**Pod介绍**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **容器、pod、service的ip和端口：**  **容器**  ** 容器本身没有独立的 IP**   * **容器共享 Pod 的网络命名空间** * **通过 容器暴露的端口 来提供服务**   ** 端口：容器内部应用监听的端口，例如 Nginx 默认 80**  ** 访问方式：**   * **Pod 内部：容器之间直接用 localhost:<端口>** * **Pod 外部：需要通过 Pod IP + 容器端口访问**   **Pod**   * **Pod 有自己的专属 IP**   + **Pod 内的所有容器共享这个 IP**   + **Pod IP + 容器端口 = Pod 对外的访问地址** * **关系：**   + **一个 Pod 可以包含多个容器（每个容器有自己的端口）**   + **所以 Pod IP 对应 多个容器端口 → 一对多关系**   **例子：**  **Pod IP: 10.244.1.5**  **├─ 容器 myapp 端口 80**  **└─ 容器 sidecar 端口 9000**  **Pod IP 对应多个端口（多个容器的端口）**  **Service：**   * **Service 有自己的 IP（ClusterIP）和端口** * **作用：把请求路由到匹配的 Pod（通过 selector）** * **关系：**   + **一个 Service IP + 端口 → 多个 Pod IP + 容器端口**   + **Service 的端口可以映射到 Pod 容器端口（targetPort）**   **例子：**  **Service IP: 10.96.0.1 端口 80**  **│**  **└─ selector: app=myapp**  **├─ Pod 10.244.1.5:80**  **├─ Pod 10.244.1.6:80**  **└─ Pod 10.244.1.7:80**  **Service 是一对多关系，把多个 Pod 聚合起来**  **所以外部访问流程是：**  **也就是说pod内，通过端口找container，端口标识容器；pod有一个ip，这个ip标识这个pod；service有一个ip：port，所以这ip：por标识这个service，service关联多个pod；外部访问还有一个nodeport（node节点的物理端口），这个端口关联一个sevice；所以访问某个nodeip:nodeport找到某个service也就是ip：port；然后负载均衡到关联的其中一个pod也就是ip：target port；然后找到pod后用target定位到container**    **Pod Yaml资源文件写法：**    **Pod创建方式**        **以上两种方式创建的pod容易误删，应该使用控制器：**        **一个pod包含两个容器**    **Kubernetes 的标签（labels）是打在 Pod、Service、Deployment、node 等 Kubernetes 对象上的，而不是打在容器（container）上的。**  **创建pod流程：**        **Pod生命周期**      **Init容器**    **command 和 args 是在容器启动时决定执行什么命令的最终组合，而是在镜像启动（运行）时执行**   |  |  | | --- | --- | | **Dockerfile** | **Kubernetes YAML** | | ENTRYPOINT | command | | CMD | args |   **Init container实例：** |

**Main Container Hooks**

|  |
| --- |
| Kubernetes 中的容器在 Pod 内有 **生命周期事件**，可以在特定阶段执行自定义命令或脚本，这就是 **Lifecycle Hooks**：   * **postStart**：容器启动后立即触发 * **preStop**：容器终止前触发   可以理解为类似 **AWK 的 BEGIN 和 END**：   * postStart → 容器“开场前的预设” * preStop → 容器“结束前的清理工作”   hook只能钩在特定的container上, 绑定不分main或附加container，是自己指定的maincontainer，绑定的container的name，一般会被作为maincontainer被编写者使用  **2️⃣ 生命周期钩子执行流程**  Pod 启动流程：  初始化容器(init containers) -> 主容器(main container)  |  +--> postStart Hook 执行  |  +--> 容器正常运行  |  +--> preStop Hook 执行（容器删除前）   * **postStart**：   + 容器刚被创建，尚未完全运行   + 钩子中可以做**资源初始化、环境准备、依赖检查**   + 如果钩子执行失败，容器会被杀死，并按 **restartPolicy** 决定是否重启 * **preStop**：   + 容器在被删除前触发   + 可以做**优雅关闭、清理资源、通知其他系统**   + 钩子执行完后，无论结果如何，Docker/Kubelet 会发送 **SIGTERM** 信号删除容器   **3️⃣ 使用示例（YAML）**  **Poststart实例：**      **prestop示例**      **进入容器**  **kubectl exec -it <pod-name> -c <container-name> -- /bin/sh**  **不进入容器**  **kubectl exec <pod-name> -c <container-name> -- <命令>**  **prestop示例**      **可以看见删除后产生了prestop执行的文件**    **模板：**  apiVersion: v1  kind: Pod  metadata:  name: lifecycle-demo  spec:  containers:  - name: myapp  image: nginx  lifecycle:  postStart:  exec:  command: ["/bin/sh", "-c", "echo 'Container started!'"]  preStop:  exec:  command: ["/bin/sh", "-c", "echo 'Container stopping!' && sleep 5"]   * postStart 打印启动信息 * preStop 打印停止信息并等待 5 秒，模拟优雅关闭   **注意**：   * postStart 不需要参数，执行失败可能导致容器重启 * preStop 需要在删除容器前完成，完成后 Kubelet 会发送 SIGTERM |

