文档中心

容器云 UK8S

产品文档

目录

产品概念

UCloud Container Service for Kubernetes（UK8S）是一项基于Kubernetes的容器管理服务，你可以在UK8S上部署、管理、扩展你的容器化应用，而无需关心Kubernetes集群自身的搭建及维护等运维类工作。

UK8S完全兼容原生的Kubernetes API，以UCloud私有网络为基础，并整合了ULB、UDisk、EIP、VPC等云产品。

使用须知

1. 集群

1. 使用 UK8S 服务，必须通过 UCloud 实名认证服务；
2. 集群中 master 节点为管理节点，不建议服务调度到 master 节点运行；
3. 单个 UK8S 集群最多添加 5000 个 Node 节点，生产集群建议不超过 1000 个；
4. 支持的 Kubernetes 版本会落后社区 2 个版本，具体以产品页面为准；
5. 如果云账户开启了 API 白名单，请在控制台**账号安全**页面**访问限制**栏中，将 10.10.10.10/32 网段加入允许 API 访问的列表。

2. 节点

1. Node 节点使用的云主机镜像默认为 Centos7.6，如果重装系统或擅自修改系统配置，可能导致集群不可用；
2. Node 节点使用的云主机系统盘默认为 SSD，支持添加数据盘，添加数据盘后，容器启动后默认运行在数据盘；
3. 云主机默认使用 N 机型，您也可以选择其他机型（如 O 型、C 型）；
4. 节点标签**UhostID=xxx**为主机管理使用标签，请勿删除修改，以免导致集群数据不正常；

3. 存储卷

支持类型

当前支持 SATA、SSD UDisk 以及 UFS 和 UFile；

在 UK8S 中使用 UDisk

在 UK8S 中使用 UFS

在 UK8S 中使用 UFile

支持地域

请参考存储相关文档说明

4. 资源命名规范

当你创建 UK8S 集群后，UK8S 会以你的名义在当前项目下创建 UHost、ULB、UDisk、EIP 等资源，更改配置或删除可能导致集群不可用，请谨慎操作。涉及的资源主要如下：

1. 由 UK8S 管理服务创建的资源，其命名规范如下：

- `uk8s-xxxxxxxx-master-m` 为 Master 节点的名称，作为集群的 Master 节点；
- `uk8s-xxxxxxxx-n-xxxxx` 为 Node 节点的名称，作为集群的 Node 节点；
- `uk8s-xxxxxxxx-master-ulb4` 为集群 ApiServer 内网 ULB 名次，作为集群内部入口；
- `uk8s-xxxxxxxx-master-ulb4-external` 为 ApiServer 外网 ULB 名称，作为集群外部入口；
- `system_udisk_uk8s-xxxxxxxx-n-xxxxx` 为 UDisk 资源名称，作为集群节点的系统盘；
- `data_udisk_uk8s-xxxxxxxx-n-xxxxx` 为 UDisk 资源名称，作为集群节点的数据盘。

2. 由 UK8S 插件创建的资源，其命名规范如下：

- `ingress-nginx.ingress.svc.uk8s-xxxx` 是 ULB 资源名称，由 UK8S 的 ULB 插件创建，用于 LoadBalancer 类型的Service，且其备注为 UID-xx-xxx，实为 Service 在 UK8S

中的 uuid。其命名规范为 svc-name.namespace.svc.uk8s-id。

- TCP_443_xxx-xxxxx 为 Vserver 名称，由 UK8S 的 ULB 插件创建，对应 LoadBalancer 类型的 Service 中不同的端口。其命名规范为 service-protocol_service-port_service_uuid。
- pvc-9393f66f-c0b5-11e9-bd6d-5254001935f2 为 UDisk 名称，由 UK8S 的存储插件创建，对应 UK8S 中的 PVC。其命名规范为 pvc-pvc_uuid。

产品价格

专有版 UK8S 不额外收费，您只需要为使用 UK8S 过程中用到的其他云产品付费。在使用 UK8S 的过程中，您可能会使用到以下产品，他们相关价格具体如下：

产品	用途	收费说明
云主机	集群 Node 节点及专有版 Master 节点	UHost收费说明
云硬盘	节点数据盘及集群所用 PV 存储卷	UDisk收费说明
文件存储	集群所用 PV 存储卷	UFS收费说明
对象存储	集群所用 PV 存储卷	UFile收费说明
负载均衡	外网 APIServer 服务，集群中需要外网暴露的 LoadBalancer 类型服务	CLB收费说明 ，ALB收费说明
弹性IP	外网 ULB 所绑定的 EIP	EIP收费说明

备注：如无特别说明，UK8S 创建以上产品时的付费模式默认为按月购买，您也可以自行指定购买方式（年/月/时）。

其他

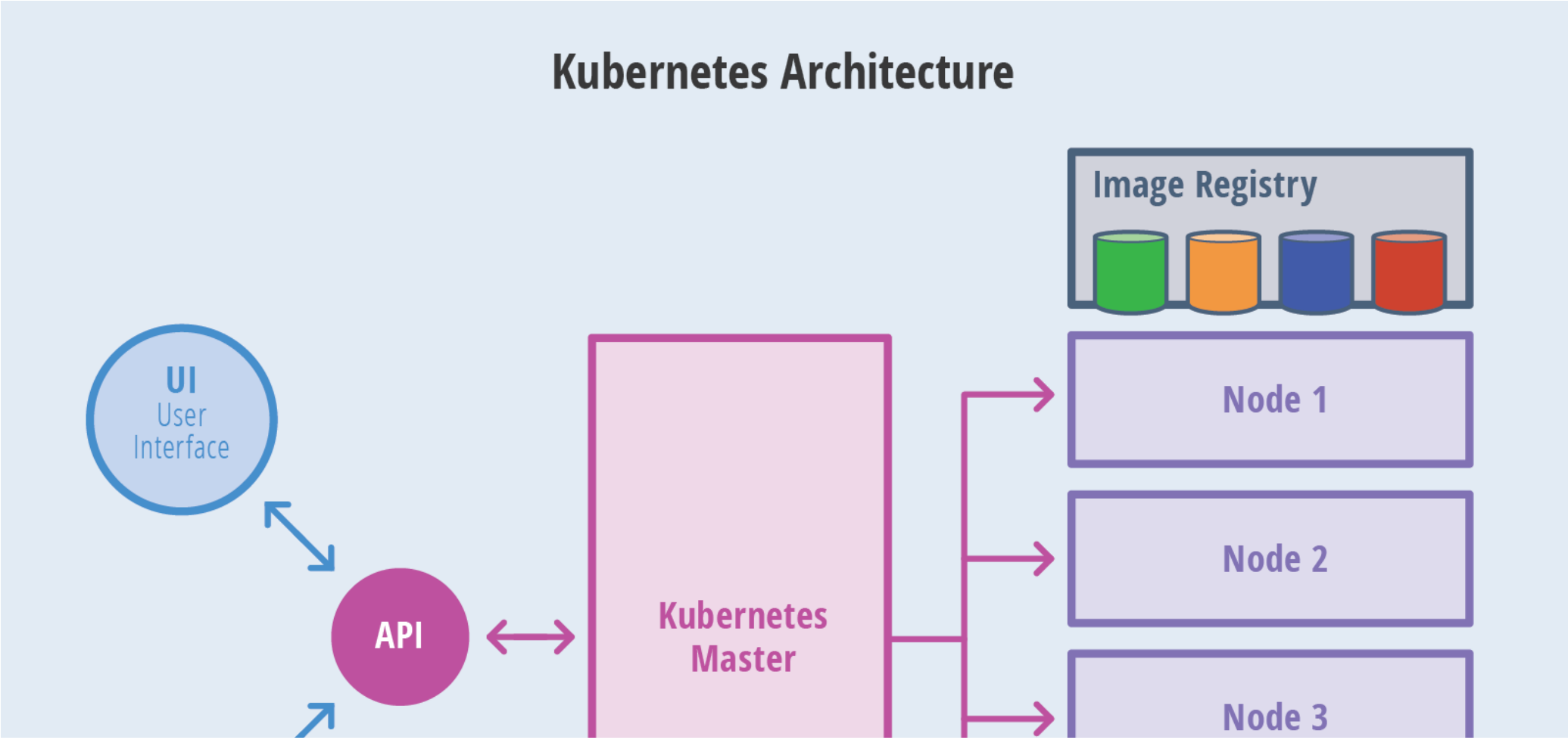
UK8S 托管版集群以集群为计费单元，支持按年、月、小时预付费，不同地域单价如下：

地域	月付单价（元/月）	时付单价（元/小时）
北京/上海/广州	800	1.67

名词解释

为了帮助你更好地使用UK8S，你需要对Kubernetes的相关组件有一个大致的了解，本文会对基础组件及概念做一个简要的介绍，如果你希望了解更多内容，请移步Kubernetes官方文档

Kubernetes基础架构



备注:图片源自Kubernetes社区

上图是一个概要性的Kubernetes架构,包含了ApiServer,Master、Node、Hub(镜像仓库)等概念,下面我们依次做个简要介绍。

ApiServer: ApiServer是操作集群的唯一入口,并提供认证、授权、访问控制、API注册和发现等机制,ApiServer作为一个组件运行在Master上。

Node: Node是Kubernetes的工作节点,其上包含运行Pod所需要的服务。Node可以是虚拟机也可以是物理机,在UK8S,目前只支持虚拟机即UHost。

Master: Master也是Kubernetes的工作节点,与Node不同的是,Master通常不运行业务Pod,而是安装了用于控制和管理集群的如ApiServer、Scheduler、Controller Manager、Cloud Controller Manager、ETCD等组件。

Hub镜像仓库 Hub提供Docker镜像的管理、存储、分发能力。

使用必读

注意:通过 UK8S 创建的云主机、云盘、EIP 等资源,删除资源请不要通过具体的产品列表页删除,否则可能导致 UK8S 运行不正常或数据丢失风险,可以通过 UK8S 将资源释放或解绑删除。

UK8S为容器应用提供编排、调度的能力,应用本身是运行在 UCloud 的 IAAS 类产品上,如 UHost、UDisk、EIP、ULB等,为了帮助你更好地使用 UK8S,你需要知道并同意:

1. 你需要授权给 UK8S 部分产品的管理权限,以便你在操作集群时可以利用到 UCloud 的其他产品,如创建一个 PVC 时,集群帮你创建一块 UDisk;
2. UK8S产品本身不收取服务费用,但通过集群创建的诸如 EIP、UDisk等资源,会产生相应的费用,详见UK8S价格;
3. 组成 UK8S 集群的基础设施,如 UHost、UDisk 等,你可以在集群管理页面查看到,请勿随意在其他产品列表页更改或删除;
4. UK8S产品版本会紧跟云原生kubernetes版本,UK8S默认支持3个最新版本(见页面推荐),针对老版本建议用户升级,若不升级可能会有部分功能无法使用。

请勿随意操作由UK8S创建的资源

UK8S 会以你的名义创建一批云资源,例如 UHost、UDisk 等,这些云资源是 UK8S 集群所依赖的基础设施,一旦更改,可能对集群操作无法预估的影响,请勿随意改动。

如何识别由UK8S创建的云资源?

由UK8S创建的云资源名称,都遵循明确的命名规范,具体详见命名规范,简要说明如下:

1. UHost名称:[cluster-id]-[nodetype]-[randomcode],如名称为uk8s-uxl1l3l0-n-4rd91的云主机,是"uk8s-uxl1l3l0"这个UK8S集群的Node节点。

2. ULB名称:[cluster-id]-[master]-ulb,如名称为uk8s-uxl1l3l0-master-4rd91的ULB,是内网ApiServer的入口,切勿删除。

请尽量规避将业务部署在Master节点

UK8S集群中的3台Master节点预先部署了k8s的管理服务,节点已设置为SchedulingDisabled,不建议将自己的业务调度到Master节点。

入门必读

Kubernetes 提供了一系列的命令行工具来辅助我们调试和定位问题，本指南列举一些常见的命令来帮助应用管理者快速定位和解决问题。

1. 定位问题

在开始处理问题之前，我们需要确认问题的类型，是 Pod ，Service ，或者 Controller（Deployment、StatefulSet） 的问题，然后分别使用不同的命令来查看故障原因。

2. Pod 常见命令

当我们发现 Pod 处于 Pending 状态，或者反复 crash，无法接受流量，可以使用以下命令来快速定位问题：

1. 获取 Pod 状态

```
kubectl -n ${NAMESPACE} get pod -o wide
```

2. 查看 Pod 的 yaml 配置

```
kubectl -n ${NAMESPACE} get pod ${POD_NAME} -o yaml
```

3. 查看 Pod 事件

```
kubectl -n ${NAMESPACE} describe pod ${POD_NAME}
```

4. 查看 Pod 日志

```
kubectl -n ${NAMESPACE} logs ${POD_NAME} ${CONTAINER_NAME}
```

5. 登录 Pod

```
kubectl -n ${NAMESPACE} exec -it ${POD_NAME} /bin/bash
```

3. Controller 常见命令

控制器负责 Pod 的生命周期管理，一般 Pod 无法被注册时，可以通过 Controller 来查看原因。这里以 Deployment 为例，介绍 Kubernetes Controller 的常用命令，其他 Controller 的命令类型与其一致。

1. 查看 Deployment 状态

```
kubectl -n ${NAMESPACE} get deploy -o wide
```

2. 查看 Deployment yaml 配置

```
kubectl -n ${NAMESPACE} get deploy ${DEPLOYMENT_NAME} -o yaml
```

3. 查看 Deployment 事件

```
kubectl -n ${NAMESPACE} describe deployment ${DEPLOYMENT_NAME}
```

4. Service 常见命令

Service 描述了一组 Pod 的访问方式，当我们发现应用无法访问时，则需要使用 Service 命令来查看故障原因。

1. 查看 Service 状态

```
kubectl -n ${NAMESPACE} get svc -o wide
```

我们可以通过上述命令查看到 Service 的类型、集群内部和外部IP、暴露的端口，以及 Selector 信息。

2. 查看 Service 事件及负载均衡信息

```
kubectl -n ${NAMESPACE} describe svc ${SERVICE_NAME}

Name: example-app
Namespace: default
Labels: app=example-app
Annotations: <none>
Selector: app=example-app
Type: ClusterIP
IP: 10.2.192.27
Port: web 8080/TCP
TargetPort: 8080/TCP
```

```
Endpoints: 192.168.59.207:8080,192.168.75.87:8080,192.168.84.90:8080
Session Affinity: None
Events: <none>
```

如上所示，我们可以通过这个命令查看到 Service 的 Endpoints 信息，Endpoints信息如果为空，则说明 Service 的配置信息有误，Service 无法将流量转发到相应的 Pod. 另外还有 Port 及 TargetPort 信息，确保与业务实际暴露的端口一致。

集群版本维护说明

UK8S是基于原生的Kubernetes提供的容器管理服务。该产品支持的集群版本会参考Kubernetes社区新版本发布节奏发布更新。以下为UK8S服务的Kubernetes版本支持机制：

版本发布

版本号	状态	发布时间	停止维护时间
1.28.15	支持	2025.01.03	-
1.26.7	支持	2023.10.24	-
1.24.12	支持	2023.05.08	-
1.22.5	停止维护	2022.01.20	2025.01.03
1.20.6及以下	停止维护	-	2023.10.25

版本支持

从2023年10月起，UK8S仅发布支持Kubernetes偶数号的三个大版本，平台支持策略如下：

集群创建

您可以创建UK8S控制台支持的三个Kubernetes双数大版本的集群。比如，控制台支持版本为 1.20.6、1.22.5和1.24.12三个时，当UK8S发布支持1.26.7后，1.20.6版本停止维护，您将无法创建该版本。

集群升级

版本升级仅支持相邻大版本的升级，不支持跨版本升级，不支持回退版本。

技术支持

UK8S控制台支持的三个版本，提供相应技术支持，如答疑、在线指导、排查问题等；过期版本的Kubernetes集群存在一定风险性，且过期版本不保证技术支持时效性，建议您及时升级Kubernetes版本。

对于过期版本，UK8S仅提供重大缺陷修复(影响集群的维护及删除，影响节点或Pod资源的创建、删除、维护，及重大漏洞等)，不提供功能缺陷修复，不提供新特性。

插件升级

CloudProvider、CSI、CNI等插件，仅在上述版本的集群中提供插件升级功能。如果您需要使用最新的插件特性，请及时升级集群版本。参考CloudProvider插件更新、CSI存储插件升级、CNI网络插件升级。

镜像版本

依据CentOS官方公告所知，其将停止维护CentOS Linux项目，UCloud所提供的基础镜像CentOS Linux源于CentOS官方，故在官方停止维护后UCloud也将停止对该基础镜像的维护。详细请参考CentOS Linux停止维护后的应对方案

Kubernetes 1.28版本重要变更说明

特别说明

- 因uk8s中cni使用的kubeconfig做了不兼容修改，cni版本不能低于1.3.4

调度逻辑优化

如果使用了调度器插件可能需要调整，参阅调度器框架变化

CephFS废弃

CephFS卷得kubernetes.io/rbd从树内移除，请使用csi接口的插件ceph-csi

SecurityContextDeny废弃

被PodSecurity取代而被废弃，具体参阅官方issues

Seccomp注解移除

Seccomp（安全计算模式）在1.19进入GA，支持通过限制Pod或容器的系统调度来提高安全性。对Alpha阶段的seccomp.security.alpha.kubernetes.io/pod和container.seccomp.security.alpha.kubernetes.io注解自v1.19起被弃用，1.27完全移除。

建议使用Pod或容器securityContext.seccompProfile字段

sidecar原生支持

该特性在1.28中为Alpha阶段，需要打开特性门控

它为init容器引入了restartPolicy字段，并使用这个字段来指示init容器是sidecar容器。当restartPolicy=Always时Kubelet将按照的顺序与其他init容器一起启动init容器，但它不会等待其完成，而是等待容器启动完成。

更多详情请参阅官方说明

AdmissionWebhookMatchCondition默认支持cel表达式

该特性在1.28中为beta阶段

它允许使用cel表达式来接受或拒绝apiserver的请求，是webhook的一种替代方案，具体请参阅cel

api弃用

CSIStorageCapacity

CSIStorageCapacity升级为storage.k8s.io/v1原先的storage.k8s.io/v1beta1在1.27中被废弃

参考链接

更多完整的变更记录请参阅 CHANGELOG 1.27，CHANGELOG 1.28

Kubernetes 1.26版本说明

从Kubernetes 1.25开始到1.26，官方废弃了很多api版本，建议您在升级之前，仔细检查集群中的API版本以及Master组件的配置，下面的清单供您参考。

api弃用

Cronjob

在Kubernetes 1.25版本后，开始不再提供batch/v1beta1 API版本的CronJob（定时任务），但可以使用batch/v1 API版本，此API从Kubernetes 1.21版本开始可用。

EndpointSlice

在Kubernetes 1.25版本后，开始不再提供discovery.k8s.io/v1beta1 API版本的EndpointSlice（端点切片），但可以使用discovery.k8s.io/v1 API版本，此API从Kubernetes 1.21版本开始可用。

Event

在Kubernetes 1.25版本后，开始不再提供events.k8s.io/v1beta1 API版本的Event，但可以使用events.k8s.io/v1 API版本，此API从Kubernetes v1.19版本开始可用。

PodDisruptionBudget

在Kubernetes 1.25版本后，开始不再提供policy/v1beta1 API版本的PodDisruptionBudget（Pod中断预算），但可以使用policy/v1 API版本，此API从Kubernetes 1.21版本开始可用。

policy/v1中值得注意的变更是：在policy/v1版本的PodDisruptionBudget中将spec.selector设置为空（{}）时，会选择名字空间中的所有 Pods，即在policy/v1beta1版本中，空的spec.selector不会选择任何Pods。如果spec.selector未设置，则在两个API版本下都不会选择任何 Pods。

PodSecurityPolicy

在Kubernetes 1.25版本后，开始不再提供policy/v1beta1 API版本中的PodSecurityPolicy（Pod安全策略），并且PodSecurityPolicy准入控制器也会被删除，请将PodSecurityPolicy迁移到Pod Security Admission或第三方准入Webhook。

有关迁移指南的更多信息，请参见从PodSecurityPolicy迁移到内置PodSecurity准入控制器。有关弃用的更多信息，请参见PodSecurityPolicy弃用：过去、现在和未来。

RuntimeClass

在Kubernetes 1.25版本后，开始不再提供node.k8s.io/v1beta1 API版本中的RuntimeClass（运行时类），但可以使用node.k8s.io/v1 API版本，此API从Kubernetes 1.20版本开始可用。

HorizontalPodAutoscaler

在Kubernetes 1.25版本后，开始不再提供autoscaling/v2beta1 API版本的HorizontalPodAutoscaler（Pod水平自动伸缩）。

在Kubernetes 1.26版本后，开始不再提供autoscaling/v2beta2 API版本的HorizontalPodAutoscaler，但可以使用autoscaling/v2 API版本，此API从Kubernetes 1.23版本开始可用。

Flow control resources

在Kubernetes 1.26版本后，开始不再提供flowcontrol.apiserver.k8s.io/v1beta1 API版本的FlowSchema和PriorityLevelConfiguration，但此API从Kubernetes 1.23版本开始，可以使用flowcontrol.apiserver.k8s.io/v1beta2；从Kubernetes 1.26版本开始，可以使用flowcontrol.apiserver.k8s.io/v1beta3。

创建集群

如果您是初次接触 Kubernetes，我们建议您预先创建好一个新的 VPC 和子网，与生产环境隔离。创建集群之前，您需要先了解下 Kubernetes 中的Node CIDR、Pod CIDR、Service CIDR等基本概念，点击查看

1. 配置集群网络信息

登录控制台私有网络 **VPC**页面，进行 VPC 网络及子网网段的规划（私有网络 VPC 文档）。在 UK8S 集群中，Pod 与其所在的 Node 同处于一个 VPC 子网下，因此 VPC 子网的网段大小决定了集群可创建的 Pod 数量上限，详情请查看Kubernetes网络。

2. 创建专有版 UK8S 集群

UK8S 专有版集群默认需要创建三个 Master 节点以保证生产环境的高可用性，登录 UK8S 服务管理控制台，在集群列表页面，点击创建集群按钮，在集群类型中选择专有版，开始进行专有版集群的创建。

基础配置

配置项	描述
集群所属 VPC	设置节点及 Pod 所处的 VPC 网络
集群所属子网	设置初始节点及 Pod 所处的子网，集群中 Node 可处于同一 VPC 下的不同子网

Service 网段	设置集群 Service 网段，Service 网段不能与节点网段重复
节点镜像	设置集群节点的 UHost 镜像，您可以选择自定义镜像，但必须基于 UK8S 标准镜像制作，请参考制作自定义镜像 若您要使用GPU节点，镜像选择参考GPU节点中的镜像说明，CPU机器镜像可选择：Centos 7.6，Ubuntu 20.04，Anolis 8.6镜像中的任意一种。

Master/Node节点配置

生产环境的 Master 配置建议可查看集群节点配置推荐。

配置项	描述
可用区	Master/Node 节点所在可用区，在具有多个可用区的地域可以选择多可用区 UK8S 集群，建议在创建集群时将 Master 节点分布于多个可用区。
节点规格	包括机型、CPU 平台、CPU、内存、系统盘类型、数据盘类型、数据盘大小等配置，详见机型与 CPU 平台。 Node 节点的数据盘会 mount 到节点的 /data 目录，集群 Node 安装 Docker 引擎时安装在 /data 目录下，如创建时 Node 节点配置使用了数据盘，手动删除数据盘会导致 Node 节点不可用，如不需要数据盘可以在创建选择时删除，Docker 引擎会安装到系统盘的 /data 目录下
硬件隔离组	Master 节点默认位于同一硬件隔离组，硬件隔离组能严格确保组内的每一台云主机都落在不同的物理机上。每个隔离组在单个可用区至多可以添加 7 台云主机，详见硬件隔离组
最大 Pod 数	单个 Node 节点可支持承载的最大 Pod 数量
标签	Node 节点标签，详见 Kubernetes 官方文档：标签和运算符 填写规则： * 标签键：必须唯一，由可选前缀和名称组成，前缀是可选的，必须是DNS子域名，不允许kubernetes.io或k8s.io的子域名，不能超过253个字符，只允许以[a-z0-9A-Z]开头和结尾，通过点.分隔；名称是必须的，少于或等于63个字符，字符只允许以[a-z0-9A-Z]开头和结尾，之间包含破折号-，下划线_，点. * 标签值：不能为空，少于或等于63个字符，字符只允许以[a-z0-9A-Z]开头和结尾，之间包含破折号-，下划线_，点.

污点	Node 节点污点，使节点能够排斥一类特定的 Pod。容忍度（Toleration）是应用于 Pod 上的，允许（但并不要求）Pod 调度到带有与之匹配的污点的节点上。污点和容忍度（Toleration）相互配合，可以用来避免 Pod 被分配到不合适的节点上。每个节点上都可以应用一个或多个污点，这表示对于那些不能容忍这些污点的 Pod，是不会被该节点接受的。 详见 Kubernetes 官方文档：污点和容忍度
Node 节点数量	初始集群 Node 节点数量限制为 1 - 10 台

管理设置

配置项	描述
集群名称	UK8S 集群名称，后期可更改
外网 APIServer	APIServer 通过 ULB 负载均衡服务对外提供暴露。内网 Master ULB 在创建集群时自动生成，如开启外网 APIServer，将自动购买一个外网 ULB 服务，起始带宽为 1MB。 APIServer 服务 ULB 命名规则为 uk8s-xxxxxxxx-master-ulb4（内网 ULB）/ uk8s-xxxxxxxx-master-ulb4-external（外网 ULB），**删除将导致集群 APIServer 服务不可用。**
K8S 版本	UK8S 集群版本
kube-proxy	默认选择为 iptables，选择标准和切换请参考：kube-proxy模式选择
容器运行时	K8S 1.19 及以上版本默认为 containerd，采用 containerd 运行时的节点，请勿自行另外安装 docker，避免配置冲突导致节点不可用。
管理员密码	适用于本次创建的所有 Master 和 Node 节点
集群本地域名	默认值为 cluster.local，用户可自定义后缀，域名由两段组成，每段不超过 63 个字符，且只能使用大小写字母和数字，不能为空。

自定义数据	是指主机初次启动或每次启动时，系统自动运行的配置脚本，该脚本可由控制台 API 等传入元数据服务器，并由主机内的 cloud-init 程序获取，脚本遵循标准 CloudInit 语法。该脚本会阻塞 UK8S 的安装脚本，即只有该脚本执行完毕后，才会开始K8S相关组件的安装，如Kubelet、Scheduler等。
初始化脚本	该脚本只在 UK8S 启动后执行一次，且是在 K8S 相关组件安装成功后执行。遵循标准shell语法，执行结果会存入到 /var/log/message/ 目录下。

集群初始化时间在 10-15 分钟左右，创建成功后，您可以通过直接登录 Master 节点访问和管理集群，也可以在同 VPC 下的云主机上通过 APIServer 管理集群。

查看集群

> 由于查看集群凭证可以直接登录集群，所以查看集群凭证的操作已归为用户角色权限中的增权限，如需查看集群凭证，请确保所在角色已开启UK8S增权限。

集群详情页面综合展示了包括集群基本信息、Master以及Node节点运行状态等信息。

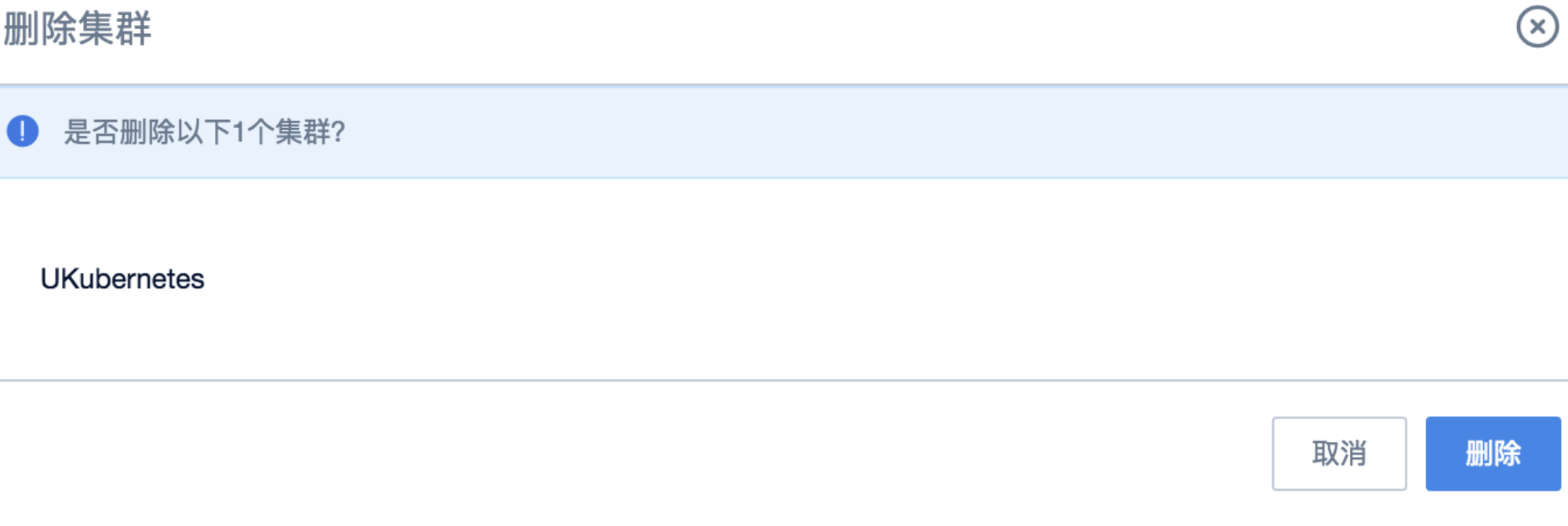
一、在集群列表页点击详情，进入集群详情页

二、集群详情页展示了包括集群信息、Node节点等信息

删除集群

你可以通过控制台直接删除现有集群，此操作为不可逆操作，请谨慎执行。

如果您在使用过程中创建过PVC、LoadBlancer类型的Service，UK8S会替代你创建UDisk、ULB等资源。建议你删除集群之前，把集群内部的Workloads、Services先删除掉，避免集群删除后，由K8S创建的一些资源依然产生费用。



kubectl命令行简介

由于查看集群凭证可以直接登录集群，所以查看集群凭证的操作已归为用户角色权限中的增权限，如需查看集群凭证，请确保所在角色已开启UK8S增权限。

kubectl是一个用于操作kubernetes集群的命令行工具，本文将简要介绍下kubectl的语法，并提供一些常见命令示例，如果你想了解深入了解kubectl的用法，请查阅官方文档kubectl overview，或使用kubectl help命令查看详细帮助。安装kubectl请查看安装及配置kubectl。

kubectl 语法

kubectl的语法示例如下：

```
kubectl [command] [TYPE] [NAME] [flags]
```

command: command意指你想对某些资源所进行的操作，常用的有create、get、describe、delete等。

TYPE: 声明command需要操作的资源类型，TYPE对大小写、单数、复数不敏感，支持缩写。比如，以下命令都是合法且等价的：

```
kubectl get pod
kubectl get pods
kubectl get po
kubectl get POD
```

NAME: 即资源的名称，NAME是大小写敏感的。如果不指定某个资源的名称，则显示所有资源，如kubectl get pods 会显示Default命名空间下所有的pod。

你还可以同时获取多个资源的详细情况，如获取同一类型的资源详情，不同类型的资源详情：

```
kubectl get pods pod1 pod2
```

```
kubectl get pod/example-pod1 replicationcontroller/example-rc1
```

flags： 可选参数，例如，你可以使用all-namespaces来获取所有namespace下的资源对象。关于各命令的flag用法请参见kubectl command

重要：命令行指定的flags将覆盖默认值和任何相应的环境变量。

更多关于kubectl命令的介绍，请使用kubectl help。

常见命令

kubectl create - 使用一个文件或者标准输入创建资源。

```
# 使用exampe-service.yaml文件创建一个“service”对象
$ kubectl create -f example-service.yaml

# 使用example-controller.yaml文件创建一个"replication"对象
$ kubectl create -f example-controller.yaml
```

kubectl describe - 获取资源的详细状态，包括初始化中的资源。

```
# 查看名为<node-name>的node节点详情
$ kubectl describe nodes <node-name>

# 查看名为<pod-name>的pod详情，包含pod的创建日志
$ kubectl describe pods/<pod-name>

# 查看所有由名为<rc-name>的replication管理的pod。
# 注意: 任何由replication controller创建的pod，其名称前缀为replication名称。
$ kubectl describe pods <rc-name>

# 查看所有pods，但不包含未初始化的pods
```

```
$ kubectl describe pods --include-uninitialized=false
```

kubectl logs - 获取某个pod的日志

```
# 获取一个pod的日志快照
$ kubectl logs <pod-name>

# 获取一个pod的实时日志流，类似于linux的'tail -f'
$ kubectl logs -f <pod-name>
```

kubectl exec - 对pod中的容器执行命令

```
# 从pod中获取运行"date"命令的输出，默认情况下，来自于pod中的第一个容器。
$ kubectl exec <pod-name> date

# 从pod中指定的容器中获取运行"date"命令的输出
$ kubectl exec <pod-name> -c <container-name> date

# 从pod中得到一个交互式tty(控制终端),并执行/bin/bash
$ kubectl exec -ti <pod-name> /bin/bash
```

安装及配置kubectl

由于查看集群凭证可以直接登录集群，所以查看集群凭证的操作已归为用户角色权限中的增权限，如需查看集群凭证，请确保所在角色已开启UK8S增权限。

本文主要演示如何在UCloud云主机上安装配置kubectl并管理Kubernetes集群，集群Master节点已默认安装kubectl工具，如果你仅需在Master节点做一些简单测试，请跳过此环节；

云主机环境

操作系统：linux，windows请移步官方文档。

所属VPC：与集群同VPC

开通外网：是

一、安装kubectl

1. 下载安装包，我们下载V1.13.5的kubectl安装包，其他版本请前往官网下载。

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.13.5/bin/linux/amd64/kubectl
```

如果您要下载最新版本的安装包，使用如下命令即可：将v1.13.5替换为$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)即可。

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
```

2. 添加执行权限

```
chmod +x ./kubectl
```

3. 移至工作路径

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

4.输入kubectl version，发现已经安装成功。

```
#kubectl version
Client Version: version.Info{Major:"1", Minor:"11", GitVersion:"v1.11.0", GitCommit:"91e7b4fd31fcd3d5f436da26c980becec37ceefe", GitTreeState:"clean",
BuildDate:"2018-06-27T20:17:28Z", GoVersion:"go1.10.2", Compiler:"gc", Platform:"linux/amd64"}
```

备注：如果您需要在ubuntu或其他linux发行版安装kubectl，亦或使用yum安装，可以参见官方文档。

二、获取并配置集群凭证

你可以通过UK8S Console、SCP、API三种途径获取您创建的集群凭证。

备注：集群内访问无需凭证，可直接访问。

1. 通过Console获取集群凭证

点击进入到<集群详情页>，点击“集群凭证”

UKubernetes服务 UK8S / **UKubernetes**

概览

基本信息

集群名称	UKubernetes
集群ID	uk8s-cnhu35
Master节点	3个
Node节点	50个
创建时间	2018-11-13 11:07:15
Apiserver地址	10.23.242.185
集群凭证	查看

Master节点列表

主机名称	资源ID	可用区	配置
uk8s-cnhu35-master-1	uhost-njqu3o	上海二可用区B	2 | 4 | 0
uk8s-cnhu35-master-2	uhost-gp2vc5	上海二可用区B	2 | 4 | 0
uk8s-cnhu35-master-3	uhost-0b5y5y	上海二可用区B	2 | 4 | 0

将集群信息复制保存到~/.kube/config文件下即可

2. 通过SCP从Master节点下载集群凭证到本地

首先点击进入集群详情页面，获取任意一台Master节点的IP，然后在本地机器执行以下命令：

```
scp root@YOURMASTERIP:~/.kube/config ~/.kube/config
```

三、访问集群

你可以执行以下命令来验证kubectl是否可以成功访问集群信息；

```
# kubectl cluster-info
```

四、设置命令自动补全

在kubectl所在节点执行安装

```
yum install bash-completion -y
```

kubectl支持命令自动补全，执行以下命令即可开启。

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

使用web kubectl

UK8S 在console中提供 web terminal，你可以通过web terminal 登录到集群内的Pod，并使用kubectl 操作和管理集群。

该Pod通过Deployment的方式启动，并通过特定的安全机制代理到UCloud控制台界面，如果你误删除了该Deployment，则无法使用console中的kubectl功能。

你可以使用下方的yaml文件重新启动一个Pod,yaml示例如下。

> 备注：uk8s-kubectl的镜像tag与您的UK8S集群版本一致，如你的UK8S版本为1.14.5，则将镜像tag改为v1.14.5即可。

```
# ------------------- kubectl Deployment ------------------- #
apiVersion: apps/v1
kind: Deployment
metadata:
labels:
k8s-app: uk8s-kubectl
name: uk8s-kubectl
namespace: kube-system
spec:
replicas: 1
selector:
matchLabels:
k8s-app: uk8s-kubectl
template:
metadata:
labels:
k8s-app: uk8s-kubectl
spec:
serviceAccountName: uk8s-kubectl
containers:
- image: uhub.service.ucloud.cn/ucloud/uk8s-kubectl:v1.14.6
imagePullPolicy: IfNotPresent
name: uk8s-kubectl
resources:
```

```
requests:
memory: "100Mi"
cpu: "100m"
limits:
memory: "500Mi"
cpu: "500m"

---
# ------------------- Service Account ------------------- #

apiVersion: v1
kind: ServiceAccount
metadata:
labels:
k8s-app: uk8s-kubectl
name: uk8s-kubectl
namespace: kube-system

---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
name: uk8s-kubectl-rolebind
annotations:
rbac.authorization.kubernetes.io/autoupdate: "true"
```

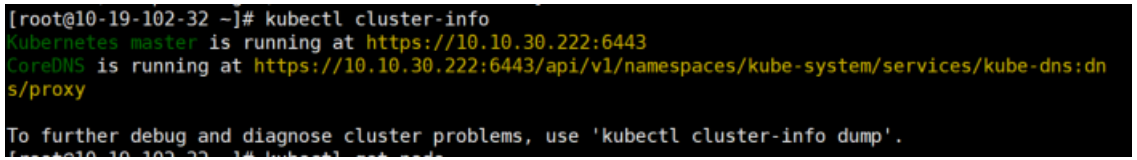
```
roleRef:
kind: ClusterRole
name: cluster-admin
apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
name: uk8s-kubectl
namespace: kube-system
```

集群更新凭证

UK8S 支持用户通过 kubectl 工具连接 kubernetes 集群，详见使用 kubectl 操作集群

更新新 Token 访问集群

集群凭证现在展示的为 Token 访问方式（非原证书访问方式），按照上面的文档链接操作，我们已经复制信息至`~/.kube/config`文件中，集群已经可以正常连接访问。



如您将您的凭证有对外展示过并且您认为目前凭证存在泄漏的风险，可以通过点击更新凭证将集群凭证进行刷新，刷新后，老的集群凭证将无法继续使用，需要从 UK8S 页面重新获取复制。

妥善保管好您的证书访问集群

UK8S 的 master 节点默认安装了 kubectl 工具，配置的为内网证书访问凭证，证书访问凭证将不会被刷新，请您妥善保管好您的证书安全

创建PVC

当前存储卷支持SSD、SATA类型的UDisk以及UFS，详见：

- 在UK8S中使用UDisk
- 在UK8S中使用UFS

创建StorageClass

在创建持久化存储卷(persistentVolume)之前,你需要先创建StorageClass,然后在PVC中使用StorageClassName。

UK8S集群默认创建了两个StorageClass,你也可以创建一个新的StorageClass,示例及说明如下:

1、CSI版本(2019年9月17日之后创建的UK8S集群)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: udisk-ssd-test
provisioner: udisk.csi.ucloud.cn #存储供应方,此处不可更改。
parameters:
type: "ssd" # 存储介质,支持sdd和sata,必填
fsType: "ext4" # 文件系统,必填
udataArkMode: "no" # 是否开启方舟模式,默认不开启,非必填
chargeType: "month" # 付费类型,支持dynamic、month、year,默认为month,非必填
quantity: "1" # 购买时长,dynamic无需填写,可购买1-9个月,或1-10年
reclaimPolicy: Delete # PV回收策略,支持Delete和Retain,默认为Delete,非必填
allowVolumeExpansion: true # 声明该存储类支持可扩容特性
mountOptions:
- debug
```

```
- rw
```

备注:1.15之前的Kubernetes版本,mountOptions无法正常使用,请勿填写,详见Issue80191

2、flexVolume版本(2019年9月17日之前创建的UK8S集群)

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: udisk-ssd-test
provisioner: ucloud/udisk
parameters:
type: ssd
reclaimPolicy: Retain
```

provisioner: 存储供应方,此处必须为ucloud/udisk,否则创建出来的StorageClass可能无效。

parameters.type: UDisk的存储介质类型,支持ssd和sata,默认为ssd。

reclaimPolicy: 回收策略,支持Delete和Retain,默认为Delete。

创建一个存储卷声明并Mount到Pod

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
```

```
name: test-pvc-claim
spec:
accessModes:
- ReadWriteOnce
storageClassName: udisk-ssd-test #注意修改为你自己创建的StorageClassName
resources:
requests:
storage: 20Gi


---
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
containers:
- name: nginx
image: nginx
volumeMounts:
- name: test
mountPath: /data
ports:
- containerPort: 80
volumes:
- name: test
```

```
persistentVolumeClaim:
claimName: test-pvc-claim
```

备注：受UDisk产品限制，PVC最小为20GB，步长为10GB。

容器启动后，我们可以登录容器执行df -h 命令，查看存储卷是否挂载成功。

创建Service

创建一个类型为LoadBalancer的Service，将MYSECRET换成自定义的SecretName即可。

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-nginx
labels:
app: ucloud-nginx
spec:
type: LoadBalancer
ports:
- protocol: TCP
port: 80
selector:
app: ucloud-nginx
```

```
---
apiVersion: v1
kind: Pod
metadata:
name: test-nginx
labels:
app: ucloud-nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
ports:
- containerPort: 80
imagePullSecrets:
- name: MYSECRET
```

系统会自动生成一个ULB，UK8S同时还支持配置ULB的各种参数，详见service

StatefulSet部署示例

在部署一些有状态的服务如 Redis、MySQL 等时，我们需要使用到 StatefulSet 这个控制器，下面介绍下如果在 UK8S 中使用 UDisk 来部署 StatefulSet 服务。

了解StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: mysql
namespace: db
spec:
selector:
matchLabels:
app: mysql
replicas: 5
serviceName: mysql
template:
PodTemplateSpec..... # 有大量省略，与Deployment一样，是关于要控制的Pod的描述
volumeClaimTemplates:
- metadata:
name: www
spec:
accessModes: [ "ReadWriteOnce" ]
storageClassName: ${YOUR_STORAGECLASS_NAME}
resources:
requests:
storage: 10Gi
```

如果我们熟悉 Deployment 的结构体，则会发现其与 StatefulSet 最大的区别在于**volumeClaimTemplates**，其他地方则基本一致。

我们细看下**volumeClaimTemplates**，发现其结构体与“PersistentVolumeClaim”完全一致，没错，**volumeClaimTemplates**其实就是PVC的模板，用来生成多个访问模式为单点读写的PVC，供 StatefulSet 管理的 Pod 使用。

像上面的示例，StatefulSet 不仅会创建出5个 Pod，同时也还会创建出5个 PVC，供对应的Pod使用，以实现每个 Pod 都具有独立的存储状态。

PVC 示例

对于有状态服务，我们推荐使用SSD UDisk、RSSD UDisk作为存储介质，当然，我们也可以使用LocalPV，但由于目前大多数云主机的数据盘也都是云盘，直接使用 LocalPV 还有各种限制，因此**强烈推荐使用 UDisk 作为存储介质**

UK8S 集群在初始化的时候，已经内置了三个与 UDisk 相关的存储类，我们只需要直接引用存储类创建 PVC 供 Pod 消费即可。下面介绍下如何创建对应的PVC。

> ⚠ **RSSD UDisk调度要求同一个RDMA区域的快杰型云主机，RDMA区域范围小于可用区，主机目前不支持指定RDMA区域创建机器。因此使用RSSD UDisk，在Pod漂移的情况下，有可能出现Pod无法调度的问题。请您使用前务必确认可以接受该风险。**

1. 使用RSSD UDisk

```
volumeClaimTemplates:
- metadata:
name: ${YOUR_NAME} # 需要与VolumeMount的名称保持一致;
spec:
storageClassName: csi-udisk-rssd #这是集群内置的StorageClass
accessModes:
- ReadWriteOnce
resources:
```

```
requests:
storage: 100Gi
```

上面我们使用的是集群内置的 StorageClass，我们也可以根据创建新的 SC，详见使用RSSD UDisk

2. 使用SSD UDisk

```
volumeClaimTemplates:
- metadata:
name: ${YOUR_NAME}
labels:
name: redis-cluster
spec:
storageClassName: ssd-csi-udisk #这是自行创建的存储介质为SSD UDisk的StorageClass
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 20Gi
```

我们看到，需要使用不同的存储介质，只需要在创建PVC时声明不同的storageClassName即可。下面我们介绍下如何创建自定义StorageClass。

3. 声明自定义的StorageClass(UDisk 类型)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
```

```
name: udisk-ssd-test
provisioner: udisk.csi.ucloud.cn #存储供应方，此处不可更改。
parameters:
type: "ssd" # 存储介质，支持ssd和sata、rssd，必填
fsType: "ext4" # 文件系统，必填
udataArkMode: "no" # 是否开启方舟模式，默认不开启，非必填
chargeType: "month" # 付费类型，支持dynamic、month、year，非必填
quantity: "1" # 购买时长，dynamic无需填写，可购买1-9个月，或1-10年
reclaimPolicy: Retain # PV回收策略，支持Delete和Retain，默认为Delete，非必填
allowVolumeExpansion: true # 声明该存储类支持可扩容特性
mountOptions:
- debug
- rw
```

上面的示例涵盖了 UDisk 的StorageClass的全部参数，我们可以根据业务需要来自定义 SC。

StatefulSet 示例

```
apiVersion: v1
kind: Service
metadata:
name: nginx
labels:
app: nginx
spec:
```

```
ports:
- port: 80
name: web
clusterIP: None
selector:
app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: web
spec:
selector:
matchLabels:
app: nginx # has to match .spec.template.metadata.labels
serviceName: "nginx"
replicas: 3 # by default is 1
template:
metadata:
labels:
app: nginx # has to match .spec.selector.matchLabels
spec:
terminationGracePeriodSeconds: 10
containers:
- name: nginx
```

```
image: uhub.service.ucloud.cn/ucloud/nginx:latest
ports:
- containerPort: 80
name: web
volumeMounts:
- name: www
mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
name: www
spec:
accessModes: [ "ReadWriteOnce" ]
storageClassName: "csi-udisk-rssd" # has to match a storageClassname existed in your cluster
resources:
requests:
storage: 100Gi
```

在上面的示例中，我们声明的名称为 Web 的 StatefulSet 控制器，将创建一个3个nginx Pod，并且为每个Pod分别挂载一个RSSD UDisk，以供其存储数据。

将pod打散

- 有些情况下我们希望将服务分散在各个节点，例子如下

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
name: nginx
spec:
selector:
matchLabels:
app: nginx # 需要下面的的标签一致
replicas: 3
template:
metadata:
labels:
app: nginx # 需要下面的matchExpressions的标签一致
spec:
affinity:
podAntiAffinity: # 反亲和
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 100
podAffinityTerm:
labelSelector:
matchExpressions:
- key: app # 注意这里的标签要和上面的标签一致
operator: In
values:
- nginx
topologyKey: "kubernetes.io/hostname"
```

```
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
```

更多调度方式参考官方文档

kubectl 常见问题

1. kubectl top 命令报错

该情况一般有以下两种可能

1. kube-system下面metrics-server Pod没有正常工作，可以通过kubectl get pods -n kube-system进行查看
2. metrics.k8s.io API地址被错误重定向，可以执行kubectl get apiservice v1beta1.metrics.k8s.io查看重定向到的service名称，并确认service当前是否可用及是否对外暴露了v1beta1.metrics.k8s.io接口。默认重定向地址为kube-system/metrics-server

情况二一般出现在部署prometheus，且prometheus提供的接口不支持v1beta1.metrics.k8s.io时。如果不需要自定义HPA指标，其实不需要此重定向操作。如果属于情况二，可以按照下面步骤操作。

1. 确认配置中的的prometheus service可用，并根据需要自定义HPA指标
2. 重新部署执行下面yaml文件，回退到普通metrics server，Grafana等不依赖此api。
3. 注意，如果您之前已经使用了自定义HPA指标，且处于线上环境，建议您仅确认prometheus service可用即可。回退到普通metrics server可能导致之前的自定义HPA指标失效，请谨慎操作。

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
name: v1beta1.metrics.k8s.io
spec:
service:
```

```
name: metrics-server
namespace: kube-system
group: metrics.k8s.io
version: v1beta1
insecureSkipTLSVerify: true
groupPriorityMinimum: 100
versionPriority: 100
```

手动增加外网凭证

如果您在创建集群的时候，没有打开外网APIServer选项，您会发现在集群管理没有外网凭证，也就无法通过外网访问APIServer了。

在这种情况下，您可以手动为集群加上外网凭证。本文将会介绍详细步骤。

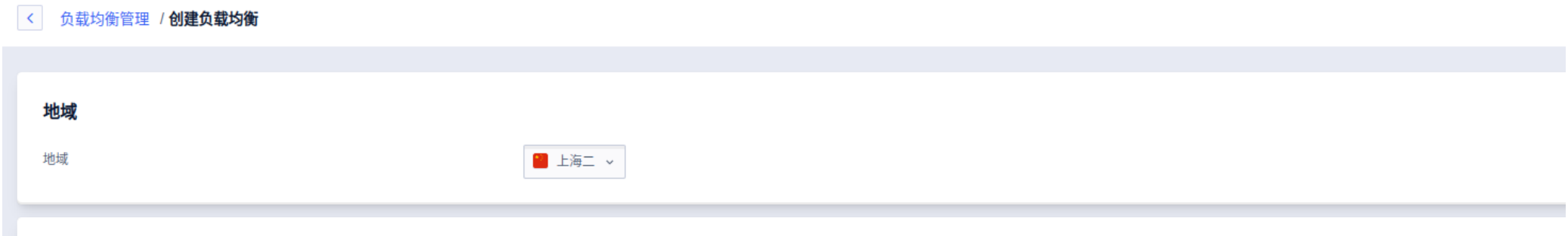
> 注意，增加外网凭证需要重启master节点的apiserver，请在业务低谷期操作。

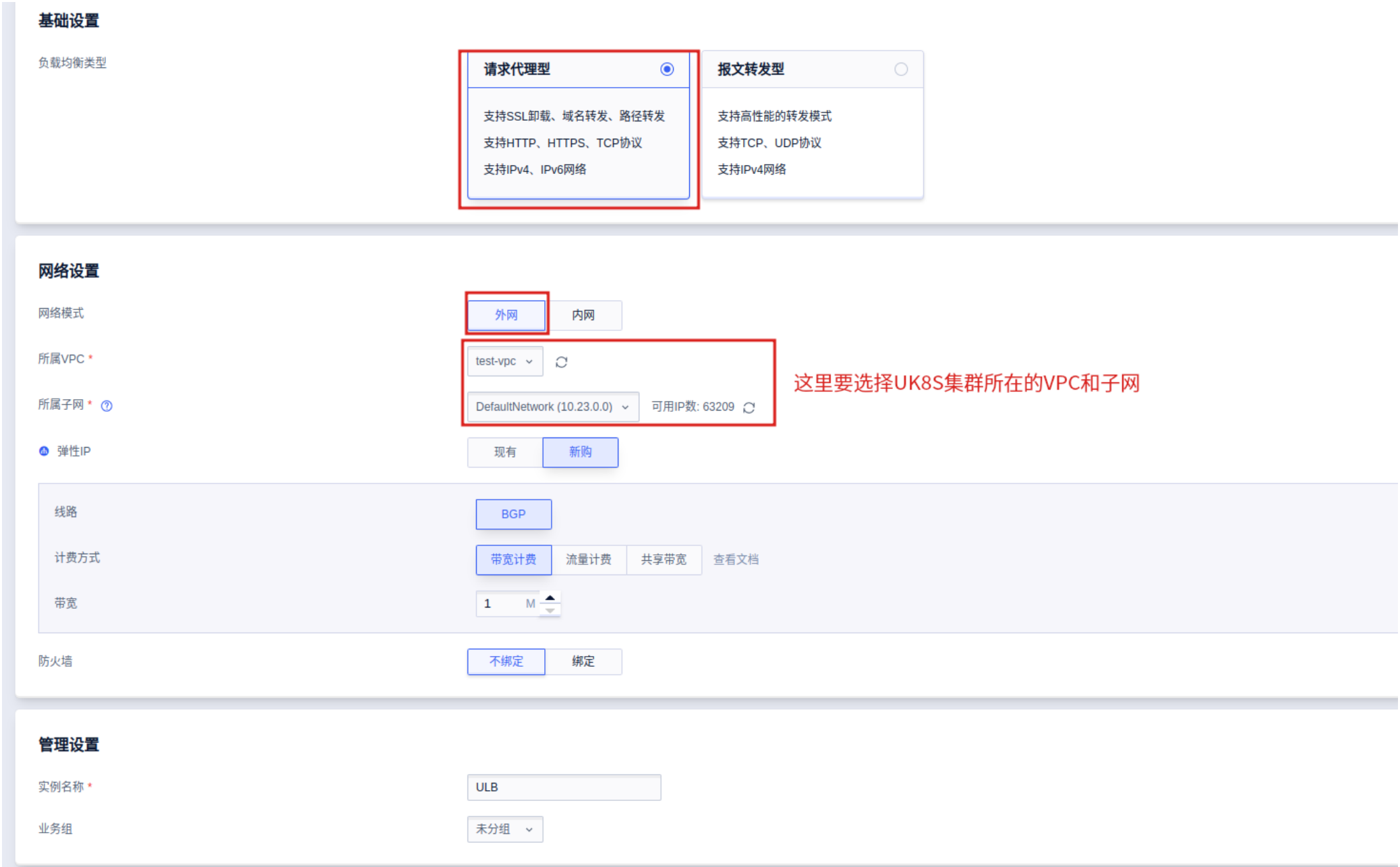
创建ULB

注意事项：由于该外网 ULB 不是随 UK8S 一起创建的，所以并不会在 UK8S 集群删除后同步删除，应当手动删除该 ULB

您需要手动创建一个外网ULB以用于访问master节点。

在“负载均衡 ULB”页面点击创建“创建负载均衡”，选择负载均衡类型为“请求代理型”，选择网络模式为“外网”，选择所属 VPC 为 UK8S 集群所在的 VPC，选择弹性 IP 相关配置后进行创建，如下图所示：

