) 绑定到该存储类后，平台会基于 Persistent Volume Claim 动态创建一个 Persistent Volume 供业务应用使用。

目录

部署卷插件

在 分布式存储 页面点击 部署 后，选择 在内部模式下部署 或 在外部模式下部署。

创建存储类

1. 进入 管理员。
2. 在左侧导航栏中，点击 存储管理 > 存储类。
3. 点击 创建存储类。

说明：以下内容以表单形式为例，您也可以选择 YAML 完成操作。

4. 选择 **CephRBD** 块存储，然后点击 下一步。
5. 按需配置参数。

参数	说明
文件系统	默认值为 **EXT4**，这是 Linux 的一种日志型文件系统，能够提供 extent 文件存储并处理大文件。文件系统容量可达 1 EiB，支持的单个文件大小最高可达 16 TiB。
回收策略	持久卷的回收策略。 - 删除：绑定的持久卷会随着持久卷声明一并删除。 - 保留：即使删除持久卷声明，绑定的持久卷仍会保留。
访问模式	仅支持 ReadWriteOnce (RWO)：只能由单个节点以读写模式挂载。
分配项目	分配可使用此类存储的项目。 如果当前没有项目需要此类存储，可先留空，后续再更新。

提示：以下参数需要在分布式存储中设置，并会直接应用于此处。

- Storage Cluster：当前集群中内置的 Ceph 存储集群。
- Storage Pool：存储集群内用于存储数据的逻辑分区。

6. 点击 创建。

创建 TopoLVM 本地存储类

TopoLVM 是一种基于 LVM 的本地存储解决方案，提供简单、易维护且高性能的本地存储服务，适用于数据库和中间件等场景。使用之前，您需要创建一个 TopoLVM 存储类。

当 Persistent Volume Claim (PVC) 绑定到该存储类后，平台会根据 Persistent Volume Claim 在节点上动态创建持久卷，供业务应用使用。

目录

背景信息

使用优势

- 与远程存储（例如 **NFS** 共享存储）相比：TopoLVM 类型的存储位于节点本地，具有更好的 IOPS 和吞吐性能，同时延迟更低。
- 与 hostPath（例如 **local-path**）相比：虽然两者都是节点本地存储，但 TopoLVM 允许将容器组灵活调度到具有足够可用资源的节点上，避免因资源不足导致容器组无法启动的问题。
- TopoLVM 默认支持自动卷扩容。修改 Persistent Volume Claim 中所需的存储配额后，无需重启容器组即可自动完成扩容。

使用场景

- 仅需要临时存储时，例如开发和调试场景。
- 对存储 I/O 要求较高时，例如实时索引。

约束和限制

仅将本地存储用于在应用层提供数据复制和备份的应用。底层节点发生故障时，本地存储无法提供独立的数据持久性保障。

有关 TopoLVM 的行为，请参阅 TopoLVM 用户手册↗。

部署卷插件

单击部署后，在新打开的页面中配置本地存储。

创建存储类

1. 进入管理员。
2. 在左侧导航栏中，单击存储管理 > 存储类。
3. 单击创建存储类。
4. 选择块存储。
5. 选择 **TopoLVM**，然后单击下一步。
6. 按如下说明配置存储类参数。
   注意：以下内容以表单示例形式展示；您也可以选择使用 YAML 创建。

参数	描述
名称	存储类的名称，在当前集群中必须唯一。
显示名称	有助于您识别或筛选的名称，例如存储类的中文描述。
设备类	设备类是在 TopoLVM 中对存储设备进行分类的一种方式，每个设备类对应一组具有相似特征的存储设备。如果没有特殊要求，请使用 自动分配 设备类。
文件系统	• **XFS** 是一种高性能日志型文件系统，适合处理并行 I/O 负载，支持大文件处理和流畅的数据传输。 • **EXT4** 是 Linux 下的一种日志型文件系统，提供扩展区文件存储并支持大文件处理，文件系统最大容量为 1 EiB，最大文件大小为 16 TiB。
回收策略	持久卷的回收策略。 • Delete：与 PVC 绑定的持久卷在删除 PVC 时也会被删除。 • Retain：即使删除 PVC，绑定的持久卷也会保留。
访问模式	ReadWriteOnce (RWO)：可由单个节点以读写方式挂载。
PVC 重建	支持跨节点 PVC 重建。启用后，必须配置重建等待时间。当使用该存储类创建 PVC 的节点发生故障时，PVC 会在等待时间后自动在其他节点上重建，以确保业务连续性。 注意： • 重建后的 PVC 不包含原始数据。 • 请确保存储节点数量大于应用实例副本数，否则可能影响 PVC 重建。
分配的项目	此类型的 Persistent Volume Claim 只能在特定项目中创建。 如果当前尚未分配项目，也可以稍后更新。

7. 确认配置信息无误后，单击创建按钮。

后续操作

一切准备就绪后，您可以通知开发人员使用 TopoLVM 功能。例如，在容器平台的存储 > **Persistent Volume Claims** 页面中创建一个 Persistent Volume Claim，并将其绑定到 TopoLVM 存储类。

创建 NFS 共享存储类

基于 Community NFS CSI（Container Storage Interface）存储驱动，提供访问多个 NFS 存储系统或账户的能力。

与传统的 NFS 访问客户端-服务器模型不同，NFS 共享存储使用 Community NFS CSI（Container Storage Interface）存储插件，这种方式更符合 Kubernetes 设计原则，并允许客户端访问多个服务器。

目录

前提条件

- 必须已配置 NFS Server，并获取其访问方式。目前，平台支持三种 NFS 协议版本：`v3`、`v4.0` 和 `v4.1`。你可以在服务器端执行 `nfsstat -s` 来检查版本信息。

部署 灵雀云容器平台 NFS CSI

通过 Web 控制台部署

1. 进入 管理员。
2. 在左侧导航栏中，点击 存储 > **StorageClasses**。
3. 点击 创建 **StorageClass**。
4. 在 **NFS CSI** 右侧，点击 部署 以跳转到 **Plugins** 页面。
5. 在 灵雀云容器平台 **NFS CSI** 插件右侧，点击⋮> 安装。
6. 等待部署状态显示为 部署成功 后，再完成部署。

通过 YAML 部署

有关 YAML 安装，请参见 通过 YAML 安装。

灵雀云容器平台 **NFS CSI** 是一个 **Non-config plugin**，模块名称为 `nfs`。

创建 NFS 共享存储类

1. 点击 创建 **Storage Class**。
   注意：以下内容以表单形式展示，但你也可以选择使用 YAML 完成操作。
2. 选择 **NFS CSI**，然后点击 下一步。
3. 配置以下参数。

参数	描述
名称	存储类的名称。在当前集群中必须唯一。
服务地址	NFS Server 的访问地址。例如：`192.168.2.11`。
路径	Server 节点上 NFS 文件系统的挂载路径。例如：`/nfs/data`。
NFS 协议版本	目前支持三个版本：`v3`、`v4.0` 和 `v4.1`。
回收策略	持久卷的回收策略。 - Delete：删除持久卷声明后，已绑定的持久卷也会被删除。

参数	描述
	- Retain：即使删除持久卷声明，已绑定的持久卷仍会保留。
访问模式	当前存储支持的所有访问模式。在后续声明持久卷时，仅可选择其中一种模式来挂载持久卷。 - ReadWriteOnce (RWO)：可由单个节点以读写方式挂载。 - ReadWriteMany (RWX)：可由多个节点以读写方式挂载。 - ReadOnlyMany (ROX)：可由多个节点以只读方式挂载。
分配项目	分配可使用此类存储的项目。 如果当前没有项目需要此类存储，请先留空，之后再更新。
subDir	使用 NFS Shared Storage Class 创建的每个 PersistentVolumeClaim (PVC) 都对应 NFS 共享中的一个子目录。默认情况下，子目录名称使用 `${pv.metadata.name}`（即 PersistentVolume 名称）模式。如果默认生成的名称不符合你的需求，可以自定义子目录命名规则。

NOTE

`subDir` 字段仅支持以下三个变量，NFS CSI Driver 会自动解析这些变量：

- `${pvc.metadata.namespace}`：PVC Namespace。
- `${pvc.metadata.name}`：PVC Name。
- `${pv.metadata.name}`：PV Name。

`subDir` 命名规则 必须 保证子目录名称唯一。否则，多个 PVC 可能会共享同一子目录，从而导致数据冲突。

推荐配置：

- `${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}`
- `<cluster-identifier>_${pvc.metadata.namespace}_${pvc.metadata.name}_${pv.metadata.name}`

此配置适用于多个 Kubernetes 集群共享同一个 NFS Server 的场景，通过在子目录命名规则中加入集群特定标识符（例如集群名称），可清晰地区分不同集群。

不推荐配置：

- `${pvc.metadata.namespace}-${pvc.metadata.name}-${pv.metadata.name}` 避免使用 - 作为分隔符，因为这可能导致子目录名称产生歧义。例如：如果两个 PVC 的名称分别为 `ns-1/test` 和 `ns/1-test`，它们都可能生成相同的子目录 `ns-1-test`。
- `${pvc.metadata.namespace}/${pvc.metadata.name}/${pv.metadata.name}` 请不要将 subDir 配置为创建嵌套目录。删除 PVC 时，NFS CSI Driver 只会删除最后一级目录 `${pv.metadata.name}`，从而在 NFS Server 上留下孤立的父目录。

4. 确认配置信息无误后，点击 创建。

部署卷快照组件

卷快照是持久卷在某一时刻的副本。如果集群使用支持快照功能的持久卷，则在用户可以创建和恢复卷快照之前，必须先提供卷快照组件。

在 ACP 4.4 及更高版本中，对于新集群，您不需要安装 灵雀云容器平台 **Snapshot Management** 集群插件。只要 ACP Storage 可用，ACP 会自动准备集群级别的快照组件。

如果集群已经有外部 Snapshot Controller 或现有的 snapshot CRD，ACP 会保留现有的快照能力，不会自动替换它。

如果集群是使用旧版 灵雀云容器平台 **Snapshot Management** 集群插件安装的，请按照 将卷快照组件从旧版插件迁移 中的说明进行迁移。

目前，平台仅支持为使用存储类动态创建的 PVC 创建卷快照。您可以基于这些快照创建新的 PVC 绑定。

提示：从快照创建 PVC 时支持的访问模式，与使用存储类创建 PVC 时支持的访问模式不同，后者在下表中以粗体标出。

用于创建卷快照的存储类	单节点读写 **(RWO)**	多节点只读 **(ROX)**	多节点读写 **(RWX)**
TopoLVM	支持	不支持	不支持
CephRBD 块存储	支持	不支持	不支持
CephFS 文件存储	支持	支持	支持

目录

后续步骤

在快照组件可用后，用户即可创建卷快照。有关详细信息，请参阅使用卷快照。

如果在移除旧版插件后快照操作无法正常工作，请按照将卷快照组件从旧版插件迁移中的说明进行处理。

创建 PV

手动创建类型为 **HostPath** 或 **NFS Shared Storage** 的静态持久卷。

- **HostPath**：将容器所在主机上的文件目录挂载到容器中的指定路径（对应 Kubernetes 的 HostPath），使容器能够使用主机文件系统进行持久化存储。如果主机变得不可访问，HostPath 也可能无法访问。
- **NFS Shared Storage**：NFS Shared Storage 使用 Community 的 NFS CSI（Container Storage Interface）存储插件，更符合 Kubernetes 的设计原则，并为多个服务提供客户端访问能力。使用前请确保当前集群已部署 **NFS** 存储插件。

目录

前提条件

- 确认要创建的持久卷大小，并确保后端存储系统当前具备提供相应存储的容量。
- 获取后端存储访问地址、要挂载的文件路径、凭证访问方式（如需要）以及其他相关信息。

持久卷示例

```yaml
# example-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 5Gi 1
  accessModes:
    - ReadWriteOnce 2
  persistentVolumeReclaimPolicy: Retain 3
  storageClassName: manual 4
  hostPath: 5
    path: "/mnt/data"
```

