Docker 与 K8S 技术串讲

MLE 算法指北

2025 年 3 月 17 日

1 太长不看版 – 常见问题汇总

1.1 k8s 相关理念

1.2 K8s 采用不可变基础设施，有什么优缺点？

优点：

- **确保一致性**，避免不同环境之间的不一致问题。
- **提高可复现性**，保障测试、生产环境一致。
- **减少环境漂移**，所有变更均通过新版本管理，防止手动修改配置导致不稳定。
- **方便版本回滚**，可以快速切换到稳定版本，提高系统可靠性。

缺点：

- **存储消耗大**，不同版本的镜像需要额外存储空间。
- **无法直接修改运行中的状态**，所有变更必须通过构建新镜像并重新部署。

1.2.1 如何优化镜像存储，避免浪费？

- **Docker 镜像分层技术 **，避免重复存储相同的部分，只存储变更内容。
- ** 使用 CI/CD 结合镜像清理策略 **：
  - 通过 **Harbor、Docker Registry** 设置镜像生命周期策略，定期删除过期镜像。
  - 使用 ‘docker image prune‘ 命令清理未使用的镜像层。

1.2.2 容器本地配置修改后，重启是否失效？如何避免？

问题：

- ** 是的，修改会丢失 **，因为容器是无状态的。

解决方案：

- ** 使用 ConfigMap 和 Secret 存储配置 **，适用于环境变量、非敏感和敏感配置信息。
- ** 将配置文件挂载到 PersistentVolume（PV）**，确保 Pod 重启后仍能访问原数据。
- ** 直接打包到新镜像 **，适用于固定不变的配置。

1.2.3 K8s 如何从当前状态变更到目标状态？

- Kubernetes 采用 ** 声明式配置（Declarative Configuration）**，用户通过 YAML 文件定义目标状态。
- ** 控制器（Controller）**（如 Deployment）持续监控实际状态，并调整使其与目标状态保持一致。

1.2.4 如何精细控制状态变更？

- **Rolling Update（滚动升级）**：逐步替换 Pod，确保无缝更新，避免服务中断。
- ** 探针机制（LivenessProbe、ReadinessProbe）**：
  - **LivenessProbe**：检查容器是否存活，失败则重启。
  - **ReadinessProbe**：检查服务是否准备就绪，未就绪的 Pod 不会接受流量。