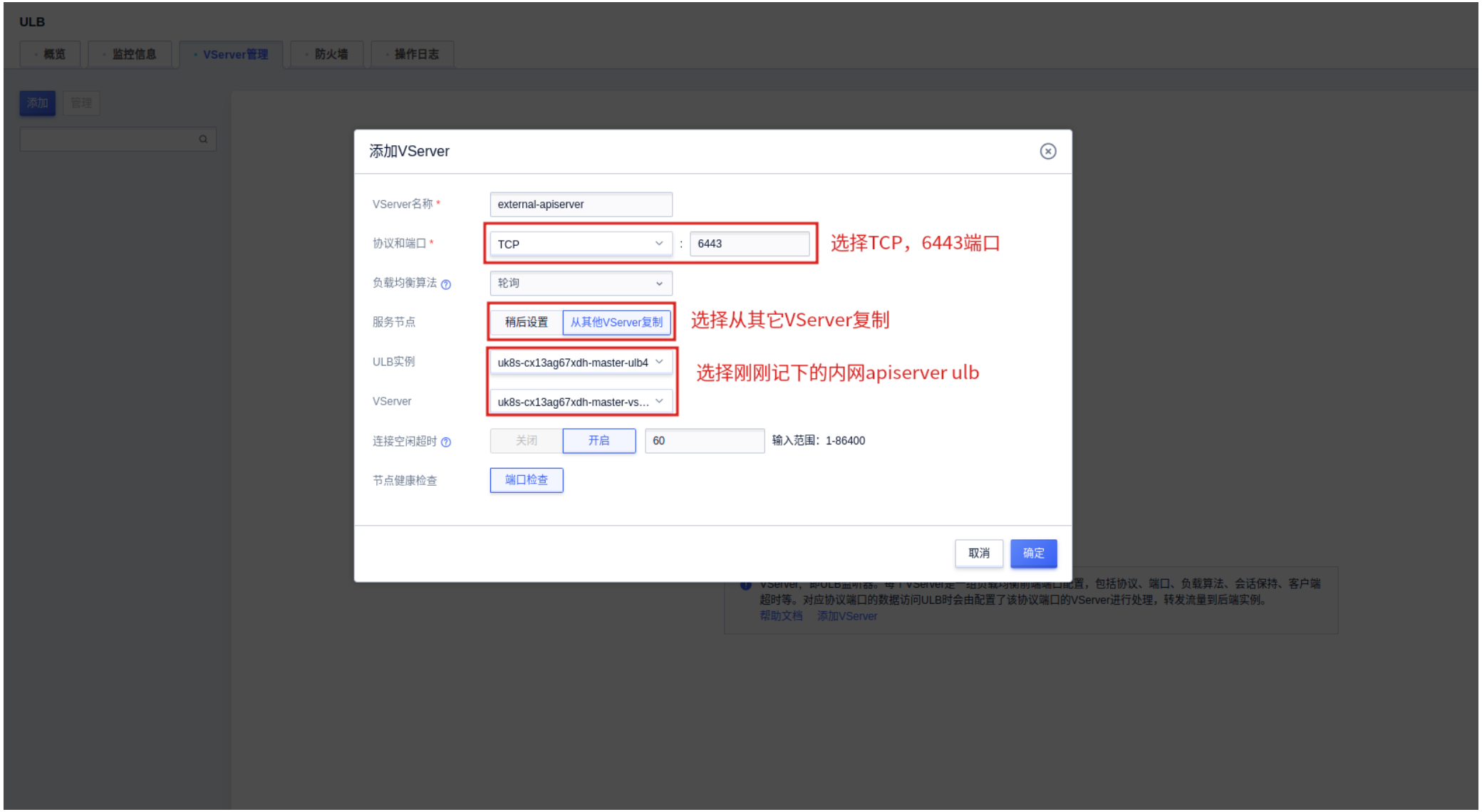




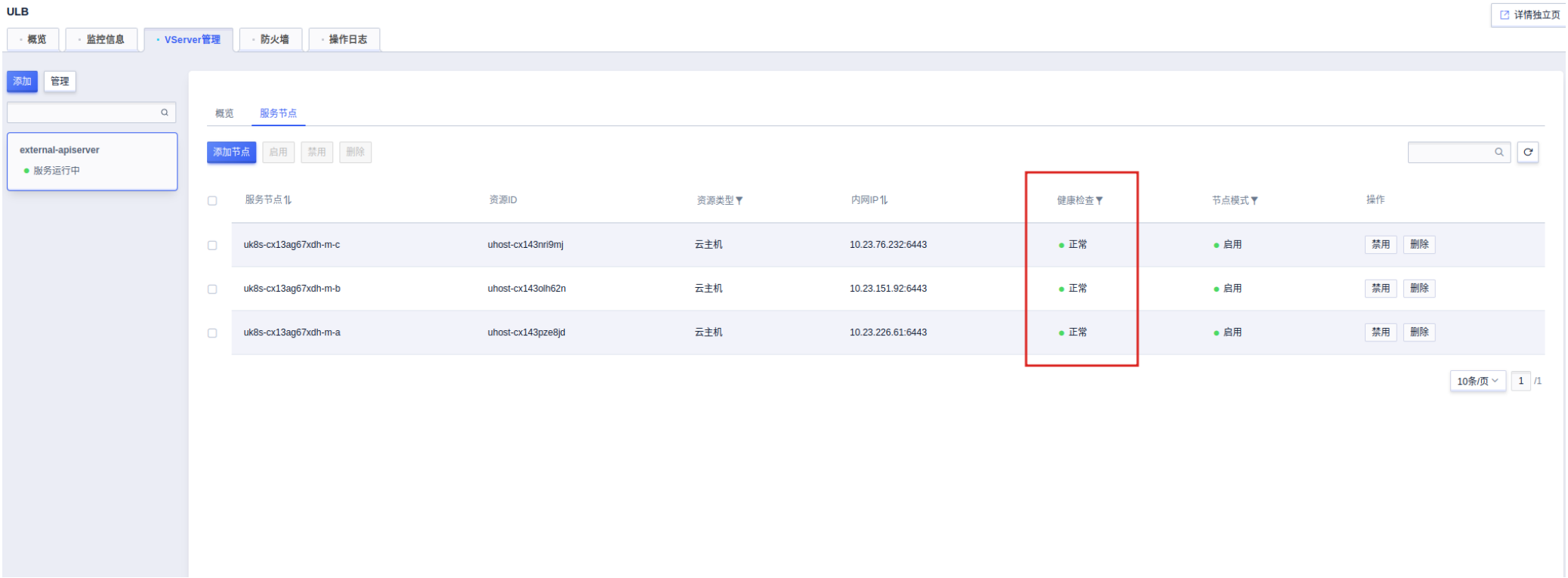
创建VServer

先找到 UK8S 内网 **APIServer** 所使用的 ULB，并记下其名称，本例中为“uk8s-cx13ag67xdh-master-ulb4”：

在步骤 1 创建的外网 ULB 详情页中，点击“VServer管理”后，点击“创建 VServer”，选择协议和端口为“TCP”协议和 6443 端口，选择服务节点为“从其他VServer复制”，选择 ULB 实例为内网 APIServer 所使用的 ULB，填写其他创建参数后进行创建，如下图所示。



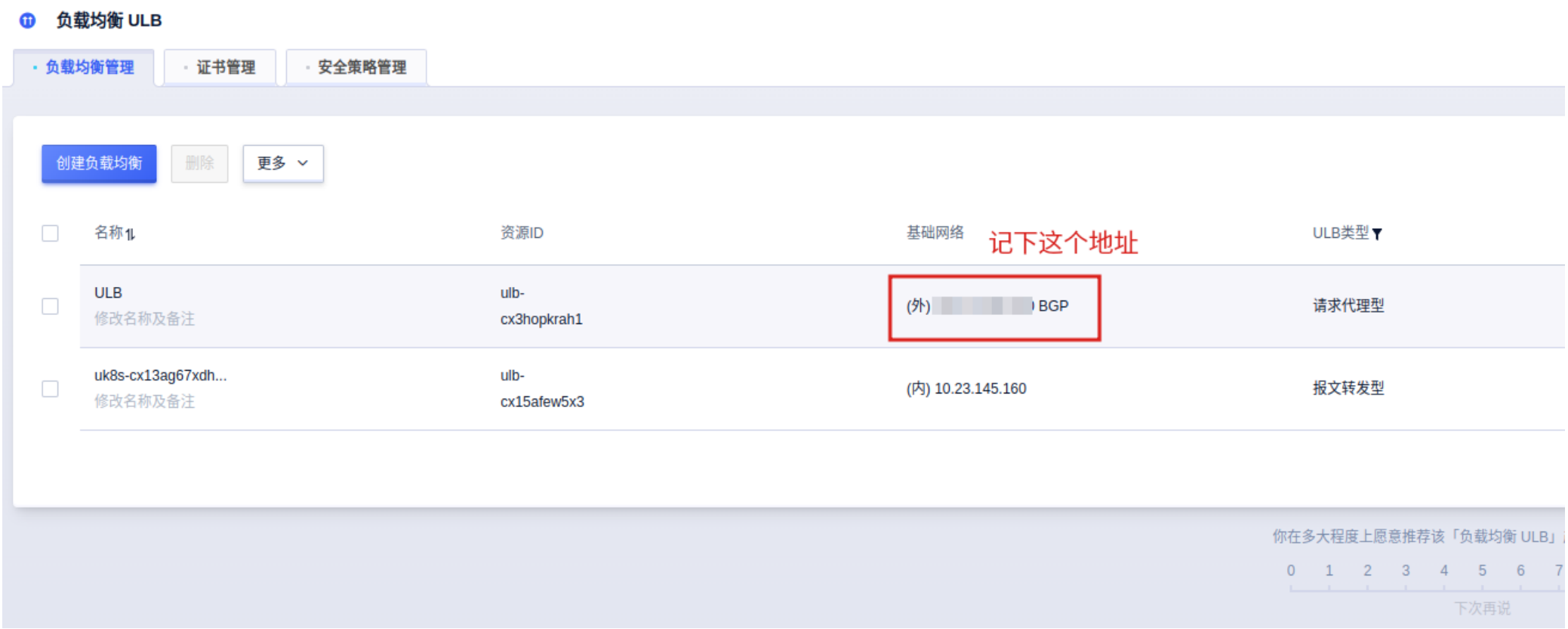
创建之后，可在 VServer 详情的服务节点页面看到三台 master 节点的健康检查状态变为“正常”，如下图：



重新生成SSL证书

注意，下面的操作需要到3个master节点都执行一次

记下上面创建的外网ULB的EIP：



这里假设EIP为99.99.99.99。

通过ssh登录到master节点，安装SSL证书工具：

```
curl -L -o cfssl https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
curl -L -o cfssljson https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
chmod +x cfssl*
sudo mv cfssl* /usr/local/bin/
```

继续在刚才已安装工具的 master 节点，查询当前 APIServer 服务的 SSL 证书信息：

```
openssl x509 -noout -text -in /etc/kubernetes/ssl/kube-apiserver.pem | grep DNS
```

你应该可以看到这样的输出：

```
DNS:kubernetes, DNS:kubernetes.default, DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster, DNS:kubernetes.default.svc.cluster.local, IP Address:127.0.0.1, IP Address:172.16.0.1, IP Address:10.23.145.160, IP Address:10.23.151.92, IP Address:10.23.76.232, IP Address:10.23.226.61
```

记下上面所有的地址。

下面，在任意位置创建一个kube-apiserver-csr.json文件，将上面所有的内网地址，以及你的ULB外网地址加到文件中：

```
{
"CN": "kubernetes",
"hosts": [
"127.0.0.1",

// 这里替换为刚刚通过openssl命令输出的IP地址
"172.16.0.1",
"10.23.145.160",
"10.23.151.92",
"10.23.76.232",
"10.23.226.61",

// 这里替换为您刚刚创建的外网ULB的EIP地址
```

```
"99.99.99.99",

// 这里替换为openssl命令输出的DNS
"kubernetes",
"kubernetes.default",
"kubernetes.default.svc",
"kubernetes.default.svc.cluster",
"kubernetes.default.svc.cluster.local"
],
"key": {
"algo": "rsa",
"size": 2048
},
"names": [{
"C": "CN",
"ST": "BeiJing",
"L": "BeiJing",
"O": "k8s",
"OU": "System"
}]
}
```

在替换之后，记得把json文件中的注释删除。

再创建一个文件`ca-config.json`**，添加如下内容：**

```
{
"signing": {
"default": {
"expiry": "87600h"
},
"profiles": {
"kubernetes": {
"usages": [
"signing",
"key encipherment",
"server auth",
"client auth"
],
"expiry": "87600h"
}
}
}
}
```

先备份一下之前的ssl证书：

```
cp -r /etc/kubernetes/ssl /etc/kubernetes/ssl-back
```

执行下面的命令生成证书：

```
cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem -ca-key=/etc/kubernetes/ssl/ca-key.pem -config=ca-config.json -profile=kubernetes kube-apiserver-csr.json | cfssljson -bare /etc/kubernetes/ssl/kube-apiserver
```

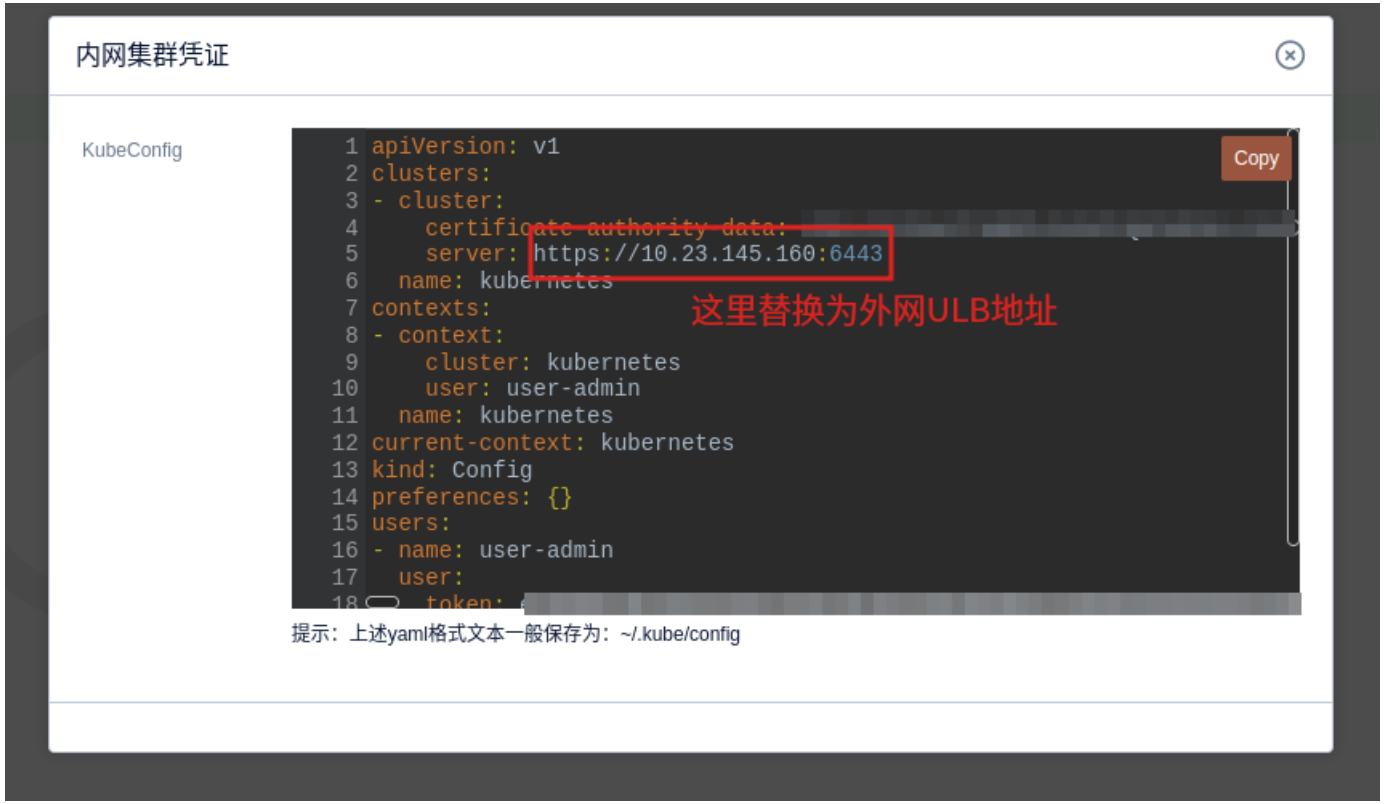
用下面的命令验证外网IP已经被加上去了(注意把99.99.99.99替换为您自己的外网IP):

```
openssl x509 -noout -text -in /etc/kubernetes/ssl/kube-apiserver.pem | grep '99.99.99.99'
```

重启apiserver,以让证书生效:

```
systemctl restart kube-apiserver
```

通过 kubectl 访问外网 APIServer 的步骤和访问内网 APIServer 一致,可在 UK8S 集群详情页复制“内网凭证”,并修改当中的 server 字段,把内网 IP 地址改为 ULB 的外网 IP 地址,如下图:



通过这个凭证您就可以在外网访问集群了。

部署Kubernetes Dashboard

Dashboard是Kubernetes社区的一个Web开源项目，你可以通过Dashboard来部署更新应用、排查应用故障以及管理Kubernetes集群资源。另外，Dashboard还提供了集群的状态，以及错误日志等信息。下面我们介绍下如何在UK8S上部署、访问DashBoard。

部署Dashboard

UK8S集群没有默认安装Dashboard，如果你希望体验社区原生Dashboard，需要自行安装，官方文档。执行以下命令安装Dashboard，使用的镜像已经去掉了Https的证书限制。

Dashboard v1.10.0

推荐kubernetes1.12及以下版本使用

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/service/dashboard.v1.10.0.yaml
```

官方兼容性提示

kubernetes版本	1.8	1.9	1.10	1.11	1.12	1.13
兼容性	✓	✓	✓	?	?	×

- ✓ 完全支持的版本范围。
- ? 由于Kubernetes API版本之间的存在变化，某些功能可能无法正常使用（测试未覆盖完整）。
- × 不支持的版本范围。

Dashboard v2.0.0-rc1

推荐kubernetes1.13及以上版本使用

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/service/dashboard.v2.0.0-rc1.yaml
```

官方兼容性提示

kubernetes版本	1.12	1.13	1.14	1.15	1.16
兼容性	?	?	?	?	✓

- ✓ 完全支持的版本范围。
- ? 由于Kubernetes API版本之间的存在变化，某些功能可能无法正常使用（测试未覆盖完整）。
- × 不支持的版本范围。

您可以通过访问或下载这个yaml文件，这个yaml中使用外网ULB暴露服务，产生额外的EIP费用。

Service的访问类型为HTTP，如果您希望使用HTTPS，请先购买SSL证书。

Dashboard v3.0.0

- http

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/service/dashboard-http.v3.0.0.yaml
```

- https

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/service/dashboard-https.v3.0.0.yaml
```

kubernetes版本	1.22	1.23	1.24	1.25	1.26
兼容性	?	?	?	✓	?

- ✓ 完全支持的版本范围。
- ? 由于Kubernetes API版本之间的存在变化，某些功能可能无法正常使用（测试未覆盖完整）。
- × 不支持的版本范围。

访问Dashboard

在上面的实例中，我们创建了一个类型为LoadBalancer的service，可直接通过Service 的外网IP（实际为ULB的外网IP）访问Dashboard。

获取到外网IP后，我们直接在浏览器中输入IP，到达登录页面，Dashboard支持kubeconfig和token两种身份验证方式，此处我们选择Token验证方式。将获取的token复制到输入框，点击登录，即可开始使用Dashboard。

> 注意：使用chrome登录，会报证书错误，点高级之后进入即可（mac电脑需要直接在键盘上盲输thisisunsafe）

Dashboard v1.10.0

查看EIP

```
kubectl get svc -n kube-system | grep kubernetes-dashboard-http
```

查看TOKEN

```
kubectl describe secret dashboard-ui -n kube-system
```

Dashboard v2.0.0-rc1

查看EIP

```
kubectl get svc -n kubernetes-dashboard | grep kubernetes-dashboard
```

查看TOKEN

```
kubectl describe secret -n kubernetes-dashboard kubernetes-dashboard-token
```

Dashboard v3.0.0

http

- http方式只支持本地地址访问

```
kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard 8080:80
```

查看TOKEN

```
kubectl describe secret -n kubernetes-dashboard kubernetes-dashboard-token
```

1.24及以上查看token

- 获取访问token，这里使用了kubernetes-dashboard下的sa kubernetes-dashboard创建了token，--duration表示token有效期

```
kubectl -n kubernetes-dashboard create token kubernetes-dashboard --duration=1h
```

https

- 获取外网ulb地址

```
kubectl -n kubernetes-dashboard get svc kubernetes-dashboard --output jsonpath="{.status.loadBalancer.ingress[*]['ip']}"
```

集群版本升级

本文主要介绍如何在 UK8S 集群中，使用版本升级功能。

大版本升级暂时仅支持大于等于 1.14 版本的 UK8S 集群；容器运行时为 Docker 时仅支持到 1.22 版本

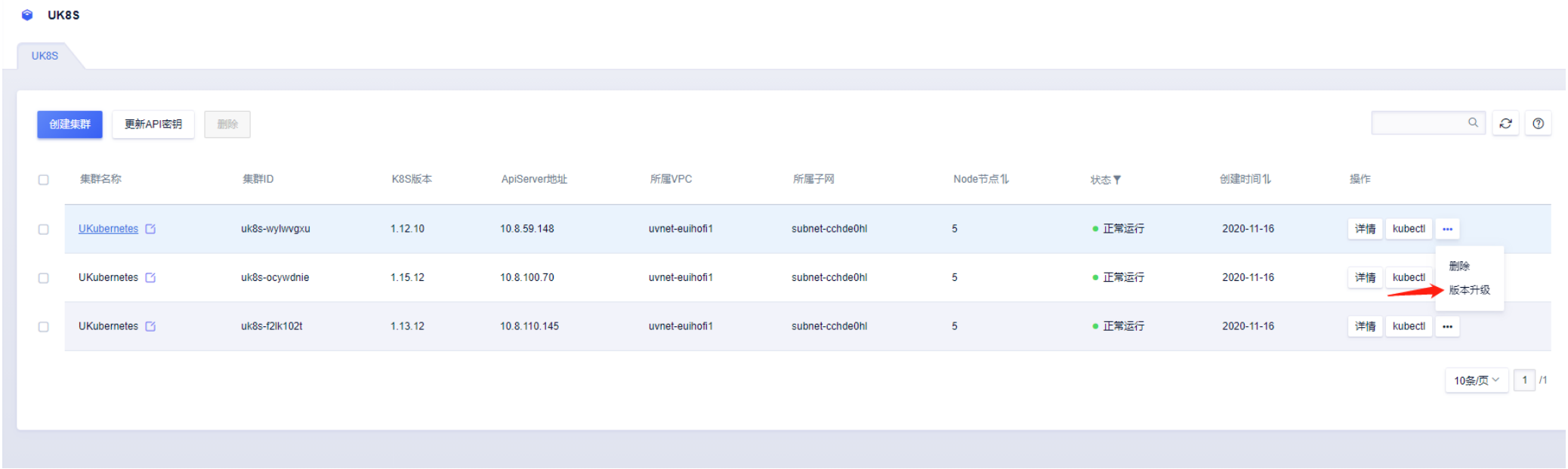
注意事项

1. 升级之前请务必关闭集群中的伸缩组插件！
2. 目前版本升级支持大版本升级和小版本升级。
3. 由于大版本之间 API 及组件变化较大，现大版本升级支持集群大于等于 1.14 版本进行使用。
4. 小版本升级支持升级至 UK8S 维护的最新小版本，具体可以点击版本升级进行查询。
5. 版本升级会导致集群内容器重启，导致重启会有以下几种情况：
    1. 1.14 版本集群进行大版本升级，升级至 1.15、1.16、1.17、1.18
    2. 1.15 版本集群进行大版本升级，升级至 1.16、1.17、1.18
    3. 单节点升级时间过长（kubernetes 中节点 NotReady 持续 5 分钟后，Pod 将会飘到其他 Ready 节点上）
6. 升级前请检查uk8s提供组件参数是否进行过手动调整，如果有调整，可能会造成升级失败或者升级成功组件参数被覆盖。

升级操作

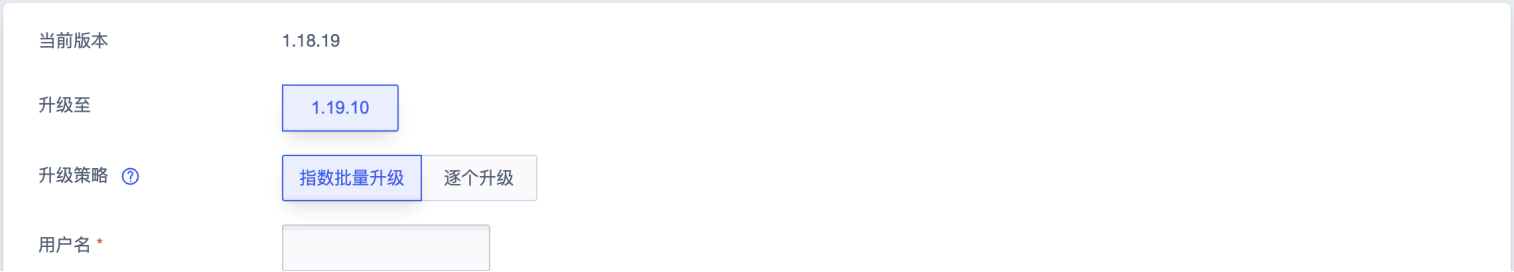
请在集群升级之前，在集群中所有节点创建拥有 Root 权限的账号且保证密码一致，关闭集群中的集群伸缩插件。

我们在集群列表页面，选择需要升级的集群，并点击操作【更多】按键可以看到【版本升级】。



点击【版本升级】后进入【UK8S 版本升级】界面：

UK8S版本升级



在此界面中，您可以设置升级的目标版本、升级策略，并提供 SSH 账号及密码，是否强制升级等。

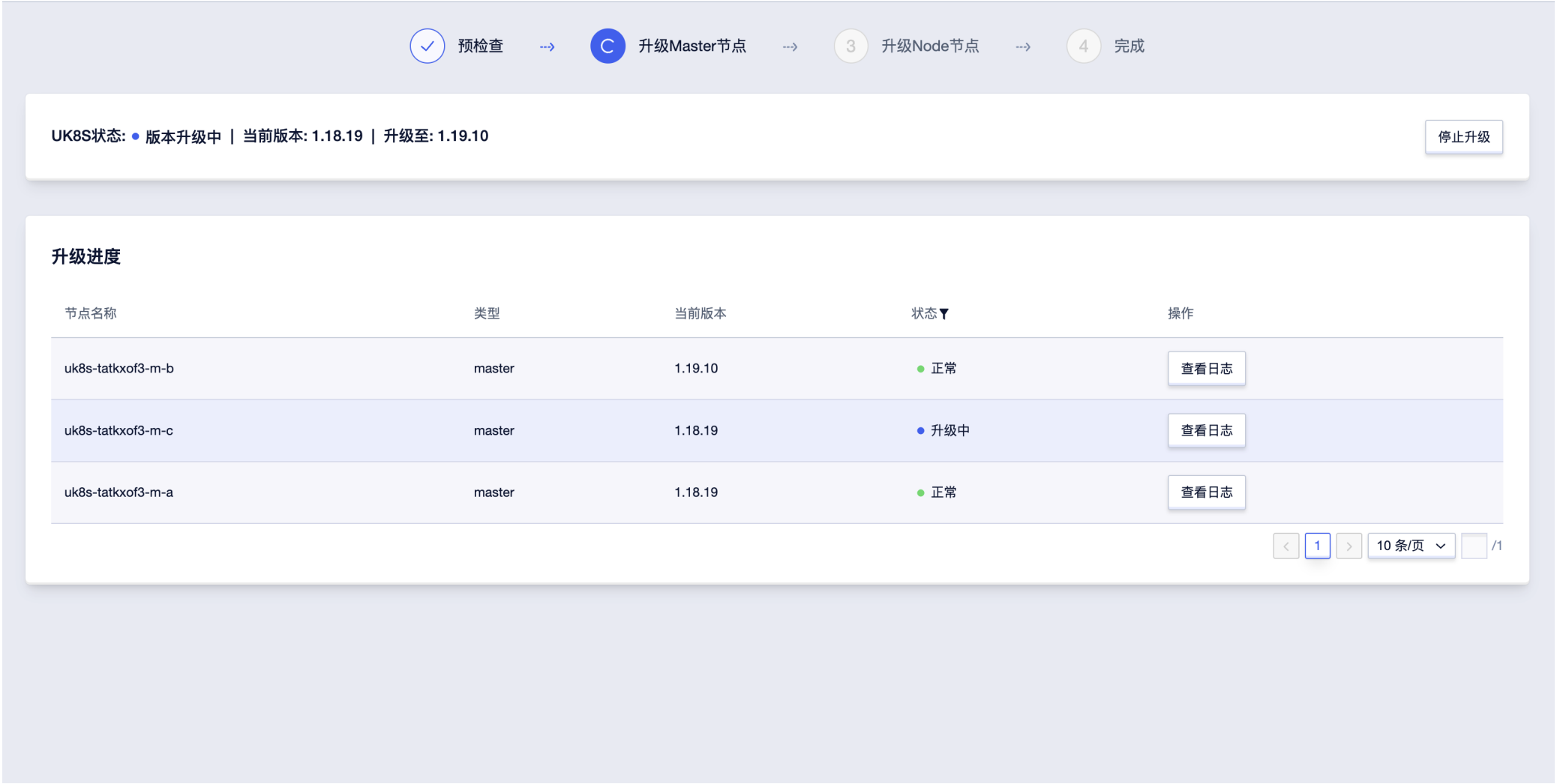
强制升级将会忽略 K8S 集群预检查，直接进行升级，仍需要输入正确的 Root 账号密码。

可选的升级策略有：

- 指数批量升级：使用指数批量升级将按每批 1，2，4，8，16，32 的数量升级节点，由于集群中的应用的多个副本都部署在被升级同一批节点上，使用此升级策略有造成服务分钟级中断的风险
- 逐个升级：使用逐个升级每次只升级 1 个节点，节点较多时升级过程较长，请合理规划升级窗口

设置好上述选项后，按【立即升级】按钮进入集群升级操作进度界面。集群升级操作较为耗时，请耐心等待。升级期间不要对集群节点进行操作，确保升级顺利完成。

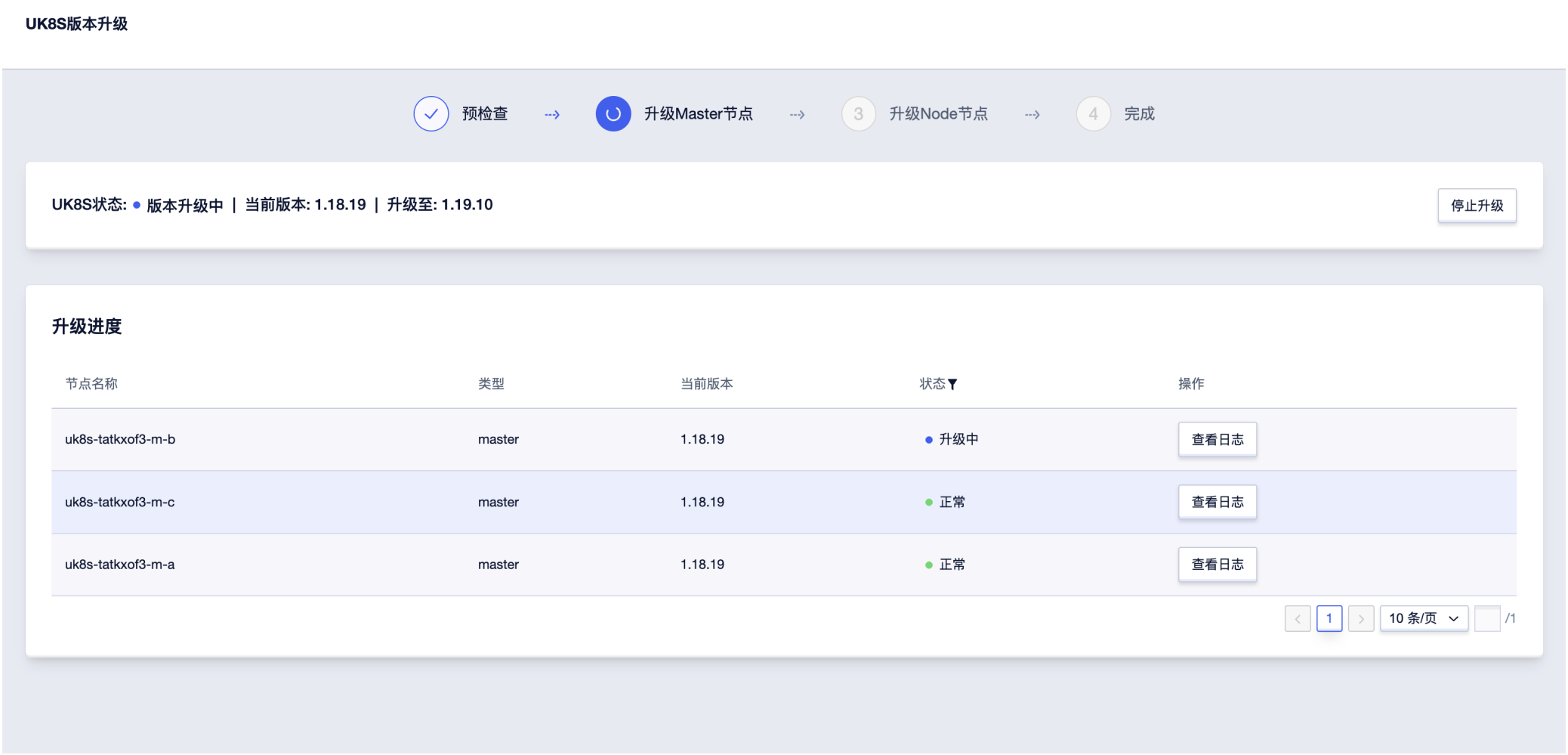
UK8S版本升级



升级集群耗时根据节点不同时间也有所不同，例如 10 台节点的集群约耗时 10 分钟。升级过程您可以关闭该页面，并随时可以通过点击集群列表中的【版本升级】再次进入查看升级进度。

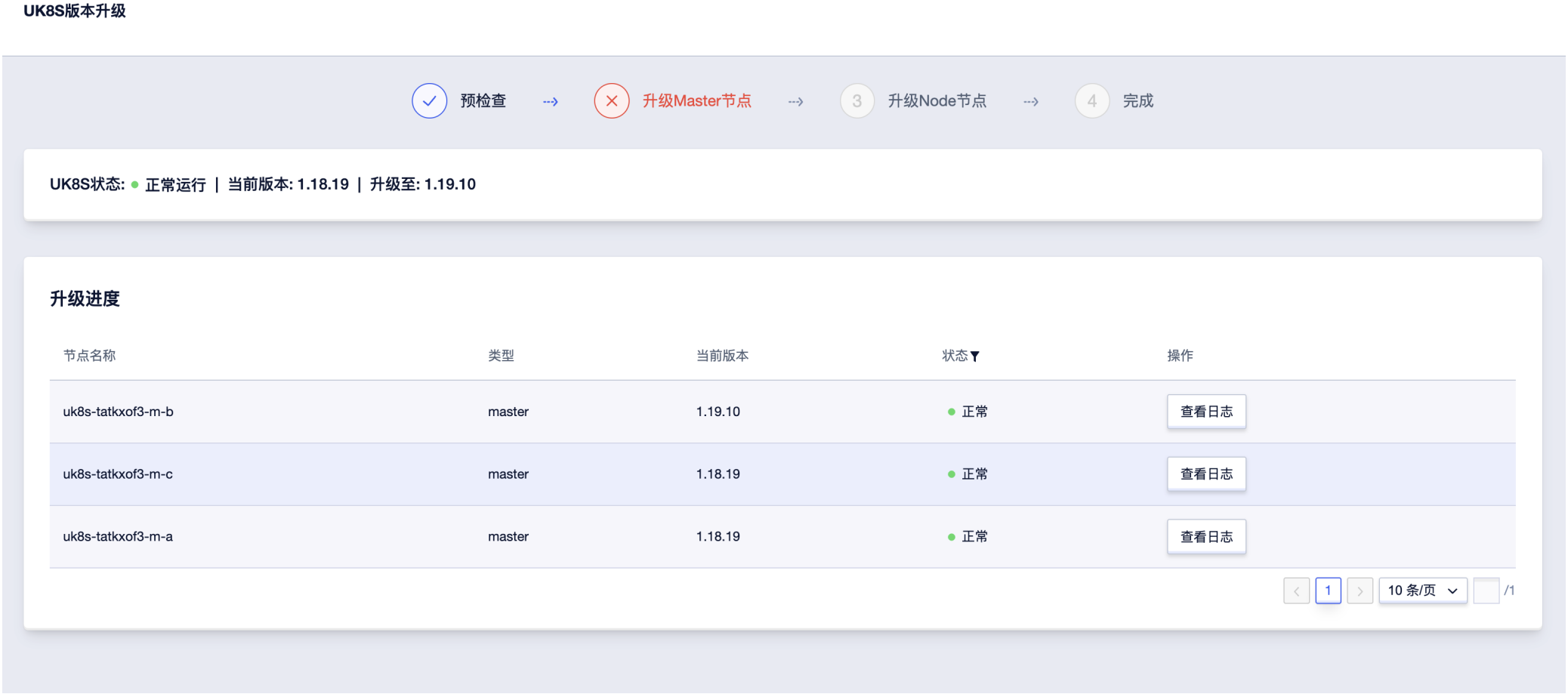
升级暂停

为了支持大规模集群升级操作过程中需要暂时停止升级，UK8S 控制台提供了【停止升级】的功能，给用户提供了更多手段控制复杂的集群升级操作。您可在升级过程中按下图所示的界面，点击【停止升级】按钮，暂停 UK8S 集群升级操作：



点击此按钮后系统将提示以下内容：

如果确实需要停止升级操作，请按【确定】按钮。系统将在当前批次的节点完成升级操作后，停止升级其它未开始的节点。下图是集群升级停止后的界面展示样例：



根据上图，您可以观察到集群中的部分 master 节点已经升级到目标版本。其它 master 节点还是原来的版本。如果您需要恢复升级，可以通过点击集群列表中的【版本升级】再次进入升级界面。

升级失败

如遇升级失败，可以在集群列表页面点击操作【更多】按键可以看到【版本升级】下载升级日志，如是您在升级期间对集群有操作导致的失败您可以根据日志提示修改后，重新进行升级操作。

如您对于日志所述内容无法进行判断，请您与我们技术支持联系，对集群进行诊断后重新进行升级。

常见升级失败原因

- 集群节点处于关闭或 kubelet 运行状态不正常
- 集群节点无法正常 SSH，密码错误等
- 集群节点可以 SSH，但无 Root 权限
- 集群伸缩插件没有关闭，导致新加入集群的节点无权限操作

集群常见问题

1. 集群详情页提示ApiServer自签https证书过期

过期的是什么证书

apiserver-loopback-client 证书，用于 kube-scheduler、kube-controller-manager 等管理组件和 kube-apiserver 之间的同节点通信。证书过期会影响管理组件之间的通信，可能导致无法正常创建Pod等问题。

可参考k8s社区官方对这个问题的解释

如何查看证书

apiserver-loopback-client 存放于 kube-apiserver 的内存中，在服务启动时自动生成，没有写入到文件中。查看证书的方法如下:

登录**master**节点执行

```
curl --resolve apiserver-loopback-client:6443:127.0.0.1 -k -v https://apiserver-loopback-client:6443 2>&1| grep -i 'server certificate' -A5
```

如何解决

逐一登录master节点，重启kube-apiserver服务(systemctl restart kube-apiserver)，重启不会影响线上业务，需注意点：

1. 重启期间不能有业务发布变更等操作
2. 逐一执行重启，不可两台及以上master同时重启。

托管版UK8S用户无法自行重启apiserver, 请联系UK8S团队

集群节点配置推荐

1. Master 配置推荐

Master 规格跟集群规模有关，集群规模越大，所需要的 Master 规格也越高，不同集群规模的，Master 节点配置推荐如下：

节点规模	Master规格
1-10 个节点	>=2 核 4G
10-100 个节点	>=4 核 8G
100-250 个节点	>=8 核 16G
250-500 个节点	>=16 核 32G
500-1000 个节点	>=32 核 64G
1000 个以上节点	联系我们

UK8S Master 节点系统盘默认 40G（最小值），用于储存 ETCD 信息及相关配置文件等。

如随着集群规模提升，有升级 Master 节点规格配置需求，请在云主机节点管理页面，逐台进行更改配置（每一台都要升级到相同配置）。

在升级下一台 Master 节点前，请确保其它两台 **Master** 节点已处于 **Ready** 状态，且 Master 节点上 Kubernetes 相关核心组件状态均处于 **active** 状态。Master 节点核心组件排障方法请参考：Node 常见故障处理

2. 如何选择 Node 配置大小

UK8S 集群要求 Node 配置不小于 2C4G，系统盘默认 40G（最小值），用于储存相关配置文件等等。

关于 Node 节点的资源预留策略

在确定 UK8S 的 Node 节点配置之前，您需要知道，UK8S Node 默认预留 **1G 内存，0.2 核 CPU** 以保障系统稳定运行。这些预留的资源是给系统及 Kubernetes 相关服务进程使用的。

且当可用内存小于 5% 时会根据 pod 资源优先级开始驱逐，pod 实际可使用的内存约为 {Memeroy of Node}-1G-5% (例如：4G内存，可用约为2.8G)，同时，单个节点可创建 Pod 和 Node CPU 核数有关。Pods 数量 = CPU 核数 x 8 (例如：2 核支持最多 16 pods，4 核支持最多 32 pods)。

因此，我们建议 Node 的配置 >= 2C4G，这是保证集群正常运行的基础配置。

对于存储资源，UK8S 的 Node 节点系统盘。节点创建时可选择数据盘挂载（亦可在节点创建完成后在主机侧挂载），如节点挂载有数据盘，将用于 Docker 存放本地镜像的，否则本地镜像等将保存在系统盘。请保证该盘磁盘空间充足，避免触发空间不足导致的镜像或 Pod 自动清理。

生产环境 Node 配置选项建议

生产环境的 Node 配置选项，可根据整个集群的日常使用的总核数以及故障容忍度、业务类型综合考量，具体如下：

假设集群总核数设定为 240 核（基于过往运营数据而定），可以容忍 10% 的错误。那么可以选择 10 台 24 核 UHost，并且高峰运行的负荷不要超过 240 * 90% = 216 核。如果容忍度小于 5%，那么可以选择 15 台 16 核的 UHost，这样就算有一台节点出现故障，剩余节点仍可以支持现有业务正常运行（工作负载自动迁移）。

从提供错误容忍度的角度看，节点配置越低，节点会更多，那可用性也会相应地提高。但也存在另外两个弊端，一是需要预留给 K8S 的资源过多，造成浪费；二是不便于容器调度，甚至会导致部分容器无法被注册。一个极端的例子，3 台 8 核的节点，可创建 6 个需要预留4核的Pod，但 12 台 2 核的节点，却无法响应一个需要预留 4 核资源的Pod请求。

综合资源有效利用率、错误容忍度两个因素，在不考虑业务混合部署、业务总体规模大小的情况下，我们建议生产环境的 Node 节点 CPU 应该介于 4 核至 32 核之间。

至于 CPU:Memory 比例，建议根据自身的应用类型申请合适的机型，例如 CPU 密集型的业务可以申请 1:2 的机型，Java 类的应用可以申请 1:4 或 1:8 的机型，如果是不同业务混合部署，最好给 Node 打好标签，配合nodeAffinity 节点亲和性合理调度Pod。

节点池

概述

UK8S引入了节点池概念，通过该功能可以更方便地对节点进行分组管理。本文档将介绍节点池的创建、管理等功能，帮助您更好地利用节点池功能来管理和优化您的Kubernetes集群。

默认节点池

创建集群时，所有 Master 节点进入默认节点池(名称为`DefaultNodeGroup`)，Node 节点不进入任何节点池

默认节点池唯一，不能修改、删除以及后续增加节点，以下类型节点均不加入任何节点池：

- 所有不选择节点池的主机添加，包括添加新节点及已有主机
- 物理云主机、裸金属云主机等异型云主机

创建节点池

要创建一个节点池，您需要执行以下步骤：

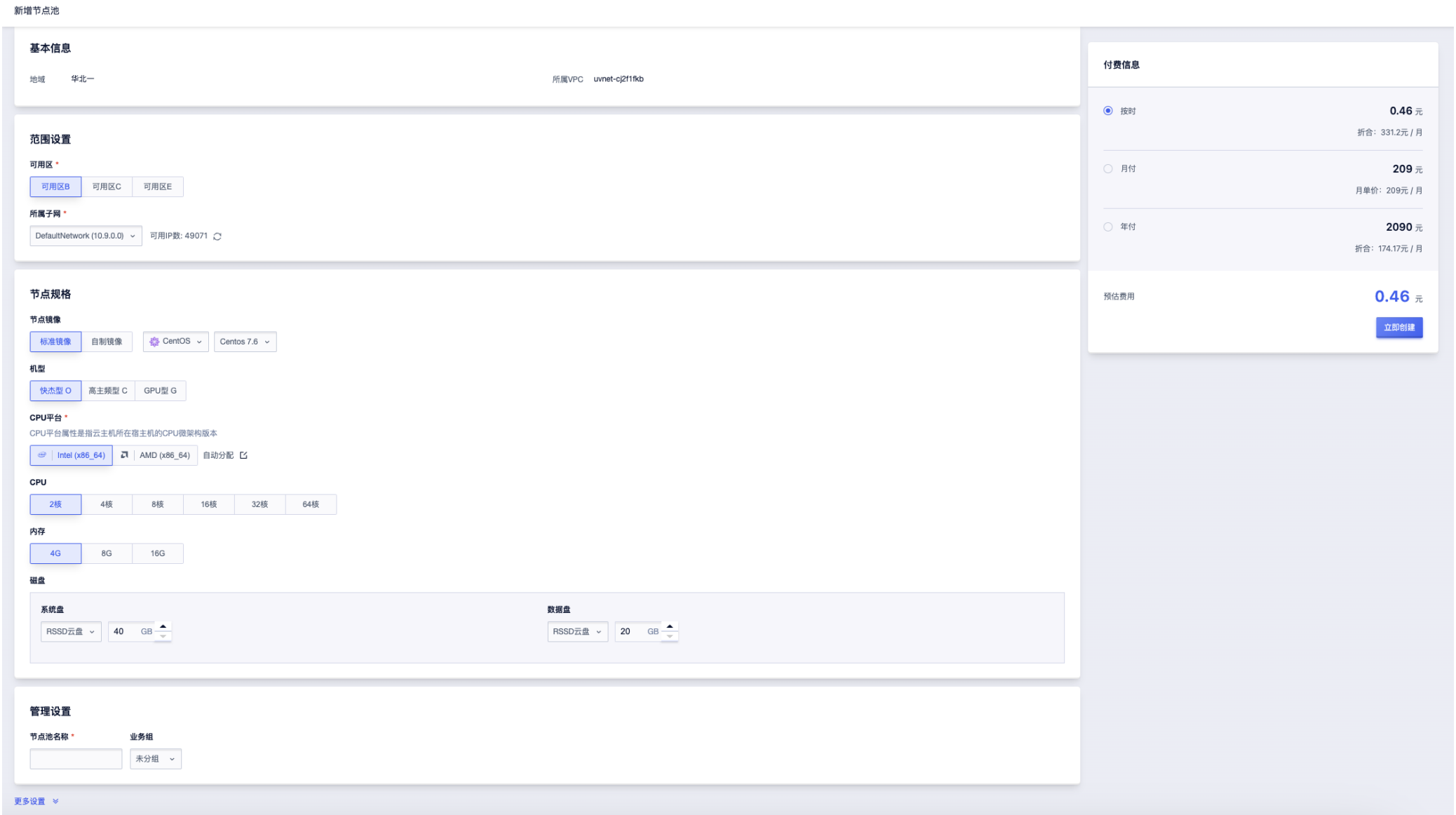
1. 登录到 UCloud 控制台，并导航到 UK8S 服务。

2. 在 Kubernetes 服务页面，选择您要创建节点池的集群。

3. 在集群详情页面，选择“节点池”选项卡。

4. 点击“新增节点池”按钮。

5. 在创建节点池页面，填写节点池的名称、节点主机信息、规格等信息。

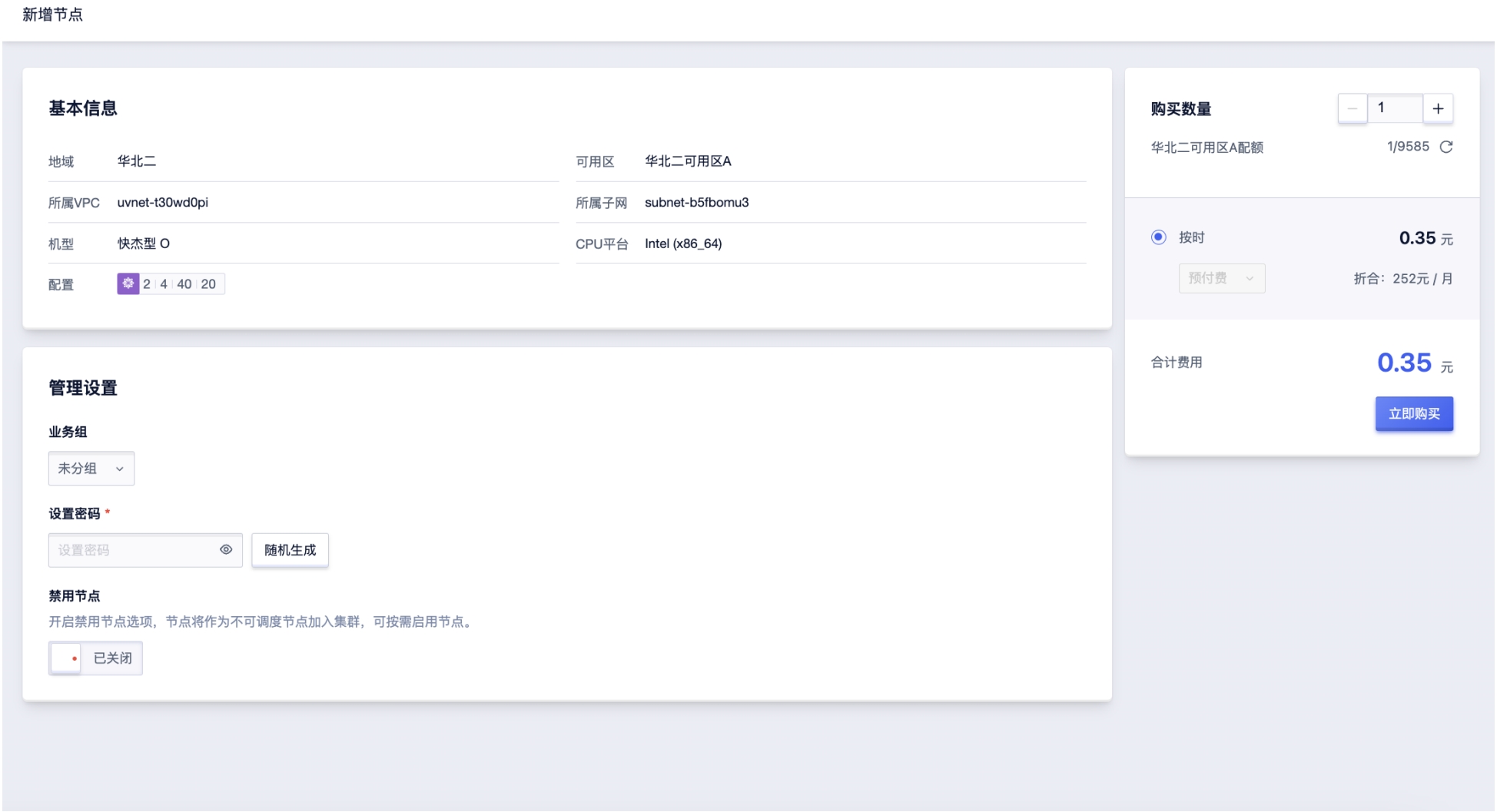


6. 点击“创建”按钮，等待节点池创建完成。

管理节点池

一旦节点池创建完成，您可以执行以下操作来管理节点池：

- 节点池增加节点：您可以根据应用程序的负载情况，手动扩容增加节点池节点已满足业务需求
  - 通过节点池添加节点，只需几步即可完成，不再需要重复选择通用的主机规格



- 删除节点池：如果您不再需要某个节点池，可以选择删除它；删除节点池时请确保该节点池下的所有节点已被移除，若仍有节点存在则不可删除。
  - 目标节点池下有节点时，移除操作为禁止状态

- **目标节点池下无节点时，移除操作可用**

添加 Node 节点

单个集群的 Node 节点上限为 5000 个，在控制台页面一次最多可添加 50 个 Node 节点。在集群详情页，节点列表页，点击**新增 Node 节点**，添加 Node 节点。

或者进入 **节点池** 页面，选择一个节点池，点击 **增加节点** 按钮在节点池增加节点

配置项	描述
所属子网	设置初始节点及 Pod 所处的子网，集群中 Node 可处于同一 VPC 下的不同子网
类型	Node 节点类型，支持 UHost 云主机（包括 GPU 云主机）
可用区	Master/Node 节点所在可用区，在具有多个可用区的地域可以选择多可用区 UK8S 集群，建议在创建集群时将 Master 节点分布于多个可用区。
节点镜像	设置集群节点的 UHost 镜像，您可以选择自定义镜像，但必须基于 UK8S 标准镜像制作，请参考制作自定义镜像 若您要使用GPU节点，镜像选择参考GPU节点中的镜像说明，CPU机器镜像可选择：Centos 7.6,Ubuntu 20.04,Anolis 8.6镜像中的任意一种。
节点规格	包括机型、CPU 平台、CPU、内存、系统盘类型、数据盘类型、数据盘大小等配置，不同类型机型配置详见：云主机 UHost / GPU云主机 / 物理云主机 UPHost / 裸金属服务器
硬件隔离组	Master 节点默认位于同一硬件隔离组，硬件隔离组能严格确保组内的每一台云主机都落在不同的物理机上。每个隔离组在单个可用区至多可以添加 7 台云主机，详见硬件隔离组
最大 Pod 数	单个 Node 节点可支持承载的最大 Pod 数量
标签	Node 节点标签，详见 Kubernetes 官方文档：标签和运算符
污点	Node 节点污点，详见 Kubernetes 官方文档：污点和容忍度

禁用节点	开启禁用节点(cordon 节点)选项，节点将作为不可调度节点加入集群，可按需启用节点。
自定义数据	是指主机初次启动或每次启动时，系统自动运行的配置脚本，该脚本可由控制台 API 等传入元数据服务器，并由主机内的 cloud-init 程序获取，脚本遵循标准 CloudInit 语法。该脚本会阻塞 UK8S 的安装脚本，即只有该脚本执行完毕后，才会开始K8S相关组件的安装，如Kubelet、Scheduler等。
初始化脚本	该脚本只在 UK8S 启动后执行一次，且是在 K8S 相关组件安装成功后执行。遵循标准shell语法，执行结果会存入到 `/var/log/message/` 目录下。

节点操作

增加节点后，可以在集群页面的节点列表页面查看节点列表。并且对节点进行：“禁用”、“删除”、“详情”、“绑定EIP”、“修改外网防火墙”等操作

uk8s-18y72s46fx6f-m-c

节点信息

节点名称	10.60.112.106
创建时间	2025-02-28
标签	查看
注释	查看
污点	查看

服务信息

服务商ID	UCloud://cn-wlc...
内网地址	10.60.112.106
容器运行时版本	containerd://1.6.10
kubelet版本	v1.28.15
kube-proxy版本	v1.28.15

节点分配

CPU	400m（requests） 1（limits）
内存	512Mi（requests） 2Gi（limits）
POD分配数量	2/110

Pods列表

文档

名称	命名空间	重启次数	内存	状态	创建时间
csi-ufile-flpmh	kube-system	0	256Mi（requests） 1Gi（limits）	Running	2025-02-28
csi-udisk-2hxfm	kube-system	0	256Mi（requests） 1Gi（limits）	Running	2025-02-28

1 10 条/页 /1

状态

类型	状态	上一次心跳	上一次变更	原因	信息
MemoryPressure	False	2025-02-28 17:23:07	2025-02-28 13:47:18	KubeletHasSufficientMemory	kubelet has sufficient memory a
DiskPressure	False	2025-02-28 17:23:07	2025-02-28 13:47:18	KubeletHasNoDiskPressure	kubelet has no disk pressure
PIDPressure	False	2025-02-28 17:23:07	2025-02-28 13:47:18	KubeletHasSufficientPID	kubelet has sufficient PID avail
Ready	True	2025-02-28 17:23:07	2025-02-28 13:47:18	KubeletReady	kubelet is posting ready status

1 10 条/页 /1

禁用

禁用是对节点执行`kubectl cordon node` 命令，使节点Unschedulable，但是不会驱逐当前节点上的Pod；

删除

删除节点时，可以勾选同时删除云主机资源和云主机上的数据盘，如果不勾选则仅从集群内把节点删除；

建议：先驱逐节点上服务再做删除节点操作，确保服务可用性。

Virtual Kubelet 虚拟节点

Virtual Kubelet 是 Kubernetes 社区的重要开源项目，基于 Virtual Kubelet 虚拟节点组件，可以实现 UCloud 两大容器产品 UK8S 和 Cube 的无缝对接，用户在 UK8S 集群中可通过 VK 组件创建 Cube 容器实例，每个 Cube 实例被视为 VK 节点上的一个Pod。

> 注意：新建虚拟节点不收取费用，虚拟节点上运行的Pod会收取实时费用；虚拟节点是Serverless的，背后由UCloud海量物理机资源(即Cube)支撑，因此无法登陆虚拟节点或是查看虚拟节点本身监控，但可查看虚拟节点上面Pod的监控