1. 存储容量。
2. 卷的挂载方式。
3. 删除 PVC 后的处理方式（Retain、Delete、Recycle）。
4. StorageClass 名称（用于动态绑定）。
5. 存储后端类型。

使用 **Web** 控制台创建 **PV**

1. 进入 管理员。
2. 在左侧导航栏中，点击 存储管理 > 持久卷（**PV**）。
3. 点击 创建持久卷。

4. 配置以下参数，然后点击 创建。

存储信息

类型	参数	描述
HostPath	Path	后端存储卷所在节点上的文件目录路径。例如：`/etc/kubernetes`。
NFS Shared Storage	Server Address	NFS 服务器的访问地址。
	Path	服务器节点上 NFS 文件系统的挂载路径，例如 `/nfs/data`。
	NFS Protocol Version	平台当前支持的 NFS 协议版本为 `v3`、`v4.0` 和 `v4.1`。您可以在服务端执行 `nfsstat -s` 查看版本信息。

使用 **CLI** 创建 **PV**

```
kubectl apply -f example-pv.yaml
```

访问模式

持久卷的访问模式受后端存储设置的相关参数影响。

访问模式	含义
ReadWriteOnce (RWO)	只能由单个节点以读写方式挂载。
ReadWriteMany (RWX)	可由多个节点以读写方式挂载。
ReadOnlyMany (ROX)	可由多个节点以只读方式挂载。

回收策略

回收策略	含义
Delete	在删除持久卷声明的同时，删除已绑定的持久卷以及后端存储卷资源。 注意：NFS Shared Storage 类型 PV 的回收策略不支持 **Delete**。
Retain	即使持久卷声明已删除，已绑定的持久卷和存储数据仍会保留。之后需要手动处理存储数据并删除持久卷。

相关操作

您可以点击列表页右侧的⋮，或者在详情页右上角点击 操作，按需更新或删除持久卷。

删除持久卷适用于以下两种场景：

- 删除未绑定的持久卷：尚未写入且不再需要写入，因此删除后可释放相应存储空间。
- 删除 **Retain** 类型的持久卷：持久卷声明已删除，但由于保留回收策略，持久卷未被同时删除。如果持久卷中的数据已备份到其他存储或不再需要，删除它同样可以释放相应存储空间。

其他资源

- 创建 PVC

创建 PVC

创建一个 PersistentVolumeClaim（PVC），并根据需要设置所请求的 PersistentVolume（PV）参数。

您可以通过可视化 UI 表单或使用自定义 YAML 编排文件来创建 PersistentVolumeClaim。

目录

前提条件

请确保命名空间中剩余的 **storage** 配额足够，以满足本次创建操作所需的存储大小。

示例 **PersistentVolumeClaim**：

```yaml
# example-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
  namespace: k-1
  annotations: {}
  labels: {}
spec:
  storageClassName: cephfs
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 4Gi
```

通过 **Web** 控制台创建 **Persistent Volume Claim**

1. 转到 **Container Platform**。
2. 在左侧边栏中，单击 **Storage > PersistentVolumeClaims (PVC)**。
3. 单击 **Create PVC**。
4. 按需配置参数。
   注意：以下内容以表单方式为例；您也可以切换到 YAML 模式完成操作。

参数	描述
Name	PersistentVolumeClaim 的名称，在当前命名空间内必须唯一。
Creation Method	- Dynamic Creation：根据 storage class 动态生成一个 PersistentVolume 并进行绑定。 - Static Binding：根据已配置的参数和现有 PersistentVolumes 进行匹配并绑定。

参数	描述
Storage Class	选择动态创建方式后，平台将根据指定 storage class 中的描述动态创建 PersistentVolume。
Access Mode	- ReadWriteOnce (RWO)：可被单个节点以读写模式挂载。 - ReadWriteMany (RWX)：可被多个节点以读写模式挂载。 - ReadOnlyMany (ROX)：可被多个节点以只读模式挂载。 提示：建议结合计划绑定到当前 PersistentVolumeClaim 的工作负载实例数量以及部署控制器类型进行选择。例如，在创建多实例 Deployment 时，由于所有实例都使用同一个 PersistentVolumeClaim，因此不建议选择只能附加到单个节点的 RWO 访问模式。
Capacity	所请求的 PersistentVolume 大小。
Volume Mode	- Filesystem：将 PersistentVolume 作为挂载到 Pod 中的文件目录进行绑定。此模式适用于任何类型的工作负载。 - Block Device：将 PersistentVolume 作为挂载到 Pod 中的原始块设备进行绑定。此模式仅适用于虚拟机。
More	- Labels - Annotations - Selector：选择静态绑定方式后，您可以使用选择器来定位带有特定标签的 PersistentVolumes。PersistentVolume 标签可用于表示存储的特殊属性，例如磁盘类型或地理位置。

5. 单击 **Create**。等待 PersistentVolumeClaim 变为 `Bound` 状态，表示 PersistentVolume 已成功匹配。

通过 **CLI** 创建 **Persistent Volume Claim**

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

操作

- 绑定 **PersistentVolumeClaim**：在创建需要持久化数据存储的应用或工作负载时，绑定 PersistentVolumeClaim 以请求符合要求的 PersistentVolume。
- 使用 **Volume Snapshots** 创建 **PersistentVolumeClaim**：使用 volume snapshots 备份应用数据，并在需要时进行恢复。详情请参见 使用 Volume Snapshots。
- 删除 **PersistentVolumeClaim**：单击详情页右上角的 **Actions** 按钮可删除 PersistentVolumeClaim。在删除之前，请确保该 PersistentVolumeClaim 未绑定到应用或工作负载，且不包含 volume snapshots。删除后，平台将根据回收策略处理 PersistentVolume，这可能会清除 PersistentVolume 中的数据并释放存储资源。在删除 PersistentVolumeClaim 之前，请先确认数据安全要求。

通过 **Web** 控制台扩展 **PersistentVolumeClaim** 存储容量

1. 在左侧导航栏中，单击 Storage > Persistent Volume Claims (PVC)。
2. 找到该 persistent volume claim，然后单击⋮> Expand。
3. 填写新的容量。
4. 单击 **Expand**。扩容过程可能需要一些时间。

通过 **CLI** 扩展 **Persistent Volume Claim** 存储容量

```
kubectl patch pvc example-pvc -n k-1 --type='merge' -p '{
  "spec": {
    "resources": {
      "requests": {
        "storage": "6Gi"
      }
    }
  }
}'
```

INFO

当 PVC 在 Kubernetes 中扩容失败时，管理员可以手动恢复 Persistent Volume Claim（PVC）状态并取消扩容请求。请参见 Recover From PVC Expansion Failure

其他资源

- 如何注解第三方存储能力

使用卷快照

卷快照是持久卷声明（PVC）的某一时刻副本，可用于配置新的持久卷声明（使用快照数据预填充），也可用于将现有持久卷声明回滚到之前的状态，从而实现备份应用数据并按需恢复的效果，确保应用数据的可靠性。

目录

前提条件

- 管理员已为当前集群部署卷快照组件 **Snapshot Controller**，并在存储集群中启用了与快照相关的功能。

- 持久卷声明必须是动态创建的，且其状态必须为 **Bound**。
- 绑定到持久卷声明的存储类必须支持快照功能，例如 **CephRBD Built-in Storage**、**CephFS Built-in Storage** 或 **TopoLVM**。

示例 VolumeSnapshot 自定义资源（CR）

这将使用 CSI 快照类为示例 pvc `PVC` 创建一个快照。

```yaml
# example-snapshot.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: example-pvc-20250527-111124
  namespace: k-1
  labels:
    snapshot.cpaas.io/sourcepvc: example-pvc
  annotations:
    cpaas.io/description: demo
spec:
  volumeSnapshotClassName: csi-cephfs-snapshotclass
  source:
    persistentVolumeClaimName: example-pvc
```

使用 Web 控制台创建卷快照

基于指定的持久卷声明（PVC）创建卷快照

方法一

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Storage > Persistent Volume Claims (PVC)**。
3. 单击列表中对应持久卷声明旁边的⋮，然后选择 **Create Volume Snapshot**。
4. 填写快照描述。此描述有助于记录持久卷的当前状态，例如 *Before Application Upgrade*。
5. 单击 **Create**。快照创建时间取决于网络状况和数据量。

当快照状态变为 **Available** 时，表示创建成功。

方法二

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Storage > Persistent Volume Claims (PVC)**。
3. 单击列表中持久卷声明的名称。
4. 切换到 **Volume Snapshots** 选项卡。
5. 单击 **Create Volume Snapshot**，并根据需要配置相关参数。
6. 单击 **Create**。快照创建时间取决于网络状况和数据量。
   当快照状态变为 **Available** 时，表示创建成功。

以自定义方式创建卷快照

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Storage > Volume Snapshots**。
3. 单击 **Create Volume Snapshot**，并根据需要配置相关参数。
4. 单击 **Create**。快照创建时间取决于网络状况和数据量。
   当快照状态变为 **Available** 时，表示创建成功。

使用 **CLI** 创建卷快照

```
kubectl apply -f example-snapshot.yaml
```

从卷快照创建持久卷声明

目前，平台仅支持使用通过 **Dynamic Provisioning** 创建的存储类所创建的 PVC 来创建卷快照。您可以基于该快照创建新的 PVC 并进行绑定。

注意：从快照创建 PVC 时支持的访问模式，与从存储类创建 PVC 时支持的访问模式不同，表中以 加粗 形式标出。

用于创建卷快照的存储类	单节点读写（**RWO**）	多节点只读（**ROX**）	多节点读写（**RWX**）
TopoLVM	支持	不支持	不支持
CephRBD Block Storage	支持	不支持	不支持
CephFS File Storage	支持	支持	支持

方法一

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Storage > Persistent Volume Claims (PVC)**。
3. 单击列表中持久卷声明的名称。
4. 切换到 **Volume Snapshots** 选项卡。
5. 单击列表中对应卷快照旁边的⋮，然后选择 **Create Persistent Volume Claim**，并配置相关参数。
6. 单击 **Create**。

方法二

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Storage > Volume Snapshots**。
3. 单击列表中对应卷快照旁边的⋮，然后选择 **Create Persistent Volume Claim**，并配置相关参数。
4. 单击 **Create**。

附加资源

- 创建 PVC

克隆 PVC

PVC 克隆会创建一个新的 PersistentVolumeClaim (PVC)，并复制现有 PVC 中的数据。当应用需要一份独立的可写数据副本时，请使用克隆，例如要基于现有应用数据准备测试环境。

克隆直接从源 PVC 创建。相比之下，卷快照会在某个时间点捕获 PVC，并可保留或稍后用于创建 PVC。当你需要直接副本时，请选择克隆；当你需要可恢复的时间点副本时，请选择卷快照。

> **NOTE**
>
> ACP 目前不提供用于克隆 PVC 的 UI。请通过应用带有 `kubectl` 的 YAML 清单来创建克隆后的 PVC。

目录

前提条件

- 仅支持动态供应的 PVC，且其 CSI driver 支持克隆。静态供应不支持 PVC 克隆。
- 源 PVC 处于 `Bound` 阶段且未被使用。
- 源 PVC 和目标 PVC 位于同一 namespace 中。
- 目标 PVC 请求的存储容量至少与源 PVC 的容量相同。
- 源 PVC 和目标 PVC 使用相同的 volume mode。目标 access mode 必须受其 storage class 和 CSI driver 支持。

ACP 存储支持

仅当 CSI driver 和后端存储支持从源卷克隆到该 storage class 时，目标 PVC 才可以使用不同的 storage class。Kubernetes 不会验证此后端兼容性。

