K8s持久化存储

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 查看K8S 支持哪些存储  kubectl explain pods.spec.volumes  👉 用途：查看 Pod 规范（spec）中 volumes 字段的详细定义与可选类型。    只讲:        访问模式的区别（进对比rwo和rwx）：   **RWO** → 本地存储，PVC/PV 都定义为 RWO，PVC 会自动绑定 Available 的 PV   **RWX** → 共享存储，PVC/PV 都定义为 RWX，PVC 会自动绑定可用 PV   **引用 PV 名称** 是可选的，只有你想指定特定 PV 才需要写 volumeName   **自动绑定** 是 Kubernetes 最常用方式          rwo示例：  每个节点都安装：nfs     |  |  |  | | --- | --- | --- | | **角色** | **是否必须安装 nfs-utils** | **作用** | | **NFS 服务端** | ✅ 必须 | 提供共享目录（含 nfsd 服务、exportfs 命令） | | **NFS 客户端** | ✅ 必须 | 挂载远程共享目录（提供 mount.nfs 工具） |   Master 搭建nfs服务器：创建目录+定义可挂载主机：  /data/nfs     |  |  | | --- | --- | | **选项** | **含义** | | **rw** | 允许客户端读写（默认是只读 ro） | | **ro** | 只读（read-only） | | **root\_squash** | **默认启用**，会把客户端的 root 用户映射为匿名用户（nfsnobody），防止 root 越权访问服务器文件系统 | | **no\_root\_squash** | **禁用 root\_squash**，即：客户端 root 用户保留 root 权限，可以在共享目录中创建、修改、删除文件 | | **all\_squash** | 把所有访问者（不论 UID）都映射成匿名用户 | | **anonuid / anongid** | 指定匿名用户对应的 UID/GID |   这里不建议用no\_root\_squash,rootn会不映射为普通用户nfsnobody  启动nfs：    Master：创建pv：rwo  挂载ip写错了：  当 PVC 被删除后，PV 的数据和状态需要处理，这就涉及 **回收策略（Reclaim Policy）**。  **2️⃣ PV 的回收策略类型**  Kubernetes PV 主要有三种回收策略：   |  |  |  | | --- | --- | --- | | **策略** | **含义** | **说明** | | **Retain（保留）** | 保留 PV 和数据 | PVC 删除后，PV 不会被删除，也不会自动回收。管理员可以手动处理数据。此时pv状态变成：released | | **Recycle（回收）** | 简单回收 | PVC 删除后，PV数据删除，然后重新变为 Available 状态。**注意：K8s 1.11 已弃用**。 | | **Delete（删除）** | 自动删除 | PVC 删除后，PV 对应的存储会被删除（比如云盘资源被释放），适合动态供应的存储（StorageClass）。 |   **PV 的主要状态**   |  |  | | --- | --- | | **状态** | **含义** | | **Available** | PV 尚未被任何 PVC 使用，可供绑定。 | | **Bound** | PV 已经被某个 PVC 绑定，正在使用中。 | | **Released** | PVC 已经删除，但 PV 的数据仍未被回收（根据回收策略处理）。 | | **Failed** | PV 回收或绑定失败，无法使用。 |   pv和pvc地数量关系？  **PV 是否可以绑定多个 PVC？**   * **答案：不可以** * 一个 PV **一次只能绑定一个 PVC** * 如果想让多个 PVC 共享存储，需要使用 **ReadWriteMany (RWX)** 类型的存储卷（如 NFS、CephFS、GlusterFS 等），但这还是 **同一个 PV 同时被多个 PVC 挂载**，而不是重复创建 PVC 绑定同一个 PV。 * 常规 PV（本地盘、云盘）只能单 PVC 绑定，类型通常是 **ReadWriteOnce (RWO)**   pvc绑定pv的要求：  **PVC 匹配 PV 的条件**  PVC 自动绑定 PV 时，**必须同时满足多个条件**：   1. **容量（storage）**    * PV 的 capacity.storage 必须 **大于等于** PVC 请求的存储量 2. **访问模式（accessModes）**    * PV 的 accessModes 必须 **包含** PVC 请求的访问模式    * 例如：      + PVC 请求 ReadWriteOnce → PV 也必须至少支持 RWO      + PVC 请求 ReadWriteMany → PV 必须支持 RWX 3. **StorageClass（可选）**    * 如果 PVC 指定了 storageClassName，PV 必须匹配同名 StorageClass 4. **Selector（可选）**    * PVC 可以通过标签选择特定 PV   如果其中任意条件不满足，PVC **不会绑定** 该 PV。  **权限包含关系总结**  从“权限强度”角度：  ROX < RWO < RWX  但注意：   * **ROX 与 RWO 并无直接包含关系**（一个是多节点只读，一个是单节点读写） * 只有 **RWX** 同时覆盖前两种模式的能力（最强）   ✅ **最终总结一句话：**  PVC 的访问模式必须被 PV 支持；RWX PV 可以匹配 RWO/ROX PVC，但 RWO/ROX PV 无法匹配 RWX PVC。 实际挂载时，以 PVC 的访问模式为准。  **一、RWO、ROX、RWX 的挂载行为与绑定关系不是一回事**   | **概念** | **含义** | | --- | --- | | **绑定（Binding）** | 一个 PVC 和一个 PV 建立的逻辑一对一关系（K8s 对象层面） | | **挂载（Mount）** | Pod 运行时访问存储的行为，可以多节点、多 Pod |   **✅ 二、Kubernetes 设计规定：**  **一个 PV 只能被一个 PVC 绑定（即使它支持 RWX 或 ROX）。**  **🔹 原因：**  K8s 的 PV/PVC 是“声明式资源绑定”，而非“多用户共享资源池”。   * 每个 PVC 对应一个 PV（1:1） * 多个 Pod 想共享一个 RWX 卷，**需要多个 Pod 挂载同一个 PVC**（不是多个 PVC）   静态pv容量浪费问题：  静态 PV 空间浪费 → 两种主要解决方向   |  |  |  |  | | --- | --- | --- | --- | | **方案** | **原理** | **优点** | **适用场景** | | **① 动态供给（StorageClass + CSI）** | PVC 申请多少，StorageClass 自动创建等量 PV | ✨最灵活、无浪费、全自动绑定 | 大多数生产场景（EBS、Ceph、NFS、Longhorn 等） | | **② 共享大 RWX PVC** | 创建一个大容量共享卷（RWX），多个 Pod 共用同一 PVC 下不同目录 | ✨减少 PV/PVC 数量、避免重复空间 | 多 Pod 共享数据（日志、上传区、缓存目录） |  |  |  |  | | --- | --- | --- | | **对比项** | **动态供给** | **大 RWX PVC** | | **是否自动创建 PV** | ✅ 是 | ❌ 否（仍需手动或提前准备） | | **空间浪费** | ❌ 无浪费 | ⚠️ 有可能被不同业务“挤占” | | **隔离性** | ✅ 强（每 PVC 独立卷） | ❌ 弱（共用一个卷） | | **共享能力** | 一般（取决于后端支持） | 强（天然多 Pod 共享） | | **适合用途** | 独立业务数据（数据库、配置、缓存等） | 多服务共享资源（日志、上传、CI 缓存等） |   master定义pvc    Master：定义pod，使用pvc：    Master：应用：pv、pvc、pod          改一下ip后，重新启动  测试挂载成功：  看pod的container内创建文件，Nfs主机共享目录是否可见：    Nfs共享主机目录：    默认node是不可创建资源对象的，因为联系不上master的serverapi pod：  必须复制master的/etc/kubernetes/admin.conf文件到同级目录下，即可访问serverapi，也就可以创建资源对象了   admin.conf 内容 → 提供 API Server 地址、认证信息、上下文信息   拷贝后 node 的 kubectl 就能访问 API Server，并有 cluster-admin 权限   |  |  | | --- | --- | | **概念** | **含义** | | **cluster-admin 权限** | 一种 Kubernetes RBAC 权限，拥有对整个集群资源的读写能力（Pod、Deployment、PVC 等） | | **master 节点** | Kubernetes 集群中运行 **控制平面组件**（API Server、Scheduler、Controller Manager、etcd）的物理或虚拟节点 |   **✅ 一、PV 和 PVC 的作用域关系**   | **对象** | **是否属于某个 namespace** | **能否跨 namespace 使用** | | --- | --- | --- | | **PersistentVolume（PV）** | ❌ **集群级资源（Cluster-scoped）** | ✅ 可以被任意命名空间的 PVC 绑定 | | **PersistentVolumeClaim（PVC）** | ✅ **命名空间内资源（Namespace-scoped）** | ❌ 只能在所属 namespace 内被 Pod 使用 |   🧩 **说明：**   * PV 是“整个集群的物理卷池”（Cluster 全局的存储资源） * PVC 是“某个命名空间的申请书” * 绑定关系是单向的：   PVC（namespace 内） → 绑定 → PV（cluster 全局）  **✅ 二、Pod 使用存储的范围限制**   * Pod **只能挂载同命名空间内的 PVC** * 不能挂载其他 namespace 的 PVC（K8s 设计上隔离租户）   # ❌ 以下场景非法  Pod (ns: team-a)  挂载 PVC (ns: team-b)  如果要实现“跨命名空间共享数据”，有两种方法：  Kubernetes 资源作用域（Scope）：   |  |  |  | | --- | --- | --- | | **类型** | **Scope** | **是否跨 Namespace 可见** | | PV | 🌐 Cluster 级 | ✅ 可跨命名空间被绑定 | | PVC | 🧩 Namespace 级 | ❌ 仅当前 namespace 可见 | | Pod / Deployment / Service / ConfigMap / Secret | 🧩 Namespace 级 | ❌ 不可跨命名空间 | | Node / StorageClass / ClusterRole / ClusterRoleBinding | 🌐 Cluster 级 | ✅ 全局生效 |   Deployment+pvc案例：  定义deployment：     1. replicas: 3 → 会创建 **3 个 Pod** 2. volumeMounts 都挂载 **同一个 PVC** (my-pvc) 3. 也就是说 **这 3 个 Pod 共享同一个存储卷**   **PVC 的访问模式要求**   * 因为 **3 个 Pod 会同时挂载同一个 PVC** * 所以 PVC 对应的 PV **必须支持多节点读写（RWX）** * **RWO**（单节点读写）不允许多个 Pod 同时读写，会导致 Pod 启动失败   ⚠️ 注意：   * RWX PVC → 可以被 **多个 Pod 同时挂载**（同一 namespace） * 如果 PV 是本地磁盘类型，通常只能 RWO → 不能用于这种共享场景   启动后可以看见nfs共享目录已创建的文件：    可以看见，挂载成功： |

Storageclass

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 本地存储storageclass示例：  Node节点创建目录+授权：    回到master，给node1打标签：      Master节点定义storageclass：    定义：  三个provisioner、reclaimpolicy、volumebindmodel  ** provisioner: kubernetes.io/no-provisioner**   * **表示 本地卷（local volume）没有自动创建 PV 的能力，需要你手动定义 PV。** * **StorageClass 只是做逻辑绑定，不会真的“创建”存储。**   ** reclaimPolicy: Retain**   * **正确。作为默认值，动态创建的 PV 会继承它。** * **手动创建 PV 时可以不写（可省略），因为 StorageClass 会作为默认策略应用。**   ** volumeBindingMode: WaitForFirstConsumer**   * **非常关键！表示：**   **PVC 不会立即绑定到某个 PV，而是 等到有 Pod 真正使用这个 PVC 时， 才会根据 Pod 的调度决策去选择一个匹配节点的 PV。**  本地磁盘pv只能使用WaitForFirstConsumer  就是本地磁盘类型的pv只能挂载本地pod，VC 提前绑定 PV → Pod 调度时不会找匹配PV 和pod；   设置 volumeBindingMode: WaitForFirstConsumer   PVC 不会立即绑定 PV   Pod 创建时 Scheduler 会**同时考虑 Pod 所在节点和 PV 节点**   Scheduler 会把 Pod 调度到 PV 所在的节点，或者选择一个节点上有可用 PV 的 PV 绑定   保证 Pod 能访问 PV → Pod 成功启动  应用storageclass看效果：    没有动态插件，还是得创建pv：    Capacity、accessmodel、persistentvolumereclaimpolicy、storageclassname、localpath、nodeaffinity  静态 PV：PVC 一创建就绑定，节点由你手动控制。 ✔️ StorageClass + 本地 PV：PVC 等 Pod 调度后再绑定，由调度器自动选对节点。  Reclaimpolicy优先级：   |  |  |  | | --- | --- | --- | | **对象** | **reclaimPolicy 生效范围** | **作用说明** | | **PersistentVolume (PV)** | 单个 PV | 当 PVC 删除后，这个 PV 本身该怎么处理（保留/删除/回收） | | **StorageClass** | 新创建的 PV（通过这个 StorageClass 动态创建） | 给通过 StorageClass 创建的 PV 设置默认回收策略，如果 PV 本身手动定义了 reclaimPolicy，则 PV 自己的值优先 |   两个都是设置pvc回收策略的，是完全一样的，只不过优先级不同，PV 自己的值优先，StorageClass的reclaimpolicy作为pv的默认reclaimpolicy值  应用pv：    定义pvc：    Pvc定义accessmodel、storage、storageclassname  应用：      定义pod：    Containers（volvmemounts）、volumes、nodeselector（可以）因为有 WaitForFirstConsumer，调度器会根据 PVC 选择正确的节点（即与 PV 匹配的节点）  Pod ─→ PVC ─→ StorageClass ─→ PV  验证挂载：  Pod调度到node1：    应用pod，进入pod在/data/创建东西：    在node1的  目录看到    原目录创建都是能看到的    容器内挂载目录看到：    **这种storageclass+localpath不常用**  NFS作为后端存储并通过StorageClass实现动态供给  全部删除干净    导入存储插件：两个node节点    定义serveraccount资源对象：      NFS 动态存储（provisioner）赋权：  kubectl create clusterrolebinding nfs-provisioner \  --clusterrole=cluster-admin \  --serviceaccount=default:nfs-provisioner  kubectl create clusterrolebinding 创建一个 **ClusterRoleBinding** 对象  nfs-provisioner **ClusterRoleBinding 的名字**。你可以理解为“这个绑定规则的名字  **--clusterrole=cluster-admin**  把一个 **集群级别的角色（ClusterRole）** 叫做 cluster-admin 的权限， 赋给接下来的对象（也就是 NFS provisioner 的 ServiceAccount）。  cluster-admin 是一个内置的最高权限角色，它几乎可以操作所有资源。  **--serviceaccount=default:nfs-provisioner**  指定目标账号是谁：   * default 是命名空间； * nfs-provisioner 是 ServiceAccount 的名字。   也就是说：  “给命名空间 default 下的 ServiceAccount nfs-provisioner 授予 cluster-admin 权限。”  **为什么 NFS provisioner 需要权限？**  NFS Provisioner（比如 nfs-subdir-external-provisioner）是一个 **Pod 控制器**，它需要在运行时：   * 监听 PVC 的创建； * 动态创建对应的 PV； * 更新 PV/PVC 状态； * 删除存储卷。   这些操作涉及 **全局资源管理**，所以必须拥有较高权限，否则 Kubernetes API 会拒绝访问（报 403 Forbidden）。  这条命令是把最高权限 cluster-admin 赋给 NFS provisioner 的服务账号，让它有权自动创建、管理、删除 PV 和 PVC，从而实现 **动态存储供应（Dynamic Provisioning）**。  