添加虚拟节点

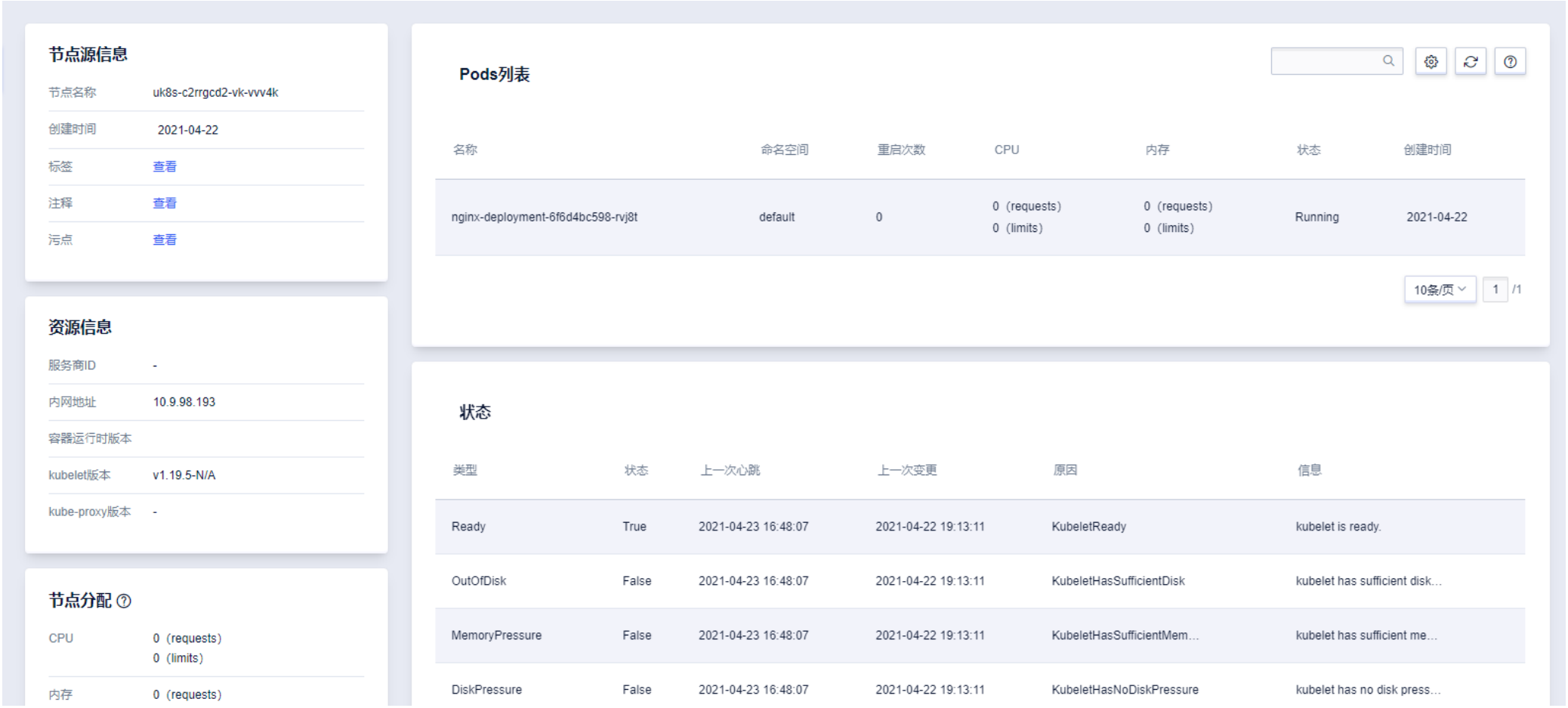
在集群节点列表页，点击「添加虚拟节点」按钮，为 UK8S 集群添加虚拟节点，当前一个 UK8S 集群支持最多添加 8 个虚拟节点，所添加的虚拟节点名称为 **uk8s-xxxxxxxx-vk-xxxxx**，该名称将被注册为虚拟节点的 **spec.nodeName**，其中 **uk8s-xxxxxxxx** 为 UK8S 集群 ID，末五位为随机生成的数字字母组合。

字段	说明
地域	VK 节点所属地域，即 UK8S 集群所在地域，不可更改。
所属子网	VK 节点及生成的 Cube Pod 所在子网，默认为 UK8S 集群 Master 节点所在子网。
可用区	VK 节点及生成的 Cube Pod 所在可用区， 当前 Cube 支持可用区：华北（北京）E，广州B。
Pod 默认配置	不指定资源 requests 情况下，VK 节点生成的 Cube Pod 的默认资源配置。
节点最大 Pod 数	节点最大可以创建的 Cube Pod 数量，当前支持最多 200 个 Cube Pod。
Cluster IP 支持	虚拟节点生成的 Cube Pod 可与 UK8S 中 Pod 通过 Cluster IP 互相访问。 当 Cube 急剧扩容时，开启该功能会导致 UK8S ApiServer 压力急剧上升。对无需使用 K8S Service 转发能力的容器，建议不开启该功能。

虚拟节点管理

节点描述

VK 节点与普通 Node 节点一样，是 UK8S 集群当中的一个 Node 对象。命令行管理时可使用 `kubectl get nodes` 等命令进行节点的管理、查看，在集群列表页点击「节点描述」按钮，亦可查看 VK 节点详细信息、节点状态、节点生成的 Cube Pod 及节点事件。



禁用及删除

您可以在控制台页面，进行 VK 节点的禁用和删除，禁用 VK 节点后应用将不能够通过 VK 节点创建 Cube 实例，现有通过 VK 节点创建的 Cube 实例将被保留。删除 VK 节点时，通过 VK 节点创建的 Cube 实例及所挂载的UDisk 将会被默认删除。

通过虚拟节点创建 Cube 实例

通过 VK 节点创建 Cube 实例的方式，与普通 Pod 资源类似，但需要在 yaml 文件 Pod spec 中添加 nodeName 或 nodeSelector 指定 VK 节点并添加污点容忍。支持直接创建 Pod，或通过 Deployment 及 StatefulSet 等控制器进行 Pod 的管理。

```
apiVersion: v1
```

```
kind: Pod
metadata:
name: nginx
spec:
# 如pod无需和api server通信，则建议不挂载token。在pod spec中加入以下字段跳过挂载
# 如果必须使用service account token, 参考下方关于「1.22版本service account token自动挂载问题」的说明
automountServiceAccountToken: false
# 通过 nodeSelector 调度到 VK 节点，nodeName 及 nodeSelector 只需配置一项
# 如以 Deploy 及 Sts 形式创建，请务必使用 nodeSelector
# 虚拟节点创建后，也可为虚拟节点添加指定标签，用于 Pod 调度的管理
nodeSelector:
type: virtual-kubelet
# 通过指定 nodeName 调度到 VK 节点
# nodeName: uk8s-xxxxxxxx-vk-xxxxx
# 添加节点容忍
tolerations:
- key: virtual-kubelet.io/provider
operator: Equal
value: ucloud
effect: NoSchedule
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
# 通过以下参数指定Pod资源配置 (如果不使用默认配置)，具体支持规格参考下方
resources:
```

```
requests:
cpu: "2"
memory: 2048Mi
```

创建 Cube 实例时，需注意特定 CPU / 内存规格有一定的比例及限制，VK 支持创建 Cube 规格配置如下：

CPU	内存
500m	512Mi/1Gi/2Gi
1	1Gi/2Gi/4Gi
2	2Gi/4Gi/8Gi
4	4Gi/8Gi/16Gi
8	8Gi/16Gi

详细注释说明

通过 VK 插件创建 Cube 时，支持在 Pod annotations，或 Deployment / Statefulset 的 spec.template.annotations 中添加相应字段，为 Cube 进行进一步配置，相应注释说明如下：

注释	参数类型	注释说明	默认值
cube.ucloud.cn/cube-tag	string	需要指定的业务组的名称	/
cube.ucloud.cn/cube-chargetype	year/month/postpay	Cube 付费模式，即按年预付费 / 按月预付费 / 按秒后付费，请参照计费说明	postpay

cube.ucloud.cn/cube-quantity	int	Cube 付费时长，月付时，此参数传 0，代表购买至月末	1
cube.ucloud.cn/cube-eip	true/false	是否需要绑定 EIP	false
cube.ucloud.cn/cube-eip-id	eip-xxxxxxxx	绑定指定 ID 的 EIP	/，仅在 cube.ucloud.cn/cube-eip: "true" 时生效， 如该项留空，则创建新的 EIP
cube.ucloud.cn/cube-eip-paymode	traffic/bandwidth/shareband width	EIP 计费模式，即流量计费 / 带宽计费 / 共享带宽模式	bandwidth
cube.ucloud.cn/cube-eip-share-band width-id	bwshare-xxxxxx	共享带宽 ID，仅在 EIP 计费模式为「共享带宽」时生效	/
cube.ucloud.cn/cube-eip-bandwidth	int	绑定 EIP 的外网带宽大小，共享带宽模式必须指定 0	2
cube.ucloud.cn/cube-eip-chargetype	year/month/dynamic	EIP 付费模式，即按年预付费 / 按月预付费 / 按时后付费	取 cube.ucloud.cn/cube-chargetype 值； 该项为 postpay 时取 dynamic
cube.ucloud.cn/cube-eip-quantity	int	EIP 付费时长	取 cube.ucloud.cn/cube-quantity 值
cube.ucloud.cn/cube-eip-release	true/false	删除 Cube 实例时是否需要释放绑定的 EIP	true
cube.ucloud.cn/cube-eip-security-gro up-id	firewall-xxxxxxxx	需要绑定的外网防火墙策略，不指定时绑定项目默认防火墙	/

说明：1.22版本支持service account token自动挂载问题

从1.22版本开始service account token挂载默认为projected volume模式，cube不支持此类型的volume。

如pod无需和api server通信，则可以不挂载token；在pod spec中加入以下字段跳过挂载

```
automountServiceAccountToken: false
```

如果pod的运行依赖service account token，则可以显式指定以secret的方式挂载

以default sa为例:

如果是1.24及更高版本的集群，由于创建 service account 时没有自动生成相应的token，首先需要创建一个secret：

```
apiVersion: v1
kind: Secret
metadata:
name: default-token
annotations:
kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
```

1.22版本不需要以上步骤，执行 `kubectl get secret` 找到 default service account 对应的 secret名称即可。

创建deployment时，显式指定service account、volume 和 mount:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
labels:
app: nginx
spec:
replicas: 1
selector:
matchLabels:
app: nginx
template:
metadata:
labels:
app: nginx
spec:
# 指定部署到虚拟节点
nodeSelector:
type: virtual-kubelet
tolerations:
- effect: NoSchedule
key: virtual-kubelet.io/provider
operator: Equal
value: ucloud
serviceAccount: default
```

```
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
ports:
- containerPort: 80
# 显式挂载
volumeMounts:
- mountPath: /var/run/secrets/kubernetes.io/serviceaccount
name: default-token
readOnly: true
volumes:
# 显式指定secret形式的volume
- name: default-token
secret:
defaultMode: 420
secretName: default-token # 1.22版本中加上token后缀
```

UDisk 存储卷挂载支持

用户可以通过声明 PVC 存储卷的方式为 VK 节点上的 Cube 实例创建挂载 UDisk 存储卷，这部分工作由 CSI UDisk 和 Virtual Kubelet 组件共同完成，存储类、持久卷声明用法与正常在 UK8S 中使用存储卷一致（包括新建 **UDisk** 及使用已有 **UDisk**，详见：在 UK8S 中使用 UDisk）。

以下是使用例子：

```
## 创建存储类 StorageClass
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: ssd-csi-udisk
provisioner: udisk.csi.ucloud.cn
parameters:
## 建议使用 SSD 或 SATA 云盘，避免因 RSSD 云盘 RDMA 区域与 Cube 无法匹配导致云盘无法挂载
type: "ssd"
## 不支持 xfs 文件系统
fsType: "ext4"
## 回收策略，支持Delete和Retain，默认为Delete，非必填
reclaimPolicy: Delete
## 如绑定模式设置为 WaitForFirstConsumer，则只能通过 pod.spec.nodeSelector 指定 VK 节点
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true # 声明该存储类支持可扩容特性
---
## 创建持久化存储卷声明 PVC
## 不支持 Volume Expansion（存储卷动态扩容）
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: logdisk-claim
spec:
```

```
accessModes:
- ReadWriteOnce
storageClassName: ssd-csi-udisk
resources:
requests:
storage: 20Gi
---
## 在 Pod 中使用 PVC
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
tolerations:
- effect: NoSchedule
key: virtual-kubelet.io/provider
operator: Equal
value: ucloud
## 如绑定模式设置为 WaitForFirstConsumer，则只能通过 pod.spec.nodeSelector 指定 VK 节点
nodeSelector:
type: virtual-kubelet
containers:
- name: http
image: uhub.service.ucloud.cn/ucloud/nginx:latest
volumeMounts:
```

```
- name: log
mountPath: /data
volumes:
- name: log
persistentVolumeClaim:
claimName: logdisk-claim
```

使用自建镜像仓库

1. 创建自建镜像仓库

以自建hub地址 `myhub.ucloud.cn` 为例，需要额外提供的信息包括镜像仓库ip地址和所在vpc。用户名和密码非必填。

参考 https://docs.ucloud.cn/cube/userguide/self_repository

```
cat > .dockerconfigjson <<EOF
{
"auths": {
"myhub.ucloud.cn": {
"username": "user-xxx",
"password": "passwd-yyy",
"registryaddr": "10.x.y.z",
"vpcid": "uvnet-xxx"
}
}
```

```
}
EOF
kubectl create secret docker-registry myhub --from-file=.dockerconfigjson
```

2. 创建pod时指定 imagePullSecret

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-myhub-deployment
labels:
app: nginx
spec:
replicas: 1
selector:
matchLabels:
app: nginx
template:
metadata:
labels:
app: nginx
spec:
automountServiceAccountToken: false
nodeSelector:
```

```
type: virtual-kubelet
tolerations:
- key: virtual-kubelet.io/provider
operator: "Equal"
value: ucloud
effect: NoSchedule
containers:
- name: nginx
image: myhub.ucloud.cn/test/nginx:latest
ports:
- containerPort: 80
imagePullSecrets:
- name: myhub
```

虚拟节点能力限制

网络

支持使用VPC，与云主机同级；不支持 hostNetwork 、CNI、Calico Policy等功能。

Pod存储

容器支持10GiB的临时存储空间，对临时存储的修改在Pod重启后会丢失，如有需要推荐挂载持久化存储。

持久化存储：支持挂载UDisk、NFS、ConfigMap、Secret，不支持其他类型的volume，例如 hostPath ，projected volume 等。注意：挂载ConfigMap和Secret时，总配置大小不能超过

30MiB。

镜像仓库

支持拉取UHub以及同地域下的自建镜像仓库(仅使用UHost自建仓库)，不支持拉取外网镜像，如DockerHub；并且镜像大小不能超过10GiB。

其他功能

支持日志，`exec`，监控等基本功能；可以使用Deployment、StatefulSet等Kubernetes控制器进行控制。

其他与Kubernetes原生Pod兼容性一致。

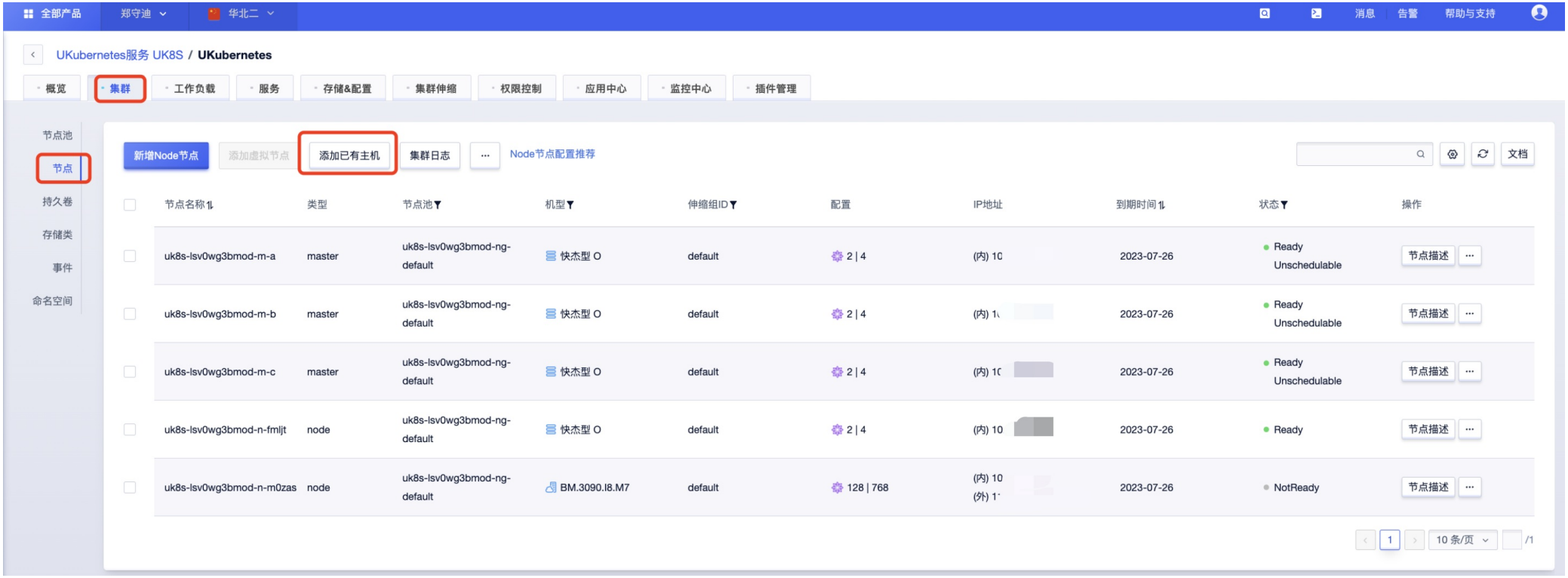
添加已有主机-云主机

UK8S 产品不仅支持增加新节点，也支持把已经创建好的云主机加入到集群内。云主机在加入集群时会按用户选择的镜像进行重装；具体云主机的创建可参考这里

操作指引

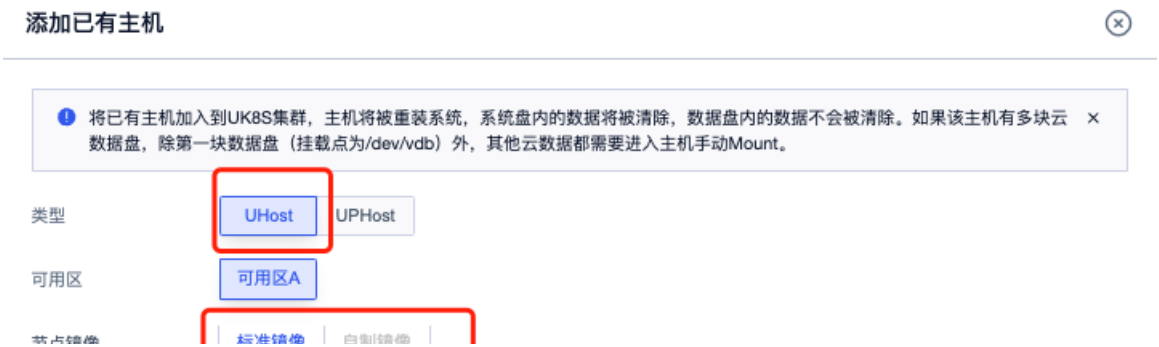
一、在集群列表页点击详情，进入集群详情页

二、进入集群节点页



三、点击添加已有主机

选择UHost，勾选需要的镜像与添加的机器列表。



要根据要选择的云主机机型来选择对应节点镜像，选择错误的镜像，可能会导致加入集群节点无法正常使用；

- CPU机器：镜像可选择：Centos 7.6,Ubuntu 20.04,Anolis 8.6镜像中的任意一种。
- GPU机器：镜像选择参考GPU节点中的镜像说明，

如果加入集群的节点所需要的节点镜像是多种类型，可通过多次加入的方式进行加入集群。

镜像选择后勾选需要添加的云主机，点击“>”按钮，加入已选择框，然后填写其他节点配置与管理员密码；

四、确定加入成功

所有信息填写后，点击“确定”按钮即可把节点自动加入集群。

添加已有主机-裸金属

目前部分地域（如乌兰察布，上海）开放支持已有裸金属资源加入集群；在集群详情页，节点列表页，点击添加已有主机，类型选择UPHost，即可添加已有裸金属资源。

具体支持情况如下：

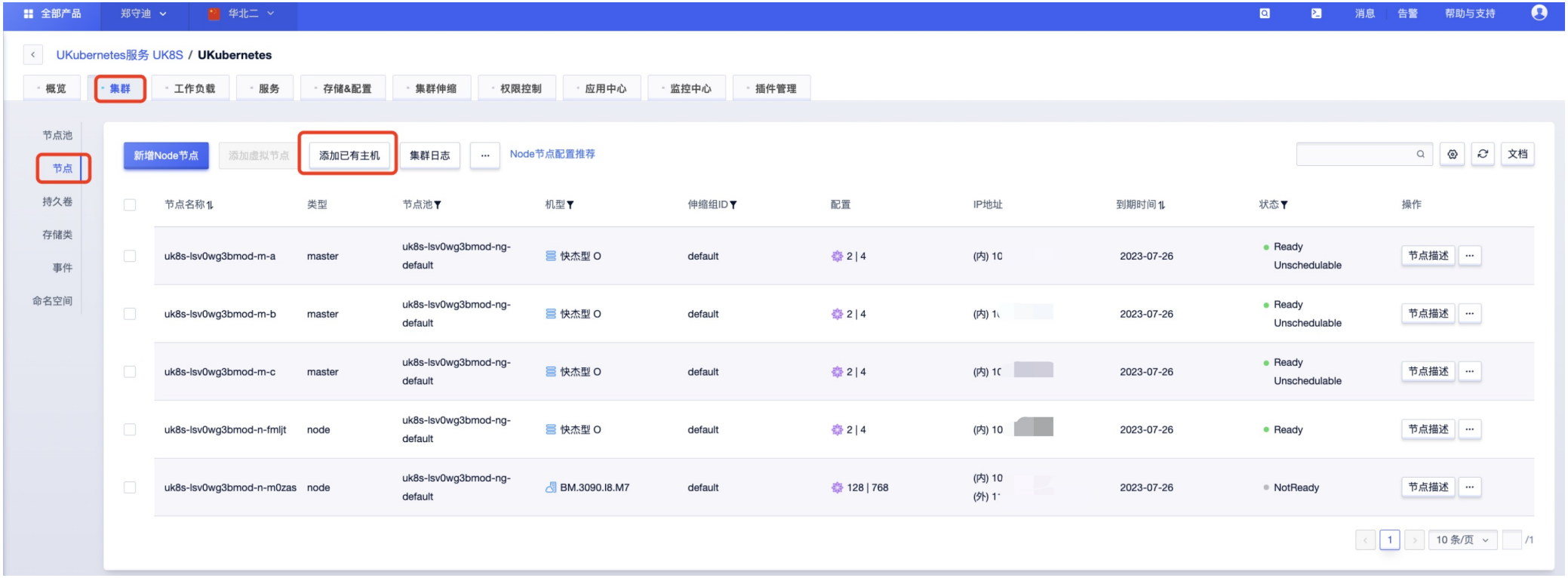
> 若有需求可以先找产品确认适配情况

- 支持裸金属机型：
  1. CPU裸金属：公有云部分标准云盘裸金属、本地盘裸金属机型
  2. GPU裸金属：高性能计算显卡5(BM.3090.I8.M7)
- 集群版本要求：1.20.6及以上
- CSI版本要求：23.07.24及以上
- CNI版本要求：v1.2.0及以上

操作指引

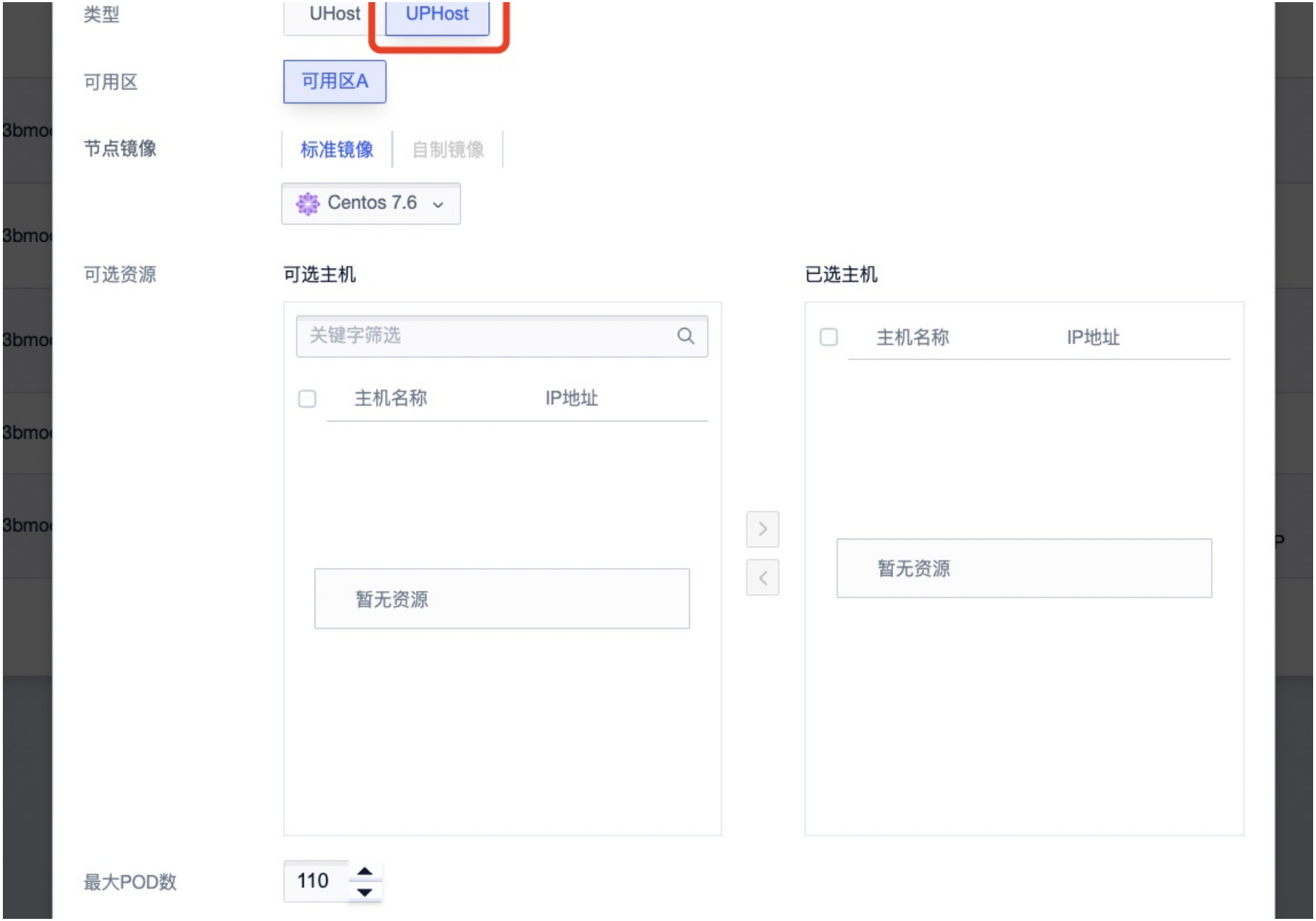
一、在集群列表页点击详情，进入集群详情页

二、进入集群节点页



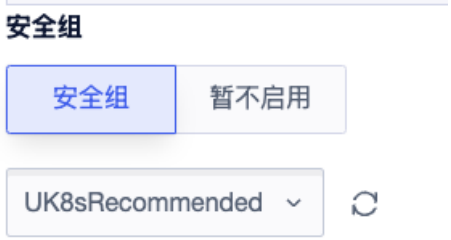
三、点击添加已有主机

选择UPHost，勾选目标裸金属资源，确认提交即可。



安全组支持

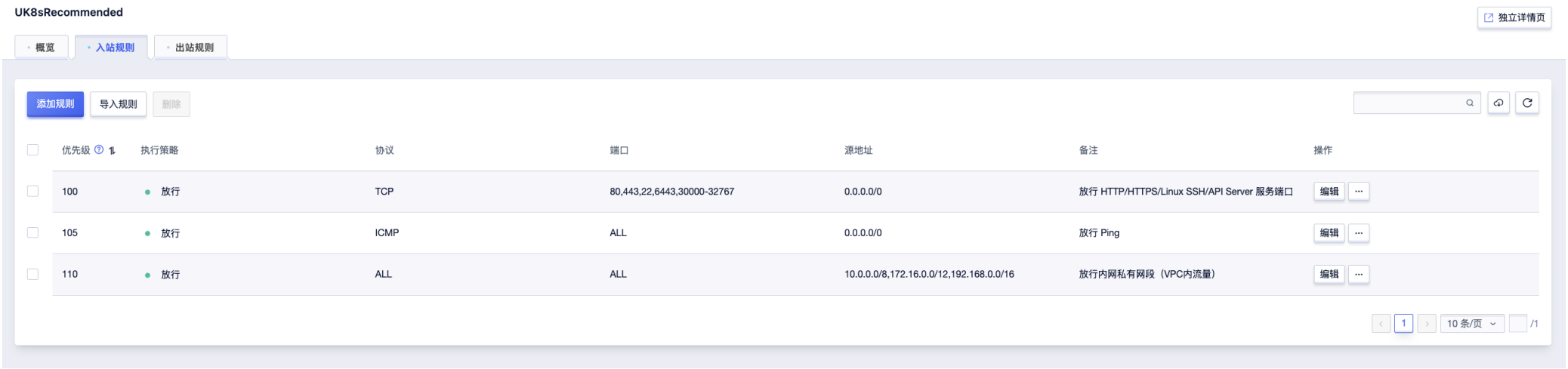
开通安全组的用户在创建集群和新增节点时，会有如下选项框：



如果选择暂不启用，则创建防火墙类型的节点，跟原先一致

如果选择安全组选项，则会选择安全组，此处可以选择的安全组的规则模板必须为UK8S模板，参考如下（图来自安全组产品页面下新建安全组）：

规则模板中对应的安全组规则如下：



如果需要使用其他类型的规则模板，则可以将目标节点的主机在安全组页面上手动绑定安全组（对于已有节点也可以这样进行操作），或者使用该模板创建云主机uhost正常启动后将其作为添加已有主机加入集群

针对这种修改了安全组配置的情况，也请尽可能保证将UK8S模板中的规则应用到您的自定义安全组规则下，否则可能会影响集群正常使用，

由于启用了安全组后网络策略变为白名单模式，所以请勿在安全组页面上将uk8s的节点的所有安全组规则解绑，这可能会导致uk8s集群网络异常

制作自定义镜像

一、前言

为了满足用户对 UK8S 节点的个性化需求，除了标准镜像外，UK8S 的 Node 节点也支持自定义镜像。但务必使用 UK8S 的标准镜像来制作自定义镜像，否则可能导致集群无法创建或节点无法添加。

下面介绍如何基于标准镜像制作自定义镜像，以及注意事项。本文档介绍的自定义镜像制作过程是完全自动化的，制作过程中无需人工干预。用户需要具备简单的 shell 编程或 Ansible 使用经验。

由于香港可用区到国内和全球其它可用区的网速较快，进行镜像复制时可以减少耗时。本文介绍的方法是在香港地域进行镜像构建，然后复制到其它可用区。请保证在香港可用区有足够的云主机配额。

二、制作自定义镜像流程

1. 安装 Packer

安装 Packer 工具，使用该工具可以方便地创建并分发自定义镜像到你需要的可用区。下面介绍了macOS 安装方式，其他环境请参考Packer官方文档

macOS 用户可以通过以下命令安装 Packer：

```
brew install packer
```

Packer 只负责创建云主机，在云主机中安装配置软件需要使用命令行脚本或 Ansible。本文档给的示例使用 Ansible，也可以转换成其它等价工具。下面介绍了Ansible 的macOS 安装方式，其他环境请参考Ansible 官方文档。

macOS 用户可以通过以下命令安装 Ansible：

```
brew install ansible
```

2. 准备公钥、私钥及项目 ID

请在 UCloud 控制台的账号管理 -> API 密钥创建或使用现有的公钥及私钥。请在 UCloud 控制台的访问控制 -> 项目管理中查找到存放即将创建的自定义镜像所属项目。请将公钥、私钥和项目 ID 设置到环境变量中，命令示例如下：

```
export UCLOUD_PUBLIC_KEY="公钥"
export UCLOUD_PRIVATE_KEY="私钥"
export UCLOUD_PROJECT_ID="项目 ID"
```

以上命令建议设置到 shell 的初始化文件中，如.zshrc 或.bashrc 等。

3. 编写 Packer 配置文件

假定该配置文件名称为 custom.json。

```
{
"variables": {
"ucloud_public_key": "{{env `UCLOUD_PUBLIC_KEY`}}",
"ucloud_private_key": "{{env `UCLOUD_PRIVATE_KEY`}}",
"ucloud_project_id": "{{env `UCLOUD_PROJECT_ID`}}"
},
```

```
"builders": [{
"type": "ucloud-uhost",
"public_key": "{{user `ucloud_public_key`}}",
"private_key": "{{user `ucloud_private_key`}}",
"project_id": "{{user `ucloud_project_id`}}",
"region": "hk",
"availability_zone": "hk-02",
"instance_type": "o-standard-2",
"source_image_id": "<REPLACE_THE_UK8S_BASE_IMAGE_ID_HERE>",
"ssh_username": "root",
"image_name": "<YOUR_IMAGE_NAME_GOES_HERE>",
"image_copy_to_mappings": [
{
"project_id": "{{user `ucloud_project_id`}}",
"region": "<REPLACE_REGION_ID_WHERE_TO_COPY>"
}
]
}],

"provisioners": [{
"type": "ansible",
"playbook_file": "./playbook.yml"
}]
```

```
}
```

请先将上述例子中尖括号中的内容替换成实际的值。需要注意到builders中几个参数：

- type: 这是Packer中对应的插件名称，无需改动。
- region: 表示REPLACE_THE_UK8S_BASE_IMAGE_ID_HERE镜像所在地域，这里建议选择香港，如果自制镜像需要下载海外资源，香港机房可直接下载。
- availability_zone: 表示REPLACE_THE_UK8S_BASE_IMAGE_ID_HERE镜像所在可用区
- instance_type: 机器类型，可以保持不变，安装GPU驱动也可以使用默认这个机型。
- ssh_username: 如果是ubuntu镜像，请切换成ubuntu

下表是香港可用区下 UK8S 支持的操作系统及版本对应的基础镜像 ID。请按需要选择合适的镜像，并用镜像 ID 栏对应的值替代<REPLACE_THE_UK8S_BASE_IMAGE_ID_HERE>：

地域	可用区	镜像 ID	操作系统	版本	支持 GPU
hk	hk-02(3002)	uimage-puxm0l	CentOS	7.6	是
hk	hk-02(3002)	uimage-x7zezb1gcv5	Ubuntu	20.04	是

如果需要将制作完成的镜像拷贝到其它地域及可用区，可在上述文件的image_copy_to_mappings中设置目标可用区，可以同时指定多个。如果不需要复制的话，请将该属性删除即可。

接下来需要编写安装配置自定义镜像的脚本。本文档给了一个使用 Ansible 的示例供参考。Packer还有其他的类型的provisioners，请参考Packer官方文档。Ansible对应的playbook.yml 如下：

```
- hosts: all
become: true

pre_tasks:
- name: Disable swap
```

```
command: swapoff -a
when: ansible_swaptotal_mb > 0

roles:
- role: custom-setup
```

上述例子中的 playbook 会关闭 swap，并执行命名`custom-setup`的 role 做进一步设置。请注意这个仅仅是示例，关闭 swap 实际上已经在 UK8S 的基础镜像中完成，无需再做此操作。

4. 运行 Packer

首次运行 Pakcer 时请在包含上述 `custom.json` 文件的目录下运行：

```
packer init .
```

如果上述命令运行有问题，请在运行路径下配置config.pkr.hcl文件，文件内容如下

```
packer {
required_plugins {
ucloud = {
version = ">= 1.0.8"
source = "github.com/ucloud/ucloud"
}
}
}
```

然后再运行以下命令：

```
packer build custom.json
```

创建镜像的过程比较耗时，中途请勿到云主机中进行任何操作，或者删除该主机，否则无法正常创建镜像。创建完成后，packer 会显示镜像的 ID。示例如下：

```
==> ucloud-uhost: Stopping instance "uhost-88888888888"
ucloud-uhost: Stopping instance "uhost-88888888888" complete
==> ucloud-uhost: Creating image xxxx-yyyyy-8.5...
ucloud-uhost: Waiting for the created image "uimage-**********" to become available...
ucloud-uhost: Creating image "uimage-**********" complete
==> ucloud-uhost: Copying images from "uimage-**********"...
ucloud-uhost: Copying image from org-******:cn-bj2:uimage-********** to org-******:cn-wlcb:uimage-**********
ucloud-uhost: Copying image from org-******:cn-bj2:uimage-********** to org-******:hk:uimage-**********
ucloud-uhost: Copying image from org-******:cn-bj2:uimage-********** to org-******:cn-gd:uimage-**********
ucloud-uhost: Waiting for the copied images to become available...
ucloud-uhost: Copying image complete
==> ucloud-uhost: Deleting instance...
ucloud-uhost: Deleting instance "uhost-88888888888" complete
Build 'ucloud-uhost' finished after 19 minutes 43 seconds.
```

自定义镜像制作完成后 Packer 会自动删除云主机，无需担心因忘记删除主机而产生不必要的费用。

三、注意事项

UK8S 基础镜像预先配置好了部署 Kubernetes 的依赖项，如软件、文件目录、内核参数等。在基于 UK8S 基础镜像制作自定义镜像时，请谨慎修改相关配置，以免基于自定义镜像创建节点时失败。下面简要说明制作自定义镜像过程中的注意事项。

3.1 系统相关

1. 默认禁用了 swap，**请勿开启；**
2. 配置了 journald 参数 Storage=persistent，**不建议修修改；**
3. 默认创建了以下目录，**请勿删除或修改；**

- /etc/kubernetes/ssl
- /etc/etcd/
- /etc/docker/
- /etc/kubelet.d/
- /var/lib/kubelet
- ~/.kube/
- /var/lib/etcd/
- /var/lib/etcd/default.etcd
- /usr/libexec/kubernetes/kubelet-plugins/volume/exec/ucloud~flexv/
- /etc/kubernetes/yaml

4. 加载了 ip_conntrack 模块，**请勿修改；**
5. 默认禁用了 IPV6，**请勿修改**
6. 对于龙蜥操作系统 8.x 版本，必须关闭 firewalld，**自定义镜像时请勿开启**

3.2 软件部分

UK8S 节点初始化依赖以下软件（部分），**请勿卸载**。

- iptables

- ipvsadm
- socat
- nfs-utils（用于挂载 ufs）
- conntrack
- earlyoom

UK8S 节点初始化时，会将预先生成的证书、配置文件、二进制文件（kube-proxy、kubelet、scheduler、docker、kubectl 等）拷贝到节点，并依次启动。因此在制作自定义镜像时，无需安装 K8S 相关组件。即使安装也不会被使用到，但可能干扰 UK8S 管理程序而导致创建集群失败。

自定义数据及初始化脚本

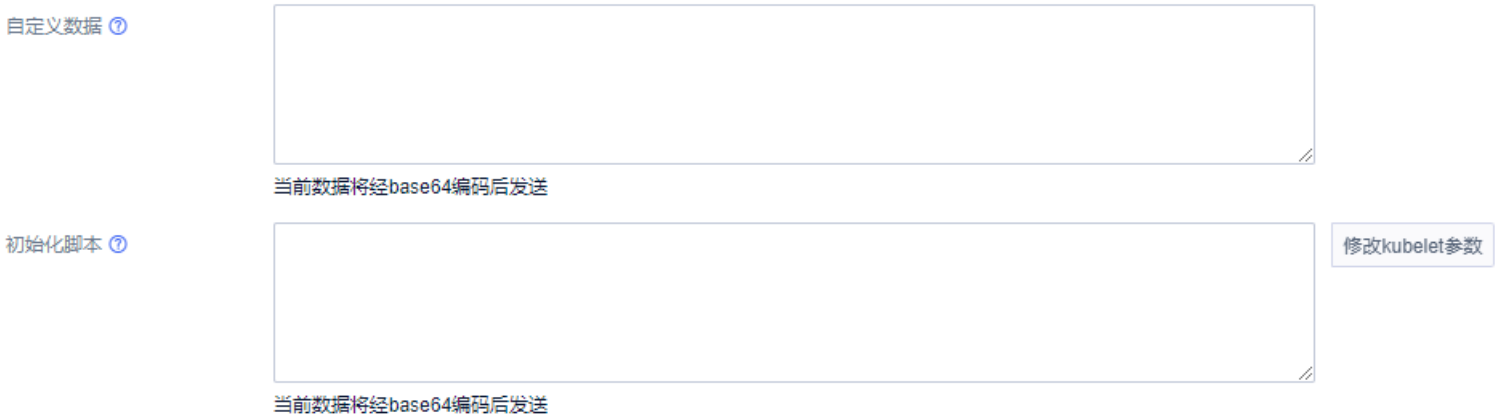
自定义数据

自定义数据是指主机初次启动或每次启动时，系统自动运行的配置脚本，该脚本可由控制台/API等传入元 数据服务器，并由主机内的cloud-init程序获取，脚本遵循标准CloudInit语法。该脚本会阻塞UK8S的安装脚本，即只有该脚本执行完毕后，才会开始K8S相关组件的安装，如Kubelet、Scheduler等。

初始化脚本

该脚本只在UK8S启动后执行一次，且是在K8S相关组件安装成功后执行。遵循标准shell语法，执行结果会存入到/var/log/message/目录下。

用户可以通过自定义数据和初始化脚本在创建时对集群进行自定义安装自有服务，比如内核修改、磁盘监控等。详细使用方法



注意事项

使用初始化脚本修改 `/etc/sysctl.conf` 时，请勿修改以下参数，会影响创建的集群正常使用。

```
net.ipv4.tcp_tw_reuse = 1
net.ipv4.conf.eth0.proxy_arp = 1
net.ipv4.ip_forward = 1
vm.max_map_count = 262144
net.netfilter.nf_conntrack_max = 1048576
kernel.unknown_nmi_panic = 0
kernel.sysrq = 1
fs.file-max = 1000000
vm.swappiness = 10
fs.inotify.max_user_watches = 10000000
net.core.wmem_max = 327679
net.core.rmem_max = 327679
net.ipv4.conf.all.send_redirects = 0
net.ipv4.conf.default.send_redirects = 0
net.ipv4.conf.all.secure_redirects = 0
net.ipv4.conf.default.secure_redirects = 0
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.default.accept_redirects = 0
fs.inotify.max_queued_events = 327679
kernel.shmmax = 68719476736
```

```
kernel.shmall = 4294967296
net.ipv4.neigh.default.gc_thresh1 = 2048
net.ipv4.neigh.default.gc_thresh2 = 4096
net.ipv4.neigh.default.gc_thresh3 = 8192
net.ipv6.conf.all.disable_ipv6 = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.conf.all.arp_ignore = 0
net.netfilter.nf_conntrack_tcp_timeout_syn_sent = 6
kernel.pid_max = 1024000
net.ipv4.ip_local_port_range = 32768 60999
net.ipv4.tcp_tw_reuse = 1
```

Node 常见故障处理

节点作为承载工作负载的实体，是 Kubernetes 一个非常重要的对象，在实际运营过程中，节点会出现各种问题，本文简要描述下节点的各种异常状态及排查思路。

1. 节点状态说明

节点情况	说明	处理办法
Ready	True 表示节点是健康的，False 表示节点不健康，Unkown 表示节点失联	
DiskPressure	True 表示节点磁盘容量紧张，False 反之	
MemoryPressure	True 表示节点内存使用率过高，False 反之	
PIDPressure	True 表示节点有太多进程在运行，False 反之	
NetworkUnavailable	True 表示节点网络配置不正常，False 反之	

2. 节点常用命令

1. 查看节点状态

```
kubectl get nodes
```

2. 查看节点事件

```
kubectl describe node ${NODE_NAME}
```

在上述两个命令看不出端倪的时候，还可以借助Linux的相关命令来辅助判断，这个时候我们就需要登录节点，通过linux相关命令来检查节点状态。

3. 检查节点联通性

3.1 网络检查：我们可以从集群的Master节点，使用 **Ping** 命令去检查该节点的网络是否可达；3.2 健康检查：登录UCloud控制台，从云主机页面查看该节点是否处于Running状态，包括查看CPU、内存使用率，确认节点是否处于高负载；

3. K8S 组件故障检查

UK8S 集群默认为 3 台 Master 节点，K8S 核心组件在 3 台 Master 节点均有部署，通过负载均衡对外提供服务。如发现组件异常，请登录相应的 Master 节点（无法定位时逐台登录 Master 节点），并通过以下命令来查看节点中组件状态是否正常、错误原因是什么，及对异常组件进行重启：

```
systemctl status ${PLUGIN_NAME}
journalctl -u ${PLUGIN_NAME}
systemctl restart ${PLUGIN_NAME}
```

UK8S 核心组件及名称：

组件	组件名称
Kubelet	kubelet
API Server	kube-apiserver

Controller Manager	kube-controller-manager
Etcd	etcd
Scheduler	kube-scheduler
KubeProxy	kube-proxy

例如，查看 APIServer 组件状态，需要执行 systemctl status kube-apiserver。

4. UK8S 页面概览页一直刷新不出来?

1. api-server 对应的 ulb4 是否被删除（uk8s-xxxxxx-master-ulb4）
2. UK8S 集群的三台 master 主机是否被删了或者关机等
3. 登陆到 UK8S 三台 master 节点，检查 etcd 和 kube-apiserver 服务是否正常，如果异常，尝试重启服务

- 3.1 systemctl status etcd / systemctl restart etcd 如果单个 etcd 重启失败，请尝试三台节点的 etcd 同时重启
- 3.2 systemctl status kube-apiserver / systemctl restart kube-apiserver

5. UK8S 节点 NotReady 了怎么办

1. kubectl describe node node-name 查看节点 notReady 的原因，也可以直接在 console 页面上查看节点详情。
2. 如果可以登陆节点，journalctl -u kubelet 查看 kubelet 的日志，system status kubelet查看 kubelet 工作是否正常。
3. 对于节点已经登陆不了的情况，如果希望快速恢复可以在控制台找到对应主机断电重启。
4. 查看主机监控，或登陆主机执行sar命令，如果发现磁盘 cpu 和磁盘使用率突然上涨，且内存使用率也高，一般情况下是内存 oom 导致的。关于内存占用过高导致节点宕机，由于内存占用过高，磁盘缓存量很少，会导致磁盘读写频繁，进一步增加系统负载，打高cpu的恶性循环

5. 内存 oom 的情况需要客户自查是进程的内存情况，k8s 建议 request 和 limit 设置的值不宜相差过大，如果相差较大，比较容易导致节点宕机。
6. 如果对节点 notready 原因有疑问，请按照UK8S人工支持联系人工支持

概述

了解K8S中的弹性伸缩

Kubernetes在多个维度、多个层次上提供不同的组件来满足不同场景下的伸缩需求，主要如下：

类型	Pod	Node
横向伸缩	HPA(Horizontal Pod Autoscaler)	Cluster Autoscaler
纵向伸缩	VPA (Vertical Pod Autoscaler)	None

- **HPA:** 即容器水平伸缩（Horizontal Pod Autoscaler）负责Pod水平伸缩的组件，是所有伸缩组件中历史最悠久的，目前支持autoscaling/v1、autoscaling/v2beta1与autoscaling/v2beta2，其中autoscaling/v1只支持CPU一种伸缩指标，在autoscaling/v2beta1中增加支持custom metrics，在autoscaling/v2beta2中增加支持external metrics。
- **CA:** 即集群伸缩(Cluster Autoscaler)，负责Node节点水平伸缩的组件，1.0.0之后已是GA阶段（General Availability，即正式发布的版本），UK8S使用的为GA版本。
- **VPA:** 即Pod的纵向伸缩，根据Pod的资源利用率、历史数据、异常事件，来动态调整负载的Request值的组件，主要关注在有状态服务、单体应用的资源伸缩场景，目前（2019年8月26日）处于beta阶段，不推荐在生产环境使用。

另外，还有cluster-proportional-autoscaler伸缩组件，可根据集群的节点数目，水平调整Pod数目的组件，目前处在GA阶段，用来根据集群规模大小动态调整CoreDNS、Ingress等关键服务的规模。另外addon-resizer组件可根据集群中节点的数目，纵向调整负载的Request的组件，目前处在beta阶段。

下面我们主要来介绍下HPA和CA这两个最常用的伸缩组件。

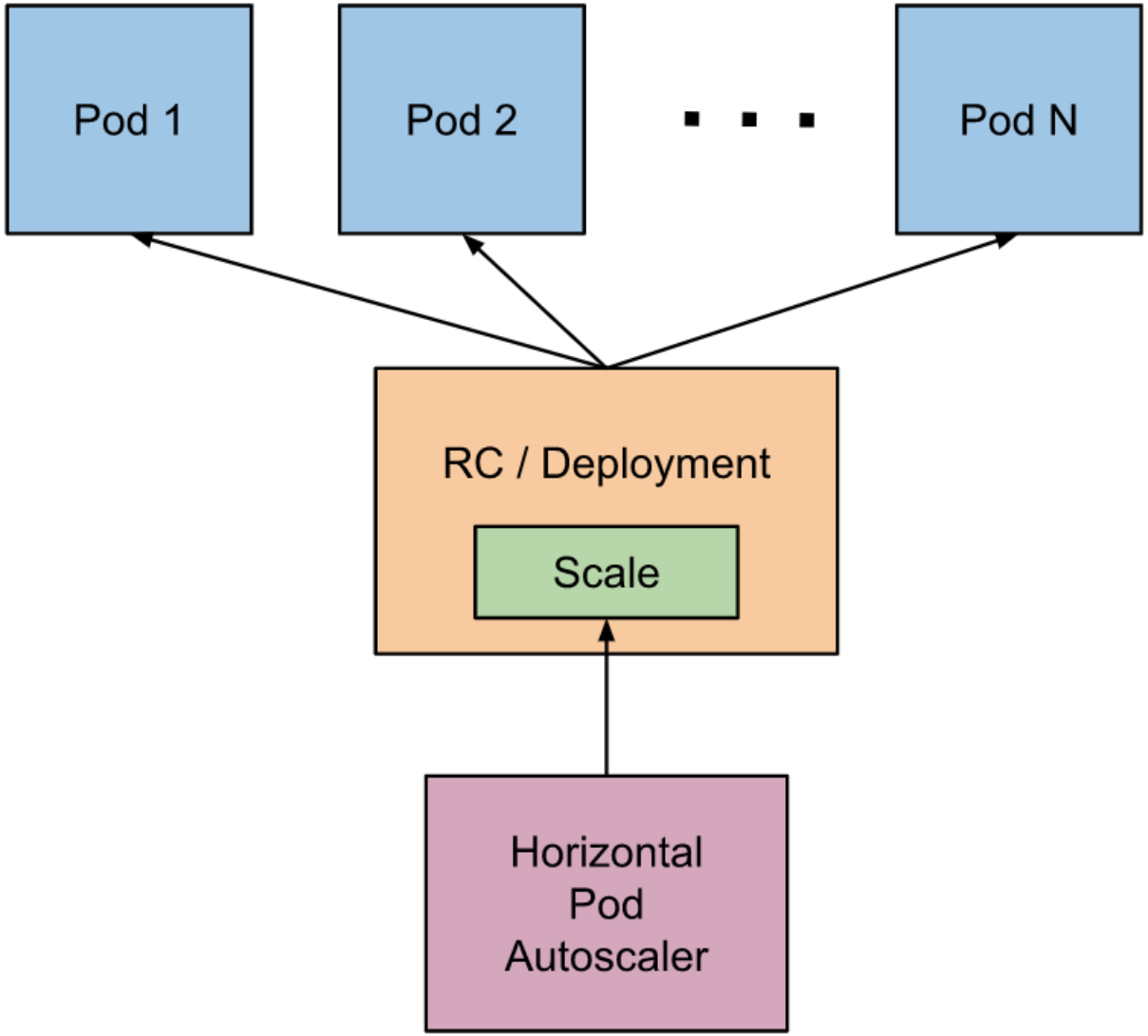
HPA

前言

HPA(Horizontal Pod Autoscaling)指Kubernetes Pod的横向自动伸缩，其本身也是Kubernetes中的一个API对象。通过此伸缩组件，Kubernetes集群便可以利用监控指标（CPU使用率等）自动扩容或者缩容服务中的Pod数量，当业务需求增加时，HPA将自动增加服务的Pod数量 ，提高系统稳定性，而当业务需求下降时，HPA将自动减少服务的Pod数量，减少对集群资源的请求量(Request)，配合Cluster Autoscaler，还可实现集群规模的自动伸缩，节省IT成本。

需要注意的是，目前默认HPA只能支持根据CPU和内存的阈值检测扩缩容，但也可以通过custom metric api 调用prometheus实现自定义metric，根据更加灵活的监控指标实现弹性伸缩。但HPA不能用于伸缩一些无法进行缩放的控制器如DaemonSet。

工作原理



HPA在K8S中被设计为一个Controller，可以简单的使用kubectl autoscale命令来创建。HPA Controller默认30秒轮询一次，查询指定的Resource中（Deployment,RC）的资源使用率，

并且将其与创建HPA时设定的指标做对比，从而实现自动伸缩的功能。

创建了HPA后，HPA会从Metric Server（UK8S中不使用Heapster）获取如某个Deployment中每一个Pod利用率的平均值，然后和HPA中定义的指标进行对比，同时计算出需要伸缩的具体值并进行操作。其算法模型大致如下：

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

例如，如果当前所有Pod的平均CPU使用量是200m，而期望值为100m，那副本数(replicas)将会翻倍。而如果当前的值为50m，那就需要减去一半的副本数(replicas)。

需要注意的是，HPA Controller中有一个tolerance（容忍力）的概念，当currentMetricValue / desiredMetricValue的比率接近1.0时，并不会触发伸缩。默认的方差为0.1，这主要是出于系统稳定性的考虑，避免集群震荡。例如，HPA的策略为cpu使用率高于50%触发扩容，那么只有当使用率大于55%时才会触发扩容动作，HPA通过扩缩Pod，尽力把Pod的使用率控制在这个45%~55%范围之间。你可以通过--horizontal-pod-autoscaler-tolerance这个参数来调整方差值。

在每次扩容和缩容后都有一个窗口时间，在执行伸缩操作后，在这个窗口时间内，不会在进行伸缩操作，可以理解为类似技能的冷却时间。默认扩容为3分钟(--horizontal-pod-autoscaler-upscale-delay)，缩容为5分钟(--horizontal-pod-autoscaler-downscale-delay)。

最后值得注意的是，**Pod没有设置 Request 时，HPA 不会工作**。

HPA 对象控制台管理

HPA 对象的添加、查看和删除，可在 UK8S 集群管理控制台**集群伸缩**页面**弹性伸缩(HPA)** 子页进行。

点击**表单添加**可通过控制台页面添加 HPA 对象，您也可以通过 yaml 进行添加。

配置项	描述
命名空间	HPA 对象所属 Namespace 命名空间
HPA 对象名称	名称必须以小写字母开头，只能包含小写字母、数字、小数点(.)和中划线(-)

应用类型	支持 Deployment 及 StatefulSet 控制器
应用名称	选择需要进行弹性伸缩的 Deployment 及 StatefulSet 对象
扩容阈值	扩缩容阈值，支持设置 CPU 及内存利用率
伸缩区间	Pod 副本数量范围

HPA API对象详解

UK8S 控制台通过 **autoscaling/v2beta2** 版本 Kubernetes API 进行 HPA 对象的创建。

> 注意：集群版本1.26之前请使用autoscaling/v2beta2，集群版本1.26开始请使用autoscaling/v2

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
name: nginxtest
namespace: default
spec:
maxReplicas: 5 #最大副本数
minReplicas: 1 #最小副本数
metrics:
# 设置触发伸缩的 CPU 利用率
- type: Resource
resource:
```

```
name: cpu
target:
averageUtilization: 50
type: Utilization
# 设置触发伸缩的 MEM 利用率
- type: Resource
resource:
name: memory
target:
averageUtilization: 50
type: Utilization
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment #需要伸缩的资源类型
name: nginxtest #需要伸缩的资源名称
```

案例实践

下面我们用一个简单的例子来看下HPA如何工作。

1. 部署测试应用

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/hpa/hpa-example.yaml
```

这是一个计算密集型的PHP应用，代码示例如下：

```
<?php
$x = 0.0001;
for ($i = 0; $i <= 1000000; $i++) {
$x += sqrt($x);
}
echo "OK!";
?>
```

2. 为测试应用开启HPA

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/hpa/hpa.yaml
```

3. 部署压测工具

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/hpa/load.yaml
```

压测工具是一个busybox容器，容器启动后循环访问测试应用。

```
while true; do wget -q -O- http://hap-example.default.svc.cluster.local; done
```

