云原生存储

云原生化存储的几种类型：

1. **传统存储云原生化**，通过 CSI 标准 同 Kubernetes 平台对接，此类型相对来说比较普遍， 用户可利用现有存储，并且基于传统存储的提供云原生存储能力稳定性好，SLA 强保障。
2. **软件定义存储云原生化**，软件定义存储，兼容传统应用和云原生应用。同样基于 CSI 标准同 Kubernetes 对接。软件定义存储通过网络使用企业中的每台机器上的磁盘空间， 并将这些分散的存储资源构成一个虚拟的存储设备，数据分散在不同的存储设备中。
3. **纯云原生存储**，此种类型的存储类型天然为云原生而生，构筑于云原生平台之上，能比较好的契合云原生特性， 并可随着应用 Pod 的迁移而迁移，具备如下特性：高可扩展性，高可用性，但相对于 通过 CSI 标准接入的传统存储可靠性低一些。

DCE 云原生存储

DCE 5.0 云原生存储基于 Kubernetes CSI 标准，可根据不同 SLA 要求及用户场景对接符合 CSI 标准的存储。 DaoCloud 推出的云原生本地存储天然具备云原生特性，满足容器场景中高扩展性、高可用性等特点。

云原生存储

云原生存储

背景

中间件服务上云的存储需求

1. 云即将成为新数字体验的核心，数据库、消息队列等中间件上云数量正在激增。 而在中间件等有状态应用上云过程中，如何保障业务应用的高性能及高可靠性是企业需要面对的问题。
2. 对于物理设施存算融合场景，如何利用现有的设备存储空间满足有状态应用上云需求。
3. 随着越来越多关键性应用上云，对于云上存储如何实现高效运维，如何保障数据存储的可靠性，如何实行全面的包括应用/控制面/后端设备监控也将成为新的挑战。

云边协同场景中的存储方案

在 Garner 2021 的十大云趋势预测中，边缘计算成为新云。 在边缘场景中，边缘端也存在数据存储需求以及边缘数据预处理需求。 在边缘端，一般采用存算融合，但边缘的资源有限，如何基于边缘端进行一定的数据存储和数据计算，同样也是云边协同场景中需要解决的问题。

云边协同场景

云边协同场景

挑战

企业就绪挑战

数据库、消息队列、大数据等有状态关键应用逐步上云，对于云上存储的性能（吞吐、时延）有新的更高的要求，此时如何满足高性能需求。

与此同时高效运维，保障数据存储的可靠性监控，实现全面的监控及问题预警，包括展示应用级/控制面挂载点以及后端设备的监控数据，并形成全链路监控，并能快速锁定存储问题。

敏捷化挑战

云原生应用场景对服务的敏捷度、灵活性要求非常高，很多场景期望容器的快速启动、灵活的调度，这样既需要存储卷，也需要敏捷地根据 Pod 的变化而调整。

1. 云盘挂载、卸载效率提高
2. 可以灵活的将块设备在不同节点进行快速的挂载切换
3. 提供存储服务的问题自动修复能力，减少人为干预
4. 在线扩容能力，为适用容器的快速扩容，如何实现应用业务不受影响的情况下在线扩容?
5. 随着越来越多的有状态应用容器化以及有状态应用部署在同一个集群，在集群规划业务分区域使用场景中，应用如何根据存储类型自动进行调度?

降本增效挑战

目前，数据中心计算密度提高的同时，如何解决存储性能瓶颈?

1. 定位排障性能和可用性问题（计划外的设备故障、故障定位及恢复）
2. 构建、配置、维护和扩展应用程序的复杂性（不同的部署环境及软件技术栈）
3. 存储瓶颈：解决计算瓶颈之后，则存储能力将成为制约计算密度的瓶颈
4. 业务应用按照业务峰值规划及设计，计算资源、存储资源的闲置率 ≥ 50%。成本高，如何感知业务进行自动扩容等资源均衡。

功能列表

DCE 5.0 云原生存储提供以下功能特性：

功能特性	描述

功能特性	描述
云原生统一管理	
混合存储接入	统一 CSI 标准化接入，实现多数据访问类型（NFS、块存储、本地存储）接入，满足不同场景需求。
动态存储管理	支持动态方式分配存储池资源，无需管理员手动管理、运维数据卷。
多数据卷创建方式	可通过存储池动态创建数据卷，以及支持快照创建数据卷。
云原生本地存储（HwameiStor）	
高性能本地卷	无需外接存储设备，实现 IO 本地化，无需网络开销，实现高性能本地吞吐，支持对性能有高要求的应用上云。
多类型数据卷	支持 LVM 类型、裸磁盘类型数据卷，满足不同磁盘需求场景。
CSI 标准	通过标准的 CSI 标准接

功能特性	描述
	入 Hwameistor 本地存储，统一使用姿势。
主备高可用	实现数据卷多副本冗余机制，保障数据高可用，提高数据读写可靠性。
数据卷扩容	业务无感扩容，支持应用在运行过程中，为挂载的数据卷进行弹性扩容。
生产可运维	
无中断升级（不影响业务数据）	数据平面、控制平面分离，控制平面升级/扩展节点时，业务应用数据读写 0 感知。
换盘	支持磁盘告警后进行换盘，保障业务应用不受影响，生产可运维。
一键驱逐节点数据卷	手动一键式驱逐某一节点数据卷，实现生产可运维。
自动驱逐节点数据卷	结合 Kubernetes 驱逐行为自动感知并驱逐节点上的数据卷。

功能特性	描述
单磁盘维度（LD）数据迁移	支持磁盘出现预警时并需要换盘，迁移所有数据，保障业务应用数据不丢失。
应用负载维度的数据迁移	支持有状态应用重调度时，以应用为维度实现数据迁移，保障业务应用 Pod 成功调度以及调度后的数据一致性。
统一仪表盘	统一仪表盘展示资源使用率情况及分布，存储资源状态及监控告警。
丰富 Metrics 指标	全方位的数据服务监控告警，实现数据盘、存储池及存储驱动的全面监控，全面保障数据安全性。

HwameiStor Release Notes

本页列出 HwameiStor 相关的 Release Notes，便于您了解各版本的演进路径和特性变化。

2024-10-30

v0.16.0

- **优化** 支持使用 LUKS 加密卷
- **优化** 卸载卷后关闭加密卷
- **优化** 将 CSI 更新至 v4.0.0
- **优化** 为了清楚起见，重命名 StorageClass 中的 secureSecret

2024-09-30

v0.15.0

- **优化** 使用 reflink 格式化文件系统（XFS）
- **修复** 修复 LVM 执行错误
- **修复** 修复节点磁盘引用未创建但已挂载的问题

2024-08-30

v0.14.9

- **优化** 为 ClusterRole 添加 verbs/patch 访问权限
- **修复** 绑定的 PVC 未被过滤的问题

2024-07-30

v0.14.8

- **优化** 将默认回收策略从删除更改为保留
- **修复** 将磁盘与分区表一起输出
- **修复** 不处理空 pvName 的问题
- **修复** 亲和力和污点跳过策略

2024-06-30

v0.14.7

- **优化** 解决亲和性问题并添加跳过亲和性的选项
- **优化** 清理数据集管理器代码

2024-04-30

v0.14.6

- **优化** 解决 lvr 不感知 pod 亲和力的问题
- **优化** 添加了在迁移期间启用数据验证的选项

2024-03-31

v0.14.4

- **优化** 数据卷卸载时候删除本地挂载路径

- **优化** 添加 PoolHDD FreeCap 打印列

2024-01-31

v0.14.1

优化

- **优化** 忽略失败策略时会跳过一些定义了 SC 但并没 SC 实例的卷
- **优化** 使用前检查本地磁盘是否存在
- **优化** 更新快照客户端
- **优化** Hwameictl 命令行支持快照
- **优化** Hwameictl 命令行支持快照添加集群

修复

- **修复** 修复 getnode api 的问题
- **修复** 修复 ctl disk_list 的问题
- **修复** 修复快照控制错误

2023-12-31

v0.14.0

优化

- **优化** 移除无意义的 poolType 的使用
- **优化** 添加 VolumeSnapshotClass

- **优化** Hwameictl 命令行支持了 snapshot 参数

修复

- **修复** Volume Group 始终保持与组中所有卷的可访问性一致
- **修复** 清理 localvolumemigrate 副本时出现 pending 的错误
- **修复** 索引器设置失败时添加退出系统语句
- **修复** StorageNodePoolDiskGet 获取 localdisk 错误的问题
- **修复** 用户删除 pvc 但未创建 pv 时的音量泄漏

2023-11-30

v0.13.3

新功能

- **新增** 支持卷克隆
- **新增** 当快照被发现时会延迟卷删除
- **新增** 根据源卷可访问性过滤节点

优化

- **优化** 仅在磁盘可用时记录磁盘声明事件

修复

- **修复** 在提交任务之前过滤 replicaSnapRestoreName
- **修复** LocalVolumeConvert 状态转换错误

- **修复** 索引器添加失败时退出
- **修复** 迁移卷时使用存储节点 IP
- **修复** 取消发布后清理副本
- **修复** 修复 apiserver getnodedisk 错误并添加 set-diskowner api

2023-10-30

v0.13.1

- **新增** 适配 Kubernetes v1.28 版本
- **新增** LVM 数据卷快照功能
- **优化** hwameistor-operator 新增组件资源配置
- **优化** 快照恢复超时时间调整
- **优化** 提高 LDM 默认日志等级

2023-8-30

v0.12.1

新功能

- **新增** 支持 ext 文件系统
- **新增** Local Disk 新特性参数
- **新增** 字段 记录 设备历史信息
- **新增** SN/ID Path 识别 磁盘
- **新增** 磁盘自动扩容功能

- **新增** 系统资源审计功能，如： 集群，存储节点，磁盘，数据卷等

修复

- **修复** LocalStoragePool 节点的 NVME 磁盘显示问题
- **修复** 迁移操作丢失副本状态问题
- **修复** 当 Disk Owner 为空时，磁盘不允许分配问题
- **修复** Failover 功能 deploy,Makefile 等问题

2023-7-05

v0.11.1

新增 支持自动检测 cgroup 版本

2023-6-25

v0.11.0

- **新增** 实现 IO 限制或 QoS
- **新增** 使用 /virtual/ 检测来识别虚拟块设备
- **修复** 存储池创建时间字段不一致问题

2023-5-26

v0.10.3

- **优化** Helm 模板中权限功相关内容

2023-5-25

v0.10.2

- **优化** Node Name . 自动转成 - 问题，影响 LDM 心跳检测
- **新增** Admin 权限功能

2023-5-17

v0.10.0

优化

- **新增** 识别并设置本地磁盘 Manager Owner 信息
- **优化** 当接收到 移除时间，将磁盘状态标记为‘inactive’
- **优化** 磁盘 Smart 指标仪表盘
- **新增** 本地存储池
- **优化** UI 组件状态展示
- **优化** 分离磁盘分配和磁盘状态更新过程
- **优化** 将 Exportor 端口重命名为 http-metrics
- **优化** 在 exporter service 中添加端口名称

缺陷

- **修复** LD bound，但是 LSN 中没有容量的问题
- **修复** Metrics 端口监听问题
- **修复** May occur not found 的问题

- **修复** Helm 中 UI tag 的问题

v0.9.3

优化

- **优化** 在使用阶段填充磁盘所有者信息
- **优化** udev 事件触发时合并磁盘自身属性
- **优化** 给 svc 添加标签
- **优化** 分离磁盘分配和磁盘状态更新过程
- **优化** 将 Exportor 端口重命名为 http-metrics
- **优化** 在 exporter service 中添加端口名称

缺陷

- **修复** LD bound，但是 LSN 中没有容量的问题
- **修复** Metrics 端口监听问题
- **修复** May occur not found 的问题
- **修复** Helm 中 UI tag 的问题

2023-3-30

v0.9.2

- **新增** UI relok8s

v0.9.1

- **新增** Volume Status 监控 Issue #741
- **修复** Local Storage 部署参数 Issue #742

v0.9.0

新功能

- **新增** 磁盘 Owner Issue #681
- **新增** Grafana DashBoard Issue #733
- **新增** Operator 安装方式，安装时自动拉取 UI Issue #679
- **新增** UI 应用 Label Issue #710

优化

- **新增** 磁盘的已使用容量 Issue #681
- **优化** 当未发现可用磁盘时，跳过打分机制 Issue #724
- **设置** DRBD 端口默认为 43001 Issue #723

2023-1-30

v0.8.0

- **优化** 中文文档
- **优化** value.yaml 文件
- **更新** Roadmap

- **优化** 当安装失败时，设置默认的失败策略

2022-12-30

v0.7.1

- **新增** HwameiStor DashBoard UI，可展现存储资源、存储节点等使用状态
- **新增** 界面管理 HwameiStor 存储节点、本地磁盘、迁移记录
- **新增** 存储池管理功能，支持界面展现存储池基本信息，及存储池对应节点信息
- **新增** 本地卷管理功能 ，支持界面执行数据卷迁移、高可用转换
- **优化** 数据迁移前不必要的日志，并规避其他 Namespace 下的 Job 执行影响

什么是 HwameiStor

HwameiStor 是一款 Kubernetes 原生的容器附加存储 (CAS) 解决方案，将 HDD、SSD 和 NVMe 磁盘形成本地存储资源池进行统一管理， 使用 CSI 架构提供分布式的本地数据卷服务，为有状态的云原生应用或组件提供数据持久化能力。

System architecture

System architecture

具体的功能特性如下：

1. 自动化运维管理

   可以自动发现、识别、管理、分配磁盘。根据亲和性，智能调度应用和数据。自动监测磁盘状态，并及时预警。

2. 高可用的数据

   使用跨节点副本同步数据， 实现高可用。发生问题时，会自动将应用调度到高可用数

据节点上，保证应用的连续性。

3. 丰富的数据卷类型

聚合 HDD、SSD、NVMe 类型的磁盘，提供非低延时，高吞吐的数据服务。

4. 灵活动态的线性扩展

可以根据集群规模大小进行动态的扩容，灵活满足应用的数据持久化需求。

产品优势

I/O 本地化

100% 本地吞吐，无网络开销，节点故障时 Pod 在副本节点启动，使用副本数据卷进行本地 IO 读写。

高性能、高可用性

- 100% IO 本地化，实现高性能本地吞吐
- 2 副本数据卷冗余备份，保障数据高可用

线性扩展

- 独立节点单元，最小 1 节点，扩展数量不限
- 控制平面、数据平面分离，节点扩展，不影响业务应用数据 I/O

CPU、内存开销小

IO 本地化，相同的 IO 读写，CPU 平稳，无较大波动，内存资源开销小

生产可运维

- 支持节点、磁盘、数据卷组（VG）维度迁移
- 支持换盘等运维行为

HwameiStor 发行版本 下载 DCE 5.0 安装 DCE 5.0 申请社区免费体验

本地磁盘管理器

LDM 架构图

LDM 架构图

目前 LDM 还处于 alpha 阶段。

基本概念

LocalDisk(LD): 这是 LDM 抽象的磁盘资源，一个 LD 代表了节点上的一块物理磁盘。

LocalDiskClaim (LDC): 这是系统使用磁盘的方式，通过创建 LDC 对象来向系统申请磁盘。用户可以添加一些对磁盘的描述来选择磁盘。

!!! note
目前，LDC 支持以下对磁盘的描述选项：

- NodeName
- Capacity
- DiskType（例如 HDD/SSD）

用法

1. 查看 LocalDisk 信息

```
kubectl get localdisk
  NAME                NODEMATCH          PHASE
  10-6-118-11-sda     10-6-118-11        Available
  10-6-118-11-sdb     10-6-118-11        Available
```

该命令用于获取集群中磁盘资源信息，获取的信息总共有三列，含义分别如下：

- **NAME:** 代表磁盘在集群中的名称。
- **NODEMATCH:** 表明磁盘所在的节点名称。
- **PHASE:** 表明这个磁盘当前的状态。

通过 kubectl get localdisk <name> -o yaml 查看更多关于某块磁盘的信息。

2. 申请可用磁盘。

1. 创建 LocalDiskClaim。

```
cat << EOF | kubectl apply -f -
  apiVersion: hwameistor.io/v1alpha1
  kind: LocalDiskClaim
  metadata:
    name: <localDiskClaimName>
  spec:
    description:
      # e.g. HDD,SSD,NVMe
      diskType: <diskType>
    # 磁盘所在节点
    nodeName: <nodeName>
    # 使用磁盘的系统名称 比如：local-storage,local-disk-manager
    owner: <ownerName>
  EOF
```

该命令用于创建一个磁盘使用的申请请求。在这个 yaml 文件里面，您可以在 description 字段添加对申请磁盘的描述，比如磁盘类型、磁盘的容量等等。

2. 查看 LocalDiskClaim 信息

```
kubectl get localdiskclaim <name>
```

3. LDC 被处理完成后，将立即被系统自动清理。如果 owner 是 local-storage，处理后的结果可以在对应的 LocalStorageNode 里查看。

本地存储

本地存储 (local Storage, LS) 是 HwameiStor 的一个模块，它旨在为应用提供高性能的本地持久化 LVM 存储卷。

本地存储架构图.png

本地存储架构图.png

目前支持的本地持久化数据卷类型：LVM。

目前支持的本地磁盘类型：HDD、SSD、NVMe。

适用场景

HwameiStor 提供两种本地数据卷：LVM、Disk。 本地存储作为 HwameiStor 的一部分，负责提供 LVM 本地数据卷，包括高可用 LVM 数据卷、非高可用 LVM 数据卷。

非高可用的 LVM 本地数据卷，适用下列场景和应用：

- 具备高可用特性的 **数据库** ，例如 MongoDB 等
- 具备高可用特性的 **消息中间件** ，例如 Kafka、RabbitMQ 等
- 具备高可用特性的 **键值存储系统** ，例如 Redis 等
- 其他具备高可用功能的应用

高可用的 LVM 本地数据卷，适用下列场景和应用：

- **数据库** ，例如 MySQL、PostgreSQL 等
- 其他需要数据高可用特性的应用

通过 hwameistor-operator 安装

hwameistor-operator 安装后会自动将 HwameiStor 相关组件拉起。 参阅通过 hwameistor-operator 安装 HwameiStor。

独立安装部署方式

开发者可以独立安装 local-storage，从源代码进行安装，主要用于开发、测试场景。 对于这种安装方式，您需要事先安装好本地磁盘管理器。

DRBD 安装器

DRBD (Distributed Replicated Block Device) 通过 Linux 内核模块和相关脚本构成，用以构建高可用性的集群。其实现方式是通过网络来镜像整个设备，可以把它看作是一种网络 RAID。

这个安装器可以直接安装 DRBD 到容器集群，目前该模块仅用于测试用途。

关于 DRBD 的商业化产品，请联系 Linbit。

DRBD 适配的内核版本

Hwameistor 安装后会自动部署 drbd-9.0.32-1，从而方便用户使用高可用性的存储类。

由于 drbd 是作为内核模块进行工作的，故建议使用与版本 drbd-9.0.32-1 对应的内核版本，如下表：

!!! note
如您当前的内核版本未包含在下表中，可以参考文档（todo）来操作

适配的内核版本：

- 5.8.0-1043-azure
- 5.8.0-1042-azure
- 5.8.0-1041-azure
- 5.4.17-2102.205.7.2.el7uek
- 5.4.17-2011.0.7.el8uek
- 5.4.0-91
- 5.4.0-90
- 5.4.0-89
- 5.4.0-88
- 5.4.0-86
- 5.4.0-84
- 5.4.0-1064-azure
- 5.4.0-1063-azure
- 5.4.0-1062-azure
- 5.4.0-1061-azure
- 5.4.0-1060-aws
- 5.4.0-1059-azure

- 5.4.0-1059-aws
- 5.4.0-1058-azure
- 5.4.0-1058-aws
- 5.4.0-1057-aws
- 5.4.0-1056-aws
- 5.4.0-1055-aws
- 5.3.18-57.3
- 5.3.18-22.2
- 5.14.0-1.7.1.el9
- 5.11.0-1022-azure
- 5.11.0-1022-aws
- 5.11.0-1021-azure
- 5.11.0-1021-aws
- 5.11.0-1020-azure
- 5.11.0-1020-aws
- 5.11.0-1019-aws
- 5.11.0-1017-aws
- 5.11.0-1016-aws
- 5.10.0-8
- 5.10.0-7
- 5.10.0-6
- 4.9.215-36.el7
- 4.9.212-36.el7
- 4.9.206-36.el7
- 4.9.199-35.el7
- 4.9.188-35.el7
- 4.4.92-6.30.1
- 4.4.74-92.38.1
- 4.4.52-2.1
- 4.4.27-572.565306
- 4.4.0-217
- 4.4.0-216
- 4.4.0-214
- 4.4.0-213
- 4.4.0-210
- 4.4.0-1133-aws
- 4.4.0-1132-aws
- 4.4.0-1131-aws
- 4.4.0-1128-aws
- 4.4.0-1121-aws
- 4.4.0-1118-aws
- 4.19.19-5.0.8
- 4.19.0-8
- 4.19.0-6

- 4.19.0-5
- 4.19.0-16
- 4.18.0-80.1.2.el8_0
- 4.18.0-348.el8
- 4.18.0-305.el8
- 4.18.0-240.1.1.el8_3
- 4.18.0-193.el8
- 4.18.0-147.el8
- 4.15.0-163
- 4.15.0-162
- 4.15.0-161
- 4.15.0-159
- 4.15.0-158
- 4.15.0-156
- 4.15.0-112-lowlatency
- 4.15.0-1113-azure
- 4.15.0-1040-azure
- 4.15.0-1036-azure
- 4.14.35-2047.502.5.el7uek
- 4.14.35-1902.4.8.el7uek
- 4.14.35-1818.3.3.el7uek
- 4.14.248-189.473.amzn2
- 4.14.128-112.105.amzn2
- 4.13.0-1018-azure
- 4.12.14-95.3.1
- 4.12.14-25.25.1
- 4.12.14-197.29
- 4.12.14-120.1
- 4.1.12-124.49.3.1.el7uek
- 4.1.12-124.26.3.el6uek
- 4.1.12-124.21.1.el6uek
- 3.10.0-957.el7
- 3.10.0-862.el7
- 3.10.0-693.el7
- 3.10.0-693.21.1.el7
- 3.10.0-693.17.1.el7
- 3.10.0-514.36.5.el7
- 3.10.0-327.el7
- 3.10.0-229.1.2.el7
- 3.10.0-123.20.1.el7
- 3.10.0-1160.el7
- 3.10.0-1127.el7
- 3.10.0-1062.el7
- 3.10.0-1049.el7

- 3.0.101-108.13.1
- 2.6.32-754.el6
- 2.6.32-696.el6
- 2.6.32-696.30.1.el6
- 2.6.32-696.23.1.el6
- 2.6.32-642.1.1.el6
- 2.6.32-573.1.1.el6
- 2.6.32-504.el6

DRBD 自助编译操作

当 DRBD 默认适配的内核版本不支持时，建议参考该文档的步骤，来编译对应的离线包并安装到对应环境中来使用 DRBD 的功能。

操作步骤

本文以内核为 4.9.212-36.el7.x86_64 为例进行演示。

编译内核

1. 确认内核版本

   ```
   uname -r
   ```

   输出结果为：

   ```
   Linux localhost.localdomain 4.9.212-36.el7.x86_64 #1 SMP Thu Feb 6 17:55:02 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
   ```

2. 下载源码

   ```
   wget https://pkg.linbit.com//downloads/drbd/9.0/drbd-9.0.32-1.tar.gz

   tar -xvf drbd-9.0.32-1.tar.gz
   ```

3. 下载安装内核编译模块

   ```
   # 下载 rpm 包，其他内核的可参阅 https://linux.cc.iitk.ac.in/mirror/centos/elrepo/kernel/el7/x86_64/RPMS/
   wget https://linux.cc.iitk.ac.in/mirror/centos/7/virt/x86_64/xen-common/Packages/k/kernel-4.9.212-36.el7.x86_64.rpm
   ```

```
# 开始安装
rpm -ivh kernel-devel-4.9.212-36.el7.x86_64.rpm
```

安装完执行检查：

```
$ ls -lF /lib/modules/`uname -r`/build
lrwxrwxrwx. 1 root root 38 Dec 5 13:47 /lib/modules/4.9.212-36.el7.x86_64/build -> /usr/src/kernels/4.9.212-36.el7.x86_64/
$ ls -lF /lib/modules/`uname -r`/source
lrwxrwxrwx. 1 root root 5 Dec 5 13:14 /lib/modules/4.9.212-36.el7.x86_64/source -> build/
```

4. 下载安装编译所需依赖

安装 gcc、make、patch、kmod、cpio、python3 以及 python3-pip 软件包：

```
yum install -y gcc make patch kmod cpio python3 python3-pip
```

安装 coccinelle：

```
git clone https://github.com/coccinelle/coccinelle.git
yum install -y ocaml ocaml-native-compilers ocaml-findlib menhir automake
./autogen
./configure
sudo make install
```

5. 开始编译

1. 进入到 drbd-9.0.32-1 目录下执行 make 命令，输出结果如下：

```
Calling toplevel makefile of kernel source tree, which I believe is in
KDIR=/lib/modules/4.9.212-36.el7.x86_64/source

make -C /lib/modules/4.9.212-36.el7.x86_64/source O=/lib/modules/4.9.212-36.el7.x86_64/build M=/home/drbd-9.0.32-1/drbd modules
COMPAT __vmalloc_has_2_params
COMPAT alloc_workqueue_takes_fmt
COMPAT before_4_13_kernel_read
COMPAT blkdev_issue_zeroout_discard
COMPAT can_include_vermagic_h
COMPAT drbd_release_returns_void
COMPAT genl_policy_in_ops
COMPAT have_BIO_MAX_VECS
COMPAT have_CRYPTO_TFM_NEED_KEY
COMPAT have_SHASH_DESC_ON_STACK
COMPAT have_WB_congested_enum
COMPAT have_allow_kernel_signal
COMPAT have_atomic_dec_if_positive_linux
```

COMPAT have_atomic_in_flight
COMPAT have_bd_claim_by_disk
COMPAT have_bd_unlink_disk_holder
COMPAT have_bdi_cap_stable_writes
COMPAT have_bdi_congested_fn
COMPAT have_bio_bi_bdev
COMPAT have_bio_bi_error
COMPAT have_bio_bi_opf
COMPAT have_bio_bi_status
COMPAT have_bio_clone_fast
COMPAT have_bio_flush
COMPAT have_bio_free
COMPAT have_bio_op_shift
COMPAT have_bio_rw
COMPAT have_bio_set_dev
COMPAT have_bio_set_op_attrs
COMPAT have_bio_start_io_acct
COMPAT have_bioset_create_front_pad
COMPAT have_bioset_init
COMPAT have_bioset_need_bvecs
COMPAT have_blk_alloc_disk
COMPAT have_blk_alloc_queue_rh
COMPAT have_blk_check_plugged
COMPAT have_blk_qc_t_make_request
COMPAT have_blk_queue_flag_set
COMPAT have_blk_queue_make_request
COMPAT have_blk_queue_merge_bvec
COMPAT have_blk_queue_plugged
COMPAT have_blk_queue_split_bio
COMPAT have_blk_queue_split_q_bio
COMPAT have_blk_queue_split_q_bio_bioset
COMPAT have_blk_queue_update_readahead
COMPAT have_blk_queue_write_cache
COMPAT have_blkdev_get_by_path
COMPAT have_d_inode
COMPAT have_fallthrough
COMPAT have_file_inode
COMPAT have_generic_start_io_acct_q_rw_sect_part
COMPAT have_generic_start_io_acct_rw_sect_part
COMPAT have_genl_family_parallel_ops
COMPAT have_hd_struct
COMPAT have_ib_cq_init_attr
COMPAT have_ib_get_dma_mr
COMPAT have_idr_alloc

COMPAT have_idr_is_empty
COMPAT have_inode_lock
COMPAT have_ktime_to_timespec64
COMPAT have_kvfree
COMPAT have_max_send_recv_sge
COMPAT have_netlink_cb_portid
COMPAT have_nla_nest_start_noflag
COMPAT have_nla_parse_deprecated
COMPAT have_nla_put_64bit
COMPAT have_nla_strscpy
COMPAT have_part_stat_h
COMPAT have_part_stat_read_accum
COMPAT have_pointer_backing_dev_info
COMPAT have_prandom_u32
COMPAT have_proc_create_single
COMPAT have_queue_flag_stable_writes
COMPAT have_ratelimit_state_init
COMPAT have_rb_augment_functions
COMPAT have_refcount_inc
COMPAT have_req_flush
COMPAT have_req_hardbarrier
COMPAT have_req_noidle
COMPAT have_req_nounmap
COMPAT have_req_op_write
COMPAT have_req_op_write_same
COMPAT have_req_op_write_zeroes
COMPAT have_req_prio
COMPAT have_req_write
COMPAT have_req_write_same
COMPAT have_revalidate_disk_size
COMPAT have_sched_set_fifo
COMPAT have_security_netlink_recv
COMPAT have_sendpage_ok
COMPAT have_set_capacity_and_notify
COMPAT have_shash_desc_zero
COMPAT have_signed_nla_put
COMPAT have_simple_positive
COMPAT have_sock_set_keepalive
COMPAT have_struct_bvec_iter
COMPAT have_struct_kernel_param_ops
COMPAT have_struct_size
COMPAT have_submit_bio
COMPAT have_submit_bio_noacct
COMPAT have_tcp_sock_set_cork

```
COMPAT  have_tcp_sock_set_nodelay
COMPAT  have_tcp_sock_set_quickack
COMPAT  have_time64_to_tm
COMPAT  have_timer_setup
COMPAT  have_void_make_request
COMPAT  hlist_for_each_entry_has_three_parameters
COMPAT  ib_alloc_pd_has_2_params
COMPAT  ib_device_has_ops
COMPAT  ib_post_send_const_params
COMPAT  ib_query_device_has_3_params
COMPAT  kmap_atomic_page_only
COMPAT  need_make_request_recursion
COMPAT  part_stat_read_takes_block_device
COMPAT  queue_limits_has_discard_zeroes_data
COMPAT  rdma_create_id_has_net_ns
COMPAT  sock_create_kern_has_five_parameters
COMPAT  sock_ops_returns_addr_len
CHK     /home/drbd-9.0.32-1/drbd/compat.4.9.212-36.el7.x86_64.h
UPD     /home/drbd-9.0.32-1/drbd/compat.4.9.212-36.el7.x86_64.h
CHK     /home/drbd-9.0.32-1/drbd/compat.h
UPD     /home/drbd-9.0.32-1/drbd/compat.h
make[4]: `drbd-kernel-compat/cocci_cache/a707960de8e44ee768894d4f66792d36/compat.patch' is up to date.
PATCH
patching file ./drbd_int.h
patching file ./drbd_req.h
patching file drbd-headers/linux/genl_magic_struct.h
patching file drbd_state.c
patching file drbd_receiver.c
patching file drbd_main.c
patching file drbd_nla.c
patching file drbd_nl.c
patching file drbd_bitmap.c
patching file drbd_sender.c
patching file drbd_transport_tcp.c
patching file drbd_actlog.c
patching file kref_debug.c
patching file drbd_req.c
patching file drbd_debugfs.c
patching file drbd-headers/linux/genl_magic_func.h
Hunk #2 succeeded at 312 (offset -20 lines).
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_debugfs.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_bitmap.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_proc.o
```

```
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_sender.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_receiver.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_req.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_actlog.o
  CC [M]  /home/drbd-9.0.32-1/drbd/lru_cache.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_main.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_strings.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_nl.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_interval.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_state.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd-kernel-compat/drbd_wrappers.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_nla.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_transport.o
  GEN     /home/drbd-9.0.32-1/drbd/drbd_buildtag.c
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_buildtag.o
  LD [M]  /home/drbd-9.0.32-1/drbd/drbd.o
  CC [M]  /home/drbd-9.0.32-1/drbd/drbd_transport_tcp.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /home/drbd-9.0.32-1/drbd/drbd.mod.o
  LD [M]  /home/drbd-9.0.32-1/drbd/drbd.ko
  CC      /home/drbd-9.0.32-1/drbd/drbd_transport_tcp.mod.o
  LD [M]  /home/drbd-9.0.32-1/drbd/drbd_transport_tcp.ko
  mv .drbd_kernelrelease.new .drbd_kernelrelease
  Memorizing module configuration ... done.
```

2. 创建目录 /lib/modules/<内核版本>/kernel/drivers/block/drbd/，内核版本根据实际情况替换

3. 把上一步编译后生成的 /home/drbd-9.0.32-1/drbd/drbd.ko 、/home/drbd-9.0.32-1/drbd/drbd_transport_tcp.ko 传至 /lib/modules/<内核版本>/kernel/drivers/block/drbd/

4. 配置 conf，创建目录把 global_common.conf 放在 /etc/drbd.d/ 目录下，drbd.conf 放在 /etc 目录下：

```bash title="global_common.conf" # DRBD is the result of over a decade of development by LINBIT. # In case you need professional services for DRBD or have # feature requests visit http://www.linbit.com
global { usage-count yes;
```

```
        # Decide what kind of udev symlinks you want for "implicit" volumes
          # (those without explicit volume <vnr> {} block, implied vnr=0):
          # /dev/drbd/by-resource/<resource>/<vnr>      (explicit volumes)
          # /dev/drbd/by-resource/<resource>             (default for implict)
          udev-always-use-vnr; # treat implicit the same as explicit volumes

          # minor-count dialog-refresh disable-ip-verification
          # cmd-timeout-short 5; cmd-timeout-medium 121; cmd-timeout-long 600;
}
common { handlers { # These are EXAMPLE handlers only. # They may have severe
  implications, # like hard resetting the node under certain circumstances. # Be
  careful when choosing your poison.
                # pri-on-incon-degr "/usr/lib/drbd/notify-pri-on-incon-degr.sh; /usr/li
  b/drbd/notify-emergency-reboot.sh; echo b > /proc/sysrq-trigger ; reboot -f";
                # pri-lost-after-sb "/usr/lib/drbd/notify-pri-lost-after-sb.sh; /usr/lib/
  drbd/notify-emergency-reboot.sh; echo b > /proc/sysrq-trigger ; reboot -f";
                # local-io-error "/usr/lib/drbd/notify-io-error.sh; /usr/lib/drbd/notify
  -emergency-shutdown.sh; echo o > /proc/sysrq-trigger ; halt -f";
                # fence-peer "/usr/lib/drbd/crm-fence-peer.sh";
                # split-brain "/usr/lib/drbd/notify-split-brain.sh root";
                # out-of-sync "/usr/lib/drbd/notify-out-of-sync.sh root";
                # before-resync-target "/usr/lib/drbd/snapshot-resync-target-lvm.sh
  -p 15 -- -c 16k";
                # after-resync-target /usr/lib/drbd/unsnapshot-resync-target-lvm.sh;
                # quorum-lost "/usr/lib/drbd/notify-quorum-lost.sh root";
        }

        startup {
                # wfc-timeout degr-wfc-timeout outdated-wfc-timeout wait-after-s
  b
        }

        options {
                # cpu-mask on-no-data-accessible

                # RECOMMENDED for three or more storage nodes with DR
  BD 9:
                # quorum majority;
                # on-no-quorum suspend-io | io-error;
        }

        disk {
                # size on-io-error fencing disk-barrier disk-flushes
                # disk-drain md-flushes resync-rate resync-after al-extents
```

```
                # c-plan-ahead c-delay-target c-fill-target c-max-rate
                # c-min-rate disk-timeout
        }

        net {
                # protocol timeout max-epoch-size max-buffers
                # connect-int ping-int sndbuf-size rcvbuf-size ko-count
                # allow-two-primaries cram-hmac-alg shared-secret after-sb-0pri
                # after-sb-1pri after-sb-2pri always-asbp rr-conflict
                # ping-timeout data-integrity-alg tcp-cork on-congestion
                # congestion-fill congestion-extents csums-alg verify-alg
                # use-rle
        }
}```
```bash title="drbd.conf" # You can find an example in /usr/share/doc/drbd../drbd.conf.example
include "drbd.d/global_common.conf"; include "drbd.d/*.res";
  ```
```

5. 加载内核

   ```
   insmod drbd.ko drbd_transport_tcp.ko
   ```

编译 drbd-tools

```
git clone -n https://github.com/LINBIT/drbd-utils.git
cd drbd-utils
git checkout v9.12.1
git clone -n https://github.com/LINBIT/drbd-headers.git
cd drbd-headers/
git checkout c757cf357edef67751b8f45a6ea894d287180087
cd ..

#安装依赖
yum install -y build-essential wget flex automake

./autogen.sh
./configure --prefix=/usr --localstatedir=/var --sysconfdir=/etc
make tools
find ./user -type f -executable -name 'drbd[a-z]*' -exec mv -v {} /usr/local/bin/ \;
```