**Pod状态**

|  |
| --- |
| **kubectl get pod 的status字段** |

**Pod重启策略**

|  |
| --- |
| **查看当前容器运行时是docker、containerd、cri-o：**    **可以看见是docker，所以kubelet 通过 dockershim 与 Docker 引擎通信 所以你确实可以用 docker ps 查看到 Pod 内的容器！** |

**优雅的删除资源对象**

|  |
| --- |
| **K8s Pod 删除的两种优雅退出方式**  **1️⃣ 默认优雅退出（没有 PreStop）**  **流程：**  **用户删除 Pod**  **│**  **▼**  **Kubelet 通知容器停止（docker stop）**  **│**  **▼**  **容器收到 SIGTERM**  **│**  **▼**  **等待 terminationGracePeriodSeconds（默认 30 秒）**  **│**  **▼**  **如果超时，发送 SIGKILL 强制终止**   * **特点：**   + **给应用程序时间完成正在处理的请求。**   + **默认等待时间是 30 秒，可通过 spec.terminationGracePeriodSeconds 或 kubectl delete --grace-period=N 修改。**   + **如果设置 --grace-period=0，则立即删除 Pod，节点上仍会有一个很短的缓冲时间才 SIGKILL。**   **2️⃣ 使用 PreStop 钩子（终止前回调）**  **流程：**  **用户删除 Pod**  **│**  **▼**  **Kubelet 调用 PreStop 钩子**  **│**  **▼**  **容器收到 SIGTERM**  **│**  **▼**  **等待 terminationGracePeriodSeconds（默认 30 秒）**  **│**  **▼**  **如果超时，发送 SIGKILL 强制终止**   * **特点：**   + **PreStop 钩子在发送 SIGTERM 前执行，可做一些清理或通知操作。**   + **剩下的流程与默认优雅退出相同：等待 → SIGKILL。**   **✅ 总结：**   * **核心就是两种流程：**   1. **默认 SIGTERM → 等待 → SIGKILL**   2. **加 PreStop → 执行 PreStop → SIGTERM → 等待 → SIGKILL** * **terminationGracePeriodSeconds 只是控制等待时间，不是第三种流程。** |

**Node节点选择器**

|  |
| --- |
| **namenode示例：**    **可以看到pod “node-pod-nginx” 分到了k8s-node01**  **nodeselector示例：**    **kubectl get nodes --show-labels #显示node标签**    **效果：** |

