Namespace

|  |
| --- |
| kubectl get ns‘  my-namespace和kube-node-lease是后来创建的      删除命名空间删除该命名空间下的有资源  示例：  apiVersion: v1  kind: ResourceQuota # **ResourceQuota** 是 Namespace 级别的配额，控制 **该 namespace 下所有 Pod/Container 的总资源**。  metadata:  name: mem-cpu-quota32  namespace: my-namespace  spec:  hard:  pods: "10" # 命名空间最多 10 个 Pod  requests.cpu: "2" # 所有 Pod 请求的 CPU 总和 ≤ 2 核  requests.memory: "2Gi" # 所有 Pod 请求的内存总和 ≤ 2Gi  limits.cpu: "4" # 所有 Pod限制的 CPU 总和 ≤ 4 核  limits.memory: "4Gi" # 所有 Pod 限制的内存总和 ≤ 4Gi  ResourceQuota必填项：  spec:  hard:  pods: "5" # 最少要指定至少一个资源指标  如果没有 spec.hard，API server 会报错：spec.hard: Required value  应用：      创建不舍资源限制pod对象：  由于没有default，所以必须指定resources，此时报错：  加上resource：  此时创建pod对象成功：  关于resourcequota和limitrange中定义工作对象：  **🚦 三种情况总结：**  **🟢 1️⃣ 无 LimitRange、无 ResourceQuota**   * ✅ 可以**不写** resources 字段。 * Pod/Deployment **照样能创建**，没有资源限制。 * 节点调度时会根据实际可用资源“尽力分配”。   ➤ 属于最自由但不可控的状态。  **🟡 2️⃣ 有 ResourceQuota 或 LimitRange，但没有 default/defaultRequest**   * ⚠️ 你必须手动写： * resources: * requests: * cpu: ... * memory: ... * limits: * cpu: ... * memory: ... * 否则：   + **ResourceQuota** 无法判断资源占用（报错）；   + **LimitRange** 要求满足 min/max 范围（报错）；   + 最终 **Pod 创建失败**。   ➤ 属于“有限制但没默认值”的严格控制模式。  **🔵 3️⃣ 有 ResourceQuota 或 LimitRange，并定义了 default/defaultRequest**   * ✅ 可以**不写** resources。 * Kubernetes 会**自动补上** default（limit） 和 defaultRequest（request）。 * 自动补完的资源值会在 quota 检查前生效，所以 Pod 能正常创建。   ➤ 属于“有规则但智能自动补全”的最佳实践。  **⚙️ 一句话总结：**  **LimitRange 负责“单 Pod/容器”资源范围与默认值**， **ResourceQuota 负责“命名空间总体”资源上限统计**。  有 default → 系统帮你补； 没 default → 你必须手动写。 |

