存储

介绍

介绍

核心概念

核心概念

Persistent Volume (PV)

Persistent Volume Claim (PVC)

通用临时卷（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 Volume Snapshot 组件

通过 Web 控制台部署

通过 YAML 部署

创建 PV

前提条件

示例 PersistentV

通过 Web 控制台

Creating PVCs

前提条件

PersistentVolumeClaim 示例：

通过 Web 控制台创建 Persistent Volume C

通过 CLI 创建 Persistent Volume Claim

操作

通过 Web 控制台扩容 PersistentVolumeCl

通过 CLI 扩容 Persistent Volume Claim 存

其他资源

使用卷快照

前提条件

卷快照自定义资源（CR）示例

通过 Web 控制台创建卷快照

通过 CLI 创建卷快照

通过卷快照创建持久卷声明

其他资源

实用指南

通用临时卷

- 临时卷示例
- 主要特性
- 何时使用通用临时卷
- 与 emptyDir 的区别

使用 emptyDir

- emptyDir 示例
- 可选的 Medium 设置
- 主要特性
- 常见用例

使用本地卷配

- 前提条件
- 操作步骤
- 自动发现本地存

配置持久卷声明的灾难恢复

- 概述
- 术语
- 前提条件
- 部署 Alauda Build of VolSync
- 配置定时同步
- 配置一次性同步
- 启用应用 PVC 的灾难恢复
- 计划迁移
- 故障切换
- 故障恢复后切换回主集群（Failback）

第三方存储能力注解指南

1. 入门指南
2. 示例 ConfigMap
3. 更新现有能力描述
4. 与旧格式的兼容性
5. 常见问题解答

故障排除

从 PVC 扩容失败中恢复

- 操作步骤
- 额外提示

对象存储

介绍

限制

核心概念

Overview

Core Resources

安装

前提条件

安装 Alauda Co

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**。它们定义了集群内存储的请求、分配和配置方式。在底层，**CSI**（Container Storage Interface）驱动通常负责实际的存储供应和挂载。下面我们简要介绍每个组件，并重点说明 CSI 驱动的作用。

目录

Persistent Volume (PV)

PersistentVolume (PV) 是集群中已被预配的一块存储（可以由管理员静态预配，也可以通过 StorageClass 动态预配）。它代表底层存储资源，例如云服务商的磁盘或网络附加文件系统，并作为集群中的一种资源存在，类似于节点。

Persistent Volume Claim (PVC)

PersistentVolumeClaim (PVC) 是对存储的请求。用户定义所需的存储容量和访问模式（例如读写权限）。如果有合适的 PV 可用，或者可以通过 StorageClass 动态预配，PVC 就会与该 PV “绑定”。绑定后，Pod 可以引用 PVC 来持久化或共享数据。

通用临时卷（Generic Ephemeral Volumes）

Kubernetes 引入的通用临时卷功能允许在 Pod 生命周期内使用由 CSI 驱动的 `temporary` 卷，这类似于 `emptyDir`，但功能更强大，支持挂载任何类型的 CSI 卷（支持快照、扩展等功能）。

更多用法请参考 Generic ephemeral volumes

emptyDir

1. emptyDir 是一种临时存储卷，类型为空目录。
2. 它在 Pod 被调度到节点时创建，存储位于该节点的本地文件系统（默认是节点磁盘）。
3. 当 Pod 被删除时，emptyDir 中的数据也会被清除。

更多用法请参考 Using an emptyDir

hostPath

在 Kubernetes 中，hostPath 卷是一种特殊类型的卷，它将宿主节点文件系统中的文件或目录直接映射到 Pod 的容器中。

- 允许 Pod 访问宿主节点上的文件或目录。
- 适用于：
  - 访问宿主级资源
  - 调试
  - 使用节点上已有的数据

ConfigMap

Kubernetes 中的 ConfigMap 是一种 API 对象，用于以键值对形式存储非敏感的配置信息。它允许将配置与应用代码解耦，使应用更具可移植性且易于管理。

Secret

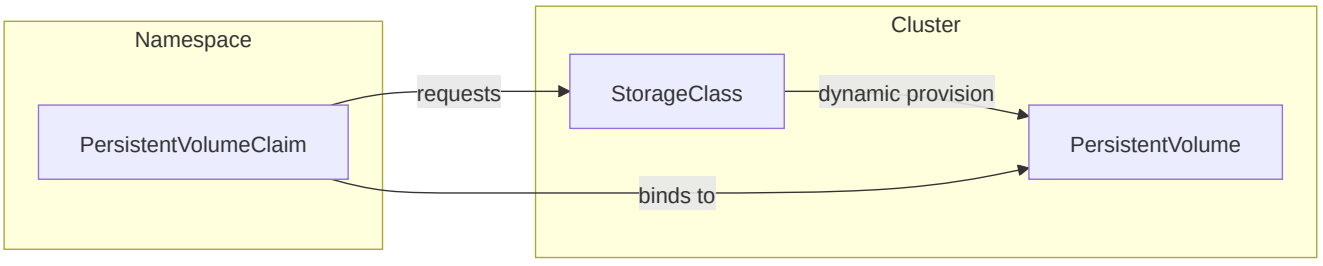
在 Kubernetes 中，Secret 是一种 API 对象，用于存储敏感数据，如：

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

Secret 通过避免将敏感数据直接存储在 Pod 规格或容器镜像中，帮助保护这些数据。

StorageClass

StorageClass 描述了卷应如何被动态预配。它映射到特定的预配器（通常是 CSI 驱动），并可包含存储层级、性能特征或其他后端配置参数。通过创建多个 StorageClass，可以为开发者提供多种类型的存储选择。



图示：PVC、PV 和 StorageClass 之间的关系。

Container Storage Interface (CSI)

Container Storage Interface (CSI) 是 Kubernetes 用于集成存储驱动的标准 API。它允许第三方存储提供商构建外部插件，即可以在不修改 Kubernetes 本身的情况下安装或更新存储驱动。

一个 CSI 驱动 通常包含两个组件：

1. 控制器组件：运行在集群中（通常以 Deployment 形式），负责高级操作，如 创建 或 删除卷。对于网络存储，它还可能负责将卷附加和卸载到节点。
2. 节点组件：运行在每个节点上（通常以 DaemonSet 形式），负责在该节点上 挂载 和 卸载卷。它与 kubelet 通信，确保卷对 Pod 可用。

当用户创建引用使用 CSI 驱动的 StorageClass 的 PVC 时，CSI 驱动会监测该请求并相应地进行存储预配（如果需要动态预配）。存储创建完成后，驱动通知 Kubernetes，Kubernetes 创建对应的 PV 并将其绑定到 PVC。每当 Pod 使用该 PVC 时，驱动的节点组件负责挂载卷，使存储在容器内可用。

通过利用 **PV**、**PVC**、**StorageClass** 和 **CSI**，Kubernetes 实现了强大且声明式的存储管理方式。管理员可以定义一个或多个 StorageClass 来代表不同的存储后端或性能层级，而开发者只需通过 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

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 Volume Snapshot 组件

通过 Web 控制台部署

通过 YAML 部署

创建 PV

前提条件

示例 PersistentV

通过 Web 控制台

Creating PVCs

前提条件

PersistentVolumeClaim 示例：

通过 Web 控制台创建 Persistent Volume C

通过 CLI 创建 Persistent Volume Claim

操作

通过 Web 控制台扩容 PersistentVolumeCl

通过 CLI 扩容 Persistent Volume Claim 存

其他资源

使用卷快照

前提条件

卷快照自定义资源（CR）示例

通过 Web 控制台创建卷快照

通过 CLI 创建卷快照

通过卷快照创建持久卷声明

其他资源

创建 CephFS 文件存储类型存储类

CephFS 文件存储是内置的 Ceph 文件存储系统，为平台提供基于 Container Storage Interface (CSI) 的存储访问方式，提供安全、可靠且可扩展的共享文件存储服务，适用于文件共享、数据备份等场景。操作前，需先创建 CephFS 文件存储类。

在持久卷声明（PVC）中绑定存储类后，平台将根据持久卷声明在节点上动态创建持久卷供业务应用使用。

目录

部署卷插件

点击 部署 后，在 分布式存储 页面，进行创建存储服务或接入存储服务。

创建存储类

1. 进入 管理员。
2. 在左侧导航栏点击 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示例展示，也可选择使用 YAML 创建。