存储类型	**PVC** 克隆支持情况	**storage class** 要求
CephRBD 块存储	支持	源 PVC 和目标 PVC 必须在同一个 Ceph 集群中使用兼容的 RBD storage class。如果 Ceph CSI 支持所选的后端配置，则可以使用不同的 RBD storage class，包括使用不同 pool 的 storage class。
CephFS 文件存储	支持	源 PVC 和目标 PVC 必须在同一个 Ceph 集群和同一个 CephFS file system 中使用兼容的 storage class。
TopoLVM	支持 thin-provisioned volumes	源 PVC 和目标 PVC 必须使用相同的 storage class。如果创建克隆时使用不同的 storage class，且其 device class 与源 PVC 不同，则可能无法调度。
NFS Shared Storage	不支持	NFS Shared Storage 不支持 PVC 克隆。

有关创建源 PVC 的信息，请参阅创建 PVC。

创建克隆后的 PVC

创建一个名为 `clone-pvc.yaml` 的清单。请将 namespace、storage class、access mode、volume mode、capacity 和 PVC 名称替换为存储驱动支持的值。在选择目标 storage class 之前，请参阅ACP 存储支持。

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cloned-pvc
  namespace: application-space
spec:
  storageClassName: clone-capable-csi
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 10Gi
  dataSource:
    apiGroup: ""
    kind: PersistentVolumeClaim
    name: source-pvc
```

应用该清单：

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

验证克隆

检查目标 PVC 的状态：

```
kubectl get pvc cloned-pvc -n application-space
```

当 `STATUS` 列显示 `Bound` 时，表示存储驱动已对克隆卷完成供给并绑定。随后，你可以在工作负载中挂载 `cloned-pvc`，并验证是否能够访问预期的源数据。

检查 PVC 事件以排查供给或克隆错误：

```
kubectl describe pvc cloned-pvc -n application-space
```

如果 PVC 仍然处于 `Pending`，或者创建失败，请查看此命令输出的事件。确认所选 storage class 支持 PVC 克隆，并且源 PVC、namespace、请求容量、volume mode 和 access mode 都满足前提条件。

实用指南

通用临时卷

使用 emptyDir

从旧版插件迁

使用本地卷配置持久化存储

使用 NFS 配置持久存储

为持久卷声明

第三方存储能力注解指南

a Bu
步
同步
: 启

故障切换
故障恢复后回切

通用临时卷

Kubernetes 中的通用临时卷是一项功能，允许您使用现有的 StorageClasses 和 CSI 驱动程序，为每个 Pod 动态创建临时（短暂）卷，而无需预先定义 PersistentVolumeClaims（PVC）。

它结合了动态供应的灵活性和 Pod 级别卷声明的简便性。

- 它们是临时卷，会自动：
  - 在 Pod 启动时创建
  - 在 Pod 终止时删除
- 使用与 PersistentVolumeClaim 相同的底层机制
- 需要支持动态供应的 CSI（容器存储接口）驱动程序

目录

临时卷示例

此示例会自动为 Pod 使用指定的 `StorageClass` 创建一个临时 PVC。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ephemeral-demo
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo hello > /data/hello.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: ephemeral-volume
  volumes:
    - name: ephemeral-volume
      ephemeral: 1
        volumeClaimTemplate:
          metadata:
            labels:
              type: temporary
          spec:
            accessModes: [ "ReadWriteOnce" ]
            resources:
              requests:
                storage: 1Gi
            storageClassName: standard
```

1 `Pod` 会使用此模板创建一个 `PVC`。

主要特性

特性	描述
临时性	Pod 删除时，卷也会被删除
动态供应	由支持动态供应的任何 CSI 驱动支持
无需单独 **PVC**	VolumeClaim 直接嵌入在 Pod 规范中

特性	描述
CSI 驱动支持	兼容任何支持的 CSI 驱动（如 EBS、RBD、Longhorn 等）

何时使用通用临时卷

- 当您需要具备以下功能的临时存储时：
  - 可调整大小的卷
  - 快照
  - 加密
  - 非节点本地存储（例如云块存储）
- 适用于：
  - 缓存中间数据
  - 临时工作目录
  - 流水线、AI/ML 工作流

与 **emptyDir** 的区别

特性	`emptyDir`	通用临时卷
底层存储	节点本地磁盘或内存	任何支持 CSI 的后端
存储功能	基础功能	支持快照、加密等高级功能
使用场景	简单的临时存储	需要高级临时存储的场景
可重新调度	否（绑定节点）	是（如果 CSI 卷可附加）

使用 emptyDir

在 Kubernetes 中，emptyDir 是一种简单的临时卷类型，为 Pod 在其生命周期内提供临时存储。它在 Pod 被分配到某个节点时创建，并在 Pod 从该节点移除时删除。

目录

emptyDir 示例

该 Pod 创建了一个临时卷，挂载在 /data 路径，并与容器共享。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-demo
spec:
  containers:
    - name: app
      image: busybox
      command: ["sh", "-c", "echo hello > /data/hello.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

可选的 **Medium** 设置

您可以选择数据的存储位置：

```yaml
emptyDir:
  medium: "Memory"
```

Medium	描述
(默认)	使用节点的磁盘、SSD 或网络存储，具体取决于您的环境
`Memory`	使用 RAM（`tmpfs`）以实现更快访问（但数据是易失性的）

主要特性

特性	描述
初始为空	创建时无数据

特性	描述
容器间共享	同一卷可被 Pod 中的多个容器使用
Pod 删除时销毁	Pod 被移除时，卷也被销毁
节点本地	卷存储在节点的本地磁盘或内存中
速度快	适合对性能敏感的临时存储空间

常见用例

- 缓存中间构建产物
- 缓冲日志
- 临时工作目录
- 在同一 Pod 内的容器间共享数据（如 sidecar 容器）

从旧版插件迁移 Volume Snapshot 组件

在 ACP 4.4 中，新集群不再需要单独安装 灵雀云容器平台 **Snapshot Management** 集群插件。如果现有的业务集群已经安装了旧版插件，请使用此操作步骤将该集群迁移到 ACP 4.4 的 snapshot 组件，同时不删除现有的 snapshot 资源。

目录

相关文档

适用范围

仅当满足以下所有条件时，本操作步骤才适用：

- 业务集群使用的是旧版 灵雀云容器平台 **Snapshot Management** 集群插件。
- 你要将该业务集群迁移到 ACP 4.4 或更高版本。
- 你希望在删除旧版插件后由 ACP 管理集群级 snapshot 组件。

如果其他平台或管理员管理的组件仍在提供 Snapshot Controller，请勿使用此操作步骤。一个集群中只能有一个 Snapshot Controller 进行管理。

影响

卸载旧版插件到新的 snapshot 组件就绪之间，预计会有一段短暂的间隔。在此间隔内，集群中没有正在运行的 Snapshot Controller。新的 snapshot 创建或删除请求可能会一直处于 Pending 状态，直到新组件就绪。

现有的 `VolumeSnapshot`、`VolumeSnapshotContent`、`VolumeSnapshotClass`、`VolumeGroupSnapshot`、`VolumeGroupSnapshotContent` 和 `VolumeGroupSnapshotClass` 资源会被保留。

WARNING

在此迁移过程中不要删除 snapshot CRD。删除 snapshot CRD 可能会删除所有集群范围的 snapshot 对象并导致数据丢失。

WARNING

不要使用会删除 snapshot CRD 或 snapshot 自定义资源的清理操作。

前提条件

- 你对 global 集群和目标业务集群具有管理员访问权限。
- 目标业务集群运行在 ACP 4.4 或更高版本，或者正在升级到 ACP 4.4。
- ACP Storage 组件已安装在目标业务集群中。
- 在执行 `kubectl patch` 命令的工作站上，已安装 `jq` 命令行工具。
- 你已为 snapshot 操作预留维护窗口。

操作步骤

1 记录当前的 **snapshot** 资源

在进行更改之前，请在目标业务集群中运行以下命令：

```
kubectl get crd | grep -E 'snapshot.storage.k8s.io|groupsnapshot.storage.k8s.io'
kubectl get volumesnapshots.snapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumesnapshotcontents.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumesnapshotclasses.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshots.groupsnapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumegroupsnapshotcontents.groupsnapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshotclasses.groupsnapshot.storage.k8s.io --ignore-not-found
```

请保留输出结果，以便在迁移后进行比较。

2 确认已安装旧版插件

在 global 集群中运行以下命令。将 `<workload-cluster-name>` 替换为目标业务集群名称：

```
kubectl get moduleinfo \
  -l cpaas.io/module-name=snapshot,cpaas.io/cluster-name=<workload-cl
uster-name>
```

如果此命令返回资源，则表示已安装旧版 Snapshot Management 插件。

3 卸载旧版插件

从 global 集群卸载旧版插件。你可以使用 Web 控制台或 YAML。

使用 Web 控制台：

1. 进入 **Administrator** > **Marketplace** > **Cluster Plugins**。
2. 选择目标业务集群。
3. 找到 灵雀云容器平台 **Snapshot Management**。
4. 卸载该插件。

使用 YAML 时，从 global 集群中删除插件安装资源：

```
kubectl delete moduleinfo \
  -l cpaas.io/module-name=snapshot,cpaas.io/cluster-name=<workload-cl
uster-name>
```

卸载操作会移除旧版 Snapshot Controller 工作负载，但不得删除 snapshot CRD 或现有 snapshot 资源。

4 验证旧版控制器已移除

在目标业务集群中运行以下命令：

```
kubectl get deployment -A | grep -E 'snapshot-controller|snapshot' ||
true
```

仅在不再存在旧版 Snapshot Controller Deployment 时继续下一步。

如果仍有其他 Snapshot Controller 在运行，请停止操作。在另一个控制器仍处于活动状态时启用 ACP 接管 snapshot 管理，可能会导致同一集群中出现两个 Snapshot Controller。

5 启用 **ACP snapshot** 管理

在目标业务集群中运行以下命令：

```
kubectl patch storagefoundation default --type=json -p "$(
  kubectl get storagefoundation default -o json | jq -c '
    (.spec.operators // []) as $operators |
    (
      if any($operators[]?; .name == "snapshot") then
        $operators | map(
          if .name == "snapshot" then
            .enabled = true | .upgradePolicy.mode = "Automatic"
          else
            .
          end
        )
      else
        $operators + [{
          "name": "snapshot",
          "enabled": true,
          "upgradePolicy": {"mode": "Automatic"}
        }]
      end
    ) as $updated |
    [{
      "op": (if .spec.operators == null then "add" else "replace" en
d),
      "path": "/spec/operators",
      "value": $updated
    }]
  '
)"
```

此补丁会添加或更新 `snapshot` 条目，并保留所有现有条目。仅在旧版控制器已移除后使用此设置。

6 等待 snapshot 组件就绪

在目标业务集群中运行以下命令：

```
kubectl get storagefoundation default \
  -o jsonpath='{range .status.operators[?(@.name=="snapshot")]}phase=
{.phase}{"\n"}{range .conditions[*]}{.type}={.status} reason={.reaso
n} message={.message}{"\n"}{end}{end}'
```

当 `snapshot` 条目报告 `phase=Ready` 时，表示迁移成功。

7 验证 snapshot CRD 是否存在

在目标业务集群中运行以下命令：

```
for crd in \
  volumesnapshots.snapshot.storage.k8s.io \
  volumesnapshotcontents.snapshot.storage.k8s.io \
  volumesnapshotclasses.snapshot.storage.k8s.io \
  volumegroupsnapshots.groupsnapshot.storage.k8s.io \
  volumegroupsnapshotcontents.groupsnapshot.storage.k8s.io \
  volumegroupsnapshotclasses.groupsnapshot.storage.k8s.io; do
  kubectl get crd "$crd"
done
```

snapshot 组件就绪后，应存在全部 6 个 CRD。

8 验证现有 snapshot 资源已保留

将以下输出与迁移前记录的输出进行比较：

```
kubectl get volumesnapshots.snapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumesnapshotcontents.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumesnapshotclasses.snapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshots.groupsnapshot.storage.k8s.io -A --ignore-not-found
kubectl get volumegroupsnapshotcontents.groupsnapshot.storage.k8s.io --ignore-not-found
kubectl get volumegroupsnapshotclasses.groupsnapshot.storage.k8s.io --ignore-not-found
```