4. 查看测试应用负载情况

```
kubectl top pods | grep hpa-example
```

5. 当测试应用的CPU平均负载超过55%后，我们发现HPA将开始扩容Pod

```
kubectl get deploy | grep hpa-example
```

Vertical Pod Autoscaler (VPA) 使用文档

1. 初识VPA

1.1. 什么是 VPA

VPA 是一个自动调整容器中垂直资源（CPU和内存）的工具。VPA 根据容器实际资源使用情况，自动调整Pod的资源请求，以确保Pod在运行时有足够的资源。

1.2. VPA 的优势

优势

- **资源优化：** VPA通过动态调整容器资源请求，提高资源利用率。
- **性能改进：** 通过确保Pod有足够的资源，VPA提高了应用程序的性能和稳定性。
- **自动化：** VPA能够在不需要人为干预的情况下自动调整资源。

1.2.1. VPA 的限制

1. 由于VPA会自动调整Pod的资源请求，因此Pod可能会在后台重启，Pod可能会调度到其他的节点。

2. 请避免VPA和HPA同时使用。

2. 安装

2.1. 先决条件

- Kubernetes 1.10+

2.2. 部署

2.2.1. 部署 VPA 服务证书

```
curl -sfL https://docs.ucloud.cn/uk8s/yaml/vpa/gencerts.sh | sh -
```

2.3. 部署 VPA 服务

1.22及版本以上

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/vpa/vpa.yaml
```

1.22以下

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/vpa/vpa-less-1.22.yaml
```

2.4. 部署一个 VPA 对象

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/vpa/deployment.yaml
```

3. 查看 VPA 对象

```
kubectl describe vpa nginx-vpa
```

上述命令输出为 VPA 为 Deployment 推荐的值。

```
Recommendation:
Container Recommendations:
Container Name: nginx
Lower Bound:
Cpu: 25m
Memory: 262144k
Target:
Cpu: 25m
Memory: 262144k
Uncapped Target:
Cpu: 25m
Memory: 262144k
Upper Bound:
Cpu: 50032m
Memory: 107227776429
```

定时伸缩

HPA(Horizontal Pod Autoscaling) 指 Kubernetes Pod 的横向自动伸缩，是 Kubernetes 集群利用监控指标自动扩容或者缩容服务中的 Pod 数量，其中监控指标利用 CPU 内存等。定时伸缩不同的是通过定时器进行 Pod 的数量的伸缩，用于已知的高并发，在高并发来临前提前扩容业务进行应对。

1. 在UK8S使用定时伸缩

1.1 开启定时伸缩

在 UK8S 集群管理页面中点击**集群伸缩**标签页，选择**定时伸缩CronHPA**，点击立即开启安装 CronHPA 控制插件，开启定时伸缩功能。

1.2 添加定时伸缩条件

用户点击添加进入新增定时任务页面，在页面中需要输入定时器的名字、选择需要伸缩的对象、执行计划的时间和目标 Pod 数量。如勾选「单次执行」选项，则表明该定时伸缩任务仅需执行一次，非周期性执行。

1.2 针对计划表语法说明

针对计划表语法使用和 CronTab 一致的语法，下面列举几种常用语法，详细语法请参考链接

Crontab格式(前5位为时间选项,这里我们只用到了前5位)

```
<分钟> <小时> <日> <月份> <星期> <命令>
```

每天一次,0点0分执行

```
0 0 * * *
```

每周一次,0点0分执行

```
0 0 * * 0
```

每月一次,0点0分执行

```
0 0 1 * *
```

⚠ CronTab 的命令时间为 UTC 时间,任务真实执行时间用户可以进行 +8 小时计算。

1.3 示例 yaml

我们针对 nginx-deployment 这个应用设置了 up5 和 down2 两个执行计划,分别设置的是 `40 8 * * *` 和 `50 8 * * *`,即应用将在北京时间 16 点 40 分扩容到 5 个,在 16 点 50 分缩容到 2 个,并每天执行。

```
apiVersion: autoscaling.ucloud.cn/v1
kind: CronHorizontalPodAutoscaler
metadata:
name: "nginx-cronhpa"
namespace: default
spec:
jobs: # 执行计划，可在同一个 CronHPA 任务中添加多个执行计划
- name: "up5"
schedule: "40 8 * * * "
targetSize: 5
runOnce: false
- name: "down2"
schedule: "50 8 * * * "
targetSize: 2
runOnce: false
scaleTargetRef: # 目标执行对象，支持 Deployment、StatefulSet 及 HPA 资源对象
apiVersion: apps/v1
kind: Deployment
name: nginx-deployment
```

2. CronHPA 定时伸缩支持 HPA 对象

CronHPA 插件支持在创建时，选择原有的 HPA 对象，兼容规则如下：

HPA 配置 min/max	CronHPA 目标Pod数	Deployment 当前 Pod 数	扩缩结果	说明
1/10	5	5	HPA:5/10 Deployment:5	CronHPA 目标副本数 > HPA 副本数下限,修改 HPA 中的副本数下限
5/10	4	5	HPA:4/10 Deployment:5	CronHPA 目标副本数 < HPA 副本数下限,修改 HPA 中的副本数下限 当业务下降低于 HPA 设定阈值范围时,HPA 将调整 Deployment 中副本数为 4
1/10	11	5	HPA:11/11 Deployment:11	CronHPA 目标副本数 > HPA 副本数上限,同时修改 HPA 中的副本数上限与下限

Cluster Autoscaler

前言

Cluster Autoscaler(CA)用于自动调整集群中Node节点的数量,以满足业务需求。

我们知道,在创建Pod的时候,我们可以为每个容器指定CPU、内存、GPU等资源的请求量(Request),Kubernetes的调度组件(scheduler)会通过Request来判断将Pod调度到哪个Node节点。如果集群内没有节点有足够的空闲余量,则该Pod将无法被成功创建,而是一直处于Pending状态,直到有新的Node节点加入集群或者存量Pod被删除释放出空闲余量。

CA组件查找无法被成功调度的Pod,并遍历伸缩组,判断通过伸缩组模版扩容的新节点是否满足要求,如果判断新加节点可以使得Pod被成功调度,则CA将扩容集群。

CA组件同样也会缩容集群,缩容的触发条件为某个Node节点的Request请求率低于缩容阈值,不过缩容并不是马上进行的,而是等待一段时间(目前默认是10分钟),可以通过--scale-down-unneeded-time这个参数来修改。

和HPA不同,CA不是内置的,而是以Deployment的形式运行在Kubernetes集群中,UK8S已支持CA,你可以在UK8S的管理界面中配置CA。

工作原理

CA的扩容触发条件为**存在因为集群资源不足导致无法成功创建的Pod**，这里的资源包括**CPU、内存和GPU**。以GPU为例，当Pod申请了GPU资源nvidia.com/gpu（参考GPU节点使用文档），但因集群中无GPU节点而处于pending状态时，CA就会在配置了GPU机型模版的伸缩组中自动扩容节点。

CA的缩容触发条件为**node节点在一定时间内（默认10分钟）资源请求率（Request）低于缩容阈值（如50%）且节点上的所有Pod都能被调度到其他节点上**。

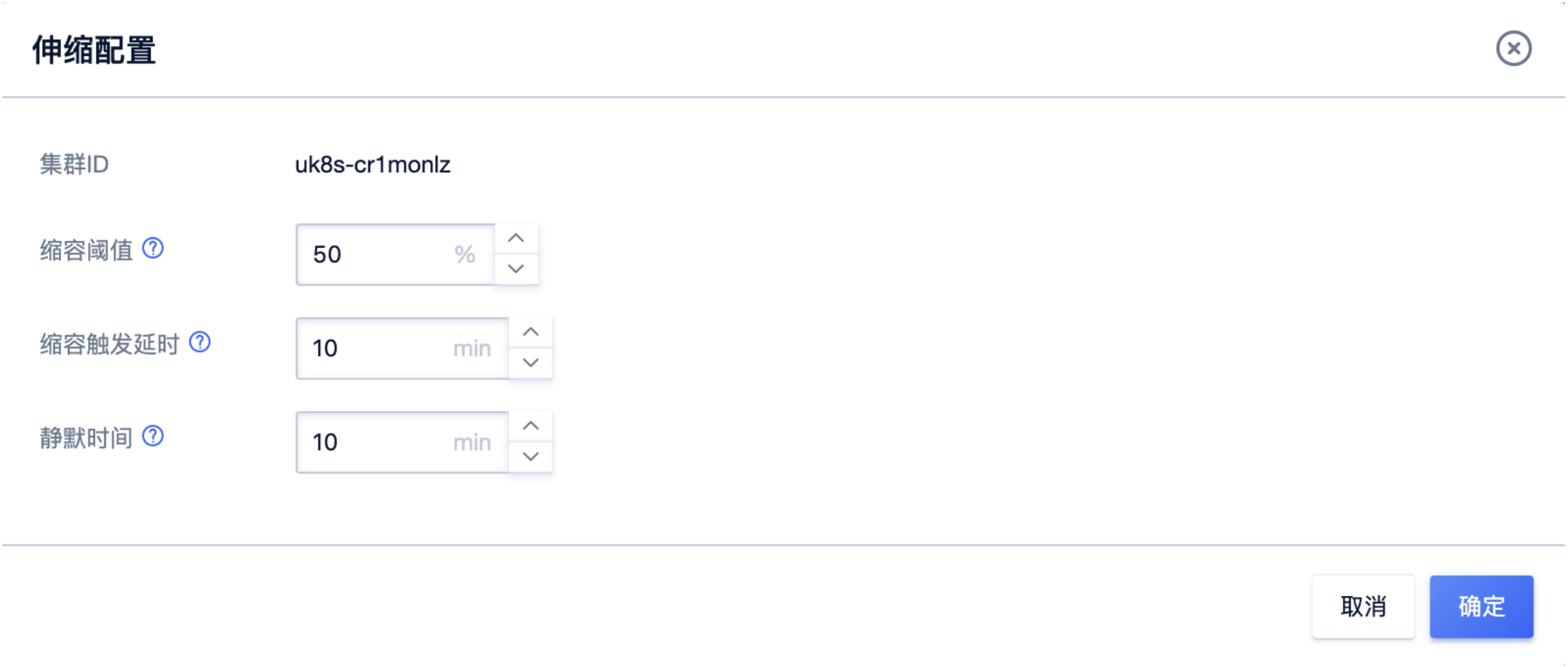
值得注意的是**节点上的所有Pod都能被调度到其他节点**这个条件，很多配置了CA的同学会疑问某节点资源申请量低于阈值却没有触发缩容，原因其实很简单，如果这个节点上运行了一个独立的Pod（没有被任何控制器管理），因为Pod无法被重新调度，为了保证业务正常运行，则节点的缩容不会进行。

在UK8S中使用集群伸缩

1、创建伸缩配置

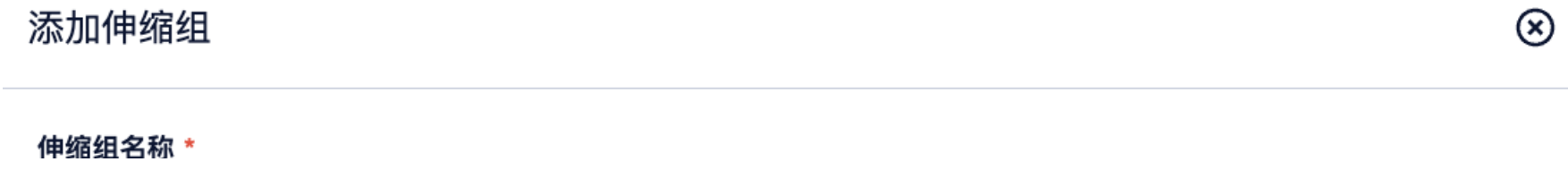
2、填写配置参数

一般默认值即可



3、创建伸缩组

重要，即触发集群扩容时，Node节点的配置，伸缩区间主要用于防范因为DDos等导致的无限制扩容。



扩容优先级

1

节点池 *

test

节点池	test(uk8s-144k0l3xpua6-ng-pvdee)	机型	快杰型 O
CPU平台	Intel (x86_64)	配置	2 | 4 | 40 | 20
付费方式	按时	预估月单价	396

伸缩区间

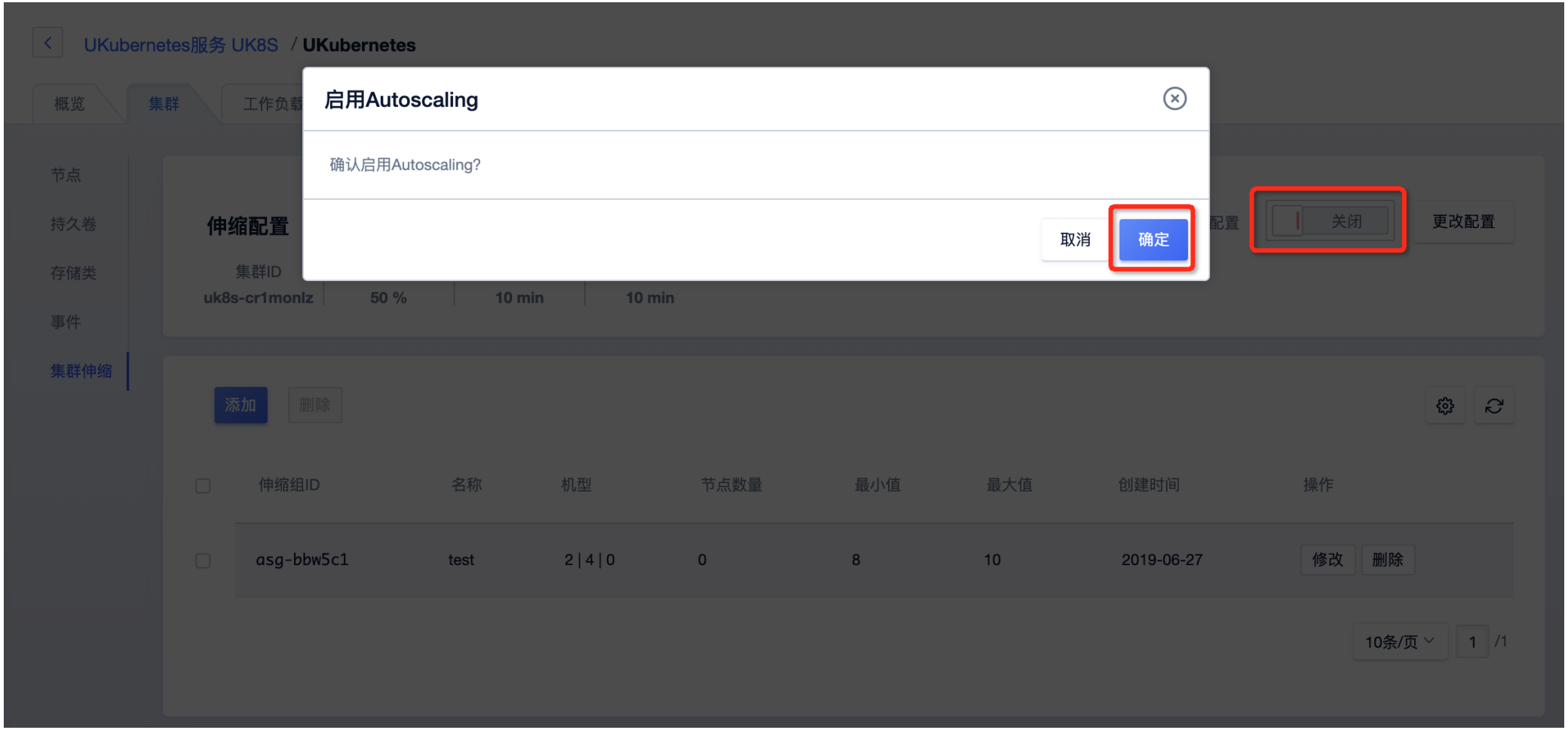
0 - 1

设置密码 *

设置密码 随机生成

4、开启集群伸缩

创建完伸缩组后，我们之后还需要开启伸缩组，点击开启操作后，你的UK8S集群会出现一个Cluster-Autoscaler的Deployment，如果手动删除该Deployment，会导致集群伸缩无法正常工作，您需要在集群伸缩页面先关闭，再开启以触发重新创建。



CA参数说明

CA本身有很多命令参数，可以调整伸缩的一些行为。可以通过更改CA deployment的`args`参数来调整。

下面是一些CA参数以及说明：

参数	类型	默认值	说明
scale-down-delay-after-add	Duration	10min	扩容后进行缩容的延迟。
scale-down-delay-after-delete	Duration	同scan-interval	删除节点后进行缩容的延迟。
scale-down-unneeded-time	Duration	10min	节点标记为unneeded之后，多久进行缩容。
node-deletion-delay-timeout	Duration	2min	CA等待节点删除完成的超时时间。
scan-interval	Duration	10s	多久进行一次扩缩容扫描。
max-nodes-total	int	0	最大扩容节点数量。
cores-total	String	[0:32E+04]	集群的CPU核心扩缩容范围。
memory-total	String	[0:64E+05]	集群的内存扩缩容范围。

基于自定义指标的容器弹性伸缩

前言

HPA(Horizontal Pod Autoscaling)指Kubernetes Pod的横向自动伸缩，其本身也是Kubernetes中的一个API对象。通过此伸缩组件，Kubernetes集群便可以利用监控指标（CPU使用率等）自动扩容或者缩容服务中的Pod数量，当业务需求增加时，HPA将自动增加服务的Pod数量 ，提高系统稳定性，而当业务需求下降时，HPA将自动减少服务的Pod数量，减少对集群资源的请求量(Request)，配合Cluster Autoscaler，还可实现集群规模的自动伸缩，节省IT成本。

需要注意的是，目前默认HPA只能支持根据CPU和内存的阈值检测扩缩容，但也可以通过custom metric api 调用prometheus实现自定义metric，根据更加灵活的监控指标实现弹性伸缩。但HPA不能用于伸缩一些无法进行缩放的控制器如DaemonSet。

启用custom.metrics.k8s.io服务

在开始此步骤之前，请确认你已按照前述教程安装了Prometheus。

这里简单介绍下HPA的工作原理，默认情况下，其通过metrics.k8s.io这个本地服务来获取Pod的CPU、Memory指标，CPU和Memory这两者属于核心指标，而metrics.k8s.io服务对应的后端服务一般是metrics server，这是UK8S默认安装的服务。

而如果HPA要通过非CPU、内存的其他指标来伸缩容器，我们则需要部署一套监控系统如Prometheus，让prometheus采集各种指标，但是prometheus采集到的metrics并不能直接给k8s用，因为两者数据格式不兼容，因此另外一个组件prometheus-adapter，将prometheus的metrics数据格式转换成K8S API接口能识别的格式。另外我们还需要在K8S注册一个服务（即custom.metrics,k8s.io），以便HPA能通过/apis/访问。

我们申明一个v1beta1.custom.metrics.k8s.io的APIService，并提交。

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
name: v1beta1.custom.metrics.k8s.io
spec:
group: custom.metrics.k8s.io
groupPriorityMinimum: 100
insecureSkipTLSVerify: true
service:
name: prometheus-adapter
namespace: monitoring
port: 443
version: v1beta1
versionPriority: 100
```

上述示例中的spec.service.prometheus-adapter在之前文档中已经安装并部署完毕。提交部署后，我们执行“kubectl get apiservice | grep v1beta1.custom.metrics.k8s.io”，确认该服务可用状态为True。

还可以通过下述方法来查看Prometheus采集了哪些指标。

```
kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/ | jq .

kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/namespace/default/pods/*/ | jq .

curl 127.0.0.1:8080/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods/*/http_requests
```

修改原有prometheus-adapater的配置文件

为了让HPA能够用到Prometheus采集到的指标，prometheus-adapter通过使用promql来获取指标，然后修改数据格式，并把重新组装的指标和值通过自己的接口暴露。而HPA会通过/apis/custom.metrics.k8s.io/代理到prometheus-adapter的service上来获取这些指标。

如果把Prometheus的所有指标到获取一遍并重新组装，那adapter的效率必然十分低下，因此adapter将需要读取的指标设计成可配置，让用户通过configmap来决定读取Prometheus的哪些监控指标。

关于config的语法规则，详见config-workthrough，这里不再赘述。

由于我们前面已经安装了prometheus-adapter,因此我们现在只需要修改其配置文件并重启即可，原始的配置文件只包含cpu和memory两个Resource metrics，我们只需要在其前面追加需要给HPA用到的metrics即可。

```
apiVersion: v1
data:
```

```
config.yaml: |
resourceRules:
cpu:
containerQuery: sum(rate(container_cpu_usage_seconds_total{<<.LabelMatchers>>,container_name!="POD",container_name!="",pod_name!=""}[1m])) by (<<.GroupBy>>)
nodeQuery: sum(1 - rate(node_cpu_seconds_total{mode="idle"}[1m]) * on(namespace, pod) group_left(node) node_namespace_pod:kube_pod_info:{<<.LabelMatchers>>}) by (<<.GroupBy>>)
resources:
overrides:
node:
resource: node
namespace:
resource: namespace
pod_name:
resource: pod
containerLabel: container_name
memory:
containerQuery: sum(container_memory_working_set_bytes{<<.LabelMatchers>>,container_name!="POD",container_name!="",pod_name!=""}) by (<<.GroupBy>>)
nodeQuery: sum(node_memory_MemTotal_bytes{job="node-exporter",<<.LabelMatchers>>} - node_memory_MemAvailable_bytes{job="node-exporter",<<.LabelMatchers>>}) by (<<.GroupBy>>)
resources:
overrides:
instance:
resource: node
```

```
namespace:
resource: namespace
pod_name:
resource: pod
containerLabel: container_name
window: 1m
kind: ConfigMap
metadata:
name: adapter-config
namespace: monitoring
```

我们以常见的请求数为例，追加一个指标，其名称为http_request,**资源类型为**Pod。

```
apiVersion: v1
data:
config.yaml: |
rules:
- seriesQuery: '{__name__=~"^http_requests_.*",kubernetes_pod_name!="",kubernetes_namespace!=""}'
seriesFilters: []
resources:
overrides:
kubernetes_namespace:
resource: namespace
kubernetes_pod_name:
resource: pod
```

```
name:
matches: ^(.*)_(total)$
as: "${1}"
metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)
resourceRules:
cpu:
containerQuery: sum(rate(container_cpu_usage_seconds_total{<<.LabelMatchers>>,container_name!="POD",container_name!="",pod_name!=""}[1m]))
by (<<.GroupBy>>)
nodeQuery: sum(1 - rate(node_cpu_seconds_total{mode="idle"}[1m]) * on(namespace, pod) group_left(node) node_namespace_pod:kube_pod_info:
{<<.LabelMatchers>>}) by (<<.GroupBy>>)
resources:
overrides:
node:
resource: node
namespace:
resource: namespace
pod_name:
resource: pod
containerLabel: container_name
memory:
containerQuery: sum(container_memory_working_set_bytes{<<.LabelMatchers>>,container_name!="POD",container_name!="",pod_name!=""}) by
(<<.GroupBy>>)
nodeQuery: sum(node_memory_MemTotal_bytes{job="node-exporter",<<.LabelMatchers>>} - node_memory_MemAvailable_bytes{job="node-exporter",
<<.LabelMatchers>>}) by (<<.GroupBy>>)
resources:
```

```
overrides:
instance:
resource: node
namespace:
resource: namespace
pod_name:
resource: pod
containerLabel: container_name
window: 1m
kind: ConfigMap
metadata:
name: adapter-config
namespace: monitoring
```

修改完毕并提交后，如果为了立马生效，我们可以删除掉原有的prometheus-adapter的Pod，使得配置文件立马生效。

当然只有这些指标还是略微不够，社区提供了一个rules的示例：adapater-config标准样例

集群网络

概述

在我们创建一个Kubernetes集群时，为了让集群正常工作，我们需要为三类资源对象规划网段，分别是Node，Pod，Service，他们都需要唯一的网络标示。作为一个生产级别的容器编排与调度系统，Kubernetes要求各网络方案必须满足以下三点要求：

1. Pod与Pod之间网络互通，且不需要经过NAT转换。
2. Pod与Node之间网络互通，且不需要经过NAT转换。
3. Pod内部容器看到的IP，与外部应用看到的IP，应该是一样的。

基于这个准则实现的网络插件，意味着Pod必须有一个独立的IP，这与虚拟机时代的网络模型完全一致，让业务从虚拟机迁移到Kubernetes，包括协同工作提供了良好的基础。

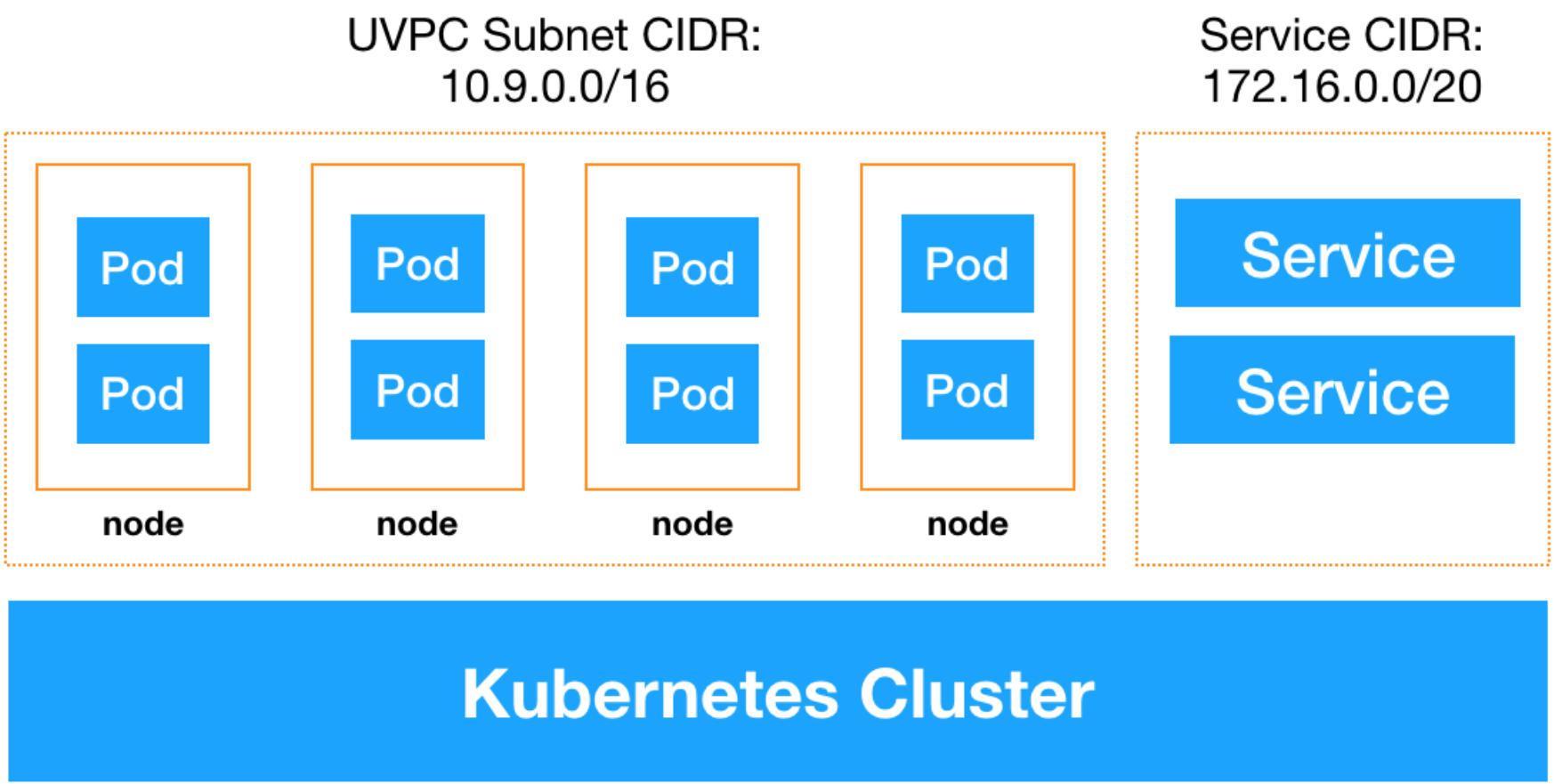
UK8S网络模型

Kubernetes自身只提供了网络规范和开放接口，Kubernetes用户可以安装开源的网络插件或者自行开发CNI插件，对于UK8S而言，自行开发CNI插件，需要解决以下几个网络问题。

1. Pod与Pod之间的通信问题。
2. Pod与UHost、UDB等云资源之间的通信问题。
3. Pod与Service之间的通信。Kubernetes社区提供了IPtables和IPVS两套方案，UK8S使用的IPtables方案，如果对这块的具体原理感兴趣，可以查看官方文档，此处不再赘述。
4. 集群外部与Service之间的通信。Kubernetes提供了LoadBalancer类型的Service，UK8S已支持，具体请参见Service

综合以上几点，与第三方插件通常用的overlay方案不同，而我们结合公有云的特点，使用了underlay方案。

Pod 与Node 同属一个子网，IP都由SDN网络分配，Service 的ClusterIP 只在集群内部使用，用户只需要分配一个与VPC子网不重叠的网段即可，网段示意图如下:



经测试，该网络方案下，Pod之间的网络通信性能与虚拟机之间相差无几。

集群通信

一、集群内部通信

1. 集群内Pod与Pod内网互通；
2. 集群内Pod与Node内网互通；
3. 集群内Pod与Service内网互通；

二、与云资源通信

1. 集群内Pod与UHost、UDB、UMem等资源内网互通（同VPC，下同）；
2. 集群内Pod与PHost内网互通；
3. 集群内Pod与混合云内网互通；

网络隔离策略 NetworkPolicy

在 UK8S 集群中中，默认情况下，所有的 Pod 都是互通的，即任一 Pod 既可以接收来自集群中任何 Pod 发送的请求，也可以向任一集群中 Pod 发送请求。

但在实际业务场景中，为了保障业务安全，网络隔离是非常必要的。下面介绍下如何在 UK8S 中实现网络隔离。

安装前检查

⚠ 在安装 Calico 网络隔离插件之前，请务必确认 CNI 版本大于等于 19.12.1，否则会删除Node上原有的网络配置，导致 Pod 网络不通。CNI 版本查询及升级请参考：CNI 网络插件升级。

检查kubernetes版本 <=1.26.7，且>= 1.16.4，并且集群需要通外网拉取Uhub以外的镜像。

确认集群中是否使用组件ipamd：

```
kubectl -n kube-system get ds cni-vpc-ipamd
```

如果没有使用，可以忽略下面检查；如果已经使用ipamd，确认ipamd是否开启Calico网络策略支持；使用如下命令查看参数--calicoPolicyFlag是否为true：

```
kubectl -n kube-system get ds cni-vpc-ipamd -o=jsonpath='{.spec.template.spec.containers[0].args}{"\t"}{"\n"}'
```

```
["--availablePodIPLowWatermark=3","--availablePodIPHighWatermark=50","--calicoPolicyFlag=true","--cooldownPeriodSeconds=30"]
```

如果--calicoPolicyFlag参数不为true，需要使用如下命令开启：

```
kubectl -n kube-system patch ds cni-vpc-ipamd -p '{"spec":{"template":{"spec":{"containers":[{"name":"cni-vpc-ipamd","args":["--availablePodIPLowWatermark=3","--availablePodIPHighWatermark=50","--calicoPolicyFlag=true","--cooldownPeriodSeconds=30"]}]}}}}'
```

1. 安装插件

为了在 UK8S 中实现网络隔离，需要部署 Calico 的 Felix 和 Typha 组件，组件模块已容器化，直接在 UK8S 通过 kubectl 命令安装即可.

UK8S 提供了两种版本的 Calico 组件来实现网络隔离，分别兼容以下 UK8S 版本，请自行选择.

UK8S version	Calico version
<=1.24.12	3.10.0
1.26.7	3.25.2

```
calico_version="3.25.2" && kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/networkpolicy/${calico_version}-calico-policy-only.yaml
```

2. NetworkPolicy 规则解析

安装完 Calico 的网络隔离策略组件后，我们就可以在 UK8S 中创建 NetworkPolicy 对象，用于实现 Pod 的访问控制，如下所示。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: test-network-policy
namespace: default
spec:
podSelector:
matchLabels:
role: db
policyTypes:
- Ingress
- Egress
ingress:
- from:
- ipBlock:
cidr: 172.17.0.0/16
except:
- 172.17.1.0/24
- namespaceSelector:
matchLabels:
project: myproject
- podSelector:
matchLabels:
```

```
role: frontend
ports:
- protocol: TCP
port: 6379
egress:
- to:
- ipBlock:
cidr: 10.0.0.0/24
ports:
- protocol: TCP
port: 5978
```

下面简要描述下各个参数的作用：

- **spec.podSelector**: 这个参数的意义用于决定该NetworkPolicy的作用域，即对那些Pod有效。上面的示例表示对default namespace下携带了role=db 标签(label)的pod生效。这里要说明下，NetworkPolicy是namespace级别的资源对象。
- **spec.ingress.from**: 入向请求访问控制，即接受哪些源的请求。支持IP、namespace、pod三种控制模式，上面的示例表示放行源地址为172.17/16中除172.17.1/24 之外的请求、或任何携带了标签projcet=myproject的namespace下的所有pod、或Default namespace里携带了role=frontend标签的pod。from中的多组规则是或逻辑，满足上述三个条件中的一个即放行。其中namespaceSelector这个字段，用于筛选来自多个namespace下的请求源。
- **spec.ingress.ports**: 声明开放访问的端口，如果不填写则默认开放所有。上面的示例表示只允许访问6379端口。from和ports是与逻辑，即指允许上述from规则下放行的源访问6379端口（TCP）。
- **spec.egress**: 声明允许访问的目的地址，与from类似。上面的示例表示只允许请求IP为10.0.0.0/24网段的地址，并且只允许访问该网段地址的5978端口（TCP）。

通过上面的描述，我们应该清楚，NetworkPolicy是一个白名单机制，即一旦开启NetworkPolicy，除非显式指定，否则一概拒绝。

3. 示例

3.1 限制一组Pod只允许访问VPC内部的资源(不能访问外网)

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
name: pod-egress-policy
spec:
podSelector:
matchLabels:
pod: internal-only
egress:
- to:
- ipBlock:
cidr: 10.9.0.0/16
```

3.2 限制暴露了公网服务的Service的来源IP

首先创建一个通过外网ULB4对公网暴露服务的应用

```
apiVersion: v1
kind: Service
metadata:
```

```
name: ucloud-nginx
labels:
app: ucloud-nginx
spec:
type: LoadBalancer
externalTrafficPolicy: Local
ports:
- protocol: TCP
port: 80
selector:
app: ucloud-nginx
---
apiVersion: v1
kind: Pod
metadata:
name: test-nginx
labels:
app: ucloud-nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
ports:
- containerPort: 80
```

上述应用创建完毕后，我们可以通过外网ULB IP直接访问应用，现在我们设置只允许从办公室环境访问该应用，假设办公室出口IP为106.10.10.10。

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
name: access-nginx
spec:
podSelector:
matchLabels:
app: ucloud-nginx
ingress:
- from:
- ipBlock:
cidr: 106.10.10.10/24 #验证时请改为你客户端的出口IP。
- ipBlock:
cidr: 10.23.248.0/21 #地域级别公共服务网段，否则ULB健康检查会失败导致隔离策略不生效，详见下文
```

4. 放行 VPC 公共服务网段

公共服务网段主要用于内网 DNS、ULB 健康检查等，建议在配置 NetworkPolicy 时，放行各地域的公共服务网段。

各地域公共服务网段请参考 VPC 文档：VPC 网段使用限制

固定 IP 使用方法

固定 IP 适用于对容器固定 IP 强依赖的场景。

在传统的虚拟机部署形式下，部分客户依赖虚拟机 IP 地址，用于问题排查、监控、流量分配等，固定 IP 支持能够帮助用户更好地从虚拟机向容器迁移，提升运维效率。对 IP 无限制的业务不推荐您使用固定 **IP** 模式。

固定 IP 仅支持 **StatefulSet** 形式的资源控制器。

1. 固定 IP 插件安装和升级

请先通过UK8S控制台`应用中心` -> `固定IP管理`功能安装相关插件。后续插件版本更新也可以通过此页面操作。

2. 创建固定 IP 类型的 StatefulSet

当前仅支持通过 Yaml 形式创建固定 IP 类型的 StatefulSet，需要您在 spec.template.annotations 添加相应注释进行配置，后续版本将支持通过控制台表单创建：

注释	注释说明	参数类型	默认值
network.beta.kubernetes.io/ucloud-statefulset-static-ip	是否需要开启固定 IP 功能	true / false	false

network.beta.kubernetes.io/ucloud-statefulset-ip-claim-policy	IP 回收策略，即 Pod 销毁及绑定的 VpcIP 解绑后释放的时间	hms / Never 例：1h10m20s 代表 VpcIP 解绑后 1 小时 10 分 20 秒后被释放	Never

以下为创建一个 StatefulSet 类型的 Nginx 应用并对外暴露的 Yaml 范本

```
apiVersion: v1
kind: Service
metadata:
name: nginx
labels:
app: nginx
spec:
ports:
- port: 80
name: nginx
clusterIP: None
selector:
app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: web-test
```

```
namespace: default
spec:
selector:
matchLabels:
app: nginx
serviceName: "nginx"
replicas: 5
template:
metadata:
annotations:
# 声明需要开启固定 IP 功能
network.beta.kubernetes.io/ucloud-statefulset-static-ip: "true"
# 设定 VpcIP 释放时间为 300 秒
network.beta.kubernetes.io/ucloud-statefulset-ip-claim-policy: "300s"
labels:
app: nginx
spec:
terminationGracePeriodSeconds: 10
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
ports:
- containerPort: 80
name: web
```

3. 异常情况说明

1. Node 节点宕机 / 移除

节点宕机或从集群中移除，将触发 Pod 的强制迁移，Pod 绑定的 VpcIP 将会被保留，Pod 调度到新的 Node 节点后，管理服务将更新、记录相应的映射关系。

如果该节点上曾经有固定 IP Pod，缩容后该 Pod 被终止，并且没有因为扩容等再重新拉起或被调度至其他节点，节点上将保留有该 Pod 的 VpcIP，在移除节点时，该 VpcIP 会因未被任何 Pod 占用被解除与该节点的绑定关系并释放。此时如再在其他节点重新拉起该 Pod，可能会因该 VpcIP 已被其他应用占用而导致 Pod 无法重新拉起。**因此在移除节点前，请务必确认节点上已无未被 Pod 占用的 VpcIP**，具体步骤请参照「4. 节点下线步骤」。

2. Node 节点强制删除

指在云主机页面（而非通过 UK8S 集群管理功能）进行 Node 节点云主机资源的删除，此情况下 UK8S 管理服务无法进行保留 VpcIP 操作，VpcIP 将会被主机服务强制释放，并存在被其他资源占用的可能性。

固定 IP 组件将会尝试以指定 IP 形式重新申请旧有 VpcIP，并绑定至新拉起的 Pod 上，但如相应的 VpcIP 已被占用，将出现更新失败情况。

3. 同一 StatefulSet 中出现不同 VPC 子网的 Pod

固定 IP 功能暂**不支持跨 VPC 子网**，在 CNI 工作原理下，Pod 与所在 Node 节点处在同一 VPC 子网。如在 StatefulSet 扩容、Pod 异地更新、Node 节点宕机等情况下，Pod 被调度到非同子网的 Node 节点上，将会出现 Pod 创建/更新失败的错误。

建议您合理分配您的子网及网段，避免在集群中存在多子网现象，或通过标签等将 StatefulSet 指定调度到同一子网 Node 节点上。

4. 节点下线步骤

1. 确认待下线节点的 Mac 信息

登入需要删除的节点，通过 ifconfig 命令查看 eth0 网卡的 Mac 地址。

固定 IP 插件通过自定义资源对象 **vpcIpClaim** 记录 VpcIP、Pod 及运行节点的对应关系。如果节点上有未驱逐的固定 IP Pod，也可以通过 kubectl get pods -o wide 查看该节点上运行的固定 IP Pod 的名称，并再通过 kubectl describe vpcIpClaims <pod-name> 查看 CRD 信息，CRD 中 Status.Mac 可以确定主机的 Mac 地址。

2. 通过 Mac 地址查找归属于下线节点的固定 IP

通过 kubectl 命令，查找出该节点上未被占用 VpcIP 对应的 CRD 对象信息。

```
# 将 grep 命令后的 mac 地址替换为待下线的节点 mac 地址
kubectl get vpcIpClaims -l attached=false -o=json | jq '.items[]|.metadata.name + " " + .status.mac' | grep 52:54:00:26:6E:DA
"web-test-11 52:54:00:26:6E:DA"
"web-test-13 52:54:00:26:6E:DA"
"web-test-14 52:54:00:26:6E:DA"
```

该显示结果说明，在 Mac 地址为 52:54:00:26:6E:DA 的节点上，曾经运行过 web-test-11、web-test-13、web-test-14 这几个 Pod。

3. 调整 StatefulSet 副本数目，保证固定 IP 被占用

执行 kubectl patch sts web-test -p '{"spec":{"replicas":15}}'，将 Sts 副本数调整为 15，保证 web-test-14 拉起，相应的固定 IP 被占用。

执行 kubectl drain <node-name> --ignore-daemonsets，将待下线节点清空，如节点已禁用及清空，则可忽略这一步。

4. 节点下线

再次执行 kubectl get vpcIpClaims -l attached=false -o=json | jq '.items[]|.metadata.name + " " + .status.mac' | grep 52:54:00:26:6E:DA，确定节点上已无未被占用的

VpcIP 后，在 UK8S 控制台移除节点。

最后，再次执行 `kubectl patch sts web-test -p '{"spec":{"replicas":<new-replicas>}}'`，将 Sts 副本数调整为期望值。

网络插件 升级

UK8S 提供的 CNI（Container Network Interface）基于 UCloud VPC 网络实现，因此会随着 VPC 的功能迭代同步更新版本，以提升容器网络的稳定性及性能。UK8S提供 CNI 在线升级的功能，插件升级不会影响现有 Pod 的网络。CNI 升级成功后：

1. 其实现的网络特性将作用于新申请的 Pod；
2. 老的 Pod 如果也需要获得对应的网络特性，则需要滚动升级以触发 Pod 重建；

网络插件包括两个组件：CNI和ipamd。前者负责配置容器网络，后者负责管理 VPC IP 池子，二者可以互相独立升级。您可以在控制台分别看到这两个组件的管理卡片。

下面将介绍下如何在线升级这两个组件。

⚠ 集群网络插件升级时，请勿进行服务发布等操作

1. CNI插件升级

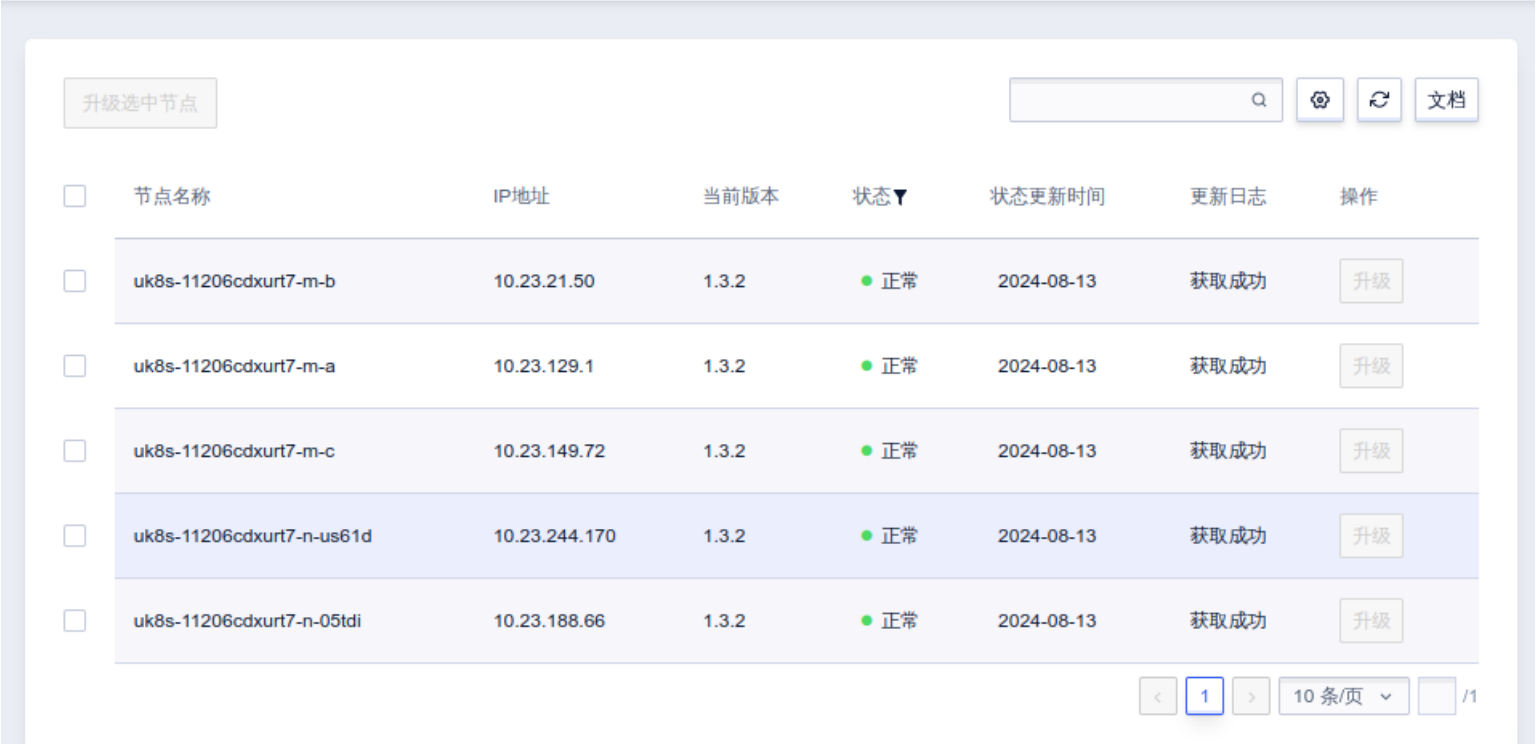
CNI是部署在节点上面的二进制文件，升级或查看版本都需要在您的集群中运行任务。

需要查看当前集群的CNI版本，您可以进入「插件管理-网络插件」页面，找到「CNI插件升级」卡片，点击「开启」，我们会启动任务查询您集群的CNI版本：

查询任务大约需要3分钟的执行时间，在此过程中请不要操作集群。查询完成之后，您就可以点击「管理集群插件版本」来查看您的CNI版本了：

您可以看到每个节点的CNI版本，并且可以对它们进行升级。您可以一次性升级一个节点，或者批量进行升级：

集群插件版本



升级CNI同样需要执行任务，升级过程约需要 1-3分钟，升级过程中「当前版本」字段会显示为「升级中」，升级完成后显示最新版本号。

如升级失败，可以再尝试「强制升级」，或与我们技术支持联系。部分节点由于版本原因，可能无法获取到「当前CNI版本」，直接点击强制升级即可。

建议先升级单台节点，如果升级成功，则再进行批量升级。当所有节点都升级成功后，可关闭插件升级服务，后续有升级需求时再开启。

如控制台页面无法查看 CNI 版本信息，请在集群中任一节点执行 `/opt/cni/bin/cnivpc version` 查看。

对于非维护版本的集群，无法将CNI升级到最新版本，只能升级到LTS（长期维护）版本。详见：集群版本维护说明。

2. ipamd插件升级

ipamd是通过Daemonset部署的，它的版本查询和升级不需要执行任务，可以马上完成。

ipamd是可选组件，它可以有效提高网络插件分配IP的效率，在新版本的集群中，ipamd都是默认开启的。详见：CNI Ipamd预分配VPC IP实现原理和部署架构。

需要查看当前集群的ipamd版本，您可以进入「插件管理-网络插件」页面，找到「ipamd插件」卡片，应该可以看到当前ipamd版本和最新版本。

如果您的集群中没有部署ipamd，需要先点击「开启」部署ipamd：

如果您的ipamd不是最新版本，可以直接点击「升级版本」进行升级：

对于非维护版本的集群，无法开启或升级ipamd。

3. 网络插件更新纪要

以下的版本更新纪要可能有延后性，如果您想获取最新的CNI更新日志以及所有源代码，请到我们的开源仓库查看：uk8s-cni-vpc。

⚠ 带-lts后缀的版本表示长期维护版本，仅包括重大BUG的修复，不包括新功能。

版本	类型	更新说明	发布时间

1.3.4	Feature	适配1.28集群	2024年12月24日
1.3.3	Bugfix	1. 解决在多虚拟网卡的场景下可能出现Pod网络不通的问题。 2. 主动发现子网IP不足的场景，减少接口报错。	2024年11月13日
1.3.2	Bugfix	1. 解决在ipamd和cni同时启动时，可能会出现同一个IP被分配给不同的Pod的问题。 2. 解决在BoltDB数据库文件损坏时，Pod会拿到不可用IP的问题。	2024年08月05日
1.3.1	Bugfix	解决在Linux 5.x内核上创建的Pod网络不通的问题。	2024年08月05日
1.3.0	Feature	1. 适配1.26集群 2. 分配pod ip前通过VPC检查可用性，防止Pod拿到不可用IP导致网络不通	2024年06月19日
1.2.3	Bugfix	修复ipamd不存在时使用固定IP功能存在的问题。	2023年11月23日
1.2.2	Feature	适配裸金属类型节点。	2023年08月22日
1.2.1	Bugfix	修复某些场景下pod ip可能被误分配到其他云主机的虚拟网卡上的问题。	2023年08月22日
1.2.0	Feature	1. 安全起见，彻底去除CNI中GC相关逻辑，新增cnivpctl命令行，将GC和一些其他调试命令移动到命令行工具中。 2. 通过持久化IP池，修复ipamd异常退出后导致的IP泄漏问题。 3. 避免在使用network policy时pod路由规则可能被重复添加导致的报错。	2023年07月18日
1.0.4-lts	LTS	1. 解决在ipamd和cni同时启动时，可能会出现同一个IP被分配给不同的Pod的问题。 2. 解决在BoltDB数据库文件损坏时，Pod会拿到不可用IP的问题。	2024年08月05日
1.0.3-lts	LTS	修复某些场景下pod ip可能被误分配到其他主机的虚拟网卡上的问题。	2023年08月22日
1.0.2	Bugfix	在检测IP冲突时，忽略EINTR错误，以防止误报。	2023年01月16日
1.0.1	Bugfix	修复ipamd申请IP时没有刷新token导致的调用失败问题。	2022年01月04日

1.0.0	Feature	1. CNI正式开源，地址：uk8s-cni-vpc。 2. 重构ipamd相关代码，增加水位控制、IP借调等相关特性。ipamd将作为默认组件。	2022年12月29日
22.05.1	Enhancement	根据 rfc5227 在 Pod 创建时对申请到的内网 IP 进行冲突检测，达到一定重试次数后，如果仍然冲突，会释放当前 IP，并重新申请 IP	2022年05月30日
22.04.1	Feature	将 ip_local_port_range 改为 32768 ~ 60999	2022年04月01日
21.12.2	Feature	ipamd 支持开启 calico policy 特性，IP 会写入到 Pod 的 annotation，calico 可以通过感知该 IP 来及时下发规则	2021年12月28日
21.12.1	Enhancement	1. 优化 ipamd 申请 IP 流程，ipamd 会在申请到 IP 以后立刻发送 garp ； 2. 优化释放 IP 时候获取 mac 的流程。	2021年12月14日
21.10.1	Bugfix	1. 修复固定 IP 意外释放导致 StatefulSet Pod 不可用的问题； 2. 修复 CNI 抢占文件锁超时导致释放 IP 失败的问题； 3. IPAMD 插件开启后，将默认使用其管理 IP。	2021年11月4日
21.07.1	Bugfix	解决部分节点无法获取 CNI 版本问题	2021年7月1日
21.06.1	Feature	支持 Pod 固定 IP（固定 IP 使用方法）	2021年6月23日
21.01.3	Bugfix	兼容开启了弹性网卡的UHost节点，解决其无法出外网的问题	2021年1月29日
21.01.2	Feature	将 Pod 的默认 MTU 设置为1452	2021年1月15日
21.01.1	Enhancement	ipamd 申请 IP 机制优化	2021年1月1日
20.07.1	Enhancement	支持 garp 机制，优化 Pod 网络首包延时问题	2020年7月16日

文档更新可能滞后，最新版本请以产品页面为准。

CNI Ipamd预分配VPC IP实现原理和部署架构

背景与原理

受制于UCloud底层网络技术现状，Pod的VPC IP新申请后，需要先进行arping以确保流表已经下发并且没有冲突，该过程至少需要5s的时间，最长需要15s。也就是说，Pod至少需要5s时间才能创建出来，再加上镜像拉取以及各种初始化操作，Pod创建时间基本延长到了平均10s以上。这对于要求快速拉起和销毁的Pod来说，几乎是不可接受的。

并且，因为所有Pod的创建和销毁都需要调用VPC服务，在VPC服务因为各种因素不可达时，会导致Pod无法被创建或销毁，集群处于一个几乎不可用的状态，发布流程会被完全阻塞。

为了解决上述问题，在CNI层面，可以通过预先分配好一批VPC IP，维护VPC IP池，给新创建的Pod分配IP池中的VPC IP。由于池子中的IP已经提前从UNetwork API中申请好并完成arping操作，因此能够立即分配给Pod，缩短5-15s的Pod创建时间。

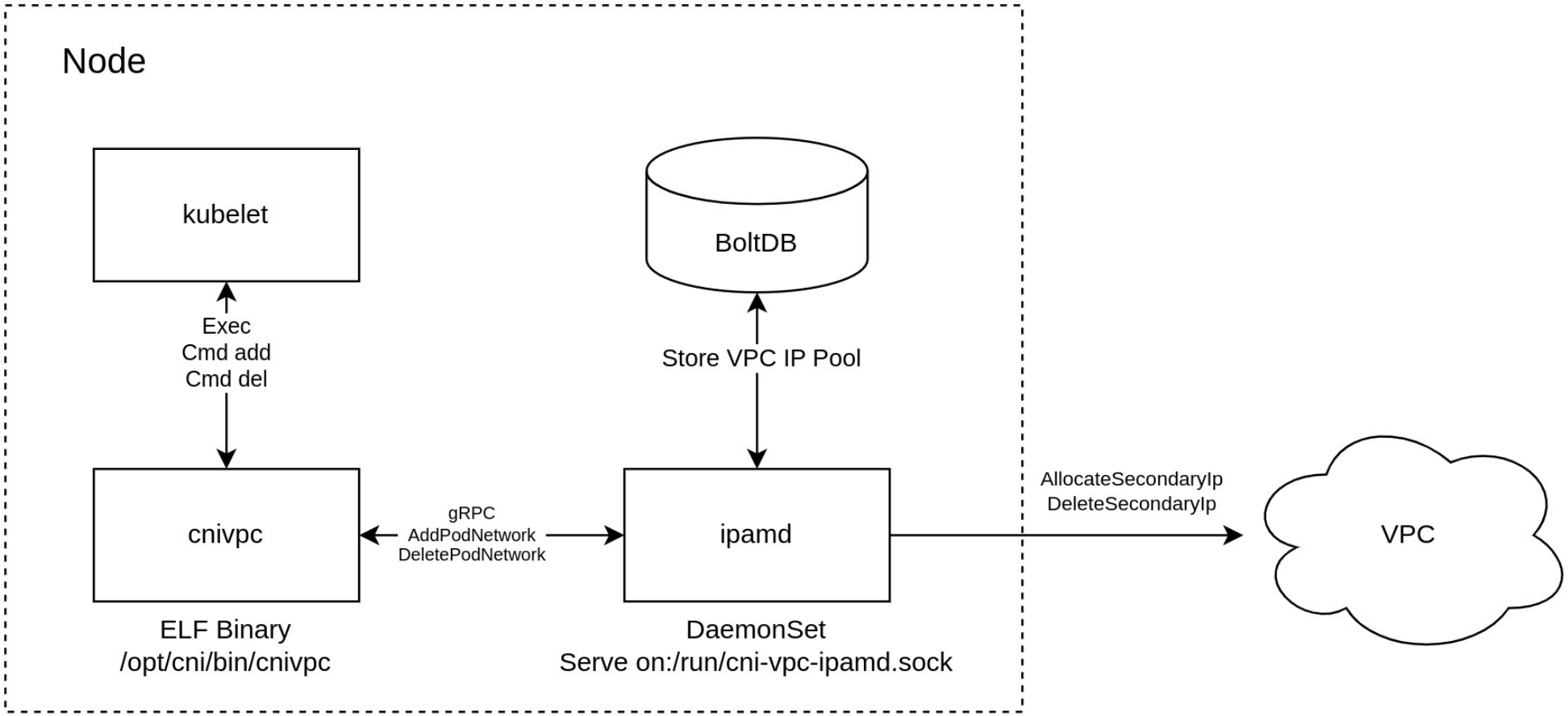
此外，在VPC挂掉的情况下，池子也能绕开VPC服务独自承担Pod IP的申请和回收（在池子中IP充足时），提高集群的可用性。

CNI预分配IP方案详解

新版CNI分成两大部分：

- cnivpc二进制文件。作为与Kubelet通信的入口。Kubelet通过调用该二进制可执行文件实现Pod网络的创建和摘除。
- ipamd服务。一个负责申请，维护，释放VPC IP的常驻Daemon进程，通过Unix Domain Socket向cnivpc提供分配和释放VPC IP的gRPC API。它作为DaemonSet部署在UK8S集群中。您可以将其理解为类似Calico IPAM的组件。

总体架构如下图所示。



核心流程包括：

1. ipamd内部包含了一个控制循环，它会定时从BoltDB（本地数据库）中检查目前可用的VPC IP是否低于IP池的低水位。如果低于低水位，则调用UNetwork AllocateSecondaryIp API给所在Node分配IP并将IP存入BoltDB中。如果VPC IP高于高水位，则调用UNetwork DeleteSeconaryIp将BoltDB中多出来的IP释放掉。
2. ipamd通过unix:/run/cni-vpc-ipamd.sock向cnivpc提供以下三个gRPC接口：
   - **Ping**: ipamd服务可用性探活接口。cnivpc每次申请，释放IP前都会调用该接口。如果失败，cnivpc的工作流程退化回原方案。
   - **AddPodNetwork**：给Pod分配IP接口，如果BoltDB IP池中存在可用IP，则直接从IP池中给Pod分配IP；否则立刻向UNetwork API申请IP，这种情况下，Pod启动依然需要5-15

秒。
    - **DelPodNetwork**：释放Pod IP接口。Pod被销毁后，IP会进入冷却状态，冷却30s之后，会被重新放回IP池子中。
3. ipamd服务如果被杀，它会响应Kubelet发送来的SIGTERM信号，停止gRPC服务并删除对应的Unix Domain Socket文件。
4. ipamd服务是可选组件，即使它异常终止了，cnivpc也能正常工作，但是会丧失预分配的能力。
5. 如果ipamd发现VPC的IP已经被分配完了，会尝试从其他同一子网的ipamd的池子中借用IP。如果其他ipamd也没有可用IP了，创建Pod会报错。

相关入口参数

- **--availablePodIPLowWatermark=3**: VPC IP预分配低水位，单位：个。默认为3
- **--availablePodIPHighWatermark=50**：VPC IP预分配高水位，单位：个。默认为50
- **--cooldownPeriodSeconds=30**: VPC IP冷却时间，Pod IP在被归还之后，需要经过冷却后才会被重新放回池子中，这是为了确保路由被销毁，单位：秒。默认为30s

部署方法

直接在集群中部署cni-vpc-ipamd.yml。

检查ipamd是否启动:

```
# kubectl get pod -o wide -n kube-system -l app=cni-vpc-ipamd
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
cni-vpc-ipamd-6v6x8 1/1 Running 0 59s 10.10.135.117 10.10.135.117 <none> <none>
cni-vpc-ipamd-tcc5b 1/1 Running 0 59s 10.10.7.12 10.10.7.12 <none> <none>
cni-vpc-ipamd-zsspc 1/1 Running 0 59s 10.10.183.70 10.10.183.70 <none> <none>
```

注意：在1.20以上版本的集群中，ipamd将默认安装

调试

在安装了ipamd的集群中，您可以通过cnivpctl命令来查看集群中池子的情况。

登录任一一台节点（可以是master或node），使用下面的命令可以列出集群中使用了ipamd的节点，以及它们池子中IP的数量：

```
$ cnivpctl get node
NODE SUBNET POOL
192.168.45.101 subnet-dsck39bnlhu 9
192.168.47.103 subnet-dsck39bnlhu 4
```

可以见到，目前我的集群有两个节点，池子中分别有9个和4个IP。

通过cnivpctl get pool可以进一步查看某个节点池子中所有IP：

```
$ cnivpctl -n 192.168.45.101 get pool
IP RECYCLED COOLDOWN AGE
192.168.32.35 21h false 21h
192.168.34.138 21h false 21h
192.168.35.38 21h false 21h
192.168.36.86 21h false 21h
192.168.43.106 21h false 21h
192.168.43.227 <none> false 21h
```

```
192.168.45.207 21h false 21h
192.168.45.229 <none> false 21h
192.168.45.59 <none> false 21h
```

不加-n参数，可以看到所有池子的IP，通过-o wide可以同时列出节点：

```
$ cnivpctl get pool -owide
IP RECYCLED COOLDOWN AGE NODE
192.168.32.35 21h false 21h 192.168.45.101
192.168.34.138 21h false 21h 192.168.45.101
192.168.35.38 21h false 21h 192.168.45.101
192.168.36.86 21h false 21h 192.168.45.101
192.168.43.106 21h false 21h 192.168.45.101
192.168.43.227 <none> false 21h 192.168.45.101
192.168.45.207 21h false 21h 192.168.45.101
192.168.45.229 <none> false 21h 192.168.45.101
192.168.45.59 <none> false 21h 192.168.45.101
192.168.40.121 21h false 21h 192.168.47.103
192.168.43.73 <none> false 21h 192.168.47.103
192.168.44.59 <none> false 21h 192.168.47.103
192.168.45.19 <none> false 21h 192.168.47.103
```

通过cnivpctl get pod可以看到某个节点的Pod占用了哪些IP：

```
$ cnivpctl -n 192.168.47.103 get pod
```

```
NAMESPACE NAME IP AGE
kube-system coredns-798fcc8f9d-jzccm 192.168.42.69 21h
kube-system csi-udisk-controller-0 192.168.43.110 21h
kube-system metrics-server-fff8f8668-rgfmr 192.168.36.42 21h
default nginx-deployment-66f49c7846-gtpbk 192.168.40.14 21h
default nginx-deployment-66f49c7846-mfgp4 192.168.34.55 21h
default nginx-deployment-66f49c7846-s54jj 192.168.40.126 21h
kube-system uk8s-kubectl-68bb767f87-tpzng 192.168.42.53 21h
```

这个命令的输出有点类似于`kubectl get pod`，但是只会输出占用了VPC IP的Pod，不会列出`HostNetwork`或者使用其它网络插件的Pod。

同样，不加`-n`参数，也可以列出所有Pod：

```
$ cnivpctl get pod -owide
NAMESPACE NAME IP AGE NODE
kube-system coredns-798fcc8f9d-gdrbq 192.168.41.246 21h 192.168.45.101
kube-system coredns-798fcc8f9d-jzccm 192.168.42.69 21h 192.168.47.103
kube-system csi-udisk-controller-0 192.168.43.110 21h 192.168.47.103
kube-system metrics-server-fff8f8668-rgfmr 192.168.36.42 21h 192.168.47.103
default nginx-deployment-66f49c7846-gtpbk 192.168.40.14 21h 192.168.47.103
default nginx-deployment-66f49c7846-mfgp4 192.168.34.55 21h 192.168.47.103
default nginx-deployment-66f49c7846-s54jj 192.168.40.126 21h 192.168.47.103
kube-system uk8s-kubectl-68bb767f87-tpzng 192.168.42.53 21h 192.168.47.103
```

`cnivpctl`除了以上常用命令，还有一些高级用法，例如：

- `cnivpctl get unuse`:列出泄漏的IP。
- `cnivpctl pop <node> [ip]`:从指定节点的池子中弹出一个IP。
- `cnivpctl push <node> [ip]`:分配一个新的IP给指定节点的池子。
- `cnivpctl release <node> [ip]`:释放指定节点中泄漏的IP(危险操作,谨慎执行)。

关于这个命令更详细的用法,参考`cnivpctl -h`。

常见疑问

Q:如何配置VPC IP池水位大小?

A: 可以通过ipamd程序入口参数--availablePodIPLowWatermark和--availablePodIPHighWatermark配置,例如:

```
containers:
- name: cni-vpc-ipamd
image: uhub.service.ucloud.cn/uk8s/cni-vpc-ipamd:1.2.3
args:
- "--availablePodIPLowWatermark=3"
- "--availablePodIPHighWatermark=50"
- "--calicoPolicyFlag=true"
- "--cooldownPeriodSeconds=30"
```

注意: 如果VPC IP池水位较低,节点突然被调度到大量Pod时,VPC IP池可用IP耗尽后新Pod使用的IP为最新从UNetwork API中申请的VPC IP,此时Pod依然需要经历数秒才能访问托管区;如果VPC IP水位池较高,集群节点数量较大,可能会导致子网空间IP耗尽,无法分配新VPC IP。

另外，请确保availablePodIPLowWatermark小于等于availablePodIPHighWatermark，否则ipamd启动会报错！

Q: VPC服务挂了之后，我的Pod创建和销毁会受到影响，该怎么配置ipamd来消除这种影响?

A: 是的，如果没有ipamd，VPC服务一旦出问题，您的集群Pod创建和销毁将会无法进行。ipamd被设计出来一部分原因就是为了解决这个问题。

但是，如果ipamd配置不合理，池子中常驻的IP数量太小，VPC挂了之后，ipamd还是无法完全承担Pod的IP分配任务。

如果您对集群的可用性要求较高，希望在UCloud VPC后台系统失联的情况下也能完全正常使用集群，可以调整ipamd的低水位参数availablePodIPLowWatermark，将其设置为您的节点最大Pod数量，例如110。这样，ipamd就会事先分配足够多的IP，能够承担当前节点所有Pod的创建和销毁。关于如何调整水位，参考上一节。

这样虽然ipamd会在一开始分配很多IP，但是稍后ipamd就能完全承担Pod IP的管理了。

注意：在您这么做之前，请确保节点所在子网足够大。否则ipamd会因为IP不足无法预分配期望数量的IP。

Q: ipamd占用了我太多IP!

A: 得益于ipamd的借用机制，即使您子网的IP被ipamd消耗完了，ipamd之间也可以互相调度IP。所以不需要担心IP的调度问题。

如果您确实不希望ipamd占用这么多IP，可以修改ipamd的两个水位参数，在最极端情况下，将它们修改为0，ipamd将不会预分配任何IP。

Q：如果节点BoltDB文件(/opt/cni/networkstorage.db)损坏，是否会导致VPC IP泄露?

A:会。如果发生这种情况，您可以登录任一节点，使用下面的命令扫描并列出某个节点上面泄漏的IP：

```
cnivpctl get unuse -n xx.xx.xx.xx
```

列出并确认这些IP没有被使用后，使用下面的命令清理并释放泄漏的IP：

```
cnivpctl release xx.xx.xx.xx
```

该命令会列出将要释放的IP供您二次确认，请确保它们没有被任何Pod使用，确认后ipamd会自动释放这些IP。

或者，您可以直接删除节点，其绑定的VPC IP会被自动释放。

Q: ipamd运行起来似乎有问题，怎么诊断?

A：cnivpc的调用日志是Node节点的/var/log/cnivpc.log; ipamd的日志可以通过kubectl logs观察，也可以在Node节点的/var/log/ucloud/下找到。

此外，kubelet的日志也通常必不可缺，登录Node节点执行

```
# journalctl -u kubelet --since="12:00"
```

观察kubelet运行日志。

kubectl get events也是您排查诊断问题的好帮手。

如果依然无法定位解决问题，请联系UK8S技术团队。

CNI 相关常见问题

1. CNI 插件升级为什么失败了?

1. 检查节点是否设置了污点及禁止调度(master 节点的默认污点不计算在内),带有污点的节点需要选择强制升级(不会对节点有特殊影响)。
2. 如果强制升级失败的,请重新点击 cni 强制升级。
3. 执行`kubectl get pods -n kube-system |grep plugin-operation`找到对应插件升级的 pod,并 describe pod 查看 pod 失败原因。

2. 为什么我的集群连不上外网?

UK8S 使用 VPC 网络实现内网互通,默认使用了 UCloud 的 DNS,wget 获取信息需要对 VPC 的子网配置网关,需要在 UK8S 所在的区域下进入VPC产品,对具体子网配置 NAT 网关,使集群节点可以通过 NAT 网关拉取外网数据,具体操作详见VPC创建NAT网关 。

Service 介绍

本章节主要为您简要介绍 Kubernetes 中的一个重要概念 Service(即服务,本文中两者等同),以及ULB的相关知识。

1. Service 介绍

Service 是 Kubernetes 集群中的一个资源对象,用于定义如何访问一组带有相同特征的Pods。通常情况下,Service 通过Label Selector 来确定目标Pods,ExternalName Services 例外,关于 Service 的详细介绍,请参阅官方文档中 Services章节

Kubernetes 提供了四种类型的 Service,分别用于不同的业务场景,默认的类型是 ClusterIp 。

ClusterIp

ClusterIp 是 Kubernetes 中默认的服务类型 (ServiceType),选择此种类型,对应的 Service 将被分配一个集群内部的 IP 地址,只能在集群内部被访问。

NodePort

在每台 Node 的固定端口上暴露服务,选择 NodePort 的服务类型,集群会自动创建一个 ClusterIp 类型的服务,负责处理Node接收到的外部流量。集群外部的 Client 可以通过 `<NodeIp>:<NodePort>` 的方式访问该服务。

LoadBalancer

通过集群外部的负载均衡设备来暴露服务,负载均衡设备一般由云厂商提供或者使用者自行搭建,在 UK8S中,我们通过 UCloud CloudProvider 插件集成了 ULB,后面会有专门的介绍。需要注意的是,创建一个 LoadBalancer 的 Service,集群会自动创建一个 NodePort 和 ClusterIp 类型的 Service,用于接收从 ULB 接入的流量。

ExternalName

将服务映射到一个 DNS 域名(如example.test.ucloud.cn),DNS 域名可通过 spec.externalName 参数配置。

2. ULB 简要介绍

ULB 提供了4层(基于IP+端口)和7层(基于 URL 等应用层信息)两种负载均衡类型,下表为4层和7层 ULB 的区别:

负载均衡类型	网络模式	支持协议
报文转发	内网、外网	TCP、UDP
请求代理	内网、外网	HTTP、HTTPS、TCP

如果你希望对 ULB 有深入的了解,请访问ULB 产品介绍,

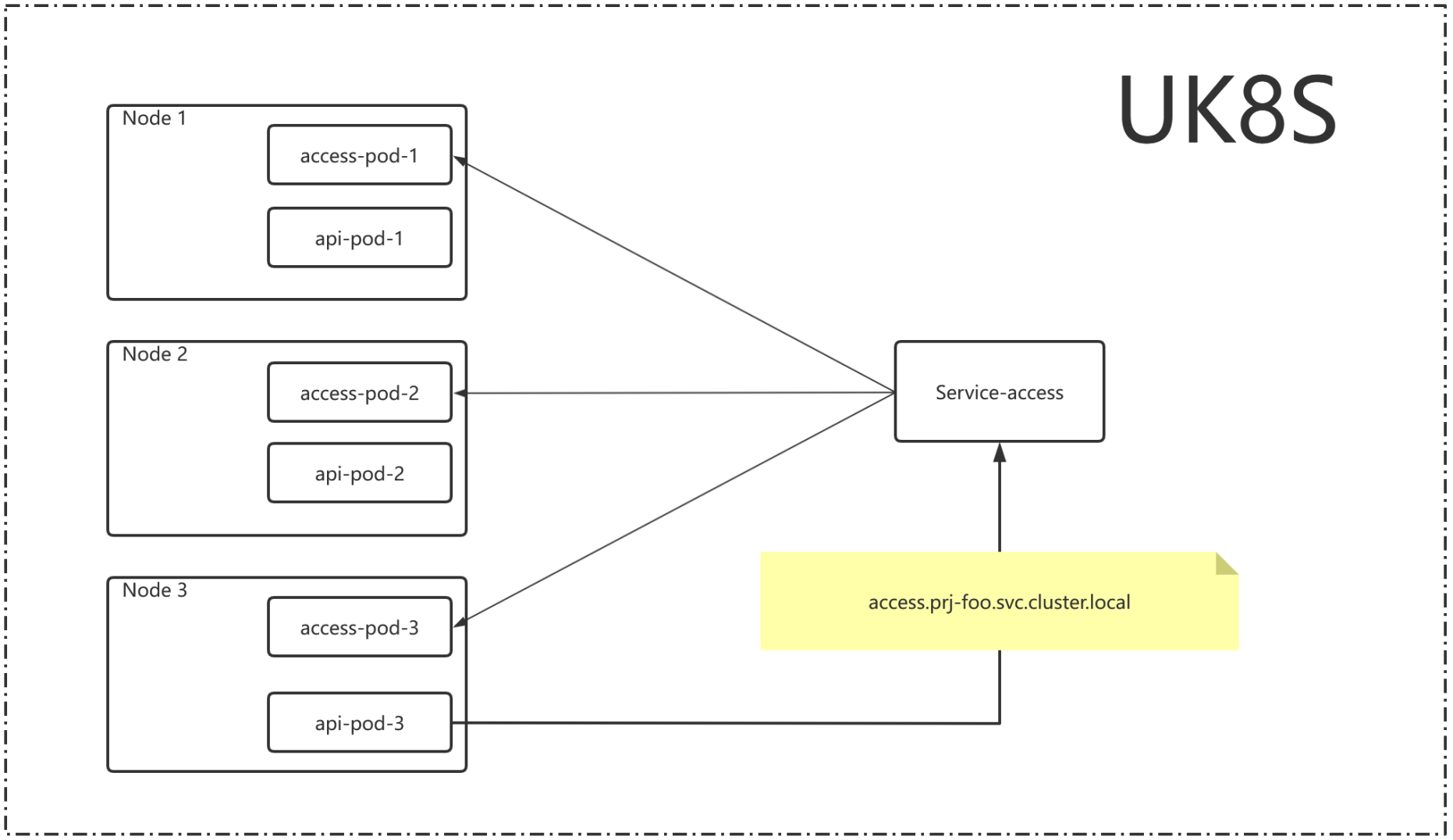
在 UK8S,我们同时支持报文转发及请求代理类型 ULB,你可以通过 Service yaml 中 Annotations 的形式自行配置 ULB 参数 。

下面我们分别介绍下如何通过 ULB 在内网、外网访问 Service。

集群内访问 Service

当我们在 UK8S 集群内部署好服务，配置了 svc 之后，如果访问服务的应用也在 k8s 集群内，则可以通过域名的方式访问服务

一、获取服务地址



当我们服务访问的发起端(我们称为 client,这里以 api-pod-3 为例) 和 服务的接收端 (这里我们称为 server,这里以 access 为例) 同时运行在 UK8S 中时,一般使用 k8s 域名访问

server 服务，k8s 会自动将流量转发到对应的 pod 中。

访问的地址如下：

```
[servicename].[namespace].[resourcetype].[clusterdomain]
```

- servicename：服务的名字，比如上面的 access
- namespace：服务所在的命名空间，上面对应 access 的命名空间 prj-foo
- resourcetype: 资源类型，访问类型为 service 时值统一为 svc
- clusterdomain：集群域名，在控制台，具体某一个 k8s 实例的详情中获取，概览->基本信息中的集群本地域名可以获取具体的值，一般为 `cluster.local`

二、服务访问示例

1. HTTP 服务访问

如果服务是 HTTP 服务，则我们可以通过 HTTP client 访问，对应的端口是 svc 配置的端口

```
curl http://access.prj-foo.svc.cluster.local:8080/
```

2. tcp 服务访问

同理，在 tcp 服务中，我们使用服务地址作为我们访问时的 host，服务的端口作为访问时的 port。

比如说 access 是一个 grpc server

```
func main() {
```

```
conn, err := grpc.Dial("access.prj-foo.svc.cluster.local:8080", grpc.WithInsecure())
...
defer conn.Close()

client := pb.NewSearchServiceClient(conn)
resp, err := client.Search(context.Background(), &pb.SearchRequest{
Request: "gRPC",
})
...
}
```

通过内网ULB访问Service

1. 使用提醒

- cloudprovider 版本 < 22.07.1

如果您的cloudprovider版本低于22.07.1，请勿修改由UK8S创建的ULB及Vserver的名称和备注，否则会导致Service异常无法访问。如果版本等于或高于22.07.1，则允许修改ULB的名称和备注（注意Vserver的依然不可更改）。如果您有修改ULB名称和备注的需求，请升级您的cloudprovider到最新版本，详见CloudProvider 插件更新。

- 相关ULB删除

如 ULB 为 UK8S 在创建 Service 时同步创建，删除 Service 时 ULB 会同步删除，请勿将 ULB 关联其他 Vserver，如需多个 Service 共用 ULB，可先创建 ULB，并在创建 Service 时关联已有 ULB，详情请见使用已有 ULB。

- ALB和NLB使用

目前请求代理型 CLB 存在一系列配额限制，可能会在使用过程造成服务问题，因此如有七层代理的需求，推荐您使用应用型负载均衡ALB；如有四层代理的需求，推荐您使用网络负载均衡NLB。如需使用 ALB 或 NLB 产品，请升级 cloudprovider 版本到 24.12.24 及以上，参考CloudProvider 插件更新。

- 参数修改

除外网EIP外，ULB相关参数目前均不支持Update修改，如不确认如何填写，请咨询UCloud 技术支持。

2. 使用UDP协议前必读

- 监控检查

目前ULB4针对UDP协议的健康检查支持ping和port两种模式，默认为ping，强烈推荐改为port；

- ping 健康检查须知

ping检查会发送目标IP为ulb-ip的ICMP Ping报文到后端节点。在UK8S的实现中，后端节点仅配置了针对UDP端口的网络包转发规则，而没有在网卡上绑定ulb-ip，因此无法响应以上ping报文，默认情况下ping健康检查会失败。如果需要使用ping健康检查，请参考 ULB文档 - 报文转发模式服务节点配置为后端节点绑定ulb-ip;

- port 健康检查须知

port健康检查的后端实现是对UDP端口发送UDP报文("Health Check" 字符串)和针对RS IP发送ICMP Ping报文。如果超时时间内回复了UDP报文则认为健康；如果超时时间没收到UDP回包，则以Ping的探测结果为准，因此您的应用程序需要响应UDP健康检查报文。

- **回包长度**

需要注意的是UDP回包长度不要超过1440，以避免可能的分片导致ULB4无法收到健康检查响应，导致健康检查失败。

3. 选择ULB4还是ULB7

ULB支持“报文转发(ULB4)”及“请求代理（ULB7）”两种转发模式，外网模式下推荐使用ULB4，因为ULB4的性能更好；请求代理（ULB7）转发模式建议选择应用型负载均衡ALB，传统型负载均衡CLB在使用过程中会受到一些传统型负载均衡配额限制，目前ALB是收费，具体价格请参考ULB计费说明。

4. 操作指南

4.1 通过ULB7对外暴露服务（http/https）

⚠ 使用 ALB 时，推荐升级 CloudProvider 版本到 >= 24.08.13。

UK8S在集群内可以直接使用 LoadBalancer 类型的Service，如果需要对外提供http/https协议，建议选择应用型负载均衡ALB；用户可以通过Service的"annotations"来配置ULB类型以及其他参数；更多参数信息可参考ULB参数说明。

负载均衡所使用的 SSL 证书的管理，请参见 ULB 文档：添加证书

如果用户选择ULB类型为外网时，还需要注意关于外网带宽设置，EIP计费模式的选择；

```
# 代表ULB网络类型，outer为外网，inner为内网；outer为默认值，此处可省略。
"service.beta.kubernetes.io/ucloud-load-balancer-type": "outer"
# bandwidth下默认为2Mpbs,建议显式声明带宽大小，避免费用超标。
"service.beta.kubernetes.io/ucloud-load-balancer-eip-bandwidth": "2"
# 付费模式，支持month，year，dynamic，默认为month
"service.beta.kubernetes.io/ucloud-load-balancer-eip-chargetype": "month"
# 付费时长，默认为1，chargetype为dynimic时无效
"service.beta.kubernetes.io/ucloud-load-balancer-eip-quantity": "1"
```

下面是内网ULB7的使用例子：

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-nginx-out-tcp-new
labels:
```

```
app: ucloud-nginx-out-tcp-new
annotations:
# 代表ULB类型，outer为外网，inner为内网；outer为默认值，此处可省略；
"service.beta.kubernetes.io/ucloud-load-balancer-type": "inner"
# 选择ULB类型，“application”表示使用应用型负载均衡ALB，其他类型参考：ULB参数说明；
"service.beta.kubernetes.io/ucloud-load-balancer-listentype": "application"
# 表示ULB协议类型，http与https等价，表示应用型负载均衡ULB7；如果选择了https协议，还配置证书与端口；
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol": "https"
# 声明要绑定的SSL证书Id，需要先将证书上传至UCloud；
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-cert": "ssl-qsmo0c7o9y1"
# 声明使用SSL协议的Service端口，多个用","分隔；
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-port": "443,8443"
# ALB目前是计费，默认是按月收费，用户可以调整付费方式；
"service.beta.kubernetes.io/ucloud-load-balancer-paymode": "month"
spec:
type: LoadBalancer
ports:
- protocol: TCP
port: 443
targetPort: 80
name: https
- protocol: TCP
port: 8443
targetPort: 80
name: ssl
```

```
- protocol: TCP
port: 80
targetPort: 80
name: http
selector:
app: ucloud-nginx-out-tcp-new
---
apiVersion: v1
kind: Pod
metadata:
name: test-nginx-out-tcp
labels:
app: ucloud-nginx-out-tcp-new
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
ports:
- containerPort: 80
```

4.2 通过ULB4对外暴露服务（TCP）

⚠ 使用 NLB 时,推荐升级 CloudProvider 版本到 >= 24.12.24。

对于TCP协议的服务，如果内网暴露只需要在metadata.annotations 指定 load-balancer-type为inner，外网load-balancer-type为outer，其他参数都有默认值，可不填写。

建议选择网络型负载均衡NLB；用户可以通过Service的"annotations"来配置ULB类型以及其他参数；更多参数信息可参考ULB参数说明。

下面是内网NLB的使用例子：

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-nginx-out-tcp-new
labels:
app: ucloud-nginx-out-tcp-new
annotations:
# ULB类型，默认为outer，支持outer、inner
"service.beta.kubernetes.io/ucloud-load-balancer-type": "inner"
# 选择ULB类型，“network”表示使用网络型负载均衡NLB
"service.beta.kubernetes.io/ucloud-load-balancer-listentype": "network"
# 用于声明ULB协议类型，并非应用协议，tcp和udp均代表ULB4，https和http均代表ULB7；
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol": "tcp"
# 对于ULB4而言，不论容器端口类型是tcp还是udp，均建议显式声明为port。
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-type": "port"
# 控制创建ULB所在子网，填写子网ID
"service.beta.kubernetes.io/ucloud-load-balancer-subnet-id": "subnet-xxxx"

spec:
type: LoadBalancer
```

```
ports:
- protocol: TCP
port: 80
targetPort: 80
selector:
app: ucloud-nginx-out-tcp-new
---
apiVersion: v1
kind: Pod
metadata:
name: test-nginx-out-tcp
labels:
app: ucloud-nginx-out-tcp-new
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
ports:
- containerPort: 80
```

4.3 通过ULB4对外暴露服务（UDP）

⚠ 使用 NLB 时,推荐升级 CloudProvider 版本到 >= 24.12.24。

如果你的应用是UDP协议，则务必显式声明健康检查的类型为port（端口检查），否则默认为ping，可能导致ULB误认为后端业务不正常。如果需要外网暴露请注意修改 ucloud-load-balancer-type 为 outer

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-inner-udp-new
labels:
app: ucloud-inner-udp-new
annotations:
# ULB类型，默认为outer，支持outer、inner
"service.beta.kubernetes.io/ucloud-load-balancer-type": "inner"
# 选择ULB类型，“network”表示使用网络型负载均衡NLB
"service.beta.kubernetes.io/ucloud-load-balancer-listentype": "network"
# 表示ULB协议类型，tcp和udp均代表ULB4，https和http均代表ULB7；
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol": "udp"
# 对于ULB4而言，不论容器端口类型是tcp还是udp，均建议显式声明为port。
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-type": "port"
# 控制创建ULB所在子网，填写子网ID
"service.beta.kubernetes.io/ucloud-load-balancer-subnet-id": "subnet-xxxx"
spec:
type: LoadBalancer
ports:
- name: udp
protocol: UDP
```

```
port: 53
targetPort: 53
selector:
app: ucloud-inner-udp-new
---
apiVersion: v1
kind: Pod
metadata:
name: test-inner-udp
labels:
app: ucloud-inner-udp-new
spec:
containers:
- name: dns
image: uhub.service.ucloud.cn/library/coredns:1.4.0
ports:
- name: udp
containerPort: 53
protocol: UDP
```

4.4 通过ULB4对外暴露服务（TCP和UDP协议混用）

⚠ 使用 NLB 时，推荐升级 CloudProvider 版本到 >= 24.12.24。

24.03.5以后的 CloudProvider 插件，当"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol"为tcp/udp或省略时，ULB4同时支持TCP和UDP两种协议。下文示例中，对外暴露了两个端口，其中80端口使用TCP协议，53端口使用UDP协议。

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-nginx-out-tcp-new
labels:
app: ucloud-nginx-out-tcp-new
annotations:
# 代表ULB网络类型，outer为外网，inner为内网；outer为默认值，此处可省略。
"service.beta.kubernetes.io/ucloud-load-balancer-type": "inner"
# 选择ULB类型，“network”表示使用网络型负载均衡NLB
"service.beta.kubernetes.io/ucloud-load-balancer-listentype": "network"
# 表示ULB协议类型，tcp、udp与tcp/udp等价，表示ULB4；使用tcp/udp或省略时时，同时支持TCP和UDP协议
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol": "tcp/udp"
# 对于ULB4而言，不论容器端口类型是tcp还是udp，均建议显式声明为port。
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-type": "port"
spec:
type: LoadBalancer
ports:
- name: tcp-default
protocol: TCP
port: 80
targetPort: 80
```

```
- name: udp
protocol: UDP
port: 53
targetPort: 53
selector:
app: ucloud-nginx-out-tcp-new
---
apiVersion: v1
kind: Pod
metadata:
name: test-nginx-out-tcp1
labels:
app: ucloud-nginx-out-tcp-new
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
ports:
- containerPort: 80
protocol: TCP
- name: dns
image: uhub.service.ucloud.cn/library/coredns:1.4.0
ports:
- name: udp
containerPort: 53
```

```
protocol: UDP
```

使用已有的ULB

UK8S支持在创建 LoadBalancer 类型的 Service 时，指定使用已有的 ULB 实例，而不是创建一个新的ULB实例。

也支持多个Service复用一个ULB实例，但存在以下规则限制：

1. 已有的 ULB 实例，必须是你自行创建的 ULB 实例，不能是 UK8S 插件创建出来的，否则会导致 ULB 被意外删除（在UK8S内删除Service，ULB也会被同步删除）。
2. 多个Service复用一个ULB实例时，Service端口不能冲突，否则新Service无法创建成功。
3. 指定已有的ULB实例创建LoadBalancer Service，Service被删除后，ULB实例不会被删除，仅删除对应的Vserver。
4. 通过UK8S创建的Vserver命名规范为`Protocol-ServicePort-ServiceUUID`，请勿随意修改，否则可能导致脏数据。

下面我们来看下如何使用已有的ULB实例。

使用已有ALB

⚠ 使用 ALB 时，推荐升级 CloudProvider 版本到 >= 24.08.13。

```
apiVersion: v1
kind: Service
metadata:
```

```
name: https
labels:
app: https
annotations:
"service.beta.kubernetes.io/ucloud-load-balancer-id": "alb-rpfirtgx4l4" # 替换为自己的alb id
# 申明使用alb
"service.beta.kubernetes.io/ucloud-load-balancer-listentype": "application"
# 申明使用http协议
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol": "http"
spec:
type: LoadBalancer
ports:
- protocol: TCP
port: 443
targetPort: 8080
name: https
- protocol: TCP
name: http
port: 80
targetPort: 8080
selector:
app: https
```

使用已有NLB

⚠ 使用 NLB 时,推荐升级 CloudProvider 版本到 >= 24.12.24。

```
apiVersion: v1
kind: Service
metadata:
name: https
labels:
app: https
annotations:
"service.beta.kubernetes.io/ucloud-load-balancer-id": "nlb-rpfirtgx4l4" # 替换为自己的nlb id
# 申明使用nlb
"service.beta.kubernetes.io/ucloud-load-balancer-listentype": "network"
# 申明使用tcp协议
"service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol": "tcp"
spec:
type: LoadBalancer
ports:
- protocol: TCP
port: 443
targetPort: 8080
```

```
name: https
- protocol: TCP
name: http
port: 80
targetPort: 8080
selector:
app: https
```

使用已有的内网ULB

声明使用已有的内网ULB，需要声明至少两个annotations。

```
apiVersion: v1
kind: Service
metadata:
name: https
labels:
app: https
annotations:
service.beta.kubernetes.io/ucloud-load-balancer-id: "ulb-ofvmd1o4" #替换成自己的ULB Id
service.beta.kubernetes.io/ucloud-load-balancer-type: "inner"
spec:
type: LoadBalancer
ports:
```

```
- protocol: TCP
port: 443
targetPort: 8080
selector:
app: https
```

使用已有的外网ULB（7层）

```
apiVersion: v1
kind: Service
metadata:
name: https
labels:
app: https
annotations:
service.beta.kubernetes.io/ucloud-load-balancer-id: "ulb-ofvmd1o4"
service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol: "https"
# http与https等价,均表示使用7层负载均衡
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-cert: "ssl-b103etqy"
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-port: "443"
# 443端口启用SSL,80端口依然为HTTP
spec:
type: LoadBalancer
```

```
ports:
- protocol: TCP
port: 443
targetPort: 8080
- protocol: TCP
port: 80
targetPort: 8080
selector:
app: https
```

使用已有的外网ULB（4层）

```
apiVersion: v1
kind: Service
metadata:
name: https
labels:
app: https
annotations:
service.beta.kubernetes.io/ucloud-load-balancer-id: "ulb-ofvmd1o4"
service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol: "tcp"
# 表示使用4层负载均衡
spec:
```

```
type: LoadBalancer
ports:
- protocol: TCP
port: 443
targetPort: 8080
- protocol: TCP
port: 80
targetPort: 8080
selector:
app: tcp
```

ULB 参数说明

本文主要描述用于创建 LoadBalancer 类型的 Service 时，与 ULB 相关的 Annotations 说明。

参数注意项

1. 目前除了外网 ULB 绑定的 EIP 的带宽值以外，其他参数暂时不支持修改，请谨慎配置。
2. 外网 ULB 绑定的 EIP 的带宽值，必须通过 Annotations 修改，Annotations 将会覆盖控制台修改的配置。

1. ULB 相关参数说明

注意：Service创建之后，以下参数不允许修改！！！

字段	默认值	说明
service.beta.kubernetes.io/ucloud-load-balancer-type	outer	负载均衡网络类型，枚举值为 inner / outer。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol	tcp	VServer 协议类型，枚举值为 tcp / udp / http / https。

service.beta.kubernetes.io/ucloud-load-balancer-listentype	/	负载均衡类型，枚举值为 packetstransmit / requestproxy / application / network。 1. ULB 类型为请求代理，VServer 协议为 tcp：该参数配置为 requestproxy，vserver-protocol 配置为 tcp； 2. ULB 类型为请求代理，VServer 协议为 http / https：无需配置该参数，自动按照 vserver-protocol 配置； 3. ULB 类型为报文转发，VServer 协议为 tcp / udp：无需配置该参数，自动按照 vserver-protocol 配置。 4. ULB类型为ALB，该参数配置为 application（仅24.03.13版本之后支持），vserver-protocol 可选 http / https； 5. ULB类型为NLB，该参数配置为network（仅24.12.24版本之后支持），vserver-protocol可选tcp / udp； 负载均衡类型及 VServer 协议类型详情说明请参见：负载均衡类型
service.beta.kubernetes.io/ucloud-load-balancer-vserver-method	roundrobin	VServer的负载均衡模式，枚举值为 roundrobin（轮询）、source（源地址）、consistenthash（一致性哈希）、sourceport（源地址计算端口）、consistenthashport（端口一致性哈希）、leastconn(最小连接数,clb4不支持)、backup(主备,alb独占)、sourcehash（源地址哈希，nlb独占）、weightleastconn（加权最小连接，nlb独占）。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-client-timeout	0/60	使用 ULB4 时，表示连接保持时间，单位为秒，取值 [60, 900]，0 表示禁用连接保持，默认为 0（不需要指定参数）。 使用 ULB7 时，表示空闲连接的回收时间，单位为秒，取值为 (0, 86400]，0 表示禁用连接保持，默认为 60。 指定参数后，persistence-type 不能为 none。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-session-persistence-type	none	会话保持方式，枚举值为 none（关闭）、serverinsert（自动生成 KEY）、userdefined（用户自定义 KEY）。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-port	/	开启 ssl 协议的端口，多个用 "," 分隔开，必须和 ssl-cert 同时指定。

service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-cert	/	ssl 证书 ID，必须和 ssl-port 同时指定，需要先将证书上传至 UCloud。 负载均衡所使用的 SSL 证书的管理，请参见 ULB 文档：添加证书
service.beta.kubernetes.io/ucloud-load-balancer-vserver-session-persistence-info	/	用户自定义 KEY，persistence-type 为 userdefined 时有效。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-type	port	健康检查方式，枚举值为 port / path / customize。 请求代理型 TCP 协议仅支持 port，其他协议支持 port 和 path；报文转发型 TCP 协议仅支持 port，UDP 协议支持 port 和 customize；ALB支持Port，HTTP；NLB支持Port，UDP
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-domain	/	clb7 的 monitor-type 为 path 时, 或者 alb 的 monitor-type 为 HTTP 时有效，指 http 检查域名。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-path	none	clb7 的 monitor-type 为 path 时, 或者 alb 的 monitor-type 为 HTTP 时有效，指 http 检查路径。 默认值为空，表示检查根路径/；如要自定义路径，只需提供根路径后面的部分，如health或status/ready
service.beta.kubernetes.io/ucloud-load-balancer-subnet-id	VPC 默认子网	创建 ULB 所在子网，填写子网 ID，如 subnet-xxxxxxxx。
service.beta.kubernetes.io/ucloud-load-balancer-remove-unscheduled-backend	true	移除不可被调度节点，枚举值 true / false，设置为 false 后节点不可调度时不会自动被 ULB 剔除。 仅在 21.04.1 及以后 cloudprovider 版本中支持。

service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-req msg	/	UDP 健康检查发出的请求报文，仅在 protocol 设置为 udp 时生效。 仅在 21.05.1 及以后 cloudprovider 版本中支持。
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-res pmsg	/	UDP 健康检查请求应收到的响应报文，仅在 protocol 设置为 udp 时生效。 仅在 21.05.1 及以后 cloudprovider 版本中支持。
service.beta.kubernetes.io/ucloud-load-balancer-paymode	month	alb和nlb付费模式，支持 month（按月付费）、year（按年付费）、dynamic（按时付费）。关于ALB付费相关参见:ALB 产品定价。关于NLB付费相关参见：NLB产品定价。
service.beta.kubernetes.io/ucloud-load-balancer-quantity	1	alb和nlb付费时长，chargetype 为 dynamic 时无需填写
service.beta.kubernetes.io/ucloud-load-balancer-firewall-id	/	在创建CLB时，绑定防火墙。默认不进行绑定

CloudProvider 版本查看及升级请参见：CloudProvider 插件更新

2. 外网 ULB 绑定 EIP 相关参数说明

注意：Service创建之后，以下参数仅service.beta.kubernetes.io/ucloud-load-balancer-eip-bandwidth可以修改，其它参数不允许修改！！！

字段	默认值	说明

service.beta.kubernetes.io/ucloud-load-balancer-eip-paymode	bandwidth	计费模式，支持 traffic（流量计费）、bandwidth（带宽计费）、sharebandwidth（共享带宽）。
service.beta.kubernetes.io/ucloud-load-balancer-eip-sharebandwidthid	/	共享带宽 ID，仅在 eip-paymode 为 sharebandwidth 时生效。
service.beta.kubernetes.io/ucloud-load-balancer-eip-bandwidth	2	外网带宽，单位为 Mbps。共享带宽模式下无需指定，或者配置为 0；流量计费模式下，该参数为流量计费 EIP 带宽上限。
service.beta.kubernetes.io/ucloud-load-balancer-eip-chargetype	month	付费模式，支持 month（按月付费）、year（按年付费）、dynamic（按时付费）。
service.beta.kubernetes.io/ucloud-load-balancer-eip-quantity	1	付费时长，chargetype 为 dynamic 时无需填写。

获取真实客户端IP

网络编程中如何获得对端IP

1. 如果是HTTP1.1协议，一般的反向代理或者负载均衡设备（如ULB7）支持X-Forwarded-For头部字段，会在用户的请求报文中加入类似**X-Forwarded-For:114.248.238.236**的头部。Web应用程序只需要解析该头部即可获得用户真实IP。
2. 如果是TCP或UDP自定义协议，可以客户端在协议字段里定义一个大端unsigned字段来保存自身IP，服务端把该字段解析出来然后调用inet_ntoa(3)等函数获得ipv4点分字符串。
3. 如果2中协议不支持填写自身IP，则服务端可以通过socket系统调用getpeername(2)来获取对端地址。下文讨论此方式。

Kubernetes Loadbalancer ULB4碰到的问题

UK8S使用ULB4和ULB7来支持Loadbalancer类型的Service。对于ULB7，由于只支持HTTP协议且默认支持X-Forwarded-For头部，所以Web服务可以很容易获取客户端的真实IP。但对于使用ULB4接入的纯四层协议的服务来说，可能需要使用getpeername(2)来获取客户端真实IP。然而由于目前kube-proxy采用Iptables模式，后端pod内的应用程序的网络库调用getpeername(2)会无法获得正确的IP地址。以下例子可以说明问题。

部署一个简单的webserver，通过Loadbalancer ULB4外网模式接入。

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-nginx
```

```
labels:
app: ucloud-nginx
annotations:
service.beta.kubernetes.io/ucloud-load-balancer-type: "outer"
service.beta.kubernetes.io/ucloud-load-balancer-vserver-method: "source"
spec:
type: LoadBalancer
ports:
- protocol: "TCP"
port: 80
targetPort: 12345
selector:
app: ucloud-nginx


---
apiVersion: v1
kind: Pod
metadata:
name: test-nginx
labels:
app: ucloud-nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/uk8s/uk8s-helloworld:stable
```

```
ports:
- containerPort: 12345
```

部署完毕后，Service状态如下所示，可以通过EIP 117.50.3.206访问该服务。

```
# kubectl get svc ucloud-nginx
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
ucloud-nginx LoadBalancer 172.17.179.247 117.50.3.206 80:43832/TCP 112s
```

服务本身的源码非常简单，只返回客户端地址，如下所示。

```
package main

import (
"fmt"
"io"
"log"
"net/http"
"net/http/httputil"
)

func main() {
log.Println("Server hello-world")
http.HandleFunc("/", AppRouter)
http.ListenAndServe(":12345", nil)
```

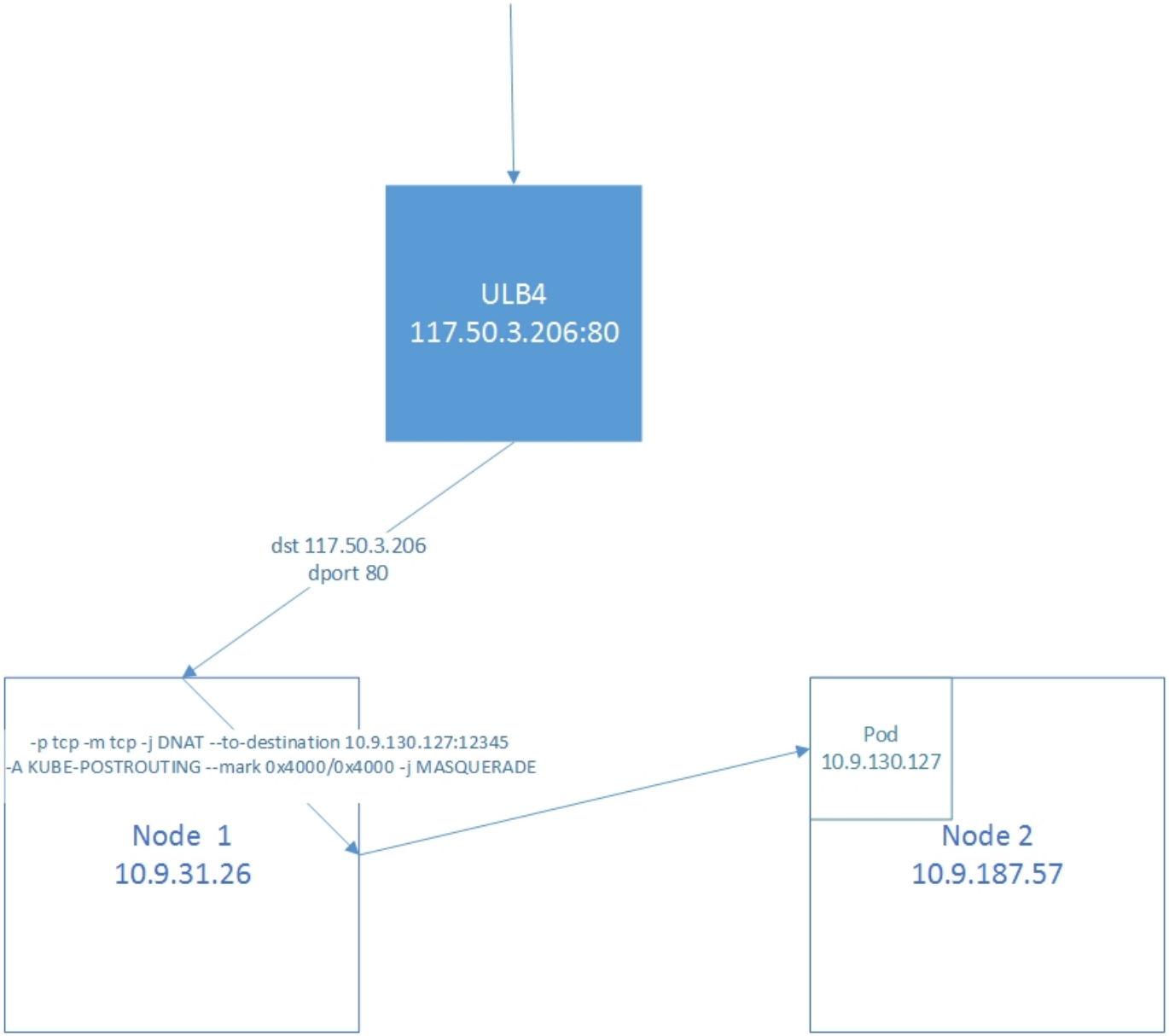
```
}

func AppRouter(w http.ResponseWriter, r *http.Request) {
dump, _ := httputil.DumpRequest(r, false)
log.Printf("%q\n", dump)
io.WriteString(w, fmt.Sprintf("Guest come from %v\n", r.RemoteAddr))
return
}
```

在外网通过浏览器访问该服务，如下所示。

结果显示用户的访问IP地址是一台云主机的内网IP地址，显然不正确。

原因解释

Loadbalancer创建成功后，ULB4的VServer将UK8S集群中的每个Node云主机节点作为自身的RS节点，RS端口为Service申明的Port值（注意不是NodePort）。ULB4将访问流量转发到其中一个RS后，RS根据本机上kube-proxy生成的iptables规则将流量DNAT到后端Pod中，如下所示。



图中ULB4先将流量转发到Node1中，Node1中根据iptables DNAT规则，将流量转发给Node2中的Pod。需要注意的是，Node1将IP包转发到Node2前，对这个包有一次SNAT操作。准确

地说，是一次MASQUERADE操作，规则如下。

```
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traffic requiring SNAT" -m mark --mark 0x4000/0x4000 -j MASQUERADE
```

这条规则将源地址改成Node1的本机地址，即10.9.31.26。当然，这个 SNAT 操作只针对本Service转发出来的包，否则普通的IP包也受到影响了，而判定IP包是否由本Service转发出来的依据是改包上是有有个"0x4000"标志，这个标志则是在DNAT操作前打上去的。

由于IP请求包的源地址被修改，Pod内的程序网络库通过getpeername(2)调用获取到的对端地址是Node1的IP地址而不是客户端真实的地址。

为什么需要对流出的包做SNAT操作呢?

原因比较简单。参考下图，当Node1上的Pod处理完请求后，需要发送响应包，如果没有SNAT操作，Pod收到的请求包源地址就是client的IP地址，这时候Pod会直接将响应包发给client的IP地址，但对于client程序来说，它明明没有往PodIP发送请求包，却收到来自Pod的IP包，这个包很可能会被client丢弃。而有了SNAT，Pod会将响应包发给Node1，Node1再根据DNAT规则产生的conntrack记录，将响应包通过返回给client。

```
client
\ ^
\ \
v \
ulb4
\ ^
\ \
v \
node 1 <--- node 2
| ^ SNAT
| | --->
v |
```

```
endpoint
```

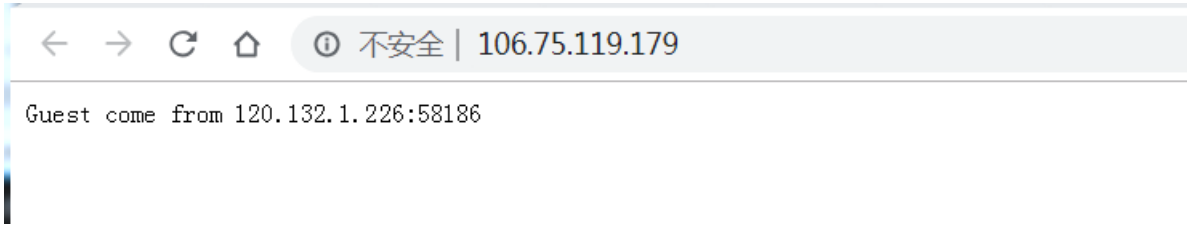
如何获取源IP?

对于Pod需要明确知道客户端来源地址的情况,我们需要显示地将Service的spec.externalTrafficPolicy设置成Local,如下修改。

```
apiVersion: v1
kind: Service
metadata:
name: ucloud-nginx
labels:
app: ucloud-nginx
annotations:
service.beta.kubernetes.io/ucloud-load-balancer-type: "outer"
service.beta.kubernetes.io/ucloud-load-balancer-vserver-method: "source"
spec:
type: LoadBalancer
ports:
- protocol: "TCP"
port: 80
targetPort: 12345
selector:
app: ucloud-nginx
```

```
externalTrafficPolicy: Local
```

重新部署服务后，再用浏览器访问，可以发现Pod正确获取了浏览器的访问IP。



而这个机制的原理也很简单，当设置了externalTrafficPolicy为Local时，Node上的iptables规则会设置只将IP包转发到在本机上运行的Pod，如果本机上无对应Pod在运行，此包将被DROP。如下图，这样Pod可以直接使用client的源地址进行回包而不需要SNAT操作。

```
client
\ ^
\ \
v \
ulb4
^ / \
/ / \ VServer健康检查失败
/ v X
node 1 node 2
^ |
| |
| v
endpoint
```

对于其他未运行Service对应Pod的Node节点来说，ULB VServer对其健康检查探测会因为iptables的DROP规则而失败，这样来自用户的请求永远不会被发往这些节点上，可以确保这些请求都能被正确响应。

概览 服务节点

添加节点 启用 禁用 删除

服务节点	资源ID	内网IP	健康检查	节点模式	操作
uk8s-s1ro3ezy-node-5	uhost-jtpsbdx0	10.9.92.102:80	失效	启用	禁用 删除
uk8s-s1ro3ezy-node-6	uhost-e1exvg0l	10.9.41.116:80	正常	启用	禁用 删除
uk8s-s1ro3ezy-node-1	uhost-bbob1tel	10.9.31.26:80	失效	启用	禁用 删除
uk8s-s1ro3ezy-node-3	uhost-o5bawh4t	10.9.187.57:80	失效	启用	禁用 删除
uk8s-s1ro3ezy-node-4	uhost-b3z12sez	10.9.17.119:80	失效	启用	禁用 删除

Service 流量策略

Service 的流量策略：外部流量策略和内部流量策略，分别用来控制从外部源路由的流量和控制来自内部源的流量如何被路由；你可以设置.spec.externalTrafficPolicy和.spec.internalTrafficPolicy 字段来控制 Kubernetes 如何将流量路由到健康（“就绪”）的后端。

官方文档参考这里

extenalTrafficPolicy

extenalTrafficPolicy用于控制外部流量的如何被路由，有效值为 Cluster 和 Local。通常来自集群外部请求，默认情况下，源 IP 会做 SNAT，Pod 看到的源 IP 是 Node IP。

- Cluster（默认）：
  - 使用此值时，流量被路由到集群中的任意节点，无论该节点是否具有正在运行的服务Pod。如果节点没有服务的Pod，流量会被转发到有服务的 Pod 的节点。
  - 使用此值时，会导致客户端的源 IP 被节点的 IP 屏蔽，如不需要保留源 IP 时可选择此有效值。
- Local：
  - 使用此值时，只将IP包转发到在本机上运行的服务Pod。如果接收流量的节点没有服务的 Pod，则流量将被丢弃。
  - 使用此值时，保留了客户端的源 IP。当保留源 IP 至关重要时，例如出于日志记录或安全目的，可选此值。

internalTrafficPolicy

internalTrafficPolicy用于控制来自内部源的流量如何被路由，处理来自集群内部的流量，有效值为 Cluster 和 Local。

- Cluster(默认):
  - 使用此值时,在所有节点访问内部流量都可以访问,如果节点上没有服务Pod,会将内部流量转发到有准备就绪的服务Pod的节点。
  - 使用此值时,会导致客户端的源 IP 被节点的 IP 屏蔽,如不需要保留源 IP 时可选择此有效值。
- Local :
  - 使用此值时,节点仅会将流量路由到本地节点上准备就绪的服务Pod,如果节点没有访问服务Pod,节点会丢弃该流量。
  - 开启内网网络策略之后,即使其他节点上面有正常工作的服务Pod,在自己的节点如果没有准备就绪的服务Pod,流量也是无法正常转发。

ULB中流量策略

externalTrafficPolicy: Cluster

当选择Cluster有效值时,不论Pod是否运行在该节点,ULB的VServer中的服务节点会包含集群中所有node节点;在这种情况下集群内其他节点也可以正常访问ULB的IP。

在一个集群内有2个节点,服务只部署一个实例Pod,查看ULB可以看到VServer的服务节点有2个。

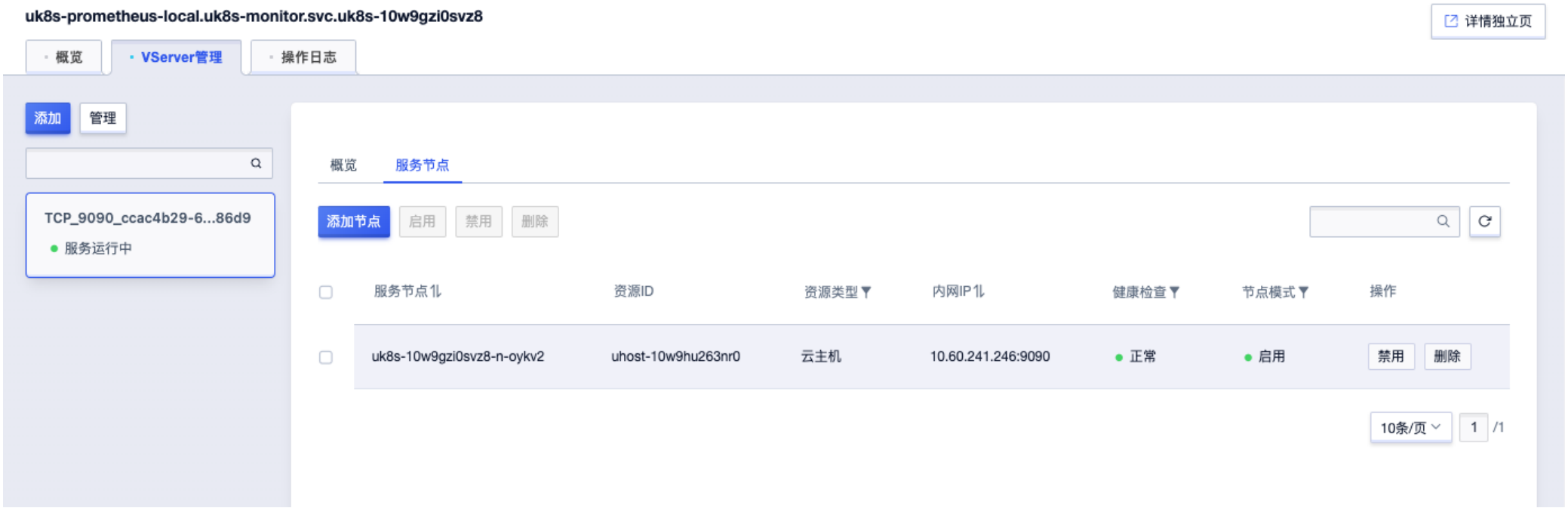
```
user@[redacted] Downloads % kubectl -n uk8s-monitor get po prometheus-uk8s-prometheus-0 -o wide
NAME                           READY   STATUS    RESTARTS   AGE    IP              NODE            NOMINATED NODE   READINESS GATES
prometheus-uk8s-prometheus-0   2/2     Running   0          5d2h   10.60.172.192   10.60.241.246   <none>           <none>
```

externalTrafficPolicy: Local

当选择Local值时，ULB的VServer 中的服务节点只会包含集群中服务Pod所在的node节点。

注意集群内服务访问ULB的IP时，请求会直接在集群内被转发而不会发送到ULB。因此Local模式下，仅当需要访问Service的客户端Pod与Service的后端服务在同一个节点上时，可以成功访问。其他节点上的Pod均无法访问ULB IP。

可以看到当Service的externalTrafficPolicy设置为Local时，ULB的后端只有一个节点。


uk8s-prometheus-local.uk8s-monitor.svc.uk8s-10w9gzi0svz8
详情独立页
概览
VServer管理
操作日志
添加
管理
TCP_9090_ccac4b29-6...86d9
服务运行中
概览
服务节点
添加节点
启用
禁用
删除
服务节点
资源ID
资源类型
内网IP
健康检查
节点模式
操作
uk8s-10w9gzi0svz8-n-oykv2
uhost-10w9hu263nr0
云主机
10.60.241.246:9090
正常
启用
禁用
删除
10条/页
1
/1

集群 ULB 误删处理

1. 前置操作

负载均衡(ULB)分内网和外网两种,在误删情况下,首先需要重建 **ULB**,并且保证原 **ULB IP** 地址不变。

- 对于内网 ULB,需要联系技术支持,非标创建指定内网 IP 的 ULB;
- 对于外网 ULB,需要创建指定 EIP 的 ULB(前提是原有 EIP 未被其他客户申请,详见:外网弹性IP),并在创建外网 ULB 时,选择绑定该 EIP。

2. Master ULB 误删

Master ULB,即集群 APIServer 绑定的 ULB,用于访问三台 Master 节点上 APIServer,内网 Master ULB 在创建集群时自动生成,如创建时开启外网 APIServer 则同时生成外网 Master ULB。

命名规则为 uk8s-xxxxxxxx-master-ulb4(内网 ULB)/ uk8s-xxxxxxxx-master-ulb4-external(外网 ULB)。

恢复步骤如下:

1. 创建 ULB 时类型需要指定为报文转发型;
2. 新建端口为 6443 的 VServer,指定类型为 TCP;
3. 添加三台 Master 节点为服务节点。

3. 集群 Service ulb 误删

1. 创建 ULB 时 ULB 类型需要与 Service 的类型相匹配，服务类型为 TCP/UDP 时指定报文转发，为 HTTP/HTTPS 时指定请求代理类型；
2. 删除集群内原 Service；
3. 根据文档重新绑定 ULB 和 Service：使用已有ULB创建服务。

ULB 相关常见问题处理

1. 如何区别使用ULB4还是ULB7?