**亲和性和污点容忍**

|  |
| --- |
| **Node节点亲和性**  **Node和pod的两种亲和性：**      **硬亲和的一个个参数：**      **参数解释：**        **Matchexpression的参数：**    **--prefered**  **|**  **-- required**  **|**  **--nodeselect terms**  **| —match fields**  **|**  **--match expression**  **|**  **|--key**  **|--operator**  **|--values**  **硬亲和示例：**    **Operator：是in**  **直接执行不行，因为没有zone标签值为foo或bar**    **~~换成软亲和：~~**    **创建node标签：**  **kubectl label nodesk8s-node01 zone=foo**  **此时调度成功：**    **而且是node01上:**    **软亲和示例：**    **没有标签也调度成功了：**      **解释：**  **硬亲和不满足 → Pod Pending，软亲和不起作用；硬亲和满足 → 按软亲和打分选择最优 Node。**  **Pod节点亲和性**    Podaffinity也分软和硬亲和  硬亲和参数：    域：指的是不同的k8s系统    pod硬亲和，示例：  定义两个pod    这两个pod写到一个yaml文件里    第二个节点需要调度到有标签app=myapp2的pod的node上；  第一个节点满足；所以会调度到第一个节点所在的node  最后一个行topologyKey: kubernetes.io/hostname：   **Pod Affinity**：   * 如果设置 topologyKey: kubernetes.io/hostname，表示 Pod 必须调度到 **和目标 Pod 在同一个 Node**。    **Pod Anti-Affinity**：   * 如果设置 topologyKey: kubernetes.io/hostname，表示 Pod 不允许调度到 **和目标 Pod 在同一个 Node**。   **topologyKey 的作用**   * **定义拓扑域（Topology Domain）**，用于指定 Pod Affinity/Anti-Affinity 的范围。 * 当调度器考虑 Pod Affinity/Anti-Affinity 时，它会在 **同一个拓扑域内** 查找符合条件的 Pod。 * 常用的拓扑 Key：   + kubernetes.io/hostname → Node 级别（同一个节点）   + topology.kubernetes.io/zone → Zone 级别（同一个可用区）   + topology.kubernetes.io/region → Region 级别（同一个地域）   #看node标签  kubectl get pods --show-labels # **Pod 的标签（labels）**  **pod节点反亲和性**  pod硬亲和反亲和示例      第一个pod有标签app=myapp3，所以第二个pod不会调度到第一个pod所在的node  **拓扑域**  默认情况下，kubectl label **不允许覆盖已有标签**。  --overwrite （flag）   如果 zone 标签不存在 → 新增标签   如果 zone 标签已存在 → **覆盖原来的值**，改为 foo      matchLabels: app=myapp3 #匹配 已有 Pod，寻找标签为 app=myapp3 的 Pod  这里是antiaffinity+pod+Kubernetes.io/hostname:所以是不和app=myapp3的node在一起；当前 Pod **不能被调度到** 任何一个已经运行着标签为 app=myapp3 的 Pod 的 Node 上 |