现有 snapshot 资源应仍然存在。

回滚

如果 snapshot 组件未变为 `Ready`，请检查 snapshot 状态：

```
kubectl get storagefoundation default \
  -o jsonpath='{range .status.operators[?(@.name=="snapshot")]}phase={.phase}{"\n"}{range .conditions[*]}{.type}={.status} reason={.reason} message={.message}{"\n"}{end}{end}'
```

如果你需要停止 ACP snapshot 管理，同时保留 snapshot CRD 和 snapshot 资源，请将 `snapshot` 条目设置为 `enabled: false`：

```
kubectl patch storagefoundation default --type=json -p "$(
  kubectl get storagefoundation default -o json | jq -c '
    (.spec.operators // []) as $operators |
    (
      if any($operators[]?; .name == "snapshot") then
        $operators | map(
          if .name == "snapshot" then
            .enabled = false | .upgradePolicy.mode = "Automatic"
          else
            .
          end
        )
      else
        $operators + [{
          "name": "snapshot",
          "enabled": false,
          "upgradePolicy": {"mode": "Automatic"}
        }]
      end
    ) as $updated |
    [{
      "op": (if .spec.operators == null then "add" else "replace" end),
      "path": "/spec/operators",
      "value": $updated
    }]
  '
)"
```

要恢复旧版插件，请在 global 集群中通过 **Administrator** > **Marketplace** > **Cluster Plugins** 重新安装 灵雀云容器平台 **Snapshot Management**。仅在 ACP snapshot 管理已被禁用且其控制器不再运行时执行此操作。

故障排查

Snapshot 状态报告 `Skipped`

`Skipped` 表示集群中已经存在一个或多个 snapshot CRD，并且 ACP 没有自动接管 snapshot 管理。在从旧版插件迁移期间，在你显式启用 ACP snapshot 管理之前，这属于预期行为：

```
name: snapshot
enabled: true
```

Snapshot 操作仍处于 Pending

检查是否正在运行 Snapshot Controller：

```
kubectl get deployment -A | grep -E 'snapshot-controller|snapshot'
```

如果没有控制器在运行且 snapshot 状态报告为 `Skipped`，请确认旧版插件已卸载，然后启用 ACP snapshot 管理。

在另一个控制器仍在运行时启用了 snapshot 管理

这可能会导致同一集群中出现两个 Snapshot Controller。请先禁用 ACP snapshot 管理：

```
kubectl patch storagefoundation default --type=json -p "$(
  kubectl get storagefoundation default -o json | jq -c '
    (.spec.operators // []) as $operators |
    (
      if any($operators[]?; .name == "snapshot") then
        $operators | map(
          if .name == "snapshot" then
            .enabled = false | .upgradePolicy.mode = "Automatic"
          else
            .
          end
        )
      else
        $operators + [{
          "name": "snapshot",
          "enabled": false,
          "upgradePolicy": {"mode": "Automatic"}
        }]
      end
    ) as $updated |
    [{
      "op": (if .spec.operators == null then "add" else "replace" end),
      "path": "/spec/operators",
      "value": $updated
    }]
  '
)"
```

在移除 ACP snapshot 组件后，请决定应由哪个组件提供集群级 snapshot 能力。集群中只应保留一个 Snapshot Controller。

相关文档

- 部署 Volume Snapshot 组件
- 使用 Volume Snapshot
- Cluster Plugin

使用本地卷配置持久化存储

ACP 可以使用本地卷进行持久化存储。本地持久卷允许您通过标准 persistent volume claim 接口访问本地存储设备，例如磁盘或分区。

由于系统会感知卷的节点约束，因此可以在不手动将 Pod 调度到节点的情况下使用本地卷。但是，本地卷仍然受底层节点可用性的限制，不适用于所有应用程序。

> **NOTE**
>
> 本地卷只能作为静态创建的 persistent volume 使用。

目录

前提条件

- 下载与您的平台架构对应的 **Alauda Build of LocalStorage** 安装包。
- 使用 上架软件包 机制上传 **Alauda Build of LocalStorage** 安装包。
- 您有一块满足以下条件的本地磁盘：
  - 它已连接到某个节点。
  - 它未挂载。
  - 它不包含分区。

操作步骤

1 安装 Local Storage Operator

1. 登录后，进入 **Administrator** 页面。
2. 单击 **Marketplace** > **OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **Alauda Build of LocalStorage**，单击 **Install**，然后进入 **Install Alauda Build of LocalStorage** 页面。
   配置参数：

参数	推荐配置
Channel	默认通道为 `stable`。
Installation Mode	`Cluster`：集群中的所有命名空间共享一个用于创建和管理的 Operator 实例，因此资源占用更低。
Installation Place	选择 `Recommended`，命名空间仅支持 **acp-storage**。
Upgrade Strategy	`Manual`：当 Operator Hub 中有新版本时，需要手动确认，才能将 Operator 升级到最新版本。

2 使用 Local Storage Operator 提供本地卷

无法通过动态供给创建本地卷。相反，可以通过 Local Storage Operator 创建 persistent volume。local volume provisioner 会在所定义资源中指定的路径下查找任何文件系统或块卷设备。

1. 创建本地卷资源。该资源必须定义本地卷所在的节点和路径。

> **NOTE**
>
> 不要对同一设备使用不同的 storage class 名称。这样会创建多个 persistent volumes（PVs）。

在集群的 **control node** 上执行命令。

```
cat << EOF | kubectl create -f -
apiVersion: "local.storage.openshift.io/v1"
kind: "LocalVolume"
metadata:
  name: "local-disks"
  namespace: "acp-storage" 1
spec:
  nodeSelector: 2
    nodeSelectorTerms:
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - worker-01
          - worker-02
          - worker-03
  storageClassDevices:
    - storageClassName: "local-sc" 3
      forceWipeDevicesAndDestroyAllData: false 4
      volumeMode: Filesystem 5
      fsType: xfs 6
      devicePaths: 7
        - /path/to/device 8
EOF
```

1 安装 Local Storage Operator 的命名空间，默认值为 `acp-storage`。

2 可选：包含本地存储卷所连接节点列表的节点选择器。此示例使用从 `kubectl get node` 获取的节点主机名。如果未定义该值，Local Storage Operator 将尝试在所有可用节点上查找匹配的磁盘。

3 创建 persistent volume 对象时要使用的 storage class 名称。如果该 storage class 不存在，Local Storage Operator 会自动创建它。请务必使用能够唯一标识这组本地卷的 storage class。

4 控制 operator 在使用前是否清除所列设备。默认值为 `false`。警告：将 `forceWipeDevicesAndDestroyAllData: true` 设置为破坏性操作，它会擦除这些设备上现有的分区表、文件系统签名和数据。仅当您确认可以安全清除这些设备时才使用。

5 卷模式，可以是 `Filesystem` 或 `Block`，用于定义本地卷类型。

6 可选：本地卷首次挂载时创建的文件系统。只有在 `volumeMode` 设置为 `Filesystem` 时才需要配置此参数。

7 包含可选本地存储设备列表的路径。

8 请将此值替换为 `LocalVolume` 资源 `by-id` 的实际本地磁盘 filepath，例如 `/dev/disk/by-id/wwn`。当 provisioner 成功部署后，会为这些本地磁盘创建 PV。

2. 验证已创建 persistent volumes：
   在集群的 **control node** 上执行命令。

```
kubectl get pv
```

示例输出：

```
NAME                CAPACITY   ACCESS MODES   RECLAIM POLICY   STA
TUS      CLAIM   STORAGECLASS   REASON   AGE
local-pv-n8xlrd2a   100Gi      RWO            Delete           Ava
ilable           local-sc                88m
local-pv-tc78vc73   100Gi      RWO            Delete           Ava
ilable           local-sc                82m
local-pv-q86px4df   100Gi      RWO            Delete           Ava
ilable           local-sc                48m
```

> **NOTE**

编辑 LocalVolume 对象不会更改现有 persistent volumes 的 fsType 或 volumeMode，因为这样做可能会导致破坏性操作。

3. 创建本地卷持久卷声明

   在集群的 **control node** 上执行命令。

```
cat << EOF | kubectl create -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-pvc-name 1
spec:
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem 2
  resources:
    requests:
      storage: 100Gi 3
  storageClassName: local-sc 4
EOF
```

   1. PVC 的名称。
   2. PVC 的类型。默认为 `Filesystem`。
   3. PVC 可用的存储容量。
   4. 声明所需的 storage class 名称。

自动发现本地存储设备

Local Storage Operator 会自动执行本地存储发现。

前提条件

- 您已经安装了 Local Storage Operator。

操作步骤

1. 创建 LocalVolumeDiscovery 对象

   在集群的 **control node** 上执行命令。

```
cat << EOF | kubectl create -f -
apiVersion: local.storage.openshift.io/v1alpha1
kind: LocalVolumeDiscovery
metadata:
  name: auto-discover-devices
  namespace: acp-storage
spec:
  nodeSelector: (1)
    nodeSelectorTerms:
      - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - worker-01
              - worker-02
              - worker-03
  tolerations: (2)
    - operator: Exists
EOF
```

   (1) 可选：必须运行自动检测策略的节点。如果未定义该值，Local Storage Operator 将尝试在所有可用节点上执行自动检测。

   (2) 可选：如果指定，则 tolerations 是传递给 LocalVolumeDiscovery Daemon 的 toleration 列表

2. 验证发现结果

   在集群的 **control node** 上执行命令。

```
kubectl -n acp-storage get localvolumediscoveryresult
```

   示例输出：

```
NAME                              AGE
discovery-result-worker-01        21m
discovery-result-worker-02        21m
discovery-result-worker-03        21m
```

会为 LocalVolumeDiscovery 对象中选定的节点生成一个专用的 LocalVolumeDiscoveryResult 对象，您可以通过它查看在该节点上发现的块设备。

使用 NFS 配置持久存储

ACP 集群支持使用 NFS 的持久存储。PersistentVolumes (PVs) 和 PersistentVolumeClaims (PVCs) 提供一个抽象层，用于在项目内创建和使用存储卷。虽然可以将 NFS 配置细节直接嵌入 Pod 定义中，但这种方式不会将该卷创建为一个独立、隔离的集群资源，从而增加冲突风险。

目录

前提条件

- 在 ACP 中将存储挂载为卷之前，底层基础设施中必须已存在该存储。
- 要创建 NFS 卷，只需要一份 NFS 服务器和导出路径列表。

操作步骤

1 为 PV 创建对象定义

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-example 1
spec:
  capacity:
    storage: 1Gi 2
  accessModes:
  - ReadWriteOnce 3
  nfs: 4
    path: /tmp 5
    server: 10.0.0.3 6
  persistentVolumeReclaimPolicy: Retain 7
EOF
```

1 卷名称。

2 存储量。

3 虽然这看起来与控制对卷的访问有关，但实际上它的作用类似于标签，用于将 PVC 与 PV 匹配。目前不会基于 accessModes 强制执行任何访问规则。

4 正在使用的卷类型，在本例中为 nfs 插件。

5 NFS 服务器地址。

6 NFS 导出路径。

7 PVC 被删除后会发生什么（Retain、Delete、Recycle）。

2 验证 PV 已创建

命令

```
kubectl get pv
```

输出示例

```
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATU
S   CLAIM   STORAGECLASS   REASON   AGE
pv-nfs-example   1Gi         RWO             Retain           Avail
able                                 10s
```

3 创建一个引用该 PV 的 PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-claim1
spec:
  accessModes:
  - ReadWriteOnce 1
  resources:
    requests:
      storage: 1Gi 2
  volumeName: pv-nfs-example 3
  storageClassName: ""
```