通过service.metadata.annotations中的service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol值来声明使用ULB4还是ULB7。

如果值为tcp或者udp则使用ULB4。

如果是http或者https,则使用ULB7。

2. 如何区别ULB7是不是ALB

通过查看service.metadata.annotations中service.beta.kubernetes.io/ucloud-load-balancer-listentype是否为application来确定是alb,或者在service上找到对应LB对应的ID,查看是否在控制台上LB产品页面的应用型负载均衡页面内。

3. 使用 ULB4 时 Vserver 为什么会有健康检查失效

1. 如果 svc 的 externalTrafficPolicy 为 Local 时,这种情况是正常的,失败的节点表示没有运行对应的 pod
2. 需要在节点上抓包 tcpdump -i eth0 host <ulb-ip> 查看是否正常, ulb-ip 替换为 svc 对应的实际 ip
3. 查看节点上 kube-proxy 服务是否正常 systemctl status kube-proxy
4. 执行iptables -L -n -t nat |grep KUBE-SVC 及 ipvsadm -L -n查看转发规则是否下发正常

4. ULB4 对应的端口为什么不是 NodePort 的端口

1. K8S 节点上对 ulb_ip+serviceport 的端口组合进行了 iptables 转发，所以不走 nodeport
2. 如果有兴趣，可以通过在节点上执行iptables -L -n -t nat 或者ipvsadm -L -n 查看对应规则

5. 我想在删除LoadBalancer类型的Service并保留EIP该怎么操作?

修改 Service 类型，原 type: LoadBalancer 修改为 NodePort 或者 ClusterIP，再进行删除 Service 操作，EIP 和 ULB 会保留。

因该操作 EIP 和 ULB 资源将不再受 UK8S 管理，所以需要手动至 ULB 和 EIP 页面进行资源解绑和绑定。

6. 更改报文转发ULB的EIP之后在uk8s不生效

如果uk8s中某个Service绑定了报文转发类型的ULB，而用户手动更改了ULB的EIP，会发现无法通过新的EIP来访问Service。

这是因为cloudprovider组件无法监听到ULB的变动，没有触发对账流程将新的EIP写入iptables以实现转发。

如果您遇到该问题，可以在修改EIP之后使用下面的命令重启cloudprovider以强制触发对账流程：

```
kubectl -n kube-system rollout restart deploy cloudprovider-ucloud
```

注意，在重启之后，可能需要2分钟左右的时间生效。

7. Service换绑后原ULB无法重新绑定

当Service换绑ULB之后，原来的ULB会有vserver残留，这时候别的Service绑定原来ULB会报类似如下的错误：

```
Error syncing load balancer: failed to ensure load balancer: vserver(s) have already been created for one or more ports
```

如果遇到，需要手动将原来ULB中的vserver清理掉。

8. 如何手动更改Service绑定ULB的EIP

1. 在控制台的ULB管理页面，手动变更对应ULB的EIP。
2. 在Service中，增加service.beta.kubernetes.io/ucloud-load-balancer-id字段，设置为对应的ULB ID。
3. 等待一段时间，观察Serivce中的EIP相关Annotation是否更新。

9. 一个LoadBalancer的Service是否支持多端口?

支持，UK8S会为service.spec.ports下每个ServicePort分别创建一个VServer，VServer端口为Service端口。

10. 是否支持多协议?

目前UK8S CloudProvider 插件版本 >= 19.05.3 同时支持HTTP和HTTPS协议，cloudprovider 插件版本 >= 24.03.5 也支持UDP和TCP协议混用。

11. 如果Loadbalancer创建外网ULB后，用户在ULB控制台页面绑定了新的EIP，会被删除吗?

只有访问SVC的ExternalIP才能把流量导入后端Pod，访问其他EIP无效。删除SVC时，所有EIP都会被删除。

12. ULB配置迁移怎么操作?

因配置迁移实际为监听器迁移，故针对所有用到该ULB监听器的Service资源，需新建一个使用已有ALB的Service

将service.beta.kubernetes.io/ucloud-load-balancer-id-provision改为service.beta.kubernetes.io/ucloud-load-balancer-id值为迁移后的albid，新增service.beta.kubernetes.io/ucloud-load-balancer-listentype值为application

- 如果监听器为https协议则需额外配置ssl参数service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol、service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-cert与service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-port

以下样例展示了ulb配置迁移了一个带有80，443，8443端口监听器的ulb，其中80为http，443以及8443为https，原有Service如下：

```
# 迁移前的Service
apiVersion: v1
kind: Service
metadata:
annotations:
# 迁移前的ulbid
service.beta.kubernetes.io/ucloud-load-balancer-id-provision: ulb-xxx
service.beta.kubernetes.io/ucloud-load-balancer-paymode: month
service.beta.kubernetes.io/ucloud-load-balancer-type: inner
```

```
service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol: https
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-cert: ssl-xxx
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-port: 443,8443
name: service1
namespace: default
spec:
ports:
- name: https
port: 443
protocol: TCP
targetPort: 80
- name: ssl
port: 8443
protocol: TCP
targetPort: 80
- name: http
port: 80
protocol: TCP
targetPort: 80
selector:
app: service1
type: LoadBalancer
```

对80以及443端口进行ulb配置迁移后新建的Service如下：

```
apiVersion: v1
kind: Service
metadata:
annotations:
# 迁移后的albid
service.beta.kubernetes.io/ucloud-load-balancer-id: alb-xxx
# “application”表示使用应用型负载均衡ALB
service.beta.kubernetes.io/ucloud-load-balancer-listentype: "application"
service.beta.kubernetes.io/ucloud-load-balancer-paymode: month
service.beta.kubernetes.io/ucloud-load-balancer-type: inner
service.beta.kubernetes.io/ucloud-load-balancer-vserver-protocol: https
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-cert: ssl-xxx
service.beta.kubernetes.io/ucloud-load-balancer-vserver-ssl-port: 443
name: service2
namespace: default
spec:
ports:
- name: http
port: 80
protocol: TCP
targetPort: 80
- name: https
port: 443
protocol: TCP
```

```
targetPort: 80
selector:
app: service2
type: LoadBalancer
```

创建Service资源后同时建议业务侧将原先Servive的服务流量进行切换到该新建Service上。

Ingress 支持

本文介绍如何在 UK8S 中安装和配置基于 nginx 的 Ingress Controller 。

不同版本的 Kubernetes 安装配置方法有所不同。

如果您的 Kubernetes 版本为 1.26 及以上，请参考 kubernetes Nginx Ingress 安装配置说明 (1.26 ~ latest)。

如果您的 Kubernetes 版本为 1.19 ~ 1.25，请参考 kubernetes Nginx Ingress 安装配置说明 (1.19 ~ 1.25)。

如果您的 Kubernetes 版本为 1.13 ~ 1.18，请参考 kubernetes Nginx Ingress 安装配置说明 (1.13 ~ 1.18)。

如果您想了解 Ingress 的高级用法，请参考 Ingress 高级用法

Nginx Ingress

本文适用于的 K8S 版本为 1.26+，对于不同版本，请参考：Ingress 支持

什么是 Ingress

Ingress 是从 Kubernetes 集群外部访问集群内部服务的入口，同时为集群内的 Service 提供七层负载均衡能力。

一般情况下，Service 和 Pod 仅可在集群内部通过 IP 地址访问，所有到达集群边界的流量或被丢弃或被转发到其他地方，前面的 service 章节我们提供了创建 LoadBalancer 类型的 Service 方法，借助于 Kubernetes 提供的扩展接口，UK8S 会创建一个与该 Service 对应的负载均衡服务即 ULB 来承接外部流量，并路由到集群内部。但在诸如微服务等场景下，一个 Service 对应一个负载均衡器，管理成本明显过高，Ingress 因此应运而生。

我们可以把 Ingress 理解为 Service 提供能力的 “Service”，为后端不同 Service 提供代理的负载均衡服务，我们可以在 Ingress 配置可供外部访问 URL、负载均衡、SSL、基于名称的

虚拟主机等。

下面我们通过在 UK8S 中部署 Nginx Ingress Controller，来了解一下 Ingress 的使用过程。

一、部署 Ingress Controller

为了使 Ingress 正常工作，集群内必须部署 Ingress Controller。与其他类型的控制器不同，其他类型的控制器如 Deployment 通常作为 kube-controller-manager 二进制文件的一部分，在集群启动时自动运行。而 Ingress Controller 则需要自行部署，Kubernetes 社区提供了以下 Ingress Controller 供选择，分别如下：

1. Nginx
2. HAProxy
3. Envoy
4. Traefik

这里我们选择 Nginx 作为 Ingress Controller，部署 Nginx Ingress Controller 非常简单，执行以下指定即可。

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/ingress_nginx/mandatory_1.26.yaml
```

在 mandatory_1.26.yaml 这个文件里，是 Ingress Controller 的定义，我们可以把 yaml 文件下载到本地仔细研读下。这里简要简述下部分 yaml 字段的意义。

这个 yaml 定义了一个使用 ingress-nginx-controller 作为镜像的 pod 副本集，这个 Pod 主要功能就是监听 Ingress 对象以及它所代理的后端 Service 变化的控制器。当一个新的 Ingress 对象由用户创建后，控制器会根据 Ingress 对象所定义的内容，生成一份对应的 Nginx 配置文件（也就是我们熟知的 /etc/nginx/nginx.conf），并使用这个配置文件启动一个 Nginx 服务。而一旦 Ingress 对象被更新，控制器会更新这个配置文件。需要注意的是，如果只是被代理的 Service 对象被更新，控制器所管理的 nginx 服务不需要重新加载，这是因为 ingress-nginx-controller 通过 nginx lua 实现了 upstream 的动态配置。

另外，在这个 yaml 文件中，我们还看到定义了 ConfigMap，ingress-nginx-controller 可以通过 ConfigMap 对象来对 Nginx 配置文件进行定制，示例如下：

```
apiVersion: v1
data:
```

```
allow-snippet-annotations: "false"
kind: ConfigMap
metadata:
labels:
app.kubernetes.io/component: controller
app.kubernetes.io/instance: ingress-nginx
app.kubernetes.io/name: ingress-nginx
app.kubernetes.io/part-of: ingress-nginx
app.kubernetes.io/version: 1.10.5
name: ingress-nginx-controller
namespace: ingress-nginx
```

需要注意的是，ConfigMap 中的 key 和 value 只支持字符串，因此对于整数等类型，需要使用双引号，例如"100"，详细资料见 Nginx-Ingress-ConfigMap。

本质上，这个 ingress-nginx-controller 是一个可以根据 Ingress 对象和被代理后端 Service 的变化，自动进行更新的 Nginx 负载均衡器。

容器默认使用 UTC 时间，如果要使用宿主机时区，参见 Pod 时区问题

二、集群外部访问 nginx ingress

上面我们已经在 UK8S 中部署了一个 nginx ingress controller，并且为了在集群外部可达，我们还创建了一个外部可达的 LoadBalancer 类型的 Service，如下所示。

```
apiVersion: v1
kind: Service
metadata:
annotations:
"service.beta.kubernetes.io/ucloud-load-balancer-type": "inner"
```

```
labels:
app.kubernetes.io/component: controller
app.kubernetes.io/instance: ingress-nginx
app.kubernetes.io/name: ingress-nginx
app.kubernetes.io/part-of: ingress-nginx
app.kubernetes.io/version: 1.10.5
name: ingress-nginx-controller
namespace: ingress-nginx
spec:
externalTrafficPolicy: Local
ports:
- appProtocol: http
name: http
port: 80
protocol: TCP
targetPort: http
- appProtocol: https
name: https
port: 443
protocol: TCP
targetPort: https
selector:
app.kubernetes.io/component: controller
app.kubernetes.io/instance: ingress-nginx
app.kubernetes.io/name: ingress-nginx
```

```
type: LoadBalancer
```

这个 Service 的唯一工作，就是将所有携带 ingress-nginx 标签的 Pod 的 80 和 433 端口暴露出去，我们可以通过以下方式获取到这个 Service 的外部访问入口。

需要注意的是样例中的 LoadBalancer 创建的 ULB 为内网模式，如果创建外网 ULB，需要将这个 Service 的 `metadata.annotations."service.beta.kubernetes.io/ucloud-load-balancer-type"` 改为 "outer"，更多参数请官方文档 ULB 参数说明。

```
$ kubectl get svc -n ingress-nginx
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
ingress-nginx-controller LoadBalancer 172.30.48.77 xxx.yy.xxx.yy 80:30052/TCP,443:31285/TCP 14m
......
```

部署完 Ingress Controller 和它所需要的 Service 后，我们就可以使用通过它来将集群内部的其他 Service 代理出去了。

三、创建两个应用

在下面的 yaml 中，我们定义了 2 个镜像名为 echo-nginx 的应用，主要是输出 nginx 应用自身的一些全局变量。

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: demo-app-1
labels:
app: demo-app-1
spec:
replicas: 2
```

```
selector:
matchLabels:
app: demo-app-1
template:
metadata:
labels:
app: demo-app-1
spec:
containers:
- name: demo-app-1
image: uhub.service.ucloud.cn/jenkins_k8s_cicd/echo_nginx:v11
ports:
- containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
name: demo-app-1-svc
spec:
ports:
- port: 80
targetPort: 80
protocol: TCP
name: http
selector:
```

```
app: demo-app-1
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: demo-app-2
labels:
app: demo-app-2
spec:
replicas: 3
selector:
matchLabels:
app: demo-app-2
template:
metadata:
labels:
app: demo-app-2
spec:
containers:
- name: demo-app-2
image: uhub.service.ucloud.cn/jenkins_k8s_cicd/echo_nginx:v11
ports:
- containerPort: 80
---
apiVersion: v1
```

```
kind: Service
metadata:
name: demo-app-2-svc
labels:
spec:
ports:
- port: 80
targetPort: 80
protocol: TCP
name: http
selector:
app: demo-app-2
```

我们将上述 yaml 保存为 demo-app.yaml，并执行如下命令创建应用。

```
kubectl apply -f demo-app.yaml
```

四、定义 Ingress 对象

我们已经部署了 nginx ingress controller 并将其暴露到外网，并且创建好了两个应用，接下来我们就可以定义一个 ingress 对象，来把两个应用代理出集群。

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: demo-app-ingress
```

```
spec:
ingressClassName: nginx
defaultBackend:
service:
name: demo-app-1-svc
port:
number: 80
rules:
- host: demo-app.example.com
http:
paths:
- path: /demo-app-1
pathType: Prefix
backend:
service:
name: demo-app-1-svc
port:
number: 80
- path: /demo-app-2
pathType: Prefix
backend:
service:
name: demo-app-2-svc
port:
number: 80
```

上述 yaml 文件定义了一个 Ingress 对象，其中 ingress.spec.rules 便是 ingress 的代理规则集合。

我们先看 host 字段，他的值必须是一个标准的域名格式字符串，而不能是 IP 地址。而 host 字段定义的值，就是这个 Ingress 的入口。这样就意味着，当用户访问demo-app.example.com的时候，实际访问到的就是这个 Ingress 对象。这样，Kubernetes 就能使用 IngressRule 来对你的请求进行下一步转发。

接下来是 path 字段。你可以简单理解为，这里的每个 path 都对应一个后端 Service。所以，我们例子里，定义了两个 path，他们分别对应 demo-app-1 和 demo-app-2 这两个 Deployment 的 Service（即：demo-app-1-svc 和 demo-app-2-svc）。

每条 http 规则包含以下信息：一个 host 配置项（比如demo-app.example.com），path 列表（比如：/demo-app-1和/demo-app-2），每个 path 都关联一个 backend（比如：demo-app-1-svc的 80 端口）。在 LoadBalancer 将流量转发到 backend 之前，所有的入站请求都要先匹配 host 和 path。

当没有匹配到规则中的任意一组 host 和 path 时，ingress.spec.defaultBackend 中定义的默认 backend 将会生效，将不匹配 host 和 path 的流量转发至 defaultBackend 处理。

我们将上述 yaml 保存为 ingress.yaml，并通过以下命令直接创建 ingress 对象：

```
kubectl apply -f ingress.yaml
```

接下来，我们可以查看这个 ingress 对象

```
$ kubectl get ingresses.networking.k8s.io
NAME CLASS HOSTS ADDRESS PORTS AGE
demo-app-ingress <none> demo-app.example.com 80 5m39s

$ kubectl describe ingresses.networking.k8s.io demo-app-ingress
Name: demo-app-ingress
Labels: <none>
Namespace: ingress-nginx
```

```
Address:
Default backend: demo-app-1-svc:80 (172.20.145.127:80,172.20.187.251:80)
Rules:
Host Path Backends
---- ---- --------
demo-app.example.com
/demo-app-1 demo-app-1-svc:80 (172.20.145.127:80,172.20.187.251:80)
/demo-app-2 demo-app-2-svc:80 (172.20.40.180:80,172.20.43.118:80,172.20.62.63:80)
Annotations: <none>
Events: <none>
```

从 rule 规则可以看到，我们的定义的 Host 是 demo-app.example.com，他有两条转发规则（Path），分别转发给 demo-app-1-svc 和 demo-app-2-svc。

当然，在 Ingress 的 yaml 文件里，你还可以定义多个 Host，来为更多的域名提供负载均衡服务。

接下来，我们就可以通过访问这个 Ingress 的地址和端口，访问到我们前面部署的应用了，比如，当我们访问 http://demo-app.example.com/demo-app-2 时，应该是 demo-app-2 这个 Deployment 负责响应我的请求，当创建 LoadBalance 时选择外网模式并绑定了 EIP，那么可以直接通过外网访问，我们在本地 /etc/hosts 中添加 Service 外网 ip 以及域名后即可直接从外网通过域名访问。如果是在内网模式下，仅 VPC 内的资源可以通过 Ingress 访问。

```
$ cat /etc/hosts
......

xxx.yy.xxx.yy demo-app.example.com
```

```
$ curl http://demo-app.example.com/demo-app-2
Scheme: http
```

```
Server address: 172.20.43.118:80
Server name: demo-app-2-5f6c5df698-rvsmc
Date: 12/Jan/2022:02:56:38 +0000
URI: /demo-app-2
Request ID: ba34c07f5cc78e74629041df5568977a
```

五、TLS 支持

在上述的 ingress 对象中，我们没有给 Host 指定 TLS 证书，ingress controller 支持通过指定一个包含 TLS 私钥以及证书的 secret 来加密站点。

我们先创建一个包含 tls.crt 以及 tls.key 的 secret，在生成证书的时候，需要确保证书的 CN 包含`demo-app.example.com`，并将证书内容使用 base64 编码。

```
$ HOST=demo-app.example.com

$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj "/CN=${HOST}/O=${HOST}"
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'tls.key'
```

```
$ kubectl create secret tls demo-app-tls --key tls.key --cert tls.crt
secret/demo-app-tls created

$ kubectl describe secret demo-app-tls
Name: demo-app-tls
```

```
Namespace: ingress-nginx
Labels: <none>
Annotations: <none>

Type: kubernetes.io/tls

Data
====
tls.crt: 1229 bytes
tls.key: 1708 bytes
```

然后在 ingress 对象中，通过 ingress.spec.tls 字段引用 secret，ingress controller 即会加密来保护客户端到 ingress 之间的通信管道。

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: demo-app-ingress
spec:
ingressClassName: nginx
tls:
- hosts:
- demo-app.example.com
secretName: demo-app-tls
defaultBackend:
service:
```

```
name: demo-app-1-svc
port:
number: 80
rules:
- host: demo-app.example.com
http:
paths:
- path: /demo-app-1
pathType: Prefix
backend:
service:
name: demo-app-1-svc
port:
number: 80
- path: /demo-app-2
pathType: Prefix
backend:
service:
name: demo-app-2-svc
port:
number: 80
```

此时可通过 https 协议访问 ingress，但需要注意的是由于 ingress 的证书是我们自签生成的，所以在使用 curl 等工具访问时，需要略过证书校验，生产环境中建议您使用经过 **CA** 机构签名的 **TLS** 证书。

```
$ curl --insecure https://demo-app.example.com/demo-app-2
Scheme: http
Server address: 172.20.40.180:80
Server name: demo-app-2-5f6c5df698-pvr45
Date: 12/Jan/2022:03:19:58 +0000
URI: /demo-app-2
Request ID: 4cc35ddb1a301977f0477ded4c09d5df
```

六、设置访问白名单

部分场景下，我们的业务只允许指定的 IP 地址访问，这可以通过添加 annotations 来实现，即 `nginx.ingress.kubernetes.io/whitelist-source-range`，值则是 CIDR 列表，多个 CIDR 间用 ","分开。示例如下：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: demo-app-ingress
annotations:
nginx.ingress.kubernetes.io/whitelist-source-range: 172.16.0.0/16,172.18.0.0/16
spec:
ingressClassName: nginx
tls:
- hosts:
- demo-app.example.com
```

```
secretName: demo-app-tls
defaultBackend:
service:
name: demo-app-1-svc
port:
number: 80
rules:
- host: demo-app.example.com
http:
paths:
- path: /demo-app-1
pathType: Prefix
backend:
service:
name: demo-app-1-svc
port:
number: 80
- path: /demo-app-2
pathType: Prefix
backend:
service:
name: demo-app-2-svc
port:
number: 80
```

ingress 高级用法

多个Ingress Controller SVC

如果您只运行了一个ingress controller，想通过多个ULB对外提供服务（例如需要在ULB中绑定SSL证书），可以参考这个yaml文件。

```
apiVersion: v1
kind: Service
metadata:
name: ingress-nginx2
namespace: ingress-nginx
labels:
app.kubernetes.io/name: ingress-nginx
app.kubernetes.io/part-of: ingress-nginx
spec:
type: LoadBalancer
ports:
- name: http
port: 80
targetPort: 80
protocol: TCP
- name: https
port: 443
targetPort: 443
```

```
protocol: TCP
selector:
app.kubernetes.io/name: ingress-nginx
app.kubernetes.io/part-of: ingress-nginx
```

这里我新建了一个把ingress controller暴露出集群的svc，名为ingress-nginx2，这时针对这个nginx ingress controller就拥有了2个svc,对应了2个ULB。

```
[root@10-10-10-194 ~]# kubectl get svc -n ingress-nginx
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
ingress-nginx LoadBalancer 172.17.23.246 xx.xx.xx.xx 80:32677/TCP,443:39787/TCP 10d
ingress-nginx2 LoadBalancer 172.17.7.114 yy.yy.yy.yy 80:47962/TCP,443:45958/TCP 29m
```

用户可以解析增加n1 xx.xx.xx.xx和n2 yy.yy.yy.yy进行区分流量入口，这个操作流程将使用同一套ingress controller，多个SVC的使用场景，逻辑如下图。

```
ULB1 ULB2
| |
ing_svc1 ing_svc2
| |
-----------------
|
ingress controller
|
-----------------
| |
app_svc1 app_svc2
```

```
| |
app_pod1 app_pod2
```

多个 Ingress Controller

如果您运行了多个ingress controller在您的kubernetes集群中(例如同时运行了nginx和traefik),则需要在使用ingress资源对象时进行声明操作,例如:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
name: traefik-web-ui
namespace: kube-system
annotations:
kubernetes.io/ingress.class: traefik
# 声明使用 traefik 作为指定的ingress controller
# 也可以替换成 nginx 等已安装的ingress controller
spec:
rules:
- host: traefik-ui.minikube
http:
paths:
- path: /
backend:
serviceName: traefik-web-ui
servicePort: web
```

如果您部署了不同类型的ingress controller(例如nginx和traefik),而不指定注释类型,将导致这两个或者所有的ingress controller都在努力满足ingress的需求,并且所有的ingress controller都在争抢更新ingress的状态。

使用 **DaemonSet** 部署

ingress-controller可以通过Deployment或DaemonSet部署,各有利弊:

- 使用Deployment时,可伸缩性可以更好,因为在使用DaemonSet时您将拥有每个节点的单一Pod模型,而在使用Deployment时,基于环境您可能需要更少的Pod。
- 当节点加入群集时,DaemonSet会自动扩展到新节点,而Deployment仅在需要时在新节点上进行调度。
- DaemonSet 确保只有一个节点有且只有一个副本。如果副本数要小于或大于集群节点数,建议通过 Deployment进行设置。

CloudProvider 插件更新

UK8S 通过 CloudProvider 插件，实现集群中 LoadBalancer 类型服务（SVC）与 UCloud ULB 产品的创建和绑定。

版本问题

如您集群 CloudProvider 为以下版本，请您务必及时按照文档更新，避免影响业务。升级过程不影响线上业务，但仍建议您在业务闲时进行更新，如有问题请及时与我们联系。

- 低于20.10.1版本

20.10.1 之前的版本存在 bug，会在 Service 重启时发生误判，导致重复创建 ULB 实例并写入集群，造成集群相应 Service 的不可用。为此我们对 CloudProvider 插件进行了优化，完善 ULB 相关接口的调用逻辑，避免了上述问题的出现。

- 24.03.13版本

从 24.03.13 版本开始支持了应用型负载均衡ALB，后续版本修复了 ALB 使用中的一些问题，推荐升级到 24.09.18 版本来使用 ALB。

- 24.12.24版本

从24.12.24版本开始支持了网络型负载均衡NLB，推荐升级到该版本或以上来使用NLB。

1. 版本查看及插件升级

1.1 控制台操作

针对维护版本之内UK8S 集群可以在控制台管理页面「插件-CloudProvider」页面，开启 CloudProvider 插件升级功能，开启 CloudProvider 插件升级功能会在集群中执行 CloudProvider 插件查询任务，大约需要 3 分钟，在此过程中请不要操作集群。升级功能开启后，即可看到 CloudProvider 插件版本信息，点击「升级 CloudProvider」即可进行升级。

升级过程约需要 1-3 分钟，升级过程中「当前版本」字段会显示为「升级中」，升级完成后显示最新版本号，如升级失败，可以再次尝试升级，或与我们技术支持联系。

> ⚠ 集群 CloudProvider 插件升级时，请勿进行服务发布等操作

1.2 手动升级

如果集群版本不在我们的维护版本之内，控制台将无法直接进行升级，参见：UK8S版本维护说明。

这时候，您可以手动升级cloudprovider，请执行下面的命令：

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cloudprovider/24.08.13.yml
```

1.3 老版本升级

如果 Kubernetes 版本在 1.14 以前，或者在控制台无法查看到 CloudProvider 版本信息，则需要通过命令行进行升级。

> 登录您集群的 Master 节点，如执行systemctl status ucloudcp是运行状态的话，请务必在**3 个 Master 节点**关闭二进制程序ucloudcp并更新。

1. 关闭老版本 CloudProvider

请分别登陆3台master节点，执行以下命令

```
systemctl stop ucloudcp
systemctl disable ucloudcp
```

2. 需要配置前置ConfigMap

请填写如下相关信息并保存文件为userdata.yaml。

```
apiVersion: v1
kind: ConfigMap
metadata:
name: uk8sconfig
namespace: kube-system
data:
UCLOUD_ACCESS_PRIKEY: xxxxxxxxxxxxxxx #API PRIKEY
UCLOUD_ACCESS_PUBKEY: xxxxxxxxxxxxxxx #API_PUBKEY
UCLOUD_API_ENDPOINT: http://api.service.ucloud.cn
UCLOUD_PROJECT_ID: org-xxxxxx #集群所在的项目ID
UCLOUD_REGION_ID: cn-bj2 #集群所在的地域，参考：https://docs.ucloud.cn/api/summary/regionlist
UCLOUD_SUBNET_ID: subnet-xxxxxx #集群所在的子网ID
UCLOUD_UK8S_CLUSTER_ID: uk8s-xxxxxx #UK8S集群名称
UCLOUD_VPC_ID: uvnet-xxxxxx #集群所在的VPC ID
```

3. 请执行创建ConfigMap

```
kubectl apply -f userdata.yaml
```

4. 请执行部署 CloudProvider

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cloudprovider/22.07.1.yml
```

5. 检查是否部署成功

如 Pod 处于非 running 状态，请您及时与我们技术支持联系，更新成功后，即可在控制台查看版本信息并进行后续升级。

```
kubectl get pod -n kube-system -l app=cloudprovider-ucloud -o wide
```

2. 变更记录

更新版本：24.12.24

更新时间：2024 年 12 月 24 日

更新内容：

- 支持NLB

更新版本：24.11.13

更新时间：2024 年 11 月 13 日

更新内容：

- 支持在创建ULB时指定要绑定的防火墙

更新版本：24.11.05

更新时间：2024 年 11 月 05 日

更新内容：

- 修复了使用**新购的内网ALB**创建svc，在svc更新时会额外绑定EIP的问题

更新版本：24.10.10

更新时间：2024 年 10 月 10 日

更新内容：

- 修复了使用**已有内网ALB**创建svc时会额外购买EIP的问题

更新版本：24.09.18

更新时间：2024 年 9 月 18 日

更新内容：

- 修复了创建ALB时，在service中指定subnet-id无法生效的问题

更新版本：24.08.13

更新时间:2024 年 8 月 13 日

更新内容:

- 修复了在集群节点超过20个之后ALB无法使用的问题

更新版本:24.06.28

更新时间:2024 年 6 月 28 日

更新内容:

- 修复了多个svc绑定同一个ALB时存在的问题

更新版本:24.03.13

更新时间:2024 年 3 月 14 日

更新内容:

- 支持ALB

更新版本:24.03.5

更新时间:2024 年 3 月 7 日

更新内容:

- 支持ULB4的tcp和udp协议混用

更新版本:23.04.2

更新时间:2023 年 7 月 6 日

更新内容:

- 减少telemetry超时时间,避免telemetry超时影响ULB的正常使用

更新版本:22.10.1

更新时间:2022 年 10 月 25 日

更新内容:

- 优化 ULB Backend 添加逻辑,加快 LoadBalancer 类型的 Service 获取 external IP 的速度
- 优化外网 ULB 绑定的 EIP 的默认带宽配置,如果没有在 Service 中明确指定带宽,将不会更新已有 EIP 的带宽值,对于由 Cloud Provider 新创建的 EIP,仍使用默认带宽。

更新版本:22.07.1

更新时间:2022 年 7 月 15 日

更新内容:

- 优化ULB的查询逻辑,提高修改、删除Service的速度。
- 增加通过EIP进行ULB匹配的兜底逻辑。从而防止因为用户手动修改了ULB名称或备注导致Service无法正确跟ULB匹配而发生ULB重建的问题。

更新版本:22.06.2

更新时间:2022 年 6 月 28 日

更新内容:

- 修复当 service 的 externalTrafficPolicy 属性为 Local,缺少 ULB ID 信息注入的缺陷

更新版本：22.06.1

更新时间:2022 年 6 月 10 日

更新内容:

- 更新依赖的 cloud provider 框架更新至v0.20.15，以引入kubernetes/cloud-provider#38 使得节点状态更新能够尽快的通知到cloud provider master、unready 以及具有 "node.kubernetes.io/exclude-from-external-load-balancers" label的节点不会加入到 lb 的 rs 中，而不再根据SchedulingDisabled 状态（k8s cloud provider 框架升级带来行为变化）

更新版本：21.10.2

更新时间:2021 年 11 月 4 日

更新内容:

- 优化了对 ULB 名称的校验，修复首次创建 VServer 失败导致 Service 无法创建的问题

更新版本：21.10.1

更新时间:2021 年 10 月 14 日

更新内容:

- 使用指定 ULB 创建 LoadBalancer 类型 Service 时，不能指定由 CloudProvider 创建的 ULB。

更新版本：21.05.2

更新时间:2021 年 5 月 28 日

更新内容:

- 加上 Endpoint Controller，感知 Endpoint 变化，在 externalTrafficPolicy=Local 的情况下，只会将运行了 Pod 的主机节点加入 ULB VServer Backends。
- **注意：**Endpoint 频繁变动会导致 ULB API 大量调用，需要在 cloudprovider 中添加如下参数，做限速处理。如需要调整参数，则必须通过命令行的方式，对插件进行调整。

```
containers:
- name: cloudprovider-ucloud
image: uhub.service.ucloud.cn/uk8s/cloudprovider-ucloud:21.05.2
args:
- "--endpoint-updates-batch-period=10" ## 所有 Endpoint 事件每隔 N 秒处理一次，默认为 10
```

更新版本：21.05.1

更新时间：2021 年 5 月 11 日

更新内容：

- 支持 UDP 类型 ULB 健康检查模式，需要在 Service 的 Annotations 中声明一下参数，ULB 健康检查机制请参考：健康检查，ULB 注释详解请参考：ULB 参数说明

```
annotations:
## 以下参数仅对 UDP 类型 ULB 生效
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-type: "customize" ## 设置健康检查类型为 customize，代表 UDP 健康检查
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-reqmsg: "" ## 代表 UDP 健康检查发出的请求报文
service.beta.kubernetes.io/ucloud-load-balancer-vserver-monitor-respmsg: "" ## 代表 UDP 健康检查请求应收到的响应报文
```

更新版本：21.04.1

更新时间：2021 年 4 月 22 日

更新内容：

- 可通过 Service Annotation，为节点添加注释，设定其在被设置为不可调度（即执行 Cordon 节点操作）后，不会自动被 ULB 剔除

```
annotations:
## 默认为 'true'，设置为 'false' 后节点不可调度时不会自动被 ULB 剔除
service.beta.kubernetes.io/ucloud-load-balancer-remove-unscheduled-backend: "false"
```

更新版本：21.02.1

更新时间：2021 年 2 月 6 日

更新内容：

- 支持内网 ULB7
- 支持 ULB7 的 TCP 模式

更新版本：20.10.2

更新时间：2020年10月29日

更新内容：

- 更新支持创建指定子网的ULB，详细设置请查看通过内网ULB访问Service。

更新版本：20.10.1

更新时间：2020年10月20日

更新内容：

- 动态创建成功LoadBalancer类型的Service后，插件将把关联的ULB Id注入到Service的Annotations中，以规避因API超时等导致的ULB重建。

更新版本：20.09.4

更新时间：2020年9月18日

变更内容：

- 升级UCloud Go SDK修复ULB防火墙设置，
- 增加新创建时，使用临时的ULB客户端进行API调用。

更新版本：20.09.3

更新时间：2020年9月17日

变更内容：

- 更新监控信息(非强制更新)。

更新版本：20.09.2

更新时间：2020年9月8日

变更内容：

- 增加cloudprovider重启后调用ULB相关接口的容错性，提高集群的运行的稳定性。

Volume 介绍

概念

我们知道，容器中的磁盘文件是临时的，一旦容器运行结束，其文件也会丢失。如果数据需要长期存储，那就需要对容器数据做持久化支持，在就涉及到Kubernetes中的一个核心概念Volume。Kubernetes 和 Docker 类似，也是通过 Volume 的方式提供对存储的支持。Kubernetes 中 的Volume 与Docker 中的 Volume 类似，主要的区别如下：

1. Kubernetes中的Volume被定义到Pod层面，Volume可以被 Pod 中的多个容器挂载到相同或不同的路径。
2. Kubernetes 中的 Volume 与 Pod 的生命周期相同，但与容器的生命周期不相关。当容器终止或重启时，Volume 中的数据不会丢失。
3. 当 Pod 被删除时，Volume 才会被清理。并且数据是否丢失取决于 Volume 的具体类型，比如：emptyDir 类型的 Volume 数据会丢失，而 PV 类型的数据则不会丢失。

Volume的本质是一个目录，其中可以包含数据，Pod中的容器可以访问该目录。该目录的形式，支持该目录的介质以及目录的内容取决于所使用的特定卷类型。Kubernetes 目前支持多种 Volume 类型，常用的类型如下：

- cephfs
- glusterfs
- nfs
- csi
- FlexVolume
- emptyDir

- hostPath
- local
- persistentVolumeClaim
- secret
- configMap

这里只提到了一些常用的Volume类型，更多详见官网文档Volume。

常见Volume类型

emptyDir

emptyDir与Pod的生命周期完全一致，它 在 Pod 分配到 Node 上时被创建，Kubernetes 会在 Node 上自动分配一个目录，因此无需指定 Node 宿主机上对应的目录文件。这个目录的初始内容为空，当 Pod 从 Node 上移除（Pod 被删除或者 Pod 发生迁移）时，emptyDir 中的数据会被永久删除。emptyDir Volume 主要用于某些应用程序无需永久保存的临时目录，或者在Pod中的多个容器之间共享数据等。emptryDir 默认使用主机磁盘进行存储的，也可以使用其它介质作为存储，比如：网络存储、内存等，设置 emptyDir.medium 字段的值为 Memory 就可以使用内存进行存储。

hostPath

hostPath 类型的 Volume 允许挂载 Node 节点上的文件或目录到 Pod 中，不过hostPath类型的Volume很少被使用，因为每个Node节点上的文件可能不同，最终导致一组 Pod（如 Deployment）在不同Node节点上的行为可能会有所不同，且Pod一旦被调度其他Node节点会导致数据错乱。另外要在Node节点上创建的文件或目录写入内容，需要在容器中以 root 身份运行进程，存在一定的安全隐患。

Secret

Secret对象用于存储和管理敏感信息，例如密码，OAuth令牌和ssh密钥。将此类信息放入一个secret中可以更好地控制它的用途，并降低意外暴露的风险。比如我们可以将Docker鉴权的信息放入到Secret，再挂载到的Pod中，用于拉取镜像时做权限认证，示例如下：1、创建Secret

```
# MYSECRET为secret 名称，请自行指定
kubectl create secret docker-registry MYSECRET \
--docker-server=uhub.service.ucloud.cn \
--docker-username=YOUR_UCLOUD_USERNAME@EMAIL.COM \
--docker-password=YOUR_UHUB_PASSWORD
```

2、Pod挂载Secret

```
apiVersion: v1
kind: Pod
metadata:
name: secret-test
spec:
containers:
- name: secret-test
image: centos
command:
- sleep
- "3600"
volumeMounts:
```

```
- name: config
mountPath: /root/.docker/
volumes:
- name: config
secret:
secretName: MYSECRET
items:
- key: .dockerconfigjson
path: config.json
mode: 0644
```

3、查看挂载效果

```
$ kubectl exec secret-test -- cat /root/.docker/config.json
{"auths":{"uhub.service.ucloud.cn":
{"username":"YOUR_UCLOUD_USERNAME@EMAIL.COM","password":"YOUR_UHUB_PASSWORD","auth":"dXNlcm5hbWU6cGFzc3dvcmQ="}}}
```

PV&PVC&StorageClass

Kubernetes 目前主要使用 PersistentVolume、PersistentVolumeClaim、StorageClass 三个 API 对象来进行持久化存储，下面分别介绍下这三个API对象。

PV

PV 的全称是PersistentVolume（持久化卷）。PersistentVolume 是 Volume 的一种类型，是对底层存储的一种抽象。PV 由集群管理员进行创建和配置，与Node一样，PV 也是属于集群

级别的资源。PV 包含存储类型、存储大小和访问模式。PV 的生命周期独立于 Pod，即使用它的Pod被 销毁时，PV 可以依然存在。

PersistentVolume 通过插件机制实现与共享存储的对接。Kubernetes 目前支持以下插件类型，其中FlexVolume和CSI是Kubernetes的标准插件，用于集成各云厂商的存储设备，UK8S便是基于CSI和flexVolume来集成UDisk、UFS、UFile等UCloud存储介质。

- FlexVolume
- CSI
- NFS
- RBD (Ceph Block Device)
- CephFS
- Glusterfs
- HostPath
- Local

PVC

PVC 的全称是PersistentVolumeClaim（持久化卷声明），PVC 是用户对存储资源的一种请求。PVC 和 Pod 比较类似，Pod 消耗的是节点资源，PVC 消耗的是 PV 资源。Pod 可以请求 CPU 和内存，而 PVC 可以请求特定的存储空间和访问模式。对于真正使用存储的用户不需要关心底层的存储实现细节，只需要直接使用 PVC 即可。

StorageClass

由于不同的应用程序对于存储性能的要求也不尽相同，比如：读写速度、并发性能、存储大小等。如果只能通过 PVC 对 PV 进行静态申请，显然这并不能满足任何应用对于存储的各种需求。为了解决这一问题，Kubernetes 引入了一个新的资源对象：StorageClass，通过 StorageClass 的定义，集群管理员可以先将存储资源定义为不同类型的资源，比如快速存储、慢速存储、共享存储（多读多写）、块存储（单读单写）等。

当用户通过 PVC 对存储资源进行申请时，StorageClass 会使用 Provisioner（不同存储资源对应不同的 Provisioner）来自动创建用户所需 PV。这样应用就可以随时申请到合适的存储资源，而不用担心集群管理员没有事先分配好需要的 PV。

UK8S支持存储挂载的地域

产品	地域
UDisk	请参考存储产品UDisk支持地域
UFS	请参考存储产品UFS支持地域
UFile	请参考存储产品UFile支持地域

在UK8S中使用UDISK

UK8S支持直接在集群中使用UDisk作为持久化存储卷。

备注:

1. 所有云主机均支持 SSD/SATA UDisk，**如果节点的云主机类型为快杰，则也支持 RSSD UDisk**；
2. SSD/SATA UDisk的最小值为 1GB，最大值为8000GB，RSSD UDisk 最大值为 32000GB；
3. UDisk和云主机必须位于同一可用区，如果您的集群是跨可用区模式，在应用部署的时候请注意。
4. **RSSD云盘有额外的限制条件，请见在UK8S中使用RSSD UDisk**。
5. UDisk以10g为计费单位，不满10g按10g计算，超过10g向上取整到10的整数倍;

1. 存储类 StorageClass

在创建持久化存储卷（PersistentVolume）之前，你需要先创建 StorageClass，然后在 PVC 中使用 StorageClassName。

UK8S 集群默认创建了三个 StorageClass，你也可以创建一个新的StorageClass，示例及说明如下：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: udisk-ssd-test
```

```
provisioner: udisk.csi.ucloud.cn #存储供应方，此处不可更改。
parameters:
type: "ssd" # 存储介质，支持ssd,rssd和sata，必填
fsType: "ext4" # 文件系统，必填
udataArkMode: "no" # 是否开启方舟模式，默认不开启，非必填
chargeType: "month" # 付费类型，支持dynamic、month、year,不填默认为按小时。
quantity: "1" # 购买时长，dynamic无需填写，可购买1-9个月，或1-10年
reclaimPolicy: Delete # PV回收策略，支持Delete和Retain，默认为Delete，非必填
volumeBindingMode: WaitForFirstConsumer # 强烈建议配置该参数
allowVolumeExpansion: true # 声明该存储类支持可扩容特性
mountOptions:
- debug
- rw
```

备注：1.15之前的Kubernetes版本，mountOptions无法正常使用，请勿填写，详见Issue80191

2. 创建持久化存储卷声明 PVC

2.1 新建 UDisk

使用新建 UDisk，则可直接创建 PVC 对象，CSI 会自动创建 UDisk 并关联。如果storageClass中的 volumeBindingMode 设置为 WaitForFirstConsumer，则需要等待Pod使用PVC后才会创建真正的UDisk。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: test-pvc-claim
spec:
accessModes:
- ReadWriteOnce
## storageClassName必须与上文创建的 StorageClass 的name一致
storageClassName: udisk-ssd-test
resources:
requests:
storage: 20Gi
```

2.2 使用已有 UDisk

> 如需使用已有 UDisk，需先创建 PV 对象并与已有 UDisk 绑定，再创建 PVC 对象、使用与 PV 相同的声明进行关联

创建持久化存储卷 PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
name: test-pvc-claim
spec:
accessModes:
- ReadWriteOnce
capacity:
storage: 20Gi
csi:
driver: udisk.csi.ucloud.cn
volumeAttributes:
type: ssd # 磁盘类型，枚举值为ssd,sata,rssd
volumeHandle: bs-qg55w254 # 请修改为自己的UDiskId
# nodeAffinity：强烈建议添加此字段
persistentVolumeReclaimPolicy: Retain
# storageClassName必须与上文创建的 StorageClass 的name一致
storageClassName: udisk-ssd-test
```

注意：根据使用UDisk的Pod调度策略，为了保证后续调度可以顺利执行，强烈建议您创建时为PV添加nodeAffinity字段。由于不同版本以及不同Storage Class本部分的内容不尽相同，可以参照相同Storage Class CSI自动自动创建出来PV的对应字段。

2.3 创建 PVC 并与 PV 关联

spec.storageClassName、spec.resources.requests.storage、volumeName需要与pv相对应。

```
kind: PersistentVolumeClaim
apiVersion: v1
```

```
metadata:
name: test-pvc-claim
spec:
accessModes:
- ReadWriteOnce
storageClassName: udisk-ssd-test
resources:
requests:
storage: 20Gi
volumeName: test-pvc-claim
```

3. 在 Pod 中使用 PVC

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
volumeMounts:
- name: test
```

```
mountPath: /data
ports:
- containerPort: 80
volumes:
- name: test
persistentVolumeClaim:
claimName: test-pvc-claim
```

4. 使用StatefulSet

如果有部署多副本有状态应用的需求，建议使用StatefulSet而不是Deployment，因为Deployment中多个pod会共享同一个磁盘，如果pod在不同节点就会导致问题。而StatefulSet能让多个pod使用自己的磁盘。

一般会使用StatefulSet来为每个pod动态创建pvc和udisk磁盘，下面的yaml可以作为参考：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: web
spec:
selector:
matchLabels:
app: nginx
serviceName: "nginx"
```

```
replicas: 4
template:
metadata:
labels:
app: nginx
spec:
terminationGracePeriodSeconds: 10
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
ports:
- containerPort: 80
name: web
volumeMounts:
- name: www
# 云盘挂载的目录
mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
name: www
spec:
accessModes: [ "ReadWriteOnce" ]
# 要使用的StorageClass,表明要创建什么类型的磁盘
storageClassName: "udisk-ssd-test"
resources:
```

```
requests:
storage: 20Gi
```

在UK8S中使用RSSD UDisk

RSSD UDisk最高IOPS可达120万、延时低于0.1ms，数据持久性为99.999999%，最大容量32000G，适用于数据库、Elastic Search等需要低延时的IO密集型业务。UK8S支持将RSSD UDisk作为容器的持久化存储卷，但前提是集群中必须有快杰机型的节点(当前仅快杰云主机支持挂载RSSD UDisk)。

下面演示下如何在UK8S集群中使用RSSD UDisk。

限制条件

1. 集群中必须有快杰机型的节点，否则创建出来的PV将无法使用；
2. Kubernetes版本不低于1.18；
3. **csi-udisk插件版本必须大于等于22.09.1，如果小于，请到控制台升级，详情见RSSD云盘挂载问题；**
4. StorageClass中的volumeBindingMode必须显式设置为WaitForFirstConsumer（否则创建出来的RSSD UDisk可能无法挂载）；

使用示例

1. 创建PVC,这部分与创建SATA、SSD UDisk的PVC完全一致。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
```

```
name: logdisk-claim
spec:
accessModes:
- ReadWriteOnce
storageClassName: csi-udisk-rssd
resources:
requests:
storage: 10Gi
```

2. 创建Pod

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: http
labels:
app: http
spec:
strategy:
type: Recreate
replicas: 1
selector:
matchLabels:
app: http
template:
```

```
metadata:
labels:
app: http
spec:
containers:
- name: http
image: uhub.service.ucloud.cn/wxyz/httpudisk:1.0
imagePullPolicy: Always
ports:
- containerPort: 8080
volumeMounts:
- name: log
mountPath: /data
volumes:
- name: log
persistentVolumeClaim:
claimName: logdisk-claim
```

UDisk 动态扩容

本文档主要描述如何在 UK8S 中扩容 UDisk 类型的 PVC，包括在线扩容和离线扩容两种场景。

1. 限制条件

1. UK8S Node 节点实例的创建时间必须晚于 2020 年 5 月，不满足此条件的节点，则必须先对 Node 节点进行先关机，再开机操作。
2. Kubernetes 版本不低于 1.14，如集群版本是 1.14 及 1.15，必须在三台 Master 节点 /etc/kubernetes/apiserver 文件中配置 --feature-gates=ExpandCSIVolumes=true，并通过 systemctl restart kube-apiserver 重启 APIServer。并需要在 node 节点中修改/etc/kubernetes/kubelet 文件中配置，增加--feature-gates=ExpandCSIVolumes=true，执行systemctl restart kubelet 重启 kubelet。对于 1.14 版本的集群，如果需要在线扩容 (pod 不重启)，需要同时配置 ExpandInUsePersistentVolumes=true的特性开关。1.13 及以下版本不支持该特性，1.16 及以上版本无需配置；
3. CSI-UDisk 版本不低于 20.08.1，CSI 版本更新及升级请查看：CSI 更新记录及升级指南;
4. 扩容时声明的期望容量大小必须是 10 的整数倍，单位为 Gi;
5. 只支持动态创建的 PVC 扩容，且 storageClass 必须显示声明可扩容(见后文)；

2. 扩容 UDisk 演示

2.1 创建 UDisk 存储类，显式声明可扩容

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: csi-udisk-ssd
provisioner: udisk.csi.ucloud.cn # provisioner 必须为 udisk.csi.ucloud.cn
parameters:
type: "ssd"
fsType: "ext4"
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true # 必须声明该存储类支持可扩容特性
```

2.2 通过该存储类创建 PVC，并挂载到 Pod

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: udisk-volume-expand
spec:
accessModes:
- ReadWriteOnce
storageClassName: csi-udisk-ssd
resources:
requests:
```

```
storage: 10Gi
---
apiVersion: v1
kind: Pod
metadata:
name: udisk-expand-test
labels:
app: udisk
spec:
containers:
- name: http
image: uhub.service.ucloud.cn/ucloud/nginx:1.17.10-alpine
imagePullPolicy: Always
ports:
- containerPort: 8080
volumeMounts:
- name: udisk
mountPath: /data
volumes:
- name: udisk
persistentVolumeClaim:
claimName: udisk-volume-expand
```

Pod 启动后，我们分别查看下 PV、PVC 以及容器内的文件系统大小，可以发现，目前都是 10Gi

```
# kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
pvc-25b83584-35de-43e4-ad23-c1fc638a09e2 10Gi RWO Delete Bound default/udisk-volume-expand ssd-csi-udisk 2m26s

# kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
udisk-volume-expand Bound pvc-25b83584-35de-43e4-ad23-c1fc638a09e2 10Gi RWO ssd-csi-udisk 2m30s

# kubectl exec -it udisk-expand-test -- df -h
Filesystem Size Used Avail Use% Mounted on
...
/dev/vdc 9.8G 37M 9.7G 1% /data
...
```

2.3 在线扩容 PVC

执行 kubectl edit pvc udisk-volume-expand，将 spec.resource.requests.storage 改成 20Gi，保存后退出，大概在一分钟左右，PV、PVC 以及容器内的文件系统大小容量属性都变成了 20Gi。

```
# kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
pvc-25b83584-35de-43e4-ad23-c1fc638a09e2 20Gi RWO Delete Bound default/udisk-volume-expand ssd-csi-udisk 2m26s
# kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
```

```
udisk-volume-expand Bound pvc-25b83584-35de-43e4-ad23-c1fc638a09e2 20Gi RWO ssd-csi-udisk 2m30s
# kubectl exec -it udisk-expand-test -- df -h
Filesystem Size Used Avail Use% Mounted on
...
/dev/vdc 20G 37M 19.7G 1% /data
...
```

同时登录 UDisk 控制台，发现 UDisk 展示容量也增大到了 20Gi。这样我们完成了 Pod 不重启，服务不停机的数据卷在线扩容。

2.4 离线扩容 PVC（推荐）

在上面的示例中，我们完成了数据卷的在线扩容。但在高 IO 的场景下，Pod 不重启进行数据卷扩容，有小概率导致文件系统异常。最稳定的扩容方案是先停止应用层服务、解除挂载目录，再进行数据卷扩容。下面我们演示下如何进行停服操作。

上文步骤 2.3 完成的时候，我们有一个 Pod 且挂载了一个 20Gi 的数据卷，现在我们需要对数据卷进行停服扩容。

1. 基于上文的示例 yaml，去掉 PVC 相关的内容，单独创建一个名为 udisk-expand-test 的 yaml，只保留 Pod 的相关信息。然后删除 Pod，但保留 PVC 和 PV。

```
# kubectl delete po udisk-expand-test
pod "udisk-expand-test" deleted
```

此时，PV 和 PVC 依然相互 Bound，对应的 UDisk 已经从云主机中卸载，处于可用状态。

2. 修改 PVC 信息，将 spec.resource.requests.storage 改成 30Gi, 保存并退出。

等待一分钟左右后，执行 `kubectl get pv`，当 PV 的容量增长到 30Gi 后，重建 Pod。需要注意的是，此时执行 `kubectl get pvc` 的时候，返回的 PVC 容量依然是 20Gi，这是因为文件系统尚未扩容完毕，PVC 处于 FileSystemResizePending 状态。