1.3 容器基础概念

1.3.1 镜像与容器

- 镜像：由 **Dockerfile** 构建，包含应用代码、依赖和运行环境。
- 容器：由镜像启动的实例，隔离运行应用。

1.3.2 容器与镜像的关系，容器运行时（CRI）是什么？

- 容器是由镜像启动的实例。
- CRI 允许 Kubernetes 与不同的容器运行时（如 Docker、Containerd、CRI-O）通信。

1.3.3 容器与镜像的关系，容器运行时（CRI）是什么？

- 容器是由镜像启动的实例。
- CRI 允许 Kubernetes 与不同的容器运行时（如 Docker、Containerd、CRI-O）通信。

1.3.4 Pod 为什么通常只运行一个容器？

- 一个 Pod 可以运行多个容器，但通常只有一个主容器，其他容器（Sidecar）用于日志收集、代理等辅助功能。

1.3.5 升级一个容器的镜像会导致其他容器重启吗？

- 不会，Pod 级别的变更不会影响其他 Pod 内的容器，除非它们共享相同的存储或网络。

1.4 资源与配置

1.4.1 节点池与 K8s 集群的关系？

- 节点池是一组具有相同配置的节点，一个 K8s 集群可以包含多个节点池。

1.4.2 ReplicaSet 和 Deployment 的区别？

- ReplicaSet 只负责维持 Pod 的副本数。Deployment 在 ReplicaSet 之上，提供滚动升级、回滚等能力。

1.4.3 为什么生产环境建议直接使用 Deployment 而不是 ReplicaSet？

- Deployment 管理 ReplicaSet，支持平滑升级、回滚，而单独使用 ReplicaSet 不具备这些功能。

1.4.4 如何手动向 ReplicaSet 添加容器？

- 不推荐直接修改 ReplicaSet，应该修改 Deployment 定义，并让其自动调整 ReplicaSet。

1.4.5 升级已有应用如何控制更新顺序？

- 滚动升级（默认）：逐步更新 Pod，确保服务不中断。蓝绿部署：同时运行新旧版本，切换流量。金丝雀发布：先升级部分 Pod，观察后再全面更新。

1.4.6 ConfigMap 和本地配置文件的关系？

- ConfigMap 提供动态配置管理，比本地配置文件更灵活，适用于 K8s 原生应用。

1.4.7 K8s 如何实现服务发现？

- 通过 Service 资源，提供负载均衡和内部 DNS 解析：ClusterIP（默认，集群内访问）NodePort（暴露到节点端口）LoadBalancer（云负载均衡）ExternalName（外部服务映射）

2 容器技术概述与核心原理

容器技术（Container Technology）是一种轻量级的虚拟化技术，能够将应用及其依赖封装到独立的容器环境中，使应用能够在各种计算环境之间快速、稳定、一致地运行。本节主要介绍容器的基本概念、设计思想、与传统虚拟机技术的比较，以及容器实现的核心技术。

2.1 容器基本概念

容器是一种基于操作系统内核特性的进程隔离运行环境，其核心思想是实现应用程序级别的虚拟化。容器具有如下特点：

- **轻量级虚拟化**：容器共享宿主机内核，无需安装完整操作系统，资源占用少；
- **环境一致性**：容器内应用程序所需的依赖项打包在一起，实现跨平台一致运行；
- **快速启动与部署**：容器启动速度远高于传统虚拟机，秒级启动，灵活性高；
- **易于迁移和部署**：便于跨平台部署与迁移，实现开发与运维的高效协作。

2.2 容器设计思想

容器技术的设计理念基于以下几点：

1. **资源隔离与封装**：不同应用或进程之间互不干扰，拥有各自独立的运行环境；
2. **一次构建，处处运行**（Build once, run everywhere）：确保在任何环境运行结果都相同；
3. **不可变基础设施（Immutable Infrastructure）**：避免环境漂移，提升系统稳定性；
4. **敏捷部署与弹性扩展**：快速部署、启动、终止容器，灵活地动态扩缩容。

2.3 容器与虚拟机（VM）对比

容器技术与传统虚拟机（如 VMware 或 KVM）的对比

表 1: 容器与虚拟机技术的对比

对比维度	容器（Containers）	虚拟机（VM）
启动速度	秒级（秒或毫秒）	分钟级
资源占用	较低（共享内核）	较高（需完整操作系统）
资源隔离	进程级隔离，共享内核	完全隔离，每个 VM 独立内核
性能开销	极低（进程级开销）	较高（虚拟化整个操作系统）
部署灵活性	非常灵活，易迁移	相对灵活，但镜像较大
安全性	隔离度较低（共享内核）	隔离度较高（独立内核）

2.4 容器核心技术

容器技术依赖于 Linux 内核的多个关键技术实现，主要包括：

2.4.1 进程隔离（Namespaces）

Linux 内核中的 Namespace 技术实现了进程级别的资源隔离。常见的 Namespace 类型包括：

- **PID Namespace**：进程 ID 隔离，容器进程间互不可见；
- **Network Namespace**：提供独立的网络栈、虚拟网络接口；
- **Mount Namespace**：文件系统挂载点隔离，容器拥有独立的文件系统；
- **IPC Namespace**：进程间通信（IPC）隔离；
- **UTS Namespace**：主机名、域名隔离；
- **User Namespace**：用户和用户组隔离。