1. 访问模式不会强制执行安全性，而是充当标签，用于将 PV 与 PVC 匹配。
2. 该声明会查找提供 1Gi 或更大容量的 PV。
3. 要使用的 PV 名称。

4 验证已创建 persistent volume claim

命令

```
kubectl get pvc
```

输出示例

```
NAME            STATUS   VOLUME           CAPACITY   ACCESS MODE
S   STORAGECLASS   AGE
nfs-claim1      Bound    pv-nfs-example   1Gi        RWO
10s
```

通过分区导出强制执行磁盘配额

要强制执行磁盘配额和大小限制，可以使用磁盘分区。将每个分区指定为专用导出点，每个导出点对应一个独立的 PersistentVolume (PV)。

虽然 ACP 要求 PV 名称唯一，但管理员还必须确保每个导出的 NFS 服务器和路径组合都唯一。

这种分区方式可实现精确的容量管理。开发人员通过指定所需容量（例如 `10Gi`）来申请持久存储，而 ACP 会将该请求匹配到一个由分区或导出提供至少该容量的 PV。配额强制适用于分配的分区或导出中可用的存储空间。

NFS 卷安全性

NFS 卷安全性取决于权限匹配。在配置用于共享的 NFS 卷之前，请检查 POSIX 权限、进程 UID 和 supplemental groups。

开发人员可以通过以下任一方式请求 NFS 存储：

- 按名称引用 PersistentVolumeClaim (PVC)，或
- 在其 Pod 规范的 volumes 部分直接配置 NFS 卷插件。

在 NFS 服务器上，/etc/exports 文件定义可访问目录的导出规则。每个已导出的目录都会保留其原生 POSIX owner/group IDs。

ACP NFS 插件的关键行为：

1. 在将卷挂载到容器时保留源目录中的精确 POSIX 所有权和权限
2. 运行容器时不会强制进程 UID 与挂载所有权匹配——这是有意为之的安全措施

例如，考虑一个具有以下服务器端属性的 NFS 目录：

命令

```
ls -l /share/nfs -d
```

输出示例

```
drwxrws---. nfsnobody 5555   /share/nfs
```

命令

```
id nfsnobody
```

输出示例

```
uid=65534(nfsnobody) gid=65534(nfsnobody) groups=65534(nfsnobody)
```

因此，容器必须使用 UID 65534（nfsnobody 所有者）运行，或者在其 supplemental groups 中包含 5555，才能访问该目录。

NOTE

注意 65534 的所有者 ID 仅用作示例。尽管 NFS 的 root_squash 会将 root、uid 0 映射为 nfsnobody，也就是 uid 65534，但 NFS 导出可以使用任意所有者 ID。NFS 导出并不要求所有者必须是 65534。

组 ID

NFS 访问管理建议（当导出权限已固定时） 如果无法修改 NFS 导出的权限，建议通过 supplemental groups 管理访问。

ACP 中的 supplemental groups 是控制共享文件存储（如 NFS）访问的一种常见机制。

与块存储相比：对块存储卷（例如 iSCSI）的访问通常通过在 Pod 的 securityContext 中设置 fsGroup 值来管理。此方法利用挂载时文件系统组所有权的变更。

> **NOTE**
>
> 要获得对持久存储的访问权限，通常优先使用 supplemental group ID，而不是 user ID。

由于示例目标 NFS 目录的 group ID 为 5555，Pod 可以在 Pod 的 securityContext 定义中使用 supplementalGroups 来定义该 group ID。例如：

```
spec:
  containers:
    - name:
    ...
  securityContext: 1
    supplementalGroups: [5555] 2
```

1. securityContext 必须在 pod 级别定义，而不是在某个特定容器下定义。
2. 为 Pod 定义的 GID 数组。在这种情况下，数组中有一个元素。其他 GID 应以逗号分隔。

用户 ID

用户 ID 可以在容器镜像中定义，也可以在 Pod 定义中定义。

> **NOTE**
>
> 通常优先使用 supplemental group ID 来获得对持久存储的访问权限，而不是使用 user ID。

在上面所示的目标 NFS 目录示例中，容器需要将其 UID 设置为 65534，暂不考虑 group ID，因此可以将以下内容添加到 Pod 定义中：

```
spec:
  containers: 1
  - name:
  ...
    securityContext:
      runAsUser: 65534 2
```

1 Pod 包含一个针对每个容器的 securityContext 定义，以及一个适用于 Pod 中所有容器的 pod securityContext。

2 65534 是 nfsnobody 用户。

导出设置

要允许任意容器用户读写该卷，NFS 服务器上的每个已导出卷都应符合以下条件：

- 每个导出项都必须按以下格式导出：

```
# replace 10.0.0.0/24 to trusted CIDRs/hosts
/<example_fs> 10.0.0.0/24(rw,sync,root_squash,no_subtree_check)
```

- 必须配置防火墙以允许到挂载点的流量。
  - 对于 NFSv4，请配置默认端口 2049（nfs）。

```
iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
```

  - 对于 NFSv3，需要配置三个端口：2049（nfs）、20048（mountd）和 111（portmapper）。

```
iptables -I INPUT 1 -p tcp --dport 2049 -j ACCEPT
iptables -I INPUT 1 -p tcp --dport 20048 -j ACCEPT
iptables -I INPUT 1 -p tcp --dport 111 -j ACCEPT
```

- 必须设置 NFS 导出和目录，使目标 Pod 可以访问它们。可以将导出设置为由容器的主 UID 拥有，或者按照上面的组 ID 示例，通过 supplementalGroups 为 Pod 提供组访问权限。

回收资源

NFS 实现了 灵雀云容器平台 可回收插件接口。自动流程会根据每个 PV 上设置的策略处理回收任务。

默认情况下，PV 设置为 Retain。

一旦某个 PVC 的声明被删除且 PV 被释放，就不应重复使用该 PV 对象。应使用与原始卷相同的基本卷详情创建一个新的 PV。

例如，管理员创建一个名为 nfs1 的 PV：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs1
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

用户创建 PVC1，它会绑定到 nfs1。然后用户删除 PVC1，释放对 nfs1 的声明。这会使 nfs1 变为 Released。如果管理员希望提供同一个 NFS 共享，应创建一个具有相同 NFS 服务器详细信息但不同 PV 名称的新 PV：

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs2
spec:
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.1
    path: "/"
```

不建议删除原始 PV 后再使用相同名称重新创建。尝试手动将 PV 的状态从 Released 更改为 Available 会导致错误并可能造成数据丢失。

为持久卷声明配置灾难恢复

要为 Application PVC 实现跨集群灾难恢复，请使用 **Alauda Build of VolSync**。

目录

概述

Alauda Build of VolSync 是一个 operator，用于在集群内或跨集群对持久卷进行异步复制。VolSync 提供的复制与底层存储系统无关，因此可以对通常不支持远程复制的存储类型进行双向复制。此外，它还可以在不同类型（以及不同厂商）的存储之间进行复制。

术语

术语	说明
Primary Cluster	活跃的生产站点。
Secondary Cluster	备用恢复站点；保持待命状态，随时准备在发生灾难时接管。
Stateful Application	使用 PVC 进行数据持久化的应用。
ReplicationSource	ReplicationSource 是一个 VolSync 资源，可用于定义源 PVC 和复制 mover 类型，从而使你能够将 PVC 数据复制或同步到远程位置。
ReplicationDestination	ReplicationDestination 是一个 VolSync 资源，可用于定义 VolSync 复制或同步的目标位置。
Data Movers	VolSync 的 data mover 负责将数据从一个位置复制到另一个位置。 支持的 mover： • Rclone • Restic • Rsync

前提条件

- 下载与你的平台架构对应的 **Alauda Build of VolSync** 安装包。
- 使用上架软件包机制将 **Alauda Build of VolSync** 安装包上传到 Primary 和 Secondary 两个集群。
- 两个集群都已部署 灵雀云容器平台 快照管理。
- PVC 所使用的存储必须由 **CSI** 提供，并支持 **snapshot** 功能。

部署 **Alauda Build of VolSync**

1. 登录后，进入 管理员 页面。
2. 单击 **Marketplace** > **OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **Alauda Build of VolSync**，单击 **Install**，进入 **Install Alauda Build of VolSync** 页面。
   配置参数：

参数	推荐配置
Channel	默认 channel 为 `stable`。
Installation Mode	`Cluster`：集群中所有 namespace 共享一个单独的 Operator 实例进行创建和管理，因此资源占用更低。
Installation Place	选择 `Recommended`，Namespace 仅支持 **volsync-system**。
Upgrade Strategy	`Manual`：当 Operator Hub 中有新版本时，需要手动确认才能将 Operator 升级到最新版本。

配置定时同步

完成 PVC 的定时同步配置后，VolSync 会按照指定的间隔，自动将数据从 `ReplicationSource` 同步到 `ReplicationDestination`。

下面配置的是从 Primary 集群到 Secondary 集群的同步。如果需要从 Secondary 集群同步到 Primary 集群，请通过交换集群角色来调整下面的示例。

1 创建 rsync-tls Data Mover Secret

在 **Primary** 和 **Secondary** 两个集群上都创建该 Secret；如果 Secret 已存在，则跳过此步骤。

Command

```yaml
apiVersion: v1
data:
  psk.txt: <psk>
kind: Secret
metadata:
  name: <name>
  namespace: <namespace>
type: Opaque
```

Example

```yaml
apiVersion: v1
data:
  # echo -n 1:23b7395fafc3e842bd8ac0fe142e6ad1 | base64
  psk.txt: MToyM2I3Mzk1ZmFmYzNlODQyYmQ4YWMwZmUxNDJlNmFkMQ==
kind: Secret
metadata:
  name: volsync-rsync-tls
  namespace: default
type: Opaque
```

参数：

参数	说明
name	secret 的名称
namespace	secret 的 namespace，应与应用相同
psk	该字段遵循 stunnel 期望的格式：`<id>:<至少 32 位十六进制数字>`。 例如：`1:23b7395fafc3e842bd8ac0fe142e6ad1`。

2 创建 ReplicationDestination 资源

在 **Secondary** 集群上创建 `ReplicationDestination`

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: rd-<pvc-name>
  namespace: <namespace>
spec:
  rsyncTLS:
    copyMethod: Snapshot
    destinationPVC: <pvc-name>
    keySecret: <key-secret>
    serviceType: <service-type>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: rd-pvc-01
  namespace: default
spec:
  rsyncTLS:
    copyMethod: Snapshot
    destinationPVC: pvc-01
    keySecret: volsync-rsync-tls
    serviceType: NodePort
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
EOF
```

参数：

参数	说明
namespace	与应用相同的 Namespace
pvc-name	应用使用的已存在 PVC 的名称
key-secret	包含用于认证与源端连接的 TLS-PSK 密钥的 Secret 名称，请在 步骤 1 中创建
service-type	VolSync 会创建一个 Service，以允许源端连接到目标端。该字段决定该 Service 的类型。允许的值为 `ClusterIP`、`LoadBalancer` 或 `NodePort`。
storageclass-name	应用 PVC 所使用的 storageclass 名称

参数	说明
volumesnapshotclass-name	与应用 PVC 对应的 volumesnapshotclass 名称

NOTE

关于 **service type**

如果指定 `ClusterIP`，Service 将接收从“集群网络”地址池分配的 IP 地址。默认情况下，这组地址无法从集群外部访问，因此不太适合跨集群复制。不过，诸如 Submariner 之类的网络附加组件可以打通集群网络，因此这是一个不错的选择。

如果指定 `LoadBalancer`，将分配一个可从外部访问的 IP 地址。这需要集群支持负载均衡器，例如云厂商提供的负载均衡器，或者在物理集群场景下使用 MetalLB。虽然这是在云环境中分配可访问地址的最简单方式，但负载均衡器通常会产生额外成本，并且数量有限。