4. 选择 **CephFS** 文件存储，点击 下一步。
5. 按照以下说明配置相关参数。

参数	说明
回收策略	持久卷的回收策略。 - Delete：当持久卷声明被删除时，绑定的持久卷也会被删除。 - Retain：即使持久卷声明被删除，绑定的持久卷仍然保留。
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。 - ReadWriteOnce (RWO)：可被单个节点以读写方式挂载。 - ReadWriteMany (RWX)：可被多个节点以读写方式挂载。
分配项目	请分配可以使用此类型存储的项目。 如果当前没有需要使用此类型存储的项目，可暂不分配，后续再更新。

提示：以下参数需在分布式存储中设置，并会直接应用于此处。

- 存储集群：当前集群内置的 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，能够提供高 IOPS 和低延迟的存储服务，适用于数据库、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，平台将基于 Persistent Volume Claim 动态创建 Persistent Volume 供业务应用使用。

目录

部署卷插件

点击 部署 后，在 分布式存储 页面，进行创建存储服务或接入存储服务。

创建存储类

1. 进入 管理员。
2. 在左侧导航栏点击 存储管理 > 存储类。
3. 点击 创建存储类。
   注意：以下内容为表单形式示例，也可以选择 YAML 完成操作。

4. 选择 **CephRBD** 块存储，点击 下一步。

5. 按需配置参数。

参数	描述
文件系统	默认为 **EXT4**，Linux 的日志文件系统，能够提供范围文件存储并处理大文件。文件系统容量可达 1 EiB，支持的文件大小最高可达 16 TiB。
回收策略	持久卷的回收策略。 - Delete：绑定的持久卷会随着持久卷声明一起被删除。 - Retain：即使持久卷声明被删除，绑定的持久卷仍会被保留。
访问模式	仅支持 ReadWriteOnce (RWO)：可被单个节点以读写模式挂载。
分配项目	请分配可使用该类型存储的项目。 如果当前没有需要该类型存储的项目，可以选择不分配，后续再更新。

提示：以下参数需在分布式存储中设置，并会直接应用到此处。

- 存储集群：当前集群内置的 Ceph 存储集群。
- 存储池：存储集群中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、中间件等场景。使用前需要先创建 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 动态在节点上创建持久卷供业务应用使用。

目录

背景信息

使用优势

- 相较于远程存储（如 **NFS** 共享存储）：TopoLVM 类型存储位于节点本地，提供更优的 IOPS 和吞吐性能，以及更低的延迟。

- 相较于 hostPath（如 **local-path**）：虽然两者均为节点本地存储，但 TopoLVM 支持灵活调度容器组到资源充足的节点，避免因资源不足导致容器组无法启动的问题。
- TopoLVM 默认支持自动扩容。修改 Persistent Volume Claim 中的存储配额后，无需重启容器组即可自动完成扩容。

使用场景

- 仅需临时存储时，如开发调试。
- 存储 I/O 需求较高时，如实时索引。

约束与限制

请尽量仅在应用层能实现数据复制和备份的场景下使用本地存储，如 MySQL，避免因本地存储缺乏数据持久性保障而导致数据丢失。

了解更多 ↗

部署卷插件

点击部署后，在新打开的页面进行配置本地存储。

创建存储类

1. 进入 管理员。
2. 在左侧导航栏点击 存储管理 > 存储类。
3. 点击 创建存储类。
4. 选择 块存储。
5. 选择 **TopoLVM**，然后点击 下一步。
6. 按照下述说明配置存储类参数。
   注意：以下内容以表单示例形式展示，也可选择使用 YAML 创建。

参数	说明
名称	存储类名称，在当前集群内必须唯一。
显示名称	用于帮助识别或筛选的名称，如存储类的中文描述。
设备类	设备类是 TopoLVM 中对存储设备的分类方式，每个设备类对应一组具有相似特性的存储设备。如无特殊需求，使用 自动分配 设备类。
文件系统	• **XFS** 是高性能的日志文件系统，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 下的日志文件系统，提供 extent 文件存储，支持大文件处理，最大文件系统容量为 1 EiB，最大文件大小为 16 TiB。
回收策略	持久卷的回收策略。 • Delete：绑定的持久卷会随着 PVC 一起被删除。 • Retain：即使 PVC 被删除，绑定的持久卷仍然保留。
访问模式	ReadWriteOnce (RWO)：可被单个节点以读写方式挂载。
PVC 重建	支持跨节点的 PVC 重建。启用时需配置 重建等待时间。当使用该存储类创建的 PVC 所在节点发生故障时，等待时间后 PVC 会自动在其他节点重建以保证业务连续性。 注意： • 重建的 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，否则会影响 PVC 重建。
分配项目	该类型的持久卷声明只能在特定项目中创建。 若当前未分配项目，也可后续更新。

7. 确认配置信息无误后，点击 创建 按钮。

后续操作

准备就绪后，可通知开发人员使用 TopoLVM 功能。例如，在容器平台的 存储 > 持久卷声明 页面创建 Persistent Volume Claim 并绑定到 TopoLVM 存储类。

创建 NFS 共享存储类

基于社区 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 共享存储采用社区 NFS CSI（Container Storage Interface）存储插件，更符合 Kubernetes 设计理念，允许客户端访问多个服务器。

目录

前提条件

- 必须配置好 NFS 服务器，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`。您可以在服务器端执行 `nfsstat -s` 来查看版本信息。

部署 Alauda Container Platform NFS CSI 插件

通过 Web 控制台部署

1. 进入 管理员。
2. 在左侧导航栏点击 **Storage** > **StorageClasses**。
3. 点击 **Create StorageClass**。
4. 在 **NFS CSI** 右侧点击部署，跳转到 **Plugins** 页面。
5. 在 `Alauda Container Platform NFS CSI` 插件右侧，点击⋮> **Install**。
6. 等待部署状态显示 **Deployment Successful** 后完成部署。

通过 YAML 部署

参考 Installing via YAML

`Alauda Container Platform NFS CSI` 是一个 非配置插件，模块名为 `nfs`

创建 NFS 共享存储类

1. 点击 **Create Storage Class**。
   注意：以下内容以表单形式展示，您也可以选择通过 YAML 完成操作。
2. 选择 **NFS CSI** 并点击 **Next**。
3. 按照以下说明配置相关参数。

参数	说明
Name	存储类名称，必须在当前集群内唯一。
Service Address	NFS 服务器的访问地址。例如：`192.168.2.11`。
Path	NFS 文件系统在服务器节点上的挂载路径。例如：`/nfs/data`。
NFS Protocol Version	当前支持三种版本：`v3`、`v4.0` 和 `v4.1`。

参数	说明
Reclaim Policy	持久卷的回收策略。 - Delete：当持久卷声明被删除时，绑定的持久卷也会被删除。 - Retain：即使持久卷声明被删除，绑定的持久卷仍然保留。
Access Modes	当前存储支持的所有访问模式。在后续声明持久卷时，只能选择其中一种模式来挂载持久卷。 - ReadWriteOnce (RWO)：单个节点可读写挂载。 - ReadWriteMany (RWX)：多个节点可读写挂载。 - ReadOnlyMany (ROX)：多个节点只读挂载。
Allocated Projects	请分配可以使用此类存储的项目。 如果当前没有项目需要此类存储，可以暂时不分配，后续再更新。
subDir	使用 NFS 共享存储类创建的每个 PersistentVolumeClaim (PVC) 对应 NFS 共享中的一个子目录。默认子目录命名规则为 `${pv.metadata.name}`（即 PersistentVolume 名称）。如果默认生成的名称不符合需求，可以自定义子目录命名规则。

NOTE

`subDir` 字段仅支持以下三种变量，NFS CSI Driver 会自动解析：

- `${pvc.metadata.namespace}`：PVC 命名空间。
- `${pvc.metadata.name}`：PVC 名称。
- `${pv.metadata.name}`：PV 名称。

`subDir` 命名规则 必须 保证子目录名称唯一，否则多个 PVC 可能共用同一子目录，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}`