**污点和容忍度**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **污点：定义在node上**  **容忍度：定义在pod上**    **污点的完整定义：**  **kubectl taint nodes <节点名> <key>=<value>:<effect>**   |  |  | | --- | --- | | **字段** | **含义** | | **key** | 污点的键（标识污点类型，例如 node-role.kubernetes.io/master） | | **value** | 污点的值（可选，例如 true 或自定义字符串） | | **effect** | 污点效果（**必须项**）： - NoSchedule：不允许新 Pod 调度上来 - PreferNoSchedule：尽量不调度，但不是强制 - NoExecute：不允许新 Pod，上来的旧 Pod 会被驱逐 |       **排斥效果：**    **Toleration（容忍）的完整定义：**    **污点和容忍的匹配？**  **逐个检查该节点的所有 Taint，判断 Pod 的 Tolerations 是否能全部匹配。 只要有一个 Taint 不被任何 Toleration 容忍 → 该节点就 ❌ 不可调度。**   |  |  |  | | --- | --- | --- | | **检查顺序** | **字段** | **匹配逻辑** | | ① | **key** | 必须匹配。除非 Toleration 的 key 为空（表示通配全部 key）。 | | ② | **operator** | 决定是否检查 value：- Equal → 必须同时匹配 value。- Exists → 不检查 value，只要 key 存在即可。 | | ③ | **value** | 仅当 operator=Equal 时才要求相同。 | | ④ | **effect** | 必须匹配相同的 effect（NoSchedule、NoExecute、PreferNoSchedule）。如果 Toleration 没写 effect → 表示通配所有 effect。 |  |  |  | | --- | --- | | **条件** | **是否能匹配** | | key、value、effect 全都对应上 | ✅ 匹配 | | operator=Exists 且 key、effect 对应上 | ✅ 匹配 | | operator=Equal 但 value 不同 | ❌ 不匹配 | | key 不存在于 Toleration | ❌ 不匹配 | | effect 不同 | ❌ 不匹配 |   **如何看node的污点定义：**    **这里的node-role.kubernetes.io/master:NoSchedule**  **没有写 =value，所以只包含：**   |  |  |  | | --- | --- | --- | | **字段** | **值** | **说明** | | **key** | Node-role.kubernetes.io/master | 污点的键，表示节点的角色类型 | | **value** | *（省略）* | 没有设置 value（为空） | | **effect** | NoSchedule | 表示不允许没有容忍的 Pod 调度上来 |   **默认情况下，Kubernetes 的 master 节点会被打上一个 污点（taint），防止普通 Pod 调度上去：**  **kubectl describe node k8s-master | grep Taints**  **通常会看到类似：**  **Taints: node-role.kubernetes.io/master:NoSchedule**  **为什么kube-apiserver-k8s-maste可以调度到master？**    **kubectl describe pods kube-apiserver-k8s-master -n kube-system | tail -3**  **查看 kube-system 命名空间中 kube-apiserver-k8s-master 这个 Pod 的详细信息，然后只显示输出的最后三行内容。**  **Tolerations：Pod 的容忍设置（例如 NoExecute、NoSchedule）**  **Tolerations::NoExecute op=Exists为什么匹配**  **Taints: node-role.kubernetes.io/master:NoSchedule？**  **这个 Tolerations: NoExecute op=Exists 看起来 没有 key，那它是怎么容忍 master 上的污点的呢？**  **我们来一步一步解释清楚👇**  **🧩 1️⃣ 实际情况：系统组件的 Pod 有“通用容忍”**  **Kubernetes 自带的系统组件（例如 kube-apiserver、kube-controller-manager、kube-scheduler 等）， 在生成 Pod 时（由静态 Pod 清单 /etc/kubernetes/manifests/\*.yaml 创建）， 通常都会自动带上一条 “通配符容忍”，类似：**  **tolerations:**  **- operator: "Exists"**  **或者在 describe 里看到：**  **Tolerations: NoExecute op=Exists**  **🧩 2️⃣ operator: Exists 且 没有 key 的含义**  **没有 key + operator: Exists = 匹配任意污点的任意 key。**  **也就是说：**   * **它不管污点的 key 是什么；** * **也不管污点有没有 value；** * **只要节点上有任何带 NoExecute（或对应 effect）的污点，这个 Pod 都可以容忍。**   **换句话说，它是一个 全局容忍（wildcard toleration）。**  **统组件的 Pod 实际上通常会同时拥有多条容忍，比如（在完整 YAML 中可以看到）：**  **tolerations:**  **- operator: "Exists"**  **effect: "NoSchedule"**  **- operator: "Exists"**  **effect: "NoExecute"**  **describe 命令在结尾只打印最后一条时，可能被你看到的是 NoExecute op=Exists，但实际上它还有一条对应 NoSchedule 的容忍。**  **如果你执行：**  **kubectl get pod kube-apiserver-k8s-master -n kube-system -o yaml | grep -A5 tolerations**  **你就能看到它其实有多条容忍规则（NoSchedule、NoExecute 都在），只是 describe 的 tail -3 只显示了其中一条。**  **完全正确，你理解得很精准 👍**   * **系统组件 Pod（kube-apiserver、controller-manager、scheduler 等）：为了保证能在 master 节点运行，通常在 YAML 中会定义 通配 key 的容忍（operator: Exists），并覆盖 NoSchedule 和 NoExecute 污点。**   + **这样不管 master 节点上有哪种污点，这些核心组件都能正常调度。** * **普通 Pod：一般不会定义这么宽泛的容忍度，因为：**   + **没必要占用 master 节点资源；**   + **容忍过多污点可能导致调度到不期望的节点，破坏资源隔离策略。**   **所以，只有系统组件默认会使用“全局容忍”策略，普通业务 Pod 通常只针对特定污点或完全不设置容忍。**  **示例**  **kubectl taint node k8s-node02 node-type=production:NoSchedule #阻止新的，运行的不影响**      **添加污点最后+减号就是删除污点**    **去掉污点noschedule：pod可以添加到了node02**  **此时再加上noschedule污点：pod没被剔除**    **删除node01的NoExecute污点：**  **Pod跑到node01**  **此时再加上NoExecute污点：pod被删除了**    **配置容忍度（yaml）**      **谁也不匹配**  **修改yaml：**    **匹配node02**  **修改operator为：exist**    **此时都key匹配，但是noschedule匹配node02**  **修改tolerations：**  **此时全匹配**  **删除污点两种方式：**    **其他案例：** |