进行验证

```
$ drbdadm -V

DRBDADM_BUILDTAG=GIT-hash: \ 6aec3180805d84f142f422013810af10cf4f95acdrbd-headers\
build\ by\ @buildkitsandbox\,\ 2022-10-12\ 10:40:34
DRBDADM_API_VERSION=2
DRBD_KERNEL_VERSION_CODE=0x090020
DRBD_KERNEL_VERSION=9.0.32
DRBDADM_VERSION_CODE=0x090c01
DRBDADM_VERSION=9.12.1
```

调度器

调度器 (scheduler) 是 HwameiStor 的重要组件之一。 它自动将 Pod 调度到配有 HwameiStor 存储卷的正确节点。 使用调度器后，Pod 不必再使用 NodeAffinity 或 NodeSelector 字段来选择节点。 调度器能处理 LVM 和 Disk 存储卷。

- 安装

  调度器应在集群中以 HA 模式部署，这是生产环境中的最佳实践。

- 通过 hwameistor-operator 安装

  hwameistor-operator 安装后会自动将 HwameiStor 相关组件拉起。 参阅通过 hwameistor-operator 安装 HwameiStor。

- 通过 YAML 部署（针对开发）

  ```
  kubectl apply -f deploy/scheduler.yaml
  ```

点击此处查看 scheduler.yaml

准入控制器

准入控制器是一种 webhook，可以自动验证 HwameiStor 数据卷，协助将 schedulerName

修改为 hwameistor-scheduler。 具体信息，请参见 Kubernetes 动态准入控制。

- 识别 HwameiStor 数据卷

  准入控制器可以获取 Pod 使用的所有 PVC，并检查每个 PVC 存储制备器。 如果制备器的名称后缀是 *.hwameistor.io，表示 Pod 正在使用 HwameiStor 提供的数据卷。

- 验证资源

  准入控制器只验证 Pod 资源，并在创建资源时就进行验证。

  !!! info
  为确保 HwameiStor 的 Pod 可以顺利启动，不会校验 HwameiStor 所在的命名空间下的 Pod。

驱逐器

驱逐器是 HwameiStor 系统运维的重要组件，能够保障 HwameiStor 在生产环境中持续正常运行。 当系统节点或者应用 Pod 由于各种原因被驱逐时，驱逐器会自动发现节点或者 Pod 所关联的 HwameiStor 数据卷， 并自动将其迁移到其他节点，从而保证被驱逐的 Pod 可以调度到其他节点上并正常运行。

在生产环境中，应该采用高可用模式部署驱逐器。

如何使用

请参考数据卷驱逐

指标采集器

指标采集器是 HwameiStor 的一个系统指标服务器。它服务采集系统资源的相关指标， 例如磁盘、数据卷、存储空间、数据卷操作等等，并将这些指标提供给 Prometheus 模块。该

模块由 Operator 进行部署。

资源

Hwameistor 在 Kubernetes 已有的 PV 和 PVC 对象类基础上，Hwameistor 定义了更丰富的对象类，把 PV/PVC 和本地数据盘关联起来。

名称	缩写	Kind	功能
clusters	hmcluster	Cluster	HwameiStor 集群
events	evt	Event	HwameiStor 集群的审计日志
localdiskclaims	ldc	LocalDiskClaim	筛选并分配本地数据盘
localdisknodes	ldn	LocalDiskNode	裸磁盘类型数据卷的存储节点
localdisks	ld	LocalDisk	节点上数据盘，自动识别空闲可用的数据盘
localdiskvolumes	ldv	LocalDiskVolume	裸磁盘类型数据卷
localstoragenodes	lsn	LocalStorageNode	LVM 类型数据卷的存储节点
localvolumeconverts	lvconvert	LocalVolumeConvert	将普通 LVM 类型数据卷转化为高可用

名称	缩写	Kind	功能
localvolumeexpands	lvexpand	LocalVolumeExpand	LVM 类型数据卷扩容 LVM 类型数据卷的容量
localvolumegroups	lvg	LocalVolumeGroup	LVM 类型数据卷组
localvolumemigrates	lvmigrate	LocalVolumeMigrate	迁移 LVM 类型数据卷
localvolumereplicas	lvr	LocalVolumeReplica	LVM 类型数据卷的副本
localvolumereplicasnapshotrestores	lvrsrestore,lvrsnaprestore	LocalVolumeReplicaSnapshotRestore	恢复 LVM 类型数据卷副本的快照
localvolumereplicasnapshots	lvrs	LocalVolumeReplicaSnapshot	LVM 类型数据卷副本的快照
localvolumes	lv	LocalVolume	LVM 类型数据卷
localvolumesnapshotrestores	lvsrestore,lvsnaprestore	LocalVolumeSnapshotRestore	恢复 LVM 类型数据卷快照
localvolumesnapshots	lvs	LocalVolumeSnapshot	LVM 类型数据卷快照 volume
resizepolicies		ResizePolicy	PVC 自动扩容策略

如有需要，可以使用以下命令查看 api-resources :

```
$ kubectl api-resources --api-group=hwameistor.io

NAME                                SHORTNAMES                APIVERSION
NAMESPACED   KIND
```

clusters hmcluster hwameistor.io/v1alpha1 false Cluster
events evt hwameistor.io/v1alpha1 false Event
localdiskactions lda hwameistor.io/v1alpha1 false LocalDiskAction
localdiskclaims ldc hwameistor.io/v1alpha1 false LocalDiskClaim
localdisknodes ldn hwameistor.io/v1alpha1 false LocalDiskNode
localdisks ld hwameistor.io/v1alpha1 false LocalDisk
localdiskvolumes ldv hwameistor.io/v1alpha1 false LocalDiskVolume
localstoragenodes lsn hwameistor.io/v1alpha1 false LocalStorageNode
localvolumeconverts lvconvert hwameistor.io/v1alpha1 false LocalVolumeConvert
localvolumeexpands lvexpand hwameistor.io/v1alpha1 false LocalVolumeExpand
localvolumegroups lvg hwameistor.io/v1alpha1 false LocalVolumeGroup
localvolumemigrates lvmigrate hwameistor.io/v1alpha1 false LocalVolumeMigrate
localvolumereplicas lvr hwameistor.io/v1alpha1 false LocalVolumeReplica
localvolumereplicasnapshotrestores lvrsrestore,lvrsnaprestore hwameistor.io/v1alpha1 false LocalVolumeReplicaSnapshotRestore
localvolumereplicasnapshots lvrs hwameistor.io/v1alpha1 false LocalVolumeReplicaSnapshot
localvolumes lv hwameistor.io/v1alpha1 false LocalVolume
localvolumesnapshotrestores lvsrestore,lvsnaprestore hwameistor.io/v1alpha1 false LocalVolumeSnapshotRestore
localvolumesnapshots lvs hwameistor.io/v1alpha1 false LocalVolumeSnapshot
resizepolicies hwameistor.io/v1alpha1 false ResizePolicy

准备工作

平台准备

- Kubernetes 容器平台版本兼容性：

Kubernetes	Hwameistor V0.4.3	Hwameistor >=v0.5.0	Hwameistor >= 0.13.0
>=1.18&&<=1.20	是	否	否
1.21	是	是	否
1.22	是	是	否
1.23	是	是	否
1.24	是	是	是
1.25	否	是	是
1.26	否	是	是
1.27	否	否	是
1.28	否	否	是

- 已部署 CoreDNS

- 高可用功能需要安装和当前运行的 Kernel 版本一致的 kernel-devel，可通过命令检查：

```
uname -r
    3.10.0-1160.el7.x86_64
    yum list installed |grep kernel
    kernel.x86_64                    3.10.0-1160.el7          @anaconda

    kernel-tools.x86_64              3.10.0-1160.el7          @anaconda
    kernel-tools-libs.x86_64         3.10.0-1160.el7          @anaconda
```

如不一致，可通过如下方式安装：

```
yum install -y kernel-devel-$(uname -r) # 安装 kernel-devel
```

- 已安装 LVM2，如未安装请参考如下安装方式:

=== “CentOS/RHEL、Rocky 和 Kylin”

```console
yum install -y lvm2
```

=== “Ubuntu”

```console
apt-get install -y lvm2
apt-get install -y linux-headers-$(uname -r)
```

支持的操作系统

架构	支持操作系统	推荐
AMD 64	centos 7.4+	操作系统推荐 CentOS 7.9
	Redhat 8.4+	操作系统推荐 Redhat 8.4
	Redhat 7.4+	操作系统推荐 Redhat 7.9
	Ubuntu 19+	操作系统推荐 Ubuntu 20.04
ARM 64	银河麒麟 OS V10 SP2	银河麒麟 OS V10 SP2

Secure Boot

高可用功能暂时不支持 Secure Boot，确认 Secure Boot 是 disabled 状态：

```
$ mokutil --sb-state
SecureBoot disabled

$ dmesg | grep secureboot
[ 0.000000] secureboot: Secure boot disabled
```

磁盘类型

HwameiStor 支持物理硬盘 (HDD)、固态硬盘 (SSD) 和 NVMe 闪存盘.

测试环境里，每个主机必须要有至少一块空闲的 10GiB 数据盘。

生产环境里，建议每个主机至少要有一块空闲的 200GiB 数据盘，而且建议使用固态硬盘 (SSD)。

网络规划

生产环境里，开启高可用模式后，建议使用有冗余保护的万兆 TCP/IP 网络。 可通过修改网卡的方式指定网卡 IP 提前进行规划。

生产环境资源要求

如果在生产环境中部署 Hwameistor，请指定资源配置，并且避免部署到 Master 节点上，因此 values.extra.prod.yaml 文件中提供了一些推荐值，资源配置如下：

??? note “点击查看 values.extra.prod.yaml”

```yaml title="values.extra.prod.yaml"
scheduler:
  replicas: 3
  resources:
    limits:
      cpu: 300m
      memory: 300Mi
    requests:
      cpu: 1m
      memory: 20Mi

admission:
  replicas: 3
  resources:
    limits:
      cpu: 300m
      memory: 300Mi
    requests:
      cpu: 1m
      memory: 20Mi

evictor:
```

```
  replicas: 2
  resources:
    limits:
      cpu: 300m
      memory: 300Mi
    requests:
      cpu: 1m
      memory: 20Mi

metrics:
  replicas: 2
  resources:
    limits:
      cpu: 300m
      memory: 300Mi
    requests:
      cpu: 1m
      memory: 20Mi

apiserver:
  replicas: 2
  resources:
    limits:
      cpu: 300m
      memory: 300Mi
    requests:
      cpu: 1m
      memory: 20Mi

localDiskManager:
  tolerationsOnMaster: false
  registrar:
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  manager:
    resources:
      limits:
        cpu: 300m
        memory: 300Mi
```

```
        requests:
          cpu:  1m
          memory:  20Mi

localDiskManagerCSIController:
  replicas:  3
  priorityClassName:  system-node-critical
  provisioner:
    resources:
      limits:
        cpu:  500m
        memory:  500Mi
      requests:
        cpu:  1m
        memory:  20Mi
  attacher:
    resources:
      limits:
        cpu:  500m
        memory:  500Mi
      requests:
        cpu:  1m
        memory:  20Mi

localStorage:
  tolerationsOnMaster:  false
  priorityClassName:  system-node-critical
  registrar:
    resources:
      limits:
        cpu:  500m
        memory:  500Mi
      requests:
        cpu:  1m
        memory:  20Mi
  member:
    resources:
      limits:
        cpu:  500m
        memory:  500Mi
      requests:
        cpu:  1m
        memory:  20Mi
```

```
localStorageCSIController:
  replicas: 3
  priorityClassName: system-node-critical
  provisioner:
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  attacher:
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  resizer:
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
```

1. 如通过 Helm 创建时可通过如下方式修改资源：

```
helm install hwameistor ./hwameistor \
    -n hwameistor --create-namespace \
    -f ./hwameistor/values.yaml \
    -f ./hwameistor/values.extra.prod.yaml
```

2. 如通过 UI 界面安装，请手动将如上资源通过 YAML 中的 Resource 值进行配置，否则默认不配置：

pro-Resource

pro-Resource

通过界面安装 HwameiStor

本文介绍如何通过平台界面安装 HwameiStor。

前提条件

- 待使用节点上已准备空闲 HDD、SSD 磁盘
- 已完成准备工作中事项
- 如需要使用高可用数据卷，请提前安装好 DRBD
- 如部署环境为生产环境，请提前阅读生产环境资源要求
- 如果您的 Kubernetes 发行版使用不同的 kubelet 目录，请提前确认 kubeletRootDir。

  详细信息请参考自定义 kubelet 根目录。

安装步骤

请确认您的集群已成功接入 **容器管理** 平台，然后执行以下步骤安装 HwameiStor。

1. 在左侧导航栏点击 **容器管理** —> **集群列表** ，然后找到准备安装 HwameiStor 的集群名称。
2. 在左侧导航栏中选择 **Helm 应用** -> **Helm 模板**，找到并点击 **HwameiStor** 。

   UI Install01

   UI Install01
3. 在 **版本** 中选择希望安装的版本，点击 **安装** 。
4. 在安装界面，填写所需的安装参数。如需要部署生产环境，建议调整资源配置：生产环境资源要求。

   HwameistorUI02

   HwameistorUI02

   HwameistorUI03

HwameistorUI03

- Global Setting —> global image Registry：

  设置所有镜像的仓库地址，默认已经填写了可用的在线仓库。 如果是私有化环境，可修改为私有仓库地址。

- Global Setting —> K8s image Registry：

  设置 K8s 镜像仓库地址，默认已经填写可用在线仓库。 如果私有化环境，可修改为私有仓库地址。

- Global Setting —> kubelet Root Dir：

  默认的 kubelet 目录为 /var/lib/kubelet。 如果您的 Kubernetes 发行版使用不同的 kubelet 目录，必须设置参数 kubeletRootDir。 详细信息请参考自定义 kubelet 根目录。

- Config Settings —> DRBDStartPort：

  默认以 43001 开始，当开启 DRBD 时，每创建一个高可用数据卷，需要占用主副本数据卷所在节点的一组端口。 请先安装好 DRBD。

- **StorageClass 配置**

  HwameiStor 部署后，会自动创建 StorageClass。

  - AllowVolumeExpansion：默认为关闭状态，开启后，基于 StorageClass 创建的数据卷可以扩容。
  - DiskType：创建的存储池（StorageClass）的磁盘类型，支持类型有：HDD、SSD。默认为 HDD。
  - Enable HA：默认关闭 HA，即创建的数据卷为非高可用，当开启后，使用该 StorageClass 创建的数据卷可以设置为高可用数据卷。请在开启前完成 DRBD 安装。

- Replicas：非 HA 模式下，Replicas 数量为 1；当开启 HA 模式后，Replicas 数量可以为 1 或者 2，并且数量为 1 时，可以转换为 2。
- ReclaimPolicy：数据卷删除时，数据的保留策略默认为 delete。
  1. Delete：当删除数据卷时，数据一并删除。
  2. Retain：当删除数据卷时，保留数据。

5. 参数输入完成后，点击 **确定** 完成创建，完成创建后可点击 **Helm 应用** 查看 **HwameiStor** 安装状态。

HwameistorUI04

HwameistorUI04

6. 安装完成！要验证安装效果，请参见下一章安装后检查。

通过 hwameistor-operator 安装 HwameiStor

本文介绍在平台界面通过 hwameistor-operator 安装 HwameiStor，operator 安装后会自动将 HwameiStor 相关组件拉起。hwameistor-operator 负责如下事项：

- 所有组件的全生命周期管理 (LCM)：
  - LocalDiskManager
  - LocalStorage
  - Scheduler
  - AdmissionController
  - VolumeEvictor
  - Exporter
  - Apiserver
  - Graph UI
- 根据不同目的和用途配置节点磁盘
- 自动发现节点磁盘的类型，并以此自动创建 HwameiStor 存储池

- 根据 HwameiStor 系统的配置和功能自动创建相应的 StorageClass

前提条件

- 待使用节点上已准备空闲 HDD、SSD 磁盘。
- 已完成准备工作中事项。
- 如部署环境为生产环境，请提前阅读生产环境资源要求。
- 如果您的 Kubernetes 发行版使用不同的 kubelet 目录，请提前确认 kubeletRootDir。

  详细信息请参考自定义 kubelet 根目录。

!!! info
如果没有可用的干净磁盘，operator 不会自动创建 StorageClass。
operator 会在安装过程中自动纳管磁盘，可用的磁盘会被添加到 LocalStorage 的 pool 里。
如果可用磁盘是在安装后提供的，则需要手动下发 LocalDiskClaim 将磁盘纳管到 LocalStorage Node 里。
一旦 LocalStorageNode 的 pool 里有磁盘，operator 就会自动创建 StorageClass。
也就是说，如果没有容量，就不会自动创建 StorageClass。

安装步骤

请确认您的集群已成功接入 **容器管理** 平台，然后执行以下步骤安装 HwameiStor。

1. 在左侧导航栏点击 **容器管理** —> **集群列表**，然后找到准备安装 HwameiStor 的集群名称。
2. 在左侧导航栏中选择 **Helm 应用** -> **Helm 模板** ，找到并点击 **hwameistor-operator** 。

   点击

   点击
3. 在 **版本** 中选择希望安装的版本，点击 **安装** 。
4. 在安装界面，填写所需的安装参数。

   Operator02

   Operator02

Value.yaml 参数如下，默认可不进行修改：

```yaml
global:
    targetNamespace: hwameistor
    notClaimDisk: false
    hwameistorImageRegistry: ghcr.m.daocloud.io
    k8sImageRegistry: k8s.m.daocloud.io
    hwameistorVersion: v0.14.3
  operator:
    replicas: 1
    imageRepository: hwameistor/operator
    tag: v0.14.6
  localDiskManager:
    tolerationOnMaster: true
    kubeletRootDir: /var/lib/kubelet
    manager:
      imageRepository: hwameistor/local-disk-manager
      tag: v0.14.3
    csi:
      registrar:
        imageRepository: sig-storage/csi-node-driver-registrar
        tag: v2.5.0
        resources:
          limits:
            cpu: 500m
            memory: 500Mi
          requests:
            cpu: 1m
            memory: 20Mi
      controller:
        replicas: 1
      provisioner:
        imageRepository: sig-storage/csi-provisioner
        tag: v2.0.3
        resources:
          limits:
            cpu: 500m
            memory: 500Mi
          requests:
            cpu: 1m
            memory: 20Mi
      attacher:
        imageRepository: sig-storage/csi-attacher
        tag: v3.0.1
```

```
        resources:
          limits:
            cpu: 500m
            memory: 500Mi
          requests:
            cpu: 1m
            memory: 20Mi
    localStorage:
      disable: false
      tolerationOnMaster: true
      kubeletRootDir: /var/lib/kubelet
      member:
        imageRepository: hwameistor/local-storage
        tag: v0.14.3
        hostPathSSHDir: /root/.ssh
        hostPathDRBDDir: /etc/drbd.d
      csi:
        registrar:
          imageRepository: sig-storage/csi-node-driver-registrar
          tag: v2.5.0
          resources:
            limits:
              cpu: 500m
              memory: 500Mi
            requests:
              cpu: 1m
              memory: 20Mi
        controller:
          replicas: 1
        provisioner:
          imageRepository: sig-storage/csi-provisioner
          tag: v3.5.0
          resources:
            limits:
              cpu: 500m
              memory: 500Mi
            requests:
              cpu: 1m
              memory: 20Mi
        attacher:
          imageRepository: sig-storage/csi-attacher
          tag: v3.0.1
          resources:
            limits:
```

```
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  resizer:
    imageRepository: sig-storage/csi-resizer
    tag: v1.0.1
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  monitor:
    imageRepository: sig-storage/csi-external-health-monitor-controller
    tag: v0.8.0
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  snapshotController:
    imageRepository: sig-storage/snapshot-controller
    tag: v6.0.0
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
        memory: 20Mi
  snapshotter:
    imageRepository: sig-storage/csi-snapshotter
    tag: v6.0.0
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 1m
```

```
            memory: 20Mi
    migrate:
      rclone:
        imageRepository: rclone/rclone
        tag: 1.53.2
      juicesync:
        imageRepository: hwameistor/hwameistor-juicesync
        tag: v1.0.4-01
    snapshot:
      disable: false
  scheduler:
    disable: false
    replicas: 1
    imageRepository: hwameistor/scheduler
    tag: v0.14.3
  admission:
    disable: false
    replicas: 1
    imageRepository: hwameistor/admission
    tag: v0.14.3
    failurePolicy: Ignore
  evictor:
    disable: true
    replicas: 0
    imageRepository: hwameistor/evictor
    tag: v0.14.3
  apiserver:
    disable: false
    replicas: 1
    imageRepository: hwameistor/apiserver
    tag: v0.14.3
    authentication:
      enable: false
      accessId: admin
      secretKey: admin
  exporter:
    disable: false
    replicas: 1
    imageRepository: hwameistor/exporter
    tag: v0.14.3
  auditor:
    disable: false
    replicas: 1
    imageRepository: hwameistor/auditor
```

```
    tag: v0.14.3
  failoverAssistant:
    disable: false
    replicas: 1
    imageRepository: hwameistor/failover-assistant
    tag: v0.14.3
  pvcAutoResizer:
    disable: false
    replicas: 1
    imageRepository: hwameistor/pvc-autoresizer
    tag: v0.14.3
  localDiskActionController:
    disable: false
    replicas: 1
    imageRepository: hwameistor/local-disk-action-controller
    tag: v0.14.3
  ui:
    disable: false
    replicas: 1
    imageRepository: hwameistor/hwameistor-ui
    tag: v0.16.0
  ha:
    disable: false
    module: drbd
    deployOnMaster: 'yes'
    imageRepository: hwameistor/drbd9-shipper
    drbdVersion: v9.0.32-1
    shipperChar: v0.4.1
  drbdRhel7:
    imageRepository: hwameistor/drbd9-rhel7
  drbdRhel8:
    imageRepository: hwameistor/drbd9-rhel8
  drbdRhel9:
    imageRepository: hwameistor/drbd9-rhel9
  drbdKylin10:
    imageRepository: hwameistor/drbd9-kylin10
  drbdBionic:
    imageRepository: hwameistor/drbd9-bionic
  drbdFocal:
    imageRepository: hwameistor/drbd9-focal
  preHookJob:
    imageRepository: dtzar/helm-kubectl
    tag: 3.9
```

- hwameistorImageRegistry：

设置 HwameiStor 镜像的仓库地址，默认已经填写了可用的在线仓库。 如果是私有化环境，可修改为私有仓库地址。

- K8s image Registry：

  设置 K8s 镜像仓库地址，默认已经填写可用在线仓库。 如果私有化环境，可修改为私有仓库地址。

- DRBD：

  如果需要使用高可用数据卷，请开启 DRBD 模块，如安装时未开启，请参考开启 DRBD。

- replicas：

  各组件副本数量，建议使用 `2` 副本。

5. 确认参数无误后，点击 **确定** 完成安装，完成安装后可点击 **Helm 应用** 查看 **hwameistor-operator** 安装状态。

Operator03

Operator03

6. operator 安装完成后，HwameiStor 组件（Local Storage、Local Disk Manager 等）默认进行安装！ 可点击 **工作负载** -> **无状态工作负载** ，选择对应命名空间，查看 HwameiStor 组件状态。

HwameiStor 状态

HwameiStor 状态

通过命令行验证安装效果，请参安装后检查。

查看安装系统的状态

此处，以三个节点的 kubernetes 集群为例，安装 HwameiStor 存储系统。

```
$ kubectl get node
NAME              STATUS     ROLES     AGE     VERSION
10-6-234-40     Ready     control-plane,master     140d     v1.21.11
10-6-234-41     Ready     <none>                   140d     v1.21.11
10-6-234-42     Ready     <none>                   140d     v1.21.11
```

查看 HwameiStor 系统组件

以下 Pod 必须在系统中正常运行:

```
$ kubectl -n hwameistor get pod
NAME                                                READY     STATUS
RESTARTS          AGE
drbd-adapter-k8s-master-rhel7-gtk7t                 0/2       Completed  0
   23m
drbd-adapter-k8s-node1-rhel7-gxfw5                  0/2       Completed  0
      23m
drbd-adapter-k8s-node2-rhel7-lv768                  0/2       Completed  0
     23m
hwameistor-admission-controller-dc766f976-mtlvw     1/1       Running    0
   23m
hwameistor-apiserver-86d6c9b7c8-v67gg               1/1       Running    0
      23m
hwameistor-auditor-54f46fcbc6-jb4f4                 1/1       Running    0
     23m
hwameistor-exporter-6498478c57-kr8r4                1/1       Running    0
      23m
hwameistor-failover-assistant-cdc6bd665-56wbw       1/1       Running    0
     23m
hwameistor-local-disk-csi-controller-6587984795-fztcd   2/2       Running    0                  2
3m
hwameistor-local-disk-manager-7gg9x                 2/2       Running    0
      23m
hwameistor-local-disk-manager-kqkng                 2/2       Running    0
      23m
hwameistor-local-disk-manager-s66kn                 2/2       Running    0
      23m
hwameistor-local-storage-csi-controller-5cdff98f55-jj45w   6/6       Running    0                23
m
hwameistor-local-storage-mfqks                      2/2       Running    0
      23m
hwameistor-local-storage-pnfpx                      2/2       Running    0
      23m
hwameistor-local-storage-whg9t                      2/2       Running    0
```

```
      23m
hwameistor-pvc-autoresizer-86dc79d57-s2l68                1/1     Running   0
    23m
hwameistor-scheduler-6db69957f-r58j6                      1/1     Running   0
      23m
hwameistor-ui-744cd78d84-vktjq                            1/1     Running   0
       23m
hwameistor-volume-evictor-5db99cf979-4674n                1/1     Running   0
      23m
```

local-disk-manager 和 local-storage 组件是以 DaemonSets 方式进行部署的，必须在每个节点上运行。

查看 HwameiStor CRD（即 API）

以下 HwameiStor CRD 必须安装在系统上。

```
$ kubectl api-resources --api-group hwameistor.io
NAME                  SHORTNAMES   APIVERSION                NAMESPACED   KIND
clusters              hmcluster    hwameistor.io/v1alpha1    false        Cluster
events                evt          hwameistor.io/v1alpha1    false        Event
localdiskclaims       ldc          hwameistor.io/v1alpha1    false        LocalDiskClaim
localdisknodes        ldn          hwameistor.io/v1alpha1    false        LocalDiskNode
localdisks            ld           hwameistor.io/v1alpha1    false        LocalDisk
localdiskvolumes      ldv          hwameistor.io/v1alpha1    false        LocalDiskVolume
localstoragenodes     lsn          hwameistor.io/v1alpha1    false        LocalStorageNode
localvolumeconverts   lvconvert    hwameistor.io/v1alpha1    false        LocalVolumeConvert
localvolumeexpands    lvexpand     hwameistor.io/v1alpha1    false        LocalVolumeExpand
localvolumegroups     lvg          hwameistor.io/v1alpha1    false        LocalVolumeGroup
localvolumemigrates   lvmigrate    hwameistor.io/v1alpha1    false        LocalVolumeMigrate
localvolumereplicas   lvr          hwameistor.io/v1alpha1
```

false LocalVolumeReplica
localvolumereplicasnapshotrestores lvrsrestore,lvrsnaprestore hwameistor.io/v1alpha1 false
LocalVolumeReplicaSnapshotRestore
localvolumereplicasnapshots lvrs hwameistor.io/v1alpha1
false LocalVolumeReplicaSnapshot
localvolumes lv hwameistor.io/v1alpha
1 false LocalVolume
localvolumesnapshotrestores lvsrestore,lvsnaprestore hwameistor.io/v1alpha1 fals
e LocalVolumeSnapshotRestore
localvolumesnapshots lvs hwameistor.io/v1alpha1
false LocalVolumeSnapshot
resizepolicies hwameistor.io/v1alpha1
false ResizePolicy

详情参见 CRD 表。

查看 LocalDiskNodes 和 LocalDisks

HwameiStor 自动扫描每个节点上的磁盘，并为每一块磁盘生成一个 CRD 资源 LocalDisk (LD)。 没有被使用的磁盘，其状态被标记为 PHASE: Available。

```
$ kubectl get localdisknodes
NAME FREECAPACITY TOTALCAPACITY TOTALDISK STATUS AGE
k8s-master Ready 28h
k8s-node1 Ready 28h
k8s-node2 Ready 28h

$ kubectl get localdisks
NAME NODEMATCH DEVICEPATH PH
ASE AGE
localdisk-2307de2b1c5b5d051058bc1d54b41d5c k8s-node1 /dev/sdb Available 28h
localdisk-311191645ea00c62277fe709badc244e k8s-node2 /dev/sdb Available 28h
localdisk-37a20db051af3a53a1c4e27f7616369a k8s-master /dev/sdb Available 28h
localdisk-b57b108ad2ccc47f4b4fab6f0b9eaeb5 k8s-node2 /dev/sda Bound 28h
localdisk-b682686c65667763bda58e391fbb5d20 k8s-master /dev/sda Bound 28h
localdisk-da121e8f0dabac9ee1bcb6ed69840d7b k8s-node1 /dev/sda Bound 28h
```

查看 LocalStorageNodes 及存储池

HwameiStor 为每个存储节点创建一个 CRD 资源 LocalStorageNode (LSN)。 每个 LSN 将会

记录该存储节点的状态，及节点上的所有存储资源，包括存储池、数据卷、及相关配置信息。

```
$ kubectl get lsn
NAME            IP              STATUS   AGE
10-6-234-40     10.6.234.40     Ready    3m52s
10-6-234-41     10.6.234.41     Ready    3m54s
10-6-234-42     10.6.234.42     Ready    3m55s

$ kubectl get lsn 10-6-234-41 -o yaml
apiVersion: hwameistor.io/v1alpha1
kind: LocalStorageNode
metadata:
  creationTimestamp: "2023-04-11T06:46:52Z"
  generation: 1
  name: 10-6-234-41
  resourceVersion: "13575433"
  uid: 4986f7b8-6fe1-43f1-bdca-e68b6fa53f92
spec:
  hostname: 10-6-234-41
  storageIP: 10.6.234.41
  topogoly:
    region: default
    zone: default
status:
  pools:
    LocalStorage_PoolHDD:
      class: HDD
      disks:
      - capacityBytes: 10733223936
        devPath: /dev/sdb
        state: InUse
        type: HDD
      - capacityBytes: 1069547520
        devPath: /dev/sdc
        state: InUse
        type: HDD
      - capacityBytes: 1069547520
        devPath: /dev/sdd
        state: InUse
        type: HDD
      - capacityBytes: 1069547520
        devPath: /dev/sde
        state: InUse
        type: HDD
```

```
    - capacityBytes: 1069547520
      devPath: /dev/sdf
      state: InUse
      type: HDD
    - capacityBytes: 1069547520
      devPath: /dev/sdg
      state: InUse
      type: HDD
    freeCapacityBytes: 16080961536
    freeVolumeCount: 1000
    name: LocalStorage_PoolHDD
    totalCapacityBytes: 16080961536
    totalVolumeCount: 1000
    type: REGULAR
    usedCapacityBytes: 0
    usedVolumeCount: 0
    volumeCapacityBytesLimit: 16080961536
  state: Ready
```

查看 StorageClass

HwameiStor Operator 在完成 HwameiStor 系统组件安装和系统初始化之后，会根据系统配置 （例如：是否开启 HA 模块、磁盘类型等）自动创建相应的 StorageClass 用于创建数据卷。

```
$ kubectl get sc
NAME                                PROVISIONER               RECLAIMPOLICY
  VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
hwameistor-storage-lvm-hdd          lvm.hwameistor.io   Delete          WaitForFirst
Consumer   false          23h
hwameistor-storage-lvm-hdd-convertible   lvm.hwameistor.io   Delete          WaitForFirstCon
sumer   false          23h
hwameistor-storage-lvm-hdd-ha       lvm.hwameistor.io   Delete          WaitForFirstC
onsumer   false          23h
```

存储网卡配置

HwameiStor 支持使用单独的网卡进行数据卷同步，可以避免使用通信网卡带来流量阻塞问

题。

!!! note
- 【🔥提前配置】：存储网卡的配置属于存储系统的前置性配置，建议在 HwameiStor 系统安装前 **提前配置**。
- 【运行中配置】：如果 HwameiStor 已经部署，后面进行上述配置修改，那么**对之前已经创建出来的数据卷不会生效**，也就是还会使用之前的网卡进行数据卷同步。
- 如需要修改多个节点存储网卡，请逐个配置，目前无法批量配置

前提条件

已经提前完成存储网卡规划，请参考网卡规划。

配置步骤

配置方式分为 2 种：

1. 通过 LocalStorage CR 配置
2. 通过节点注释标记

修改 LocalStorage CR 配置

1. 在左侧导航栏点击 容器管理 —> 集群列表，找到待修改网卡配置的集群，进入集群详情 。
2. 在左侧导航栏中选择 自定义资源，找到 localstoragenodes.hwameistor.io，点击进入详情。

Ethedit01

Ethedit01

3. 找到待修改节点并点击编辑 YAML，修改 spec 中的 storage-ipv4=172.30.40.12 参数，指定 IP 地址为已规划网卡 IP 网卡规划。

ethedit02

ethedit02

ethedit03

ethedit03

4. 完成后点击保存，并选择下一个节点进行修改。

通过节点注释标记

1. 查看 local-storage 的 ENV: **NODE_ANNOTATION_KEY_STORAGE_IPV4** 的值，默认是 **localstorage.hwameistor.io/storage-ipv4**
2. 将节点上的 **存储网卡地址通过注释的方式标记**

```
kubectl annotate node <your_storage_node> localstorage.hwameistor.io``/storage-ipv4``=172.30.46.12
```

3. **重启** 节点上的 local-storage 服务
4. **验证** 配置是否生效

```
kubectl get lsn <your_storage_node> -o yaml |``grep` `-i storageIP
```

5. 修改成功后，进行下一个节点修改。

自定义 Kubelet 根目录

!!! warning
默认的 `kubelet` 目录为 `/var/lib/kubelet`。
如果您的 Kubernetes 发行版使用不同的 `kubelet` 目录，必须设置参数 `kubeletRootDir`。

例如，在将 /var/snap/microk8s/common/var/lib/kubelet/ 用作 kubelet 目录的 Canonical 的 MicroK8s 上。

1. 通过界面操作，HwameiStor 需要需要此修改参数界面中 Global Setting —> Kubelet Root Dir 参数，并将参数设置为 /var/snap/microk8s/common/var/lib/kubelet/。详情请参考通过界面创建
2. 如果通过 Helm 创建，请执行如下安装命令。

```
helm install hwameistor ./hwameistor \
        -n hwameistor --create-namespace \
        --set kubeletRootDir=/var/snap/microk8s/common/var/lib/kubelet/
```

升级

Helm 让 HwameiStor 的升级变得非常简单。只需运行以下命令：

```
helm upgrade -n hwameistor hwameistor -f new.values.yaml
```

升级过程中将以滚动的方式重启每个 HwameiStor Pod。

!!! warning
在升级 HwameiStor 期间，这些卷将继续不间断地为 Pod 服务。

卸载 Hwameistor

本章节介绍了两种卸载 HwameiStor 系统的方式。

!!! danger
请务必先备份好所有数据，再卸载 HwameiStor。

方式一：卸载并保留已有数据

如果想要卸载 HwameiStor 的系统组件，但是保留已经创建的数据卷并服务于数据应用，采用下列方式：

```
$ kubectl get cluster.hwameistor.io
NAME                AGE
cluster-sample      21m

$ kubectl delete clusters.hwameistor.io  hwameistor-cluster
```

最终，所有的 HwameiStor 系统组件（Pods）将被删除。用下列命令检查，结果为空。

```
kubectl -n hwameistor get pod
```

方式二：完全卸载

如果想要卸载 HwameiStor 所有组件，并删除所有数据卷及数据，采用下列方式，请谨慎操作。

1. 清理有状态数据应用

    1. 删除应用
    2. 删除数据卷 PVC

       相关的 PV、LV、LVR、LVG 都将被删除。

2. 清理 HwameiStor 系统组件

    1. 删除 HwameiStor 组件

       ```
       kubectl delete clusters.hwameistor.io hwameistor-cluster
       ```

    2. 删除 HwameiStor 系统空间

       ```
       kubectl delete ns hwameistor
       ```

    3. 删除 CRD、Hook 以及 RBAC

       ```
       kubectl get crd,mutatingwebhookconfiguration,clusterrolebinding,clusterrole -o nam
         e \
           | grep hwameistor \
           | xargs -t kubectl delete
       ```

    4. 删除 StorageClass

       ```
       kubectl get sc -o name \
           | grep hwameistor-storage-lvm- \
           | xargs -t kubectl delete
       ```