该配置适用于多个 Kubernetes 集群共享同一 NFS 服务器，通过在子目录命名规则中加入集群标识（如集群名）实现集群间的区分。

不推荐配置：

- `${pvc.metadata.namespace}-${pvc.metadata.name}-${pv.metadata.name}` 避免使用 - 作为分隔符，可能导致子目录名称歧义。例如：两个 PVC 分别命名为 `ns-1/test` 和 `ns/1-test`，都可能生成相同的子目录 `ns-1-test`。
- `${pvc.metadata.namespace}/${pvc.metadata.name}/${pv.metadata.name}` 不要配置 subDir 生成嵌套目录。NFS CSI Driver 删除 PVC 时只会删除最后一级目录 `${pv.metadata.name}`，会在 NFS 服务器上遗留父目录。

4. 确认配置信息无误后，点击 **Create**。

部署 Volume Snapshot 组件

Volume snapshot 指的是持久卷的快照，是持久卷在某一特定时间点的副本。如果集群使用支持快照功能的持久卷，则可以部署 volume snapshot 组件以启用该功能。

目前，平台仅支持为通过存储类动态创建的 PVC 创建 volume snapshot。您可以基于这些快照创建新的 PVC 绑定。

提示：通过快照创建 PVC 时支持的访问模式与通过存储类创建 PVC 时支持的访问模式有所不同，以下表格中以加粗标示。

用于创建 **Volume Snapshot** 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

目录

通过 Web 控制台部署

1. 进入 管理员。
2. 点击 **Marketplace** > 集群插件，进入 集群插件 列表页面。
3. 找到 `Alauda Container Platform Snapshot Management` 集群插件，点击 安装，稍等片刻直到部署成功。

通过 YAML 部署

参考 通过 YAML 安装

`Alauda Container Platform Snapshot Management` 是一个非配置插件，模块名称为 `snapshot`

创建 PV

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

- **HostPath**：将容器所在主机的文件目录挂载到容器内指定路径（对应 Kubernetes 的 HostPath），允许容器使用主机的文件系统进行持久化存储。如果主机不可访问，则 HostPath 可能无法访问。
- **NFS Shared Storage**：NFS 共享存储使用社区 NFS CSI（Container Storage Interface）存储插件，更符合 Kubernetes 设计理念，支持多服务客户端访问能力。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有能力提供相应的存储容量。
- 获取后端存储访问地址、要挂载的文件路径、凭证访问（如需）及其他相关信息。

示例 PersistentVolume

```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) PVC 删除后如何处理（Retain、Delete、Recycle）。

(4) StorageClass 名称（用于动态绑定）。

(5) 存储后端类型。

通过 Web 控制台创建 PV

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Storage Management** > **Persistent Volumes (PV)**。
3. 点击 **Create Persistent Volume**。
4. 参考以下说明配置参数后，点击 **Create**。

存储信息

类型	参数	说明
HostPath	Path	支持存储卷的节点上文件目录的路径。例如：`/etc/kubernetes`。
NFS Shared Storage	Server Address	NFS 服务器的访问地址。
	Path	NFS 文件系统在服务器节点上的挂载路径，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`。可在服务器端执行 `nfsstat -s` 查看版本信息。

通过 **CLI** 创建 **PV**

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

访问模式

持久卷的访问模式由后端存储设置的相关参数决定。

访问模式	含义
ReadWriteOnce (RWO)	只能被单个节点以读写方式挂载。
ReadWriteMany (RWX)	可以被多个节点以读写方式挂载。
ReadOnlyMany (ROX)	可以被多个节点以只读方式挂载。

回收策略

回收策略	含义
Delete	删除持久卷声明的同时，删除绑定的持久卷以及后端存储卷资源。 注意：NFS Shared Storage 类型的 PV 不支持 **Delete** 回收策略。
Retain	即使持久卷声明被删除，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⋮或详情页右上角的 **Operations**，根据需要更新或删除持久卷。

删除持久卷适用于以下两种场景：

- 删除未绑定的持久卷：未被写入且不再需要写入，删除后释放对应存储空间。
- 删除 **Retain** 状态的持久卷：持久卷声明已删除，但因 Retain 回收策略未同步删除。如果持久卷中的数据已备份到其他存储或不再需要，删除该持久卷也可释放对应存储空间。

其他资源

- 创建 PVC

Creating PVCs

创建 PersistentVolumeClaim（PVC），并根据需要设置请求的 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

确保命名空间中有足够剩余的存储配额，以满足此次创建操作所需的存储大小。

PersistentVolumeClaim 示例：

```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. 进入 **Container Platform**。
2. 在左侧边栏点击 **Storage > PersistentVolumeClaims (PVC)**。
3. 点击 **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	说明
Name	PersistentVolumeClaim 的名称，必须在当前命名空间内唯一。
Creation Method	- 动态创建：根据存储类动态生成 PersistentVolume 并进行绑定。 - 静态绑定：根据配置参数匹配并绑定已有的 PersistentVolume。
Storage Class	选择动态创建方式后，平台将根据所选存储类描述动态创建 PersistentVolume。

参数	说明
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：建议结合计划绑定当前 PersistentVolumeClaim 的工作负载实例数量及部署控制器类型考虑。例如，创建多实例部署（Deployment）时，由于所有实例共用同一个 PersistentVolumeClaim，不建议选择只能挂载到单节点的 RWO 访问模式。
Capacity	请求的 PersistentVolume 大小。
Volume Mode	- Filesystem：将 PersistentVolume 绑定为挂载到 Pod 的文件目录。此模式适用于任何类型的工作负载。 - Block Device：将 PersistentVolume 绑定为挂载到 Pod 的原始块设备。此模式仅适用于虚拟机。
More	- Labels - Annotations - Selector：选择静态绑定方式后，可使用选择器定位带有特定标签的 PersistentVolume。PersistentVolume 标签可用于标注存储的特殊属性，如磁盘类型或地理位置。

5. 点击 **Create**。等待 PersistentVolumeClaim 状态变为 `Bound`，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**：创建需要持久化数据存储的应用或工作负载时，绑定 PersistentVolumeClaim 以请求符合条件的 PersistentVolume。
- 使用 **Volume Snapshots** 创建 **PersistentVolumeClaim**：帮助备份应用数据并按需恢复，保障业务应用数据的可靠性。详情请参见 Using Volume Snapshots。
- 删除 **PersistentVolumeClaim**：可在详情页右上角点击 **Actions** 按钮，根据需要删除 PersistentVolumeClaim。删除前请确保 PersistentVolumeClaim 未绑定任何应用或工作负载，且不包含任何卷快照。删除后，平台将根据回收策略处理 PersistentVolume，可能会清除 PersistentVolume 中的数据并释放存储资源。请根据数据安全性谨慎操作。

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

1. 在左侧导航栏点击 Storage > Persistent Volume Claims (PVC)。
2. 找到对应的 PersistentVolumeClaim，点击⋮> Expand。
3. 填写新的容量。
4. 点击 Expand。扩容过程可能需要一些时间，请耐心等待。

通过 **CLI** 扩容 **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 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。详情请参见 Recover From PVC Expansion Failure

其他资源

- 如何标注第三方存储能力

使用卷快照

卷快照是持久卷声明（PVC）的某一时间点副本，可用于配置新的持久卷声明（预填充快照数据）或将现有持久卷声明回滚到之前的状态，实现备份应用数据并按需恢复的效果，从而保证应用数据的可靠性。

目录

前提条件

- 管理员已为当前集群部署了卷快照组件 **Snapshot Controller**，并在存储集群中启用了快照相关功能。
- 持久卷声明必须是动态创建的，且状态为 **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**。

卷快照自定义资源（CR）示例

该示例使用 CSI 快照类创建了名为 example-pvc 的 PVC 的快照。

```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
```

通过 Web 控制台创建卷快照

基于指定持久卷声明（PVC）创建卷快照

方法一

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Storage > Persistent Volume Claims (PVC)**。
3. 在列表中对应的持久卷声明旁点击⋮，选择 **Create Volume Snapshot**。
4. 填写快照描述，该描述有助于记录当前持久卷状态，例如 *应用升级前*。
5. 点击 **Create**。快照创建时间取决于网络状况和数据量，请耐心等待。
   当快照状态变为 `Available` 时，表示创建成功。