3 创建 ReplicationSource 资源

在 **Primary** 集群上创建 `ReplicationSource`

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: <pvc-name>
  trigger:
    schedule: <schedule>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: pvc-01
  trigger:
    schedule: "*/10 * * * *"
EOF
```

参数：

参数	说明
namespace	Namespace 名称，与应用相同
pvc-name	应用 PVC 的名称
key-secret	在 步骤 1 中创建的 volsync secret 名称
address	指定复制目标的 ssh server 地址。可直接从 `ReplicationDestination` 的 `.status.rsync.address` 字段获取。

参数	说明
port	连接目标端的 service port
storageclass-name	应用 PVC 所使用的 storageclass 名称
volumesnapshotclass-name	与应用 PVC 对应的 volumesnapshotclass 名称
schedule	同步计划，由 cronspec 定义，因此非常灵活。既可以指定时间间隔，也可以指定具体时间和/或日期。

4 检查同步状态

检查来自 `ReplicationSource` 的同步状态

Command

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status}'
```

Example

```
kubectl -n default get ReplicationSource rs-pvc-01 -o jsonpath='{.status}'
```

上一次同步完成于 `.status.lastSyncTime` ，耗时 `.status.lastSyncDuration` 秒。

下一次计划同步时间为 `.status.nextSyncTime` 。

配置一次性同步

一次性同步由手动触发。通过在 `ReplicationSource` 资源的 trigger 规范下为 manual 字段设置一个唯一字符串来控制。应用该配置后，同步任务会立即运行一次。

1 创建一次性 **ReplicationSource** 资源

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>-latest
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: <pvc-name>
  trigger:
    manual: <manual-id>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01-latest
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: pvc-01
  trigger:
    manual: latest
```

与 定时同步 的唯一区别是 `.spec.trigger` 应设置为 **manual**。

2 检查同步状态

Command

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status.lastManualSync}'
```

Example

```
kubectl -n default get ReplicationSource rs-pvc-01-latest -o jso
npath='{.status.lastManualSync}'
```

如果输出与 `<manual-id>` 匹配，则表示同步已完成。

为应用 PVC 启用灾难恢复

1 部署有状态应用

1. 在 **Primary** 集群上部署有状态应用

▶ 单击查看

2. 在 **Secondary** 集群上创建应用 PVC

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-01
  namespace: default
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: sc-cephfs
  volumeMode: Filesystem
EOF
```

2 配置 PVC 灾难恢复

设置 `Primary-to-Secondary` 同步

参见 配置定时同步

计划迁移

用户场景：

在两个集群都正常运行时，将业务服务从 Primary 集群迁移到 Secondary 集群。

操作步骤

1. 缩减应用 **Pod**
   在 **Primary** 集群上缩减所有使用 dr PVC 的应用 Pod。
2. 删除 **ReplicationSource** 资源
   在 **Primary** 集群上删除 `ReplicationSource`

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01
   ```

3. 创建一次性同步
   从 **Primary** 集群发起一次同步任务，确保 **Secondary** 集群中的数据是 `up-to-date`。
   在 **Primary** 集群上创建 `ReplicationSource`
   参见 配置一次性同步
4. 删除一次性同步
   一次性同步完成后，删除一次性 `ReplicationSource` 资源

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

Example

```
kubectl -n default delete ReplicationSource rs-pvc-01-latest
```

5. 删除 **ReplicationDestination** 资源
   在 **Secondary** 集群上删除 `ReplicationDestination`

Command

```
kubectl -n <namespace> delete ReplicationDestination <rd-name>
```

Example

```
kubectl -n default delete ReplicationDestination rd-pvc-01
```

6. 扩容应用 **Pod**
   在 **Secondary** 集群上扩容所有使用 dr PVC 的应用 Pod。
7. 建立 **Secondary-to-Primary** 同步
   通过在 **Primary** 集群上创建 `ReplicationDestination`，并在 **Secondary** 集群上创建 `ReplicationSource`，为 PVC 灾难恢复建立 `Secondary-to-Primary` 集群同步。
   参见 配置定时同步

故障切换

用户场景：

Primary 集群突然关闭后，将服务切换到 Secondary 集群。

操作步骤

为确保数据完整性（以防 primary 集群在同步过程中发生故障），请在 **Secondary** 集群上执行本地同步。使用从应用 PVC 的最后一个 snapshot 恢复得到的 PVC 作为源端，使用应用当前的 PVC 作为目标端来执行数据同步。

1. 恢复 **PVC**

   在 **Secondary** 集群上从 `ReplicationDestination` 恢复 PVC

   Command

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restored-<pvc-name>
  namespace: <namespace>
spec:
  accessModes: [<access-modes>]
  dataSourceRef:
    kind: ReplicationDestination
    apiGroup: volsync.backube
    name: <rd-name>
  resources:
    requests:
      storage: <pvc-size>
  storageClassName: <storageclass-name>
EOF
```

   Example

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restored-pvc-01
  namespace: default
spec:
  accessModes: [ReadWriteMany]
  dataSourceRef:
    kind: ReplicationDestination
    apiGroup: volsync.backube
    name: rd-pvc-01
  resources:
    requests:
      storage: 10Gi
  storageClassName: sc-cephfs
EOF
```

2. 创建本地 **ReplicationSource** 资源

   在 **Secondary** 集群上创建 `ReplicationSource` 资源

Command

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-<pvc-name>-local
  namespace: <namespace>
spec:
  rsyncTLS:
    address: <address>
    copyMethod: Snapshot
    keySecret: <key-secret>
    port: <port>
    storageClassName: <storageclass-name>
    volumeSnapshotClassName: <volumesnapshotclass-name>
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: restored-<pvc-name>
  trigger:
    manual: <manual-id>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: rs-pvc-01-local
  namespace: default
spec:
  rsyncTLS:
    address: 192.168.129.201
    copyMethod: Snapshot
    keySecret: volsync-rsync-tls
    port: 30532
    storageClassName: sc-cephfs
    volumeSnapshotClassName: csi-cephfs-snapshotclass
    moverSecurityContext:
      fsGroup: 65534
      runAsGroup: 65534
      runAsNonRoot: true
      runAsUser: 65534
      seccompProfile:
        type: RuntimeDefault
  sourcePVC: restored-pvc-01
  trigger:
    manual: latest
EOF
```

参数参见 配置一次性同步

3. 等待同步完成

Command

```
kubectl -n <namespace> get ReplicationSource <rs-name> -o jsonpath='{.status.lastManualSync}'
```

Example

```
kubectl -n default get ReplicationSource rs-pvc-01-local -o jsonpath='{.status.lastManualSync}'
```

如果输出与 `<manual-id>` 匹配，则表示同步已完成。

4. 删除本地 **ReplicationSource**

   在 **Secondary** 集群上删除本地 `ReplicationSource`

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01-local
   ```

5. 删除 **ReplicationDestination**

   在 **Secondary** 集群上删除 `ReplicationDestination`

   Command

   ```
   kubectl -n <namespace> delete ReplicationDestination <rd-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationDestination rd-pvc-01
   ```

6. 扩容应用 **Pod**

   在 **Secondary** 集群上扩容所有应用 Pod。

故障恢复后回切

用户场景：

Primary 集群现已恢复并可正常运行，因此需要将服务切回该集群。

操作步骤

1. 缩减 **Primary** 集群上的应用 **Pod**
   当 primary 集群重新上线后，应用 Pod 会自动恢复。但是，必须先缩减服务以停止流量。在将 secondary 集群中的最新数据同步到 primary 集群后，再扩容应用以恢复正常运行。
2. 删除 **Primary** 集群上的 **ReplicationSource**
   需要先删除在 **Primary** 集群故障前创建的 `ReplicationSource`。

   Command

   ```
   kubectl -n <namespace> delete ReplicationSource <rs-name>
   ```

   Example

   ```
   kubectl -n default delete ReplicationSource rs-pvc-01
   ```

3. 从 **Secondary** 集群同步最新数据
   建立 `Secondary-to-Primary` 一次性同步。
   在 **Primary** 集群上创建 `ReplicationDestination`，然后在 **Secondary** 集群上创建一次性 `ReplicationSource`
   参见 配置一次性同步
4. 删除 **ReplicationDestination** 和 **ReplicationSource**
   数据同步完成后，删除一次性资源
   在 **Secondary** 集群上删除 `ReplicationSource`

   Command

```
kubectl -n <namespace> delete ReplicationSource <rs-name>
```

Example

```
kubectl -n default delete ReplicationSource rs-pvc-01-latest
```

在 **Primary** 集群上删除 `ReplicationDestination`

Command

```
kubectl -n <namespace> delete ReplicationDestination <rd-name>
```

Example

```
kubectl -n default delete ReplicationDestination rd-pvc-01
```

5. 迁移应用

   在 **Secondary** 集群上缩减应用 Pod

   在 **Primary** 集群上扩容应用 Pod

6. 建立 **Primary-to-Secondary** 同步

   参见 配置定时同步

第三方存储能力注解指南

> 功能概述： 通过在 `kube-public` 命名空间中添加一个 **StorageDescription** 类型的 ConfigMap，平台会自动检测每个第三方 StorageClass 的快照支持情况以及支持的卷模式和访问模式（包括块存储特有的访问模式）。PVC 创建界面将仅显示有效选项，帮助您轻松选择和使用合适的存储功能。

目录

---

1. 入门指南

1.1 创建或更新 **ConfigMap**

> 重要提示： 请务必在 `kube-public` 命名空间中 执行以下操作，否则平台无法识别存储能力。

编辑或创建一个名称以 `sd-` 开头的 ConfigMap，例如 `sd-capabilities-enhanced`：

```
kubectl -n kube-public edit configmap sd-capabilities-enhanced
```

必需的标签

```
metadata:
  labels:
    features.alauda.io/type: StorageDescription
```

1.2 填充 `data` 字段

每个 `key` 对应一个 StorageClass 的 `provisioner`，其值是描述其能力的 YAML 字符串。主要字段说明：

字段	类型	说明
`snapshot`	`Boolean`	表示是否支持卷快照
`volumeMode`	`List[String]`	支持的卷模式；至少包含 `Filesystem` 或 `Block` 中的一个
`accessModes`	`List[String]`	当 `volumeMode` 为 `Filesystem` 时可用的访问模式
`blockAccessModes`	`List[String]`	针对 `Block` 卷特有的访问模式（可选）

> 如果省略 `blockAccessModes`，平台会对 Block 卷回退使用 `accessModes`。

1.3 应用配置

```
kubectl apply -f sd-capabilities-enhanced.yaml
```

应用后，UI 会自动调整可用选项，例如：

- 选择 **Block** 卷模式时，访问模式下拉框将显示 `blockAccessModes`。
- 如果 `snapshot: true`，则 PVC 页面会启用快照相关操作。