    5. 删除 hwameistor Operator

       ```
       helm uninstall hwameistor-operator -n hwameistor
       ```

最后，你仍然需要清理每个节点上的 LVM 配置，并采用额外的系统工具（例如 wipefs）清除磁盘上的所有数据。

LVM 节点扩展

存储系统可以通过增加存储节点实现扩容。在 HwameiStor 里，通过下列步骤可以添加新的存储节点。

步骤

1. 准备新的存储节点

在 Kubernetes 集群中新增一个节点，或者，选择一个已有的集群节点（非 HwameiStor 节点）。该节点必须满足 准备工作 要求的所有条件。 本例中，所用的新增存储节点和磁盘信息如下所示：

- name: k8s-worker-4
- devPath: /dev/sdb
- diskType: SSD disk

新增节点已经成功加入 Kubernetes 集群之后，检查并确保下列 Pod 正常运行在该节点上，以及相关资源存在于集群中：

```
kubectl get node
NAME STATUS ROLES AGE VERSION
k8s-master-1 Ready master 96d v1.24.3-2+63243a96d1c393
k8s-worker-1 Ready worker 96h v1.24.3-2+63243a96d1c393
k8s-worker-2 Ready worker 96h v1.24.3-2+63243a96d1c393
k8s-worker-3 Ready worker 96d v1.24.3-2+63243a96d1c393
k8s-worker-4 Ready worker 1h v1.24.3-2+63243a96d1c393
kubectl -n hwameistor get pod -o wide | grep k8s-worker-4
hwameistor-local-disk-manager-c86g5 2/2 Running 0 19h 10.6.182.105 k
8s-worker-4 <none> <none>
hwameistor-local-storage-s4zbw 2/2 Running 0 19h 192.168.140.82 k
8s-worker-4 <none> <none>
```

检查 LocalStorageNode 资源

```
kubectl get localstoragenode k8s-worker-4
NAME IP ZONE REGION STATUS AGE
k8s-worker-4 10.6.182.103 default default Ready 8d
```

2. 添加新增存储节点到 HwameiStor 系统

为增加存储节点创建资源 LocalStorageClaim，以此为新增存储节点构建存储池。这样，节点就已经成功加入 HwameiStor 系统。具体如下：

```
$ kubectl apply -f - <<EOF
apiVersion: hwameistor.io/v1alpha1
kind: LocalDiskClaim
metadata:
  name: k8s-worker-4
spec:
  nodeName: k8s-worker-4
  owner: local-storage
  description:
    diskType: SSD # 可能需要根据节点磁盘实际情况调整为 HDD 或者 NVMe
EOF
```

3. 后续检查

完成上述步骤后，检查新增存储节点及其存储池的状态，确保节点和 HwameiStor 系统的正常运行。具体如下：

```
kubectl get localstoragenode k8s-worker-4
apiVersion: hwameistor.io/v1alpha1
kind: LocalStorageNode
metadata:
  name: k8s-worker-4
spec:
  hostname: k8s-worker-4
  storageIP: 10.6.182.103
  topogoly:
    region: default
    zone: default
status:
  pools:
    LocalStorage_PoolSSD:
      class: SSD
      disks:
        - capacityBytes: 214744170496
          devPath: /dev/sdb
          state: InUse
          type: SSD
      freeCapacityBytes: 214744170496
      freeVolumeCount: 1000
      name: LocalStorage_PoolSSD
      totalCapacityBytes: 214744170496
      totalVolumeCount: 1000
      type: REGULAR
```

```
    usedCapacityBytes: 0
    usedVolumeCount: 0
    volumeCapacityBytesLimit: 214744170496
    volumes:
  state: Ready
```

裸磁盘节点扩容

裸磁盘存储节点为应用提供裸磁盘类型的数据卷，并且维护了该存储节点上面的裸磁盘和裸磁盘数据卷的对应关系。

步骤

1. 准备新的存储节点

在 Kubernetes 集群中新增一个节点，或者，选择一个已有的集群节点（非 HwameiStor 节点）。该节点必须满足 Prerequisites 要求的所有条件。 本例中，所用的新增存储节点和磁盘信息如下所示：

- name: k8s-worker-2
- devPath: /dev/sdb
- diskType: SSD disk

新增节点已经成功加入 Kubernetes 集群之后，检查并确保下列 Pod 正常运行在该节点上，以及相关资源存在于集群中：

```
$ kubectl get node
NAME           STATUS   ROLES                  AGE     VERSION
k8s-master-1   Ready    master                 96d     v1.24.3-2+63243a96d1c393
k8s-worker-1   Ready    worker                 96h     v1.24.3-2+63243a96d1c393
k8s-worker-2   Ready    worker                 96h     v1.24.3-2+63243a96d1c393

$ kubectl -n hwameistor get pod -o wide | grep k8s-worker-2
hwameistor-local-disk-manager-sfsf1   2/2   Running   0   19h   10.6.128.150   k8
s-worker-2   <none>   <none>

# 检查 LocalDiskNode 资源
```

```
$ kubectl get localdisknode k8s-worker-2
NAME FREECAPACITY TOTALCAPACITY TOTALDISK STATUS AG
E
k8s-worker-2 Ready 21d
```

2. 添加新增存储节点到 HwameiStor 系统

首先，需要将磁盘 sdb 的 owner 信息修改成 local-disk-manager，具体如下：

```
$ kubectl edit ld k8s-worker-2-sdb
apiVersion: hwameistor.io/v1alpha1
kind: LocalDisk
metadata:
  name: k8s-worker-2-sdb
spec:
  devicePath: /dev/sdb
  nodeName: k8s-worker-2
+ owner: local-disk-manager
...
```

为增加存储节点创建资源 LocalStorageClaim，以此为新增存储节点构建存储池。这样，节点就已经成功加入 HwameiStor 系统。具体如下：

```
$ kubectl apply -f - <<EOF
apiVersion: hwameistor.io/v1alpha1
kind: LocalDiskClaim
metadata:
  name: k8s-worker-2
spec:
  nodeName: k8s-worker-2
  owner: local-disk-manager
  description:
    diskType: SSD
EOF
```

3. 后续检查

完成上述步骤后，检查新增存储节点及其存储池的状态，确保节点和 HwameiStor 系统的正常运行。具体如下：

```
$ kubectl get localdisknode k8s-worker-2 -o yaml
apiVersion: hwameistor.io/v1alpha1
```

```
kind: LocalDiskNode
metadata:
  name: k8s-worker-2
spec:
  nodeName: k8s-worker-2
status:
  pools:
    LocalDisk_PoolSSD:
      class: SSD
      disks:
      - capacityBytes: 214744170496
        devPath: /dev/sdb
        state: Available
        type: SSD
      freeCapacityBytes: 214744170496
      freeVolumeCount: 1
      totalCapacityBytes: 214744170496
      totalVolumeCount: 1
      type: REGULAR
      usedCapacityBytes: 0
      usedVolumeCount: 0
      volumeCapacityBytesLimit: 214744170496
      volumes:
  state: Ready
```

数据盘扩展

当存储系统中的某个节点存储容量不足时，可以通过为该节点增加磁盘扩充容量。 在 HwameiStor 中，可以通过下列步骤完成为节点增加磁盘（数据盘）。

具体操作步骤如下：

步骤

准备新的存储磁盘

在 Kubernetes 集群中新增一个节点，或者，选择一个已有的集群节点（非 HwameiStor 节

点）。 该节点必须满足 Prerequisites 要求的所有条件。 本例中，所用的新增存储节点和磁盘信息如下所示：

- name: k8s-worker-4
- devPath: /dev/sdb
- diskType: SSD disk

在新增磁盘被插入到 HwameiStor 存储节点 k8s-worker-4 后，检查该节点上的新磁盘状态，如下：

1. 检查新增磁盘是否成功插入节点，并被正确识别

```
# 1. 检查新增磁盘是否成功插入节点，并被正确识别
$ ssh root@k8s-worker-4
$ lsblk | grep sdc
sdc             8:32       0       20G   1 disk

# 2. 检查 HwameiStor 是否为新增磁盘正确创建资源 LocalDisk，并且状态为 `Unclaimed`
$ kubectl get localdisk | grep k8s-worker-4 | grep sdc
k8s-worker-4-sdc     k8s-worker-4          Unclaimed
```

2. 检查 HwameiStor 是否为新增磁盘正确创建资源 LocalDisk，并且状态为 Unclaimed

```
kubectl get localdisk | grep k8s-worker-4 | grep sdc
k8s-worker-4-sdc     k8s-worker-4          Unclaimed
```

将新增磁盘加入到节点的存储池

通过创建资源 LocalDiskClaim，将新增磁盘加入节点的存储池。 完成下列操作后，新磁盘应该被自动加入节点的 SSD 存储池中。如果该节点上没有 SSD 存储池，HwameiStor 会为其自动创建，并将新磁盘加入其中。

```
$ kubectl apply -f - <<EOF
apiVersion: hwameistor.io/v1alpha1
kind: LocalDiskClaim
metadata:
  name: k8s-worker-4-expand
spec:
  nodeName: k8s-worker-4
  description:
```

```
    diskType: SSD
EOF
```

后续检查

完成上述步骤后，检查新增磁盘及其存储池的状态，确保节点和 HwameiStor 系统的正常运行。具体如下：

```
apiVersion: hwameistor.io/v1alpha1
kind: LocalStorageNode
metadata:
  name: k8s-worker-4
spec:
  hostname: k8s-worker-4
  storageIP: 10.6.182.103
  topogoly:
    region: default
    zone: default
status:
  pools:
    LocalStorage_PoolSSD:
      class: SSD
      disks:
      - capacityBytes: 214744170496
        devPath: /dev/sdb
        state: InUse
        type: SSD
      - capacityBytes: 214744170496
        devPath: /dev/sdc
        state: InUse
        type: SSD
      freeCapacityBytes: 429488340992
      freeVolumeCount: 1000
      name: LocalStorage_PoolSSD
      totalCapacityBytes: 429488340992
      totalVolumeCount: 1000
      type: REGULAR
      usedCapacityBytes: 0
      usedVolumeCount: 0
      volumeCapacityBytesLimit: 429488340992
      volumes:
  state: Ready
```

预留磁盘

本章节介绍如何在界面上预留磁盘，此功能的使用场景如下：

运维管理员有磁盘的规划需求，需要预留部分磁盘不被 Hwameistor 使用，需要自行进行 Reserved 操作

使用说明

1. Local Disk 的使用状态和预留状态为 2 个状态，任何状态的 LD 都可以 Reserved/Unreserved。
2. Hwameistor 在启动初期会自动检测系统盘并标记为 Reserveds
3. LD 只有三种状态：**Pending，Available，Bound。**LD 使用状态和 Reserved 情况如下：

使用状态 | 是否 Reserved | 场景说明 |

: ——– | :———— | :———————————————– |

Pending | – | 初始化状态 |

Pending | Reserved | 初始化瞬间被用户预留 |

Available | – | 空闲磁盘，可被 Hwameistor 分配 |

Available | Reserved | 空闲磁盘但是规划作为他用，无法被 Hwameistor 分配无文件系统，但实际此磁盘已被使用，同时被用户标记成 Reserved |

Bound | – | 被 Hwameistor 使用的磁盘被系统或者外部程序使用，并手动去除了 Reserved 状态 |

Bound | Reserved | 被系统或者外部程序使用，并手动去除 Reserved 状态被 Hwamristor 使用，同时用户手动标记为 Reserved 状态，当磁盘上的数据释放后，不在被 Hwameistor 系统使用 |

操作步骤

进入对应集群，选择存储–>Hwameistor；点击节点 进入节点详情页面，找到对应的磁盘；点击 …，选择预留磁盘；点击确认进行预留。

预留后磁盘将不会被 Hwameistor 使用。

创建存储池

下文所述为 HwameiStor 原生的命令行存储池操作方式。DCE 5.0 在集成 HwameiStor 作为本地存储之后，为其提供了图形化 UI，可以通过表单和 YAML 创建存储池。推荐使用 UI 执行存储池的增删改查操作，方便通过直观的界面管理各类存储池。

下面的例子来自一个 4 节点的 Kubernetes 集群：

```
$ kubectl get no
NAME                STATUS     ROLES     AGE     VERSION
k8s-master-1     Ready      master    82d     v1.24.3-2+63243a96d1c393
k8s-worker-1     Ready      worker    36d     v1.24.3-2+63243a96d1c393
k8s-worker-2     Ready      worker    59d     v1.24.3-2+63243a96d1c393
k8s-worker-3     Ready      worker    36d     v1.24.3-2+63243a96d1c393
```

创建 LocalDiskClaim 对象

HwameiStor 根据存储介质类型创建 LocalDiskClaim 对象来创建存储池。 要在所有 Kubernetes Worker 节点上创建一个 HDD 存储池，用户需要通过 storageNodes 参数输入各个 Worker 节点名：

```
helm template ./hwameistor \
    -s templates/post-install-claim-disks.yaml \
    --set storageNodes='{k8s-worker-1,k8s-worker-2,k8s-worker-3}' \
    | kubectl apply -f -
```

或者通过以下方法指定所有 Worker 节点：

```
sn="$( kubectl get no -l node-role.kubernetes.io/worker -o jsonpath="{.items[*].metadata.name}" | tr ' ' ',' )"

helm template ./hwameistor \
    -s templates/post-install-claim-disks.yaml \
    --set storageNodes="{$sn}" \
  | kubectl apply -f -
```

验证 LocalDiskClaim 对象

LocalDiskClaim 对象在挂载（bound）后会被自动删除，你可以运行以下命令查看其是否已

被删除：

```
$ kubectl get ldc
No resources found
```

验证 StorageClass

运行以下命令：

```
$ kubectl get sc hwameistor-storage-lvm-hdd
NAME PROVISIONER RECLAIMPOLICY VOLUMEBI
NDINGMODE ALLOWVOLUMEEXPANSION AGE
hwameistor-storage-lvm-hdd lvm.hwameistor.io Delete WaitForFirstConsumer tr
ue 114s
```

验证 LocalDisk 对象

运行以下命令：

```
[root@k8s-master home]# kubectl get ld
```

输出类似于：

```
NAME NODEMATCH DEVICEPATH OW
NER PHASE STATE AGE
localdisk-2307de2b1c5b5d051058bc1d54b41d5c k8s-node1 /dev/sdb local-storage Bo
und Active 5d23h
localdisk-311191645ea00c62277fe709badc244e k8s-node2 /dev/sdb
Available Active 5d23h
localdisk-37a20db051af3a53a1c4e27f7616369a k8s-master /dev/sdb A
vailable Active 5d23h
```

localdisk-b57b108ad2ccc47f4b4fab6f0b9eaeb5 k8s-node2 /dev/sda system Bound Active 5d23h
localdisk-b682686c65667763bda58e391fbb5d20 k8s-master /dev/sda system Bound Active 5d23h
localdisk-da121e8f0dabac9ee1bcb6ed69840d7b k8s-node1 /dev/sda system Bound Active 5d23h

观察 VG (可选)

在一个 Kubernetes Worker 节点上，观察为 LocalDiskClaim 对象创建 VG。

运行以下命令：

```
$ vgdisplay LocalStorage_PoolHDD
  --- Volume group ---
  VG Name               LocalStorage_PoolHDD
  System ID
  Format                lvm2
  Metadata Areas        2
  Metadata Sequence No  1
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                0
  Open LV               0
  Max PV                0
  Cur PV                2
  Act PV                2
  VG Size               199.99 GiB
  PE Size               4.00 MiB
  Total PE              51198
  Alloc PE / Size       0 / 0
  Free  PE / Size       51198 / 199.99 GiB
  VG UUID               jJ3s7g-iyoJ-c4zr-3Avc-3K4K-BrJb-A5A5Oe
```

!!! note
在安装 HwameiStor 期间也可以通过设置 `storageNode` 参数配置存储池：

```console
helm install hwameistor ./hwameistor \
    -n hwameistor --create-namespace \
    --set storageNodes='{k8s-worker-1,k8s-worker-2,k8s-worker-3}'
```

卷的迁移

Migrate 迁移功能是 HwameiStor 中重要的运维管理功能，当应用绑定的数据卷所在节点副本损坏时， 卷副本可以通过迁移到其他节点，并在成功迁移到新节点后，将应用重新调度到新节点，并进行数据卷的绑定挂载。

基本概念

LocalVolumeGroup(LVG) (数据卷组) 管理是 HwameiStor 中重要的一个功能。当应用 Pod 申请多个数据卷 PVC 时， 为了保证 Pod 能正确运行，这些数据卷必须具有某些相同属性，例如：数据卷的副本数量，副本所在的节点。 通过数据卷组管理功能正确地管理这些相关联的数据卷，是 HwameiStor 中非常重要的能力。

前提条件

LocalVolumeMigrate 需要部署在 Kubernetes 系统中，需要部署应用满足下列条件：

- LVM2 已安装，支持 lvm 类型的卷
- 应用 Pod 申请多个数据卷 PVC 时，对应数据卷需要使用相同配置 sc
- 基于 LocalVolume 粒度迁移时，默认所属相同 LocalVolumeGroup 的数据卷不会一并迁移（若一并迁移，需要配置开关 MigrateAllVols：true）

界面操作步骤

1. 创建 convertible StorageClass

   通过界面安装，请参考如何创建 StorageClass

2. 创建多个 PVC

通过界面创建多个 PVC，请参考如何创建 PVC

3. 部署多数据卷 Pod

通过界面创建应用，请参考如何创建工作负载，并挂载已创建好的 2 个 PVC

4. 解除挂载多数据卷 Pod

迁移之前请先解除 PVC 挂载，可通过 **编辑工作负载** 进行解除挂载。

unbound01

unbound01

5. 创建迁移任务

进入对应集群 –> 点击左侧 **容器存储** –> **Hwameistor** 进入 **Hwameistor** 界面，选择已经解绑的本地卷， 对应的 PVC 为 pvc-test01、pvc-test02，点击 **…** 选择 **迁移** ，选择 **源节点** 、 **目标节点** 。

- **源节点** ： 本地卷副本所在的节点。
- **目标节点** ： 指定后原副本将迁移至目标节点，如选择 **自动选择** ，则本地卷副本将自动调度至其他节点。

如两个/多个本地卷挂载在同一个应用上，则两个卷会自动组成 本地卷组 统一进行迁移。

migration01

migration01

6. 点击对应的本地卷，进入详情查看迁移状态。

在线试用步骤

1. 创建 convertible StorageClass

可执行如下命令，PVC 进行创建：

```
cd ../../deploy/
kubectl apply -f storageclass-convertible-lvm.yaml
```

2. 创建多个 PVC

执行如下命令，PVC 进行创建：

```
kubectl apply -f pvc-multiple-lvm.yaml
```

3. 部署多数据卷 Pod

执行如下命令，PVC 进行创建：

```
kubectl apply -f nginx-multiple-lvm.yaml
```

4. 解除挂载多数据卷 Pod

执行如下命令，PVC 进行创建：

```
kubectl patch deployment nginx-local-storage-lvm --patch '{"spec": {"replicas": 0}}' -n hwa
    meistor
```

5. 创建迁移任务

```
cat > ./migrate_lv.yaml <<- EOF
    apiVersion: hwameistor.io/v1alpha1
    kind: LocalVolumeMigrate
    metadata:
      namespace: hwameistor
      name: <localVolumeMigrateName>
    spec:
    sourceNode: <sourceNodeName>
      targetNodesSuggested:
      - <targetNodesName1>
      - <targetNodesName2>
      volumeName: <volName>
      migrateAllVols: <true/false>
    EOF
kubectl apply -f ./migrate_lv.yaml
```

6. 查看迁移状态

```
kubectl get LocalVolumeMigrate  -o yaml
apiVersion: v1
    items:
    - apiVersion: hwameistor.io/v1alpha1
      kind: LocalVolumeMigrate
      metadata:
      annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"hwameistor.io/v1alpha1","kind":"LocalVolumeMigrate","metadata":{"anno
```

tations":{},"name":"localvolumemigrate-1","namespace":"hwameistor"},"spec":{"migrateAll
Vols":true,"sourceNodesNames":["dce-172-30-40-61"],"targetNodesNames":["172-30-45-223
"],"volumeName":"pvc-1a0913ac-32b9-46fe-8258-39b4e3b696a4"}}
creationTimestamp: "2022-07-07T12:34:31Z"
generation: 1
name: localvolumemigrate-1
namespace: hwameistor
resourceVersion: "12828637"
uid: 78af7f1b-d701-4b03-84de-27fafca58764
spec:
abort: false
migrateAllVols: true
sourceNodesNames:
- dce-172-30-40-61
targetNodesNames:
- 172-30-45-223
volumeName: pvc-1a0913ac-32b9-46fe-8258-39b4e3b696a4
status:
replicaNumber: 1
state: InProgress
kind: List
metadata:
resourceVersion: ""
selfLink: ""

7.查看迁移成功状态

```
[root@172-30-45-222 deploy]# kubectl get lvr
NAME                                                CAPACITY        NODE
          STATE     SYNCED     DEVICE
                                    AGE
pvc-1a0913ac-32b9-46fe-8258-39b4e3b696a4-9cdkkn     1073741824     172-30-45-223
Ready     true        /dev/LocalStorage_PoolHDD-HA/pvc-1a0913ac-32b9-46fe-8258-39b4e3
b696a4     77s
pvc-d9d3ae9f-64af-44de-baad-4c69b9e0744a-7ppmrx     1073741824     172-30-45-223     R
eady     true        /dev/LocalStorage_PoolHDD-HA/pvc-d9d3ae9f-64af-44de-baad-4c69b9e0
744a     77s
```

8.迁移成功后，重新挂载数据卷 Pod

```
kubectl patch deployment nginx-local-storage-lvm --patch '{"spec": {"replicas": 1}}' -n hwa
meistor
```

数据卷驱逐

数据卷驱逐是 HwameiStor 系统的重要功能，保障 HwameiStor 在生产环境中持续正常运行。当 Kubernetes 节点或者应用 Pod 由于某种原因被驱逐时，系统会自动发现节点或者 Pod 所关联的 HwameiStor 数据卷，并自动将其迁移到其他节点，从而保证被驱逐的 Pod 可以调度到其他节点并正常运行。 此外，运维人员也可以主动迁移数据卷，从而平衡系统资源，保证系统平稳运行。参考数据卷迁移

驱逐节点

在 Kubernetes 系统中，可以使用下列命令驱逐节点，将正在该节点上运行的 Pod 移除并迁移到其他节点上。 同时，HwameiStor 将 Pod 使用的数据卷从该节点迁移到其他节点，保证 Pod 可以在其他节点上正常运行。

```
kubectl drain k8s-node-1 --ignore-daemonsets=true
```

可以使用下列命令查看所关联的 HwameiStor 数据卷是否迁移成功。

```
kubectl get LocalStorageNode k8s-node-1 -o yaml
```

输出类似于：

```
apiVersion: hwameistor.io/v1alpha1
kind: LocalStorageNode
metadata:
  creationTimestamp: "2022-10-11T07:41:58Z"
  generation: 1
  name: k8s-node-1
  resourceVersion: "6402198"
  uid: c71cc6ac-566a-4e0b-8687-69679b07471f
spec:
  hostname: k8s-node-1
  storageIP: 10.6.113.22
  topogoly:
    region: default
    zone: default
status:
```

```
...
pools:
  LocalStorage_PoolHDD:
    class: HDD
    disks:
    - capacityBytes: 17175674880
      devPath: /dev/sdb
      state: InUse
      type: HDD
    freeCapacityBytes: 16101933056
    freeVolumeCount: 999
    name: LocalStorage_PoolHDD
    totalCapacityBytes: 17175674880
    totalVolumeCount: 1000
    type: REGULAR
    usedCapacityBytes: 1073741824
    usedVolumeCount: 1
    volumeCapacityBytesLimit: 17175674880
    volumes: # 确保 volumes 为空
state: Ready
```

同时，可以使用下列命令查看被驱逐节点上是否还有 HwameiStor 的数据卷。

```
kubectl get localvolumereplica
```

输出类似于：

```
NAME                                                CAPACITY        NODE        S
TATE    SYNCED    DEVICE
                AGE
pvc-1427f36b-adc4-4aef-8d83-93c59064d113-957f7g    1073741824    k8s-node-3    Ready    true
      /dev/LocalStorage_PoolHDD-HA/pvc-1427f36b-adc4-4aef-8d83-93c59064d113    20h
pvc-1427f36b-adc4-4aef-8d83-93c59064d113-qlpbmq    1073741824    k8s-node-2    Ready    true
      /dev/LocalStorage_PoolHDD-HA/pvc-1427f36b-adc4-4aef-8d83-93c59064d113    30m
pvc-6ca4c0d4-da10-4e2e-83b2-19cbf5c5e3e4-scrxjb    1073741824    k8s-node-2    Ready    true
      /dev/LocalStorage_PoolHDD/pvc-6ca4c0d4-da10-4e2e-83b2-19cbf5c5e3e4          30m
pvc-f8f017f9-eb09-4fbe-9795-a6e2d6873148-5t782b    1073741824    k8s-node-2    Ready    true
      /dev/LocalStorage_PoolHDD-HA/pvc-f8f017f9-eb09-4fbe-9795-a6e2d6873148    30m
```

在一些情况下，重启节点时，用户希望仍然保留数据卷在该节点上。可以通过在该节点上添加下列标签实现：

```
kubectl label node k8s-node-1 hwameistor.io/eviction=disable
```

驱逐 Pod

当 Kubernetes 节点负载过重时，系统会选择性地驱逐一些 Pod，从而释放一些系统资源，保证其他 Pod 正常运行。 如果被驱逐的 Pod 使用了 HwameiStor 数据卷，系统会捕捉到这个被驱逐的 Pod，自动将相关的 HwameiStor 数据卷迁移到其他节点，从而保证该 Pod 能正常运行。

迁移 Pod

运维人员可以主动迁移应用 Pod 和其使用的 HwameiStor 数据卷，从而平衡系统资源，保证系统平稳运行。 可以通过下列两种方式进行主动迁移：

- 方法 1

```
kubectl label pod mysql-pod hwameistor.io/eviction=start
kubectl delete pod mysql-pod
```

- 方法 2

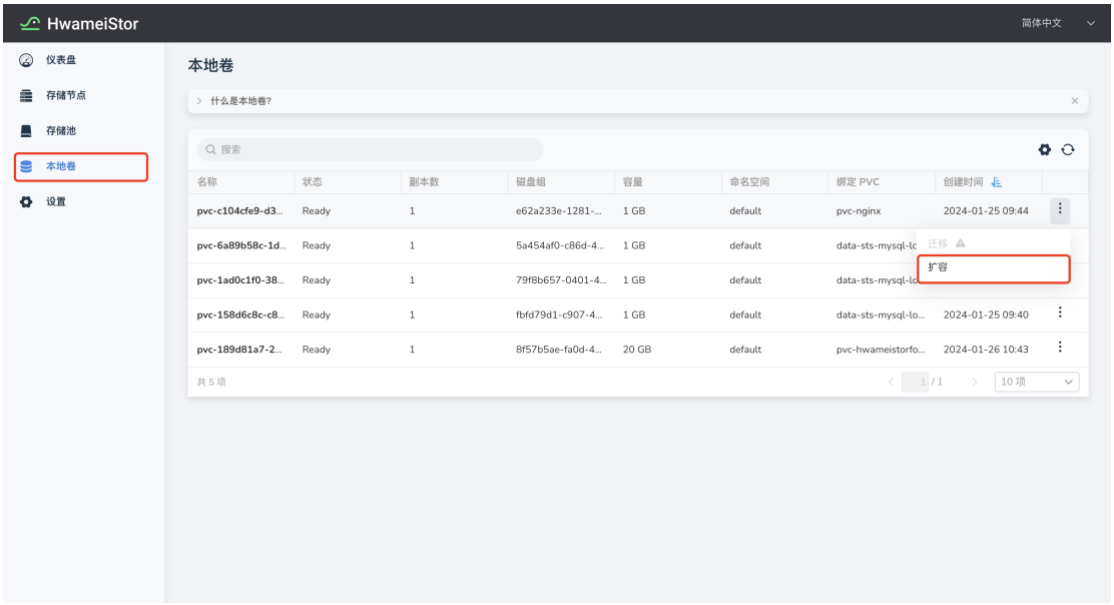
```
$ cat << EOF | kubectl apply -f -
apiVersion: hwameistor.io/v1alpha1
kind: LocalVolumeMigrate
metadata:
  name: migrate-pvc-6ca4c0d4-da10-4e2e-83b2-19cbf5c5e3e4
spec:
  sourceNode: k8s-node-1
  targetNodesSuggested:
  - k8s-node-2
  - k8s-node-3
  volumeName: pvc-6ca4c0d4-da10-4e2e-83b2-19cbf5c5e3e4
  migrateAllVols: true
EOF

$ kubectl delete pod mysql-pod
```

数据卷的扩容

HwameiStor 支持 CSI 卷扩容 。这个功能实现了通过修改 PVC 的大小在线扩容卷。其中目前支持手动、自动两种方式的扩容。

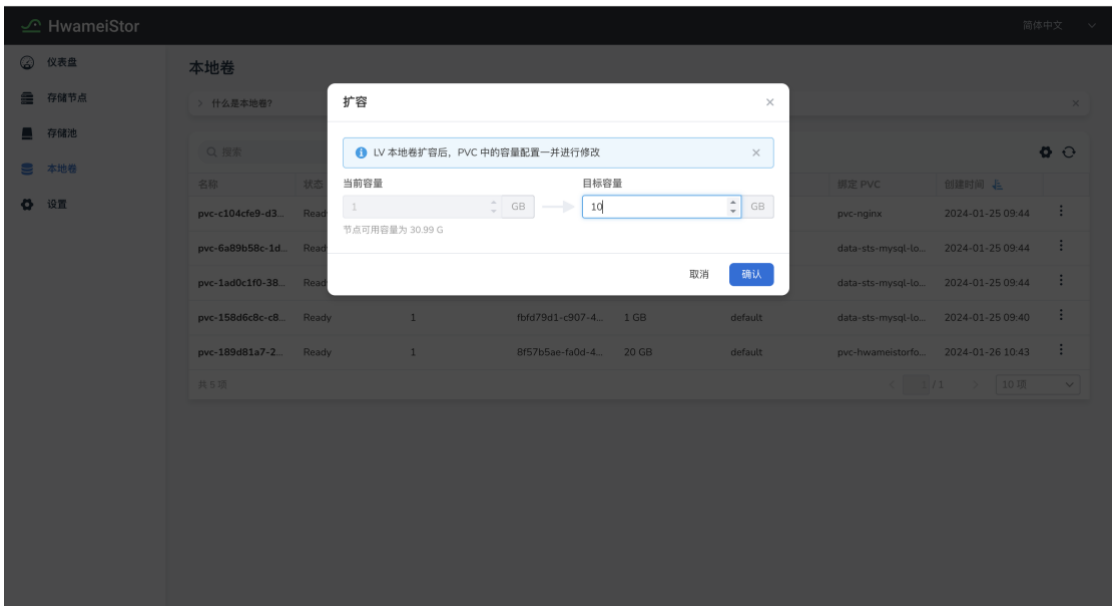
手动扩容数据卷

1. 进入对应集群，选择 **容器存储** -> **HwameiStor**
2. 点击 **本地卷** ，在本地卷列表中，点击一个本地卷右侧的 ⋮， 选择 **扩容**



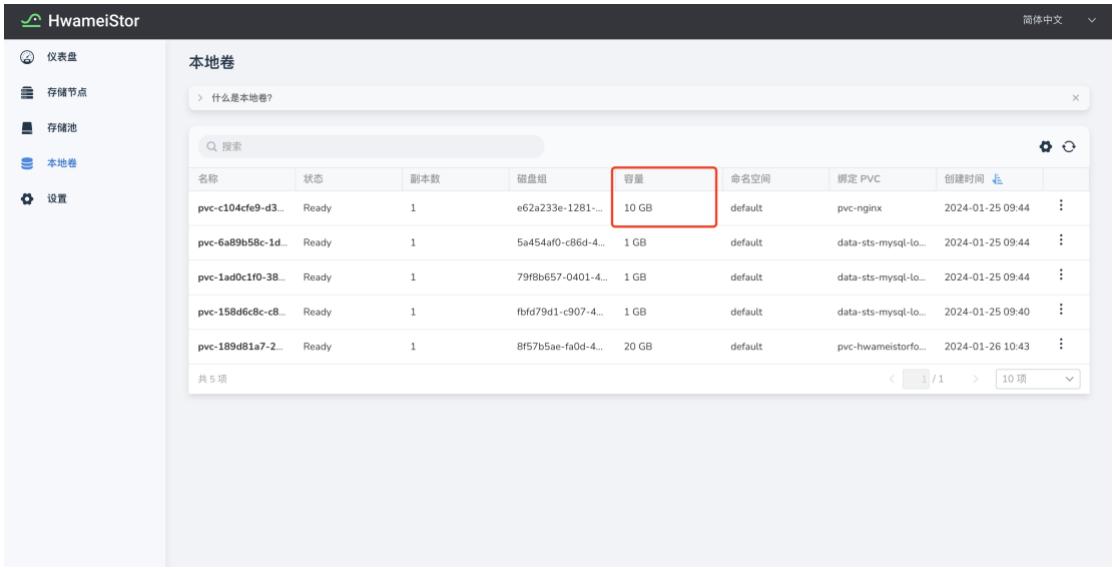
expand01

3. 在 **扩容** 对话框中填写扩容后的大小，本例从 1G 扩容至 10G，点击 **确定**



expand02

4. 刷新当前列表，观察到当前本地卷的容量已经从 1G 变成了 10G



expand03

自动扩容数据卷

组件 hwameistor-pvc-autoresizer 提供了 PVC 自动扩容的能力。 扩容行为是通过 ResizePolicy 这个 CRD 来控制的。

ResizePolicy

下面是一个示例 CR：

```
apiVersion: hwameistor.io/v1alpha1
kind: ResizePolicy
metadata:
  name: resizepolicy1
spec:
  warningThreshold: 60
  resizeThreshold: 80
  nodePoolUsageLimit: 90
```

warningThreshold、resizeThreshold、resizeThreshold 三个 int 类型的字段都表示一个百分比。

- warningThreshold 目前暂时还没有关联任何告警动作，它是作为一个目标比例，即扩容完成后卷的使用率会在这个比例以下。
- resizeThreshold 指示了一个使用率，当卷的使用率达到这个比例时，扩容动作就会被触发。
- nodePoolUsageLimit 表示节点存储池使用率的上限，如果某个池的使用率达到了这个比例， 那么落在这个池的卷将不会自动扩容。

匹配规则

这是一个带有 label-selector 的示例 CR。

```
apiVersion: hwameistor.io/v1alpha1
kind: ResizePolicy
metadata:
  name: example-policy
spec:
  warningThreshold: 60
  resizeThreshold: 80
  nodePoolUsageLimit: 90
  storageClassSelector:
    matchLabels:
      pvc-resize: auto
  namespaceSelector:
```

```
    matchLabels:
      pvc-resize: auto
  pvcSelector:
    matchLabels:
      pvc-resize: auto
```

ResizePolicy 有三个 label-selector：

- pvcSelector 表示被这个 selector 选中的 PVC 会依照选中它的 policy 自动扩容。
- namespaceSelector 表示被这个 selector 选中的 namespace 下的 PVC 会依照这个 policy 自动扩容。
- storageClassSelector 表示从被这个 selector 选中的 StorageClass 创建出来的 PVC 会依照这个 policy 自动扩容。

这三个 selector 之间是“且”的关系，如果你在一个 ResizePolicy 里指明了多个 selector，那么要符合全部的 selector 的 PVC 才会匹配这个 policy。如果 ResizePolicy 中没有指明任何 selector， 它就是一个集群 ResizePolicy，也就像是整个集群中所有 PVC 的默认 policy。

FAQ

如何观察扩容过程

增加的容量越多，扩容所需时间越长。可以在 PVC 的事件日志中观察整个扩容的过程.

```
kubectl describe pvc data-sts-mysql-local-0
Events:
  Type     Reason                  Age                From
                Message
  ----     ------                  ----               ----
      -------
  Warning  ExternalExpanding       34s                volume_expand
           Ignoring the PVC: didn't find a plugin capable of expanding the volume; waiting for
 an external controller to process this PVC.
  Warning  VolumeResizeFailed      33s                external-resizer lvm.hwameistor.
io  resize volume "pvc-b9fc8651-97b8-414c-8bcf-c8d2708c4ee8" by resizer "lvm.hwameistor.io"
failed: rpc error: code = Unknown desc = volume expansion not completed yet
  Normal   Resizing                32s (x2 over 33s)  external-resizer lvm.hwameistor.i
```

o External resizer is resizing volume pvc-b9fc8651-97b8-414c-8bcf-c8d2708c4ee8
Normal FileSystemResizeRequired 32s external-resizer lvm.hwameistor.i
o Require file system resize of volume on node
Normal FileSystemResizeSuccessful 11s kubelet
MountVolume.NodeExpandVolume succeeded for volume "pvc-b9fc8651-97b8-414c-8bcf-c8d2708c4ee8" k8s-worker-3