方法二

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

自定义方式创建卷快照

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

通过 **CLI** 创建卷快照

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

通过卷快照创建持久卷声明

当前平台仅支持基于使用 动态供应 存储类创建的 PVC 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：基于快照创建 PVC 时支持的访问模式与基于存储类创建 PVC 时支持的访问模式不同，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 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. 在左侧导航栏点击 **Storage > Persistent Volume Claims (PVC)**。
3. 点击列表中持久卷声明的名称。
4. 切换到 **Volume Snapshots** 标签页。
5. 在对应的卷快照旁点击⋮，选择 **Create Persistent Volume Claim**，并配置相关参数。
6. 点击 **Create**。

方法二

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Storage > Volume Snapshots**。
3. 在对应的卷快照旁点击⋮，选择 **Create Persistent Volume Claim**，并配置相关参数。
4. 点击 **Create**。

其他资源

- 创建 PVC

实用指南

通用临时卷

- 临时卷示例
- 主要特性
- 何时使用通用临时卷
- 与 emptyDir 的区别

使用 emptyDir

- emptyDir 示例
- 可选的 Medium 设置
- 主要特性
- 常见用例

使用本地卷配

- 前提条件
- 操作步骤
- 自动发现本地存

配置持久卷声明的灾难恢复

- 概述
- 术语
- 前提条件
- 部署 Alauda Build of VolSync
- 配置定时同步
- 配置一次性同步
- 启用应用 PVC 的灾难恢复
- 计划迁移
- 故障切换
- 故障恢复后切换回主集群（Failback）

第三方存储能力注解指南

1. 入门指南
2. 示例 ConfigMap
3. 更新现有能力描述
4. 与旧格式的兼容性
5. 常见问题解答

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
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）

使用本地卷配置持久存储

Alauda Container Platform 可以通过使用本地卷来配置持久存储。本地持久卷允许您通过标准的持久卷声明接口访问本地存储设备，例如磁盘或分区。

本地卷可以在无需手动调度 Pod 到节点的情况下使用，因为系统会识别卷的节点约束条件。然而，本地卷仍然受限于底层节点的可用性，并不适用于所有应用场景。

> **NOTE**
>
> 本地卷只能作为静态创建的持久卷使用。

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。

- 使用上传软件包机制上传该 **Alauda Build of LocalStorage** 安装包。
- 您拥有符合以下条件的本地磁盘：
  - 已挂载到某个节点。
  - 未被挂载使用。
  - 不包含分区。

操作步骤

1 安装 Local Storage Operator

1. 登录，进入管理员页面。
2. 点击 **Marketplace** > **OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **Alauda Build of LocalStorage**，点击 **Install**，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`：集群中所有命名空间共享单个 Operator 实例进行创建和管理，资源使用更低。
Installation Place	选择 `Recommended`，命名空间仅支持 **acp-storage**。
Upgrade Strategy	`Manual`：当 Operator Hub 有新版本时，需要手动确认升级 Operator 到最新版本。

2 使用 Local Storage Operator 预配本地卷

本地卷不能通过动态预配创建，而是由 Local Storage Operator 创建持久卷。Local Volume Provisioner 会在定义的资源中指定的路径查找任何文件系统或块卷设备。

1. 创建本地卷资源。该资源必须定义节点和本地卷的路径。

> **NOTE**
>
> 不要对同一设备使用不同的存储类名称，否则会创建多个持久卷（PV）。

在集群的控制节点上执行命令。

```
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) Local Storage Operator 安装的命名空间，默认是 `acp-storage`。

(2) 可选：包含本地存储卷所在节点列表的节点选择器。本示例使用节点主机名，可通过 `kubectl get node` 获取。如果未定义该值，Local Storage Operator 会尝试在所有可用节点上查找匹配磁盘。

(3) 创建持久卷对象时使用的存储类名称。如果该存储类不存在，Local Storage Operator 会自动创建。请确保使用唯一标识这组本地卷的存储类名称。

4 控制 Operator 是否在使用前擦除列出的设备。默认值为“false”。警告：设置 `forceWipeDevicesAndDestroyAllData: true` 会破坏性地擦除设备上的现有分区表/文件系统签名和数据。仅在确定设备可以安全擦除时使用。

5 卷模式，取值为 `Filesystem` 或 `Block` ，定义本地卷的类型。

6 可选：首次挂载本地卷时创建的文件系统类型。仅当 `volumeMode` 设置为 `Filesystem` 时配置此参数。

7 包含本地存储设备路径列表的路径。

8 将此值替换为实际本地磁盘的文件路径，如 `LocalVolume` 资源中的 `by-id` 路径，例如 `/dev/disk/by-id/wwn` 。当预配器成功部署时，会为这些本地磁盘创建 PV。

2. 验证持久卷是否创建成功：
   在集群的控制节点上执行命令。

```
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 对象不会更改已有持久卷的 fsType 或 volumeMode，因为这样可能导致破坏性操作。

3. 创建本地卷持久卷声明
   在集群的控制节点上执行命令。

```
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. PVC 所需的存储类名称。

自动发现本地存储设备

Local Storage Operator 支持自动发现本地存储设备。

前提条件

- 已安装 Local Storage Operator。

操作步骤

1. 创建 LocalVolumeDiscovery 对象
   在集群的控制节点上执行命令。

```
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：指定自动检测策略运行的节点。如果未定义该值，Local Storage Operator 会尝试在所有可用节点上自动检测。

2 可选：指定传递给 LocalVolumeDiscovery Daemon 的容忍度列表。

2. 验证发现结果

   在集群的控制节点上执行命令。

```
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
```

   LocalVolumeDiscovery 对象中选定节点会生成专用的 LocalVolumeDiscoveryResult 对象，您可以从中查看该节点上发现的块设备信息。

使用 NFS 配置持久存储

Alauda Container Platform 集群支持使用 NFS 的持久存储。Persistent Volumes (PVs) 和 Persistent Volume Claims (PVCs) 为项目内存储卷的配置和使用提供了抽象层。虽然可以直接在 Pod 定义中嵌入 NFS 配置细节，但这种方式不会将卷创建为独立的、隔离的集群资源，增加了冲突的风险。

目录

前提条件

- 底层基础设施中必须先存在存储，才能在 Alauda Container Platform 中将其挂载为卷。
- 配置 NFS 卷只需提供 NFS 服务器列表和导出路径。

操作步骤

1 创建 PV 的对象定义

```
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 卷的名称。

2 存储容量。

3 虽然看似与控制对卷的访问有关，但实际上它类似于标签，用于将 PVC 与 PV 匹配。目前，基于 accessModes 不会强制执行访问规则。

4 使用的卷类型，此处为 nfs 插件。

5 NFS 服务器地址。

6 NFS 导出路径。

7 PVC 删除后采取的操作（Retain、Delete、Recycle）。

2 验证 PV 是否创建成功

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) 访问模式不强制安全控制，而是作为标签匹配 PV 和 PVC。

(2) 此声明请求容量为 1Gi 或更大容量的 PV。

(3) 要使用的 PV 名称。

4 验证持久卷声明是否创建成功

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
```

通过分区导出实现磁盘配额

为了强制执行磁盘配额和大小限制，可以利用磁盘分区。将每个分区分配为专用导出点，每个导出对应一个独立的 PersistentVolume (PV)。

虽然 Alauda Container Platform 要求 PV 名称唯一，但确保每个导出的 NFS 卷的服务器和路径唯一性仍由管理员负责。

这种分区方式实现了精确的容量管理。开发者请求指定容量（例如 10Gi）的持久存储，ACP 会匹配至少满足该容量的分区/导出支持的 PV。请注意：配额限制适用于分配的分区/导出内的可用存储空间。