```
# kubectl edit pvc udisk-volume-expand
persistentvolumeclaim/udisk-volume-expand edited
# kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
pvc-25b83584-35de-43e4-ad23-c1fc638a09e2 30Gi RWO Delete Bound default/udisk-volume-expand ssd-csi-udisk 20m
# kubectl create -f udisk-expand-test.yml
```

当 Pod 重新创建成功后，可以发现，PV、PVC 的容量大小都是 30Gi，同时在容器中执行 df 看到的对应文件系统容量也是 30Gi。

从 Flexvolume UDisk 存储卷升级到 CSI UDisk 存储卷

Kubernetes v1.13 以及更早版本的用户，Pod 通过 Flexvolume 存储卷的方式挂载 UDisk 块存储卷。由于不支持拓扑感知动态调度等基础特性，Flexvolume 方案早已停止演进，而 CSI 已经成为容器存储实现标准。

使用 Flexvolume 创建 UDisk 挂载卷的 UK8S 早期用户目前面临着集群升级时将 Flexvolume PV 转换成 CSI PV 的问题，本文档提供一个示例，用于说明如何完成这一转换。

> ⚠ 升级时会造成服务中断，请合理规划迁移时间，并做好相关备份。

1. Flexvolume UDisk 存储卷说明

如下是一个已经挂载了 Flexvolume UDisk 存储卷 Workload 的 yaml 文件 nginx-fv.yaml，它包含了一个 StorageClass，一个 Pod 声明和其引用的 PVC。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: udisk-ssd-flexvolume
parameters:
type: ssd # 磁盘类型，枚举值为ssd,sata,rssd
provisioner: ucloud/udisk
reclaimPolicy: Delete
```

```
volumeBindingMode: Immediate
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: nginx-fv
spec:
accessModes:
- ReadWriteOnce
storageClassName: udisk-ssd-flexvolume
resources:
requests:
storage: 10Gi
---
apiVersion: v1
kind: Pod
metadata:
name: nginx
labels:
app: nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
imagePullPolicy: Always
```

```
ports:
- containerPort: 8080
volumeMounts:
- name: nginxdisk
mountPath: /data
volumes:
- name: nginxdisk
persistentVolumeClaim:
claimName: nginx-fv
```

执行 `kubectl apply -f nginx-fv.yaml`，发现相应的 StorageClass、Pod 及相关联的 PVC 已经创建成功，此外 cloudprovider 还会为这个 PVC 创建并绑定一个 PV 对象（名称与 PVC 对象的 **Volume** 值相同），如下所示：

```
Name: pvc-8b7946f7-1214-11ec-8f6b-5254003e805f-bsm-upc4bc0v
Labels: <none>
Annotations: pv.kubernetes.io/provisioned-by: ucloud/udisk
Finalizers: [kubernetes.io/pv-protection]
StorageClass: udisk-ssd-flexvolume
Status: Bound
Claim: default/nginx-fv
Reclaim Policy: Delete
Access Modes: RWO
VolumeMode: Filesystem
Capacity: 10Gi
Node Affinity: <none>
```

```
Message:
Source:
Type: FlexVolume (a generic volume resource that is provisioned/attached using an exec based plugin)
Driver: ucloud/flexv
FSType:
SecretRef: nil
ReadOnly: false
Options: map[diskId:bsm-upc4bc0v]
Events: <none>
```

在 Source.Options 可以发现，这个 PV 对应的实际 UDisk 实例为 bsm-upc4bc0v。

2. 升级步骤

⚠ 升级时会造成服务中断，请合理规划迁移时间，准备好所有需要的 yaml 文件，并做好相关备份。

2.1 确认原有 PV 回收策略为 Retain

如果 PV 的回收策略不是 **Retain**，则需要通过以下命令将其回收策略改成 **Retain**。这时即使您删除 Pod 和对应的 PVC，可以发现，PV 依然存在，对应的 UDisk 实例也依然保留。

⚠ 如删除 Flexvolume 创建的 PV，则对应的 UDisk 会被删除，如需要相应的 UDisk，请确保该 PV 不会被删除。

```
kubectl patch pv <your-pv-name1> <your-pv-name2> <your-pv-name3> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

2.2 确认集群中已安装 CSI 插件

首先，请参考CSI 升级指南中命令行升级的方法，为集群安装最新的 CSI 插件。

2.3 删除原有 Pod 及 PVC

通过 kubectl delete -f nginx-fv.yaml，删除原有 PVC 及 Pod，这时可以发现由于 PV 回收策略为 **Retain**，PV 及相应的 UDisk 仍被保留，PV 状态为 **Release**。

2.4 通过 CSI 以指定 UDisk 创建新的 PV 及 PVC

接下来，我们将以原先的 UDisk 数据盘，创建 PV 并关联 PVC，详见在 UK8S 中使用 UDisk文档中「使用已有 UDisk 部分」。

以下为示例文件 nginx-csi-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: nginx-csi
spec:
accessModes:
- ReadWriteOnce
capacity:
storage: 10Gi
csi:
```

```
driver: udisk.csi.ucloud.cn
volumeAttributes:
type: ssd
volumeHandle: bsm-upc4bc0v # 请修改为自己的UDiskId
persistentVolumeReclaimPolicy: Retain
storageClassName: udisk-ssd
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: nginx-csi
spec:
accessModes:
- ReadWriteOnce
storageClassName: udisk-ssd
resources:
requests:
storage: 10Gi
volumeName: nginx-csi
```

执行 `kubectl apply -f nginx-csi-pv.yaml` 后，我们可以看到新的 PV 和 PVC 已经创建成功。

2.5 将 PVC 挂载到相应的 Pod

详见在 UK8S 中使用 UDisk文档，以下为示例 yaml 文件 nginx-csi.yaml

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
labels:
app: nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
imagePullPolicy: Always
ports:
- containerPort: 8080
volumeMounts:
- name: nginxdisk
mountPath: /data
volumes:
- name: nginxdisk
persistentVolumeClaim:
claimName: nginx-csi # 注意名称与新的 PVC 相对应
```

执行 `kubectl apply -f nginx-csi.yaml` 后，我们可以看到新的 Pod 已经创建成功，并与相应的 PVC 绑定。

升级成功后，原有的 FlexVolume 安装文件可保留，只要在申明新的 StorageClass、PV 和 PVC 时，按照 CSI 相应的规范进行声明即可。

RSSD云盘挂载问题

一般的云盘在挂载的时候，需要确保其跟主机处于同一个可用区下面。但是对于RSSD云盘来说，除了可用区，还需要确保它们处于同一个RDMA集群中。RDMA是一个比可用区更小的概念，它在控制台界面是隐藏的，只能通过API指定和查询。

总体来说，一个RSSD云盘对于主机有以下要求：

- 主机必须是O型，即快杰型主机。
- 主机跟云盘处于同一可用区。
- 主机跟云盘处于同一RDMA。

如果满足前两点但是不满足第三点，挂载会失败。失败的报错一般是（关注错误码为17218）：

```
[17218] Command failed: add_udisk.py failed
```

RDMA还有一个问题是，它是随时可能发生变更的，即主机可能对云盘或主机的RDMA进行迁移，并且变更后无法通知下游服务。

CSI在处理使用了RSSD云盘的Pod时，需要解决下面两个问题：

- 新建RSSD云盘时，确保其RDMA跟Pod所处的节点一致。
- 重新调度使用了RSSD云盘的Pod时，确保调度的节点是O型，并且RDMA跟云盘一致。

一般来说，用户不需要关心上述问题，但是因为CSI历史设计的原因，22.09.1以前的CSI在处理RSSD云盘时，如果遇到RDMA迁移，将会发生很严重的挂载失败问题，下面我将对CSI调度RSSD云盘机制进行详细的讲解，以方便您更好地理解这个问题，并知晓为什么在使用RSSD云盘的情况下要把CSI升级到22.09.1及以上。

静态调度

这是22.09.1以下版本CSI采用的方案。

在22.09.1以及以前的csi中，是通过在pv上增加nodeAffinity来实现的。有关节点亲和性，请见官方文档：Assign Pods to Nodes using Node Affinity。

对于快杰型云主机，其node上面会保存一个拓扑label保存RDMA字段，例如：

```
topology.udisk.csi.ucloud.cn/rdma-cluster-id: 9002_25GE_D_R006
```

这表示这个node处于006这个RDMA集群中。

对于RSSD云盘PV，会在其nodeAffinity上保存RDMA：

```
nodeAffinity:
required:
nodeSelectorTerms:
- matchExpressions:
- key: topology.udisk.csi.ucloud.cn/rdma-cluster-id
operator: In
values:
- 9002_25GE_D_R006
```

上面两个字段都是CSI写入的，并且一旦写入，就不再可以变更。在调度的时候，通过节点亲和性机制，就确保了使用RSSD云盘的Pod只会被调度到RDMA匹配的节点上面：

如果RDMA一直不发生变更，这没有问题，但是一旦发生RDMA迁移，UK8S无法探测到这种迁移，也就无法更新这上面的信息，就发生了数据不一致。

即使UK8S能够探测到这种迁移，因为nodeAffinity是immutable，我们也无法通过更新字段的方式来更新信息。

而如果出现数据不一致，就会发生严重的问题，假设云盘的实际RDMA为005，但是其在UK8S中保存的RDMA为006，根据节点亲和性，使用了它的Pod会被调度到RDMA为006的节点上面，跟实际的RDMA不匹配，最终会导致云盘挂载失败。

如果您的CSI版本低于22.09.1，出现了RSSD云盘挂载不上的情况，并且在CSI日志中发现17218错误码，那么十有八九就是因为RDMA迁移导致的问题。

可见，我们不应该在UK8S集群中以任何方式储存RDMA信息，这样的信息是完全不可靠的。我们应该动态地获取RDMA进行调度。

动态调度

> 这是22.09.1以及以上版本CSI采用的方案(此版本要求Kubernetes版本不低于1.18)。

在22.09.1及以后版本的CSI中，将不再采用节点亲和性来调度RSSD云盘，所有RDMA信息都将会动态获取并调度。在不考虑存量数据的情况下，通过动态调度即可解决RDMA迁移的问题。

要想实现动态调度，至少需要在下面两个地方插入动态逻辑：

- 创建RSSD云盘时，动态获取节点的RDMA。
- 调度RSSD云盘时，动态获取云盘和节点的RDMA进行匹配。

下面两节将分别介绍CSI如果解决上面两个问题。

创建RSSD云盘

创建RSSD云盘是在CSI里面实现的，因此我们只需要更改CSI创建云盘的逻辑即可：

- 将topology.udisk.csi.ucloud.cn/rdma-cluster-id这个拓扑标签移除。
- 新建PV时，不再写入topology.udisk.csi.ucloud.cn/rdma-cluster-id这个nodeAffinity。
- 创建RSSD云盘时，调用API获取节点的RDMA信息，并传递给RSSD云盘创建接口，见：CreateUDisk API文档。

这样，新创建的RSSD云盘将不再依赖节点亲和性进行调度了。

调度RSSD云盘

调度器在调度包含RSSD的Pod时，需要能动态获取RSSD的RDMA 信息，这里是需要调用UCloud API的。原生的kube-scheduler肯定是无法实现，需要通过kube-scheduler提供的扩展机制来完成。

这里说明一下，KubeScheduler提供了两种扩展调度的机制，即Extender机制和Framework机制，下面简要地说明一下它们之间的差异：

- scheduler extender：需要将调度插件部署在master节点上面，调度器在调度的时候会在不同的扩展点通过HTTP的方式调用扩展。
- scheduler framework：完全编译一个独立的调度器，可以在里面插入自己的调度逻辑。单独作为Deployment部署在集群中，要使用这个调度器，需要修改schedulerName配置。

一般来说官方更加推荐使用第二种方式来扩展调度器，但是我们不希望修改schedulerName，并且第二种方式意味着需要为不同版本的Kubernetes维护不同的调度器版本，后期维护起来也更加麻烦，因此我们使用了第一种扩展方式。

这要求在集群的master节点上单独部署一个HTTP服务，它用于实现自己的调度逻辑，然后需要在调度器的配置中增加extenders相关内容：

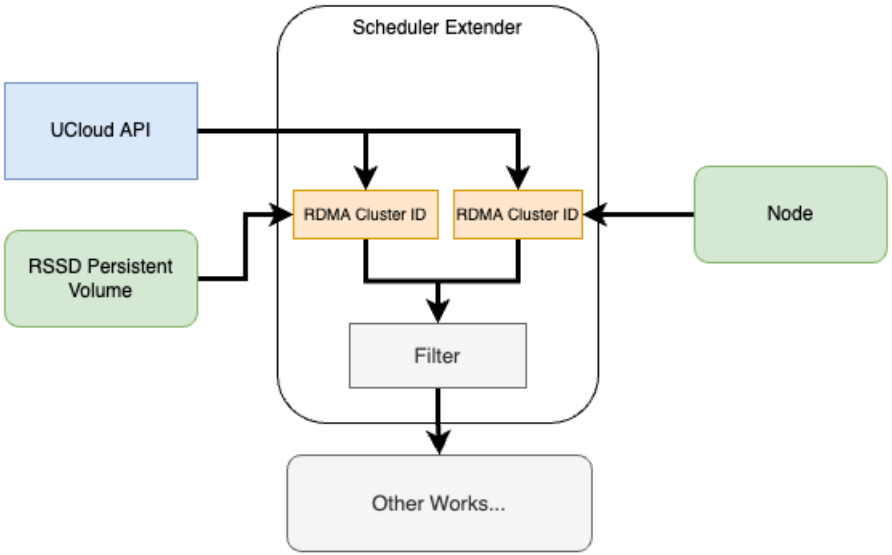
```
extenders:
- urlPrefix: http://127.0.0.1:6678/
```

```
filterVerb: filter
httpTimeout: 60s
```

这表示对调度的filter点进行扩展，扩展时调用http://127.0.0.1:6678/filter这个接口。

> 特别注意：extenders这个特性是在scheduler的v1beta2版本之后引入的，所以如果Kubernetes版本低于1.19，我们是无法进行扩展的。如果您使用了低版本的Kubernetes，应该先升级Kubernetes版本。

这样，就可以在扩展点内调用UCloud API，来动态获取RSSD以及node的RDMA Cluster信息，并根据这个信息对node进行过滤了：



scheulder-extender会检查Pod是否使用了RSSD的PV，如果使用了，会调用UCloud API获取PV以及节点的RDMA信息，并根据这个信息过滤掉不匹配的节点。

在部署的时候，需要在您集群的master节点上面新加一个systemd，叫做scheduler-extender-uk8s，可以通过下面的命令检查服务的健康：

```
systemctl status scheduler-extender-uk8s
```

如果调度出现了问题，可以通过下面的命令查看日志：

```
journalctl -u scheduler-extender-uk8s.service -f
```

在安装1.19以及以上的集群时，uk8s会自动将scheduler-extender-uk8s安装到所有master节点上面，关于如何处理现有集群，请参考后面的内容。

通过控制台升级CSI并安装scheduler-extender

这里一定要特别注意，新版本的csi必须配合scheduler-extender使用，所以，在升级22.09.1的CSI时，请到控制台操作，不要通过自行修改image完成。

新的csi版本中将会集成scheduler-extender的版本，格式为{csi-version}-se{scheduler-extender-version}，例如22.09.1-se22.08.3表示csi的版本为22.09.1，scheduler-extender版本为22.08.3。如果集群中没有安装scheduler-extender，格式为{csi-version}-se-unknown，例如21.09.1-se-unknown。

通过这种集成版本号的方式，可以很方便地将csi跟scheduler-extender绑定在一起，并且不需要单独增加scheduler-extender插件的管理页面。

查询版本

查询csi版本可以通过直接查询StatefulSet中的image完成，而查询scheduler-extender版本比较复杂，需要登录集群的master节点并通过命令调用的方式完成，因为scheduler-extender是通过systemd部署的。

这里就需要异步任务介入了，我们需要在master节点上面部署Job来完成scheduler-extender版本的查询。原来的csi版本查询是同步完成的，这里需要修改为异步调用，跟cni的版本查询类似。

升级不一致问题

以后在升级csi的时候，必须首先升级或安装scheduler-extender，因为前文介绍过，新的csi必须依赖scheduler-extender才可以工作。如果scheduler-extender升级或安装失败，必须停止整个csi升级过程。

这里可能产生的不一致是scheduler-extender安装成功了，但是csi没有升级成功。这会导致集群中RSSD PV通过scheduler-extender和通过nodeAffinity的约束同时存在。而这两个约束做的事情无非都是确保RSSD PV被调度到RDMA Cluster一致的node上面，只不过一个是动态的，一个是静态的。两个约束同时存在实际上是不冲突的。

综上：

- 可以容忍scheduler-extender安装成功，csi升级失败，因为这相当于同时存在了两个约束，后续让客户重试升级csi即可。
- 不可以容忍scheduler-extender未安装的情况下升级csi，因为这相当于集群不存在对RSSD PV的约束了。

所以我们没有做升级失败的回滚，只要确保先升级scheduler-extender再升级csi即可。

除此之外，在升级csi时，我们还加了下面的约束：

- 如果集群的版本低于1.19.x，不允许升级csi，需要客户先把集群版本升级上去。
- 如果客户集群中存在包含RDMA nodeAffinity的PV，不允许升级csi，因为这需要人为介入来hack数据（详见后面处理历史存量数据的章节）。

处理历史存量数据

上面就解决了所有新建RSSD云盘的问题，但是缺少对于存量数据的处理。

一般想到的是，在升级完成之后把PV上面的nodeAffinity数据移除就好了，但是Kubernetes有个很蛋疼的设计，就是nodeAffinity是immutable的，无法直接进行修改。而只要nodeAffinity存在，kube-scheduler就会根据节点亲和性来约束PV，当磁盘迁移时，就会发生问题。

我们需要通过特殊的手段来把PV上面的nodeAffinity移除掉。这里的手段是非常hack的，需要直接修改etcd的数据，因此这会是一个非常非标的操作，不能集成自动化工具中，需要人工介入进行处理。

如果您的集群中有存量RSSD云盘数据，请联系我们的技术支持，我们会手动为您修复数据。

在节点宕机时恢复挂载了云盘的Pod

警告：节点宕机后，在卸载云盘时无法到节点上面进行unmount操作，因此无法安全地卸载云盘。下面的文档涉及对云盘的强制解绑，这种操作可能会造成数据丢失。如果无法接受这种风险，切勿进行本文档的操作（此时应该尝试恢复节点）。

如果磁盘数据容忍丢失（例如储存日志数据），并且希望Pod尽快恢复，可以参考本文档的操作。

节点宕机之后，如果Pod挂载了云盘，是无法被自动恢复的。因为Kubernetes无法确定节点是真正宕机了还是因为网络问题暂时失联，所以云盘此时还有可能存在数据写入。如果此时贸然进行Pod迁移可能会破坏云盘。

如果是正常的Pod，在经过容忍时间之后，会自动进行驱逐，详见我们的另外一篇文档：Pod 容忍节点异常时间调整。

如果您确定节点宕机了，并且想要恢复宕机节点上面挂载了云盘的Pod，请参考本文档进行操作。

场景复现

下面我们创建一个3节点的集群：

```
$ kubectl get node
NAME STATUS ROLES AGE VERSION
10.9.133.163 Ready,SchedulingDisabled master 4m26s v1.22.5
10.9.138.243 Ready <none> 4m16s v1.22.5
```

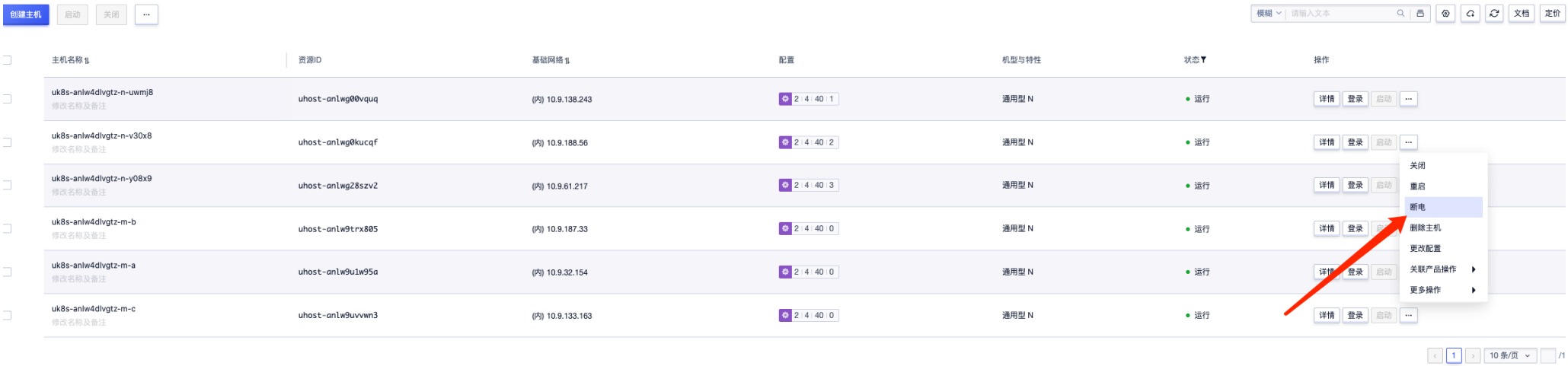
```
10.9.187.33 Ready,SchedulingDisabled master 4m41s v1.22.5
10.9.188.56 Ready <none> 3m58s v1.22.5
10.9.32.154 Ready,SchedulingDisabled master 4m43s v1.22.5
10.9.61.217 Ready <none> 3m49s v1.22.5
```

在里面创建一个StatefulSet，启动6个pod，它们都挂载了ssd云盘：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
name: test-web-ssd
spec:
selector:
matchLabels:
app: nginx
replicas: 6
serviceName: "test-nginx-ssd"
template:
metadata:
labels:
app: nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
```

```
ports:
- containerPort: 80
name: web
volumeMounts:
- name: test-nginx-ssd
mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
name: test-nginx-ssd
spec:
accessModes: [ "ReadWriteOnce" ]
storageClassName: "ssd-csi-udisk"
resources:
requests:
storage: 1Gi
```

在控制台界面，手动将一个云主机进行断电操作，以模拟宕机的场景：



过了一会，会发现该节点处于NotReady状态，Pod卡在了Terminating，而StatefulSet控制器不会为它们启动新的副本或进行云盘卸载等操作：

```
$ kubectl get node 10.9.188.56
NAME STATUS ROLES AGE VERSION
10.9.188.56 NotReady <none> 12m v1.22.5
```

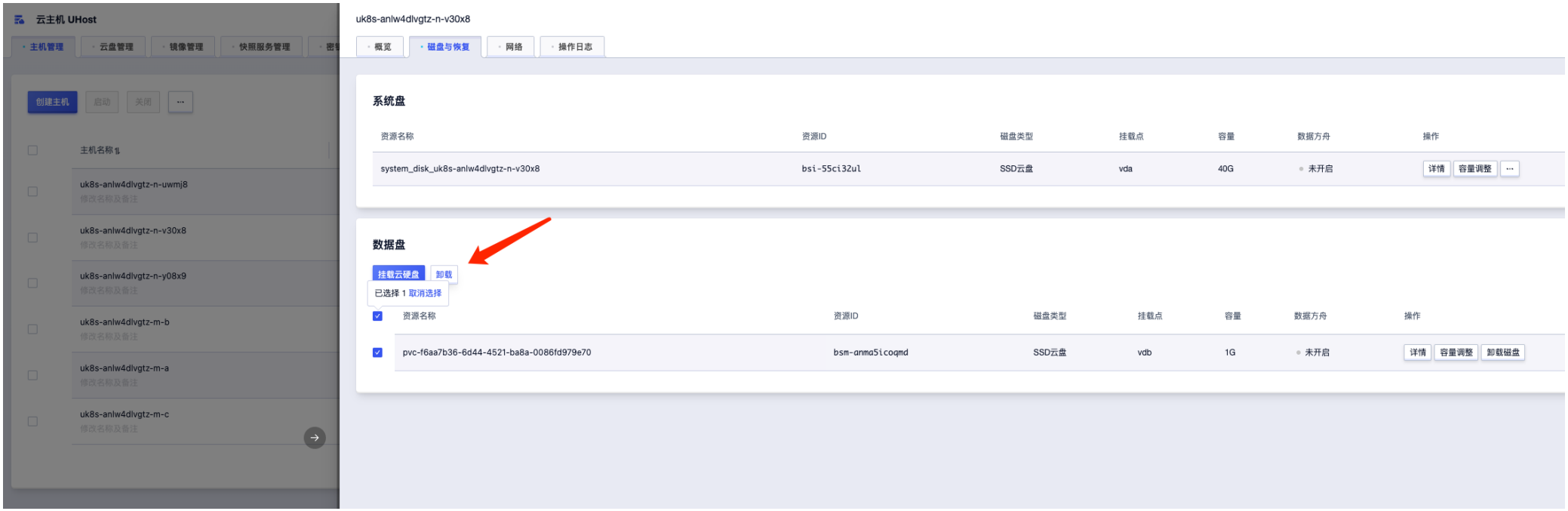
```
$ kubectl get po -o wide | grep '10.9.188.56'
test-web-ssd-3 1/1 Terminating 0 9m52s 10.9.48.46 10.9.188.56 <none> <none>
```

修复Pod

如果该节点短时间内无法恢复，我们应该考虑恢复中断的Pod以确保业务的正常运行。

首先，在确认节点宕机后，到控制台界面对异常节点上面的云盘进行手动卸载的操作：

> 警告：这一步可能造成云盘部分数据的丢失，请确认可以接受风险再进行操作，如果无法容忍数据的丢失，切勿进行操作。



随后，在Kubernetes中，将异常的node资源删除掉，这样才能释放异常的Pod并进行重建：

```
$ kubectl delete node 10.9.188.56
node "10.9.188.56" deleted
```

而此时调度器并不知道云盘已经卸载，，我们还需要手动删除对应的volumeattachment资源：

```
$ kubectl get volumeattachment | grep '10.9.188.56'
csi-53bbdc72c17cc6ecc7c87b2b9b6526679175e53f0b88b52384f9773c0abbcded udisk.csi.ucloud.cn pvc-f6aa7b36-6d44-4521-ba8a-0086fd979e70
10.9.188.56 true 13m
```

```
$ kubectl delete volumeattachment csi-53bbdc72c17cc6ecc7c87b2b9b6526679175e53f0b88b52384f9773c0abbcded
volumeattachment.storage.k8s.io "csi-53bbdc72c17cc6ecc7c87b2b9b6526679175e53f0b88b52384f9773c0abbcded" deleted
```

等待Pod重建，重建后会复用原来的云盘：

```
$ kubectl get po test-web-ssd-3
NAME READY STATUS RESTARTS AGE
test-web-ssd-3 1/1 Running 0 103s
```

在UK8S中使用UFS

本文档介绍如何在UK8S集群中，使用UFS作为K8S底层的存储支持，UFS为共享存储，可以同时为多个Pod提供服务。

前置条件

- 在UFS产品页面购买UFS实例并设置好挂载点，操作完毕后，您会得到UFS挂载地址和目录，类似10.19.255.192:/
- 集群节点安装nfs-utils，使用''yum install -y nfs-utils''命令，2019年5月1日以后的UK8S节点已默认安装nfs-utils。
- UFS与UK8S集群必须处于同一VPC，否则文件系统无法成功挂载。

创建PV

需要在集群内手动创建持久化存储卷，yaml示例如下两种：

UFS 容量型

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: ufspv4
spec:
```

```
capacity:
storage: 10Gi
accessModes:
- ReadWriteMany
persistentVolumeReclaimPolicy: Retain
nfs:
path: /
server: 10.19.255.12 # 请修改为你UFS的挂载地址
mountOptions:
- nolock
- nfsvers=4.0 # 必须与创建的UFS协议一致
```

UFS SSD性能型

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: ufspv4
spec:
capacity:
storage: 10Gi
accessModes:
- ReadWriteMany
persistentVolumeReclaimPolicy: Retain
nfs:
```

```
path: /
server: 10.9.136.11 # 请修改为你UFS的挂载地址
mountOptions:
- nolock
- nfsvers=4.0 # 必须与创建的UFS协议一致
```

yaml关键字段：

spec.nfs

spec.nfs.path 此处填写UFS挂载点的路径，通过NFS来创建PV，不支持自动创建子目录，你可以预先创建好一个子目录。

spec.nfs.server 此处填写UFS挂载地址

创建pv:

```
# kubectl apply -f ufspv.yml
persistentvolume/ufspv created
```

创建PVC

yaml示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```
name: ufsclaim
spec:
accessModes:
- ReadWriteMany
resources:
requests:
storage: 8Gi
```

创建完PVC后，可以发现PV与PVC已经绑定。

```
# kubectl get pv ufspv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
ufspv 8Gi RWX Retain Bound default/ufsclaim
```

在Pod中挂载UFS

```
apiVersion: v1
kind: Pod
metadata:
name: myufspod
spec:
containers:
- name: myfrontend
```

```
image: uhub.service.ucloud.cn/wxyz/uk8s-helloworld:1.8
volumeMounts:
- mountPath: "/var/www/html"
name: mypd
volumes:
- name: mypd
persistentVolumeClaim:
claimName: ufsclaim
```

创建完Pod之后，我们可以通过''kubectl exec''命令进入容器，执行df命令查看pod是否挂载到UFS

```
# df -h
Filesystem Size Used Avail Use% Mounted on
...
10.19.255.192:/ufs-w4wmpkev 1.0T 0 1.0T 0% /var/lib/kubelet/pods/c800f8a7-5c38-11e9-8aae-525400fa7819/volumes/kubernetes.io~nfs/ufs
...
```

动态PV 使用UFS

背景

前面我们描述了通过创建静态 PV 的方式在 UK8S 中使用 UFS，但这种方式存在两个问题：一是每次都需要手动创建 PV 和 PVC，非常不便；二是无法自动在 UFS 创建子目录，需要预先配置。

下面介绍一个名为nfs-subdir-external-provisioner的开源项目，项目地址为 nfs-subdir-external-provisioner。此项目可以为我们提供一个基于 UFS 的StorageClass：在业务需要 UFS 存储资源时，只需要创建 PVC，nfs-client-provisioner就会自动创建 PV，并在 UFS 下创建一个名为${namespace}-${pvcName}-${pvName}的子目录。

工作原理

我们将 nfs 相关参数通过环境变量传入到nfs-client-provisioner，这个 provisioner 通过 Deployment 控制器运行一个 Pod 来管理 nfs 的存储空间。服务启动后，我们再创建一个StorageClass，其 provisioner 与nfs-client-provisioner服务内的provisioner-name一致。nfs-client-provisioner会 watch 集群内的 PVC 对象，为其提供适配 PV 的服务，并且会在 nfs 根目录下创建对应的目录。这些官方文档的描述已经比较详细了，不再赘述。

以下说明如何在 UK8S 中使用这个服务来管理 UFS。

操作指南

1、克隆项目

克隆项目nfs-subdir-external-provisioner, 在deploy/目录下，我们需要关注三个文件，分别是rbac.yaml, deployment.yaml, class.yaml。

```
# git clone https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner.git
# cd nfs-subdir-external-provisioner/deploy/
# ls
class.yaml kustomization.yaml rbac.yaml test-claim.yaml
deployment.yaml objects test-pod.yaml
```

2、修改nfs-client-provisioner服务的 namespace

我们将把nfs-client-provisioner服务和 RBAC 需要的资源都部署在系统插件所在的kube-systemnamespace。

```
sed -i "s/namespace:.*/namespace: kube-system/g" ./rbac.yaml ./deployment.yaml
```

3、修改deployment.yaml

nfs-client-provisioner服务启动时，需要挂载 UFS，官网文档是通过spec.volume.nfs来声明的，我们这里改为静态声明 PV 的方式。具体原因说明如下:

UFS 文件系统在 mount 到云主机时需要指定额外的mountOption参数，但spec.volume.nfs不支持这个参数。而PersistentVolume的声明中可以支持这个参数，因此我们通过挂载静态 PV 的方式来完成首次挂载。

deployment.yaml文件改动处较多，建议直接替换即可。注意将环境变量NFS_SERVER和NFS_PATH,以及path和server分别修改为UFS的Server地址和挂载路径。

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nfs-client-provisioner
```

```
labels:
app: nfs-client-provisioner
namespace: kube-system
spec:
replicas: 1
strategy:
type: Recreate
selector:
matchLabels:
app: nfs-client-provisioner
template:
metadata:
labels:
app: nfs-client-provisioner
spec:
serviceAccountName: nfs-client-provisioner
containers:
- name: nfs-client-provisioner
# 使用uhub提供的镜像以获取更快的拉取速度
image: uhub.service.ucloud.cn/uk8s/nfs-subdir-external-provisioner:v4.0.2
volumeMounts:
- name: nfs-client-root
mountPath: /persistentvolumes
env:
- name: PROVISIONER_NAME
```

```
value: ucloud/ufs # 修改为ucloud/ufs
- name: NFS_SERVER
value: 10.9.x.x ## 这里需要修改为UFS的Server地址
- name: NFS_PATH
value: / ## 这里改成UFS的挂载路径
volumes:
- name: nfs-client-root
persistentVolumeClaim: #这里由 nfs 改为 persistentVolumeClaim
claimName: nfs-client-root
---
# 创建 PVC
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: nfs-client-root
namespace: kube-system
spec:
volumeName: "nfs-client-root"
storageClassName: "" # 指定为空,不可省略,防止使用默认存储类
accessModes:
- ReadWriteMany
resources:
requests:
storage: 200Gi
---
```

```
#手动创建 PV 并指定 mountOption
apiVersion: v1
kind: PersistentVolume
metadata:
name: nfs-client-root
spec:
capacity:
storage: 200Gi
accessModes:
- ReadWriteMany
persistentVolumeReclaimPolicy: Retain
nfs:
path: / ## 和上面nfs-client-provisioner的NFS_PATH变量保持一致
server: 10.9.x.x ## 这里直接写UFS的Server地址即可。和上面nfs-client-provisioner的NFS_SERVER变量保持一致
mountOptions:
- nolock
- nfsvers=4.0
```

4、修改class.yaml

最后需要修改的是StorageClass的定义文件 class.yaml。

主要是新增了mountOption参数，这个值会传递给nfs-client-provisioner; 如果不加的话，挂载UFS的时候会失败。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
```

```
name: nfs-client
provisioner: ucloud/ufs # 和nfs-client-provisioner的PROVISIONER_NAME环境变量一致
parameters:
onDelete: "retain" # 配置 PV 的回收策略,详情见下文
mountOptions: # 新增mountOption参数
- nolock
- nfsvers=4.0
```

此外您可以在parameters中指定 PV 的回收策略: 删除、保留或者归档。上面示例中配置了删除 PV 后保留对应的目录。

参数	选项及作用	默认选项
onDelete	"delete": 删除目录 "retain": 保留目录 "" (空值): 取决于archiveOnDelete参数	空值
archiveOnDelete	"true": 归档, 目录会被重命名为archived-<volume.Name> "false": 删除目录	"true"

5、执行部署

依次执行

```
# kubectl create -f rbac.yaml
# kubectl create -f deployment.yaml
# kubectl create -f class.yaml
```

6、验证

创建test-nfs-sc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: test-claim
spec:
storageClassName: nfs-client
accessModes:
- ReadWriteMany
resources:
requests:
storage: 1Mi
---
kind: Pod
apiVersion: v1
metadata:
name: test-pod
spec:
containers:
- name: test-pod
image: uhub.service.ucloud.cn/uk8s/busybox:1.31.1
command:
```

```
- "/bin/sh"
args:
- "-c"
- "echo 1 > /mnt/SUCCESS; sleep 10000000"
volumeMounts:
- name: nfs-pvc
mountPath: "/mnt"
restartPolicy: "Never"
volumes:
- name: nfs-pvc
persistentVolumeClaim:
claimName: test-claim
```

创建测试 pod 并验证挂载的 UFS 可读写:

```
# kubectl create -f test-nfs-sc.yaml
# kubectl exec test-pod -- /bin/sh -c 'ls /mnt/ && cat /mnt/*'
SUCCESS
1
# kubectl delete -f test-nfs-sc.yaml
```

升级指南

从旧版external-storage 到新版nfs-subdir-external-provisioner的升级流程

升级背景

旧版provisioner能支持的最新k8s版本为1.23, 其中1.20及以上需要在apiserver上添加--feature-gates=RemoveSelfLink=false支持。
1.24及更新的k8s版本必须使用新版provisioner(因为已经不支持该apiserver参数)。

对存量pvc和pod的影响

使用旧版provisioner (即managed-nfs-storage storage class) 申请的pvc，升级完成后可以继续挂载使用，pod内的挂载不受影响，pod重启后仍然可以挂载。pod和pvc删除后不会清理ufs上对应的文件，如需释放空间应手动删除文件。

升级流程

结合新版provisioner部署文档（见本文前部分）进行升级。

rbac

确认rbac.yaml中指定的namespace与原nfs-client-provisioner ServiceAccount所在namespace相同，比如同为default或同为kube-system, 修改完成后执行kubectl apply -f rbac.yaml升级

deployment

1. 确认namespace与上面rbac部署的namespace相同
2. 删除原来的deployment: kubectl delete deploy nfs-client-provisioner
3. apply新的deployment.yaml。因为创建静态pvc和pv的部分和原来相同, 这一步apply的结果应如下所示

```
deployment.apps/nfs-client-provisioner configured
```

```
persistentvolumeclaim/nfs-client-root unchanged
persistentvolume/nfs-client-root unchanged
```

storage class

直接apply -f class.yaml即可。(旧的storage class managed-nfs-storage 可以不删除)。

在UK8S中使用UPFS

本文档介绍如何在UK8S集群中，使用UPFS作为K8S底层的存储支持，UPFS为共享存储，可以同时为多个Pod提供服务。

UPFS 使用必读

- UPFS产品目前仅支持部分地域
- **UPFS的其他限制条件，请见UPFS产品使用限制**

前置条件

- 在UFS产品页面购买UFS实例并设置好挂载点，操作完毕后，您会得到两个UPFS挂载地址，类似101.66.127.139:10109,101.66.127.140:10109

手动部署CSI

因目前的UK8S版本均不封装UPFS CSI，需要自行部署

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/upfs.25.03.14/rbac-controller.yml
```

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/upfs.25.03.14/rbac-node.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/upfs.25.03.14/csi-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/upfs.25.03.14/csi-node.yml
```

创建存储类StorageClass

接下来进行创建StorageClass操作

创建StorageClass时需要注意以下两个参数:

- uri:文件系统URL(URL详细规则请见UPFS主要概念中的文件系统URL部分)
- path:表示宿主上挂载upfs的目录结构,可自行命名,默认值为: /,一个UPFS实例可以对应多个不同path的StorageClass(同一个UPFS实例即文件系统url,使用相同的path即相同StorageClass的pvc可以实现共享数据,同理,使用不同的path的StorageClass即可实现数据分离)
- autoProvisionSubdir: 该参数需要在upfs-csi版本大于等于upfs-25.03.14支持,默认不启用,开启该参数且配置值为true之后,该StorageClass创建出的pvc可以实现数据分离(针对之前创建pvc的不生效),在upfs上该pvc对应的目录类似: /example/pvc-ae961bc8-2c97-414e-9e7b-bde3e28efee9

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: csi-upfs
provisioner: upfs.csi.ucloud.cn
parameters:
uri: 101.66.127.139:10109,101.66.127.140:10109/upfs-xxxx
path: /example
```

```
# autoProvisionSubdir: "true" # upfs-csi版本大于等于upfs-25.03.14支持
```

创建PVC

yaml示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: logupfs-claim
spec:
storageClassName: csi-upfs
accessModes:
- ReadWriteMany
resources:
requests:
storage: 10Gi # 因实际会将整个UPFS挂载到节点上，故此处的storage可任意配置并不做限制
```

kubectl创建pvc:

```
# kubectl apply -f upfspvc.yml
persistentvolumeclaim/logupfs-claim created
```

创建完PVC后，可以发现PV与PVC已经绑定。

- 可以看到此处PVC的容量为最大256T不做限制（但实际容量务必以对应UPFS实例的容量为准）

```
# kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
pvc-ae961bc8-2c97-414e-9e7b-bde3e28efee9 256Ti RWX Delete Bound default/logupfs-claim csi-upfs 12s
#
# kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
logupfs-claim Bound pvc-ae961bc8-2c97-414e-9e7b-bde3e28efee9 256Ti RWX csi-upfs 12s
```

在Pod中挂载UPFS

```
apiVersion: v1
kind: Pod
metadata:
name: nginx-upfs
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
ports:
- containerPort: 80
```

```
volumeMounts:
- name: testupfs
mountPath: /data
volumes:
- name: testupfs
persistentVolumeClaim:
claimName: logupfs-claim
```

创建完Pod之后，我们可以通过''kubectl exec''命令进入容器，执行df命令查看pod是否挂载到UPFS

```
# df -h
Filesystem Size Used Avail Use% Mounted on
...
UPFS:upfs-xxxx 5.9T 8.5K 5.9T 1% /data
...
```

删除UPFS实例

由于UPFS资源删除需要该UPFS处于未挂载状态，而目前仅删除所有用到该UPFS实例的POD，并不能使UPFS文件系统从云主机侧卸载。

当您不需要使用到UPFS实例想要删除该UPFS实例时，需要从云主机卸载UPFS文件系统。

K8s集群上卸载UPFS文件系统属高危操作，请确认该文件系统不再被pod或其他服务使用后再执行

登录到所有使用过该UPFS的pod所在的node(云主机)上执行命令：

```
## 查看是否有该UPFS的挂载点
# df -h |grep upfs-xxxx
Filesystem Size Used Avail Use% Mounted on
...
UPFS:upfs-xxxx 5.9T 8.5K 5.9T 1% /data/kubelet/plugins/kubernetes.io/csi/upfs.csi.ucloud.cn/uri/101.66.127.139:10109,101.66.127.140:10109/upfs-xxxx

## 卸载UPFS操作
# umount /data/kubelet/plugins/kubernetes.io/csi/upfs.csi.ucloud.cn/uri/101.66.127.139:10109,101.66.127.140:10109/upfs-xxxx
```

在UK8S中使用US3

UK8S支持在集群中使用US3对象存储作为持久化存储卷

支持UK8S版本:大于1.14.6(2019年9月17日之后创建)

US3 使用必读

US3对象存储适合用户上传、下载静态数据文件,如视频,图片等文件。

如果您的业务对于读写性能有很高的需求,如实时快速写入日志,推荐使用UDisk或者UFS作为UK8S集群的持久化存储,US3不能提供像本地文件系统一样的功能。

⚠ 21.09.1 版本之前的CSI UFile,当CSI Pod异常重启时,会造成所在节点上使用US3/UFile的pod挂载点失效,如果您的业务使用US3/UFile,请务必确认当前版本,并按照CSI升级文档尽快升级。如有疑问,请与我们技术支持联系。

⚠ 受限于US3,请勿将同一个bucket绑定给多个PVC并挂载给同一个Pod,会造成Pod无法启动!建议每一个bucket只绑定一个PVC使用。

手动部署CSI

对于没有预装US3 csi的集群，请执行以下命令来部署

集群版本 1.14~1.20

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.21.11.2/csi-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.21.11.2/csi-node.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.21.11.2/rbac-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.21.11.2/rbac-node.yml
```

集群版本 1.22及以上

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.24.10.08_v1.22/csi-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.24.10.08_v1.22/csi-node.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.24.10.08_v1.22/rbac-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.24.10.08_v1.22/rbac-node.yml
```

已支持UK8S挂载US3的地域（持续更新）

UK8S已经支持挂载US3，具体支持地域请查看 US3接入域名

一、创建US3授权Secret

由于US3带有地域属性和操作权限控制，我们需要手动创建Secret和StorageCLass。

首先我们事先在US3的控制台创建好对象存储目录，并为这个目录生成一个授权令牌（Token），如图：

Token创建管理教程可以参考文档。

在Kubernetes中为此令牌创建Secret，如下所示：

```
apiVersion: v1
kind: Secret
metadata:
name: us3-secret
namespace: kube-system
stringData:
accessKeyID: TOKEN_9a6ec9fd-9cb7-4510-8ded-xxxxxxxx # 非账号公钥，为US3的令牌公钥。
secretAccessKey: c429c8e5-e4e6-4366-bf93-xxxxxx # 非账号私钥，为US3的令牌私钥。
endpoint: http://internal.s3-cn-bj.ufileos.com
```

字段说明：

accessKeyID: US3公钥

secretAccessKey: US3私钥

endpoint: 对应地域接入S3服务URL，具体请查看US3接入域名

> 对应地域服务URL参考已支持UK8S挂载US3的地域（持续更新）章节，推荐使用内网地址。

二、创建存储类StorageClass

接下来进行创建StorageClass操作，如下可以看到我们在这个StorageClass定义了US3的backet并关联使用了前一步创建的Secret。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
name: csi-ufile
provisioner: ufile.csi.ucloud.cn
parameters:
bucket: csis3-bucketname # 事先申请好的US3 Bucket
path: /csis3-dirname/ # 表示挂载时相对Bucket根文件的目录结构，默认为/（23.09.12及之后版本支持）。
csi.storage.k8s.io/node-publish-secret-name: us3-secret # 关联前一步创建的Secret
csi.storage.k8s.io/node-publish-secret-namespace: kube-system
```

```
# 如所需挂载的US3 Bucket与所用uk8s集群不在同一项目下，请配置以下两个参数(24.10.08及之后版本支持)
csi.storage.k8s.io/provisioner-secret-name: us3-secret # 关联前一步创建的Secret
csi.storage.k8s.io/provisioner-secret-namespace: kube-system # 仅支持kube-system
```

三、创建持久化存储卷声明（PVC）

US3单个Bucket的存储空间理论上是无上限的，所以PV和PVC中的容量参数没有实际意义，这里的requests信息不会真实生效。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: logs3-claim
spec:
storageClassName: csi-ufile
accessModes:
- ReadWriteMany
resources:
requests:
storage: 10Gi
```

四、在pod中使用PVC

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:latest
ports:
- containerPort: 80
volumeMounts:
- name: test
mountPath: /data
volumes:
- name: test
persistentVolumeClaim:
claimName: logs3-claim
```

CSI更新

在操作CSI更新之前，请务必仔细阅读下面的注意事项，如果有疑问，请咨询我们的技术支持。

1. 注意事项

- 集群进行存储插件升级时，需要操作**k8s关键组件**，请在业务低谷期操作，并且请勿进行服务发布。
- **切勿自行通过修改CSI的image的方式进行升级，否则CSI将无法工作。**请一定在控制台完成CSI的升级。
- 如果集群版本不在我们的维护版本之内，控制台将无法直接进行升级，参见：UK8S版本维护说明。
- **22.09.1**：如果要使用**RSSD云盘，请将CSI升级到22.09.1或以上的版本**，详情见RSSD云盘挂载问题。
- **21.09.1**：老版本升级到21.09.1或者以上的版本，会造成使用US3/UFile的pod挂载点失效，如果您的业务使用了US3/UFile，请务必确认当前版本，如有疑问，请与我们技术支持联系。

2. 版本查看及插件升级

在 UK8S 集群控制台管理页面「插件-存储插件」页面，开启 CSI 存储插件升级功能，开启 CSI 插件功能会在集群中执行 CSI 插件查询任务，大约需要 3 分钟，在此过程中请不要操作集群。升级功能开启后，即可看到 CSI 插件版本信息，点击「升级 CSI」即可进行升级。

升级过程约需要 1 分钟，升级过程中「当前版本」字段会显示为「升级中」，升级完成后显示最新版本号，如升级失败，请与我们技术支持联系。

当所有节点都升级成功后，可关闭插件升级服务，后续有升级需求时再开启。

3. 手动升级

如果集群版本不在我们的维护版本之内，您可以手动升级csi，请执行下面的命令(仅能升级到旧版本，不建议用RSSD云盘)：

UDisk CSI升级

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/udisk.21.11.3-lts/csi-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/udisk.21.11.3-lts/csi-node.yml
```

UFile CSI升级

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.21.11.2/csi-controller.yml
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/volume/us3.21.11.2/csi-node.yml
```

4. 变更记录

版本名称中带**se**的表明是我们自研的调度扩展组件版本，用于支持UDisk的调度,详情见RSSD云盘挂载问题。其升级方式也是在 UK8S 集群控制台管理页面「插件-存储插件」页面点击「升级 CSI」按钮。

版本	更新时间	更新内容
se25.02.13	2025.02.13	修复**se25.01.07**中的bug：兼容pvc和pv的storageclass不一致的极端情况
25.01.16	2025.01.16	udisk云盘以10g为计费单位，不满10g按10g计算，超过10g向上取整到10的整数倍

se25.01.16	2025.01.16	支持RSSD云盘跨Rdma Cluster挂载云主机
se25.01.07	2025.01.07	Pod调度时，限制主机云盘数量，并禁止调度至迁移中的主机(此版本存在不兼容pvc和pv的storageclass不一致的bug，该bug在**se25.02.13**中修复)
24.10.08	2024.10.08	支持跨项目挂载us3 bucket
24.08.14	2024.08.14	修复裸金属机器挂载云盘时的校验错误的问题
24.07.18	2024.07.18	挂载云盘时增加校验，防止在管理信息错误的情况下进行挂载
24.07.01	2024.07.01	下线了API调用监控指标上报，用户侧无影响
24.06.07	2024.06.07	解决同时卸载多块云盘时偶发卸载失败的问题
23.09.12	2023.09.12	1. 支持挂载us3 bucket指定目录; 2. 修复容器中非root用户可能无法访问us3控制台创建的文件的问题
23.07.24	2023.08.01	RSSD云盘支持裸金属
22.09.1	2022.09.17	动态调度RSSD云盘，以解决RSSD云盘挂载问题
21.11.3-lts	2024.12.16	基于21.11.2版本，解决同时卸载多块云盘时偶发卸载失败的问题
21.11.2	2021.11.22	1. csi udisk支持方舟模式; 2. csi udisk支持指定业务组; 3. 优化us3 挂载参数
21.11.1	2021.11.04	1. 适配了 s3fs 返回成功而实际挂载失败的情况； 2. 修复因 US3 公私钥长度变化导致的挂载失败； 3. 始终通过节点 us3lancher 服务操作挂载

21.09.1	2021.09.07	将s3fs挂载操作放在Node上进行，使用US3/UFile存储注意，升级到此版本会造成使用US3/UFile的pod挂载点失效；
21.08.1	2021.08.12	优化 CSI 插件调度机制，避免被驱逐
21.07.1	2021.07.05	支持云盘裸金属
21.04.1	2021.04.28	支持 UDisk 相关参数暴露在 Kubelet Metrics 中
21.03.1	2021.03.15	解决节点被删除后，volumeattachment 未被删除导致存储无法卸载的问题
21.01.1	2021.01.13	UDisk的起始大小变更为1GB
20.10.1	2020.10.14	支持CSI限制节点最大可挂载卷的数量 避免UCloud API客户端可能产生的并发竞争行为

存储常见问题

1. PV PVC StorageClass 以及 UDisk 的各种关系?

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: udisk-ssd-test
provisioner: udisk.csi.ucloud.cn #存储供应方,此处不可更改。
---
apiVersion: v1
kind: PersistentVolumeClaim
spec:
  storageClassName: ssd-csi-udisk
```

用户只需要设置好 StorageClass,在使用 pvc 时,csi-udisk 插件会自动完成 UDisk 的创建挂载 mount 等一系列的操作,主要流程如下

1. StorageClass 设置相关参数,与 CSI 插件绑定。
2. pvc 与 StorageClass 进行绑定。
3. K8S 观察到使用 StorageClass 的新建 pvc,会自动创建 pv,并交给 CSI 插件完成新建 UDisk 的工作。
4. pv 与 pvc 绑定完成,CSI 插件完成后续 UDisk 的挂载和 mount 等工作。
5. UCloud 的 CSI 插件查看可以通过`kubectl get pods -o wide -n kube-system |grep udisk` 查看(一个总的 controller 及每个 node 对应的 pod)

1.1 Statefulset 中使用 PVC

1. Statefulset 控制器中的 pvctemplate 字段，可以设置 K8S 集群在对应 pvc 不存在时自动创建pvc，使得上述流程更加自动化(pvc和pv均由UK8S来建)。
2. Statefulset 只负责创建不负责删除 pvc，因此对应多余的 pvc 需要手动删除

2. VolumeAttachment 的作用

VolumeAttachment 并不由用户自己创建，因此很多用户并不清楚它的作用，但是在 pvc 的使用过程中，VolumeAttachment 有着很重要的作用

1. VolumeAttachment所表示的，是 K8S 集群中记载的 pv 和某个 Node 的挂载关系。可以执行`kubectl get volumeattachment |grep pv-name` 进行查看
2. 这个挂载关系和 UDisk 与云主机的挂载关系往往是一致的，但是有时可能会出现不一致的情况。
3. 不一致的情况多见于 UDisk 已经从云主机卸载，但是 VolumeAttachment 记录中仍然存在，UDisk 是否挂载在云主机上，可以通过如何查看 PVC 对应的 UDisk 实际挂载情况来查看
4. 对于不一致的情况，可用选择手动删除对应的 VolumeAttachment 字段，并新建一个相同的 VolumeAttachment（新建后 ATTACHED 状态为 false）
5. 如果不能删除，可以通过`kubectl logs csi-udisk-controller-0 -n kube-system csi-udisk` 查看 csi-controller 日志定位原因
6. 一般 kubelet 手动删除不掉的情况，可能是对应的节点已经不存在了，此时直接 edit volumeattachment 删除 finalizers 字段即可

```
[root@10-9-112-196 ~]# kubectl get volumeattachment |grep pvc-e51b694f-ffac-4d23-af5e-304a948a155a
NAME                                                   ATTACHER          PV                                     NODE         ATTACHED   AGE
csi-1d52d5a7b4c5c172de7cfc17df71c312059cf8a2d7800e05f46e04876a0eb50e   udisk.csi.ucloud.cn   pvc-e51b694f-ffac-4d23-af5e-
304a948a155a   10.9.184.108   true       2d2h
```

2.1 VolumeAttachment 文件示例

```
apiVersion: storage.k8s.io/v1
kind: VolumeAttachment
```

```
metadata:
  annotations:
    csi.alpha.kubernetes.io/node-id: 10.9.184.108 # 绑定的节点ip
  finalizers:
  - external-attacher/udisk-csi-ucloud-cn
  name: csi-1d52d5a7b4c5c172de7cfc17df71c312059cf8a2d7800e05f46e04876a0eb50e #  VolumeAttachment名称
spec:
  attacher: udisk.csi.ucloud.cn
  nodeName: 10.9.184.108 #绑定的节点ip
  source:
    persistentVolumeName: pvc-e51b694f-ffac-4d23-af5e-304a948a155a # 绑定的pv名称
```

3. 如何查看 PVC 对应的 UDisk 实际挂载情况

对应关系表

UK8S资源类型	与主机对应关系
PV	UDisk 的磁盘
VolumeAttachment	磁盘与主机的挂载关系(vdb,vdc 的块设备)
PVC	磁盘在主机上mount的位置
pod	使用磁盘的进程

1. kubectl get pvc -n ns pvc-name 查看对应的 VOLUME 字段，找到与 pvc 绑定的 pv，一般为（pvc-e51b694f-ffac-4d23-af5e-304a948a155a）
2. kubectl get pv pv-name -o yaml 在 spec.csi.volumeHandle 字段，可以查看到改 pv 绑定的 UDisk盘(flexv 插件为 pv 的最后几位)
3. 在控制台查看该udisk盘的状态,是否挂载到某个主机
4. kubectl get volumeattachment |grep pv-name 查看 K8S 集群内记录的磁盘挂载状态
5. ssh 到对应的主机上，lsblk可以看到对应的盘
6. mount |grep pv-name 可用查看盘的实际挂载点，有一个 globalmount 及一个或多个 pod 的 mount 点

```
[root@10-9-184-108 ~]# mount |grep pvc-e51b694f-ffac-4d23-af5e-304a948a155a
/dev/vdc on /data/kubelet/plugins/kubernetes.io/csi/pv/pvc-e51b694f-ffac-4d23-af5e-304a948a155a/globalmount type ext4 (rw,relatime)
/dev/vdc on /data/kubelet/pods/587962f5-3009-4c53-a56e-a78f6636ce86/volumes/kubernetes.io~csi/pvc-e51b694f-ffac-4d23-af5e-
304a948a155a/mount type ext4 (rw,relatime)
```

4. 磁盘挂载的错误处理

1. 由于磁盘内容多流程长，建议在出现问题时，首先确定当前状态如何查看 PVC 对应的 UDisk 实际挂载情况
2. 如果有UK8S中状态和主机状态不一致的情况，首先进行清理，删除掉不一致的资源，之后走正常流程进行恢复

4.1 PV 和 PVC 一直卡在 terminating/磁盘卸载失败怎么办

1. 通过如何查看 PVC 对应的 UDisk 实际挂载情况确定当前 pv 和 pvc 的实际挂载状态
2. 手动按照自己的需求进行处理，首先清理所有使用该 pv 和 pvc 的所有 pod（如果 pvc 已经成功删除，则不需要这一步）
3. 如果删除 pvc 卡在 terminating，则手动 umount 掉对应的挂载路径
4. 如果删除 VolumeAttachment 卡在 terminating，则手动在控制台卸载掉磁盘（如果卡在卸载中找主机处理）
5. 如果删除 pv 卡在 terminating，则手动在控制台删除掉磁盘（删除 pv 前需要确保相关的 VolumeAttachment 已经删除完成）
6. 确保手动释放完成对应的资源后，可以通过kubectl edit 对应的资源,删除掉其中的 finalizers 字段，此时资源就会成功释放掉
7. 删除 VolumeAttachment 后，如果 pod 挂载报错，按照VolumeAttachment 文件示例中提供的yaml文件，重新补一个同名的 VolumeAttachment 即可

4.2 Pod 的 PVC 一直挂载不上怎么办?

1. kubectl get pvc -n ns pvc-name 查看对应的 VOLUME 字段，找到与 pvc 绑定的 pv，一般为(pvc-e51b694f-ffac-4d23-af5e-304a948a155a)
2. kubectl get pv pv-name -o yaml 在 spec.csi.volumeHandle 字段，可以查看到改 pv 绑定的 UDisk 盘(flexv 插件为 pv 的最后几位)
3. 找到 UDisk 磁盘后，如果控制台页面中磁盘处于可用状态或者挂载的主机不是 pod 所在主机，可以找技术支持，查看该 UDisk的挂载和卸载请求的错误日志，并联系主机同时进行处理
4. 如果没有 UDisk相关的错误日志，联系UK8S值班人员，并提供kubectl logs csi-udisk-controller-0 -n kube-system csi-udisk的日志输出及 pod 的event

5. UDisk-PVC 使用注意事项

1. 由于 UDisk 不可跨可用区，因此在建立 StorageClass 时必须指定 volumeBindingMode: WaitForFirstConsumer
2. 由于 UDisk 不可多点挂载，因此必须在 pvc 中指定 accessModes 为 ReadWriteOnce
3. 基于 UDisk 不可多点挂载，多个 pod 不可共用同一个 udisk-pvc，上一个 pod 的 udisk-pvc 未处理干净时，会导致后续 pod 无法创建，此时可以查看 VolumeAttachment 的状态进行确认

6. K8S 1.17 版本升级到 1.18 过程中云盘 Detach 问题

我们发现在 UK8S 集群从 1.17 升级至 1.18 的过程中，部分挂载 PVC 的 Pod 会出现 IO 错误。查相关日志发现是因为挂载的盘被卸载导致 IO 异常。

社区在 1.18 版本为了解决 Dangling Attachments 引入该问题。参见 Recover CSI volumes from dangling attachments

K8S 处理挂盘和卸盘的实现中，单个 Node 可以选择由 kubelet 和 controller-manager 进行管理挂盘和卸盘，上面的代码在解决 dangling attachments 问题时引入了一个新的问题，由 kubelet 管理挂盘的 Node 节点，在 controller-manager 重启后，该节点的磁盘会被强制卸载掉。

为了解决该问题，需要将由 kubelet 负责挂盘的节点改为由 controller-manager 负责挂盘。UK8S 添加的节点已经默认使用 controller-manager 负责挂盘，后续添加节点无需再手动更改

6.1 手动修改节点为controller-manager挂盘

检查 Kubelet 配置

检查节点的 /etc/kubernetes/kubelet.conf 的配置。如果 enableControllerAttachDetach 的值为 false 则需要把该值修改为 true。

然后执行命令 systemctl restart kubelet 重启 Kubelet。

检查 Node 状态

执行命令 kubectl get no $IP -o yaml 查看 Node 的 status 中 volumesAttached 是否有数据，且数据是否与 volumesInUse 的数据一致。

Node annotations 中应该有 volumes.kubernetes.io/controller-managed-attach-detach: "true" 的记录。

如确认上述数据一致，且 Annotations 中有相应记录，则可以正常进行升级。如有问题，请联系技术支持。

7. Flexv 插件导致 pod 删除失败

7.1 现象描述

使用flexv插件自动创建pv绑定到pod，删除pod时，有可能导致pod 处于Terminating状态，不能正常删除。

- kubernetes版本: 1.13
- 插件版本：Flexvolume-19.06.1

7.2 问题原因

kubelet重启后找不到volume对应的Flexvolume插件。kubelet在重启之后如果发现了orphan pod（正常的pod不会导致这个问题），就会通过pod记录volume的路径来推断出使用的

插件，但是flexv会在插件前面加入flexvolume-字段，导致kubelet推断出的名字和flexv提供的名字匹配不上。kubelet日志中会报**no volume plugin matched** 的错误，进而导致pod卡在Terminating的状态。

具体可以查看下面issue

- https://github.com/kubernetes/kubernetes/issues/80972
- https://github.com/kubernetes/kubernetes/pull/80973

7.3 解决方案

手动umount掉当前pod使用的路径，并进行清理操作。

> 谨慎操作，本操作是代替kubelet手动进行资源清理，请阅读结束下面所有步骤再进行操作.