如何观察扩容完成后的 PVC/PV

kubectl get pvc data-sts-mysql-local-0
NAME STATUS VOLUME
CAPACITY ACCESS MODES STORAGECLASS AGE
data-sts-mysql-local-0 Bound pvc-b9fc8651-97b8-414c-8bcf-c8d2708c4ee8 2Gi R
WO hwameistor-storage-lvm-hdd 96m
kubectl get pv pvc-b9fc8651-97b8-414c-8bcf-c8d2708c4ee8
NAME CAPACITY ACCESS MODES RECLA
IM POLICY STATUS CLAIM STORAGECLASS
REASON AGE
pvc-b9fc8651-97b8-414c-8bcf-c8d2708c4ee8 2Gi RWO Delete
Bound default/data-sts-mysql-local-0 hwameistor-storage-lvm-hdd 96m

数据卷 IO 限速

在 HwameiStor 中，它允许用户指定 Kuberentes 集群上卷的最大 IOPS 和吞吐量。

请按照以下步骤创建具有最大 IOPS 和吞吐量的卷并创建工作负载来使用它。

前提条件

- 集群已经安装 HwameiStor
- cgroup v2 要求：
  - 操作系统发行版启用 cgroup v2
  - Linux 内核为 5.8 或更高版本

更多信息，请参见 Kubernetes 官网。

使用 IOPS 和吞吐量参数创建新的 StorageClass

1. 进入 **容器管理** 模块，在集群列表中找到已安装 HwameiStor 的集群，点击该集群的名称。
2. 在左侧导航栏中选择 **容器存储** -> **存储池 (SC)** ，并点击左上角按钮 **创建存储池(SC)** 。

sc01

sc01

3. 进入创建存储池界面，特别注意填写以下参数，其他参数可参考表单创建。
   - 名称：本示例输入 hwameistor-storage-lvm-hdd-sample 。
   - 存储系统：选择 **HwameiStor** 。
   - 存储类型：支持使用 **LVM 类型** 、 **裸辞盘类型** ，本示例选择 **LVM 类型** 。
   - 磁盘类型：支持 **HDD** 、 **SSD** ，本示例选择 **HDD** 。
   - 自定义参数：需要填写以下四个参数，
     - poolType: REGULAR ： 指定存储池类型，暂时仅支持 REGULAR。
     - csi.storage.k8s.io/fstype: xfs ：指定所需的文件系统类型，不定义默认为 ext4。
     - provision-iops-on-creation: "100" ： 指定创建时卷的最大 IOPS。
     - provision-throughput-on-creation: 1Mi ：指定创建时卷的最大吞吐量。

sc02

sc02

sc03

sc03

4. 点击 **确定** ，创建成功后返回 SC 列表界面。

使用 StorageClass 创建 PVC

1. 在左侧导航栏中选择 **容器存储** -> **数据卷声明 (PVC)** ，并点击左上角按钮 **创建数据**

卷声明（PVC）。

pvc01

pvc01

2. 进入创建数据卷声明界面，填写以下参数。
   - 名称：本示例输入 pvc-hwameistor-sample 。
   - 存储池：选择上述创建的 SC，名称为 hwameistor-storage-lvm-hdd-sample。
   - 容量：本示例输入 10。
   - 访问模式：默认选中 ReadWriteMany。

pvc02

pvc02

3. 点击 **确定** ，创建成功后返回 SC 列表界面。完成后，您可以创建 Deployment 来使用 PVC。

创建带有 PVC 的 Deployment

1. 在左侧导航栏中选择 **工作负载** -> **无状态负载** ，并点击左上角按钮 **镜像创建** 。

deploy01

deploy01

2. 需要注意可以填写以下参数：
   - image：本示例输入 daocloud.io/daocloud/testtools:latest。
   - 数据存储：
     - 类型：选择 数据卷声明(PVC)。
     - 数据卷声明（PVC）：选择 pvc-hwameistor-sample。
     - 容器路径（mountPath）：输入 /data。

   其余参数无特别要求。

deploy02

deploy02

3. 创建 Deployment 后，在详情界面点击 **控制台** ，执行以下命令测试卷的 IOPS 和吞吐量：

```
fio -direct=1 -iodepth=128 -rw=randwrite -ioengine=libaio -bs=4K -size=50M -numjobs=1
-runtime=600 -group_reporting -filename=/data/file.txt -name=Rand_Write_IOPS_Test
```

预期会输出如下：

deploy03

deploy03

!!! note

由于 cgroupv1 限制，最大 IOPS 和吞吐量的设置可能对非直接 IO 不生效。

更改数据卷的最大 IOPS 和吞吐量

最大 IOPS 和吞吐量在 StorageClass 的参数上指定，您不能直接更改它，因为它现在是不可变的。

与其他存储厂商不同的是，HwameiStor 是一个基于 Kubernetes 的存储解决方案，它定义了一组操作原语 基于 Kubernetes CRD。 这意味着您可以修改相关的 CRD 来更改卷的实际最大 IOPS 和吞吐量。

以下步骤显示如何更改数据卷的最大 IOPS 和吞吐量。

查找给定 PVC 对应的 LocalVolume CR

```
$ kubectl get pvc pvc-sample
NAME STATUS VOLUME CAPAC
ITY ACCESS MODES STORAGECLASS AGE
demo Bound pvc-c354a56a-5cf4-4ff6-9472-4e24c7371e10 10Gi RWO
hwameistor-storage-lvm-hdd 5d23h
pvc-sample Bound pvc-cac82087-6f6c-493a-afcd-09480de712ed 10Gi RWO
hwameistor-storage-lvm-hdd-sample 5d23h

$ kubectl get localvolume
NAME POOL REPLICAS C
APACITY USED STATE RESOURCE PUBLISHED FSTYPE AGE
pvc-c354a56a-5cf4-4ff6-9472-4e24c7371e10 LocalStorage_PoolHDD 1 107374182
40 33783808 Ready -1 master xfs 5d23h
```

pvc-cac82087-6f6c-493a-afcd-09480de712ed LocalStorage_PoolHDD 1 107374182
40 33783808 Ready -1 master xfs 5d23h

根据打印输出，PVC 的 LocalVolume CR 为 pvc-cac82087-6f6c-493a-afcd-09480de712ed。

修改 LocalVolume CR

kubectl edit localvolume pvc-cac82087-6f6c-493a-afcd-09480de712ed

在编辑器中，找到 spec.volumeQoS 部分并修改 iops 和 throughput 字段。 顺便说一下，

空值意味着没有限制。

最后，保存更改并退出编辑器。设置将在几秒钟后生效。

!!! note
将来，一旦 Kubernetes 支持[此特性](https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/3751-volume-attributes-class#motivation)，我们将允许用户直接修改卷的最大 IOPS 和吞吐量。

检查数据卷的实际 IOPS 和吞吐量

HwameiStor 使用 cgroupv1 来限制数据卷的 IOPS 和吞吐量，因此您可以使用以下命令来

检查数据卷的实际 IOPS 和吞吐量。

```
$ lsblk
NAME                  MAJ: MIN  RM     SIZE  RO  TYPE  MOUNTPOINT
sda                   8: 0      0     160G   0  disk
├─sda1                 8: 1      0       1G   0  part  /boot
└─sda2                 8: 2      0     159G   0  part
  ├─centos-root  253: 0      0     300G   0  lvm   /
  ├─centos-swap  253: 1      0     7.9G   0  lvm
  └─centos-home  253: 2      0  101.1G   0  lvm   /home
sdb                   8: 16     0     100G   0  disk
├─LocalStorage_PoolHDD-pvc--cac82087--6f6c--493a--afcd--09480de712ed
                      253:3      0      10G   0  lvm   /var/lib/kubelet/pods/3d6bc980-68ae-4a65-a1c8-8
b410b7d240f/v
└─LocalStorage_PoolHDD-pvc--c354a56a--5cf4--4ff6--9472--4e24c7371e10
                      253:4      0      10G   0  lvm   /var/lib/kubelet/pods/521fd7b4-3bef-415b-8720-0
9225f93f231/v
sdc                   8: 32     0     300G   0  disk
└─sdc1                 8: 33     0     300G   0  part
  └─centos-root  253: 0      0     300G   0  lvm   /
```

```
sr0                    11:  0       1      973M    0  rom

$  cat  /sys/fs/cgroup/blkio/blkio.throttle.read_iops_device
253:  3  100

$  cat  /sys/fs/cgroup/blkio/blkio.throttle.write_iops_device
253:  3  100

$  cat  /sys/fs/cgroup/blkio/blkio.throttle.read_bps_device
253:  3  1048576

$  cat  /sys/fs/cgroup/blkio/blkio.throttle.write_bps_device
253:  3  1048576
```

数据卷快照

在 HwameiStor 中，它允许用户创建数据卷的快照（VolumeSnapshot），且可以基于数据卷快照进行还原、回滚操作。

!!! note
目前仅支持对非高可用 LVM 类型数据卷创建快照。

为了避免数据不一致，请先暂停或者停止 I/O 然后再打快照。

请按照以下步骤创建 VolumeSnapshotClass 和 VolumeSnapshot 来使用它。

创建新的 VolumeSnapshotClass

默认情况下，HwameiStor 在安装过程中不会自动创建这样的 VolumeSnapshotClass，因此您需要手动创建如下 VolumeSnapshotClass。

```yaml
kind: VolumeSnapshotClass
apiVersion: snapshot.storage.k8s.io/v1
metadata:
  name: hwameistor-storage-lvm-snapshot
  annotations:
    snapshot.storage.kubernetes.io/is-default-class: "true"
parameters:
  snapsize: "1073741824" # 指定创建卷快照的大小
```

```
driver: lvm.hwameistor.io
deletionPolicy: Delete
```

!!! note
如果不指定 snapsize 参数，那么创建的快照大小和源卷的大小一致。

创建 VolumeSnapshotClass 后，您可以使用它来创建 VolumeSnapshot。

使用 VolumeSnapshotClass 创建 VolumeSnapshot

示例 VolumeSnapshot 如下：

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: snapshot-local-storage-pvc-lvm
spec:
  volumeSnapshotClassName: hwameistor-storage-lvm-snapshot
  source:
    persistentVolumeClaimName: local-storage-pvc-lvm # 指定要创建快照的 PVC
```

创建 VolumeSnapshot 后，您可以使用如下命令检查 VolumeSnapshot。

```
$ kubectl get vs
NAME                          READYTOUSE   SOURCEPVC           SOURCESNAPSHOTCONTENT   RESTORESIZE   SNAPSHOTCLASS                 SNAPSHOTCONTENT                                    CREATIONTIME   AGE
snapshot-local-storage-pvc-lvm   true         local-storage-pvc-lvm                  1Gi           hwameistor-storage-lvm-snapshot   snapcontent-0fc17697-68ea-49ce-8e4c-7a791e315110   53y            2m57s
```

创建 VolumeSnapshot 后，您可以使用如下命令检查 HwameiStor 本地卷快照。

```
$ kubectl get lvs
NAME                                               CAPACITY     SOURCEVOLUME                               STATE   MERGING   INVALID   AGE
snapcontent-0fc17697-68ea-49ce-8e4c-7a791e315110   1073741824   pvc-967baffd-ce10-4739-b996-87c9ed24e635   Ready                       5m31s
```

- CAPACITY：快照的容量大小
- SOURCEVOLUME：快照的源卷名称
- MERGING：快照是否处于合并状态（一般由 **回滚操作** 触发）
- INVALID：快照是否失效（一般在 **快照容量写满** 触发）

- AGE：快照真实创建的时间（不同于 CR 创建的时间，这个时间是底层快照数据卷的创建时间）

创建 VolumeSnapshot 后，您可以对 VolumeSnapshot 进行还原、回滚操作。

对卷快照进行还原操作

可以创建 PVC，对卷快照执行还原操作：

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-storage-pvc-lvm-restore
spec:
  storageClassName: local-storage-hdd-lvm
  dataSource:
    name: snapshot-local-storage-pvc-lvm
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

对卷快照进行回滚操作

!!! note
对快照进行回滚必须先 **停止源卷的 I/O** ，比如先停止应用，等待回滚操作完全结束，并 **确认数据一致性** 之后再使用回滚后的数据卷。

可以通过创建资源 LocalVolumeSnapshotRestore，对卷快照执行回滚操作：

```yaml
apiVersion: hwameistor.io/v1alpha1
kind: LocalVolumeSnapshotRestore
metadata:
  name: rollback-test
spec:
  sourceVolumeSnapshot: snapcontent-0fc17697-68ea-49ce-8e4c-7a791e315110 # 指定要进行回滚操作的本地卷快照
  restoreType: "rollback"
```

对创建的 LocalVolumeSnapshotRestore 进行观察，可以通过状态了解整个回滚的过程。回滚结束后，对应的 LocalVolumeSnapshotRestore 会被删除。

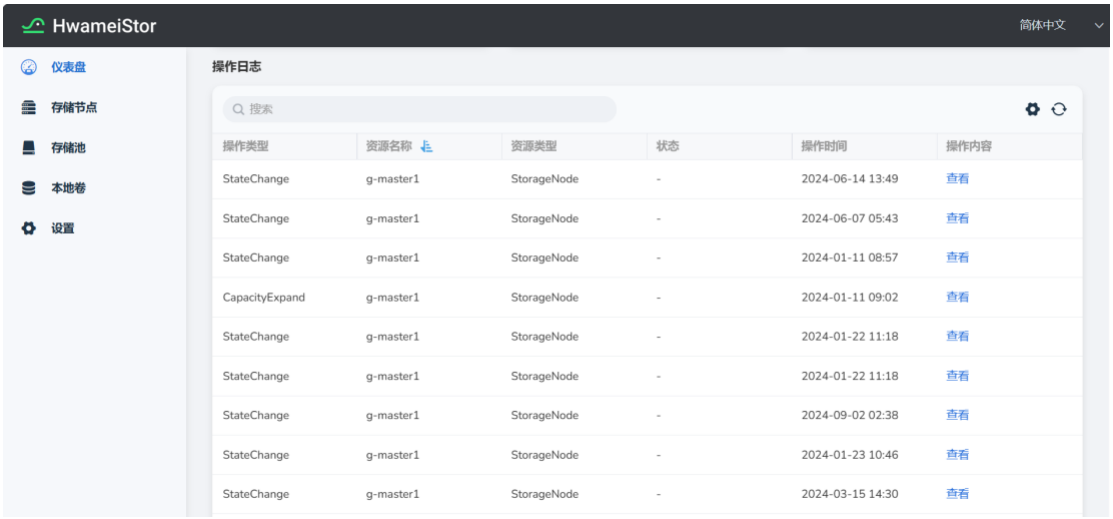
```
$ kubectl get LocalVolumeSnapshotRestore -w
NAME TARGETVOLUME SOURCESNAPSHOT STATE AGE
restore-test2 pvc-967baffd-ce10-4739-b996-87c9ed24e635 snapcontent-0fc17697-68ea-49ce-8e4c-7a791e315110 Submitted 0s
restore-test2 pvc-967baffd-ce10-4739-b996-87c9ed24e635 snapcontent-81a1f605-c28a-4e60-8c78-a3d504cbf6d9 InProgress 0s
restore-test2 pvc-967baffd-ce10-4739-b996-87c9ed24e635 snapcontent-81a1f605-c28a-4e60-8c78-a3d504cbf6d9 Completed 2s
```

查看操作日志

为了记录 HwameiStor 集群系统的使用和操作历史信息，HwameiStor 提供了系统操作日志。该操作日志具有 HwameiStor 系统语义，易于用户查阅、解析。 操作日志针对 HwameiStor 系统中的每类资源，记录其使用操作信息。该资源包括：Cluster、Node、StoragePool、Volume 等等。

查看操作日志的具体步骤如下：

1. 在左侧导航栏，点击 **容器管理** -> **集群列表** ，点击集群名称进入对应集群；点击 **容器存储** -> **HwameiStor** 进入 **HwameiStor** 界面，在 **仪表盘** 页面底部，可以看到 **操作日志** 列表。
2. 展示了操作类型、资源名称、资源类型、状态、操作时间、操作内容字段。



audit01

其中资源类型支持：

- Cluster
- StorageNode
- Disk
- DiskNode
- Pool
- Volume
- DiskVolume

操作内容：可以查看到更多的操作细节信息，如下图展示了 StorageNode 资源的一条审计的操作内容。

audit02

概览

HwameiStor 提供两种本地数据卷：LVM、Disk。

本地磁盘作为 HwameiStor 的一部分,负责提供 LVM 本地数据卷,其中包括高可用 LVM 数据卷、非高可用 LVM 数据卷。

非高可用的 LVM 本地数据卷，适用下列场景和应用：

- 具备高可用特性的 **数据库**，例如 MongoDB
- 具备高可用特性的 **消息中间件**，例如 Kafka、RabbitMQ
- 具备高可用特性的 **键值存储系统**，例如 Redis
- 其他具备高可用功能的应用

高可用的 LVM 本地数据卷，适用下列场景和应用：

- **数据库**，例如 MySQL、PostgreSQL
- 其他需要数据高可用特性的应用

热备份机制

单节点热备份

Raid 5 保障，可容忍 1 组磁盘故障。

控制流和数据流彼此独立，保证数据访问的稳定性。

单节点热备份

单节点热备份

跨节点热备份

Raid 5 + 主备副本保障。

规划了 HA 专用网络逻辑接口 dce-storage 在节点间同步存储流量。跨节点同步复制数据，保证数据的热备份。

跨节点热备份

跨节点热备份

数据再平衡

通过数据卷迁移技术实现数据在集群中的均衡放置。在线移动数据到有更多富裕空间的节点上。

数据再平衡

数据再平衡

数据卷类型变更

为了支持某些特殊场景，允许单副本数据卷变更为多副本，支持跨节点热备份。

数据卷类型变更

数据卷类型变更

使用非高可用本地卷

使用 HwameiStor 能非常轻松的运行有状态的应用。本文使用一个 MySQL 应用作为例子。

前提条件

- Hwameistor 已经安装成功
- 已经完成存储池创建，如未创建请进行存储池 StorageClass 创建。

目前 HwameiStor 安装成功后，Helm chart 会默认安装一个名为 hwameistor-storage-lvm-hdd 的 StorageClass，可使用此存储池创建本地数据卷。

1. 点击 **容器管理** ，选择对应集群，进入集群详情，点击 **容器存储** ，选择 **存储池** 。

   sc01

   sc01

2. 点击 **查看 YAML**，查看详情。

```yaml
apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    name: hwameistor-storage-lvm-hdd
  parameters:
    convertible: "false"
    csi.storage.k8s.io/fstype: xfs
    poolClass: HDD
    poolType: REGULAR
    replicaNumber: "1"
    striped: "true"
    volumeKind: LVM
  provisioner: lvm.hwameistor.io
  reclaimPolicy: Delete
  volumeBindingMode: WaitForFirstConsumer
  allowVolumeExpansion: true
```

如果这个 StorageClass 在安装时创建失败，可以通过界面创建 StorageClass 或 YAML 创建方式完成安装，YAML 方式如下：

```
kubectl apply -f examples/sc-local.yaml
```

操作步骤

通过界面创建 StatefulSet

1. 点击 **容器管理** ，选择对应集群，进入集群详情，点击 **容器存储** ，选择 **工作负载** 下的 **有状态工作负载** ，点击 **镜像创建** 。

imagecreate

2. 完成 **基本信息** 进入到下一步，点击 **创建数据卷声明模板** 输入如下参数信息：

pvctmp

   - 存储池：已经创建的本地存储池。
   - 容量：本地数据卷容量大小。
   - 访问模式：Pod 读写模式，建议使用 ReadWriteOnce。
   - 容器路径：数据存储挂载到容器上的路径。

3. 完成后点击 **确定** ，连续点击 **下一步** 完成创建。创建完成后点击 **数据卷** 列表查看对应的数据卷状态是否正常。

pvctmp02

通过 YAML 创建 StatefulSet

!!! note
下面的 MySQL Yaml 文件来自 [Kubernetes 的官方 Repo](https://github.com/kubernetes/website/blob/main/content/en/examples/application/mysql/mysql-statefulset.yaml)

在 HwameiStor 和 StorageClass 就绪后，一条命令就能创建 MySQL 容器和它的数据卷:

```
kubectl apply -f sts-mysql_local.yaml
```

请注意 volumeClaimTemplates 使用 storageClassName: hwameistor-storage-lvm-hdd:

```
spec:
  volumeClaimTemplates:
  - metadata:
```

```
    name: data
    labels:
      app: sts-mysql-local
      app.kubernetes.io/name: sts-mysql-local
  spec:
    storageClassName: hwameistor-storage-lvm-hdd
    accessModes: ["ReadWriteOnce"]
    resources:
      requests:
        storage: 1Gi
```

和 schedulerName: hwameistor-scheduler:

```
spec:
  template:
    spec:
      schedulerName: hwameistor-scheduler
```

查看 MySQL 容器和 PVC/PV

在这个例子里，MySQL 容器被调度到了节点 k8s-worker-3。

```
$ kubectl get po -l app=sts-mysql-local -o wide
NAME READY STATUS RESTARTS AGE IP NO
DE
sts-mysql-local-0 2/2 Running 0 3m08s 10.1.15.154 k8s-worker-3
```

```
$ kubectl get pvc -l app=sts-mysql-local
NAME STATUS VOLUME
CAPACITY ACCESS MODES STORAGECLASS AGE VOLUMEMO
DE
data-sts-mysql-local-0 Bound pvc-accf1ddd-6f47-4275-b520-dc317c90f80b 1Gi R
WO hwameistor-storage-lvm-hdd 3m Filesystem
```

查看 LocalVolume 的对象

通过查看和 PV 同名的 LocalVolume(LV)，可以看到本地卷创建在了节点 k8s-worker-3 上：

```
$ kubectl get lv pvc-accf1ddd-6f47-4275-b520-dc317c90f80b
```

```
NAME POOL REPLICAS C
APACITY ACCESSIBILITY STATE RESOURCE PUBLISHED AGE
pvc-accf1ddd-6f47-4275-b520-dc317c90f80b LocalStorage_PoolHDD 1 107374182
4 Ready -1 k8s-worker-3 3m
```

[可选] 扩展 MySQL 应用成一个三节点的集群

HwameiStor 支持 StatefulSet 的横向扩展. StatefulSet 容器都会挂载一个独立的本地卷:

```
$ kubectl scale sts/sts-mysql-local --replicas=3

$ kubectl get po -l app=sts-mysql-local -o wide
NAME                   READY   STATUS     RESTARTS   AGE     IP             N
ODE
sts-mysql-local-0      2/2     Running    0          4h38m   10.1.15.154    k8s-worker-3
sts-mysql-local-1      2/2     Running    0          19m     10.1.57.44     k8s-worker-2
sts-mysql-local-2      0/2     Init: 0/2  0          14s     10.1.42.237    k8s-worker-1

$ kubectl get pvc -l app=sts-mysql-local -o wide
NAME                         STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS                AGE   VOLUME
MODE
data-sts-mysql-local-0       Bound    pvc-accf1ddd-6f47-4275-b520-dc317c90f80b   1Gi        R
WO             hwameistor-storage-lvm-hdd   3m07s   Filesystem
data-sts-mysql-local-1       Bound    pvc-a4f8b067-9c1d-450f-aff4-5807d61f5d88   1Gi        R
WO             hwameistor-storage-lvm-hdd   2m18s   Filesystem
data-sts-mysql-local-2       Bound    pvc-47ee308d-77da-40ec-b06e-4f51499520c1   1Gi        R
WO             hwameistor-storage-lvm-hdd   2m18s   Filesystem

$ kubectl get lv
NAME                                       POOL                    REPLICAS   C
APACITY      ACCESSIBILITY   STATE   RESOURCE   PUBLISHED   AGE
pvc-47ee308d-77da-40ec-b06e-4f51499520c1   LocalStorage_PoolHDD    1          107374182
4            Ready           -1      k8s-worker-1   2m50s
pvc-a4f8b067-9c1d-450f-aff4-5807d61f5d88   LocalStorage_PoolHDD    1          1073741824
             Ready           -1      k8s-worker-2   2m50s
pvc-accf1ddd-6f47-4275-b520-dc317c90f80b   LocalStorage_PoolHDD    1          107374182
4            Ready           -1      k8s-worker-3   3m40s
```

使用高可用卷

HwameiStor 使用开源的 DRBD 数据同步技术创建**高可用卷**，本章节展示 高可用卷的使用

这里我们使用一个 MySQL 应用作为例子。

!!! note

下面的 MySQL Yaml 文件来自 [Kubernetes 的官方 Repo](https://github.com/kubernetes/website/blob/main/content/en/examples/application/mysql/mysql-statefulset.yaml)

前提条件

- HwameiStor 已经安装成功
- 已经完成高可用存储池创建，如未创建请进行存储池 StorageClass 创建。
- 已经部署 DRBD 内核组件

目前 HwameiStor 安装成功后，Helm chart 会默认安装一个名为 hwameistor-storage-lvm-hdd 的 StorageClass，可使用此存储池创建本地数据卷。

1. 点击容器管理 -> 选择对应集群，进入集群详情，点击容器存储，确认是否已创建高可用存储池

   sc01

   sc01

2. 点击查看 YAML，查看详情。StorageClass “hwameistor-storage-lvm-hdd-ha” 使用参数 replicaNumber: "2" 开启高可用功能：

   sc-yaml

   sc-yaml

   ```
   $ kubectl apply -f examples/sc_ha.yaml
   $ kubectl get sc hwameistor-storage-lvm-hdd-ha -o yaml

   apiVersion: storage.k8s.io/v1
   kind: StorageClass
   metadata:
     name: hwameistor-storage-lvm-hdd-ha
   parameters:
     replicaNumber: "2"
     convertible: "false"
     csi.storage.k8s.io/fstype: xfs
     poolClass: HDD
     poolType: REGULAR
     striped: "true"
     volumeKind: LVM
   provisioner: lvm.hwameistor.io
   reclaimPolicy: Delete
   ```

```
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
```

创建 StatefulSet

在 HwameiStor 和 StorageClass 就绪后，一条命令就能创建 MySQL 容器和它的数据卷:

```
kubectl apply -f exapmles/sts-mysql_ha.yaml
```

请注意 volumeClaimTemplates 使用 storageClassName: hwameistor-storage-lvm-hdd-ha:

```
spec:
  volumeClaimTemplates:
  - metadata:
      name: data
      labels:
        app: sts-mysql-ha
        app.kubernetes.io/name: sts-mysql-ha
    spec:
      storageClassName: hwameistor-storage-lvm-hdd-ha
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

查看 MySQL Pod and PVC/PV

在这个例子里，MySQL 容器被调度到了节点 k8s-worker-3。

```
$ kubectl get po -l app=sts-mysql-ha -o wide
NAME                READY   STATUS    RESTARTS   AGE     IP              NODE
sts-mysql-ha-0      2/2     Running   0          3m08s   10.1.15.151     k8s-worker-1

$ kubectl get pvc -l app=sts-mysql-ha
NAME                      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS                AGE   VOLUMEMODE
data-sts-mysql-ha-0       Bound    pvc-5236ee6f-8212-4628-9876-1b620a4c4c36   1Gi        RWO            hwameistor-storage-lvm-hdd   3m    Filesystem
```

查看 LocalVolume and LocalVolumeReplica 对象

通过查看和 PV 同名的 LocalVolume(LV)，可以看到本地卷创建在了节点 k8s-worker-1 和节点 k8s-worker-2.

```
$ kubectl get lv pvc-5236ee6f-8212-4628-9876-1b620a4c4c36

NAME                                       POOL                    REPLICAS   CAPACITY     ACCESSIBILITY   STATE   RESOURCE   PUBLISHED                                  AGE
pvc-5236ee6f-8212-4628-9876-1b620a4c4c36   LocalStorage_PoolHDD    1          1073741824   Ready           -1      k8s-worker-1,k8s-worker-2   3m
```

LocalVolumeReplica (LVR) 进一步显示每个节点上的后端逻辑卷设备：

```
kubectl get lvr
NAME                                          CAPACITY     NODE          STATE   SYNCED   DEVICE                                                                AGE
5236ee6f-8212-4628-9876-1b620a4c4c36-d2kn55   1073741824   k8s-worker-1   Ready   true     /dev/LocalStorage_PoolHDD-HA/5236ee6f-8212-4628-9876-1b620a4c4c36   4m
5236ee6f-8212-4628-9876-1b620a4c4c36-glm7rf   1073741824   k8s-worker-3   Ready   true     /dev/LocalStorage_PoolHDD-HA/5236ee6f-8212-4628-9876-1b620a4c4c36   4m
```

HwameiStor 常见问题

本页列举 HwameiStor 本地存储在使用过程中可能会遇到的一些疑问、回答和解决办法。

HwameiStor 调度器在 Kubernetes 平台中是如何工作的？

HwameiStor 的调度器是以 Pod 的形式部署在 HwameiStor 的命名空间。

调度器 Pod

调度器 Pod

当应用（Deployment 或 StatefulSet ）被创建后，应用的 Pod 会被自动部署到已配置好具

备 HwameiStor 本地存储能力的 Worker 节点上。

HwameiStor 如何应对应用多副本工作负载的调度？与传统通用型共享存储有什么不同？

HwameiStor 建议使用有状态的 StatefulSet 用于多副本的工作负载。

有状态应用 StatefulSet 会将复制的副本部署到同一 Worker 节点，但会为每一个 Pod 副本创建一个对应的 PV 数据卷。 如果需要部署到不同节点分散工作负载，需要通过 podAffinity 手动配置。

分散工作负载

分散工作负载

由于无状态应用 Deployment 不能共享 Block 数据卷，所以建议使用单副本。

对于传统通用型共享存储：

有状态应用 StatefulSet 会将复制的副本优先部署到其他节点以分散工作负载，但会为每一个 Pod 副本创建一个对应的 PV 数据卷。 只有当副本数超过 Worker 节点数的时候会出现多个副本在同一个节点。

无状态应用 Deployment 会将复制的副本优先部署到其他节点以分散工作负载，并且所有的 Pod 共享一个 PV 数据卷（目前仅支持 NFS）。 只有当副本数超过 Worker 节点数的时候会出现多个副本在同一个节点。对于 block 存储，由于数据卷不能共享，所以建议使用单副本。

如何运维一个 Kubernetes 节点上的数据卷？

HwameiStor 提供了数据卷驱逐和迁移功能。在移除或者重启一个 Kubernetes 节点的时候，

可以将该节点上的 Pod 和数据卷自动迁移到其他可用节点上，并确保 Pod 持续运行并提供服务。

移除节点

为了确保 Pod 的持续运行，以及保证 HwameiStor 本地数据持续可用，在移除 Kubernetes 节点之前， 需要将该节点上的 Pod 和本地数据卷迁移至其他可用节点。可以通过下列步骤进行操作：

1. 排空节点

   kubectl drain NODE --ignore-daemonsets=true. --ignore-daemonsets=true

   该命令可以将节点上的 Pod 驱逐，并重新调度。同时，也会自动触发 HwameiStor 的数据卷驱逐行为。 HwameiStor 会自动将该节点上的所有数据卷副本迁移到其他节点，并确保数据仍然可用。

2. 检查迁移进度。

   ```
   kubectl get localstoragenode NODE -o yaml
   apiVersion: hwameistor.io/v1alpha1
     kind: LocalStorageNode
     metadata:
       name: NODE
     spec:
       hostname: NODE
       storageIP: 10.6.113.22
       topogoly:
         region: default
         zone: default
     status:
       ...
       pools:
         LocalStorage_PoolHDD:
           class: HDD
           disks:
           - capacityBytes: 17175674880
             devPath: /dev/sdb
             state: InUse
   ```

```
    type: HDD
  freeCapacityBytes: 16101933056
  freeVolumeCount: 999
  name: LocalStorage_PoolHDD
  totalCapacityBytes: 17175674880
  totalVolumeCount: 1000
  type: REGULAR
  usedCapacityBytes: 1073741824
  usedVolumeCount: 1
  volumeCapacityBytesLimit: 17175674880
 volumes:   # (1)!
state: Ready
```

1. 确保 volumes 字段为空

同时，HwameiStor 会自动重新调度被驱逐的 Pod，将它们调度到有效数据卷所在的节点上，并确保 Pod 正常运行。

3. 从集群中移除节点

```
kubectl delete nodes NODE
```

重启节点

重启节点通常需要很长的时间才能将节点恢复正常。在这期间，该节点上的所有 Pod 和本地数据都无法正常运行。 这种情况对于一些应用（例如，数据库）来说，会产生巨大的代价，甚至不可接受。

HwameiStor 可以立即将 Pod 调度到其他数据卷所在的可用节点，并持续运行。 对于使用 HwameiStor 多副本数据卷的 Pod，这一过程会非常迅速，大概需要 10 秒 （受 Kubernetes 的原生调度机制影响）；对于使用单副本数据卷的 Pod， 这个过程所需时间依赖于数据卷迁移所需时间，受用户数据量大小影响。

如果用户希望将数据卷保留在该节点上，在节点重启后仍然可以访问，可以在节点上添加下列标签， 阻止系统迁移该节点上的数据卷。系统仍然会立即将 Pod 调度到其他有数据卷副本的节点上。

1. 添加一个标签（可选）。

   如果在节点重新启动期间不需要迁移数据卷，你可以在排空（drain）节点之前将以下标签添加到该节点。

   ```
   kubectl label node NODE hwameistor.io/eviction=disable
   ```

2. 排空节点。

   ```
   kubectl drain NODE --ignore-daemonsets=true. --ignore-daemonsets=true
   ```

   - 如果执行了第 1 步，待第 2 步成功后，用户即可重启节点。
   - 如果没有执行第 1 步，待第 2 步成功后，用户察看数据迁移是否完成（方法如同移除节点的第 2 步）。 待数据迁移完成后，即可重启节点。

   在前两步成功之后，用户可以重启节点，并等待节点系统恢复正常。

3. 节点恢复至 Kubernetes 的正常状态。

   ```
   kubectl uncordon NODE
   ```

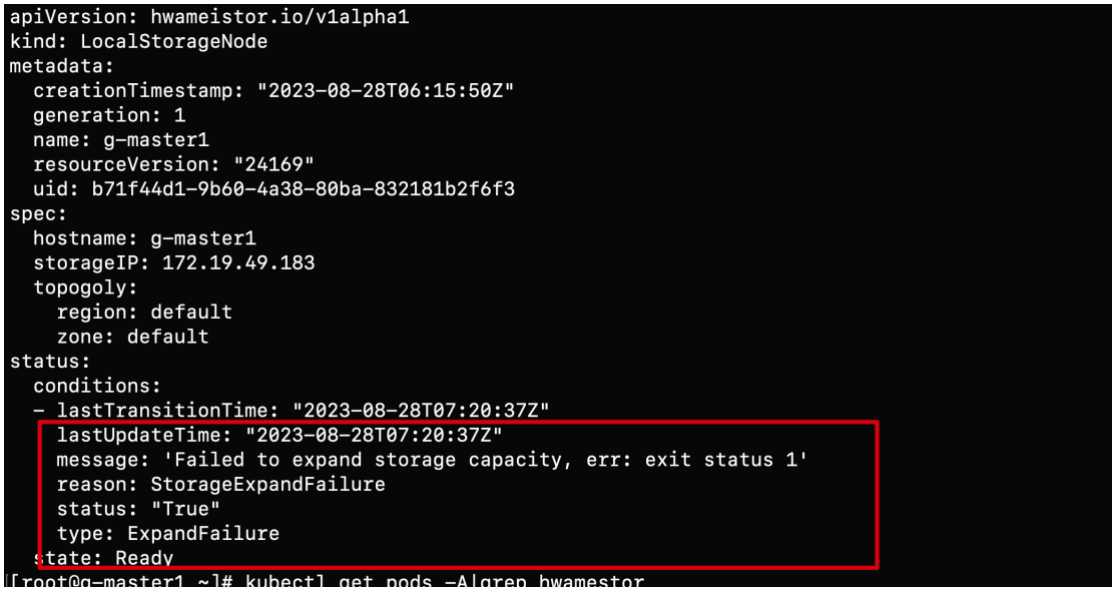
对于传统通用型共享存储

有状态应用 StatefulSet 会将复制的副本优先部署到其他节点以分散工作负载，但会为每一个 Pod 副本创建一个对应的 PV 数据卷。 只有当副本数超过 Worker 节点数的时候会出现多个副本在同一个节点。

无状态应用 Deployment 会将复制的副本优先部署到其他节点以分散工作负载，并且所有的 Pod 共享一个 PV 数据卷 （目前仅支持 NFS）。只有当副本数超过 Worker 节点数的时候会出现多个副本在同一个节点。对于 block 存储， 由于数据卷不能共享，所以建议使用单副本。