NFS 卷安全性

本节介绍 NFS 卷的安全机制，重点是权限匹配。假设读者具备 POSIX 权限、进程 UID 和附加组的基础知识。

开发者通过以下方式请求 NFS 存储：

- 通过名称引用 PersistentVolumeClaim (PVC)，或
- 在 Pod 规格的 volumes 部分直接配置 NFS 卷插件。

在 NFS 服务器上，/etc/exports 文件定义了可访问目录的导出规则。每个导出目录保留其原生的 POSIX 所有者/组 ID。

Alauda Container Platform 的 NFS 插件关键行为：

1. 挂载卷到容器时，保留源目录的精确 POSIX 所有权和权限

2. 运行容器时不强制进程 UID 与挂载所有权匹配——这是有意的安全设计

例如，考虑以下 NFS 目录的服务器端属性：

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)
```

此时，容器必须以 UID 65534（nfsnobody 所有者）运行，或在其附加组中包含 5555，才能访问该目录。

> **NOTE**
>
> 注意 65534 所有者 ID 仅为示例。尽管 NFS 的 root_squash 会将 root（uid 0）映射为 nfsnobody（uid 65534），但 NFS 导出可以有任意所有者 ID。NFS 导出不必是所有者 65534。

组 ID

推荐的 NFS 访问管理（当导出权限固定时） 当无法修改 NFS 导出的权限时，推荐通过附加组管理访问。

在 Alauda Container Platform 中，附加组是控制共享文件存储（如 NFS）访问的常用机制。

与块存储对比：块存储卷（如 iSCSI）的访问通常通过在 pod 的 securityContext 中设置 fsGroup 实现，该方法依赖挂载时文件系统组所有权的更改。

> **NOTE**
>
> 通常建议使用附加组 ID 来访问持久存储，而非使用用户 ID。

由于示例目标 NFS 目录的组 ID 为 5555，pod 可以在其 securityContext 的 supplementalGroups 中定义该组 ID。例如：

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

1. securityContext 必须定义在 pod 级别，而非某个具体容器下。
2. 定义 pod 的 GID 数组，此处数组中有一个元素，多个 GID 用逗号分隔。

用户 ID

用户 ID 可以在容器镜像中定义，也可以在 Pod 定义中指定。

> **NOTE**
>
> 通常建议使用附加组 ID 来访问持久存储，而非使用用户 ID。

在上述示例目标 NFS 目录中，容器需要将其 UID 设置为 65534（暂不考虑组 ID），因此可以在 Pod 定义中添加：

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

1 Pod 包含针对每个容器的 securityContext 定义，以及适用于所有容器的 pod 级 securityContext。

2 65534 是 nfsnobody 用户。

导出设置

为了允许任意容器用户读写卷，NFS 服务器上的每个导出卷应满足以下条件：

- 每个导出必须使用如下格式导出：

```
# 将 10.0.0.0/24 替换为可信的 CIDRs/主机
/<example_fs> 10.0.0.0/24(rw,sync,root_squash,no_subtree_check)
```

- 防火墙必须配置为允许访问挂载点的流量。
  - 对于 NFSv4，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
```

- NFS 导出和目录必须设置为目标 pod 可访问。要么将导出目录所有权设置为容器的主 UID，要么通过 supplementalGroups 提供 pod 组访问权限，如上文组 ID 所示。

资源回收

NFS 实现了 Alauda Container Platform 的 Recyclable 插件接口。自动流程根据每个持久卷设置的策略处理回收任务。

默认情况下，PV 设置为 Retain。

当 PVC 的声明被删除，且 PV 被释放后，该 PV 对象不应被重复使用。应创建一个新的 PV，基本卷信息与原 PV 相同。

例如，管理员创建了名为 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，绑定到 nfs1。用户随后删除 PVC1，释放对 nfs1 的声明。此时 nfs1 状态变为 Released。如果管理员想继续使用相同的 NFS 共享，应创建一个新的 PV，使用相同的 NFS 服务器信息，但 PV 名称不同：

```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 PV 后用相同名称重新创建。尝试手动将 PV 状态从 Released 改为 Available 会导致错误和潜在数据丢失。

配置持久卷声明的灾难恢复

要实现应用 PVC 的跨集群灾难恢复，请使用 **Alauda Build of VolSync**。

目录

操作流程

概述

Alauda Build of VolSync 是一个 operator，用于在集群内或跨集群异步复制持久卷。VolSync 提供的复制与存储系统无关，允许对通常不支持远程复制的存储类型进行复制。此外，它还能跨不同类型（及厂商）的存储进行复制。

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，准备在灾难发生时接管。
Stateful Application	使用 PVC 进行数据持久化的应用。
ReplicationSource	ReplicationSource 是 VolSync 资源，用于定义源 PVC 和复制数据搬运方式，使您能够将 PVC 数据复制或同步到远程位置。
ReplicationDestination	ReplicationDestination 是 VolSync 资源，用于定义 VolSync 复制或同步的目标位置。
Data Movers	VolSync 的数据搬运器负责将数据从一个位置复制到另一个位置。 支持的搬运器包括： • Rclone • Restic • Rsync

前提条件

- 下载 与您的平台架构对应的 **Alauda Build of VolSync** 安装包。
- 使用上传软件包机制将 **Alauda Build of VolSync** 安装包上传到主集群和备集群。
- 在主集群和备集群均已部署 **Alauda Container Platform Snapshot Management**。
- PVC 使用的存储必须由 **CSI** 供应，并支持 快照 功能。

部署 **Alauda Build of VolSync**

1. 登录，进入 **Administrator** 页面。
2. 点击 **Marketplace** > **OperatorHub** 进入 **OperatorHub** 页面。
3. 找到 **Alauda Build of VolSync**，点击 **Install**，进入 **Install Alauda Build of VolSync** 页面。
   配置参数：

参数	推荐配置
Channel	默认通道为 `stable`。
Installation Mode	`Cluster`：集群内所有命名空间共享一个 Operator 实例进行创建和管理，资源占用较低。
Installation Place	选择 `Recommended`，命名空间仅支持 **volsync-system**。
Upgrade Strategy	`Manual`：Operator Hub 有新版本时，需要手动确认升级 Operator 到最新版本。

配置定时同步

配置 PVC 的定时同步后，VolSync 会按照指定的时间间隔自动将数据从 `ReplicationSource` 同步到 `ReplicationDestination`。

本节介绍从主集群同步到备集群的配置步骤。若需从备集群同步到主集群，请将以下示例中的集群角色（主集群和备集群）互换。

1 创建 rsync-tls 数据搬运 Secret

在 主集群 和 备集群 上创建该 Secret；如果 Secret 已存在则跳过此步骤。

命令

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

示例

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

参数说明：

参数	说明
name	Secret 的名称
namespace	Secret 所在的命名空间，应与应用相同

参数	说明
psk	此字段遵循 stunnel 期望的格式：`<id>:<至少 32 个十六进制数字>`。 例如，`1:23b7395fafc3e842bd8ac0fe142e6ad1`。

2 创建 ReplicationDestination 资源

在 备集群 创建 `ReplicationDestination`

命令

```
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
```

示例

```
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
```

参数说明：

参数	说明
namespace	命名空间，应与应用相同
pvc-name	预先存在的 PVC 名称，应用所使用的
key-secret	包含 TLS-PSK 密钥的 Secret 名称，用于与源端认证，参见 步骤 1
service-type	VolSync 创建一个 Service 以允许源端连接目标端。此字段决定该 Service 的类型。允许值为 `ClusterIP`、`LoadBalancer` 或 `NodePort`。
storageclass-name	应用 PVC 使用的 storageclass 名称

参数	说明
volumesnapshotclass-name	对应应用 PVC 的 volumesnapshotclass 名称

NOTE

关于 **Service** 类型

如果指定为 `ClusterIP`，Service 会从“集群网络”地址池中分配一个 IP 地址。默认情况下，这些地址集群外不可访问，因此不适合跨集群复制。但某些网络插件如 Submariner 可以桥接集群网络，使其成为可行选项。

如果指定为 `LoadBalancer`，会分配一个外部可访问的 IP 地址。这需要集群支持负载均衡器，如云厂商提供的负载均衡器或物理集群中的 MetalLB。虽然这是云环境中分配可访问地址最简单的方法，但负载均衡器通常会产生额外费用且数量有限。

3 创建 **ReplicationSource** 资源

在 主集群 创建 `ReplicationSource`

命令

```
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
```

示例

```
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
```

参数说明：

参数	说明
namespace	命名空间名称，应与应用相同
pvc-name	应用 PVC 的名称
key-secret	volsync Secret 名称，参见步骤 1
address	指定复制目标的 ssh 服务器地址，可直接取自 ReplicationDestination 的 `.status.rsync.address` 字段。

参数	说明
port	连接目标的服务端口
storageclass-name	应用 PVC 使用的 storageclass 名称
volumesnapshotclass-name	对应应用 PVC 的 volumesnapshotclass 名称
schedule	同步计划，使用 cronspec 定义，支持灵活的时间间隔、具体时间和/或日期设置。

4 检查同步状态

从 `ReplicationSource` 查看同步状态

命令

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

示例

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

最近一次同步完成时间为 `.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 规范下设置唯一字符串的 manual 字段来控制。应用配置后，同步任务会立即执行一次。

1 创建一次性 **ReplicationSource** 资源

命令