Namespace 保证容器内部进程和外部环境互不影响，创造独立的运行空间。

2.4.2 资源配额与控制（Control Groups，Cgroups）

Cgroups 是 Linux 内核的一种特性，用于实现容器资源管理，包括：

- 限制进程对 CPU、内存、磁盘 I/O 和网络带宽等资源的使用；
- 确保容器之间的资源隔离，避免资源竞争导致系统性能下降；
- 提高系统稳定性与资源利用效率，实现资源公平分配。

例如，CPU 配额控制公式：

$$CPU\_Quota = CPU_{shares} \times \frac{\text{容器权重}}{\text{所有容器权重之和}}$$

通过合理分配配额，实现资源利用的公平性和高效性。

2.4.3 容器进程管理工具

容器的进程管理工具用于管理容器生命周期，包括容器的创建、运行、监控与终止，常见工具包括：

- **Docker Engine**：提供基础的容器运行、创建、销毁功能；
- **containerd**：符合 OCI 标准的运行时，轻量、高效，Kubernetes 默认支持的容器运行时；
- **CRI-O**：专为 Kubernetes 设计的容器运行时，精简高效，更易于 Kubernetes 集成；
- **runC**：Docker 和 CRI-O 等工具使用的底层运行时，负责实际进程创建与管理。

不同的容器运行时工具各具特点，可根据场景需求灵活选择。

2.5 容器接口标准：OCI 与 CRI

容器接口标准化对容器生态发展至关重要，目前业界最重要的两个容器标准分别为 OCI（Open Container Initiative）和 CRI（Container Runtime Interface）。

2.5.1 开放容器标准（OCI）

OCI（Open Container Initiative）是由 Docker 和其他容器厂商共同制定的开源容器技术标准，旨在定义容器镜像与运行时环境的统一规范。

OCI 标准定义 OCI 标准主要包括两大规范：

- **OCI Runtime Specification（运行时规范）**：
  - 描述容器运行时必须提供的标准接口和环境；
  - 规范了容器的生命周期（创建、启动、停止、删除等）；
  - 定义了容器状态与操作接口（如创建、启动、停止、删除容器的 API 标准）。
- **OCI Image Specification（镜像规范）**：
  - 定义了容器镜像的标准结构；
  - 镜像的元数据（Manifest、配置文件）以及层（Layers）的定义；
  - 镜像存储、分发和管理的统一规范。

OCI 标准的重要意义 OCI 标准的统一使得不同容器运行时工具之间实现了互操作性。例如：

- Docker、Containerd 和 CRI-O 等容器运行时都支持 OCI 标准；
- 镜像的构建与分发实现标准化，使得镜像在不同平台上无缝迁移；
- 容器生态统一，避免厂商锁定，降低用户使用与迁移成本。

OCI 兼容工具 目前主流的 OCI 兼容工具包括：

- **Containerd**（Docker 运行时引擎底层）；
- **CRI-O**（专为 Kubernetes 优化的容器运行时）；
- **Podman**（无守护进程的容器引擎）。

OCI 标准的推广极大促进了容器生态的快速发展，推动了容器技术的统一与生态繁荣。

2.5.2 容器运行时接口（CRI）

CRI（Container Runtime Interface，容器运行时接口）是 Kubernetes 提出的标准接口规范，旨在抽象 Kubernetes 与底层容器运行时之间的交互方式，使 Kubernetes 支持不同容器运行时环境。

CRI 规范的设计目的 CRI (Container Runtime Interface) 旨在实现：

- Kubernetes 与容器运行时之间的标准化交互；
- 隔离 Kubernetes 对特定容器运行时的依赖；
- 允许 Kubernetes 支持多种容器运行时（如 Docker、Containerd、CRI-O）。

CRI 的架构与工作原理 CRI 的整体架构如下：

- Kubernetes 调用标准的 CRI 接口操作容器生命周期；
- CRI 运行时负责将 CRI 调用翻译为容器运行时（如 Containerd、CRI-O）的具体操作。