LocalStorageNode 查看出现报错如何处理？

当查看 LocalStorageNode 出现如下报错：



faq_04

可能的错误原因：

1. 节点没有安装 LVM2，可通过如下命令进行安装：

   ```
   rpm -qa | grep lvm2   # (1)!
   yum install lvm2 # (2)!
   ```

   1. 确认 LVM2 是否安装
   2. 在每个节点上确认 LVM 已安装

2. 确认节点上对应磁盘的 GPT 分区：

   ```
   blkid /dev/sd*   # (1)!
   wipefs -a /dev/sd* # (2)!
   ```

   1. 确认磁盘分区是否干净
   2. 磁盘清理

使用 hwameistor-operator 安装后为什么没有自动创建 StorageClass

可能的原因：

1. 节点没有可自动纳管的剩余裸盘，可通过如下命令进行检查：

   ```
   kubectl get ld # (1)!
   kubectl get lsn <node-name> -o yaml # (2)!
   ```

1. 检查磁盘

2. 检查磁盘是否被正常纳管

2. HwameiStor 相关组件（不包含 drbd-adapter）没有正常工作，可通过如下命令进行检查：

```
kubectl get pod -n hwameistor  # (1)!
kubectl get hmcluster -o yaml  # (2)!
```

1. 确认 Pod 是否运行正常

2. 查看 health 字段

!!! note
drbd-adapter 组件只有在启用高可用时才生效，如果没有启用，可以忽略相关错误。

Rook-ceph 存储方案

DCE 5.0 支持众多第三方存储方案，我们针对 Rook-ceph 进行了详细的测试，并最终将其作为 Addon 集成了应用商店中。 以下是对 Rook-ceph 的详细调研和测评。

有关应用商店 Addon 的图形化界面安装、部署、卸载等操作说明，将于稍后提供。

简介

Rook 是一个开源云原生存储编排器，为各种存储解决方案（存储提供商）提供平台、框架和支持并与云原生环境原生集成。

Rook 支持多个存储提供商。它将分布式存储软件转变为自我管理、自我扩展和自我修复的存储服务。 它通过自动执行存储提供商的部署、引导、配置、预配、扩展、升级、迁移、灾难恢复、监视和资源管理来实现此目的。 当与 Kubernetes 一起使用时，Rook 使用 Kubernetes 调度和编排平台提供的功能，为调度、生命周期管理、资源管理、安全性、监控和用户体验提供无缝体验。

- Rook-ceph 架构

rook-ceph 架构

rook-ceph 架构

- Rook 管理

rook 管理

rook 管理

- CSI 存储资源调配

rook 资源调配

rook 资源调配

- Ceph 数据路径

rook 数据路径

rook 数据路径

- 分布式架构

Ceph 是一种分布式、可扩展的开源存储解决方案，适用于块、对象和共享文件系统存储。 Ceph 在过去几年中已经发展成为开源分布式存储解决方案的标准，在大中型企业中进行了多年的生产部署。

rook 分布式架构

rook 分布式架构

- 组件：Ceph 存储池、归置组（PG）、对象、OSD

rook 组件

rook 组件

- Ceph RBD 工作原理

rook 原理

rook 原理

测试验证

准备环境

本次测试使用 4 个虚拟机节点来部署一个 Kubernetes 集群：

- 1 个 master 节点
- 3 个 worker 节点

其中 kubelet 版本为 1.23.6

```
[root@k8s-10-6-162-21 ~]# kubectl get no
NAME                 STATUS   ROLE
S                    AGE      VERSION
k8s-10-6-162-21      Ready    control-plane,master   19
d       v1.23.6
k8s-10-6-162-22      Ready    <non
e>                   19d      v1.23.6
k8s-10-6-162-23      Ready    <non
e>                   19d      v1.23.6
k8s-10-6-162-24      Ready    <non
e>                   13d      v1.23.6
```

部署 Rook-Ceph

克隆 GitHub 源码

```
[root@k8s-10-6-162-21 ~]# git clone https://github.com/rook/rook.git

[root@k8s-10-6-162-21 ~]# ls
anaconda-ks.cfg  calico.yaml  rook  rook-ceph-image  rook-ceph-image.zip

[root@k8s-10-6-162-21 rook]# git branch
master
* remotes/origin/release-1.7
```

```
[root@k8s-10-6-162-21 ceph]# cd /root/rook/cluster/examples/kubernetes/ceph

[root@k8s-10-6-162-21 ceph]# ls
ceph-client.yaml create-external-cluster-resources.py flex object-openshift.yaml rgw-external.yaml
cluster-external-management.yaml create-external-cluster-resources.sh flex-migrator.yaml object-test.
yaml scc.yaml
cluster-external.yaml csi images.txt object-user.yaml storageclass-bucket-delete.yaml
cluster-on-local-pvc.yaml dashboard-external-https.yaml import-external-cluster.sh object.yaml stora
geclass-bucket-retain.yaml
cluster-on-pvc.yaml dashboard-external-http.yaml monitoring operator-openshift.yaml test-data
cluster-stretched-aws.yaml dashboard-ingress-https.yaml nfs-test.yaml operator.yaml toolbox-job.ya
ml
cluster-stretched.yaml dashboard-loadbalancer.yaml nfs.yaml osd-purge.yaml toolbox.yaml
cluster-test.yaml direct-mount.yaml object-bucket-claim-delete.yaml pool-ec.yaml volume-replicatio
n-class.yaml
cluster.yaml filesystembak.yaml object-bucket-claim-retain.yaml pool-mirrored.yaml volume-replica
tion.yaml
common-external.yaml filesystem-ec.yaml object-ec.yaml pool-test.yaml
common-second-cluster.yaml filesystem-mirror.yaml object-external.yaml pool.yaml
common.yaml filesystem-test.yaml object-multisite-pull-realm.yaml pre-k8s-1.16
crds.yaml filesystem.yaml object-multisite.yaml rbdmirror.yaml
```

部署 Operator、CRD 和参数

```
[root@k8s-10-6-162-21 ceph]# kubectl create -f crds.yaml -f common.yaml -f operator.yaml

[root@k8s-10-6-162-21 ceph]# kubectl get crd | grep ceph
cephblockpools.ceph.rook.io 2022-10-14T02: 22:34Z
cephclients.ceph.rook.io 2022-10-14T02: 22:34Z
cephclusters.ceph.rook.io 2022-10-14T02: 22:35Z
cephfilesystemmirrors.ceph.rook.io 2022-10-14T02: 22:35Z
cephfilesystems.ceph.rook.io 2022-10-14T02: 22:35Z
cephnfses.ceph.rook.io 2022-10-14T02: 22:35Z
cephobjectrealms.ceph.rook.io 2022-10-14T02: 22:35Z
cephobjectstores.ceph.rook.io 2022-10-14T02: 22:35Z
cephobjectstoreusers.ceph.rook.io 2022-10-14T02: 22:35Z
cephobjectzonegroups.ceph.rook.io 2022-10-14T02: 22:35Z
cephobjectzones.ceph.rook.io 2022-10-14T02: 22:35Z
cephrbdmirrors.ceph.rook.io 2022-10-14T02: 22:35Z
objectbucketclaims.objectbucket.io 2022-10-14T02: 22:35Z
objectbuckets.objectbucket.io 2022-10-14T02: 22:35Z
volumes.rook.io 2022-10-14T02: 22:35Z
```

部署 cluster 和 toolbox

[root@k8s-10-6-162-21 ceph]# kubectl create -f cluster.yaml

[root@k8s-10-6-162-21 ceph]# kubectl create -f toolbox.yaml

[root@k8s-10-6-162-21 ceph]# kubectl get po -n rook-ceph -owide
NAME READY STATUS RESTARTS AGE IP
NODE NOMINATED NODE READINESS GATES
csi-cephfsplugin-bmbtc 3/3 Running 6 (4d1h ago) 7d7h 10.6.162.24
k8s-10-6-162-24 <none> <none>
csi-cephfsplugin-provisioner-5c8b6d6f4-hvfg6 6/6 Running 678 (38m ago) 7d7h
10.244.56.106 k8s-10-6-162-23 <none> <none>
csi-cephfsplugin-provisioner-5c8b6d6f4-w6snr 6/6 Running 545 (38m ago) 4d1h
10.244.169.249 k8s-10-6-162-22 <none> <none>
csi-cephfsplugin-r87qt 3/3 Running 6 (4d1h ago) 7d7h 10.6.162.23
k8s-10-6-162-23 <none> <none>
csi-cephfsplugin-xqrmr 3/3 Running 6 (4d1h ago) 7d7h 10.6.162.22
k8s-10-6-162-22 <none> <none>
csi-rbdplugin-mcpm6 3/3 Running 6 (4d1h ago) 7d7h 10.6.162.2
2 k8s-10-6-162-22 <none> <none>
csi-rbdplugin-pj4cc 3/3 Running 6 (4d2h ago) 7d7h 10.6.162.23
k8s-10-6-162-23 <none> <none>
csi-rbdplugin-provisioner-8564cfd44-mnjr8 6/6 Running 665 (38m ago) 4d8h 10.244.56.102
8s-10-6-162-23 <none> <none>
csi-rbdplugin-provisioner-8564cfd44-tgzc8 6/6 Running 545 (37m ago) 4d1h 10.244.169.248
k8s-10-6-162-22 <none> <none>
csi-rbdplugin-tnpbc 3/3 Running 6 (4d1h ago) 7d7h 10.6.162.24
k8s-10-6-162-24 <none> <none>
rook-ceph-crashcollector-k8s-10-6-162-22-5c4444c7bb-pd6tf 1/1 Running 0 27h 10.244.169.208
k8s-10-6-162-22 <none> <none>
rook-ceph-crashcollector-k8s-10-6-162-23-775d89c75-kgnzq 1/1 Running 2 (4d2h ago) 7d7h 1
0.244.56.105k8s-10-6-162-23 <none> <none>
rook-ceph-crashcollector-k8s-10-6-162-24-6cc87876c6-tk6q2 1/1 Running 0 27h 10.244.49.18
k8s-10-6-162-24 <none> <none>
rook-ceph-mds-myfs-a-94f74bd56-k2gt7 1/1 Running 0 27h 10.244.49.2
2 k8s-10-6-162-24 <none> <none>
rook-ceph-mds-myfs-b-5679797bbd-jr8gt 1/1 Running 69 (38m ago) 27h 10.244.169.
209 k8s-10-6-162-22 <none> <none>
rook-ceph-mgr-a-5b69d56b4d-6z87x 1/1 Running 0 2d7h 10.244.49.
9 k8s-10-6-162-24 <none> <none>
rook-ceph-mon-a-7589d48f7d-lgps2 1/1 Running 25 (3h44m ago) 21h 10.244.56.1
22 k8s-10-6-162-23 <none> <none>

```
rook-ceph-mon-b-6577cdc487-fk564 1/1 Running 161 (3h14m ago) 7d7h 10.244.169.
244 k8s-10-6-162-22 <none> <none>
rook-ceph-mon-d-66bdbd9dc-pn6wc 1/1 Running 83 (38m ago) 4d 10.244.49.
61 k8s-10-6-162-24 <none> <none>
rook-ceph-operator-866c96498c-mm2xd 1/1 Running 4 (4d1h ago) 7d7h 10.244.169.
242 k8s-10-6-162-22 <none> <none>
rook-ceph-osd-0-6d774dcf99-6qggj 1/1 Running 167 (38m ago) 7d3h 10.244.169.
245 k8s-10-6-162-22 <none> <none>
rook-ceph-osd-1-7f7c599677-qx4sh 1/1 Running 26 (4d2h ago) 7d3h 10.244.56.10
1 k8s-10-6-162-23 <none> <none>
rook-ceph-osd-2-7564c58d96-bknjf 1/1 Running 125 (35m ago) 7d3h 10.244.169.
243 k8s-10-6-162-22 <none> <none>
rook-ceph-osd-3-5599b7bdcb-wmdkx 1/1 Running 0 4d1h 10.244.49.
51 k8s-10-6-162-24 <none> <none>
rook-ceph-osd-4-7c6d79cdff-4fzdc 1/1 Running 0 4d1h 10.244.49.55
k8s-10-6-162-24 <none> <none>
rook-ceph-osd-5-ffb7b5ff7-phssl 1/1 Running 26 (4d2h ago) 7d3h 10.244.56.103
k8s-10-6-162-23 <none> <none>
rook-ceph-osd-prepare-k8s-10-6-162-22-kjwzt 0/1 Completed 0 5h34m 10.244.169.22
2 k8s-10-6-162-22 <none> <none>
rook-ceph-osd-prepare-k8s-10-6-162-23-fngbr 0/1 Completed 0 5h34m 10.244.56.67
k8s-10-6-162-23 <none> <none>
rook-ceph-osd-prepare-k8s-10-6-162-24-p7c28 0/1 Completed 0 5h34m 10.244.49.27
k8s-10-6-162-24 <none> <none>
rook-ceph-tools-55f548895f-8pdhm 1/1 Running 0 4d1h 10.244.56.1
07 k8s-10-6-162-23 <none> <none>
```

部署 dashboard

```
[root@k8s-10-6-162-21 ~]# kubectl apply -f dashboard-external-https.yaml

[root@k8s-10-6-162-21 ~]# kubectl get svc -n rook-ceph
NAME TYPE CLUSTER-IP EXTERNAL-IP P
ORT(S) AGE
csi-cephfsplugin-metrics ClusterIP 10.111.92.191 <none> 8080/TCP,808
1/TCP 8d
csi-rbdplugin-metrics ClusterIP 10.106.136.163 <none> 8080/TCP,8
081/TCP 8d
rook-ceph-mgr ClusterIP 10.97.170.29 <none> 9283/TCP
8d
rook-ceph-mgr-dashboard ClusterIP 10.103.18.81 <none> 8443/TCP 8
d
rook-ceph-mgr-dashboard-external-https NodePort 10.111.121.36 <none> 8443: 30738/T
CP 8d
```

rook-ceph-mon-a ClusterIP 10.107.216.105 <none> 6789/TCP,3300/TCP 8d
rook-ceph-mon-b ClusterIP 10.100.168.72 <none> 6789/TCP,3300/TCP 8d
rook-ceph-mon-d ClusterIP 10.101.56.41 <none> 6789/TCP,3300/TCP 5d16h
dashboard-1
dashboard-1
dashboard-2
dashboard-2
dashboard-3
dashboard-3

操作 Ceph 存储管理所用的 Rook 工具

[root@k8s-10-6-162-21 ceph]# kubectl exec -it rook-ceph-tools-55f548895f-fzbq2 -n rook-ceph -- bash

[root@rook-ceph-tools-55f548895f-8pdhm /]# ceph -s
cluster:
id: d65486e5-a8c3-4ca5-8ff6-9daddfc0f6c4
health: HEALTH_OK

services:
mon: 3 daemons, quorum a,b,d (age 71m)
mgr: a(active, since 30h)
mds: 1/1 daemons up, 1 hot standby
osd: 6 osds: 6 up (since 68m), 6 in (since 7d)

data:
volumes: 1/1 healthy
pools: 4 pools, 480 pgs
objects: 464 objects, 1009 MiB
usage: 3.8 GiB used, 596 GiB / 600 GiB avail
pgs: 480 active+clean

io:
client: 1.5 KiB/s rd, 2 op/s rd, 0 op/s wr

[root@rook-ceph-tools-55f548895f-8pdhm /]# ceph mgr services
{
"dashboard": "https://10.244.49.9:8443/",
"prometheus": "http://10.244.49.9:9283/"
}

```
[root@rook-ceph-tools-55f548895f-8pdhm /]# ceph df
- -- RAW STORAGE ---
CLASS SIZE AVAIL USED RAW USED %RAW USED
hdd 600 GiB 596 GiB 3.8 GiB 3.8 GiB 0.63
TOTAL 600 GiB 596 GiB 3.8 GiB 3.8 GiB 0.63

- -- POOLS ---
POOL ID PGS STORED OBJECTS USED %USED MAX AVAIL
device_health_metrics 1 64 0 B 0 0 B 0 189 GiB
replicapool 4 128 880 MiB 290 2.6 GiB 0.45 189 GiB
myfs-metadata 7 256 1.5 MiB 25 3.0 MiB 0 283 GiB
myfs-data0 8 32 115 MiB 149 231 MiB 0.04 283 GiB

[root@rook-ceph-tools-55f548895f-8pdhm /]# ceph osd status
ID HOST USED AVAIL WR OPS WR DATA RD OPS RD DATA STATE
0 k8s-10-6-162-22 524M 99.4G 0 0 0 89 exists,up
1 k8s-10-6-162-23 677M 99.3G 0 0 0 0 exists,up
2 k8s-10-6-162-22 501M 99.5G 0 0 0 0 exists,up
3 k8s-10-6-162-24 694M 99.3G 0 0 1 15 exists,up
4 k8s-10-6-162-24 744M 99.2G 0 0 0 0 exists,up
5 k8s-10-6-162-23 747M 99.2G 0 0 0 0 exists,up
```

Ceph 存储配置

创建 RBD 池和归置组 (PG):

```
[root@rook-ceph-tools-55f548895f-8pdhm /]# ceph osd pool create replicapool 128

[root@k8s-10-6-162-21 kubernetes]# ceph osd pool application enable replicapool rbd
```

Creating storage class:

```
[root@k8s-10-6-162-21 rbd]# pwd
/root/rook/deploy/examples/csi/rbd
[root@k8s-10-6-162-21 rbd]# kubectl apply -f storageclass.yaml

[root@k8s-10-6-162-21 ~]# kubectl get sc
NAME PROVISIONER RECLAIMPOLICY VOLUMEBINDINGMOD
E ALLOWVOLUMEEXPANSION AGE
rook-ceph-block rook-ceph.rbd.csi.ceph.com Delete Immediate true 5d20h
rook-cephfs rook-ceph.cephfs.csi.ceph.com Delete Immediate true 2d17h
```

应用部署验证

在 RBD 上部署 Mysql 和 WordPress

```
[root@k8s-10-6-162-21 examples]# cd kubernetes/
ceph mysql2-cephfs.yaml mysql.yaml README.md wordpress.yaml
[root@k8s-10-6-162-21 kubernetes]# pwd
/root/rook/cluster/examples/kubernetes
[root@k8s-10-6-162-21 kubernetes]# kubectl apply -f mysql.yaml
[root@k8s-10-6-162-21 kubernetes]# kubectl apply -f wordpress.yaml
```

在 Cephfs 上部署 Mysql

```
[root@k8s-10-6-162-21 kubernetes]# kubectl apply -f mysql2-cephfs.yaml

[root@k8s-10-6-162-21 kubernetes]# kubectl get po
NAME                                                              READY S
TATUS RESTARTS AGE
nginx-85b98978db-gbjtb                                            1/1 Running 0 6d18h
nginx-85b98978db-rbprt                                            1/1 Running 0 6d18h
nginx-85b98978db-w5m24                                            1/1 Running 0 6d18h
wordpress-b98c66fff-2bh8n                                         1/1 Running 5 (6d17h ago) 6d17h
wordpress-mysql-79966d6c5b-wqv9b                                  1/1 Running 0 6d17h
wordpress-mysql2-6857bbcf7d-2rl9w                                 1/1 Running 0 3d19h

[root@k8s-10-6-162-21 kubernetes]# kubectl get svc
NAME                    TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes              ClusterIP 10.96.0.1 <none> 443/TCP 22d
nginx                   NodePort 10.107.222.243 <none> 80: 31090/TCP 21d
wordpress               LoadBalancer 10.109.11.61 <pending> 80: 32249/TCP 6d23h
wordpress-mysql         ClusterIP None <none> 3306/TCP 6d23h
wordpress-mysql2        ClusterIP None <none> 3306/TCP 3d19h

[root@rook-ceph-tools-55f548895f-8pdhm /]# ceph osd pool ls
device_health_metrics
replicapool
myfs-metadata
myfs-data0
                                        dashboard-4
dashboard-4
                                        dashboard-5
dashboard-5
```

dashboard-6

dashboard-6

```
[root@k8s-10-6-162-21 kubernetes]# kubectl get pv,pvc
NAME                                                        CAPACITY   ACCESS
 MODES   RECLAIM POLICY   STATUS   CLAIM                    STORAGECLASS   REASON   AGE
persistentvolume/pvc-19515505-cd8f-41d1-ae91-bb42c3eb64f3   20Gi       RWO
 Delete          Bound    default/mysql-pv-claim2  rook-cephfs             3d20h
persistentvolume/pvc-7647bc80-febc-4299-a62a-8446d2c364c6   20Gi       RWO
 Delete          Bound    default/wp-pv-claim  rook-ceph-block            6d23h
persistentvolume/pvc-b07feec8-adc8-4e22-abd6-177b5db4fdb1   20Gi       RWO
 Delete          Bound    default/mysql-pv-claim  rook-ceph-block         7d

NAME                                          STATUS   VOLUME
CAPACITY   ACCESS   M
ODES   STORAGECLASS   AGE
persistentvolumeclaim/mysql-pv-claim          Bound    pvc-b07feec8-adc8-4e22-abd6-177b5db4fdb1   20
Gi   RWO   rook-ceph-block   7d
persistentvolumeclaim/mysql-pv-claim2         Bound    pvc-19515505-cd8f-41d1-ae91-bb42c3eb64f3   20
Gi   RWO   rook-cephfs   3d20h
persistentvolumeclaim/wp-pv-claim             Bound    pvc-7647bc80-febc-4299-a62a-8446d2c364c6   2
0Gi   RWO   rook-ceph-block   6d23h
```

成功部署后，查看 wordpress/mysql 上的博客：

blog-1

blog-1

还在博客上做了评论：

blog-2

blog-2

数据持久性验证

删除并重启 mysql / wordpress Pod：

```
[root@k8s-10-6-162-21 kubernetes]# kubectl delete po wordpress-mysql-79966d6c5b-wc6fs
```

blog-3

blog-3

通过应用商店部署 Rook-ceph

本文将提供在 DCE 5.0 应用商店 Addon 的图形化界面安装、部署 Rook-ceph 云原生存储

系统的操作步骤及说明。

Rook-ceph helm chart 格式转换

添加 repo

```
[root@k8s-10-6-162-31 helm-test]# helm search repo rook

NAME CHART VERSION APP VERSION DESCRIPTION
rook-release/rook-ceph v1.10.5 v1.10.5 File, Block, and Object Storage Services for yo...
rook-release/rook-ceph-cluster v1.10.5 v1.10.5 Manages a single Ceph cluster namespace for Ro ok
stable/rookout 0.1.2 1.0 DEPRECATED - A Helm chart for Rookout agent on ...
```

拉取 rook-ceph helm chart 并解压

```
[root@k8s-10-6-162-31 helm-test]# helm pull rook-release/rook-ceph
[root@k8s-10-6-162-31 helm-test]# helm pull rook-release/rook-ceph-cluster
[root@k8s-10-6-162-31 helm-test]# ls
rook-ceph-cluster-v1.10.5.tgz rook-ceph-v1.10.5.tgz

[root@k8s-10-6-162-31 helm-test]# tar xvfz rook-ceph-v1.10.5.tgz

[root@k8s-10-6-162-31 helm-test]# ls
rook-ceph rook-ceph-cluster-v1.10.5.tgz rook-ceph-v1.10.5.tgz
```

将 rook-ceph 的 values.yaml 转成 json 格式

```
[root@k8s-10-6-162-31 ~]# helm plugin install https: //github.com/karuppiah7890/helm-schema-ge n.git
karuppiah7890/helm-schema-gen info checking GitHub for tag '0.0.4'
karuppiah7890/helm-schema-gen info found version: 0.0.4 for 0.0.4/Linux/x86_64
karuppiah7890/helm-schema-gen info installed ./bin/helm-schema-gen
Installed plugin: schema-gen

[root@k8s-10-6-162-31 helm-test]# cd rook-ceph
charts Chart.yaml README.md templates values.yaml
[root@k8s-10-6-162-31 rook-ceph]# helm schema-gen values.yaml > values.schema.json

[root@k8s-10-6-162-31 rook-ceph]# ls
charts Chart.yaml prometheus README.md values.schema.json templates values.yaml
```

[root@k8s-10-6-162-31 helm-test]# tar xvfz rook-ceph-cluster-v1.10.5.tgz

[root@k8s-10-6-162-31 rook-ceph-cluster]# ls
charts Chart.yaml prometheus README.md templates values.yaml
[root@k8s-10-6-162-31 rook-ceph-cluster]# helm schema-gen values.yaml > values.schema.json

[root@k8s-10-6-162-31 rook-ceph-cluster]# ls
charts Chart.yaml prometheus README.md values.schema.json templates values.yaml

将含有 json 文件的 chart 打包压缩

[root@k8s-10-6-162-31 helm-test]# tar zcvf rook-ceph-v1.10.5.tgz rook-ceph

[root@k8s-10-6-162-31 helm-test]# tar zcvf rook-ceph-cluster-v1.10.5.tgz rook-ceph-cluster

[root@k8s-10-6-162-31 helm-test]# ls
rook-ceph-cluster-v1.10.5.tgz rook-ceph-v1.10.5.tgz rook-ceph rook-ceph-cluster rook-ceph-cluster-v1.10.5.tgz rook-ceph-v1.10.5.tgz

DaoCloud 镜像仓库操作

将含有 json 格式的 chart 包上传

进入镜像仓库

进入镜像仓库

镜像仓库上传 chart 包

镜像仓库上传 chart 包

镜像仓库上传 chart 包

DCE 5.0 集群安装 Rook-ceph

将集群接入 DCE 5.0

集群接入

集群接入

集群接入

集群接入

安装 rook-ceph

rook-ceph 安装

rook-ceph 安装

rook-ceph 安装

rook-ceph 安装

rook-ceph 组件

rook-ceph 组件

rook-ceph-sc

rook-ceph-sc

应用部署验证

```
[root@k8s-10-6-162-31 kubernetes]# pwd
/root/rook/cluster/examples/kubernetes

[root@k8s-10-6-162-31 kubernetes]# kubectl apply -f mysql.yaml
service/wordpress-mysql created
persistentvolumeclaim/mysql-pv-claim created
deployment.apps/wordpress-mysql created
[root@k8s-10-6-162-31 kubernetes]# kubectl get svc -A
NAMESPACE NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
```

```
default kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 31d
default wordpress-mysql ClusterIP None <none> 3306/TCP 19s
kube-system kube-dns ClusterIP 10.96.0.10 <none> 53/UDP,53/TCP,9153/TCP 31d
rook-ceph rook-ceph-mgr ClusterIP 10.100.204.145 <none> 9283/TCP 18h
rook-ceph rook-ceph-mgr-dashboard ClusterIP 10.96.206.31 <none> 8443/TCP 18h
rook-ceph rook-ceph-mon-a ClusterIP 10.101.168.203 <none> 6789/TCP,3300/TCP 18h
rook-ceph rook-ceph-mon-b ClusterIP 10.102.39.21 <none> 6789/TCP,3300/TCP 18h
rook-ceph rook-ceph-mon-c ClusterIP 10.109.128.35 <none> 6789/TCP,3300/TCP 18h
rook-ceph rook-ceph-rgw-ceph-objectstore ClusterIP 10.108.107.0 <none> 80/TCP 18h

[root@k8s-10-6-162-31 kubernetes]# kubectl get po -owide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS G
ATES
wordpress-mysql-79966d6c5b-5v2r4 1/1 Running 0 12m 10.244.19.148 k8s-10-6-162-31 <none>
<none>

[root@k8s-10-6-162-31 kubernetes]# kubectl get pv -A
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLAS
S REASON AGE
pvc-9c7558cc-a152-4893-a88d-5054ab76e73e 20Gi RWO Delete Bound default/mysql-pv-claim c
eph-block 36s

[root@k8s-10-6-162-31 kubernetes]# kubectl get po -owide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS G
ATES
wordpress-mysql-79966d6c5b-5v2r4 1/1 Running 0 12m 10.244.19.148 k8s-10-6-162-31 <none>
<none>
```

至此，Rook-ceph 在 DCE 5.0 Add-on 应用商店的部署安装验证测试完成！

通过 Helm 部署 Rook-Ceph

本文将提供用 Helm 部署 Rook-Ceph 云原生存储系统的操作步骤及说明。

Helm 安装

```
[root@k8s-10-6-162-31 ~]# wget https: //get.helm.sh/helm-v3.10.1-linux-amd64.tar.gz

[root@k8s-10-6-162-31 ~]# tar xvfz helm-v3.10.1-linux-amd64.tar.gz
linux-amd64/
linux-amd64/helm
```

linux-amd64/LICENSE
linux-amd64/README.md
[root@k8s-10-6-162-31 ~]#
[root@k8s-10-6-162-31 ~]#
[root@k8s-10-6-162-31 ~]# ls
anaconda-ks.cfg calico.yaml helm-v3.10.1-linux-amd64.tar.gz linux-amd64 rook rook-ceph-image rook-ceph-image.zip
[root@k8s-10-6-162-31 ~]# cd linux-amd64
[root@k8s-10-6-162-31 linux-amd64]# ls
helm LICENSE README.md
[root@k8s-10-6-162-31 linux-amd64]# mv helm /usr/bin
[root@k8s-10-6-162-31 linux-amd64]#
[root@k8s-10-6-162-31 linux-amd64]# cd /root
[root@k8s-10-6-162-31 ~]# helm
The Kubernetes package manager

Common actions for Helm:

- helm search: search for charts
- helm pull: download a chart to your local directory to view
- helm install: upload the chart to Kubernetes
- helm list: list releases of charts

添加 rook repo

helm repo add rook-release https://charts.rook.io/release

[root@k8s-10-6-162-31 ~]# helm repo list
NAME URL
rook-release https://charts.rook.io/release
stable http://mirror.azure.cn/kubernetes/charts/
aliyun https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts

[root@k8s-10-6-162-31 ~]# helm search repo rook-ceph
NAME CHART VERSION APP VERSION DESCRIPTION
rook-release/rook-ceph v1.10.5 v1.10.5 File, Block, and Object Storage Services for yo...
rook-release/rook-ceph-cluster v1.10.5 v1.10.5 Manages a single Ceph cluster namespace for Rook

Helm 安装 rook operator

pwd[root@k8s-10-6-162-31 ~]# helm install --namespace rook-ceph rook-ceph rook-release/rook-ceph --create-namespace --set image.repository=rook/ceph --set csi.cephcsi.image=quay.io/cephcsi/cephcsi: v3.7.2 --set csi.registrar.image=registry.k8s.io/sig-storage/csi-node-driver-registrar:v2.5.1 --set csi.provisioner.image=registry.k8s.io/sig-storage/csi-provisioner:v3.3.0 --set csi.snapshotter.image=registry.k8s.io/sig-storage/csi-snapshotter:v6.1.0 --set csi.attacher.image=registry.k8s.io/sig-storage/csi-attacher:v4.0.0 --set csi.resizer.image=registry.k8s.io/sig-storage/csi-resizer:v1.6.0

[root@k8s-10-6-162-31 ~]# helm ls -A
NAME NAMESPACE REVISION UPDATED
STATUS CHART APP VERSION
rook-ceph rook-ceph 1 2022-11-07 13: 58:04.376723834 +0800 CST
deployed rook-ceph-v1.10.5 v1.10.5

[root@k8s-10-6-162-31 ~]# kubectl get po -n rook-ceps
NAMESPACE NAME READY
STATUS RESTARTS AGE
rook-ceph rook-ceph-operator-964d7fbbd-j2krp 1/1 R
unning 0 39m

Helm 安装 rook-ceph cluster 及 ceph tool

[root@k8s-10-6-162-31 ~]# helm install --namespace rook-ceph rook-ceph-cluster rook-release/rook-ceph-cluster --set operatorNamespace=rook-ceph --set cephClusterSpec.storage.deviceFilter="^sd." --set cephClusterSpec.cephVersion.image=quay.io/ceph/ceph: v17.2.3
NAME: rook-ceph-cluster
LAST DEPLOYED: Mon Nov 7 14:49:28 2022
NAMESPACE: rook-ceph
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The Ceph Cluster has been installed. Check its status by running:
kubectl --namespace rook-ceph get cephcluster

Visit https: //rook.io/docs/rook/latest/CRDs/ceph-cluster-crd/ for more information about the Ceph CRD.