配置nfs目录和export权限文件：（master主机）    创建挂载目录：    添加一条    再导入一次：    定义deployment provisioner：它就是一个 Deployment，对应一个控制器 Pod    指定:  Replicas:1  Strage.tpye:recteate  Serviceaccount:  Image:provision镜像  Volumemount+volumes  Env：设置provisioner name、servier、path  要声明两遍nfs：  **一、确实要配置“两遍”，但作用不同**   |  |  |  | | --- | --- | --- | | **配置部分** | **作用** | **举例** | | **volumes + volumeMounts** | 让 provisioner 这个 Pod **能访问真实的 NFS 目录**（程序才能在里面创建子目录、写文件） | mountPath: /persistentvolumes 对应 NFS /data/nfs\_pro | | **env 环境变量** | 告诉 provisioner 程序“我管理的 NFS 是谁”，方便它在创建 PV 时填入正确的 NFS 信息 | NFS\_SERVER: 192.168.80.10，NFS\_PATH: /data/nfs\_pro |   💬 也就是说：  一个是“物理访问通道”， 一个是“逻辑管理声明”。  如果只挂载卷，不写 env：程序能访问 NFS，但不知道怎么注册 PV； 如果只写 env，不挂载卷：程序知道 NFS 是谁，但无法实际访问目录创建子路径。  **Deployment 更新策略（spec.strategy.type）**  Deployment 支持两种更新策略：   |  |  | | --- | --- | | **策略类型** | **说明** | | **RollingUpdate**（默认） | 滚动更新：逐步替换 Pod，每次只替换一部分，保证服务不中断。 | | **Recreate** | 重建更新：先 **删除所有旧 Pod**，再创建新 Pod。更新期间 Pod 不可用。 |   为什么 NFS Provisioner 推荐 Recreate？  FS Provisioner 有几个特点：   1. **单实例运行**：一般只运行一个副本（replicas: 1），多个实例可能同时操作 NFS 根目录 /persistentvolumes 会产生冲突。 2. **需要独占管理目录**：如果同时运行多个 Pod，可能会重复创建 PV 或子目录。 3. **不需要高可用**：Provisioner 本身只是控制 PV 创建，短暂停机不会影响现有 Pod。   所以选择：  strategy:  type: Recreate  可以保证旧 Pod 停掉后，新的 Pod 完全接管 NFS 根目录，避免资源冲突。  我的理解：  思是说NFS Provisioner 推荐 Recreate是因为一般replicas: 1，所以设置这样合适，防止滚动更新中间有两个pod，同时操作nfs会冲突，所以建议先删除原pod在开启新pod  **这个容器镜像（nfs-subdir-external-provisioner）本身就是一个自动 NFS 目录管理器**。  **NFS 动态存储的整体链路**  你总结的这句其实就是标准流程：  “把 nfs 挂载到 provisioner 的 deployment 容器 → provisioner 自动管理 NFS → StorageClass 指定这个 provisioner → PVC 指定 StorageClass → Pod 指定 PVC”  创建storageclass：    指定provisioner  创建pvc：    这里pvc声明1Gi,后续创建的也是1Gi的pv  指定storageclass  定义pod：    指定pvc  全部应用：  应用provisioner      应用storageclass    应用pvc：    创建完pvc自动创建pv：    应用pod：    进入read-pod：  目录验证挂载：    Master nfs主机查看：    主机创建目录：    容器内查看：    本地存储不太好，因为localpath宕机，不可用，但是nfs一般有备份，而且动态分配比较安全和利用率高  态pv的storageclass默认是Immediate + Delete；静态 PV 的storageclass默认 Immediate + Retain   |  |  |  | | --- | --- | --- | | **PV 类型** | **StorageClass 默认值** | **说明** | | **动态 PV** | volumeBindingMode: Immediate + reclaimPolicy: Delete | PVC 创建时立即绑定 PV；PVC 删除后 PV 自动删除 | | **静态 PV** | volumeBindingMode: Immediate + reclaimPolicy: Retain | PVC 创建时立即绑定 PV；PVC 删除后 PV 保留，需要管理员手动清理 |   因此：   **动态 PV（通过 StorageClass + Provisioner 创建）**   * PV 是由 Provisioner 自动生成的，所以 **只能继承 StorageClass 的默认值**（volumeBindingMode、reclaimPolicy 等），除非 PV 创建后手动修改（不推荐）。    **静态 PV（管理员手动创建）**   * PV 是自己定义的，所以 **可以继承 StorageClass 的默认值，也可以显式覆盖** |