**探针**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **三种探针（livenessProbe、readinessProbe、startupProbe）都是针对容器的健康检查，而不是pod健康检查，Pod 健康状态（READY）由容器状态综合决定**      **三者对比表**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **探针** | **作用** | **失败后操作** | **触发对象** | **典型用途** | | **livenessProbe** | **判断容器是否存活** | **重启容器** | **容器** | **容器假死、死锁** | | **readinessProbe** | **判断容器是否就绪** | **移除 Service Endpoints，不重启** | **Pod 在 Service 中** | **延迟接流量，初始化依赖** | | **startupProbe** | **判断容器是否启动成功** | **重启容器** | **容器** | **启动慢容器，避免 livenessProbe 误杀** |   **核心理解**   1. **startupProbe 优先级最高**    * **如果配置了 startupProbe → livenessProbe 在 startupProbe 成功前不会生效** 2. **readinessProbe 只影响流量，不重启** 3. **livenessProbe 判断存活，失败会重启**   **示例：**      **livenessProbe 的失败策略默认是 Restart（也就是重启容器），所以 Kubernetes 会 每次探针失败就重启容器一次。**    **通过http方式探测：**       * **容器的网络命名空间发起请求，相当于在容器内部执行：**   **curl http://localhost:8081/actuator/health**   * **因此 /actuator/health 必须是容器内的应用暴露的 HTTP 接口**   **修改一下：**    **此时：**    **TCP livenessProbe**    实际使用：      **一直成功，改了端口就不成功了：**      **Readinessprobe**    **Startuprobe** |