Important Notes:
- You can only deploy a single cluster per namespace
- If you wish to delete this cluster and start fresh, you will also have to wipe the OSD disks using `sfdisk`

```
[root@k8s-10-6-162-31 examples]# helm ls -A
NAME                    NAMESPACE       REVISION        UPDATED
                                        STATUS          CHART                   APP
VERSION
rook-ceph               rook-ceph       1               2022-11-07 13:58:04.3767
23834 +0800 CST         deployed        rook-ceph-v1.10.5       v1.10.5
rook-ceph-cluster       rook-ceph       1               2022-11-07 14:49:28.709538
725 +0800 CST           deployed        rook-ceph-cluster-v1.10.5       v1.10.5

[root@k8s-10-6-162-31 ceph]# kubectl create -f toolbox.yaml
deployment.apps/rook-ceph-tools created

[root@k8s-10-6-162-31 examples]# kubectl get po -n rook-ceph
NAME                                                    READY
        STATUS          RESTARTS        AGE
csi-cephfsplugin-6jxvj                                  2/2             R
unning          0               90m
csi-cephfsplugin-h68jt                                  2/2             R
unning          0               90m
csi-cephfsplugin-provisioner-785d976fcf-p96s9           5/5             Run
ning            0               30m
csi-cephfsplugin-provisioner-785d976fcf-w7j7p           5/5             Run
ning            0               90m
csi-cephfsplugin-rzq9d                                  2/2
Running         0               90m
csi-rbdplugin-7jcgf                                     2/2
Running         0               90m
csi-rbdplugin-7z4v8                                     2/2
Running         0               90m
csi-rbdplugin-8l6c9                                     2/2
Running         0               90m
csi-rbdplugin-provisioner-dc499c7dd-62d2p               5/5             Ru
nning           2               90m
csi-rbdplugin-provisioner-dc499c7dd-kv6c5               5/5             Ru
nning           0               30m
rook-ceph-crashcollector-k8s-10-6-162-31-67678cc9f9-8ctkb       1/1             Runni
ng              0               46m
rook-ceph-crashcollector-k8s-10-6-162-32-8655c84654-9mj22       1/1             Run
ning            0               18m
rook-ceph-crashcollector-k8s-10-6-162-33-5dbcc6965c-8dk89       1/1             Runn
ing             0               46m
rook-ceph-mds-ceph-filesystem-a-b8fdd6876-s8s94         2/2             Ru
nning           0               30m
```

```
rook-ceph-mds-ceph-filesystem-b-fd667b865-49vx2 2/2 Ru
nning 0 32m
rook-ceph-mgr-a-5dbd5b49d5-2thts 3/3
Running 0 46m
rook-ceph-mgr-b-6bbc5966c8-gt2fg 3/3
Running 0 30m
rook-ceph-mon-a-56d4476dfd-hjr6b 2/2
Running 1 (47m ago) 89m
rook-ceph-mon-b-5978cc6657-zd2tb 2/2
 Running 0 86m
rook-ceph-mon-c-5d94c6c9cc-9s6p9 2/2
 Running 0 63m
rook-ceph-operator-964d7fbbd-d6s6w 1/1
Running 0 30m
rook-ceph-osd-0-79467bcb49-qtf6g 2/2
Running 0 30m
rook-ceph-osd-1-64c5945d78-72z4q 2/2
Running 0 30m
rook-ceph-osd-2-f68b5bcb7-59nwb 2/2
 Running 0 30m
rook-ceph-osd-3-99c576b7-j7prn 2/2
Running 0 30m
rook-ceph-osd-4-579d5f966f-cqw5f 2/2
Running 0 30m
rook-ceph-osd-5-655d7bfbd7-kztj2 2/2
Running 0 30m
rook-ceph-osd-prepare-k8s-10-6-162-31-tzz2f 0/1 Co
mpleted 0 23m
rook-ceph-osd-prepare-k8s-10-6-162-32-7blmc 0/1 Co
mpleted 0 22m
rook-ceph-osd-prepare-k8s-10-6-162-33-pjjmd 0/1 Co
mpleted 0 22m
rook-ceph-rgw-ceph-objectstore-a-6d8b954f7d-jx9zx 2/2 Run
ning 0 18m
rook-ceph-tools-7c8ddb978b-2mf52 1/1
Running 0 40m
```

点击此处查看 toolbox.yaml

Ceph 工具验证

```
[root@k8s-10-6-162-31 ~]# kubectl exec -it rook-ceph-tools-7c8ddb978b-2mf52 -n rook-ceph --
bash
bash-4.4$
```

```
bash-4.4$ ceph -s
cluster:
id: fcba483d-2c38-48cd-9137-fb93be678383
health: HEALTH_WARN
1 MDSs report slow metadata IOs
Reduced data availability: 6 pgs inactive

services:
mon: 3 daemons, quorum a,b,c (age 16m)
mgr: a(active, since 115s), standbys: b
mds: 1/1 daemons up, 1 standby
osd: 6 osds: 0 up, 6 in (since 96s)

data:
volumes: 1/1 healthy
pools: 6 pools, 6 pgs
objects: 0 objects, 0 B
usage: 0 B used, 0 B / 0 B avail
pgs: 100.000% pgs unknown
6 unknown
```

至此，用 Helm 部署安装 rook-ceph 完成。

如何实现 Ceph Dashboard 仪表盘

本页演示如何在 DCE 5.0 中导入并成功使用 Ceph 监控面板。

在 DCE 5.0 中部署 Rook-ceph

先部署 Rook-ceph，再部署 rook-ceph-cluster。

- 由于目前 Ceph 还没离线化支持，所以需要在工作集群的所在节点上增加代理（测试中是在 demo-dev 环境以及内网搭建的工作集群）

```
ip r add default via 10.6.102.1 dev ens192
```

- 部署 rook-ceph-cluster 时由于工作节点数量有限，还需要设置：allowMultiplePerNode: true

在工作集群中部署 Insight Agent

采集 rook-ceph-cluster 的监控指标需要先安装 Insight Agent， 然后创建 CR ServiceMonitor 来采集 rook-ceph-cluster 的监控信息。

1. rook-ceph-cluster 的监控指标通过 9283 端口暴露。

   port 9283

   port 9283

2. 在工作集群中为 rook-ceph-mgr 创建 ServiceMonitor。

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    operator.insight.io/managed-by: insight
  name: rook-ceph-sm
  namespace: rook-ceph
spec:
  endpoints:
    - honorLabels: true
      port: http-metrics
  namespaceSelector:
    any: true
  selector:
    matchLabels:
      app: rook-ceph-mgr
      rook_cluster: rook-ceph
```

在全局服务集群中部署 GrafanaDashboard

参考 Dashboard 模板部署 GrafanaDashboard。

导入模板时请参阅 Insight 导入仪表盘。

```yaml
apiVersion: integreatly.org/v1alpha1
kind: GrafanaDashboard
metadata:
  labels:
    app: insight-grafana-operator
    operator.insight.io/managed-by: insight
  name: ceph-dashboard
```

```
  namespace: insight-system
spec:
  json: >
    {
        "__inputs": [],
        "__elements": {},
        "__requires": [
        {
            "type": "panel",
            "id": "gauge",
            "name": "Gauge",
            "version": ""
        },
        {
            "type": "grafana",
            "id": "grafana",
            "name": "Grafana",
            "version": "9.0.5"
        },
....
```

查看 Ceph 监控面板

目前的面板还未区分集群，后续将陆续优化增加 cluster 标识选项。

监控面板

监控面板

创建 Ceph 文件存储(CephFS) 存储类

前提条件

参考文档通过应用商店部署 Rook-ceph 通过应用商店安装 rook-ceph、rook-ceph-cluster。

操作步骤

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储** -> **存储池(SC)** -> **创建存储池(SC)** 。

2. 填写基本信息，参数说明如下：

   - 存储池名称、CSI 驱动、回收策略、磁盘绑定模式在创建后不可修改。
   - CSI 存储驱动：输入 rook-ceph.cephfs.csi.ceph.com。
   - 自定义参数定义如下内容：

clusterID	rook-ceph	运行 rook-ceph 集群的命名空间
csi.storage.k8s.io/fstype	ext4	指定卷的文件系统类型。如果未指定，csi-provisioner 将默认设置为 “ext4”，不推荐使用 ”xfs”
csi.storage.k8s.io/controller-expand-secret-name	rook-csi-rbd-provisioner	创建卷时 CSI provisioner 使用的 Kubernetes Secret 的名称
csi.storage.k8s.io/controller-expand-secret-namespace	rook-ceph	上述 Secret 所在的命名空间
csi.storage.k8s.io/node-stage-secret-name	rook-csi-cephfs-node	执行卷扩展操作时

clusterID	rook-ceph	运行 rook-ceph 集群的命名空间
csi.storage.k8s.io/node-stage-secret-namespace	rook-ceph	CSI 控制器使用的 Secret 的名称 执行卷扩展操作时 Secret 所在的命名空间
csi.storage.k8s.io/provisioner-secret-name	rook-csi-cephfs-provisioner	节点上卷挂载操作时 CSI 节点插件使用的 Secret 的名称
csi.storage.k8s.io/provisioner-secret-namespace	rook-ceph	节点上卷挂载操作时 Secret 所在的命名空间
fsName	ceph-filesystem	定义卷的 CephFS 文件系统名称
pool	ceph-filesystem-data0	定义卷的 Ceph 池名称

fs01

fs01

fs02

fs02

3. 填写完点击 **确定** 即可创建成功

创建 Ceph 块存储(RBD) 存储类

前提条件

参考文档通过应用商店部署 Rook-ceph 通过应用商店安装 rook-ceph、rook-ceph-cluster。

操作步骤

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储** -> **存储池(SC)** -> **创建存储池(SC)**。
2. 填写基本信息，参数说明如下：
   - 存储池名称、CSI 驱动、回收策略、磁盘绑定模式在创建后不可修改。
   - CSI 存储驱动：输入 rook-ceph.rbd.csi.ceph.com。
   - 自定义参数定义如下内容：

参数	值	说明
clusterID	rook-ceph	运行 rook-ceph 集群的命名空间。
csi.storage.k8s.io/fstype	ext4	指定卷的文件系统类型。如果未指定，csi-provisioner 将默认设置为 “ext4”，官方文档不推荐使用 ”xfs”。
csi.storage.k8s.io/controller-expand-secret-n	rook-csi-rbd-provisioner	指定了执行卷扩展操

参数	值	说明
ame		作时 CSI 控制器使用的 Secret 的名称
csi.storage.k8s.io/controller-expand-secret-namespace	rook-ceph	指定了执行卷扩展操作时 Secret 所在的命名空间
csi.storage.k8s.io/node-stage-secret-name	rook-csi-rbd-node	指定了节点上卷挂载操作时 CSI 节点插件使用的 Secret 的名称
csi.storage.k8s.io/node-stage-secret-namespace	rook-ceph	指定了节点上卷挂载操作时 Secret 所在的命名空间
csi.storage.k8s.io/provisioner-secret-name	rook-csi-rbd-provisioner	指定了创建卷时 CSI provisioner 使用的 Kubernetes Secret 的名称
csi.storage.k8s.io/provisioner-secret-namespace	rook-ceph	指定了上述 Secret 所在的命名空间
imageFeatures	layering	指定创建的块设备（block device）支持的特性。layering 是

参数	值	说明
imageFormat	2	其中一种特性，它允许块设备支持快照功能。适用于 imageFormat 为“2”的情况。 Ceph 块存储的映像（image）格式，默认为 2，且是推荐的格式，它支持许多高级特性，如快照、克隆和动态调整大小等。
pool	ceph-blockpool	定义 Ceph 集群存储池的名称，将被用于存储数据。

block01

block01

block02

block02

3. 填写完点击 **确定** 即可创建成功。

创建 Ceph 对象存储(RGW) 存储类

前提条件

参考文档通过应用商店部署 Rook-ceph 通过应用商店安装 rook-ceph、rook-ceph-cluster。

操作步骤

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储** -> **存储池(SC)** -> **创建存储池(SC)**。
2. 填写基本信息，参数说明如下：
   - 存储池名称、CSI 驱动、回收策略、磁盘绑定模式在创建后不可修改。
   - CSI 存储驱动：输入 rook-ceph.ceph.rook.io/bucket。
   - 自定义参数定义如下内容：

参数	值	说明
objectStoreName	ceph-objectstore	指定对象存储的名称
objectStoreNamespace	rook-ceph	指定对象存储实例所在的 Kubernetes 命名空间
region	us-east-1	通常用于指定对象存储实例的地理区域
	rgw01	
rgw01		

3. 填写完点击 **确定** 即可创建成功。

Longhorn 存储方案

DCE 5.0 支持众多第三方存储方案，我们针对 Longhorn 进行了详细的测试，并最终将其作为 Addon 集成了应用商店中。 以下是对 Longhorn 的简单调研和测评报告。

Longhorn 是一个轻量级的云原生 Kubernetes 分布式存储平台，可以在任意基础设施上运行。Longhorn 与 DCE 可以结合使用部署高可用性持久化块存储。

1. 设计与架构
   - 控制平面：Longhorn Manager 以 DaemonSet 部署
   - 数据平面：Longhorn Engine 是 storage controller 可以有多个 replicas

架构图

架构图

2. Longhorn 存储卷
   - 支持存储卷的 Thin provisioning
   - 存储卷维护模式 maintenance mode 用于 snapshot reverting operation 操作
   - 每一个存储卷的 replica 包含多个快照 snapshots.
   - 默认的存储卷的 replica 数量可以在 settings 设置. 当存储卷被挂载以后 replica 数量可以通过 UI 改变。
   - Longhorn 是一个 crash-consistent 块存储方案，在创建快照 snapshot 之前会自动先同步 sync 命令

存储卷

存储卷

3. 数据备份及外部二级存储

- 用于备份的 NFS/S3 兼容的外部二级存储是独立于 Kubernetes 集群以外的。即使在 Kubernetes 集群不可用的情况下数据依然可用
- Longhorn 也会将存储卷同步到灾备集群（DR）的二级存储用于灾难数据恢复
- 备份是多个快照数据的扁平化的集合。
- 支持连续重复快照及备份。
- 支持 CSI 存储卷的 Clone

数据备份

数据备份

4. 高可用 High Availability
   - 支持 Replica 自动平衡设置
   - 支持 data locality setting：使用存储卷的 Pod 运行的节点上至少有一个 replica 副本
   - 支持显示节点存储空间使用。
   - 支持 Kubernetes Cluster Autoscaler (Experimental)
   - 支持存储卷意外卸载后的的自动恢复
   - 支持集群节点失效后的存储卷的自动恢复
5. 监控 Monitoring
   - 支持 Prometheus 和 Grafana 监控 Longhorn
   - Longhorn 指标可以被整合进 DCE 监控系统
   - 支持 kubelet 指标监控
   - 支持 Longhorn 告警策略
6. 高级功能

- 支持镜像备份
- 支持 Orphaned Replica Directories
- 支持 DCE 集群恢复：集群所有存储卷的恢复
- 支持多写操作 ReadWriteMany (RWX) workloads （NFSv4）
- 支持 Longhorn Volume 作为 iSCSI Target

通过应用商店部署 Longhorn

本文将提供在 DCE 5.0 应用商店以 Addon 的图形化界面安部署 Longhorn 云原生存储系统的操作步骤及说明。

Longhorn helm chart 格式转换

添加 repo

[root@k8s-10-6-162-31 ~]# helm search repo longhorn

NAME CHART VERSION APP VERSION DESCRIPTION
longhorn/longhorn 1.3.2 v1.3.2 Longhorn is a distributed block storage system ...

拉取 Longhorn helm chart 并解压

[root@k8s-10-6-162-31 ~]# helm pull longhorn/longhorn
[root@k8s-10-6-162-31 ~]# ls

anaconda-ks.cfg calico.yaml helm-v3.10.1-linux-amd64.tar.gz linux-amd64 longhorn-1.3.2.tgz rook rook-ceph-image rook-ceph-image.zip

[root@k8s-10-6-162-31 ~]# tar xvfz longhorn-1.3.2.tgz

longhorn/Chart.yaml
longhorn/values.yaml
longhorn/templates/NOTES.txt

longhorn/templates/_helpers.tpl
longhorn/templates/clusterrole.yaml
longhorn/templates/clusterrolebinding.yaml
longhorn/templates/crds.yaml
longhorn/templates/daemonset-sa.yaml
longhorn/templates/default-setting.yaml
longhorn/templates/deployment-driver.yaml
longhorn/templates/deployment-ui.yaml
longhorn/templates/deployment-webhook.yaml
longhorn/templates/ingress.yaml
longhorn/templates/postupgrade-job.yaml
longhorn/templates/psp.yaml
longhorn/templates/registry-secret.yaml
longhorn/templates/serviceaccount.yaml
longhorn/templates/services.yaml
longhorn/templates/storageclass.yaml
longhorn/templates/tls-secrets.yaml
longhorn/templates/uninstall-job.yaml
longhorn/.helmignore
longhorn/README.md
longhorn/app-readme.md
longhorn/questions.yaml

[root@k8s-10-6-162-31 ~]# cd longhorn
[root@k8s-10-6-162-31 longhorn]# ls

app-readme.md Chart.yaml questions.yaml README.md templates values.yaml

将 Longhorn 的 values.yaml 转成 json 格式

[root@k8s-10-6-162-31 ~]# helm plugin install https: //github.com/karuppiah7890/helm-schema-gen.git

karuppiah7890/helm-schema-gen info checking GitHub for tag '0.0.4'
karuppiah7890/helm-schema-gen info found version: 0.0.4 for 0.0.4/Linux/x86_64
karuppiah7890/helm-schema-gen info installed ./bin/helm-schema-gen
Installed plugin: schema-gen

[root@k8s-10-6-162-31 longhorn]# helm schema-gen values.yaml > values.schema.json
[root@k8s-10-6-162-31 longhorn]# ls

app-readme.md Chart.yaml questions.yaml README.md templates values.schema.json values.yaml

将含有 json 文件的 chart 打包压缩

[root@k8s-10-6-162-31 longhorn]# cd ..
[root@k8s-10-6-162-31 ~]# ls

anaconda-ks.cfg helm-v3.10.1-linux-amd64.tar.gz longhorn rook rook-ceph-image.zip
calico.yaml linux-amd64 longhorn-1.3.2.tgz rook-ceph-image values.schema.json

[root@k8s-10-6-162-31 ~]# tar zcvf longhorn-v1.3.2.tgz longhorn

longhorn/
longhorn/Chart.yaml
longhorn/values.yaml
longhorn/templates/
longhorn/templates/NOTES.txt
longhorn/templates/_helpers.tpl
longhorn/templates/clusterrole.yaml
longhorn/templates/clusterrolebinding.yaml
longhorn/templates/crds.yaml
longhorn/templates/daemonset-sa.yaml
longhorn/templates/default-setting.yaml
longhorn/templates/deployment-driver.yaml
longhorn/templates/deployment-ui.yaml
longhorn/templates/deployment-webhook.yaml
longhorn/templates/ingress.yaml
longhorn/templates/postupgrade-job.yaml
longhorn/templates/psp.yaml
longhorn/templates/registry-secret.yaml
longhorn/templates/serviceaccount.yaml
longhorn/templates/services.yaml
longhorn/templates/storageclass.yaml
longhorn/templates/tls-secrets.yaml
longhorn/templates/uninstall-job.yaml
longhorn/.helmignore
longhorn/README.md
longhorn/app-readme.md
longhorn/questions.yaml
longhorn/values.schema.json

[root@k8s-10-6-162-31 ~]# ls

anaconda-ks.cfg helm-v3.10.1-linux-amd64.tar.gz longhorn longhorn-v1.3.2.tgz rook-ceph-image v
alues.schema.json
calico.yaml linux-amd64 longhorn-1.3.2.tgz rook rook-ceph-image.zip

上传 chart 包至 DCE 5.0 镜像仓库

dce 镜像仓库-1

dce 镜像仓库-1

dce 镜像仓库-2

dce 镜像仓库-2

dce 应用商店-1

dce 应用商店-1

DCE 5.0 应用商店安装 Longhorn

dce 应用商店-2

dce 应用商店-2

[root@k8s-10-6-162-31 ~]# kubectl get po -n longhorn-system

```
NAME                                        READY   STATUS
  RESTARTS          AGE
csi-attacher-7bf4b7f996-7g2h6               1/1     Running       0
        3h13m
csi-attacher-7bf4b7f996-d996x               1/1     Running       1 (108m a
go)     3h13m
csi-attacher-7bf4b7f996-nxwcw               1/1     Running       0
        3h13m
csi-provisioner-869bdc4b79-4xgbs            1/1     Running       1 (108m a
go)     3h13m
csi-provisioner-869bdc4b79-8bjb5            1/1     Running       0
        3h13m
csi-provisioner-869bdc4b79-zmqkp            1/1     Running       0
        3h13m
csi-resizer-6d8cf5f99f-lfvhq                1/1     Running       1 (108m a
go)     3h13m
csi-resizer-6d8cf5f99f-lzkd4                1/1     Running       0
        3h13m
csi-resizer-6d8cf5f99f-zcd6g                1/1     Running       0
        3h13m
```

```
csi-snapshotter-588457fcdf-crddw                                  1/1     Running     1 (108m
ago)   3h13m
csi-snapshotter-588457fcdf-qnghk                                  1/1     Running     0
       3h13m
csi-snapshotter-588457fcdf-xbl4q                                  1/1     Running     0
       3h13m
engine-image-ei-a5371358-qtbzf                                    1/1     Running     0
       3h14m
engine-image-ei-a5371358-sgrkv                                    1/1     Running     0
       3h14m
engine-image-ei-a5371358-t8gd6                                    1/1     Running     0
       3h14m
helm-operation-install-longhorn-deploy-jf5rm-hdx82                0/1     Completed   0
     3h20m
instance-manager-e-2894727b                                       1/1     Running     0
       3h14m
instance-manager-e-c285811f                                       1/1     Running     0
       3h14m
instance-manager-e-c33ef405                                       1/1     Running     0
       3h14m
instance-manager-r-5eed6e09                                       1/1     Running     0
       3h14m
instance-manager-r-7aea1541                                       1/1     Running     0
       3h14m
instance-manager-r-832995c5                                       1/1     Running     0
       3h14m
longhorn-admission-webhook-84dbdf4b-7k9pf                         1/1     Running     0
       3h20m
longhorn-admission-webhook-84dbdf4b-w9xjz                         1/1     Running     0
       3h20m
longhorn-conversion-webhook-77f447c97b-fj52b                      1/1     Running     0
       3h20m
longhorn-conversion-webhook-77f447c97b-fxqck                      1/1     Running     0
       3h20m
longhorn-csi-plugin-mvwwd                                         2/2     Running     0
       3h13m
longhorn-csi-plugin-nj6pg                                         2/2     Running     0
       3h13m
longhorn-csi-plugin-t8smg                                         2/2     Running     0
       3h13m
longhorn-driver-deployer-7cd5cfcd64-lsjnm                         1/1     Running     0
       3h20m
longhorn-manager-lwrxj                                            1/1     Running     0
       3h20m
```

```
longhorn-manager-x7vgm                                  1/1     Running   0
                3h20m
longhorn-manager-xsn8k                                  1/1     Running   1  (3h
14m  ago)    3h20m
longhorn-ui-68bc57db67-46brf                            1/1     Running   0
                3h20m
```

修改 Longhorn 前端 service 端口：

longhorn-svc-1

longhorn-svc-1

longhorn-svc-2

longhorn-svc-2

进入 Longhorn UI：

longhorn 界面

longhorn 界面

longhorn 界面

longhorn 界面

至此，在 DCE 5.0 应用商店成功部署了 Longhorn 存储系统！

OpenEBS 存储方案

DCE 5.0 支持众多第三方存储方案，我们针对 OpenEBS 进行了相关测试，并最终将其作为 Addon 集成了应用商店中。 以下是对 OpenEBS 的调研和测评。

有关应用商店 Addon 的图形化界面安装、部署、卸载等操作说明，将于稍后提供。

什么是 CAS

容器附加存储 (CAS)是一种包含由 Kubernetes 编排的基于微服务的存储控制器的软件。 这些存储控制器可在 Kubernetes 能运行的任何地方运行，包括任何云、裸机服务器或传统共享存储系统之上。 至关重要的是，数据本身也可以通过容器访问，而不是存储在平台外共

享的横向扩展存储系统中。

openebs-1

openebs-1

CAS 是一种非常符合非聚合数据的趋势以及运行小型、松散耦合工作负载的小型自治团队。开发人员能保持自主并可以使用 Kubernetes 集群可用的任何存储来启动他们自己的 CAS 容器。

CAS 也反映了更广泛的解决方案趋势：通过在 Kubernetes 和微服务上构建并为基于 Kubernetes 的微服务环境提供功能来重塑特定类别的资源或创建新类别资源。 例如，安全、DNS、网络、网络策略管理、消息传递、跟踪、日志记录等新项目已经出现在云原生生态系统中。

CAS 的优势

敏捷

CAS 中的每个存储卷都有一个容器化存储控制器和对应的容器化副本。 因此，围绕这些组件的资源维护和调整是真正敏捷的。 Kubernetes 的滚动升级能力可以实现存储控制器和存储副本的无缝升级。 CPU 和内存等资源可以使用容器 cGroups 进行调整。

存储策略的粒度

将存储软件容器化并将存储控制器专用于每个卷，可以最大程度地细化存储策略。 使用 CAS 架构，您可以在每个卷的基础上配置所有存储策略。 此外，您可以监控每个卷的存储参数并动态更新存储策略以获得每个工作负载的预期结果。 存储吞吐量、IOPS 和延迟的控制随着卷存储策略中粒度的增加而增加。

避免锁定

避免云供应商锁定是许多 Kubernetes 用户的共同目标。 然而，有状态应用程序的数据通常仍然依赖于云提供商和技术，或者依赖于底层的传统共享存储系统、NAS 或 SAN。 使用 CAS 方法，存储控制器可以根据工作负载在后台迁移数据，实时迁移变得更加简单。 换句话说，CAS 的控制粒度以一种无中断的方式简化了有状态工作负载从一个 Kubernetes 集群到另一个集群的移动。

云原生

CAS 将存储软件容器化，并使用 Kubernetes 自定义资源定义 (CRD) 来表示底层存储资源，例如磁盘和存储池。 该模型使存储能够无缝集成到其他云原生工具中。 可以使用 Prometheus、Grafana、Fluentd、Weavescope、Jaeger 等云原生工具来配置、监控和管理存储资源。

与超融合系统类似，CAS 中卷的存储和性能是可扩展的。 由于每个卷都有自己的存储控制器，因此存储可以在节点存储容量的允许范围内扩展。 随着给定 Kubernetes 集群中容器应用程序数量的增加，将添加更多节点，从而提高存储容量和性能的整体可用性，从而使存储可用于新的应用程序容器。 这种可扩展性过程类似于 Nutanix 等成功的超融合系统。

OpenEBS 原理及架构

OpenEBS 是容器附加存储 (CAS) 模式的领先开源实现。 作为这种方法的一部分，OpenEBS 使用容器来动态配置卷并提供高可用性等数据服务。 OpenEBS 依赖并扩展 Kubernetes 本身来编排其卷服务。

openebs-2

openebs-2

OpenEBS 有很多组件，可以分为以下两大类：

- OpenEBS 数据引擎
- OpenEBS 控制平面

数据引擎

数据引擎是 OpenEBS 的核心，负责代表它们所服务的有状态工作负载对底层持久存储执行读写操作。

数据引擎负责：

- 聚合分配给它们的块设备中的可用容量，然后为应用程序划分卷。
- 提供标准系统或网络传输接口（NVMe/iSCSI）以连接到本地或远程卷
- 提供卷服务，如同步复制、压缩、加密、维护快照、访问数据的增量或完整快照等
- 在将数据持久化到底层存储设备的同时提供强一致性

OpenEBS 遵循微服务模型来实现数据引擎，其中功能进一步分解为不同的层，允许灵活地交换层并使数据引擎为未来的应用程序和数据中心技术的变化做好准备。

OpenEBS 数据引擎由以下几层组成：

openebs-3

openebs-3

卷访问层

有状态工作负载使用标准的 POSIX 兼容机制来执行读写操作。 根据工作负载的类型，应用程序可能更喜欢直接对原始块设备或使用标准文件系统（如 XFS、Ext4）执行读取和写入。 CSI 节点驱动程序或 Kubelet 将负责将卷附加到 Pod 运行所在的所需节点，必要时进行格式化并挂载文件系统以供 Pod 访问。 用户可以选择在此层设置挂载选项和文件系统权限，

这将由 CSI 节点驱动程序或 kubelet 执行。

附加卷（使用本地、iSCSI 或 NVMe）和安装（Ext4、XFS 等）所需的详细信息可通过持久卷规范获得。

卷服务层

该层通常称为 Volume Target Layer 甚至 Volume Controller 层，因为它负责提供逻辑卷。 应用程序读取和写入是通过 Volume Targets 执行的 - 它控制对卷的访问、将数据同步复制到集群中的其他节点， 并帮助决定哪个副本充当主副本并促进将数据重建到旧的或重新启动的副本。

数据引擎用于提供高可用性的实现模式是 OpenEBS 与其他传统存储控制器的区别。 与使用单个存储控制器在多个卷上执行 IO 不同，OpenEBS 为每个卷创建一个存储控制器（称为 Target/Nexus），并具有将保存卷数据的特定节点列表。 每个节点将具有使用同步复制分发的卷的完整数据。

使用单个控制器将数据同步复制到固定的节点集（而不是通过多个元数据控制器分发），减少了管理元数据的开销，还减少了与节点故障和参与重建的其他节点相关的爆炸半径失败的节点。

OpenEBS 卷服务层将卷公开为：

- 本地 PV 情况下的设备或目录路径，
- cStor 和 Jiva 情况下的 iSCSI 目标
- NVMe Target 在 Mayastor 的情况下

卷数据层

OpenEBS 数据引擎在存储层之上创建卷副本。卷副本被固定到一个节点，并在存储层之上创建。 副本可以是以下任何一项：

- 子目录 - 如果使用的存储层是文件系统目录
- 完整设备或分区设备 - 如果使用的存储层是块设备
- 逻辑卷 - 如果使用的存储层是来自 LVM 或 ZFS 的设备池。
- 如果应用程序只需要本地存储，那么将使用上述目录、设备（或分区）或逻辑卷之一创建持久卷。 OpenEBS 控制平面将用于提供上述副本之一。

OpenEBS 可以使用其复制引擎之一——Jiva、cStor 和 Mayastor，在本地存储之上添加高可用性层。 在这种情况下，OpenEBS 使用轻量级存储定义的存储控制器软件，该软件可以通过网络端点接收读/写操作， 然后传递到底层存储层。OpenEBS 然后使用此副本网络端点来跨节点维护卷的同步副本。

OpenEBS 卷副本通常经历以下状态：

- 正在初始化，在初始配置期间正在注册到其卷
- 健康，当副本可以参与读/写操作时
- 离线，当副本所在的节点或存储发生故障时
- 重建，当节点或存储故障已得到纠正并且副本正在从其他健康副本接收数据时
- 终止，当卷被删除并且副本被删除并且空间被回收时

存储层

存储层构成了持久化数据的基本构建块。存储层由连接到节点的块设备组成（通过 PCIe、SAS、NVMe 在本地或通过远程 SAN/云）。 存储层也可以是挂载文件系统之上的子目录。

存储层不在 OpenEBS 数据引擎的范围内，可用于使用标准操作系统或 Linux 软件结构的 Kubernetes 存储结构。

数据引擎将存储用作设备或设备池或文件系统目录。

控制平面

OpenEBS 上下文中的控制平面是指部署在集群中的一组工具或组件，它们负责：

- 管理 kubernetes 工作节点上可用的存储
- 配置和管理数据引擎
- 与 CSI 接口以管理卷的生命周期
- 与 CSI 和其他工具进行接口，执行快照、克隆、调整大小、备份、恢复等操作。
- 集成到其他工具中，如 Prometheus/Grafana 以进行遥测和监控
- 集成到其他工具中进行调试、故障排除或日志管理

OpenEBS 控制平面由一组微服务组成,这些微服务本身由 Kubernetes 管理,使 OpenEBS 真正成为 Kubernetes 原生的。 由 OpenEBS 控制平面管理的配置被保存为 Kubernetes 自定义资源。控制平面的功能可以分解为以下各个阶段：

openebs-4

openebs-4

YAML 或 Helm Chart

管理员可以使用高度可配置的 Helm Chart 或 kubectl/YAML 安装 OpenEBS 组件。OpenEBS 安装也通过管理 Kubernetes 产品支持，例如 OpenShift、EKS、DO、Rancher 作为市场应用程序或作为附加组件或插件紧密集成到 Kubernetes 发行版中，例如 MicroK8s、Kinvolk、Kubesphere。

作为 OpenEBS 安装的一部分，所选数据引擎的控制平面组件将使用标准 Kubernetes 原语（如 Deployments、DaemonSets、Statefulsets 等）安装为集群和/或节点组件。 OpenEBS 安装还负责将 OpenEBS 自定义资源定义加载到 Kubernetes 中。

OpenEBS 控制平面组件都是无状态的，依赖于 Kubernetes etcd 服务器（自定义资源）来管理其内部配置状态并报告各种组件的状态。

声明式 API

OpenEBS 支持声明式 API 来管理其所有操作，并且 API 公开为 Kubernetes 自定义资源。 Kubernetes CRD 验证器和 admission webhooks 用于验证用户提供的输入并验证操作是否被允许。

声明式 API 是 Kubernetes 管理员和用户习惯的自然扩展，他们可以在其中通过 YAML 定义意图，然后 Kubernetes 和相关的 OpenEBS 操作员将状态与用户的意图相协调。

声明式 API 可用于配置数据引擎和设置卷配置文件/策略。 甚至数据引擎的升级也是使用此 API 执行的。这些 API 可用于：

- 管理每个数据引擎的配置
- 管理存储需要管理的方式或存储池
- 管理卷及其服务 - 创建、快照、克隆、备份、恢复、删除
- 管理池和卷的升级

数据引擎 Operator

从发现底层存储到创建池和卷的所有数据引擎操作都打包为 Kubernetes Operators。 每个数据引擎要么在安装期间提供的配置之上运行，要么通过相应的 Kubernetes 自定义资源进行

控制。

数据引擎 Operator 可以在集群范围内或在特定节点上操作。 集群范围操作员通常参与涉及与 Kubernetes 组件交互的操作 - 协调各种节点上池和卷的调度或迁移。 节点级运算符对本地操作进行操作，例如在节点上可用的存储或池上创建卷、副本、快照等。

数据引擎 Operator 通常也被称为数据引擎的控制平面，因为它们有助于管理相应数据引擎提供的卷和数据服务。根据提供或需要的功能，某些数据引擎（如 cstor、jiva 和 mayastor）可以有多个 Operator，其中本地卷操作可以直接嵌入到相应的 CSI 控制器/配置器中。

CSI 驱动程序（动态卷配置器）

CSI 驱动程序充当管理 Kubernetes 中卷生命周期的促进者。 CSI 驱动程序操作由 StorageClass 中指定的参数控制或自定义。CSI 驱动程序由三层组成：

- Kubernetes 或 Orchestrator 功能 - Kubernetes 原生并将应用程序绑定到卷
- Kubernetes CSI 层 - 将 Kubernetes 本机调用转换为 CSI 调用 - 以标准方式将用户提供的信息传递给 CSI 驱动程序
- 存储驱动程序——它们是 CSI 投诉的，与 Kubernetes CSI 层密切合作以接收请求并处理它们。
- 存储驱动程序负责：

公开数据引擎的功能：

- 直接与数据引擎或数据引擎操作员交互以执行卷创建和删除操作
- 与数据引擎接口以将卷附加/分离到使用卷的容器正在运行的节点
- 与标准 linux 实用程序接口以格式化、安装/卸载卷到容器

Plug-in 插件

OpenEBS 专注于存储操作，并为其他流行工具提供插件，以执行核心存储功能之外的操作，但对于在生产中运行 OpenEBS 非常重要。 此类操作的示例包括：

- 应用程序一致的备份和恢复（通过集成到 Velero 中提供）
- 监控和警报（通过集成到 Prometheus、Grafana、警报管理器中提供）
- 执行安全策略（通过与 PodSecurityPolicies 或 Kyerno 的集成提供）
- 日志记录（通过与 ELK、Loki、Logstash 等管理员设置的任何标准日志记录堆栈集成提供）
- 可视化（通过标准 Kubernetes 仪表板或自定义 Grafana 仪表板提供）

CLI 命令行

OpenEBS 上的所有管理功能都可以通过 kubectl 执行，因为 OpenEBS 使用自定义资源来管理其所有配置并报告组件的状态。

此外，OpenEBS 还发布了 alpha 版 kubectl 插件，以帮助使用单个命令提供有关池和卷的信息，该命令聚合了通过多个 kubectl 命令获得的信息。

通过 Helm 部署并验证 OpenEBS

本文将提供用 Helm 部署并验证 OpenEBS 云原生存储系统的操作步骤及说明。 DCE 5.0 支持众多第三方存储方案，我们针对 OpenEBS 进行了相关测试，并最终将其作为 Addon 集成了应用商店中。 以下是对 OpenEBS 的调研和测评。

有关应用商店 Addon 的图形化界面安装、部署、卸载等操作说明，将于稍后提供。

测试环境

本次测试使用三个虚拟机节点部署一个 Kubernetes 集群：1 个 Master + 2 个 Worker 节点，

kubelet 版本为 1.23.6。

```
[root@k8s-10-6-162-31 ~]# kubectl get no
NAME                 STATUS   ROLES                  AGE    VERSION
k8s-10-6-162-31      Ready    control-plane,master   114d   v1.23.6
k8s-10-6-162-32      Ready    <none>                 114d   v1.23.6
k8s-10-6-162-33      Ready    <none>                 114d   v1.23.6
```

添加并更新 OpenEBS repo

```
[root@k8s-10-6-162-31 ~]# helm repo add openebs https: //openebs.github.io/charts
"openebs" has been added to your repositories

[root@k8s-10-6-162-31 ~]# helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "longhorn" chart repository
...Successfully got an update from the "openebs" chart repository
Update Complete. ⎈Happy Helming!⎈

[root@k8s-10-6-162-31 ~]# helm repo list
NAME URL
longhorn https: //charts.longhorn.io
openebs https: //openebs.github.io/charts
```

Helm 安装 OpenEBS

```
[root@k8s-10-6-162-31 ~]# helm install openebs --namespace openebs openebs/openebs --create-namespace
NAME: openebs
LAST DEPLOYED: Tue Jan 31 14:44:11 2023
NAMESPACE: openebs
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Successfully installed OpenEBS.

Check the status by running: kubectl get pods -n openebs
```

The default values will install NDM and enable OpenEBS hostpath and device storage engines along with their default StorageClasses. Use `kubectl get sc` to see the list of installed OpenEBS StorageClasses.

Note: If you are upgrading from the older helm chart that was using cStor and Jiva (non-csi) volumes, you will have to run the following command to include the older provisioners:

helm upgrade openebs openebs/openebs \
- -namespace openebs \
- -set legacy.enabled=true \
- -reuse-values

For other engines, you will need to perform a few more additional steps to enable the engine, configure the engines (e.g. creating pools) and create StorageClasses.

For example, cStor can be enabled using commands like:

helm upgrade openebs openebs/openebs \
- -namespace openebs \
- -set cstor.enabled=true \
- -reuse-values

For more information,
- view the online documentation at https: //openebs.io/docs or
- connect with an active community on Kubernetes slack #openebs channel.

查看已安装 OpenEBS 集群资源

[root@k8s-10-6-162-31 ~]# kubectl get po -nopenebs
NAME READY STAT
US RESTARTS AGE
openebs-localpv-provisioner-5646cc6748-sdh2g 1/1 Running 6
(4m44s ago) 47m
openebs-ndm-6dffs 1/1 Running
0 47m
openebs-ndm-fs2jk 1/1 Running
0 47m
openebs-ndm-m5x6d 1/1 Running
0 47m
openebs-ndm-operator-65fdff8c8d-zvl8v 1/1 Running
0 47m

```
[root@k8s-10-6-162-31 ~]# mkdir -p /data/openebs/local
[root@k8s-10-6-162-31 ~]# cd /data/openebs/local
[root@k8s-10-6-162-31 local]# pwd
/data/openebs/local

[root@k8s-10-6-162-31 ~]# kubectl get sc -A
NAME                 PROVISIONER           RECLAIMPOLICY VOLUME
BINDINGMODE          ALLOWVOLUMEEXPANSION  AGE
longhorn (default)   driver.longhorn.io    Delete        Immediate
                     true                  48d
openebs-device       openebs.io/local      Delete        WaitForFirstConsumer fal
se                   19h
openebs-hostpath     openebs.io/local      Delete        WaitForFirstConsumer fals
e                    19h
```

创建 local-hostpath-pvc 资源

```
[root@k8s-10-6-162-31 ~]# cat local-hostpath-pvc.yaml

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-hostpath-pvc
spec:
  storageClassName: openebs-hostpath
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5G

[root@k8s-10-6-162-31 ~]# cat local-hostpath-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-local-hostpath-pod
spec:
  volumes:
    - name: local-storage
      persistentVolumeClaim:
        claimName: local-hostpath-pvc
  containers:
    - name: hello-container
```

```
    image: busybox
    command:
      - sh
      - -c
      - 'while true; do echo "`date` [`hostname`] Hello from OpenEBS Local PV." >> /data
/greeting.txt; sleep $(($RANDOM % 5 + 300)); done'
    volumeMounts:
      - mountPath: /data
        name: local-storage

[root@k8s-10-6-162-31 ~]# kubectl apply -f local-hostpath-pvc.yaml
persistentvolumeclaim/local-hostpath-pvc created

[root@k8s-10-6-162-31 ~]# kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
local-hostpath-pvc Pending openebs-hostpath 8s

[root@k8s-10-6-162-31 ~]# kubectl apply -f local-hostpath-pod.yaml
pod/hello-local-hostpath-pod created

[root@k8s-10-6-162-31 ~]# kubectl get po
NAME READY STATUS RESTARTS AGE
hello-local-hostpath-pod 0/1 Pending 0 11s
longhorn-iscsi-installation-2thd5 1/1 Running 1 48d
longhorn-iscsi-installation-ctqtg 1/1 Running 1 (47d ago) 48d
longhorn-iscsi-installation-mrm4h 1/1 Running 1 (47d ago) 48d

[root@k8s-10-6-162-31 ~]# kubectl get po
NAME READY STATUS RESTARTS AGE
hello-local-hostpath-pod 1/1 Running 0 4m56s
longhorn-iscsi-installation-2thd5 1/1 Running 1 48d
longhorn-iscsi-installation-ctqtg 1/1 Running 1 (47d ago) 48d
longhorn-iscsi-installation-mrm4h 1/1 Running 1 (47d ago) 48d
```

创建 workload 并验证

```
[root@k8s-10-6-162-31 ~]# kubectl exec hello-local-hostpath-pod -- cat /data/greeting.txt
Wed Feb 1 03: 50:57 UTC 2023 [hello-local-hostpath-pod] Hello from OpenEBS Local PV.