2. 示例 ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sd-capabilities-enhanced
  namespace: kube-public
  labels:
    features.alauda.io/type: StorageDescription
data:
  storage.advanced-block-fs.com: |-
    snapshot: true
    volumeMode:
      - Filesystem
      - Block
    accessModes:
      - ReadWriteOnce
      - ReadOnlyMany
    blockAccessModes:
      - ReadWriteOnce
  storage.filesystem-basic.com: |-
    snapshot: false
    volumeMode:
      - Filesystem
    accessModes:
      - ReadWriteOnce
      - ReadWriteMany
```

3. 更新现有能力描述

1. 找到需要修改的 `provisioner` 键。
2. 调整字段值以反映实际能力。

3. 使用 `kubectl apply -f ...` 重新应用 ConfigMap。平台会轮询更新并自动刷新 UI，您也可以刷新浏览器立即查看更改。

4. 与旧格式的兼容性

- 如果缺少 `blockAccessModes`，Block 卷将继承 `accessModes`。
- 无需删除旧的 ConfigMap，只需添加新字段即可平滑升级。

5. 常见问题解答

现象	可能原因	解决方案
Block 卷访问模式列表为空	`blockAccessModes` 和 `accessModes` 都为空	至少提供其中一个
UI 仍显示过时的能力	ConfigMap 未保存或浏览器缓存	使用 `kubectl get cm` 验证，刷新页面

故障排除

从 PVC 扩容失败中恢复

当 Kubernetes 中 PVC 扩容失败时，管理员可以手动恢复 Persistent Volume Claim (PVC) 状态并取消扩容请求。

目录

操作步骤

1. 将绑定到 PVC 的 Persistent Volume (PV) 的回收策略修改为 `Retain`。为此，编辑对应的 PV 并将 `persistentVolumeReclaimPolicy` 字段设置为 `Retain`。
2. 删除原有的 PVC。
3. 手动编辑 PV，移除其规格中的 `claimRef` 条目。这样可以确保新的 PVC 能绑定到该 PV，使 PV 状态变为 `Available`。
4. 重新创建一个较小容量或底层存储提供商支持的容量的新 PVC。
5. 在新 PVC 中显式指定 `volumeName` 字段，确保其与原 PV 名称匹配。这样可以确保新 PVC 准确绑定到指定的 PV。
6. 最后，恢复 PV 的原始回收策略。

其他提示

- 确保所使用的 `StorageClass` 已启用卷扩容，通过将 `allowVolumeExpansion` 设置为 `true` 实现。
- 操作时请谨慎，避免数据丢失风险。

在非优雅节点关闭后分离 CSI 卷

如果节点意外关闭，kubelet 可能不会将关闭事件报告给控制平面。在这种情况下，使用基于 CSI 的持久卷的 Pod 可能会一直停留在故障节点上，并且卷可能不会自动分离。在 ACP 中，你可以手动添加 `out-of-service` 污点，以触发 Pod 驱逐和卷分离。

目录

何时使用此操作步骤

当满足以下所有条件时，使用此操作步骤：

- 由于断电、操作系统故障或硬件故障，工作节点已停止。
- 节点未完成优雅关闭。
- 使用基于 CSI 的持久卷的工作负载无法在其他节点上重启，因为卷仍然连接到故障节点。

前提条件

- 你知道受影响节点的名称。
- 受影响节点已完全断电，或者已确认不可用。

> **WARNING**
>
> 不要将 `out-of-service` 污点应用到仍在运行或仍在恢复中的节点上。如果节点没有完全关闭，强制分离卷可能会导致文件系统损坏或应用层数据丢失。

操作步骤

1. 检查受影响节点的状态。

```
kubectl get node <node_name>
```

2. 如果节点仍然可以部分访问，请在继续之前先在基础设施层将其完全关闭。
3. 向节点添加 `out-of-service` 污点。

```
kubectl taint node <node_name> node.kubernetes.io/out-of-service=nodesh
utdown:NoExecute
```

应用污点后，Kubernetes 会开始将 Pod 从故障节点上驱逐。随后，attach-detach controller 可以分离 CSI 卷，以便替换 Pod 可以在其他节点上挂载相同的卷。

> **WARNING**
>
> `NoExecute` 效果可能会导致 Pod 在其他节点上被删除并重新创建。如果工作负载由 StatefulSet 管理，则只有在原始卷成功分离后，替换 Pod 才会启动。

验证污点

运行以下命令：

```
kubectl describe node <node_name>
```

验证该节点是否包含以下污点：

```
node.kubernetes.io/out-of-service=nodeshutdown:NoExecute
```

你还可以检查受影响的 Pod 是否正在重新调度，以及 CSI 卷是否已从故障节点分离。

节点恢复后移除污点

在节点修复并准备重新投入服务后，移除该污点：

```
kubectl taint node <node_name> node.kubernetes.io/out-of-service=nodeshut
down:NoExecute-
```

如果节点是被替换而不是恢复的，则可以在确认替换节点上的工作负载运行正常后，从集群中移除旧的 Node 对象。

对象存储

介绍

核心概念

安装

前提条件

安装 COSI

卸载

操作指南

为 Ceph RGW 创建 BucketClass

先决条件

步骤 1 – 准备 Ceph 集群

步骤 2 – 安装 COSI 插件

步骤 3 – 准备凭证 Secret

步骤 4 – 创建 BucketClass

验证与后续步骤

创建 Bucket Request

前提条件

操作步骤

相关内容

实用指南

使用 CephObjectStoreUser（Ceph Driver）控制 COSI Bucket 的访问权限和配额

前提条件

步骤 1 — 创建 CephObjectStoreUser（包含 capability 和配额）

步骤 2 — 定义绑定到 CephObjectStoreUser 的 BucketClass

步骤 3 — 使用 BucketClaim 创建 bucket

步骤 4 — 使用 BucketAccessClass/BucketAccess 发放最小权限凭证

步骤 5 — 匿名公开读取（可选）

步骤 6 — 配额控制：在哪里强制以及如何修改

运维与故障排查

清理

介绍

Container Object Storage Interface (COSI) 是一个 Kubernetes 原生框架，用于通过标准化的声明式 API 管理对象存储服务。COSI 扩展了 Kubernetes 存储模型，使工作负载能够通过 Kubernetes 风格的对象请求对象存储资源。

COSI 使管理员能够通过熟悉的 Kubernetes 风格 API 定义、配置和使用对象存储桶。它简化了应用程序与后端对象存储系统之间的集成，自动化了存储桶及其访问凭据的生命周期。借助 COSI，Kubernetes 用户可以动态请求对象存储资源，从而减少手动配置开销并提高运维效率。

通过采用 COSI，企业可以：

- 在多个云环境和本地环境中标准化对象存储配置。
- 通过声明式资源定义动态创建和管理存储桶。
- 通过 Kubernetes Secrets 将访问凭据无缝分发给工作负载。
- 将对象存储管理与 Kubernetes 持久化存储模式保持一致，以获得统一的使用体验。

目录

限制

- COSI 目前处于 alpha 阶段。
- 目前，COSI 仅支持 Ceph RGW 驱动。
- 与旧版对象存储存储桶的集成可能需要额外的手动配置。

核心概念

目录

概述

熟悉持久化存储概念的 Kubernetes 管理员可以使用 Container Object Storage Interface (COSI) 资源，通过声明式 Kubernetes API 管理对象存储。COSI 提供了一种用于管理对象存储的声明式机制，类似于现有的 Kubernetes 持久化存储管理方式。

COSI 使用三种主要资源：**BucketClass**、**Bucket** 和 **BucketClaim**。

核心资源

COSI 定义了三种基本资源：

1. BucketClass

范围： 集群级别
对应的 **Kubernetes** 概念： 类似于 StorageClass

BucketClass 由集群管理员创建，用于定义桶的特定类型或服务级别，包括地域位置、冗余策略和性能层级。

主要功能：

- 指定桶删除策略（例如，在 BucketClaim 删除时是否删除底层桶）
- 指定 COSI driver（driverName）
- 定义供应商特定参数

YAML 示例：

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
driverName: ceph.objectstorage.k8s.io
parameters:
  objectStoreUserSecretName: rook-ceph-object-user-object-store-user-for-cosi
  objectStoreUserSecretNamespace: rook-ceph
```

2. Bucket

范围： 集群级别
对应的 **Kubernetes** 概念： 类似于 PersistentVolume (PV)

Bucket 表示外部对象存储系统中实际桶的 Kubernetes 抽象。

生命周期管理：

- 动态创建：当收到 BucketClaim 请求时，由 COSI controller 自动创建。

3. BucketClaim

范围： 命名空间级别
对应的 **Kubernetes** 概念： 类似于 PersistentVolumeClaim (PVC)

应用开发者在各自的命名空间中创建 BucketClaim 资源，以请求对象存储桶。

工作流程：

1. 用户创建一个指定 BucketClass 的 BucketClaim。
2. COSI controller 检测到该请求，并根据 BucketClass 定义，在对象存储后端动态创建桶。
3. 创建对应的 Bucket 资源，并将其绑定到 BucketClaim。
4. 生成一个包含桶访问凭据的 Secret，并自动挂载到请求该桶的 Pods 中。

YAML 示例：

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-app-bucket-claim
  namespace: my-app-ns
spec:
  bucketClassName: ceph-standard-replicated
  protocol:
    s3: {} # Defaults populated by the driver
```

资源交互工作流

以下流程展示了 COSI 资源在实践中的动态创建过程：

1. 集群管理员 创建并维护 BucketClass。
2. 命名空间用户 创建引用该 BucketClass 的 BucketClaim。
3. **COSI controller** 观察到 BucketClaim，并根据 BucketClass 定义动态创建桶。
4. controller 在 Kubernetes 中生成对应的 Bucket 资源。
5. BucketClaim 和 Bucket 相互绑定。

6. 创建一个包含存储凭据的 Secret 供 Pod 使用。

7. Pods 挂载该 Secret 并访问对象存储。

总结

COSI Resource	范围	**Kubernetes** 类比	目的
BucketClass	集群级别	StorageClass	定义桶类型和策略
Bucket	集群级别	PersistentVolume (PV)	实际桶的 Kubernetes 抽象
BucketClaim	命名空间级别	PersistentVolumeClaim (PVC)	用户对桶资源的请求

通过利用 COSI 提供的标准化 API，Kubernetes 管理员可以以声明式和可移植的方式管理对象存储资源，显著提升 Kubernetes 集群中应用与对象存储之间的集成效率。

安装

目录

前提条件

1. 下载与您的平台架构对应的 灵雀云容器平台 **COSI** 集群插件软件包。
2. 下载 灵雀云容器平台 **COSI for Ceph** 集群插件软件包。
3. 使用 上架软件包 功能上传已下载的插件软件包。
4. 使用 集群插件 安装机制将插件软件包安装到目标集群。

> **INFO**
>
> 上架软件包： 平台管理 > **Marketplace** > 上架软件包 页面。 单击右侧的 帮助文档，了解如何将集群插件发布到目标集群。有关 CLI 指南，请参见 CLI。

安装 灵雀云容器平台 **COSI**

约束和限制

- 插件仅作用于当前集群。您必须在希望启用 COSI 功能的每个集群上单独安装所需插件。
- 灵雀云容器平台 **COSI for Ceph** 插件依赖于 灵雀云容器平台 **COSI** 插件。请确保先安装 灵雀云容器平台 **COSI** 插件。

操作步骤

1. 登录平台并进入 平台管理 页面。
2. 转到 **Marketplace** > 集群插件，查看可用的集群插件列表。
3. 选择要安装插件的目标集群。
4. 找到 灵雀云容器平台 **COSI** 插件，并在⋮菜单中选择 安装 以开始安装。
5. 找到并安装 灵雀云容器平台 **COSI for Ceph** 插件。
6. 等待所有插件的状态变为 已安装。

卸载

1. 登录平台并进入 平台管理 页面。
2. 转到 **Marketplace** > 集群插件，查看已安装的集群插件列表。
3. 选择要移除插件的目标集群。
4. 找到要卸载的插件，并在⋮菜单中选择 卸载 以开始卸载过程。
5. 等待插件状态变为 就绪，表示如有需要可重新安装。

重要： 在卸载 灵雀云容器平台 **COSI** 插件之前，必须先卸载 灵雀云容器平台 **COSI for Ceph** 插件。

操作指南

为 Ceph RGW 创建 BucketClass

Ceph Object Storage 可以通过 **Container Object Storage Interface (COSI)** 暴露给 Kubernetes 工作负载，为大数据分析、备份与恢复以及机器学习场景提供对象存储。在用户可以预配 bucket 之前，需要先创建 **BucketClass**。

BucketClass 是一种模板资源，用于指定存储 driver、认证 secret 以及删除策略，这些配置将应用于基于该 BucketClass 创建的每个 bucket。

目录

先决条件

要求	说明
已运行且启用 RGW (S3) 的 Ceph 集群	可以是内部（由 Rook 管理）或外部集群。
灵雀云容器平台 COSI 插件	必须同时安装 灵雀云容器平台 **COSI** 和 灵雀云容器平台 **COSI for Ceph**。
包含 Ceph RGW 凭证的 Kubernetes Secret	在下面的 步骤 **3** 中准备。

步骤 1 – 准备 Ceph 集群

从以下选项中选择 一个：

选项	说明
内部 **Ceph**	由 Rook Operator 在平台 内部 部署和管理的 Ceph 集群。详细信息请参见 *创建存储服务*。
外部 **Ceph**	可从平台网络访问的独立 Ceph 集群。

步骤 2 – 安装 COSI 插件

安装以下集群插件：

1. 灵雀云容器平台 **COSI**
2. 灵雀云容器平台 **COSI for Ceph**

> 有关确切命令，请参见 *安装*。

步骤 3 – 准备凭证 Secret

COSI 会从 Kubernetes **Secret** 中检索 RGW 凭证。请根据您的 Ceph 部署方式选择 一种 方法。

方法 A – 自动生成（Rook 管理的 Ceph）

1. 在 *rook-ceph* 命名空间中创建一个 **CephObjectStoreUser**：