CRI 标准的实现 主流 CRI 实现包括：

- **Containerd（Docker 的后端运行时）**：
  - CNCF 认证，最流行的 CRI 实现，广泛用于生产环境；
  - 轻量高效、架构灵活；
- **CRI-O（Red Hat 开发）**：
  - 专为 Kubernetes 设计，精简高效，支持 OCI 兼容镜像；
  - 无守护进程，安全性高，广泛用于 OpenShift 平台；
- **Docker shim（已弃用）**：
  - Kubernetes 1.20 前默认使用 Docker shim 接入 Docker；
  - 目前被 Containerd 和 CRI-O 替代。

2.5.3 OCI 与 CRI 对比与关系

表 2: OCI 与 CRI 的对比与关系

特性	OCI 标准	CRI 标准
定义层级	镜像与容器运行时规范	Kubernetes 与容器运行时接口规范
侧重点	统一镜像与容器底层实现	Kubernetes 与容器运行时通信协议
适用范围	通用容器生态，Docker 等工具	Kubernetes 容器编排系统
典型实现	Containerd, CRI-O, runC, Podman	Containerd, CRI-O 等

OCI 与 CRI 的关系说明 OCI 定义了容器运行时与镜像的标准化接口，而 CRI 是 Kubernetes 与这些 OCI 标准容器运行时通信的统一接口，两者并非竞争关系，而是不同层面的标准化接口：

- OCI 专注容器底层技术的标准化；
- CRI 关注 Kubernetes 与容器运行时的对接标准化。

3 Kubernetes 基本概念与集群组件

Kubernetes（简称 K8s）是目前业界最主流的开源容器编排平台，广泛用于自动化容器化应用的部署、扩展和管理。本节详细介绍 Kubernetes 的基本概念、核心组件以及集群运行原理与架构。

3.1 Kubernetes 基本概念

Kubernetes 通过抽象和统一管理容器，提供了可靠的分布式系统管理方案，主要包含以下核心概念：

- **Pod**：Kubernetes 最小的可调度单元，由一个或多个容器组成，共享存储和网络。
- **Deployment**：定义 Pod 的期望状态，控制 Pod 的创建、更新与扩容缩容过程。
- **ReplicaSet**：Deployment 的副本控制器，确保指定数量的 Pod 实例运行。
- **Service**：提供 Pod 的负载均衡、服务发现机制。
- **Namespace**：逻辑隔离多个资源与环境的命名空间，实现资源的隔离管理。

3.2 Kubernetes 集群组件与架构

Kubernetes 集群由控制平面（Control Plane）与节点（Node）两个主要部分构成。控制平面负责管理集群状态与决策，节点则负责运行容器负载。

3.2.1 Master 节点组件

Master 节点是 Kubernetes 的控制平面，包含以下核心组件：

API Server API Server 是集群的统一入口，暴露 RESTful API，负责集群中所有资源的管理与交互：

- 所有请求（kubectl 操作）都经过 API Server；
- 实现认证、授权、访问控制、资源管理等核心功能；
- 与 etcd 存储交互，维护集群状态。

Scheduler（调度器） Scheduler 负责决定新创建的 Pod 应部署到哪个节点：

- 根据节点资源利用情况、Pod 对资源的需求做出决策；
- 调度策略包括负载均衡、资源分配公平性、亲和性（Affinity）等；
- Scheduler 可配置与扩展，适应多样的调度需求。

Controller Manager Controller Manager 负责确保集群实际状态始终与用户声明的期望状态保持一致，核心控制器包括：

- Node Controller：节点故障检测与修复；
- ReplicaSet Controller：确保 Pod 副本数符合定义；
- Deployment Controller：负责滚动升级、扩缩容；
- Service Controller：与云提供商交互，管理负载均衡器。

etcd 存储服务 etcd 是 Kubernetes 的集群状态存储组件：

- 分布式键值存储，提供强一致性、高可用性；
- 存储集群所有状态与配置信息；
- 集群内的各组件均通过 API Server 访问 etcd 获取状态信息。

3.3 Node 节点组件

Node 节点是实际运行容器负载的机器，包括以下核心组件：

Kubelet Kubelet 是 Kubernetes 节点上的主要管理代理，负责节点的 Pod 生命周期管理：

- 接受 API Server 的指令，管理 Pod 和容器；
- 负责容器启动、停止、健康监测及状态上报；
- 与容器运行时（Containerd 或 CRI-O 等）交互，实现容器的实际操作。