[root@k8s-10-6-162-31 ~]# kubectl exec -it hello-local-hostpath-pod -sh
error: you must specify at least one command for the container

[root@k8s-10-6-162-31 ~]# kubectl exec -it hello-local-hostpath-pod -- sh
/ # ls
```

```
bin data dev etc home proc root sys tmp usr var

/ # cd data
/data # ls
greeting.txt

/data # cat greeting.txt
Wed Feb 1 03: 50:57 UTC 2023 [hello-local-hostpath-pod] Hello from OpenEBS Local PV.
Wed Feb 1 03: 55:59 UTC 2023 [hello-local-hostpath-pod] Hello from OpenEBS Local PV.
Wed Feb 1 04: 01:01 UTC 2023 [hello-local-hostpath-pod] Hello from OpenEBS Local PV.
Wed Feb 1 04: 06:04 UTC 2023 [hello-local-hostpath-pod] Hello from OpenEBS Local PV.

[root@k8s-10-6-162-31 ~]# kubectl get -o yaml pv |grep 'path: '
path: /var/openebs/local/pvc-44db3536-2dc3-4b4d-bcd6-6d388a534fcd

[root@k8s-10-6-162-31 ~]# kubectl get pv
NAME CAPACITY ACCESS MODES RECL
AIM POLICY STATUS CLAIM STORAGECLASS R
EASON AGE
pvc-44db3536-2dc3-4b4d-bcd6-6d388a534fcd 5G RWO Delete
Bound default/local-hostpath-pvc openebs-hostpath 71m
[root@k8s-10-6-162-31 ~]# kubectl get -o yaml pv pvc-44db3536-2dc3-4b4d-bcd6-6d388a534fcd
|grep 'path: '
path: /var/openebs/local/pvc-44db3536-2dc3-4b4d-bcd6-6d388a534fcd
```

安装 dbench 性能测试工具

```
[root@k8s-10-6-162-31 ~]# cat fio-deploy.yaml
** NOTE: For details of params to construct an fio job, refer to this link:
** https: //fio.readthedocs.io/en/latest/fio_doc.html

- --
apiVersion: batch/v1
kind: Job
metadata:
  generateName: dbench-
spec:
  template:
    spec:
      containers:
        - name: dbench
          image: openebs/perf-test:latest
          imagePullPolicy: IfNotPresent
          env:
```

```
# storage mount point on which test files are created
- name: DBENCH_MOUNTPOINT
  value: /data
```

测试 I/O 性能

kubectl delete pod hello-local-hostpath-pod

[root@k8s-10-6-162-31 ~]# kubectl create -f fio-deploy.yaml
job.batch/dbench-rmdqr created

[root@k8s-10-6-162-31 ~]# kubectl get pod
NAME READY STATUS RESTARTS AGE
dbench-rmdqr-qbl74 1/1 Running 0 4m59s
longhorn-iscsi-installation-2thd5 1/1 Running 1 54d
longhorn-iscsi-installation-ctqtg 1/1 Running 1 (52d ago) 54d
longhorn-iscsi-installation-mrm4h 1/1 Running 1 (52d ago) 54d

[root@k8s-10-6-162-31 ~]# kubectl logs -f dbench-729cw-nqfpt
Error from server (NotFound): pods "dbench-729cw-nqfpt" not found
[root@k8s-10-6-162-31 ~]# kubectl logs -f dbench-rmdqr-qbl74
Working dir: /data

Testing Read IOPS...
read_iops: (g=0): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioe
ngine=libaio, iodepth=64
fio-3.13
Starting 1 process
read_iops: Laying out IO file (1 file / 2048MiB)

read_iops: (groupid=0, jobs=1): err= 0: pid=17: Mon Feb 6 06:27:30 2023
read: IOPS=137, BW=566KiB/s (580kB/s)(8600KiB/15193msec)
bw (KiB/s): min= 8, max= 1752, per=100.00%, avg=570.00, stdev=518.97, samples=29
iops : min= 2, max= 438, avg=142.41, stdev=129.78, samples=29
cpu : usr=0.22%, sys=1.13%, ctx=1984, majf=0, minf=16
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=100.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.1%, >=64=0.0%
issued rwts: total=2087,0,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
READ: bw=566KiB/s (580kB/s), 566KiB/s-566KiB/s (580kB/s-580kB/s), io=8600KiB (8806kB),
run=15193-15193msec

Disk stats (read/write):
dm-0: ios=2475/39, merge=0/0, ticks=1076796/25702, in_queue=1114531, util=100.00%, aggrios=2494/51, aggrmerge=0/3, aggrticks=1094965/36042, aggrin_queue=1130703, aggrutil=99.98%
sda: ios=2494/51, merge=0/3, ticks=1094965/36042, in_queue=1130703, util=99.98%

Testing Write IOPS...
write_iops: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, io engine=libaio, iodepth=64
fio-3.13
Starting 1 process

write_iops: (groupid=0, jobs=1): err= 0: pid=33: Mon Feb 6 06:27:49 2023
write: IOPS=199, BW=815KiB/s (835kB/s)(13.0MiB/16358msec); 0 zone resets
bw (KiB/s): min= 40, max= 3408, per=100.00%, avg=871.87, stdev=932.70, samples=30
iops : min= 10, max= 852, avg=217.87, stdev=233.18, samples=30
cpu : usr=0.17%, sys=1.05%, ctx=958, majf=0, minf=16
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=100.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.1%, >=64=0.0%
issued rwts: total=0,3271,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
WRITE: bw=815KiB/s (835kB/s), 815KiB/s-815KiB/s (835kB/s-835kB/s), io=13.0MiB (13.7MB), run=16358-16358msec

Disk stats (read/write):
dm-0: ios=0/4802, merge=0/0, ticks=0/1128001, in_queue=1129209, util=99.50%, aggrios=0/4826, aggrmerge=0/4, aggrticks=0/1147779, aggrin_queue=1147776, aggrutil=99.41%
sda: ios=0/4826, merge=0/4, ticks=0/1147779, in_queue=1147776, util=99.41%

Testing Read Bandwidth...
read_bw: (g=0): rw=randread, bs=(R) 128KiB-128KiB, (W) 128KiB-128KiB, (T) 128KiB-128KiB, ioengine=libaio, iodepth=64
fio-3.13
Starting 1 process

read_bw: (groupid=0, jobs=1): err= 0: pid=49: Mon Feb 6 06:28:09 2023
read: IOPS=128, BW=16.5MiB/s (17.3MB/s)(280MiB/16965msec)
bw (KiB/s): min= 256, max=50176, per=100.00%, avg=17994.26, stdev=14173.96, samples=31
iops : min= 2, max= 392, avg=140.48, stdev=110.71, samples=31

cpu : usr=0.18%, sys=1.66%, ctx=2092, majf=0, minf=16
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=100.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.1%, >=64=0.0%
issued rwts: total=2180,0,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
READ: bw=16.5MiB/s (17.3MB/s), 16.5MiB/s-16.5MiB/s (17.3MB/s-17.3MB/s), io=280MiB (294 MB), run=16965-16965msec

Disk stats (read/write):
dm-0: ios=2366/3, merge=0/0, ticks=1093765/4077, in_queue=1100333, util=99.51%, aggrios=2366/6, aggrmerge=0/0, aggrticks=1096465/8183, aggrin_queue=1104639, aggrutil=99.42%
sda: ios=2366/6, merge=0/0, ticks=1096465/8183, in_queue=1104639, util=99.42%

Testing Write Bandwidth...
write_bw: (g=0): rw=randwrite, bs=(R) 128KiB-128KiB, (W) 128KiB-128KiB, (T) 128KiB-128KiB, ioengine=libaio, iodepth=64
fio-3.13
Starting 1 process

write_bw: (groupid=0, jobs=1): err= 0: pid=65: Mon Feb 6 06:28:27 2023
write: IOPS=69, BW=9453KiB/s (9680kB/s)(145MiB/15707msec); 0 zone resets
bw (KiB/s): min= 256, max=29952, per=100.00%, avg=10800.19, stdev=9114.78, samples=26
iops : min= 2, max= 234, avg=84.35, stdev=71.20, samples=26
cpu : usr=0.15%, sys=0.67%, ctx=555, majf=0, minf=16
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=100.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=99.9%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.1%, >=64=0.0%
issued rwts: total=0,1097,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
WRITE: bw=9453KiB/s (9680kB/s), 9453KiB/s-9453KiB/s (9680kB/s-9680kB/s), io=145MiB (152MB), run=15707-15707msec

Disk stats (read/write):
dm-0: ios=0/1499, merge=0/0, ticks=0/1047546, in_queue=1137554, util=99.39%, aggrios=0/1502, aggrmerge=0/3, aggrticks=0/1152415, aggrin_queue=1152412, aggrutil=99.25%
sda: ios=0/1502, merge=0/3, ticks=0/1152415, in_queue=1152412, util=99.25%

Testing Read Latency...
read_latency: (g=0): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=4
fio-3.13
Starting 1 process

read_latency: (groupid=0, jobs=1): err= 0: pid=81: Mon Feb 6 06:28:46 2023
read: IOPS=9, BW=39.7KiB/s (40.7kB/s)(652KiB/16407msec)
slat (usec): min=25, max=224, avg=106.02, stdev=36.24
clat (usec): min=165, max=3176.6k, avg=395710.97, stdev=707730.31
lat (usec): min=227, max=3176.7k, avg=395817.82, stdev=707732.74
clat percentiles (usec):
| 1.00th=[229], 5.00th=[2147], 10.00th=[10683],
| 20.00th=[23462], 30.00th=[35390], 40.00th=[57934],
| 50.00th=[91751], 60.00th=[145753], 70.00th=[263193],
| 80.00th=[505414], 90.00th=[1652556], 95.00th=[2021655],
| 99.00th=[3170894], 99.50th=[3170894], 99.90th=[3170894],
| 99.95th=[3170894], 99.99th=[3170894]
bw (KiB/s): min= 7, max= 216, per=100.00%, avg=63.80, stdev=57.40, samples=20
iops : min= 1, max= 54, avg=15.80, stdev=14.41, samples=20
lat (usec) : 250=1.25%, 500=0.62%, 1000=1.25%
lat (msec) : 2=1.88%, 4=2.50%, 10=2.50%, 20=7.50%, 50=19.38%
lat (msec) : 100=16.25%, 250=18.12%, 500=10.00%, 750=6.25%, 1000=1.25%
cpu : usr=0.01%, sys=0.13%, ctx=162, majf=0, minf=17
IO depths : 1=0.0%, 2=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=160,0,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
READ: bw=39.7KiB/s (40.7kB/s), 39.7KiB/s-39.7KiB/s (40.7kB/s-40.7kB/s), io=652KiB (668kB), run=16407-16407msec

Disk stats (read/write):
dm-0: ios=195/15, merge=0/0, ticks=62870/9094, in_queue=113278, util=99.51%, aggrios=195/22, aggrmerge=0/4, aggrticks=72364/49995, aggrin_queue=122359, aggrutil=99.49%
sda: ios=195/22, merge=0/4, ticks=72364/49995, in_queue=122359, util=99.49%

Testing Write Latency...
write_latency: (g=0): rw=randwrite, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=libaio, iodepth=4
fio-3.13

Starting 1 process

write_latency: (groupid=0, jobs=1): err= 0: pid=97: Mon Feb 6 06:29:04 2023
write: IOPS=30, BW=123KiB/s (126kB/s)(1868KiB/15174msec); 0 zone resets
slat (usec): min=11, max=235, avg=58.23, stdev=38.98
clat (usec): min=205, max=2619.4k, avg=136822.73, stdev=357077.08
lat (usec): min=281, max=2619.5k, avg=136881.82, stdev=357079.12
clat percentiles (usec):
| 1.00th=[277], 5.00th=[383], 10.00th=[537],
| 20.00th=[10421], 30.00th=[17171], 40.00th=[23200],
| 50.00th=[32375], 60.00th=[44303], 70.00th=[68682],
| 80.00th=[92799], 90.00th=[274727], 95.00th=[784335],
| 99.00th=[1937769], 99.50th=[2365588], 99.90th=[2634023],
| 99.95th=[2634023], 99.99th=[2634023]
bw (KiB/s): min= 7, max= 688, per=100.00%, avg=184.90, stdev=186.47, samples=20
iops : min= 1, max= 172, avg=46.00, stdev=46.69, samples=20
lat (usec) : 250=0.43%, 500=9.05%, 750=2.59%
lat (msec) : 2=0.86%, 4=1.29%, 10=5.82%, 20=15.73%, 50=26.72%
lat (msec) : 100=19.40%, 250=7.54%, 500=5.39%, 750=0.43%, 1000=1.29%
cpu : usr=0.06%, sys=0.20%, ctx=222, majf=0, minf=17
IO depths : 1=0.0%, 2=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,464,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
WRITE: bw=123KiB/s (126kB/s), 123KiB/s-123KiB/s (126kB/s-126kB/s), io=1868KiB (1913kB), run=15174-15174msec

Disk stats (read/write):
dm-0: ios=0/475, merge=0/0, ticks=0/68658, in_queue=69443, util=99.48%, aggrios=0/475, aggrmerge=0/0, aggrticks=0/69983, aggrin_queue=69983, aggrutil=99.40%
sda: ios=0/475, merge=0/0, ticks=0/69983, in_queue=69983, util=99.40%

Testing Read Sequential Speed...
read_seq: (g=0): rw=read, bs=(R) 1024KiB-1024KiB, (W) 1024KiB-1024KiB, (T) 1024KiB-1024KiB, ioengine=libaio, iodepth=16
fio-3.13
Starting 1 thread

read_seq: (groupid=0, jobs=1): err= 0: pid=113: Mon Feb 6 06:29:28 2023
read: IOPS=17, BW=17.8MiB/s (18.7MB/s)(358MiB/20078msec)

bw (KiB/s): min= 2043, max=81920, per=100.00%, avg=24218.28, stdev=22785.58, samples=29
iops : min= 1, max= 80, avg=23.52, stdev=22.30, samples=29
cpu : usr=0.03%, sys=0.60%, ctx=351, majf=0, minf=0
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=99.7%, 8=0.0%, 16=0.3%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=343,0,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=16

Run status group 0 (all jobs):
READ: bw=17.8MiB/s (18.7MB/s), 17.8MiB/s-17.8MiB/s (18.7MB/s-18.7MB/s), io=358MiB (375 MB), run=20078-20078msec

Disk stats (read/write):
dm-0: ios=764/10, merge=0/0, ticks=589803/16574, in_queue=609402, util=99.61%, aggrios=764/13, aggrmerge=0/1, aggrticks=592905/15938, aggrin_queue=608802, aggrutil=99.52%
sda: ios=764/13, merge=0/1, ticks=592905/15938, in_queue=608802, util=99.52%

Testing Write Sequential Speed...
write_seq: (g=0): rw=write, bs=(R) 1024KiB-1024KiB, (W) 1024KiB-1024KiB, (T) 1024KiB-1024KiB, ioengine=libaio, iodepth=16
...
fio-3.13
Starting 4 threads
write_seq: Laying out IO file (1 file / 2798MiB)

write_seq: (groupid=0, jobs=1): err= 0: pid=129: Mon Feb 6 06:29:59 2023
write: IOPS=5, BW=6679KiB/s (6839kB/s)(187MiB/28670msec); 0 zone resets
bw (KiB/s): min= 2043, max=67449, per=59.42%, avg=16001.91, stdev=15698.61, samples=22
iops : min= 1, max= 65, avg=15.50, stdev=15.23, samples=22
cpu : usr=0.04%, sys=0.17%, ctx=149, majf=0, minf=0
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=99.4%, 8=0.0%, 16=0.6%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,172,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=16
write_seq: (groupid=0, jobs=1): err= 0: pid=130: Mon Feb 6 06:29:59 2023
write: IOPS=6, BW=6827KiB/s (6991kB/s)(189MiB/28350msec); 0 zone resets
bw (KiB/s): min= 2048, max=57344, per=60.09%, avg=16183.05, stdev=15552.27, samples=21
iops : min= 2, max= 56, avg=15.67, stdev=15.15, samples=21
cpu : usr=0.04%, sys=0.18%, ctx=160, majf=0, minf=0
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, >=64=0.0%

submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=99.4%, 8=0.0%, 16=0.6%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,174,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=16
write_seq: (groupid=0, jobs=1): err= 0: pid=131: Mon Feb 6 06:29:59 2023
write: IOPS=6, BW=6932KiB/s (7098kB/s)(189MiB/27921msec); 0 zone resets
bw (KiB/s): min= 2048, max=55185, per=75.56%, avg=20347.18, stdev=14701.69, samples=17
iops : min= 2, max= 53, avg=19.71, stdev=14.20, samples=17
cpu : usr=0.03%, sys=0.19%, ctx=163, majf=0, minf=0
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=99.4%, 8=0.0%, 16=0.6%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,174,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=16
write_seq: (groupid=0, jobs=1): err= 0: pid=132: Mon Feb 6 06:29:59 2023
write: IOPS=6, BW=6767KiB/s (6929kB/s)(189MiB/28600msec); 0 zone resets
bw (KiB/s): min= 2048, max=67584, per=60.14%, avg=16196.64, stdev=15468.10, samples=22
iops : min= 2, max= 66, avg=15.73, stdev=15.17, samples=22
cpu : usr=0.04%, sys=0.19%, ctx=162, majf=0, minf=0
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=100.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=99.4%, 8=0.0%, 16=0.6%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=0,174,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=16

Run status group 0 (all jobs):
WRITE: bw=26.3MiB/s (27.6MB/s), 6679KiB/s-6932KiB/s (6839kB/s-7098kB/s), io=754MiB (79 1MB), run=27921-28670msec

Disk stats (read/write):
dm-0: ios=0/1536, merge=0/0, ticks=0/3438611, in_queue=3460488, util=99.70%, aggrios=0/1553, aggrmerge=0/0, aggrticks=0/3505870, aggrin_queue=3506177, aggrutil=99.72%
sda: ios=0/1553, merge=0/0, ticks=0/3505870, in_queue=3506177, util=99.72%

Testing Read/Write Mixed...
rw_mix: (g=0): rw=randrw, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengin e=libaio, iodepth=64
fio-3.13
Starting 1 process

rw_mix: (groupid=0, jobs=1): err= 0: pid=148: Mon Feb 6 06:30:22 2023
read: IOPS=294, BW=1185KiB/s (1214kB/s)(23.5MiB/20301msec)
bw (KiB/s): min= 24, max=10088, per=100.00%, avg=2173.91, stdev=2974.57, samples=22

iops : min= 6, max= 2522, avg=543.41, stdev=743.66, samples=22
write: IOPS=93, BW=377KiB/s (386kB/s)(7656KiB/20301msec); 0 zone resets
bw (KiB/s): min= 8, max= 3000, per=100.00%, avg=754.55, stdev=998.04, samples=20
iops : min= 2, max= 750, avg=188.60, stdev=249.52, samples=20
cpu : usr=0.25%, sys=1.12%, ctx=2198, majf=0, minf=16
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=100.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.1%, >=64=0.0%
issued rwts: total=5971,1896,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
READ: bw=1185KiB/s (1214kB/s), 1185KiB/s-1185KiB/s (1214kB/s-1214kB/s), io=23.5MiB (24.6MB), run=20301-20301msec
WRITE: bw=377KiB/s (386kB/s), 377KiB/s-377KiB/s (386kB/s-386kB/s), io=7656KiB (7840kB), run=20301-20301msec

Disk stats (read/write):
dm-0: ios=2926/2535, merge=0/0, ticks=676178/621959, in_queue=1332944, util=99.60%, aggrios=2926/2630, aggrmerge=0/21, aggrticks=676172/1200603, aggrin_queue=1875954, aggrutil=100.00%
sda: ios=2926/2630, merge=0/21, ticks=676172/1200603, in_queue=1875954, util=100.00%

All tests complete.

==========
= Dbench Summary =
==========
Random Read/Write IOPS: 137/199. BW: 16.5MiB/s / 9453KiB/s
Average Latency (usec) Read/Write: 395817.82/136881.82
Sequential Read/Write: 17.8MiB/s / 26.3MiB/s
Mixed Random Read/Write IOPS: 294/93

TiDB

TiDB 是一款同时支持在线事务处理（OLTP）与在线分析处理（OATP）的融合型分布式数据库产品，具备水平扩缩容、金融级高可用、实时 HTAP（即同时支持 OLTP 和 OATP）、云原生的分布式数据库，兼容 MySQL 5.7 协议和 MySQL 生态等重要特性。TiDB 的目标是为用户提供一站式的 OLTP、OLAP、HTAP 解决方案，适合高可用、强一致要求较高、数据规

模较大等各种应用场景。

TiDB 整体架构

TiDB 分布式数据库将整体架构拆分成了多个模块，各模块之间互相通信，组成完整的 TiDB 系统。对应的架构图如下：

TiDB 架构图

TiDB 架构图

- **TiDB Server**

  SQL 层对外暴露 MySQL 协议的连接端点，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划。TiDB 层本身是无状态的，实践中可以启动多个 TiDB 实例，通过负载均衡组件（如 LVS、HAProxy 或 F5）对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个 TiDB 实例上以达到负载均衡的效果。TiDB Server 本身并不存储数据，只是解析 SQL，将实际的数据读取请求转发给底层的存储节点 TiKV（或 TiFlash）。

- **PD (Placement Driver) Server**

  整个 TiDB 集群的元信息管理模块，负责存储每个 TiKV 节点实时的数据分布情况和集群的整体拓扑结构，提供 TiDB Dashboard 管控界面，并为分布式事务分配事务 ID。PD 不仅存储元信息，同时还会根据 TiKV 节点实时上报的数据分布状态，下发数据调度命令给具体的 TiKV 节点，可以说是整个集群的“大脑”。此外，PD 本身也是由至少 3 个节点构成，拥有高可用的能力。建议部署奇数个 PD 节点。

- **存储节点**

  - TiKV Server：负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。TiKV 的 API 在 KV 键值对层面提供对分布式事务的原生支持，默认提供了 SI (Snapshot Isolation) 的隔离级别，这也是 TiDB 在 SQL 层面支持分布式事务的核心。TiDB

的 SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为对 TiKV API 的实际调用。所以，数据都存储在 TiKV 中。另外，TiKV 中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。

- TiFlash：TiFlash 是一类特殊的存储节点。和普通 TiKV 节点不一样的是，在 TiFlash 内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。

TiDB 数据库的存储

TiDB 数据库的存储

TiDB 数据库的存储

- **键值对（Key-Value Pair)**

  TiKV 的选择是 Key-Value 模型，并且提供有序遍历方法。TiKV 数据存储的两个关键点：

  - 这是一个巨大的 Map（可以类比一下 C++ 的 std::map），也就是存储的是 Key-Value Pairs。
  - 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是可以 Seek 到某一个 Key 的位置，然后不断地调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value。

- **本地存储（Rocks DB）**

  任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV 也不例外。但是 TiKV 没有选择直接向磁盘上写数据，而是把数据保存在 RocksDB 中，具体的数据落地由 RocksDB 负责。这样做的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而 RocksDB 是由 Facebook 开源的一个非常优秀的单机 KV 存储引擎，可以满足 TiKV 对单机引擎的各种要求。这里可以简单地认为 RocksDB 是一个单机的持久化 Key-Value Map。

- **Raft 协议**

TiKV 选择了 Raft 算法来保证单机失效的情况下数据不丢失不出错。简单来说，就是把数据复制到多台机器上，这样某一台机器无法提供服务时，其他机器上的副本还能提供服务。这个数据复制方案可靠并且高效，能处理副本失效的情况。

- **Region**

  TiKV 选择了按照 Key 划分 Range。某一段连续的 Key 都保存在一个存储节点上。将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，称为一个 Region。尽量让每个 Region 中保存的数据不超过一定的大小，目前在 TiKV 中默认是不超过 96MB。每一个 Region 都可以用 [StartKey，EndKey] 这样的左闭右开区间来描述。

- **MVCC**

  TiKV 实现了多版本并发控制（MVCC)。

- **分布式 ACID 事务**

  TiKV 的事务采用的是 Google 在 BigTable 中使用的事务模型：Percolator。

搭建测试环境

Kubernetes 集群

本次测试使用三台虚拟机节点部署 Kubernetes 集群，包括 1 个 Master 节点和 2 个 Worker 节点。kubelet 版本为 1.22.0。

Kubernetes cluster

Kubernetes cluster

HwameiStor 本地存储

1. 在 Kubernetes 集群上部署 HwameiStor 本地存储

   HwameiStor 本地存储

   HwameiStor 本地存储

2. 在两台 worker 节点上分别为 HwameiStor 配置一块 100G 的本地磁盘 sdb

sdb1

sdb1

sdb2

sdb2

3. 创建 storagClass

创建 StorageClass

创建 StorageClass

在 Kubernetes 上部署 TiDB

可以使用 TiDB Operator 在 Kubernetes 上部署 TiDB。TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或私有部署的 Kubernetes 集群上。

TiDB 与 TiDB Operator 版本的对应关系如下：

TiDB 版本	适用的 TiDB Operator 版本
dev	dev
TiDB >= 5.4	1.3
5.1 <= TiDB < 5.4	1.3（推荐），1.2
3.0 <= TiDB < 5.1	1.3（推荐），1.2，1.1
2.1 <= TiDB < 3.0	1.0（停止维护）

部署 TiDB Operator

1. 安装 TiDB CRDs

```
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/crd.yaml
```

2. 安装 TiDB Operator

```
helm repo add pingcap https://charts.pingcap.org/
    kubectl create namespace tidb-admin
    helm install --namespace tidb-admin tidb-operator pingcap/tidb-operator --version v1.3.2
    \
    --set operatorImage=registry.cn-beijing.aliyuncs.com/tidb/tidb-operator:v1.3.2 \
    --set tidbBackupManagerImage=registry.cn-beijing.aliyuncs.com/tidb/tidb-backup-manager:v
    1.3.2 \
    --set scheduler.kubeSchedulerImageName=registry.cn-hangzhou.aliyuncs.com/google_contai
    ners/kube-scheduler
```

3. 检查 TiDB Operator 组件

检查 TiDB Operator 组件

检查 TiDB Operator 组件

部署 TiDB 集群

```
kubectl create namespace tidb-cluster && \
kubectl -n tidb-cluster apply -f https: //raw.githubusercontent.com/pingcap/tidb-operator/master/exa
mples/basic/tidb-cluster.yaml
kubectl -n tidb-cluster apply -f https: //raw.githubusercontent.com /pingcap/tidb-operator/master/ex
amples/basic/tidb-monitor.yaml
```

部署 TiDB 集群

部署 TiDB 集群

连接 TiDB 集群

```
yum -y install mysql-client
```

connecttidb

```
connecttidb
kubectl port-forward -n tidb-cluster svc/basic-tidb 4000 > pf4000.out &
```

连接 TiDB 集群

连接 TiDB 集群

连接 TiDB 集群

连接 TiDB 集群

连接 TiDB 集群

检查并验证 TiDB 集群状态

1. 创建 Hello_world 表

```
create table hello_world (id int unsigned not null auto_increment primary key, v varcha
    r(32));
```

创建 Hello_world 表

创建 Hello_world 表

2. 查询 TiDB 版本号

```
select tidb_version()\G;
```

连接 TiDB 集群

连接 TiDB 集群

3. 查询 Tikv 存储状态

```
select * from information_schema.tikv_store_status\G;
```

查询 Tikv 存储状态

查询 Tikv 存储状态

HwameiStor 存储配置

从 storageClass local-storage-hdd-lvm 分别为 tidb-tikv 及 tidb-pd 创建一个 PVC:

HwameiStor 存储配置

HwameiStor 存储配置

HwameiStor 存储配置

HwameiStor 存储配置

HwameiStor 存储配置

HwameiStor 存储配置

kubectl get po basic-tikv-0 -oyaml

HwameiStor 存储配置

HwameiStor 存储配置

kubectl get po basic-pd-0 -oyaml

HwameiStor 存储配置

HwameiStor 存储配置

测试内容

数据库 SQL 基本能力测试

完成部署数据库集群后，执行了以下基本能力测试，全部通过。

分布式事务

测试目的：支持在多种隔离级别下，实现分布式数据操作的完整性约束即 ACID 属性

测试步骤：

1. 创建测试数据库 CREATE DATABASE testdb
2. 创建测试用表 CREATE TABLE t_test (id int AUTO_INCREMENT, name varchar(32), PRIMARY KEY (id))
3. 运行测试脚本

测试结果：支持在多种隔离级别下，实现分布式数据操作的完整性约束即 ACID 属性

对象隔离

测试目的：测试不同 schema 实现对象隔离

测试脚本：

```
create database if not exists testdb;
use testdb
create table if not exists t_test
( id                    bigint,
  name                  varchar(200),
  sale_time             datetime default current_timestamp,
  constraint pk_t_test primary key (id)
);
insert into t_test(id,name) values (1,'a'),(2,'b'),(3,'c');
create user 'readonly'@'%' identified by "readonly";
grant select on testdb.* to readonly@'%';
select * from testdb.t_test;
update testdb.t_test set name='aaa';
create user 'otheruser'@'%' identified by "otheruser";
```

测试结果：支持创建不同 schema 实现对象隔离

表操作支持

测试目的：测试是否支持创建、删除和修改表数据、DML、列、分区表

测试步骤：连接数据库后按步骤执行测试脚本

测试脚本：

```
# 创建和删除表
drop table if exists t_test;
create table if not exists t_test
( id                    bigint default '0',
  name                  varchar(200) default '' ,
  sale_time             datetime default current_timestamp,
  constraint pk_t_test primary key (id)
);
# 删除和修改
insert into t_test(id,name) values (1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e');
update t_test set name='aaa' where id=1;
update t_test set name='bbb' where id=2;
```

```
delete from t_dml where id=5;
# 修改、增加、删除列
alter table t_test modify column name varchar(250);
alter table t_test add column col varchar(255);
insert into t_test(id,name,col) values(10,'test','new_col');
alter table t_test add column colwithdefault varchar(255) default 'aaaa';
insert into t_test(id,name) values(20,'testdefault');
insert into t_test(id,name,colwithdefault ) values(10,'test','non-default ');
alter table t_test drop column colwithdefault;
# 分区表类型（仅摘录部分脚本）
CREATE TABLE employees (
    id INT NOT NULL,
fname VARCHAR(30),
lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
job_code INT NOT NULL,
store_id INT NOT NULL
)
```

测试结果：支持创建、删除和修改表数据、DML、列、分区表

索引支持

测试目的：验证多种类型的索引（唯一、聚簇、分区、Bidirectional indexes、Expression-based indexes、哈希索引等等）以及索引重建操作。

测试脚本：

```
alter table t_test add unique index udx_t_test (name);
# 默认就是主键聚簇索引
ADMIN CHECK TABLE t_test;
create index time_idx on t_test(sale_time);
alter table t_test drop index time_idx;
admin show ddl jobs;
admin show ddl job queries 156;
create index time_idx on t_test(sale_time);
```

测试结果：支持创建、删除、组合、单列、唯一索引

表达式

测试目的：验证分布式数据库的表达式支持 if、casewhen、forloop、whileloop、loop exit when 等语句（上限 5 类）

前提条件：数据库集群已经部署完成。

测试脚本：

```
SELECT CASE id WHEN 1 THEN 'first' WHEN 2 THEN 'second' ELSE 'OTHERS' END AS id_new  FROM t_test;
SELECT IF(id>2,'int2+','int2-') from t_test;
```

测试结果：支持 if、case when、for loop、while loop、loop exit when 等语句（上限 5 类）

执行计划解析

测试目的：验证分布式数据库的执行计划解析支持

前提条件：数据库集群已经部署完成。

测试脚本：

```
explain analyze select * from t_test where id NOT IN (1,2,4);
explain analyze select * from t_test a where EXISTS (select * from t_test b where a.id=b.id and b.id<3);
explain analyze SELECT IF(id>2,'int2+','int2-') from t_test;
```

测试结果：支持执行计划的解析

执行计划绑定

测试目的：验证分布式数据库的执行计划绑定功能

测试步骤：

1. 查看 sql 语句的当前执行计划
2. 使用绑定特性

3. 查看该 sql 语句绑定后的执行计划

4. 删除绑定

测试脚本：

```
explain select * from employees3 a join employees4 b on a.id = b.id where a.lname='Johnson';
explain select /*+ hash_join(a,b) */ * from employees3 a join employees4 b on a.id = b.id where a.lname='Johnson';
```

测试结果：没有使用 hint 时可能不是 hash_join，使用 hint 后一定是 hash_join。

常用函数

测试目的：验证分布式数据库的标准的数据库函数(支持的函数类型)

测试结果：支持标准的数据库函数

显式/隐式事务

测试目的：验证分布式数据库的事务支持

测试结果：支持显示与隐式事务

字符集

测试目的：验证分布式数据库的数据类型支持

测试结果：目前只支持 UTF-8 mb4 字符集

锁支持

测试目的：验证分布式数据库的锁实现

测试结果：描述了锁的实现方式，R-R/R-W/W-W 情况下阻塞情况，死锁处理方式

隔离级别

测试目的：验证分布式数据库的事务隔离级别

测试结果：支持 si 隔离级别，支持 rc 隔离级别（4.0 GA 版本）

分布式复杂查询

测试目的：验证分布式数据库的分布式复杂查询能力

测试结果：支持跨节点 join 等分布式复杂查询、操作等，支持窗口函数、层次查询

系统安全测试

这部分测试系统安全，完成数据库集群部署后，以下安全测试全部通过。

账号管理与权限测试

测试目的：验证分布式数据库的账号权限管理

测试脚本：

```
select host,user,authentication_string from mysql.user;
create user tidb IDENTIFIED by 'tidb';
select host,user,authentication_string from mysql.user;
set password for tidb =password('tidbnew');
select host,user,authentication_string,Select_priv from mysql.user;
grant select on *.* to tidb;
flush privileges ;
select host,user,authentication_string,Select_priv from mysql.user;
grant all privileges on *.* to tidb;
flush privileges ;
select * from  mysql.user where user='tidb';
revoke select on *.* from tidb;
flush privileges ;
revoke all privileges on *.* from tidb;
flush privileges ;
```

```
grant select(id) on test.TEST_HOTSPOT to tidb;
drop user tidb;
```

测试结果：

- 支持创建、修改删除账号，并配置和密码，支持安全、审计和数据管理三权分立
- 根据不同账号，对数库各个级别权限控制包括：实例/库/表/列级别

访问控制

测试目的：验证分布式数据库的权限访问控制，数据库数据通过赋予基本增删改查访问权限控制

测试脚本：