```
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
```

示例

```
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` 设置为 **manual**。

2 检查同步状态

命令

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

示例

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

如果输出与 `<manual-id>` 匹配，则同步完成。

启用应用 PVC 的灾难恢复

1 部署有状态应用

1. 在 主集群 部署有状态应用

▶ 点击查看

2. 在 备集群 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 `主集群到备集群` 的同步

参见配置定时同步

计划迁移

用户场景：

在主集群和备集群均正常运行的情况下，将业务服务从主集群迁移至备集群。

操作流程

1. 缩减应用 **Pod**
   缩减所有使用灾难恢复 PVC 的应用 Pod，位于 主集群。
2. 删除 **ReplicationSource** 资源
   删除 主集群 上的 `ReplicationSource`

   命令

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

   示例

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

3. 创建一次性同步
   从 主集群 发起同步任务，确保 备集群 数据为最新。
   在 主集群 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

   命令

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

示例

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

5. 删除 **ReplicationDestination** 资源
   删除 备集群 上的 `ReplicationDestination`

命令

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

示例

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

6. 扩展应用 **Pod**
   扩展所有使用灾难恢复 PVC 的应用 Pod，位于 备集群。
7. 设置备集群到主集群的同步
   通过在 主集群 创建 `ReplicationDestination`，在 备集群 创建 `ReplicationSource`，设置 PVC 灾难恢复的 `备集群到主集群` 同步。
   参见配置定时同步

故障切换

用户场景：

主集群突发宕机后，将服务切换至备集群。

操作流程

为确保数据完整性（防止主集群同步时发生故障），在 备集群 进行本地同步。以应用 PVC 最近快照恢复的 PVC 作为源，应用当前 PVC 作为目标，执行数据同步。

1. 恢复 **PVC**

   从 备集群 的 `ReplicationDestination` 恢复 PVC

   命令

```
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
```

   示例

```
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 备集群 创建 `ReplicationSource` 资源

   命令

```
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
```

示例

```
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

命令

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

示例

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

如果输出与 `<manual-id>` 匹配，则同步完成。

4. 删除本地 **ReplicationSource**

   删除 备集群 上的本地 `ReplicationSource`

   命令

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

   示例

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

5. 删除 **ReplicationDestination**

   删除 备集群 上的 `ReplicationDestination`

   命令

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

   示例

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

6. 扩展应用 **Pod**

   扩展 备集群 上所有应用 Pod。

故障恢复后切换回主集群（**Failback**）

用户场景：

主集群已恢复并正常运行，需要将服务切换回主集群。

操作流程

1. 缩减主集群应用 **Pod**
   主集群恢复后，应用 Pod 会自动恢复，但需先缩减服务以停止流量。同步完备集群最新数据到主集群后，再扩展应用恢复正常运行。
2. 删除主集群 **ReplicationSource**
   需先删除主集群故障前创建的 `ReplicationSource`。

   命令

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

   示例

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

3. 从备集群同步最新数据
   设置 `备集群到主集群` 的一次性同步。
   在主集群创建 `ReplicationDestination`，在备集群创建一次性 `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。
   删除备集群的 `ReplicationSource`

   命令

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

示例

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

删除主集群的 `ReplicationDestination`

命令

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

示例

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

5. 迁移应用

   缩减 备集群 应用 Pod

   扩展 主集群 应用 Pod

6. 设置主集群到备集群的同步

   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

操作步骤

额外提示

从 PVC 扩容失败中恢复

当 Kubernetes 中的 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，删除其规格中的 `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. 重新创建一个较小的 PVC，或者创建一个底层存储提供商支持的大小的 PVC。
5. 在新的 PVC 中显式指定 `volumeName` 字段，使其与原 PV 名称匹配。这样可以确保新的 PVC 准确绑定到指定的 PV。
6. 最后，恢复 PV 的原始回收策略。

额外提示

- 确保所使用的 `StorageClass` 已启用卷扩容功能，即将 `allowVolumeExpansion` 设置为 `true`。
- 操作时请谨慎，以避免数据丢失的风险。

对象存储

介绍

核心概念

安装

卸载

操作指南

为 Ceph RGW 创建 BucketClass

前提条件

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

为 MinIO 创建 BucketClass

前提条件

步骤 1 - 准备 MinIO 集群

步骤 2 - 准备凭证 Secret

步骤 3 - 创建 BucketClass

验证与后续步骤

创建 Bucket

前提条件

操作步骤

相关

实用指南

使用 CephObjectStoreUser（Ceph 驱动）控制 COSI 桶的访问权限和配额

前提条件

第 1 步 — 创建带权限和配额的 CephObjectStoreUser

第 2 步 — 定义绑定到 CephObjectStoreUser 的 BucketClass

第 3 步 — 使用 BucketClaim 创建桶

第 4 步 — 使用 BucketAccessClass/BucketAccess 颁发最小权限凭证

第 5 步 — 匿名公共读取（可选）

第 6 步 — 配额控制：在哪里强制及如何修改

运营与故障排查

清理

介绍

Container Object Storage Interface (COSI) 是一个 Kubernetes 原生框架，旨在为 Kubernetes 集群内的对象存储服务（如 AWS S3、MinIO 和 Ceph RGW）提供标准化且声明式的管理方式。COSI 扩展了 Kubernetes 的存储模型，以一种可移植、可扩展且符合 Kubernetes 原则的方式支持对象存储资源。

COSI 使管理员能够通过熟悉的 Kubernetes 风格 API 定义、创建和使用对象存储桶。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证的生命周期管理。借助 COSI，Kubernetes 用户可以动态请求对象存储资源，减少手动配置工作量，提高运维效率。

采用 COSI，企业可以：

- 在多云和本地环境中实现对象存储创建的标准化。
- 通过声明式资源定义动态创建和管理存储桶。
- 通过 Kubernetes Secrets 无缝分发访问凭证给工作负载。
- 将对象存储管理与 Kubernetes 持久存储模式对齐，实现统一体验。

目录

限制

- COSI 目前处于 alpha 版本。

- 目前，COSI 仅支持 Ceph RGW 和 MinIO 驱动。
- 与传统对象存储桶的集成可能需要额外的手动配置。

核心概念

目录

Overview

本文档向熟悉持久存储概念的 Kubernetes 管理员介绍 Container Object Storage Interface (COSI) 的核心资源和原理。COSI 提供了一种声明式机制来管理对象存储（如 AWS S3、MinIO 和 Ceph RGW），类似于现有的 Kubernetes 持久存储管理方式。

我们将介绍 COSI 中的三种主要资源——**BucketClass**、**Bucket** 和 **BucketClaim**，通过与 Kubernetes 存储资源的类比来阐明它们之间的关系和功能。

Core Resources

COSI 定义了三种核心资源：

1. BucketClass

作用域： 集群级别
对应 **Kubernetes** 概念： 类似于 StorageClass

BucketClass 由集群管理员创建，用于定义桶的具体类型或服务等级，包括地域位置、冗余策略和性能等级。

主要功能：

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

YAML 示例：

```
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

作用域： 集群级别
对应 **Kubernetes** 概念： 类似于 PersistentVolume (PV)

Bucket 表示 Kubernetes 中对外部对象存储系统（如 AWS S3、MinIO、Ceph RGW）中实际桶的抽象。

生命周期管理：

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

3. BucketClaim

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

BucketClaim 资源由应用开发者在其命名空间内创建，用于请求对象存储桶。

工作流程：

1. 用户创建指定 BucketClass 的 BucketClaim。
2. COSI 控制器检测请求，根据 BucketClass 定义动态创建对象存储后端的桶。
3. 创建对应的 Bucket 资源并绑定到 BucketClaim。
4. 生成包含桶访问凭据的 Secret，并自动挂载到请求该桶的 Pod 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
```

Resource Interaction Workflow

以下流程展示了 COSI 资源的动态创建过程：

1. 集群管理员 创建并维护 BucketClass。
2. 命名空间用户 创建引用 BucketClass 的 BucketClaim。
3. **COSI** 控制器 监听 BucketClaim，根据 BucketClass 定义动态创建桶。
4. 控制器在 Kubernetes 中生成对应的 Bucket 资源。
5. BucketClaim 与 Bucket 绑定。