Kube-proxy Kube-proxy 是节点上的网络代理，实现 Pod 网络通信与负载均衡：

- 管理 Service 的网络规则，实现集群内外流量转发；
- 支持 IPtables、IPVS 等多种负载均衡模式；
- 保证网络通信的高效性与稳定性。

容器运行时（Container Runtime） 容器运行时负责实际运行容器，如：

- Containerd（最常见，轻量级、兼容 OCI 与 CRI）；
- CRI-O（专为 Kubernetes 优化的运行时）；
- Docker（旧版本 Kubernetes 默认运行时，已弃用 Docker shim）。

容器运行时通过 CRI 接口与 Kubelet 通信，负责容器的创建、运行、停止与删除。

3.4 Kubernetes 组件交互流程

Kubernetes 各组件之间的典型交互流程如下：

- 用户通过 kubectl 命令发送请求到 API Server；
- API Server 验证请求并更新 etcd 中的期望状态；
- Scheduler 从 API Server 获取 Pod 调度请求，进行节点选择并更新 Pod 状态；
- Kubelet 定期访问 API Server 获取节点分配任务，并调用容器运行时（如 Containerd）启动容器；
- Controller Manager 定期对比集群实际状态和期望状态，进行自动修正；
- Pod 启动后，Kube-proxy 根据服务定义设置网络规则，实现服务发现和负载均衡。

3.5 Kubernetes 核心组件对比

表 3: Kubernetes 核心组件作用与特性总结

组件	功能描述
API Server	集群管理的入口，提供 REST API 与集群状态访问
etcd	存储集群状态数据，高可用键值数据库
Scheduler	根据资源情况进行 Pod 调度，决定 Pod 部署节点
Controller Manager	控制器集合，维护实际状态与期望状态一致
Kubelet	管理节点上的 Pod 与容器生命周期
Kube-proxy	提供网络代理，Service 网络规则管理
Container Runtime	容器创建、运行、停止（如 Containerd、CRI-O）

4 Kubernetes 网络原理与通信机制

Kubernetes 的网络模型主要实现了容器内通信、Pod 间通信（同节点、跨节点）以及外部网络对 Pod 的访问。本节对 Kubernetes 网络实现原理与 CNI 插件机制进行详细讲解，并探讨网络故障时的处理方法。

4.1 Kubernetes 网络模型概述

Kubernetes 网络模型有如下基本要求：

- 集群内所有 Pod 间通信无障碍（无论同节点或跨节点）；
- 每个 Pod 拥有独立 IP 地址，且可直接访问；
- 容器之间网络透明，容器无需关心网络实现细节；
- 支持灵活的外部访问机制。

Kubernetes 网络模型基于第三方 CNI 插件实现，如 Flannel、Calico、Cilium 等。

4.2 Pod 内容器通信

Pod 是 Kubernetes 最小的网络单元，同一个 Pod 中多个容器共享网络命名空间（Network Namespace），拥有相同的 IP 地址与端口空间：

- 容器通过 `localhost` 或 `127.0.0.1` 直接通信；
- 端口空间共享，不同容器监听端口不能冲突。

Pod 内通信示意如下：

```
Pod Network Namespace
    Container-1 (127.0.0.1:8080)
    Container-2 (127.0.0.1:9090)
```

容器间使用本地 loopback 地址通信，无需经过任何网络设备，性能极高。

4.3 同节点 Pod 通信原理

同节点 Pod 间通信流程：

- 每个 Pod 都连接到同一个虚拟网桥（如 docker0、cni0）；
- 虚拟网桥维护 Pod 间的转发规则；
- Pod 间通信经虚拟网桥（Bridge）快速转发。