**资源限制**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **LimitRange 是 命名空间级别对象，必须加 metadata.namespace。它只在指定命名空间生效，对其他命名空间无效。**  **作用级别：**   |  |  |  |  | | --- | --- | --- | --- | | **层级** | **type 值** | **作用对象** | **配置字段示例** | | **Pod 级别** | Pod | 限制 **整个 Pod 的总资源**（所有容器加起来） | min, max, maxLimitRequestRatio | | **Container 级别** | Container | 限制 **单个容器的资源** | min, max, default, defaultRequest, maxLimitRequestRatio |   **各字段：**   |  |  |  |  | | --- | --- | --- | --- | | **字段** | **Pod/Container** | **作用** | **说明** | | min | Pod + Container | 最小可申请 | 约束边界 | | max | Pod + Container | 最大可申请 | 约束边界 | | maxLimitRequestRatio | Container | limit/request 最大比例 | 仅 container 有 | | default | Container | 默认 limits | 仅 container 有 | | defaultRequest | Container | 默认 requests | 仅 container 有 | | requests | Container | 实际请求 | Pod YAML 中写 | | limits | Container | 实际上限 | Pod YAML 中写 |   **资源单位**  **CPU**  **特点：是一种可压缩资源，cpu资源是支持抢占的。**  **单位：CPU的资源单位是CPU(Core)的数量，是一个绝对值。**  **大小：因为CPU配额对于绝大多数的容器来说，实在是太多了，所以在Kubernetes 中通常以千分之一的CPU为最小单位，用m表示。**  **一般来说一个容器占用的CPU是100~300m，即0.1-0.3个CPU**  **注意：**  **mi代表是1024进制的**  **内存**  **特点：是不可压缩资源，当pod资源扩展的时候，如果node上资源不够，那么就会发生资源抢占，或者OOM问题+**  **单位：内存的资源以字节数为单位，是一个绝对值**  **大小：内存配额对于绝大多数容器来说很重要，在Kubernetes 中通常以 Mi单位来分配。**    **🧠CPU 是“可压缩资源”**  **✅ 含义**  **“可压缩（Compressible）” 的意思是：当资源紧张时，可以 暂时减少分配、被抢占、或降低性能，但容器不会立刻崩溃。**  **🧱 内存 是“不可压缩资源”**  **❌ 含义**  **“不可压缩（Non-compressible）” 的意思是：内存不能被抢占或减少，一旦不足，系统只能终止进程。**  **1 CPU = 1 Core（核）**   * **在 Kubernetes 中，CPU 是以核为单位的绝对值。** * **不过，K8s 允许用更小的“毫核”（millicore）来表示部分 CPU。**  |  |  |  | | --- | --- | --- | | **表示方式** | **含义** | **等价关系** | | 1 | 1 核 CPU | 1000m | | 500m | 0.5 核 CPU | 半个核心 | | 100m | 0.1 核 CPU | 1/10 个核心 | | 250m | 0.25 核 CPU | 四分之一个核心 |   **m 代表 *millicore*（千分之一核）。**  **内存的单位**  **✅ 基本单位：字节（Byte）**  **但为了书写方便，Kubernetes 支持多种进制单位：**   | **单位** | **含义** | **换算（2 的幂次，二进制）** | | --- | --- | --- | | **Ki** | **Kibibyte** | **1 Ki = 1024 B** | | **Mi** | **Mebibyte** | **1 Mi = 1024 Ki = 1,048,576 B** | | **Gi** | **Gibibyte** | **1 Gi = 1024 Mi = 1,073,741,824 B** |   **💬 注意：**   * **这里的 Mi、Gi 是 二进制单位，和我们常说的 MB、GB（十进制）不同。** * **Mi 是 Kubernetes 推荐的写法，因为更精确。** * **MB 也能用，但表示 1 MB = 1,000,000 字节（十进制）。**   **对比：**   |  |  |  | | --- | --- | --- | | **项目** | **CPU** | **内存** | | 本质 | 时间片资源（可共享） | 存储资源（独占） | | 单位 | 核（Core） | 字节（Byte） | | 最小单位 | 1m = 0.001 核 | 1Ki = 1024 字节 | | 示例 | 250m = 0.25 核 | 512Mi ≈ 0.5GB | | 可压缩性 | ✅ 可压缩 | ❌ 不可压缩 | | 超限后果 | 降速（throttle） | OOMKilled |   **模板：**  **apiVersion: v1**  **kind: LimitRange**  **metadata:**  **name: limit-range-demo**  **namespace: my-namespace**  **spec:**  **limits:**  **# Pod 级别限制**  **- type: Pod**  **min:**  **cpu: 200m**  **memory: 128Mi**  **max:**  **cpu: "4"**  **memory: 4Gi**  **# Container 级别限制**  **- type: Container**  **min:**  **cpu: 100m**  **memory: 64Mi**  **max:**  **cpu: "2"**  **memory: 2Gi**  **defaultRequest:**  **cpu: 200m**  **memory: 128Mi**  **default:**  **cpu: 500m**  **memory: 256Mi**  **maxLimitRequestRatio:**  **cpu: 5**  **memory: 4**  ** Pod 级别的 LimitRange 没有 default / defaultRequest 字段（这些字段只对 type: Container 有效）。**  ** 也就是说： 如果 Pod 内的容器没有写 requests 或 limits，Kubernetes 不会自动填充默认值。**    **requests容器启动必须满足的资源，否则允许不起来**  **limits容器启动允许使用的最大资源，否则被重启删除**  **示例：**  **Kubectl create ns my-namespace #创建namespace**  **#创建limitrange资源对象**    **你没写的字段就不会生效，也就是没有约束，这里只有下限request，没有上限limit，所以容器的 limits 不会自动填充，也没有任何上限约束。**  **创建pod.yaml**    **创建资源对象和pod，效果：**    **看详细创建的pod信息：**      综合限制案例：  **官方规范**   * **Pod 类型 (type: Pod) 的 LimitRange 只支持：**   + **min**   + **max** * **不支持：**   + **default / defaultRequest / maxLimitRequestRatio**   + **也就是说，Pod 类型 不能定义 ratio**   **官方文档明确写明：**   * **default, defaultRequest, maxLimitRequestRatio 只能用于 Container 类型。**     ** 1. 没有 LimitRange 的情况下，Pod 和 Deployment 可以创建时不设资源约束。**  ** 2.一旦有 LimitRange，所有资源对象都必须遵守 LimitRange 的规定，否则创建失败。**  ** 3.必须定义 limit 或 default，以及 request 或 defaultRequest，而 max 和 min 是可选项，用于对资源进行更精细的控制。**    **Container：**  **Cpu**  **100m<200m <资源<300<2**  **内存**  **3Mi<100Mi<资源<200Mi<1Gi**  **Pod**  **Cup**  **200m<总资源<4**  **内存**  **6Mi<总资源<2Gi**  **创建资源对象：**    **验证对象存在：**    **查看资源限制策略：**    **创建pod验证：**  **这里写错了，memory request不应该大于memory limit**   * **LimitRange 是 namespace 级别的资源约束。** * **当你在某个 namespace（例如 my-namespace）下创建 Pod 或 Deployment 时，Kubernetes 会 检查 Pod/container 的 requests 和 limits 是否符合该 namespace 下的 LimitRange 规则。** * **如果不符合（比如超过 max，低于 min，或者 request / limit 超出默认范围），Pod 会 创建失败，报错信息类似：**   **Error from server (Forbidden): pod "nginx-test" is forbidden: exceeded quota: cpu limit exceeds the max limit of LimitRange** |