6. 创建包含存储凭据的 Secret 供 Pod 使用。
7. Pod 挂载该 Secret 并访问对象存储。

Summary

COSI 资源	作用域	**Kubernetes** 类比	作用
BucketClass	集群级别	StorageClass	定义桶的类型和策略
Bucket	集群级别	PersistentVolume (PV)	Kubernetes 对实际桶的抽象
BucketClaim	命名空间级别	PersistentVolumeClaim (PVC)	用户请求桶资源

通过利用 COSI 提供的标准化 API，Kubernetes 管理员可以声明式且可移植地管理对象存储资源，大大提升应用与对象存储在 Kubernetes 集群中的集成效率。

安装

目录

前提条件

1. 下载与您的平台架构对应的**Alauda Container Platform COSI**集群插件包。
2. 根据您计划使用的对象存储方案，下载****Alauda Container Platform COSI for Ceph**或**Alauda Container Platform COSI for MinIO**集群插件包。
3. 使用**Upload Packages**功能上传下载的插件包。
4. 使用**Cluster Plugins**安装机制将插件包安装到目标集群。

> **INFO**
>
> Upload Packages： 进入**Platform Management** > **Marketplace** > **Upload Packages**页面。 点击右侧的**Help Document**获取如何将集群插件发布到您想使用的集群的操作说明。更多详情请参考CLI。

安装 Alauda Container Platform COSI

约束与限制

- 插件仅限于当前集群范围。您必须在每个需要启用 COSI 功能的集群上单独安装所需插件。
- **Alauda Container Platform COSI for Ceph** 和 **Alauda Container Platform COSI for MinIO** 都依赖于 **Alauda Container Platform COSI** 插件。请确保先安装 **Alauda Container Platform COSI** 插件。

操作步骤

1. 登录平台，进入 **Platform Management** 页面。
2. 进入 **Marketplace** > **Cluster Plugins**，访问可用集群插件列表。
3. 选择要安装插件的目标集群。
4. 找到 **Alauda Container Platform COSI** 插件，从⁝菜单中选择 **Install** 开始安装。
5. 根据所选后端，找到并安装 **Alauda Container Platform COSI for Ceph** 或 **Alauda Container Platform COSI for MinIO** 插件。
6. 等待所有插件状态变为 **Installed**。

卸载

1. 登录平台，进入 **Platform Management** 页面。
2. 进入 **Marketplace** > **Cluster Plugins**，访问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，从⁝菜单中选择 **Uninstall** 开始卸载。
5. 等待插件状态变为 **Ready**，表示可以根据需要重新安装。

重要提示： 在卸载 **Alauda Container Platform COSI** 插件之前，必须先卸载 **Alauda Container Platform COSI for Ceph** 和/或 **Alauda Container Platform COSI for MinIO**。

操作指南

为 Ceph RGW 创建 BucketClass

Ceph 对象存储可以通过 **Container Object Storage Interface (COSI)** 暴露给 Kubernetes 工作负载，提供高度可扩展和弹性的存储，适用于大数据分析、备份与恢复以及机器学习场景。用户在创建桶之前需要先创建一个 *BucketClass*。

BucketClass 是一个模板资源，指定存储驱动、认证 Secret 以及应用于从该模板创建的每个桶的删除策略。

目录

前提条件

需求	说明
运行中的启用 RGW (S3) 的 Ceph 集群	内部（Rook 管理）或外部集群均可。
Alauda Container Platform COSI 插件	必须同时安装 **Alauda Container Platform COSI** 和 **Alauda Container Platform 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

选择以下 一种 方式：

选项	说明
内部 **Ceph**	由 Rook Operator 在平台内部部署和管理的 Ceph 集群。详情请参见 *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. **Alauda Container Platform COSI**
2. **Alauda Container Platform COSI for Ceph**

> 具体命令请参考 *安装*。

步骤 3 – 准备凭证 Secret

COSI 从 Kubernetes **Secret** 中获取 RGW 凭证。根据您的 Ceph 部署选择 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**：

```yaml
# ceph-object-store-user.yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  store: object-store                # 您的 CephObjectStore 名称
  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（后续使用）：

```
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 Secret，并添加标签以便 UI 发现：

```
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 的必要条件。

步骤 4 – 创建 BucketClass

选项 1 – UI 操作流程

1. 进入 **Storage** → **Object StorageClass**，点击 **Create Object StorageClass**。
2. 选择驱动为 **Ceph Object Storage**。
3. 配置以下字段：
   - **Deletion Policy** – 当 BucketClaim 被删除时底层桶的处理方式（默认：`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)* 限制使用的特定项目。
4. 点击 **Create**。

选项 2 – YAML（GitOps 友好）

创建 `ceph-bucketclass.yaml`，正确引用 Secret：

```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 准备好后，您可以创建引用它的 **Bucket** 或 **BucketClaim** 资源，从而为您的应用程序提供兼容 S3 的对象存储。

为 MinIO 创建 BucketClass

MinIO 通过 **Container Object Storage Interface (COSI)** 与 Kubernetes 集成，提供可扩展的、兼容 S3 的对象存储，适用于分析、备份与恢复以及 ML/AI 工作负载。在配置桶之前，需要定义一个 *BucketClass*。

BucketClass 是一个模板资源，用于设置存储驱动、认证 Secret 以及应用于所有由其创建的桶的删除策略。

目录

前提条件

需求	说明
已准备好的 MinIO 集群	按照 安装指南 准备 MinIO。

需求	说明
Alauda Container Platform COSI 插件	必须安装 **acp-cosi** 和 **acp-cosi-minio**。安装步骤请参见 安装 COSI 插件。
包含 MinIO 凭证的 Kubernetes Secret	在 步骤 **2** 中准备。

步骤 1 - 准备 MinIO 集群

确保已安装并可访问 MinIO 集群。请参考 MinIO 安装文档 部署和配置您的 MinIO 环境。

步骤 2 - 准备凭证 Secret

COSI 从 Kubernetes **Secret** 中获取 MinIO 凭证。收集以下值：

- `Endpoint` - 例如 `http://minio.minio-system.svc` 或 `https://minio.example.com:9000`
- `AccessKey`
- `SecretKey`

在目标命名空间中创建 Secret，并为 UI 发现添加标签：

```
kubectl create secret generic minio-credentials -n <YOUR_NAMESPACE> \
  --from-literal=Endpoint=http://<YOUR_MINIO_ENDPOINT> \
  --from-literal=AccessKey=<YOUR_ACCESS_KEY> \
  --from-literal=SecretKey=<YOUR_SECRET_KEY>

kubectl label secret minio-credentials -n <YOUR_NAMESPACE> app=minio
```

重要提示： 标签 `app=minio` 是平台 UI 列出该 Secret 的必要条件。

注意： 键名 区分大小写，必须严格为 `Endpoint`、`AccessKey` 和 `SecretKey`。

如果您偏好 GitOps，可以声明式定义 Secret：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: minio-credentials
  namespace: <YOUR_NAMESPACE>
  labels:
    app: minio
type: Opaque
stringData:
  Endpoint: http://<YOUR_MINIO_ENDPOINT>
  AccessKey: <YOUR_ACCESS_KEY>
  SecretKey: <YOUR_SECRET_KEY>
```

步骤 3 - 创建 BucketClass

选项 1 - UI 操作流程

1. 进入 **Storage** → **Object StorageClass**，点击 **Create Object StorageClass**。
2. 选择驱动为 **MinIO Object Storage**。
3. 配置以下字段：
   - **Deletion Policy** - 当 BucketClaim 被删除时，底层桶的处理方式（默认值：`Delete`）。
   - **Secret** - 选择步骤 2 中创建的 Secret（仅显示带有 `app=minio` 标签的 Secret）。
   - **Allocate Projects** - 可选：限制使用范围到特定项目。
4. 点击 **Create**。

选项 2 - YAML（支持 GitOps）

创建 `minio-bucketclass.yaml`。以下示例使用 MinIO COSI 驱动，并指向正确的 Secret 引用。

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
driverName: minio.objectstorage.k8s.io
metadata:
  labels:
    project.cpaas.io/name: null
    project.cpaas.io/ALL_ALL: "true"
  name: minio-bucket-class
  annotations:
    cpaas.io/display-name: BucketClass for MinIO
    cpaas.io/access-mode: ""
    cpaas.io/features: ""
parameters:
  providerSecretName: <your-secret-name>
  providerSecretNamespace: <your-secret-namespace>
deletionPolicy: Delete
```