标签：

|  |
| --- |
| **标签（Label）可以打在所有 K8s 资源上**  **几乎 所有的 Kubernetes 资源对象 都可以打标签。**    **一、创建时打标签**  在创建资源时通过 --labels 参数直接加上：  kubectl run nginx --image=nginx:1.21.5 --labels="env=dev,app=web"  或  kubectl create deployment myapp --image=nginx --labels="tier=frontend,env=prod"  **✅ 二、给已存在的资源添加或修改标签**  kubectl label pod nginx-pod env=dev  如果该标签已存在，命令会报错，可用 --overwrite 参数强制修改：  kubectl label pod nginx-pod env=prod –overwrite    **✅ 三、删除标签**  kubectl label pod nginx-pod env-  **✅ 四、查看标签**  kubectl get pod --show-labels  或筛选某标签：  kubectl get pod -l app=web    指定命名空间下所有pod的标签：    筛选标签名是release的标签：  kubectl -n test get pods -I releaset    筛选标签名+值：      Service通过LabelSelector自动关联具有相同标签的Pod，实现负载均衡    解释：  义了一个 **Service** 服务对象，用于让 Kubernetes 集群内的其他 Pod 能通过一个固定的虚拟 IP 访问一组带特定标签的 Pod  **选择器 (selector)**  spec:  selector:  release: vl # 仅关联带有 release=vl 标签的 Pod   * Service 会根据 selector 去查找 Pod。 * 只有打了 release=vl 标签的 Pod 才会被选中，成为该 Service 的后端目标。 * Service 会自动生成一个虚拟 IP，把请求转发给这些 Pod，这些标签的会自动加入podginx-service加入后端   **端口配置**  ports:  - port: 80  targetPort: 80   * port: 客户端访问 Service 时用的端口（集群内虚拟 IP 的端口）。 * targetPort: 实际 Pod 中容器监听的端口。 * 这里表示：   客户端访问 Service:80 → 转发到 Pod 容器的 80 端口。  **类型 (type)**  type: ClusterIP   * ClusterIP 是默认类型。 * 它会在 **集群内部** 分配一个虚拟 IP（ClusterIP），供集群内其他 Pod 访问。 * 外部用户（集群外）无法直接访问这个 IP。   这里的集群指的是：**Cluster（集群）** 是整个 Kubernetes 系统的运行环境。  然后执行，创建service对象：    验证service对象创建成功：    **Service 只作用于当前命名空间**  当你定义一个 Service，比如：  apiVersion: v1  kind: Service  metadata:  name: nginx-service  namespace: my-namespace  spec:  selector:  app: nginx  ports:  - port: 80  它只会去查找：  属于 my-namespace 命名空间，且标签 app=nginx 的 Pod。  **Service 不会跨命名空间选 Pod**  #Deployment通过Label Selector管理其创建的Pod（如滚动更新、扩缩容）    **Deployment 的作用本质：自动创建并管理 Pod**  Deployment 不是直接运行容器的，它是一个“控制器（Controller）”对象。 它会：   1. 根据你在 .spec.template 中定义的 **Pod 模板**，自动创建 Pod； 2. 持续监控这些 Pod； 3. 如果有 Pod 挂掉，会自动重建； 4. 如果你更新了镜像或配置，它会滚动更新（平滑替换旧 Pod）。   **🧩 二、Deployment 创建 的 Pod 会自动带上标签**  在你的 YAML 里：  selector:  matchLabels:  app: nginx # Deployment 只管理带这个标签的 Pod  template:  metadata:  labels:  app: nginx # 新创建的 Pod 会自动带上这个标签  这表示：   * Deployment 会根据 template（模板）**创建新的 Pod**； * 这些新 Pod 会自动带上标签 app=nginx； * 因此，它们会**自动被 selector 匹配到并纳入管理范围**。   也就是说：  Deployment 不是去找外面已有的 Pod，而是自己创建 Pod，并自动打好标签，让自己能管理。  **🧩 三、那如果 Namespace 里已经有 app=nginx 的 Pod 呢？**  假设你手动创建了一个 Pod：  apiVersion: v1  kind: Pod  metadata:  name: nginx-manual  labels:  app: nginx  然后又创建上面的 Deployment。 会发生什么？👇  ❌ 这个已有 Pod **不会被 Deployment 接管**。  原因是：   * Deployment 的 selector 只是匹配它自己控制的 ReplicaSet 创建的 Pod； * 它不会“收编”集群中其它来源的 Pod； * 换句话说，**Deployment 只管理自己创建的 Pod**。   总结就是：  在 **Deployment** 中，template.metadata.labels **必须** 与 selector.matchLabels 匹配，这样才能确保：   * **创建的 Pod 会自动带上标签**，并且 * **Deployment 能正确管理这些 Pod**。   selector.matchLabels定义标签不会自动使创建pod打上标签，只是匹配作用template.metadata.labels才会打上标签，  应用deployment创建pod：    Node打上标签，pod调度到特定node：    定义node：    效果是： |