通信流程：

$$\text{Pod}_A \rightarrow \text{veth}_A \rightarrow \text{Bridge} \rightarrow \text{veth}_B \rightarrow \text{Pod}_B$$

这种通信机制简单高效，延迟较低。

4.4 跨节点 Pod 通信原理

跨节点 Pod 通信依靠第三方 CNI 插件实现路由和封装，主要包括以下几种：

（1）Overlay 网络（如 Flannel、Weave） 通过隧道封装通信（如 VXLAN），示意流程：

$$Pod_A \rightarrow Node_A Bridge \rightarrow VXLAN \rightarrow Node_B Bridge \rightarrow Pod_B$$

（2）BGP 路由网络（如 Calico） 利用 BGP 协议直接在网络中路由 Pod IP 地址，通信更直接，无封装开销：

$$Pod_A \rightarrow Node_A \rightarrow BGPRouting \rightarrow Node_B \rightarrow Pod_B$$

4.5 CNI 网络插件原理与作用

Kubernetes 网络接口规范为 CNI（Container Network Interface），作用如下：

- 在 Pod 创建与销毁时，为 Pod 提供和释放网络资源；
- 定义标准接口，允许第三方插件实现网络功能；
- 提供 IP 地址分配、路由管理、网络隔离等功能。

典型 CNI 插件包括：

- **Flannel**：简单易用，基于 Overlay 网络；
- **Calico**：高性能，BGP 路由实现，支持网络策略；
- **Cilium**：eBPF 驱动的高效插件，支持微服务网络安全与观测。

4.6 外部访问 Pod 的方法

Kubernetes 提供多种外部访问 Pod 的方式，包括：

（1）NodePort Service 通过节点 IP 和指定端口暴露服务：

$$\text{External Client} \rightarrow NodeIP : NodePort \rightarrow Service \rightarrow Pod$$

简单易用，适合开发测试。

（2）LoadBalancer Service 云环境中自动创建负载均衡器，转发到 Pod：

$$\text{Client} \rightarrow LoadBalancer \rightarrow Service \rightarrow Pod$$

生产环境广泛采用，但依赖云提供商支持。

（3）Ingress 资源 基于域名和路径进行流量转发：

$$\text{Client} \rightarrow IngressController \rightarrow Service \rightarrow Pod$$

更灵活，更便于维护与管理。

4.7 CNI 插件故障处理

当 CNI 插件或 Pod 网络发生故障时：

- Pod 无法启动，状态为 `ContainerCreating`；
- 节点间 Pod 无法通信；
- 网络延迟、丢包显著增加。

常用故障排查步骤：

1. 检查节点 CNI 插件状态与日志：

```
kubectl describe pod [Pod_Name]
journalctl -u kubelet
```

2. 检查 Pod IP 分配情况：

```
kubectl get pods -o wide
```

3. 检查网络接口状态：

```
ip addr
ip route
```

4. 重启节点网络组件或 Pod：

```
systemctl restart kubelet
kubectl delete pod [Pod_Name]
```

4.8 Kubernetes 网络通信场景总结

表 4: Kubernetes 网络通信场景总结

通信类型	实现方式	推荐技术方案
Pod 内容器通信	共享网络 namespace	默认支持，无需额外配置
节点内 Pod 通信	虚拟网桥通信	Linux Bridge, OVS 等
跨节点 Pod 通信	Overlay 或路由网络	Flannel, Calico, Cilium
外部访问 Pod	NodePort、LB、Ingress	LoadBalancer 或 Ingress 推荐生产使用

5 Kubernetes 存储技术与实现机制

容器运行过程中的数据持久化和共享管理对业务系统十分关键。Kubernetes 提供了灵活且完善的存储管理机制，以保证容器化应用的数据持久化与管理便捷性。本节详细介绍 Kubernetes 存储的核心概念与实现方法。