```
# ceph-object-store-user.yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  store: object-store               # name of your CephObjectStore
  capabilities:
    bucket: ["read", "write"]
    user:   ["read", "write"]
```

2. 应用该清单：

```
kubectl apply -f ceph-object-store-user.yaml
```

3. 获取自动生成的 Secret 名称（后续会用到）：

```
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph \
  -o jsonpath='{.status.info.secretName}'
```

方法 B – 手动（外部 Ceph）

1. 获取 **AccessKey**、**SecretKey** 和 **RGW Endpoint**。
2. 在目标项目/命名空间中创建一个 Secret，并为其添加标签，以便 UI 可以发现它：

```
kubectl create secret generic ceph-external-creds -n <YOUR_NAMESPACE> \
  --from-literal=AccessKey=<YOUR_ACCESS_KEY> \
  --from-literal=SecretKey=<YOUR_SECRET_KEY> \
  --from-literal=Endpoint=http://<YOUR_RGW_ENDPOINT>

kubectl label secret ceph-external-creds -n <YOUR_NAMESPACE> app=rook-ceph-rgw
```

> 重要： 标签 `app=rook-ceph-rgw` 是平台 UI 列出该 Secret 的必需条件。

步骤 4 – 创建 BucketClass

选项 1 – UI 流程

1. 导航到 **Storage** → **Object StorageClass**，然后单击 **Create Object StorageClass**。
2. 选择 **Ceph Object Storage** 作为 driver。
3. 配置以下字段：
   - **Deletion Policy** – 当其 BucketClaim 被删除时，如何处理底层 bucket（默认值：`Delete`）。
   - **Secret** – 选择在 *步骤 3* 中准备的 Secret（仅显示带有 `app=rook-ceph-rgw` 标签的 Secret）。
   - **Allocate Projects** – *(可选)* 限制仅供特定项目使用。
4. 单击 **Create**。

选项 2 – YAML（适合 GitOps）

使用正确的 Secret 引用创建 `ceph-bucketclass.yaml`：

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver
  labels:
    project.cpaas.io/ALL_ALL: "true"
driverName: ceph.objectstorage.k8s.io
deletionPolicy: Delete
parameters:
  objectStoreUserSecretName: <your-secret-name>
  objectStoreUserSecretNamespace: <your-secret-namespace>
```

应用该清单：

```bash
kubectl apply -f ceph-bucketclass.yaml
```

验证与后续步骤

验证 BucketClass：

```bash
kubectl get bucketclass
```

当 BucketClass 准备就绪后，您可以创建引用它的 **Bucket** 或 **BucketClaim** 资源，从而为您的应用程序预配兼容 S3 的对象存储。

创建 Bucket Request

使用 **Bucket Request** 可基于 **Object Storage Class** 动态创建存储桶，并自动将两者绑定。

目录

前提条件

- 安装 灵雀云容器平台 **COSI** 集群插件。
- 安装 灵雀云容器平台 **COSI for Ceph** 集群插件包。

有关详细的安装步骤，请参见 安装。

操作步骤

1. 切换到 **Container Platform** 视图。
2. 在左侧导航中，单击 存储 **> Bucket Claims**。
3. 单击 创建 **Bucket Request**。

4. 按如下所示配置参数。

参数	描述
名称	Bucket request 的名称。
Object Storage Class	用于动态创建存储桶并建立绑定的 Object Storage Class。

5. 单击 创建。等待状态变为 `Available`，这表示请求已完成并且绑定已完成。

相关内容

在 bucket request 详情页，按需单击右上角的 操作，然后选择 删除 **bucket policy**。警告：删除 bucket policy 会清除存储桶中的所有数据。删除前请谨慎操作，并确认数据备份和安全要求。

实用指南

使用 CephObjectStoreUser（Ceph Driver）控制 COSI Bucket 的访问权限和配额

Kubernetes 管理员可以结合 **CephObjectStoreUser**（**COSU**）、**BucketClass/BucketClaim** 以及 **BucketAccessClass/BucketAccess**，为由 Ceph RGW 支持的 COSI bucket 实现 最小权限访问 和 配额强制执行。

> 你将构建的内容
>
> 1. 一个带有显式 capability 和可选按用户配额的 CephObjectStoreUser；
> 2. 一个 BucketClass，用于告诉 Ceph COSI driver 需要使用哪个 COSU 凭证；
> 3. 一个或多个 BucketClaim，用于创建 bucket；
> 4. 使用 BucketAccessClass/BucketAccess 为不同 workload 提供细粒度凭证（只读、只写、读写），并支持可选的匿名读取。

目录

前提条件

- 一个运行正常的 Ceph 集群，已安装 RGW 和 Rook。
- 已安装 COSI 插件。
- 集群管理员权限（用于创建集群作用域资源）。

步骤 1 — 创建 **CephObjectStoreUser**（包含 **capability** 和配额）

`CephObjectStoreUser` 是 driver 用于在 RGW 中执行 bucket 操作的 service account。它必须位于 **`rook-ceph`** namespace 中，并且目标指向你的 `CephObjectStore`。

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  # Target CephObjectStore
  store: object-store
  # Required capabilities for COSI bucket lifecycle
  capabilities:
    bucket: read, write
    user: read, write
  # Optional per-user quotas enforced by RGW
  quotas:
    maxBuckets: 50         # limit number of buckets this user can own
    maxObjects: 500          # total objects across buckets (example)
    maxSize: 100Gi        # total logical size across buckets
  displayName: "User for COSI driver"
```

获取生成的访问密钥（Rook 会为该用户创建一个 Secret）：

```bash
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph -o yaml
# Read status.info.secretName -> the Secret holding AccessKey/SecretKey
```

> 这些 COSU 凭证由 **driver**（而不是你的应用）使用，用于代表你创建/删除 bucket。

步骤 2 — 定义绑定到 CephObjectStoreUser 的 BucketClass

`BucketClass` 会指示 driver 在创建 bucket 时使用哪个 COSU Secret。

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
# Must match the driver name
driverName: ceph.objectstorage.k8s.io
parameters:
  # Reference the Secret created for the CephObjectStoreUser
  objectStoreUserSecretName: <secret-name-from-step-1>
  objectStoreUserSecretNamespace: rook-ceph
```

> `deletionPolicy` 控制在删除 `BucketClaim` 时 bucket 的物理生命周期（`Delete` vs `Retain`）。

步骤 3 — 使用 BucketClaim 创建 bucket

通过引用 `BucketClass`，在你的应用 namespace 中创建 bucket。

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-bucket-claim
  namespace: app-a
spec:
  bucketClassName: ceph-cosi-driver-class
  # Required by the COSI API (choose the data API you need)
  protocols:
  - S3
```

等待 `status.bucketReady: true`，并记录 `status.bucketName`，它表示实际的 RGW bucket 名称。

步骤 4 — 使用 BucketAccessClass/BucketAccess 发放最小权限凭证

使用 `BucketAccessClass` 定义策略模板，并通过 `BucketAccess` 为每个 workload 签发凭证。支持的策略包括：**`readonly`**、**`writeonly`**、**`readwrite`**。通过设置 `parameters.anonymous: "true"`（字符串）即可启用匿名读取。

示例 **`BucketAccessClass`**（只读）

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccessClass
metadata:
  name: ceph-readonly-access-class
authenticationType: KEY
driverName: ceph.objectstorage.k8s.io
parameters:
  policy: readonly
  anonymous: "false"
```

使用 **`BucketAccess`** 签发凭证

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccess
metadata:
  name: my-bucket-readonly-access
  namespace: app-a
spec:
  bucketAccessClassName: ceph-readonly-access-class
  bucketClaimName: my-bucket-claim
  credentialsSecretName: my-bucket-readonly-credentials
```

driver 会写入一个在 `credentialsSecretName` 中指定名称的 Secret。对 `.data.BucketInfo` 进行解码（base64）后，即可获得 S3 client 所需的 `secretS3.endpoint`、`accessKeyID` 和 `accessSecretKey`。

> 提示：为每个 Deployment/Job 签发不同的凭证，便于在不影响其他 workload 的情况下进行轮换和吊销。

步骤 5 — 匿名公开读取（可选）

如果你需要公开静态资源托管：

```
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketAccessClass
metadata:
  name: ceph-anonymous-readonly-class
authenticationType: KEY
driverName: ceph.objectstorage.k8s.io
parameters:
  policy: readonly
  anonymous: "true"
```

将它通过 `BucketAccess` 绑定到你的 `BucketClaim`。一旦授予成功，对象即可通过未认证的 HTTP `GET` 访问（请确保网络暴露方式符合要求）。

步骤 6 — 配额控制：在哪里强制以及如何修改

作用范围： `CephObjectStoreUser` 上的 `quotas` 区块按 用户级别 生效，并由 RGW 对该用户拥有的所有 bucket 统一执行。

- `maxBuckets`：用户可创建/拥有的 bucket 数上限。
- `maxObjects`：用户可存储的对象总数上限（跨 bucket）。
- `maxSize`：允许该用户使用的总逻辑大小。

更新配额 通过编辑 COSU 资源完成：

```
kubectl -n rook-ceph edit cephobjectstoreuser user-for-cosi
# Modify spec.quotas.{maxBuckets,maxObjects,maxSize}; save and exit.
# Rook reconciles and applies the new RGW user quotas.
```

> 设计选择： 保持 **COSU** 配额 相对严格，以限制影响范围。使用 最小权限策略 通过 BAC/BA 限制某一组应用凭证在 **bucket** 内 可以执行的操作。

运维与故障排查

- **Namespace** 放置：`CephObjectStoreUser` 及其 Secret 必须位于 `rook-ceph`。应用级资源（`BucketClaim`、`BucketAccess`）应位于应用 namespace 中。
- 策略未生效：确认 BAC 中的 `bucketAccessClassName` 和 `parameters.policy`（`readonly|writeonly|readwrite`）。
- 匿名读取失败：确保 `anonymous: "true"` 是 字符串，不是布尔值；检查 endpoint 暴露以及 HTTP 访问路径（`/<bucket>/<object>`）。
- 找不到密钥：检查 `BucketAccess` Secret，并解码 `.data.BucketInfo`。
- **Bucket** 一直未就绪：检查 controller/driver 日志（例如 `kubectl -n cpaas-system logs deploy/ceph-cosi-driver -c ceph-cosi-driver`）。
- 轮换：创建新的 `BucketAccess`，将 workload 切换到新的 Secret，然后删除旧的 Secret/BA。

清理

- 通过删除其 `BucketAccess` 以及引用的 Secret，移除某个 workload 的凭证。
- 删除 bucket：删除 `BucketClaim`（行为取决于 `BucketClass.deletionPolicy`）。如果后端因为 bucket 非空而拒绝删除，请先删除对象。