```
mysql -u root -h 172.17.49.222 -P 4000
drop user tidb;
drop user tidb1;
create user tidb IDENTIFIED by 'tidb';
grant select on tidb.* to tidb;
grant insert on tidb.* to tidb;
grant update on tidb.* to tidb;
grant delete on tidb.* to tidb;
flush privileges;
show grants for tidb;
exit;
mysql -u tidb -h 172.17.49.222 -ptidb -P 4000 -D tidb -e 'select * from aa;'
mysql -u tidb -h 172.17.49.222 -ptidb -P 4000 -D tidb -e 'insert into aa values(2);'
mysql -u tidb -h 172.17.49.222 -ptidb -P 4000 -D tidb -e 'update aa set id=3;'
mysql -u tidb -h 172.17.49.222 -ptidb -P 4000 -D tidb -e 'delete from aa where id=3;'
```

测试结果：数据库数据通过赋予基本增删改查访问权限控制。

白名单

测试目的：验证分布式数据库的白名单功能

测试脚本：

```
mysql -u root -h 172.17.49.102 -P 4000
drop user tidb;
```

```
create user tidb@'127.0.0.1' IDENTIFIED by 'tidb';
flush privileges;
select * from mysql.user where user='tidb';
mysql -u tidb -h 127.0.0.1 -P 4000 -ptidb
mysql -u tidb -h 172.17.49.102 -P 4000 -ptidb
```

测试结果：支持 IP 白名单功能，支持 IP 段通配操作

操作日志记录

测试目的：验证分布式数据库的操作监控能力

测试脚本：kubectl -ntidb-cluster logs tidb-test-pd-2 --tail 22

测试结果：记录用户通过运维管理控制台或者 API 执行的关键操作或者错误操作

运维管理测试

这部分测试系统运维，完成数据库集群部署后，以下运维管理测试全部通过。

数据导入导出

测试目的：验证分布式数据库的数据导入导出的工具支持

测试脚本：

```
select * from sbtest1 into outfile '/sbtest1.csv';
load data local infile '/sbtest1.csv' into table test100;
```

测试结果：支持按表、schema、database 级别的逻辑导出导入

慢日志查询

测试目的：获取慢查询的 SQL 信息

前提条件：SQL 执行时间需大于配置的慢查询记录阈值,且 SQL 执行完毕

测试步骤：

1. 调整慢查询阈值到 100ms

2. 执行 sql

3. 查看 log/系统表/dashboard 中的慢查询信息

测试脚本：

```
show variables like 'tidb_slow_log_threshold';
set tidb_slow_log_threshold=100;
select query_time, query from information_schema.slow_query where is_internal = false order by query_time desc limit 3;
```

测试结果：可以获取慢查询信息

有关测试详情，请查阅 TiDB on hwameiStor 部署及测试记录。

— sidebar_position: 3 sidebar_label: “Minio” - –

MinIO

MinIO 简介

MinIO 是一款高性能、分布式、兼容 S3 的多云对象存储系统套件。MinIO 原生支持 Kubernetes，能够支持所有公有云、私有云及边缘计算环境。 MinIO 是 GNU AGPL v3 开源的软件定义产品，能够很好地运行在标准硬件如 x86 等设备上。

MinIO 架构

MinIO 架构

MinIO 的架构设计从一开始就是针对性能要求很高的私有云标准，在实现对象存储所需要的全部功能的基础上追求极致的性能。 MinIO 具备易用性、高效性及高性能，能够以更简单的方式提供具有弹性扩缩能力的云原生对象存储服务。

MinIO 在传统对象存储场景（如辅助存储、灾难恢复和归档）方面表现出色，同时在机器学习、大数据、私有云、混合云等方面的存储技术上也独树一帜，包括数据分析、高性能应用

负载、原生云应用等。

MinIO 架构设计

MinIO 为云原生架构设计，可以作为轻量级容器运行并由外部编排服务如 Kubernetes 管理。MinIO 整个服务包约为不到 100 MB 的静态二进制文件，即使在很高负载下也可以高效利用 CPU 和内存资源并可以在共享硬件上共同托管大量租户。 对应的架构图如下：

架构图

架构图

MinIO 用作云原生应用程序的主要存储，与传统对象存储相比，云原生应用程序需要更高的吞吐量和更低的延迟，而这些都是 MinIO 能够达成的性能指标，读/写速度高达 183 GB/秒和 171 GB/秒。

MinIO 极致的高性能离不开底层存储基础平台。本地存储在众多的存储协议中具有最高的读写性能无疑能为 MinIO 提供性能保障。HwameiStor 正是满足云原生时代要求的储存系统。它具有高性能、高可用、自动化、低成本、快速部署等优点，可以替代昂贵的传统 SAN 存储。

MinIO 可以在带有本地驱动器（JBOD/JBOF）的标准服务器上运行。 集群为完全对称的体系架构，即所有服务器的功能均相同，没有名称节点或元数据服务器。

MinIO 将数据和元数据作为对象一起写入从而无需使用元数据数据库。 MinIO 以内联、严格一致的操作执行所有功能，包括擦除代码、位 rotrot 检查、加密等。

每个 MinIO 集群都是分布式 MinIO 服务器的集合，每个节点一个进程。 MinIO 作为单个进程在用户空间中运行，并使用轻量级的协同例程来实现高并发。 将驱动器分组到擦除集（默认情况下，每组 16 个驱动器），然后使用确定性哈希算法将对象放置在这些擦除集上。

MinIO 专为大规模、多数据中心云存储服务而设计。 每个租户都运行自己的 MinIO 集群，该集群与其他租户完全隔离，从而使租户能够免受升级、更新和安全事件的任何干扰。 每个租户通过联合跨地理区域的集群来独立扩展。

node-distribution-setup

node-distribution-setup

以 HwameiStor 为底座搭建 MinIO 的优势

以 HwameiStor 为底座搭建 MinIO 存储方案，构建智、稳、敏全面增强本地存储，具备以下优势。

- 自动化运维管理

  可以自动发现、识别、管理、分配磁盘。 根据亲和性，智能调度应用和数据。自动监测磁盘状态，并及时预警。

- 高可用的数据

  使用跨节点副本同步数据， 实现高可用。发生问题时，会自动将应用调度到高可用数据节点上，保证应用的连续性。

- 丰富的数据卷类型

  聚合 HDD、SSD、NVMe 类型的磁盘，提供非低延时，高吞吐的数据服务。

- 灵活动态的线性扩展

  可以根据集群规模大小进行动态的扩容，灵活满足应用的数据持久化需求。

- 丰富的应用场景，广泛适配企业需求，适配高可用架构中间件

  类似 Kafka、ElasticSearch、Redis 等这类中间件自身具备高可用架构，同时对数据的 IO 访问有很高要求。 HwameiStor 提供的基于 LVM 的单副本本地数据卷，可以很好地满足它们的要求。

- 为应用提供高可用数据卷

  MySQL 等 OLTP 数据库要求底层存储提供高可用的数据存储，当发生问题时可快速恢复数据，同时也要求保证高性能的数据访问。 HwameiStor 提供的双副本的高可用数据卷，可以很好地满足此类需求。

- 自动化运维传统存储软件

  MinIO、Ceph 等存储软件，需要使用 Kubernetes 节点上的磁盘，可以采用 PVC/PV 的方式， 通过 CSI 驱动自动化地使用 HwameiStor 的单副本本地卷，快速响应业务系统提出的部署、扩容、迁移等需求，实现基于 Kubernetes 的自动化运维。

测试环境

按照以下步骤依次部署 Kubernetes 集群、HwameiStor 本地存储和 MinIO。

部署 Kubernetes 集群

本次测试使用了三台虚拟机节点部署了 Kubernetes 集群：1 Master + 2 Worker 节点，kubelet 版本为 1.22.0。

k8s-cluster

k8s-cluster

部署 HwameiStor 本地存储

在 Kubernetes 上部署 HwameiStor 本地存储。

查看 HwameiStor 本地存储

查看 HwameiStor 本地存储

两台 Worker 节点各配置了五块磁盘（SDB、SDC、SDD、SDE、SDF）用于 HwameiStor 本地磁盘管理。

lsblk

lsblk

lsblk

lsblk

查看 local storage node 状态。

get-lsn

get-lsn

创建了 storagClass。

get-sc

get-sc

分布式多租户源码部署安装（minio operator）

本节说明如何部署 minio operator，如何创建租户，如何配置 HwameiStor 本地卷。

部署 minio operator

参照以下步骤部署 minio operator。

1. 复制 minio operator 仓库到本地。

   ```
   git clone <https://github.com/minio/operator.git>
   ```

   helm-repo-list

   helm-repo-list

   ls-operator

   ls-operator

2. 进入 helm operator 目录：/root/operator/helm/operator。

   ls-pwd

   ls-pwd

3. 部署 minio-operator 实例。

   ```
   helm install minio-operator \
       --namespace minio-operator \
       --create-namespace \
       --generate-name .
       --set persistence.storageClass=local-storage-hdd-lvm .
   ```

4. 检查 minio-operator 资源运行情况。

   get-all

get-all

创建租户

参照以下步骤创建一个租户。

1. 进入 /root/operator/examples/kustomization/base 目录。如下修改 tenant.yaml。

   git-diff-yaml

   git-diff-yaml

2. 进入 /root/operator/helm/tenant/ 目录。如下修改 values.yaml 文件。

   git-diff-values.yaml

   git-diff-values.yaml

3. 进入 /root/operator/examples/kustomization/tenant-lite 目录。如下修改 kustomization.yaml 文件。

   git-diff-kustomization-yaml

   git-diff-kustomization-yaml

4. 如下修改 tenant.yaml 文件。

   git-diff-tenant-yaml02

   git-diff-tenant-yaml02

5. 如下修改 tenantNamePatch.yaml 文件。

   git-diff-tenant-name-patch-yaml

   git-diff-tenant-name-patch-yaml

6. 创建租户：

   kubectl apply –k .

7. 检查租户 minio-t1 资源状态：

   kubectl-get-all-nminio-tenant

   kubectl-get-all-nminio-tenant

8. 如要创建一个新的租户可以在 /root/operator/examples/kustomization 目录下建一个新的 tenant 目录（本案例为 tenant-lite-2）并对相应文件做对应修改。

   pwd-ls-ls

   pwd-ls-ls

9. 执行 kubectl apply –k . 创建新的租户 minio-t2。

   kubectl-get-all-nminio

kubectl-get-all-nminio

配置 HwameiStor 本地卷

依次运行以下命令来配置本地卷。

kubectl get statefulset.apps/minio-t1-pool-0 -nminio-tenant -oyaml
local-storage-hdd-lvm
local-storage-hdd-lvm
kubectl get pvc –A
kubectl-get-pvc
kubectl-get-pvc
kubectl get pvc export-minio6-0 -nminio-6 -oyaml
kubectl-get-pvc-export-oyaml
kubectl-get-pvc-export-oyaml
kubectl get pv
kubectl-get-pv
kubectl-get-pv
kubectl get pvc data0-minio-t1-pool-0-0 -nminio-tenant -oyaml
kubectl-get-pvc-oyaml
kubectl-get-pvc-oyaml
kubectl get lv
kubectl-get-lv
kubectl-get-lv
kubect get lvr
kubectl-get-lvr
kubectl-get-lvr

HwameiStor 与 Minlo 测试验证

完成上述配置之后，执行了基本功能测试和多租户隔离测试。

基本功能测试

基本功能测试的步骤如下。

1. 从浏览器登录 minio console： 10.6.163.52: 30401/login。

   minio-opeartor-console-login
   minio-opeartor-console-login

2. 通过 kubectl minio proxy -n minio-operator 获取 JWT。

minio-opeartor-console-login

minio-opeartor-console-login

3. 浏览及管理创建的租户信息。

tenant01

tenant01

tenant02

tenant02

tenant03

tenant03

tenant04

tenant04

tenant05

tenant05

tenant06

tenant06

4. 登录 minio-t1 租户（用户名 minio，密码 minio123）。

login-minio

login-minio

login-minio

login-minio

5. 浏览 bucket bk-1。

view-bucket-1

view-bucket-1

view-bucket-1

view-bucket-1

view-bucket-1

view-bucket-1

6. 创建新的 bucket bk-1-1。

create-bucket-1-1

create-bucket-1-1

create-bucket-1-1

create-bucket-1-1

create-bucket-1-1

create-bucket-1-1

7. 创建 path path-1-2。

create-path-1-2

create-path-1-2

create-path-1-2

create-path-1-2

8. 上传文件成功：

upload-file

upload-file

upload-file

upload-file

upload-file

upload-file

upload-file

upload-file

9. 上传文件夹成功：

upload-folder

upload-folder

upload-folder

upload-folder

upload-folder

upload-folder

upload-folder

upload-folder

10. 创建只读用户：

create-user

create-user

create-user

create-user

多租户隔离测试

执行以下步骤进行多租户隔离测试。

1. 登录 minio-t2 租户。

login-t2

login-t2

login-t2

login-t2

2. 此时只能看到 minio-t2 内容，minio-t1 的内容被屏蔽。

only-t2

only-t2

3. 创建 bucket。

create-bucket

create-bucket

create-bucket

create-bucket

4. 创建 path。

create-path

create-path

create-path

5. 上传文件。

upload-file

upload-file

upload-file

6. 创建用户。

create-user

create-user

create-user

create-user

create-user

create-user

create-user

7. 配置用户 policy。

user-policy

user-policy

user-policy

8. 删除 bucket。

delete-bucket

delete-bucket

delete-bucket

delete-bucket

delete-bucket

delete-bucket

delete-bucket

delete-bucket

结论

本次测试是在 Kubernetes 1.22 平台上部署了 MinIO 分布式对象存储并对接 HwameiStor 本地存储。在此环境中完成了基本能力测试、系统安全测试及运维管理测试。

全部测试成功通过，证实了 HwameiStor 能够完美适配 MinIO 存储方案。

Fluid

Fluid 是一个开源的 Kubernetes 原生的分布式数据集编排和加速引擎， 主要服务于云原生场景下的数据密集型应用，例如大数据应用、AI 应用等。

通过 Kubernetes 服务提供的数据层抽象，可以让数据像流体一样在诸如 HDFS、OSS、Ceph 等存储源和 Kubernetes 上层云原生应用计算之间灵活高效地移动、复制、驱逐、转换和管理。而具体数据操作对用户透明， 用户不必再担心访问远端数据的效率、管理数据源的便捷性，以及如何帮助 Kuberntes 做出运维调度决策等问题。 用户只需以最自然的 Kubernetes 原生数据卷方式直接访问抽象出来的数据，剩余任务和底层细节全部交给 Fluid 处理。

核心概念

- Dataset 数据集：通俗地说，就是应用要访问的数据集合。不同应用对应的数据集类型不同。
- Runtime 分布式缓存系统运行时：Runtime 是 Fluid 部署分布式缓存系统的一个标准框架，具体部署的分布式缓存系统就是具体的 Runtime。
  - AlluxioRuntime

  - JuiceFSRuntime
  - JinboFSRuntime
  - GooseFSRuntime
  - EFCRuntime
  - ThinRuntime

- Data access 用户数据访问：Fluid 提供了一个统一的 Fuse 接口给用户应用，该接口完全兼容 POSIX 协议。用户应用就像访问本地数据一样，访问远程数据集。

通过 Helm 模板部署 Fluid

DCE 5.0 支持了 Fluid， 并将其作为 Addon 集成了应用商店中。

1. 进入容器管理模块，在集群列表中找到需要安装 Fluid， 的集群，点击该集群的名称。

   点击集群名称

   点击集群名称

2. 在左侧导航栏中选择 Helm 应用 -> Helm 模板，找到并点击 Fluid。

   fluid-helm

   fluid-helm

3. 在安装界面，填写所需的安装参数，最后在右下角点击确定按钮。

   填写配置

   填写配置

   - 名称：组件的名称，可以输入 fluid。
   - 命名空间：选择新建命名空间，必须将名称设置为 fluid-system，否则部署会失败。
   - 版本：目前仅支持了 0.9.2。
   - 其他参数配置，使用默认参数即可。

4. 前往 Helm 应用查看部署结果。

   完成创建

完成创建

5. 也可以在当前集群详情左侧菜单栏的 工作负载 -> 容器组，选择命名空间为 fluid-system，查看所有容器组的状态。

pod

pod

参阅 Fluid 如何加速数据访问的 demo。

Kubernetes 存储

Kubernetes 为容器平台（或集群管理员）和应用开发人员提供了一些增强特性以支持运行有状态的工作负载。 这些特性确保了只要调度容器（包括卷的供应/创建、附加、挂载、解除挂载、拆分和删除）、存储容量管理（容器临时存储使用率、卷扩缩容等）， 都可以使用不同类型的文件和块存储（临时存储或持久化存储、本地存储或远程存储），这会影响基于存储的容器调度（数据重力、可用性等）和快照等常规存储操作。

在使用 HwameiStor 运行有状态工作负载时，需理解 Kubernetes 存储的几个抽象概念。

容器存储接口

容器存储接口 (CSI) 是在 Kubernetes 这种容器编排体系上将任意文件存储和块存储系统暴露给容器化工作负载的一个标准。 使用 CSI 后，像 HwameiStor 这样的第三方存储提供商可以写入和部署新的存储卷插件，例如 HwameiStor LocalDisk 和 LocalVolumeReplica，而无需修改 Kubernetes 核心代码。

当集群管理员安装 HwameiStor 时，所需的 HwameiStor CSI 驱动程序组件也将安装到 Kubernetes 集群中。

//在 CSI 之前，Kubernetes 支持使用 out-of-tree 资源调配器（也称为外部资源调配器）添加新的存储提供商。Kubernetes 树状卷 (in-tree) 早于外部资源调配器。而 Kubernetes 社区也在努力使用基于 CSI 的卷来替代树状卷。)

存储类和动态资源调配

StorageClass 为管理员提供了一种描述存储“类别”的方法。 不同的类别可能会映射到集群管理员确定的服务质量水平、备份策略或任意策略。 在某些存储系统中，StorageClass 这个概念称为“配置文件”。

有了动态资源调配功能后，集群管理员无需预先调配存储资源。 它会在用户请求时自动调配存储资源。动态卷资源调配的实现基于 StorageClass 这个抽象概念。 集群管理员可以根据需要，定义尽可能多的 StorageClass 对象，每个对象都会指定一个调配卷的插件（也称为调配器），并在资源调配时配置传递给该调配器的一组参数。

集群管理员可以在集群内定义和公开（来自相同或不同存储系统的）多种存储类型，每种类型都可以自定义参数。 这种设计还确保最终用户不必担心存储配置的复杂性和细微差别，但仍然能够从多个存储选项中进行选择。

安装 HwameiStor 后将附带了两个默认存储类，允许用户创建本地卷 (HwameiStor LocalVolume) 或副本 (HwameiStor LocalVolumeReplica)。集群管理员可以启用所需的存储引擎，然后为数据引擎创建存储类。

持久卷声明

PersistentVolumeClaim (PVC) 是由集群管理员提供的 StorageClass 服务的用户存储请求。 在容器上运行的应用可以请求某种类型的存储。例如，容器可以指定所需的存储大小或存取数据的方式（只读、单次读/写、多次读写等）。

除了存储大小和存取模式之外，管理员还可以创建存储类，为 PV 提供自定义属性，例如

磁盘类型（HDD 和 SSD）、性能水平或存储层级（常规存储或冷存储）。

持久卷

PersistentVolume (PV) 由存储提供商在用户请求 PVC 时动态调配。PV 包含容器如何消耗存储的详细信息。 Kubernetes 和卷驱动程序使用 PV 中的细节将存储附加/解除附加到容器运行的节点，并将存储挂载/解除挂载到容器。

HwameiStor 控制面动态设置 HwameiStor 本地卷和副本，并帮助在集群中创建 PV 对象。

有状态和无状态

Kubernetes 提供了几个内置的有状态和无状态负载资源，让应用开发人员定义在 Kubernetes 上运行的负载。 通过创建 Kubernetes 无状态/有状态负载，并使用 PVC 将其连接到 PV，可以运行有状态负载。

例如，可以用 YAML 创建引用 PVC 的 MySQL 无状态负载。无状态负载引用的 MySQL PVC 应该使用请求的大小和 StorageClass 创建。 一旦 HwameiStor 控制平面为所需的 StorageClass 和容量提供了 PV，则声明设置为已满足。 然后，Kubernetes 将挂载 PV 并启动 MySQL 无状态负载。

CAS 存储

容器附加存储 (Container Attached Storage, CAS) 是一种由 Kubernetes 编排的基于微服务存储控制器的软件存储体系。 其存储控制器作为 Kubernetes 集群的一部分在容器中进行管理和运行。 这就让 CAS 具有可移植性，可以在任何 Kubernetes 平台上运行，包括任何云平台、裸金属服务器或传统共享存储系统。 关键的是数据本身也可以通过容器访问，而不

是存储在非平台共享的横向扩展存储系统中。 由于 CAS 利用微服务架构，将存储解决方案与绑定到物理存储设备的应用程序保持密切关联，从而减少了 I/O 访问时间。

CAS 这种模式非常符合分布式数据的趋势，也适合采用微型松散耦合工作负载的小型自治团队。 例如，一个团队可能需要 Postgres 来提供微服务，而另一个团队的研发数据可能依赖于 Redis 和 MongoDB。 一些用例可能对性能要求较高，一些可能在 20 分钟内消失，一些是写入/读取密集型等等。 在一个大型企业中，随着规模的增长，企业越来越信任各团队自行选择所用的工具，各团队所依赖技术的差异性将越来越大。

CAS 意味着开发人员可以在工作时不必担心企业存储体系结构的底层需求。 对于 CAS 来说，云盘与 SAN、裸机或虚拟机并没有什么不同。开发人员和平台 SRE 工程师没必要开会决定下一个存储供应商，也没必要反复讨论如何设置才能支持用例。 开发人员保持自治，可以使用 Kubernetes 集群可用的任何存储来启动本身的 CAS 容器。

CAS 反映了一个更广泛的解决方案趋势，其中许多方案均是 CNCF 的项目，以 Kubernetes 和微服务为基础进行构建，形成繁荣的云原生生态体系。 如今 CNCF 的云原生生态体系包含了安全、DNS、网络、网络策略管理、消息传输、跟踪、日志等众多项目。

CAS 的优势

敏捷性

CAS 中的每个存储卷都有一个容器化存储控制器和相应的容器化副本。因此，围绕这些组件的资源维护和调整是真正敏捷化的。 Kubernetes 的滚动升级功能实现了存储控制器和存储副本的无缝升级。CPU 和内存等资源可以使用容器 cgroup 进行优化。

存储策略的粒度

对存储软件进行容器化，并将存储控制器专用于每个卷，可以实现存储策略的最大粒度。使用 CAS 体系结构，您可以按卷配置所有存储策略。 此外，您可以监视每个卷的存储参数，并动态更新存储策略，以实现每个工作负载的预期结果。 随着卷存储策略中粒度的增加，对存储吞吐量、IOPS 和延迟的控制也会增加。

避免绑定

避免被云服务商绑定是许多 Kubernetes 用户的共同目标。 然而，有状态应用的数据通常仍然依赖于云服务商及其技术，或者依赖于底层的传统共享存储系统 NAS 或 SAN。 而使用 CAS 方法后，存储控制器可以在每个工作负载的后台进行数据迁移，使得实时迁移变得更简单。 换句话说，CAS 的控制粒度以无中断的方式简化了有状态工作负载从一个 Kubernetes 集群到另一个集群的迁移。

原生云

CAS 将存储软件容器化，并使用 Kubernetes CRD 来表示底层存储资源，如磁盘和存储池。这种模式使存储能够无缝地集成到其他云主机的工具中。 可以使用 Prometheus、Grafana、Fluentd、Weavescope、Jaeger 等云主机工具来调配、监控和管理存储资源。

另外，CAS 中卷的存储和性能是可扩缩的。由于每个卷都有自己的存储控制器，因此可以在节点存储容量的允许范围内弹性扩缩。 随着给定 Kubernetes 集群中容器应用数量的增加，会添加更多节点，从而提高存储容量和性能的总体可用性，使存储可用于新的应用容器。

更小的影响范围

由于 CAS 体系结构是按工作负载划分的，并且组件是松散耦合的，所以 CAS 的影响范围比典型的分布式存储体系结构小得多。

CAS 可以通过从存储控制器到存储副本的同步复制来提供高可用性。维护副本所需的元数据简化为副本节点的信息以及有关副本状态的信息。 如果某个节点出现故障，存储控制器将通过第二个或第三个副本在正运行且数据可用的节点上轮转。 因此，使用 CAS 时，影响范围要小得多，影响仅局限在该节点上有副本的卷。

CSI 接口

CSI 即 Container Storage Interfaces，容器存储接口。 目前，Kubernetes 中的存储子系统仍存在一些问题。 存储驱动程序代码在 Kubernetes 核心存储库中进行维护，这很难测试。 Kubernetes 还需要授予存储供应商许可，便于将代码嵌入 Kubernetes 核心存储库。

CSI 旨在定义行业标准，该标准将使支持 CSI 的存储提供商能够在支持 CSI 的容器编排系统中使用。

下图描述了一种与 CSI 集成的高级 Kubernetes 原型。

CSI 接口

CSI 接口

- 引入了三个新的外部组件以解耦 Kubernetes 和存储提供程序逻辑
- 蓝色箭头表示针对 API 服务器进行调用的常规方法
- 红色箭头显示 gRPC 以针对 Volume Driver 进行调用

扩展 CSI 和 Kubernetes

为了实现在 Kubernetes 上扩展卷的功能，应该扩展几个组件，包括 CSI 规范、“in-tree”卷插件、external-provisioner 和 external-attacher。

扩展 CSI 规范

最新的 CSI 0.2.0 仍未定义扩展卷的功能。应引入新的 3 个 RPC：RequiresFSResize、ControllerResizeVolume 和 NodeResizeVolume。

```
service Controller {
  rpc CreateVolume (CreateVolumeRequest)
    returns (CreateVolumeResponse) {}
……
  rpc RequiresFSResize (RequiresFSResizeRequest)
    returns (RequiresFSResizeResponse) {}
  rpc ControllerResizeVolume (ControllerResizeVolumeRequest)
    returns (ControllerResizeVolumeResponse) {}
}
service Node {
  rpc NodeStageVolume (NodeStageVolumeRequest)
    returns (NodeStageVolumeResponse) {}
……
  rpc NodeResizeVolume (NodeResizeVolumeRequest)
    returns (NodeResizeVolumeResponse) {}
}
```

扩展 “In-Tree” 卷插件

除了扩展的 CSI 规范之外，Kubernetes 中的 csiPlugin 接口还应实现 expandablePlugin。csiPlugin 接口将扩展代表 ExpanderController 的 PersistentVolumeClaim。

```
type ExpandableVolumePlugin interface {
VolumePlugin
ExpandVolumeDevice(spec Spec, newSize resource.Quantity, oldSize resource.Quantity) (resource.
Quantity, error)
RequiresFSResize() bool
}
```

实现卷驱动程序

最后，为了抽象化实现的复杂性，应将单独的存储提供程序管理逻辑硬编码为以下功能，这

些功能在 CSI 规范中已明确定义：

- CreateVolume
- DeleteVolume
- ControllerPublishVolume
- ControllerUnpublishVolume
- ValidateVolumeCapabilities
- ListVolumes
- GetCapacity
- ControllerGetCapabilities
- RequiresFSResize
- ControllerResizeVolume

展示

以具体的用户案例来演示此功能。

- 为 CSI 存储供应商创建存储类

```
allowVolumeExpansion: true
    apiVersion: storage.k8s.io/v1
    kind: StorageClass
    metadata:
      name: csi-qcfs
    parameters:
      csiProvisionerSecretName: orain-test
      csiProvisionerSecretNamespace: default
    provisioner: csi-qcfsplugin
    reclaimPolicy: Delete
    volumeBindingMode: Immediate
```

- 在 Kubernetes 集群上部署包括存储供应商 csi-qcfsplugin 在内的 CSI 卷驱动

- 创建 PVC qcfs-pvc，它将由存储类 csi-qcfs 动态配置

```
apiVersion: v1
    kind: PersistentVolumeClaim
    metadata:
      name: qcfs-pvc
      namespace: default
```

```
....
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 300Gi
  storageClassName: csi-qcfs
```

- 创建 MySQL 5.7 实例以使用 PVC qcfs-pvc
- 为了反映完全相同的生产级别方案，实际上有两种不同类型的工作负载，包括：
  - 批量插入使 MySQL 消耗更多的文件系统容量
  - 浪涌查询请求
- 通过编辑 pvc qcfs-pvc 配置动态扩展卷容量

CRD 和 CR

CRD

CRD 是 Custom Resource Definition 的缩写，是 Kubernetes 内置原生的一个资源类型。 它是自定义资源（CR）的定义，用来描述什么是自定义资源。

CRD 可以向 Kubernetes 集群注册一种新资源，用于拓展 Kubernetes 集群的能力。 有了 CRD，就可以自定义底层基础设施的抽象，根据业务需求来定制资源类型，利用 Kubernetes 已有的资源和能力，通过乐高积木的模式定义出更高层次的抽象。

CR

CR 是 Custom Resource 的缩写，它实际是 CRD 的一个实例，是符合 CRD 中字段格式定义的一个资源描述。

CRDs + Controllers

我们都知道 Kubernetes 的扩展能力很强大，但仅有 CRD 并没有什么用，还需要有控制器 (Custom Controller) 的支撑，才能体现出 CRD 的价值，Custom Controller 可以监听 CR 的 CRUD 事件来实现自定义业务逻辑。

在 Kubernetes 中，可以说是 CRDs + Controllers = Everything。

另请参考 Kubernetes 官方文档：

- CustomResource
- CustomResourceDefinition

Volume

容器中的文件在磁盘上是临时存放的，这给容器中运行的较重要的应用程序带来一些问题。

- 问题一：当容器崩溃时文件会丢失。kubelet 会重新启动容器，但容器会以干净的状态重启。
- 问题二：在同一个 Pod 中运行多个容器并共享文件时出现此问题。

Kubernetes 卷 (Volume) 这一抽象概念能够解决这两个问题。

Kubernetes 支持很多类型的卷。Pod 可以同时使用任意数目的卷类型。 临时卷类型的生命周期与 Pod 相同，但持久卷可以比 Pod 的存活期长。 当 Pod 不再存在时，Kubernetes 也会销毁临时卷；不过 Kubernetes 不会销毁持久卷。 对于给定 Pod 中任何类型的卷，在容器重启期间数据都不会丢失。

卷的核心是一个目录，其中可能保存了数据，Pod 中的容器可以访问该目录中的数据。 所采用的特定卷类型将决定该目录如何形成、使用何种介质保存数据以及目录中存放的内容。

使用卷时，在 .spec.volumes 字段中设置为 Pod 提供的卷，并

在 .spec.containers[*].volumeMounts 字段中声明卷在容器中的挂载位置。

参考 Kubernetes 官方文档：

- 卷
- 持久卷
- 临时卷

LVM

LVM 全称为 Logical Volume Manager，即逻辑卷管理器，在磁盘分区和文件系统之间添加一个逻辑层，为文件系统屏蔽下层磁盘分区布局提供一个抽象的盘卷，在盘卷上建立文件系统。利用 LVM 可以在磁盘不用重新分区的情况下动态调整文件系统的大小，并且利用 LVM 管理的文件系统可以跨越磁盘。 当服务器添加了新的磁盘后，管理员不必将原有的文件移动到新的磁盘上，而是通过 LVM 直接扩展文件系统跨越磁盘。 也就是通过将底层的物理硬盘封装起来，然后以逻辑卷的方式呈现给上层应用。

LVM 通过对底层的硬盘进行封装，当我们对底层的物理硬盘进行操作时，不再是针对分区进行操作，而是通过一个叫做逻辑卷的东西对其进行底层的磁盘管理操作。

LVM 主要构成

- 物理存储介质（PM, physical media）：LVM 存储介质可以是分区、磁盘、RAID 阵列或 SAN 磁盘。
- 物理卷（PV, physical volume）：物理卷是 LVM 的基本存储逻辑块，但与基本的物理存储介质（如分区、 磁盘等）比较，却包含有与 LVM 相关的管理参数，创建物理卷可以用磁盘分区，也可以用磁盘本身。磁盘设备必须初始化为 LVM 物理卷，才能与

LVM 结合使用。

- 卷组 (VG, Volume Group)：LVM 卷组由一个或多个物理卷组成。
- 逻辑卷 (LV, logical volume)：LV 建立在 VG 之上，可以在 LV 之上建立文件系统。
- 物理范围 (PE, physical extents)：PV 物理卷中可以分配的最小存储单元，PE 的大小是可以指定的，默认为 4MB。
- 逻辑范围 (LE, logical extents)：LV 逻辑卷中可以分配的最小存储单元，在同一个卷组中，LE 的大小与 PE 是相同的， 并且一一对应。

LVM 优点

- 使用卷组，使多个硬盘空间看起来像是一个大的硬盘
- 使用逻辑卷，可以跨多个硬盘空间的分区 sdb1 sdb2 sdc1 sdd2 sdf
- 使用逻辑卷，可以在空间不足时动态调整它的大小
- 在调整逻辑卷大小时，不需要考虑逻辑卷在硬盘上的位置，不用担心没有可用的连续空间
- 可以在线对 LV、VG 进行创建、删除、调整大小等操作，LVM 上的文件系统也需要重新调整大小
- 允许创建快照，可以用来保存文件系统的备份
- RAID + LVM 结合使用：LVM 是软件的卷管理方式，而 RAID 是磁盘管理的方法。对于重要的数据，使用 RAID 来保护物理磁盘不会因为故障而中断业务，再用 LVM 来实现对卷的良性管理，更好地利用磁盘资源。

使用 LVM 的基本步骤

1. 物理磁盘被格式化为 PV，即空间被划分为一个个的 PE。PV 包含多个 PE。
2. 不同的 PV 加入到同一个 VG 中，即不同 PV 的 PE 全部进入到 VG 的 PE 池内。VG 包含多个 PV。
3. 在 VG 中创建 LV 逻辑卷，这个创建过程基于 PE，所以组成 LV 的 PE 可能来自不同的物理磁盘。LV 基于 PE 创建。
4. LV 直接可以格式化后挂载使用。
5. LV 的扩缩实际上就是增加或减少组成该 LV 的 PE 数量，其过程不会丢失原始数据。
6. 格式化 LV，并挂载使用。

LV 扩容

首先，确定是否有可用的扩容空间，因为空间是从 VG 里面创建的，并且 LV 不能跨 VG 扩容。若 VG 没有了容量，需要先扩 VG。步骤如下：

```
$ vgs
VG #PV #LV #SN Attr VSize VFree
vg-sdb1 1 8 1 wz--n- <16.00g <5.39g
$ lvextend -L +100M -r /dev/vg-sdb1/lv-sdb1 #将 /dev/vg-sdb1/lv-sdb 扩容 100M
```

VG 扩容

如果 VG 卷组中的空间不够了，需要添加新的磁盘，依次运行以下命令：

```
pvcreate /dev/sdc
vgextend vg-sdb1 /dev/sdb3
```

LV 快照

LVM 机制提供了对 LV 做快照的功能，以此来获得文件系统的状态一致性备份。 LVM 采用

写时复制技术（Copy-On-Write，COW），不用停止服务或将逻辑卷设为只读就可以进行备份，使用 LVM 快照功能既可以获得一致备份，又不会影响服务器的可用性。

LVM 采用的写时复制，是指创建 LVM 快照时，仅复制原始卷中数据的元数据。 换句话说，也就是在创建 LVM 逻辑卷时，并不会发生数据的物理复制。 再换句话说，仅复制元数据，不复制物理数据，因此快照的创建几乎是实时的。 当原始卷上执行写入操作时，快照会跟踪原始卷中块的变更，这时原始卷上将要变更的数据会在变更之前拷贝到快照预留的空间。

PV 和 PVC

PV（PersistentVolume，持久卷）是对存储资源的抽象，将存储定义为一种容器应用可以使用的资源。 PV 由管理员创建和配置，与存储提供商的具体实现直接相关，例如文件存储、块存储、对象存储或 DRBD 等， 通过插件式的机制进行管理，供应用访问和使用。除 EmptyDir 类型的存储卷，PV 的生命周期独立于使用它的 Pod。

PVC（PersistentVolumeClaim，持久卷声明）是用户对存储资源的一个申请。 就像 Pod 消耗 Node 的资源一样，PVC 会消耗 PV 的资源。PVC 可以申请存储空间的大小（Size）和访问模式（例如 ReadWriteOnce、ReadOnlyMany 或 ReadWriteMany)。

使用 PVC 申请的存储空间仍然不满足应用对存储设备的各种需求。 在很多情况下，应用程序对存储设备的特性和性能都有不同的要求，包括读写速度、并发性能、数据冗余等要求。这就需要使用资源对象 StorageClass，用于标记存储资源和性能，根据 PVC 的需求动态供给合适的 PV 资源。 StorageClass 和存储资源动态供应的机制经完善后，实现了存储卷的按需创建，在共享存储的自动化管理进程中实现了重要的一步。

另请参考 Kubernetes 官方文档：

- 持久卷

- 存储类
- 动态卷供应