5.1 Kubernetes 存储基本概念

Kubernetes 存储模型旨在解决容器数据持久化问题，核心概念包括：

- **Volume（卷）**：Pod 级别的持久存储，生命周期随 Pod 存在；
- **PersistentVolume（PV）**：集群管理员配置的持久化存储资源；
- **PersistentVolumeClaim（PVC）**：Pod 请求存储资源的一种声明；
- **StorageClass（存储类）**：描述不同存储类型与特性的配置模板。

5.2 卷（Volume）机制与应用场景

卷（Volume）是 Kubernetes 最基础的存储抽象，特点为：

- Pod 中的容器可以共享同一个 Volume；
- 支持多种存储后端（如本地磁盘、NFS、云存储）；
- Volume 生命周期与 Pod 一致，Pod 删除后卷被释放；
- Persistent Volume 提供长期存储，不依赖于 Pod 生命周期。

5.2.1 常见卷类型

Kubernetes 常见的卷类型包括：

- `emptyDir`：临时存储卷，Pod 生命周期内有效；
- `hostPath`：宿主机目录挂载到容器中；
- `configMap, secret`：配置和秘钥管理；
- `PersistentVolume`（PV）：提供持久化数据存储。

5.3 持久化卷（Persistent Volume, PV）与 PVC

Persistent Volume（PV） PV 是集群管理员预先提供的一种持久化存储资源，生命周期独立于 Pod：

- 提供持久存储能力，可被不同 Pod 使用；
- 独立于特定 Pod 生命周期；
- 可静态或动态创建。

Persistent Volume Claim（PVC） PVC 代表应用对存储资源的需求，由用户创建：

- PVC 定义存储需求，包括大小、访问模式等；
- Kubernetes 负责将 PVC 与合适的 PV 进行绑定。

PV 和 PVC 的交互流程如下：

$$\text{Pod} \rightarrow \text{PVC} \rightarrow \text{PV (由管理员提供或动态创建)}$$

5.4 StorageClass 与动态存储供应

StorageClass（存储类） StorageClass 用于动态创建和管理存储资源的模板：

- 用户申请存储时指定 StorageClass；
- Kubernetes 根据 StorageClass 定义自动创建 PV；
- 允许集群管理员定义不同类型（如 SSD、高性能磁盘、网络存储）的存储资源模板。

StorageClass 示例：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
```

5.5 存储访问模式（Access Modes）

Kubernetes 存储卷支持多种访问模式，主要包括：

- `ReadWriteOnce`（RWO）：可被单个节点读写；
- `ReadOnlyMany`（ROX）：可被多个节点只读访问；
- `ReadWriteMany`（RWX）：可被多个节点同时读写。

不同存储后端支持的访问模式有所不同，如表 5 所示：

表 5: Kubernetes 存储访问模式及适用场景

访问模式	典型应用场景
ReadWriteOnce（RWO）	单 Pod 数据存储，如数据库实例、状态化应用
ReadOnlyMany（ROX）	静态内容（日志、数据分析）被多个节点只读访问
ReadWriteMany（RWX）	多节点同时读写数据，共享存储（如 NFS、CephFS）

5.6 存储流程与实现原理

Kubernetes 存储流程具体如下：

1. 用户定义 PVC 申请存储；
2. Controller Manager 根据 StorageClass 动态创建 PV；
3. PVC 与 PV 自动绑定；
4. Pod 引用 PVC 实现存储卷挂载，容器启动时挂载卷到容器内部；
5. 存储卷被 Pod 使用，数据在容器生命周期结束后仍保持持久化。

5.7 存储故障处理方法

Kubernetes 存储故障常表现为：

- 存储卷无法挂载；
- Pod 状态为 `Pending`；
- 数据读写失败、延迟显著。

排查步骤：

- 检查 PV 与 PVC 状态：

```
kubectl get pv,pvc
```

- 检查节点挂载情况：

```
mount | grep pvc
dmesg | grep mount
```

- 检查存储后端状态（如 NFS、Ceph、云存储状态）；
- 检查相关 Kubernetes 日志（如 kubelet、controller-manager）。