应用清单：

```
kubectl apply -f minio-bucketclass.yaml
```

验证与后续步骤

验证 BucketClass：

```
kubectl get bucketclass
```

当 BucketClass 准备就绪后，创建引用该 BucketClass 的 **Bucket** 或 **BucketClaim** 资源，以配置由 MinIO 支持的兼容 S3 的对象存储。

创建 Bucket Request

使用 **Bucket Request** 可以基于 **Object Storage Class** 动态创建 bucket 并自动绑定二者。

目录

前提条件

- 安装 **Alauda Container Platform COSI** 集群。
- 根据计划使用的对象存储方案，安装 **Alauda Container Platform COSI for Ceph** 或 **Alauda Container Platform COSI for MinIO** 集群插件包。

有关详细的安装操作步骤，请参见 Installing。

操作步骤

1. 切换到 **Container Platform** 视图。
2. 在左侧导航栏中，点击 **Storage > Bucket Claims**。
3. 点击 **Create a Bucket Request**。
4. 按照下表配置参数。

参数	说明
Name	Bucket request 的名称。
Object Storage Class	用于动态创建 bucket 并建立绑定的 Object Storage Class。

5. 点击 **Create**。等待状态变为 `Available`，表示请求已完成且绑定成功。

相关

在 bucket request 详情页，点击右上角的 **Actions** 可根据需要选择 **Delete bucket policy**。警告： 删除 bucket 策略会清空 bucket 中的所有数据。请谨慎操作，确认数据备份和安全要求后再执行删除。

实用指南

使用 CephObjectStoreUser（Ceph 驱动）控制 COSI 桶的访问权限和配额

本指南向 Kubernetes 管理员展示如何结合 **CephObjectStoreUser (COSU)**、**BucketClass/BucketClaim** 和 **BucketAccessClass/BucketAccess**，实现基于 Ceph RGW 的 COSI 桶的最小权限访问和配额管理。

> 您将构建的内容
>
> 1. 一个具有明确权限和可选用户配额的 CephObjectStoreUser；
> 2. 一个告诉 Ceph COSI 驱动使用哪个 COSU 凭证的 BucketClass；
> 3. 一个或多个用于创建桶的 BucketClaim；
> 4. 使用 BucketAccessClass/BucketAccess 实现细粒度的按工作负载凭证（只读、只写、读写），并可选支持匿名读取。

目录

前提条件

- 一个健康的 Ceph 集群，已安装 RGW 和 Rook。
- 已安装 COSI 插件。
- 集群管理员权限（用于创建集群范围资源）。

第 1 步 — 创建带权限和配额的 CephObjectStoreUser

`CephObjectStoreUser` 是驱动用于在 RGW 中执行桶操作的服务账户。它必须位于 **`rook-ceph`** 命名空间，并指向您的 `CephObjectStore`。

```yaml
apiVersion: ceph.rook.io/v1
kind: CephObjectStoreUser
metadata:
  name: user-for-cosi
  namespace: rook-ceph
spec:
  # 目标 CephObjectStore
  store: object-store
  # COSI 桶生命周期所需的权限
  capabilities:
    bucket: read, write
    user: read, write
  # RGW 强制的可选用户配额
  quotas:
    maxBuckets: 50         # 限制该用户可拥有的桶数量
    maxObjects: 500        # 桶中对象总数（示例）
    maxSize: 100Gi         # 桶中对象总逻辑大小
  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）：

```
kubectl get cephobjectstoreuser user-for-cosi -n rook-ceph -o yaml
# 查看 status.info.secretName -> 持有 AccessKey/SecretKey 的 Secret 名称
```

> 这些 COSU 凭证由驱动程序使用（而非您的应用）来代表您创建/删除桶。

第 2 步 — 定义绑定到 CephObjectStoreUser 的 BucketClass

`BucketClass` 告诉驱动在创建桶时使用哪个 COSU Secret。

```
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClass
metadata:
  name: ceph-cosi-driver-class
deletionPolicy: Delete
# 必须匹配驱动名称
driverName: ceph.objectstorage.k8s.io
parameters:
  # 引用为 CephObjectStoreUser 创建的 Secret
  objectStoreUserSecretName: <secret-name-from-step-1>
  objectStoreUserSecretNamespace: rook-ceph
```

> `deletionPolicy` 控制删除 `BucketClaim` 时桶的物理生命周期（`Delete` 或 `Retain`）。

第 3 步 — 使用 BucketClaim 创建桶

在您的应用命名空间中通过引用 `BucketClass` 创建桶。

```yaml
apiVersion: objectstorage.k8s.io/v1alpha1
kind: BucketClaim
metadata:
  name: my-bucket-claim
  namespace: app-a
spec:
  bucketClassName: ceph-cosi-driver-class
  # COSI API 要求（选择所需的数据协议）
  protocols:
  - S3
```

等待 `status.bucketReady: true`，并记录 `status.bucketName`，即实际的 RGW 桶名称。

第 4 步 — 使用 BucketAccessClass/BucketAccess 颁发最小权限凭证

通过 `BucketAccessClass` 定义策略模板，并通过 `BucketAccess` 为每个工作负载生成凭证。支持的策略有：**`readonly`**、**`writeonly`**、**`readwrite`**。通过设置 `parameters.anonymous: "true"`（字符串）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`** 生成凭证

```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
```

驱动会写入名为 `credentialsSecretName` 的 Secret。解码 `.data.BucketInfo`（base64）即可获取 `secretS3.endpoint`、`accessKeyID` 和 `accessSecretKey`，供您的 S3 客户端使用。

> 提示：为每个 Deployment/Job 颁发独立凭证，便于轮换和吊销，避免影响其他工作负载。

第 5 步 — 匿名公共读取（可选）

如果需要公共静态资源托管：

```yaml
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"
```

将其绑定到您的 `BucketClaim` 的 `BucketAccess`。授权后，可通过未认证的 HTTP `GET` 访问对象（请确保网络暴露和访问路径正确）。

第 6 步 — 配额控制：在哪里强制及如何修改

作用范围： `CephObjectStoreUser` 上的 `quotas` 块针对每个用户生效，由 RGW 在该用户拥有的所有桶中统一强制。

- `maxBuckets`：用户可创建/拥有的桶数量上限。
- `maxObjects`：用户可存储的对象总数上限（跨桶）。
- `maxSize`：用户允许的总逻辑存储大小。

更新配额，编辑 COSU 资源：

```
kubectl -n rook-ceph edit cephobjectstoreuser user-for-cosi
# 修改 spec.quotas.{maxBuckets,maxObjects,maxSize}，保存退出。
# Rook 会自动协调并应用新的 RGW 用户配额。
```

> 设计理念： 保持 **COSU** 配额 较为严格以限制影响范围。通过 BAC/BA 使用最小权限策略，限制应用凭证在桶内的操作权限。

运营与故障排查

- 命名空间放置：`CephObjectStoreUser` 及其 Secret 必须在 `rook-ceph`，应用级资源（`BucketClaim`、`BucketAccess`）应在应用命名空间。
- 策略未生效：确认 BAC 中的 `bucketAccessClassName` 和 `parameters.policy` 是否正确（`readonly|writeonly|readwrite`）。
- 匿名读取失败：确保 `anonymous: "true"` 是字符串，非布尔值；验证 endpoint 暴露和 HTTP 访问路径（`/<bucket>/<object>`）。
- 找不到密钥：检查 `BucketAccess` Secret，解码 `.data.BucketInfo`。
- 桶状态一直未就绪：查看控制器/驱动日志（例如 `kubectl -n cpaas-system logs deploy/ceph-cosi-driver -c ceph-cosi-driver`）。
- 凭证轮换：创建新的 `BucketAccess`，将工作负载切换到新 Secret，然后删除旧 Secret 和 BA。

清理

- 删除工作负载凭证：删除对应的 `BucketAccess` 和引用的 Secret。
- 删除桶：删除 `BucketClaim`（行为遵循 `BucketClass.deletionPolicy`）。如果后端拒绝删除因桶非空，请先删除桶内对象。