1. 找到不能正常umount的pv。
2. 登录到node节点上查看mount记录。

```
mount | grep pv-name
```

3. 记录上一步匹配到的所有路径**path**,手动umount掉pv在当前节点下的路径。

```
umount path
```

4. 在上一步umount中，会有一个以/var/lib/kubelet/pods开头的目录，umount之后需要手动删除该目录。
5. 删除pvc，删除pvc之后需要手动在控制台卸载掉对应的udisk。udisk的id为pv名字的最后几位，例如pv名字是pvc-58f9978e-3133-11ea-b4d6-5254000cee42-bsm-olx0uqti，则对应的udisk名字就是bsm-olx0uqti。也可以通过describe pv拿到spec.flexVolume.options中的diskId字段。

8. 其他常见存储问题汇总

8.1 一个PVC可以挂载到多个 pod 吗?

UDisk不支持多点读写,如需要多点读写请使用UFS。

8.2 Pod删除后，如何复用原先的云盘?

可以使用静态创建PV的方法进行原有云盘绑定的方法进行复用原有云盘,详见在UK8S中使用已有UDISK

8.3 默认情况下通过原生的 nfs 直接挂载的方式是没有办法如何设置自定义参数?

不能在 Pod spec 中指定 NFS 挂载可选项。可以选择设置服务端的挂载可选项,或者使用 /etc/nfsmount.conf。此外,还可以通过允许设置挂载可选项的持久卷挂载 NFS 卷。详细可参考Kubernetes官方文档nfs的使用

8.4 upfs的pvc挂载pod失败

先确认upfs的csi部署文件是否与在UK8S中部署UPFS CSI中一致,如不一致,需要使用该文档中的部署文件部署

因同一个upfs文件系统在一个机器上只能mount一次,所以先检查对应节点上是否存在该upfs实例的挂载点,如果存在,则需要umount后,csi才会执行挂载pvc

9. 挂载UDisk云盘的Pod调度问题

⚠ **RSSD云盘挂载涉及到RDMA问题，需要动态调度，涉及内容较多且更加复杂，因此单独在文档RSSD云盘挂载问题中讲解，下面只涉及到非RSSD云盘静态调度的场景。**

相较于普通Pod，使用UDisk的Pod调度涉及到了UDisk自身挂载规则的限制，更为复杂。具体限制如下

- 普通云盘和SSD云盘挂载要求必须与云主机处于相同可用区

UDisk挂载限制在实际UK8S的使用中主要体现到以下两个方面

- 自动创建PV的过程中，如何判定创建哪个可用区的云盘
- 当Pod需要重新调度时，如何保证新调度的节点满足云盘挂载的要求

UK8S提供的csi-udisk插件，依赖K8S提供的CSI插件能力，帮助用户实现了尽可能少的介入，下面以SSD UDisk为例进行讲解。

9.1 创建PVC时自动创建UDisk

从上面的文档中可以了解到，当PVC创建完成时，CSI会自动创建PV以及UDisk，并完成绑定工作。但是创建哪个可用区的UDisk呢，如果随意选择，则会导致后续Pod调度完成后无法挂载云盘。

为此K8S提供了`WaitForFirstConsumer`机制。当StorageClass中指定了`volumeBindingMode: WaitForFirstConsumer`参数时，CSI不会立刻创建PV及云盘，以下为`WaitForFirstConsumer`模式下的工作流程。

- 手动创建PVC
- 创建Pod，并且在Pod绑定上一步中定义的PVC
- 等待Pod进行调度，此时k8s会在PVC的Annotations中增加一个字段`volume.kubernetes.io/selected-node`,用以记录Pod预计调度到的Node。注意此时查看Pod状态仍然为Pending。
- CSI查询Node云主机的可用区，创建相同可用区的云盘，并创建相应PV进行绑定
- CSI更新PV中的`spec.csi.volumeHandle`字段，记录创建的云盘ID
- CSI更新PV中的`spec.nodeAffinity`字段，记录云盘所在的可用区等信息

按照以上逻辑，可以保证Pod调度后创建的云盘顺利挂载到对应主机

但是有一个特殊情况，RSSD盘仅能挂载到快杰机型上，如果Pod首次调度到了非快杰机型上，那么后续创建云盘就会失败，因此如果您选择了RSSD盘，请确保Pod首次调度到快杰机型上。

9.2 Pod重建后调度流程

首次运行后，如果遇到服务更新，或者节点故障等原因触发Pod重建，会进行重新调度，以下为调度流程

- 清理旧Pod，完成UDisk从旧节点上清理卸载工作
- 创建新Pod
- K8S调度器会按照PV中的`spec.nodeAffinity`字段，校验节点是否可以调度
- 如果所有节点都不满足磁盘调度要求，会记录`had volume node affinity conflict`类型的EVENT到Pod，并重复上一步流程
- K8S调度器按照上一步过滤的结果，在可调度的节点范围内，继续按照普通Pod调度流程进行调度

10. CSI组件工作原理

CSI是K8S定义的容器存储接口，可以对接云厂商的多种存储。UCloud目前实现了UDisk以及UFile/US3的CSI插件。

CSI组件分为两大类，分别为Controller以及Daemonset。目前所有csi组件的pod均默认运行在`kube-system`下面，可以通过执行`kubectl get pods -n kube-system -o wide |grep csi` 进行查看。
如果遇到存储挂载问题，可以优先查看CSI Controller是否工作正常，以及节点上是否存在对应CSI Daemonset的Pod。

接下来对CSI组件进行简要介绍。

10.1 CSI Controller

CSI Controller 负责的是全局资源的管理，通过list/watch k8s中的相关资源，执行对应操作。

UDisk CSI Controller 会负责磁盘创建和删除，磁盘到云主机的卸载及挂载操作。

US3 CSI Controller 由于无需处理挂载操作，仅仅负责校验一些StorageClass中的基础信息。

10.2 CSI Daemonset

CSI Daemonset组件调度到各个节点上，负责单个节点的一些工作。与Controller模式不同，CSI Daemonset通过unix socket地址与kubelet进行通信，接收kubelet请求信息执行对应的操作。通常CSI unix socket地址为/var/lib/kubelet/csi-plugins/csi-name/csi.sock
UDisk/US3 CSI Daemonset 主要负责存储的Mount以及Umount操作

10.3 其它功能

在基础的存储管理以及挂载功能外，CSI还提供了多种其它能力。目前CSI UDisk 则实现了磁盘动态扩容（需要Controller与Daemonset）以及磁盘Metrics信息收集(需要CSI Daemonset)。

11 CSI常见问题排查

本节会以UDisk-CSI为例，从创建pvc之后每一步可能出错的点进行分析，并给出处理建议。另外本节内容仅涉及Pod创建过程中的相关内容。并基于一个假设，即上一个使用该PVC的Pod已经销毁，并且中间的所有操作及资源已经清理干净。如果上一个Pod使用的资源没有清理干净，也可以依赖本文档反推确认清理方案。

1. 通过 kubectl get pods -n kube-system -o wide 确认csi的controller及目标节点上Daemonset组件均工作正常
2. 确认PV是否创建成功，如果没有，请查看 11.1 小节
3. PV创建完成后，需要确保Pod成功调度。使用了udisk的Pod在普通调度规则上，会有额外的调度要求，具体可以看第9节
4. 如果磁盘挂载失败，请查看 11.2 小节
5. 当确认磁盘已经挂载到目标主机后，需要确认mount成功，如果mount失败，请查看11.3小节

11.1 PV没有创建成功

如果PV没有创建成功，需要确保有Pod在使用该PVC。具体原因请查看 第9.1节。

如果已有Pod在使用该PVC，则通过kubectl logs csi-udisk-controller-0 -n kube-system csi-udisk 查看controller日志，确认是否存在创建udisk失败的日志。

通过kubectl get pv <pv-name> -o yaml 记录下PV对应的udisk名称，并在控制台中查看对应的udisk存在。

一般自动创建的pv名字格式是pvc-xxxxxxxxx，这里比较容易混淆。

11.2 磁盘挂载失败

11.2.1 确保volumeattachment资源存在

为了能成功挂盘，首先需要确保volumeattachment资源存在，并且查看node的信息，确认当前是由kubelet还是controller-manager负责挂盘。

1. kubelet挂盘方式存在缺陷，目前k8s推荐使用controller-manager进行挂盘，具体查看及转换方式可以对照本文档6.1小节
2. 如果kubelet负责挂盘，并且pod日志中显示类似volumeattachment资源不存在的情况，则需要按照文档VolumeAttachment 文件示例中提供的yaml文件，重新补一个同名的volumeAttachment。
3. 如果是controller-manager负责挂盘，则需要确认k8s版本是否为1.17.1-1.17.7或1.18.1-1.18.4，这些版本controller-manager挂盘存在性能问题。
4. controller-manager日志查看方式，登录到三台master节点，执行journalctl -fu kube-controller-manager查看，注意三台master中仅有一台Master中的controller-manager为leader，即实际工作状态。
5. kubelet日志查看方式，需要登录到目标节点，执行journalctl -fu kubelet

11.2.2 确保磁盘挂载成功

1. 首先需要确认volumeattachment资源状态为true。
2. 如果状态不为true，可以查看csi-controller是否挂载过程中存在报错。
3. 如果状态为true，需要在控制台确认udisk确实挂载到了目标主机，如果确认有问题，可以联系技术支持。
4. 另外，此时需要确认，仅有一个对应的volumeattachment。因为udisk仅允许单点挂载，而us3由于允许多点挂载，并没有此限制。

11.3 磁盘Mount问题

1. 首先需要确认磁盘对应的盘符，udisk挂载由于实现原理的限制。在某些特殊情况下，页面看到的盘符和真实盘符可能不一致，盘符对应信息可以从/sys/block/vdx/serial 文件中查看到。udisk-csi已经实现了该逻辑，不会有错误挂盘的出现，但是手动排查问题需要了解此情况。
2. 确认好磁盘对应的盘符之后，可以通过mount |grep pv-name 查看挂载路径。

3. udisk根据csi标准实现了globalmount及pod mount路径，因此一个udisk正常情况下会看到两个挂载路径，一个以globalmount结尾，一个以mount结尾。
4. us3仅实现了pod mount路径，因此仅能看到一个挂载路径，且us3也不需要确认盘符。

11.4 fsGroup导致的磁盘mount缓慢/错误

1. 很多用户会遇到一个磁盘mount缓慢的问题。此时需要首先确认是否设置了fsGroup，且磁盘中的是否存在大量小文件，如果两个条件均满足，则很可能导致挂载缓慢，具体可以查看k8s官方文档。
2. pod在设置了securityContext.fsGroup之后如果存储类中没有fsType(默认有)则会导致kubelet无法正确的设置权限，出现permission denied错误。

11.5 updatedb导致的umount不成功

Centos系统会每日定时执行updatedb命令，该命令会遍历整个文件系统目录树。当挂载的udisk或us3数据量较大时，updatedb扫描耗时会较长，扫描期间的umount pvc操作都会失败。

解决方案

将pod挂载pvc的目录加入到允许updatedb跳过的目录列表。

编辑/etc/updatedb.conf，修改PRUNEPATHS变量如下即可：

```
PRUNEPATHS = "/var/lib/kubelet /data/kubelet /afs /media /mnt /net /sfs /tmp /udev /var/cache/ccache /var/lib/yum/yumdb /var/spool/cups /var/spool/squid /var/tmp /var/lib/ceph"
```

什么是Prometheus

关于Prometheus

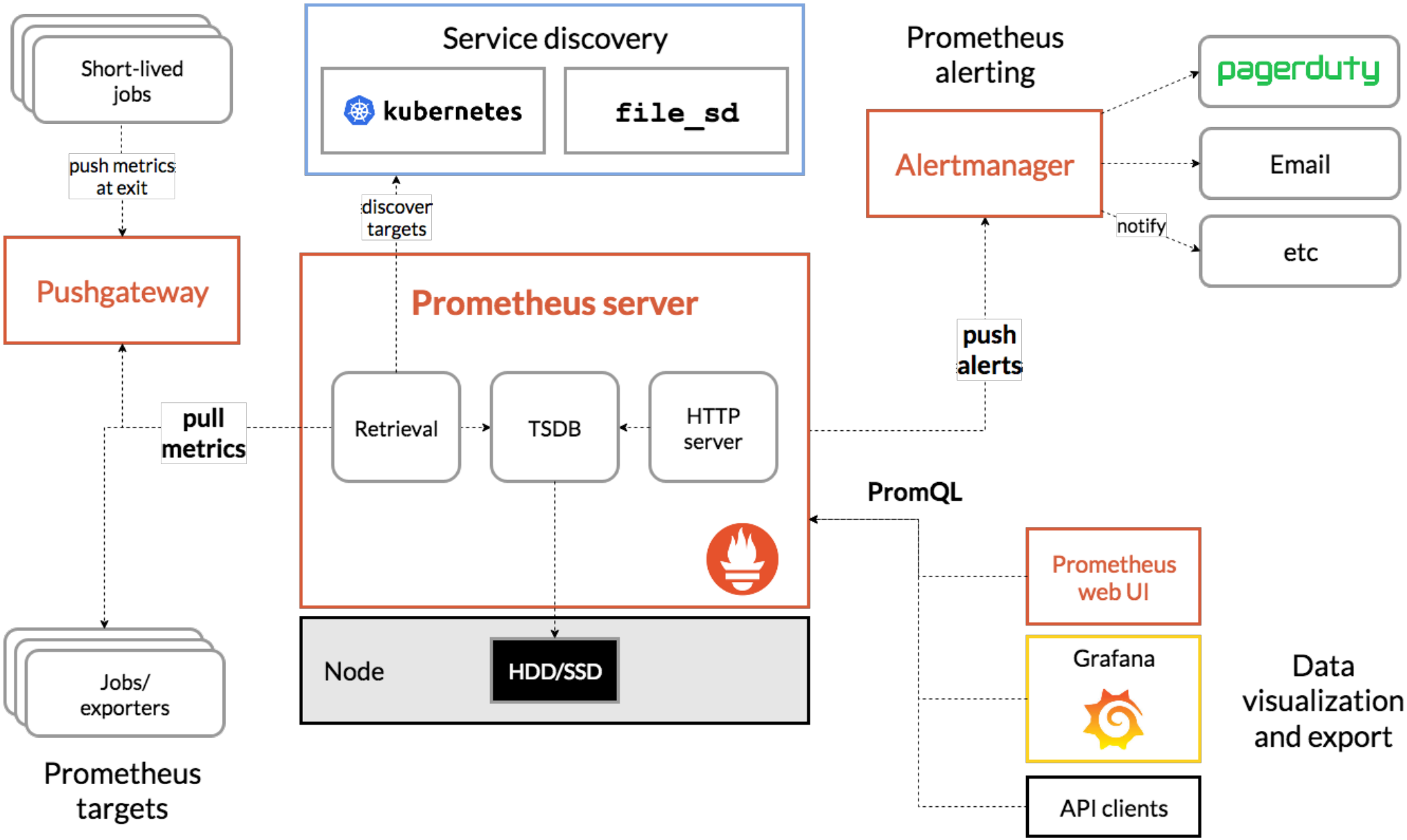
Prometheus 是一套开源的系统监控报警框架。它的设计灵感源于 Google 的 borgmon 监控系统，由SoundCloud 在 2012 年创建，后作为社区开源项目进行开发，并于 2015 年正式发布。2016 年，Prometheus 正式加入 Cloud Native Computing Foundation(CNCF)，成为受欢迎度仅次于 Kubernetes 的项目，目前已广泛应用于Kubernetes集群监控系统中，大有成为Kubernetes集群监控标准方案的趋势。

Prometheus的优势

- **强大的多维度数据模型：**
  - 时间序列数据通过 metric 名和键值对来区分。
  - 所有的 metrics 都可以设置任意的多维标签。
  - 数据模型更随意，不需要刻意设置为以点分隔的字符串。
  - 可以对数据模型进行聚合，切割和切片操作。
  - 支持双精度浮点类型，标签可以设为全 unicode。
- **灵活而强大的查询语句（PromQL）：**在同一个查询语句，可以对多个 metrics 进行乘法、加法、连接、取分数位等操作。
- **易于管理：** Prometheus server 是一个单独的二进制文件，可直接在本地工作，不依赖于分布式存储。
- **高效：**平均每个采样点仅占 3.5 bytes，且一个 Prometheus server 可以处理数百万的 metrics。
- **动态获取：**可以通过服务发现或者静态配置去获取监控的 targets。
- 使用 pull 模式采集时间序列数据，可以避免有问题的服务器推送坏的 metrics。
- 支持 push gateway 的方式把时间序列数据推送至 Prometheus server 端。
- 多种可视化图形界面。

Prometheus架构及组件

图片源于Prometheus官方文档



上图为Prometheus的架构图，包含了Prometheus的核心模块及生态圈中的组件，简要介绍如下：

- **Prometheus Server:** 用于收集和存储时间序列数据。
- **Client Library:** 客户端库，为需要监控的服务生成相应的 metrics 并暴露给 Prometheus server。当 Prometheus server 来 pull 时，直接返回实时状态的 metrics。
- **Push Gateway:** 主要用于短期的 jobs。由于这类 jobs 存在时间较短，可能在 Prometheus 来 pull 之前就消失了。为此 jobs 可以直接向 Prometheus server 端推送它们的 metrics。这种方式主要用于服务层面的 metrics，对于机器层面的 metrices，建议使用 node exporter。
- **Exporters:** 用于暴露已有的第三方服务的 metrics 给 Prometheus。
- **Alertmanager:** 从 Prometheus server 端接收到 alerts 后，会去除重复数据，分组，并路由到对应的接受方式，发出报警。

工作原理

如上图可见，Prometheus 的主要模块包括：Prometheus server, exporters, Pushgateway, PromQL, Alertmanager 以及图形界面，其大概的工作流程是：

1. Prometheus server 定期从配置好的 jobs 或者 exporters 中拉 metrics，或者接收来自 Pushgateway 发过来的 metrics，或者从其他的 Prometheus server 中拉 metrics。
2. Prometheus server 在本地存储收集到的 metrics，并运行已定义好的 alert.rules，记录新的时间序列或者向 Alertmanager 推送警报。
3. Alertmanager 根据配置文件，对接收到的警报进行处理，发出告警。
4. 在图形界面中，可视化采集数据。

Prometheus 工作的核心，是使用 Pull（抓取）的方式去搜集被监控对象的 Metrics 数据（监控指标数据），然后，再把这些数据保存在一个 TSDB（时间序列数据库，比如 OpenTSDB、InfluxDB 等）当中，以便后续可以按照时间进行检索。

适用场景

Prometheus非常适合记录纯时间序列的数据。它既适用于面向服务器等硬件指标的监控，也适用于高动态的面向服务架构的监控。对于现在流行的微服务，Prometheus的多维度数据收集和数据筛选查询语言也是非常的强大。Prometheus是为服务的可靠性而设计的，当服务出现故障时，它可以使你快速定位和诊断问题。它的搭建过程对硬件和服务没有很强的依赖关系。

Prometheus重视可靠性，即使在故障情况下，您也可以随时查看有关系统的可用统计信息。如果您需要100%的准确度，例如按请求计费，Prometheus不是一个好的选择，因为收集的数据可能不够详细和完整。

总之，在需要高可用性的业务场景，Prometheus是一个非常好的选择，但对于高精度、高准确率的业务场景，Prometheus并非最佳选择。

核心概念

为了在 Prometheus 的配置和使用中可以更加顺畅，我们对 Prometheus 中的数据模型、metric 类型以及 instance 和 job 等概念做个简要介绍。

数据模型

Prometheus 中存储的数据为时间序列，是由 metric 的名字和一系列的标签（键值对）唯一标识的，不同的标签则代表不同的时间序列。

- **metric 名字：**该名字应该具有语义，一般用于表示 metric 的功能，例如：http_requests_total, 表示 http 请求的总数。其中，metric 名字由 ASCII 字符，数字，下划线，以及冒号组成，且必须满足正则表达式 [a-zA-Z_:][a-zA-Z0-9_:]*。
- **标签：**使同一个时间序列有了不同维度的识别。例如 http_requests_total{method="Get"} 表示所有 http 请求中的 Get 请求。当 method="post" 时，则为新的一个 metric。标签中的键由 ASCII 字符，数字，以及下划线组成，且必须满足正则表达式 [a-zA-Z_:][a-zA-Z0-9_:]*。
- **样本：**实际的时间序列，每个序列包括一个 float64 的值和一个毫秒级的时间戳。
- **格式：** 如http_requests_total{method="POST",endpoint="/api/tracks"}。

metric 类型

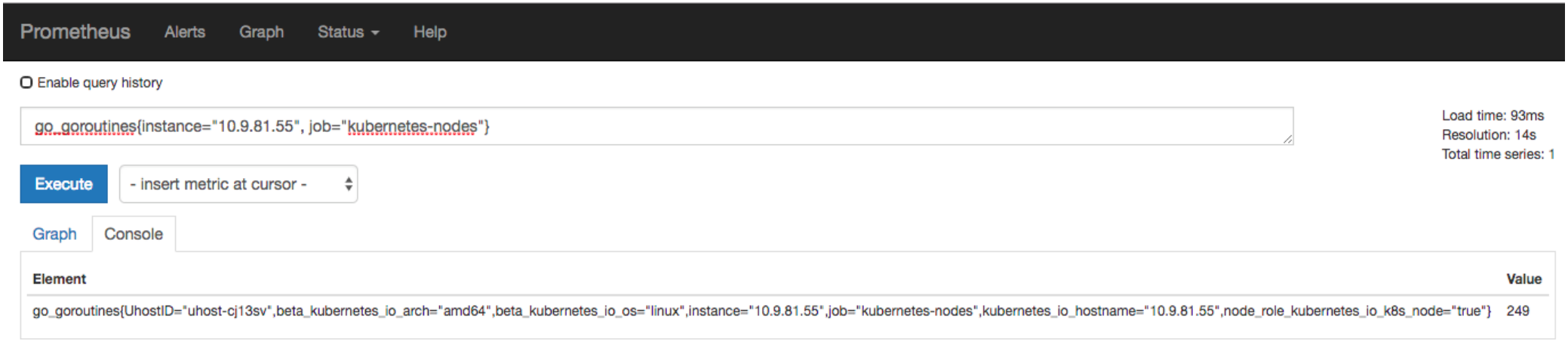
Prometheus 客户端库主要提供四种主要的 metric 类型，分别如下：

1. Counter

一种累加的 metric，典型的应用如：请求的个数，结束的任务数，出现的错误数等等。例如，查询 http_requests_total{method="get", job="kubernetes-nodes", handler="prometheus"} 返回 8，10 秒后，再次查询，则返回 14。

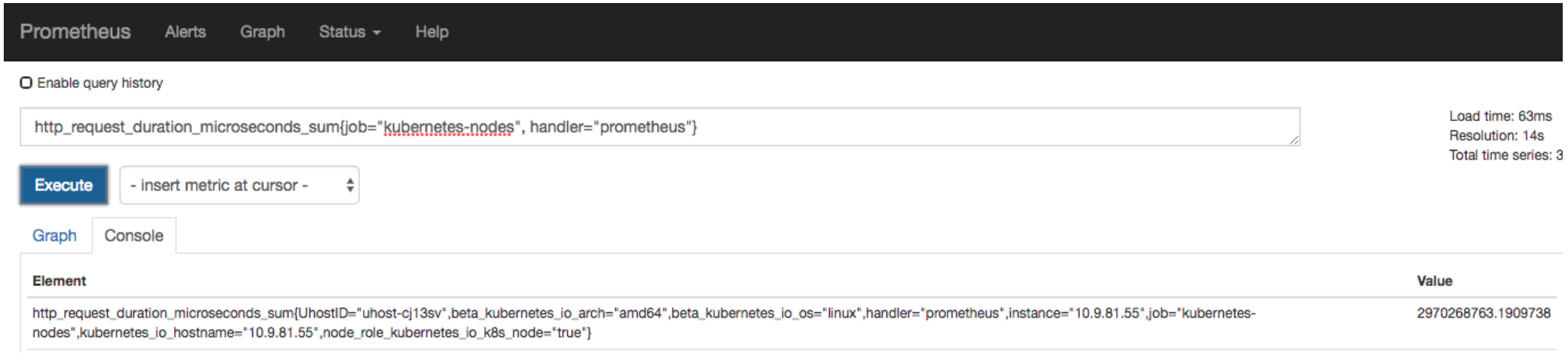
2. Gauge

一种常规的 metric，典型的应用如：温度，运行的 goroutines 的个数。例如：go_goroutines{instance="10.9.81.55", job="kubernetes-nodes"} 返回值 147，10 秒后返回 124。



3. Histogram

可以理解为柱状图，典型的应用如：请求持续时间，响应大小。可以对观察结果采样，分组及统计。例如，查询 http_request_duration_microseconds_sum{job="kubernetes-nodes", handler="prometheus"} 时，返回结果如下：



4. Summary

类似于 Histogram, 典型的应用如：请求持续时间，响应大小。提供观测值的 count 和 sum 功能。提供百分位的功能，即可以按百分比划分跟踪结果。

instance&job

instance: 一个单独 scrape 的目标，一般对应于一个进程。

jobs: 一组同类型的 instances

例如，一个 api-server 的 job 可以包含4个 instances：

- job: api-server
  - instance 1: 1.2.3.4:5670
  - instance 2: 1.2.3.4:5671
  - instance 3: 1.2.3.4:5672

- instance 4: 1.2.3.4:5673

当 scrape 目标时，Prometheus 会自动给这个 scrape 的时间序列附加一些标签以便更好的分别，例如：instance，job。

部署Prometheus

⚠ 该文档仅适用于1.22以下版本集群部署参考；1.22及以上版本集群，请通过控制台-集群详情-监控中心使用，参考：监控中心概述。

前言

对于一套Kubernetes集群而言，需要监控的对象大致可以分为以下几类：

- **Kubernetes系统组件**： Kubernetes内置的系统组件一般有apiserver、controller-manager、etcd、kubelet等，为了保证集群正常运行，我们需要实时知晓其当前的运行状态。
- **底层基础设施**： Node节点(虚拟机或物理机)的资源状态、内核事件等。
- **Kubernetes对象**： 主要是Kubernetes中的工作负载对象，如Deployment、DaemonSet、Pod等。
- **应用指标**： 应用内部需要关心的数据指标，如httpRequest。

部署Prometheus

在Kubernetes中部署Prometheus，除了手工方式外，CoreOS开源了Prometheus-Operator以及kube-Prometheus项目，使得在K8S中安装部署Prometheus变得异常简单。下面我们介绍下如何在UK8S中部署Kube-Prometheus。

1、关于Prometheus-Operator

Prometheus-operator的本职就是一组用户自定义的CRD资源以及Controller的实现，Prometheus Operator这个controller有BRAC权限下去负责监听这些自定义资源的变化，并且根据这些资源的定义自动化的完成如Prometheus Server自身以及配置的自动化管理工作。

在K8S中，监控metrics基本最小单位都是一个Service背后的一组pod，对应Prometheus中的target，所以prometheus-operator抽象了对应的CRD类型" ServiceMonitor "，这个ServiceMonitor通过 sepc.selector.labes来查找对应的Service及其背后的Pod或endpoints，通过sepc.endpoint来指明Metrics的url路径。以下面的CoreDNS举例，需要pull的Target对象Namespace为kube-system，kube-app是他们的labels，port为metrics。

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
labels:
k8s-app: coredns
name: coredns
namespace: monitoring
spec:
endpoints:
- bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
interval: 15s
port: metrics
jobLabel: k8s-app
namespaceSelector:
matchNames:
- kube-system
selector:
```

```
matchLabels:
k8s-app: kube-dns
```

2、准备工作

ssh到任意一台Master节点，克隆kube-prometheus项目。该项目源自CoreOS开源的kube-prometheus，与原始项目相比，主要作为以下优化：

- 将Prometheus和AlbertManager的数据存储介质由emptyDir改为UDisk，提升稳定性，避免数据丢失；
- 将镜像源统一修改为UHub，避免镜像拉取失败的情况出现；
- 新增UK8S专属文件目录，用于配置监控controller-manager、schduler、etcd；
- 将执行文件按目录划分，便于修改及阅读。

```
yum install git -y
git clone --depth=1 -b kube-prometheus https://github.com/ucloud/uk8s-demo.git
```

3、修改UK8S专属文件配置参数

在manifests目录下有UK8S目录，这批配置文件主要用于为UK8S中的controller-manager、schduler、etcd手动创建endpoints和svc，便于Prometheus Server通过ServiceMonitor来采集这三个组件的监控数据。

```
cd /uk8s-demo/manifests/uk8s
# 修改以下两个文件，将其中的IP替换为你自己UK8S Master节点的内网IP
vi controllerManagerAndScheduler_ep.yaml
vi etcd_ep.yaml
```

4、 备注

上面提到要修改controllerManagerAndScheduler_ep.yaml和etcd_ep.yaml这两个文件，这里解释下原因。由于UK8S的ETCD、Scheduler、Controller-Manager都是通过二进制部署的，为了能通过配置"ServiceMonitor"实现Metrics的抓取，我们必须要为其在K8S中创建一个SVC对象，但由于这三个组件都不是Pod，因此我们需要手动为其创建Endpoints。

```
apiVersion: v1
kind: Endpoints
metadata:
labels:
k8s-app: etcd
name: etcd
namespace: kube-system
subsets:
- addresses:
- ip: 10.7.35.44 # 替换成master节点的内网IP
nodeName: etc-master2
ports:
- name: port
port: 2379
protocol: TCP
- addresses:
- ip: 10.7.163.60 # 同上
nodeName: etc-master1
ports:
```

```
- name: port
port: 2379
protocol: TCP
- addresses:
- ip: 10.7.142.140 #同上
nodeName: etc-master3
ports:
- name: port
port: 2379
protocol: TCP
```

5、部署Prometheus Operator

先创建一个名为monitor的NameSpace，Monitor创建成功后，直接部署Operator，Prometheus Operateor以Deployment的方式启动，并会创建前面提到的几个CRD对象。

```
# 创建Namespace
kubectl apply -f 00namespace-namespace.yaml
# 创建Secret，给到Prometheus Server抓取ETCD数据时使用
kubectl -n monitoring create secret generic etcd-certs --from-file=/etc/kubernetes/ssl/ca.pem --from-file=/etc/kubernetes/ssl/etcd.pem --from-file=/etc/kubernetes/ssl/etcd-key.pem
# 创建Operator
kubectl apply -f operator/
# 查看operator启动状态
kubectl get po -n monitoring
# 查看CRD
```

```
kubectl get crd -n monitoring
```

6、部署整套CRD

比较关键的有Prometheus Server、Grafana、AlertManager、ServiceMonitor、Node-Exporter等，这些镜像已全部修改为UHub官方镜像，因此拉取速度相对比较快。

```
kubectl apply -f adapter/
kubectl apply -f alertmanager/
kubectl apply -f node-exporter/
kubectl apply -f kube-state-metrics/
kubectl apply -f grafana/
kubectl apply -f prometheus/
kubectl apply -f serviceMonitor/
kubectl apply -f uk8s/
```

我们可以通过以下命令来查看应用拉取状态。

```
kubectl -n monitoring get po
```

由于默认所有的SVC 类型均为ClusterIP，我们将其改为LoadBalancer，方便演示。

```
kubectl edit svc/prometheus-k8s -n monitoring
# 修改为type: LoadBalancer
[root@10-9-52-233 manifests]# kubectl get svc -n monitoring
# 获取到Prometheus Server的EXTERNAL-IP及端口
```

可以看到，所有K8S组件的监控指标均已获取到。

7、监控应用指标

我们先来部署一组Pod及SVC，该镜像里的主进程会在8080端口上输出metrics信息。

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: example-app
spec:
replicas: 3
selector:
matchLabels:
app: example-app
template:
metadata:
labels:
app: example-app
spec:
containers:
- name: example-app
image: uhub.service.ucloud.cn/uk8s_public/instrumented_app:latest
ports:
- name: web
```

```
containerPort: 8080
---
kind: Service
apiVersion: v1
metadata:
name: example-app
labels:
app: example-app
spec:
selector:
app: example-app
ports:
- name: web
port: 8080
```

再创建一个ServiceMonitor来告诉prometheus server需要监控带有label为app: example-app的svc背后的一组pod的metrics。

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
name: example-app
labels:
team: frontend
spec:
selector:
```

```
matchLabels:
app: example-app
endpoints:
- port: web
```

打开浏览器访问Prometheus Server，进入target发现已经监听起来了，对应的config里也有配置生成和导入。

8、说明

该文档只适用于kubernetes 1.14以上的版本，如果你的kubernetes版本为1.14以下，可以使用release-0.1.

监控中心概述

监控中心是UK8S提供的产品化监控方案，提供基于Prometheus的产品解决方案，涵盖Prometheus集群的全生命周期管理，以及告警规则配置、报警设置等功能，省去了自行搭建监控服务的学习成本及运维成本。

实现原理

监控中心基于CoreOS 开源的Prometheus Operator实现，部署在UK8S集群中，包含三大监控模块，分别是Prometheus、Alertmanager、Grafana，高可用模式下，Prometheus及Alertmanager分别部署2个和3个副本，也支持单节点模式。

同时，为了简化监控服务部署的负担，监控中心启动后，会默认安装NodeExporter以抓取Node节点的监控数据，并添加了Scheduler、Controller Manager、etcd、kubelet等Target，零配置即可实现UK8S的健康状态监控。

功能一览

功能点	功能说明
创建集群	一键创建Prometheus集群
销毁集群	销毁已创建的Prometheus集群
创建告警规则	创建一条告警规则，即Prometheus Rule
删除告警规则	删除Prometheus Rule
添加监控目标	添加监控目标，即Target
删除监控目标	删除监控目标，不再抓取其监控数据

添加接收人	在Alertmanager中配置邮件及微信接受人

开启监控中心

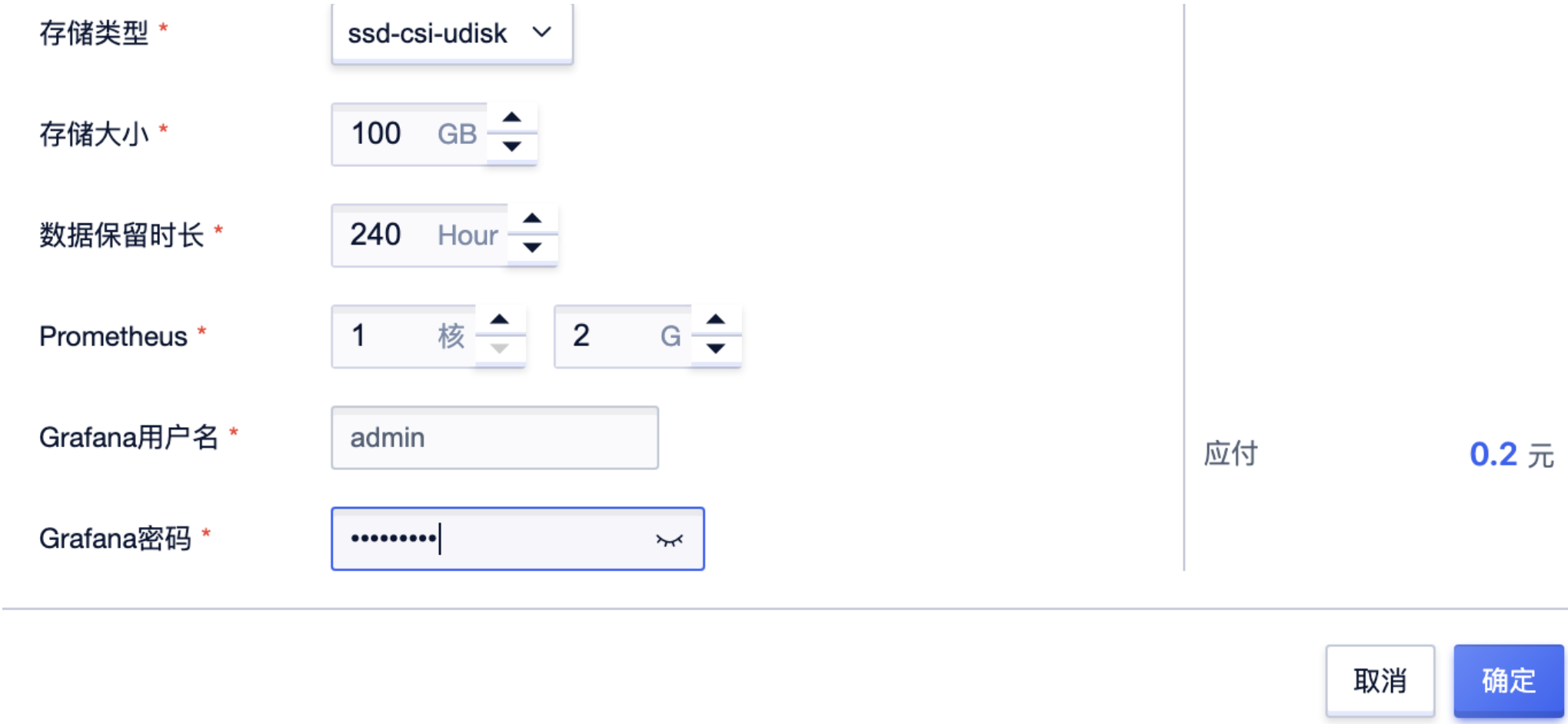
监控中心支持单节点模式和高可用两种模式，需要注意的是，开启监控需要消耗一定的CPU、内存资源，因此，如果开启勾选了高可用模式，请注意：

1. 至少有2个Node节点的可用资源大于Prometheus的容器配置。(建议可用资源大于4C8G)
2. 至少有3个Node节点的可用资源大于Alertmanager的容器配置。(建议可用资源大于1C2G)
3. 由于Prometheus和Alertmanager均需要持久性存储，因此会产生额外的UDisk费用。其中Prometheus为2块100G UDisk，Alertmanager为3块 UDisk。

开启监控

建议参数配置如下：

1. Prometheus 数据盘大小：100G以上，如果集群规模大于100台，建议磁盘大小扩展到500G；
2. Prometheus 数据保留时长：建议240小时；
3. Grafana配置：用户名和密码均可自定义；



监控开启外网

开启外网时通过选择 LB 类型创建 LB 资源或者使用已有资源调用 API 开启外网时，默认使用 CLB7。使用已有资源时，应确保端口没有冲突（监控开启外网会占用80、9090、9093端口）。CLB7 可免费使用，但存在一定限制，推荐使用 ALB，具体可在 CLB文档 和 ALB文档 中查看详情。

添加监控目标

一个监控目标可理解为Prometheus中的一个Target或Job. 原生Prometheus既支持静态配置监控目标，也支持动态服务发现。

由于K8S的Pod被设置非永久性的资源，为了正确地抓取到每个应用对应的Pod监控数据，Prometheus Operator引入了Service Monitor机制，通过监听Service后面的Endpoint（可认为是健康的Pod）来实现监控数据的采集。

因此，为了抓取一组Pod的监控数据，我们必须为这组Pod创建一个对应的Service，并暴露对应的Metrics端口。

! 这里需要强调的是，Service必须暴露Metrics端口，而非业务端口。如我们有一个应用，其应用端口为80，Metrics端口为9200，则供Prometheus抓取数据的Service端口必须是9200，如果设置为80，则不能抓取到任何监控数据。

操作说明

1. 部署应用

在下面这个例子中，我们部署了一个示例应用，该应用为一个web应用程序，其容器对外暴露了两个端口，一个是业务端口80，另一个是Metrics端口8080. 并且创建了一个Service，暴露的端口与容器端口一致。

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: example-app
```

```
spec:
replicas: 2
selector:
matchLabels:
app: example-app
template:
metadata:
labels:
app: example-app
spec:
containers:
- name: example-app
image: uhub.service.ucloud.cn/uk8s_public/instrumented_app:latest
ports:
- name: metrics
containerPort: 8080
- name: web
containerPort: 80
---
kind: Service
apiVersion: v1
metadata:
name: example-app
labels:
app: example-app
```

```
spec:
selector:
app: example-app
ports:
- name: metrics
port: 8080
- name: web
port: 80
```

2、添加监控目标

我们在UK8S的监控中心-->监控目标页面，直接选中该Service，端口名称选择“metrics”，抓取路径一般默认填写“/metrics”，如果监控指标的路径是自定义请咨询业务方。

- 命名空间:kubernetes集群上部署server的namespace名称
- 服务名称:kubenetes集群上要获取metric的service的名称
- 端口名称:获取metric的service上的端口名称
- 路径:获取metric的service访问路径

3、查看监控目标

添加完毕后，我们可以打开Prometheus 控制台，查看该监控目标是否已添加成功。

添加接收人

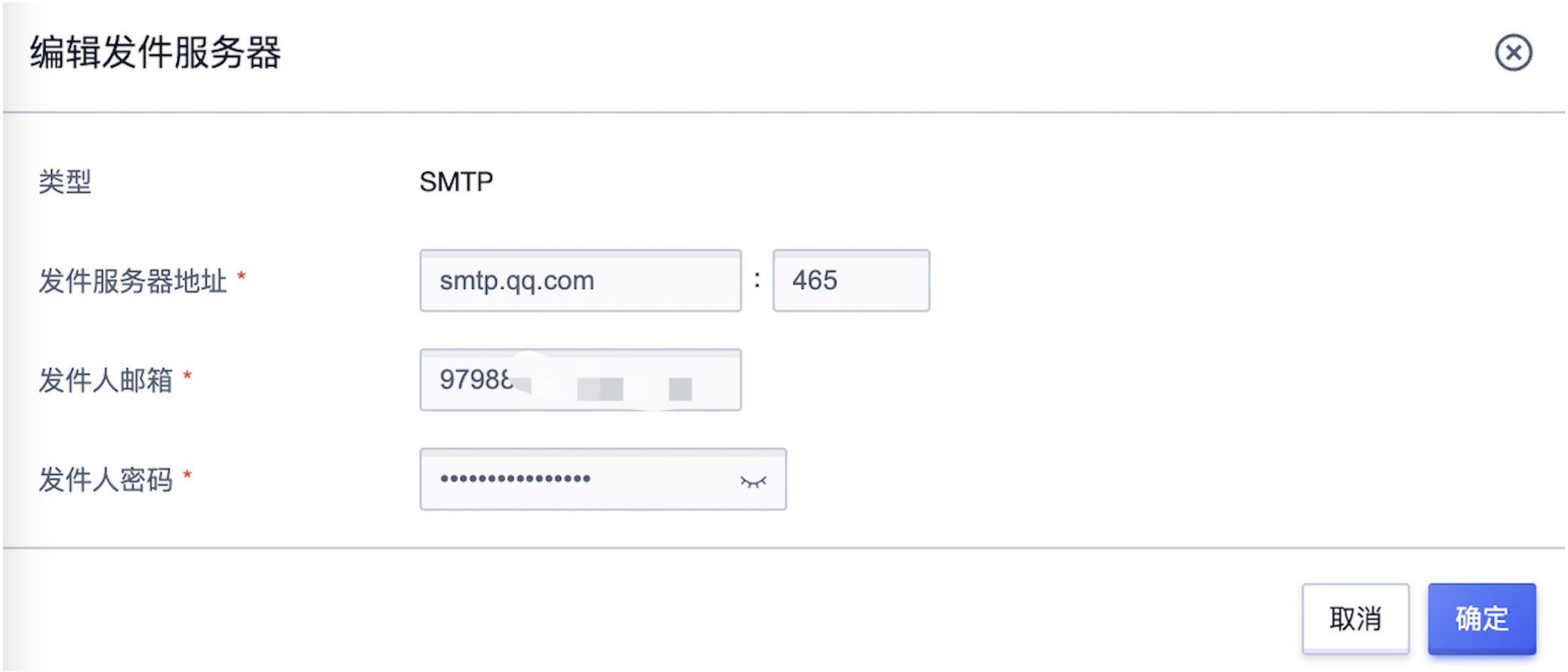
监控中心支持添加接收人邮箱、企业微信及钉钉三种告警形式，请在 UK8S 控制台「监控中心」→「收发设置」进行配置。

1. 配置发件服务器

配置接收人前，需要先设置发件人邮箱信息。目前发送信息是通过邮箱发送。

不同的邮件服务提供商对于发件服务器的配置都有较为详细的说明；这里强调两点：

1. 目前尚不支持TLS，因此请勿填写TLS端口；
2. 密码建议为客户端密码，填写邮箱登录密码可能无法发送邮件。



2. 配置邮件接收人

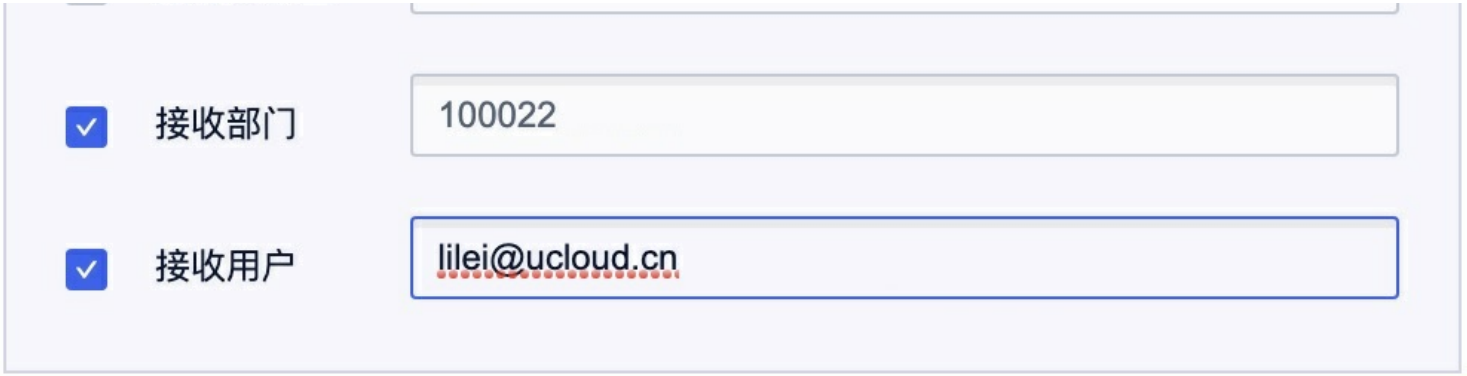
支持添加多个邮件接收人

- 名称：收件人名称
- 邮箱地址：收件的邮箱地址

3. 配置企业微信接收人

注:企业微信机器人类型请参考 '配置webhook接收人(钉钉/企业微信机器人方式)' 方式

在使用微信接收人之前，我们必须在微信管理后台创建一个应用并获取应用ID、企业ID、应用秘钥、部门ID、企业微信用户ID等信息，需要咨询您的企业微信负责人或者管理员方可获取相关信息。详细alertmanager配置的官方文档请参考这里



- 名称：为接收人定义的名称，用于alertmanager的配置
- 企业ID：企业微信唯一ID；管理员登录企业微信web页面，在我的企业->企业信息中查询；（企业微信只有一人时无法查询）
- 应用ID：管理员登录企业微信web页面，在应用管理查看自建应用详情，应用的AgentId就是此应用的ID；如果没有自建应用，可以先新建一个应用；
- 应用密钥：这里同应用ID的查询一致，在应用详情页面有secret选项，就是应用的密钥；
- 接收人：
  - 接收人标签：为接收人创建标签
  - 接收人部门：接收人所在企业部门ID，管理员进入通讯录，在通讯找到接收人所在部门，点击部门旁边三个点，然后查看部门ID
  - 接收用户：接收人用户邮箱；

由于监控中心配置了一条watchdog告警规则，只要企业微信的信息填写正确，一般10分钟以内均可从企业微信中获取到告警信息。

企业微信如果收不到告警信息，还需要设置可信任IP，可参考企业微信官方文档。管理员登录企业微信web页面，在“应用管理”中点击自建的应用进入详情页；找到“企业可信IP”进行配置。填写的IP为访问企业微信的IP（一般为集群关联的NAT网关的外网弹性IP）。

4. 配置webhook接收人(钉钉/企业微信机器人方式)

4.1 创建钉钉/企业微信/飞书机器人，获取 Webhook 地址

在使用webhook接收人(钉钉/企业微信/飞书机器人方式)之前，我们必须在钉钉/企业微信/飞书管理后台创建自定义机器人，并获取其 Webhook 地址，详情参考钉钉/企业微信/飞书相关文档。

4.2 部署配置文件

AlertManager 不支持直接接入钉钉/企业微信/飞书告警，需要进行适配转换，以下是参考社区的示例部署 yaml 文件。

请根据yaml中的提示,结合自身场景来替换yaml中的webhook地址以及image

```
apiVersion: v1
kind: ConfigMap
metadata:
name: alertmanager-webhook
namespace: uk8s-monitor
data:
config.yaml: |-
targets:
webhook1:
# 请替换为您的钉钉/企业微信/飞书机器人 Webhook 地址
url: https://oapi.dingtalk.com/robot/send?access_token=xxxxxx
---
apiVersion: v1
kind: Service
metadata:
name: alertmanager-webhook
namespace: uk8s-monitor
spec:
selector:
k8s-app: webhook
ports:
```

```
- name: http
port: 80
targetPort: 8060
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: alertmanager-webhook
namespace: uk8s-monitor
spec:
replicas: 2
selector:
matchLabels:
k8s-app: webhook
template:
metadata:
labels:
k8s-app: webhook
spec:
volumes:
- name: config
configMap:
name: alertmanager-webhook
containers:
- name: alertmanager-webhook
```

```
# 替换image值:
# 企业微信机器人: uhub.service.ucloud.cn/uk8s/prometheus-webhook-wechat:v2.0.1
# 钉钉机器人: uhub.service.ucloud.cn/uk8s/prometheus-webhook-dingtalk:v2.0.0
# 飞书机器人: uhub.service.ucloud.cn/uk8s/prometheus-webhook-feishu:v2.0.0
image: xxx:xxx
args:
- --web.listen-address=:8060
- --config.file=/config/config.yaml
volumeMounts:
- name: config
mountPath: /config
resources:
limits:
cpu: 100m
memory: 100Mi
ports:
- name: http
containerPort: 8060
```

4.3 添加接收人

在控制台「收发设置」页面「接收人」版面点击「添加」，根据类型在 Webhook 地址栏中填写

钉钉机器人: http://alertmanager-webhook.uk8s-monitor.svc/dingtalk/webhook1/send

企业微信机器人: http://alertmanager-webhook.uk8s-monitor.svc/wechat/webhook/send

飞书机器人: http://alertmanager-webhook.uk8s-monitor.svc/feishu/webhook1/send

5. 配置webhook接收人(微信公众号方式)

5.1 创建公众号以及模板，获取公众号的 appid、secret 以及模板的 templateid

该告警方式基于微信公众号的模板消息实现,使用时请遵守微信公众号模板消息运营规范

模板消息内容可参考如下(模板名称随意)

```
告警状态: {{ status.DATA }}
告警类型: {{ alertname.DATA }}
告警级别: {{ severity.DATA }}
告警实例: {{ instance.DATA }}
告警内容: {{ message.DATA }}
告警时间: {{ startsat.DATA }}
```

5.2 部署配置文件

AlertManager 同样不支持直接接入微信公众号告警,需要进行适配转换

> 请根据yaml中的提示,结合自身场景来替换yaml中的配置

```
apiVersion: v1
kind: ConfigMap
metadata:
name: alertmanager-webhook
namespace: uk8s-monitor
data:
config.yaml: |
# 请替换为您的微信公众号的appid
appid: "xxx"
# 请替换为您的微信公众号的secret
secret: "xxx"
# 请替换为您的微信公众号模板的templateid
templateid: "xxx"
# 告警组 name为组名,chatids下为该告警组下的用户的openid
chatgroups:
- name: uk8s
chatids:
- "openid1"
- name: all
chatids:
- "openid1"
- "openid2"
---
apiVersion: v1
kind: Service
```

```
metadata:
name: alertmanager-webhook
namespace: uk8s-monitor
spec:
selector:
k8s-app: webhook
ports:
- name: http
port: 80
targetPort: 8060
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: alertmanager-webhook
namespace: uk8s-monitor
spec:
replicas: 1
selector:
matchLabels:
k8s-app: webhook
template:
metadata:
labels:
k8s-app: webhook
```

```
spec:
volumes:
- name: config
configMap:
name: alertmanager-webhook
containers:
- name: alertmanager-webhook
image: uhub.service.ucloud.cn/uk8s/prometheus-webhook-wechat-public:v2.0.0
args:
- --web.listen-address=:8060
- --config.file=/config/config.yaml
volumeMounts:
- name: config
mountPath: /config
resources:
limits:
cpu: 100m
memory: 100Mi
ports:
- name: http
containerPort: 8060
```

5.3 添加接收人

在控制台「收发设置」页面「接收人」版面点击「添加」，在 Webhook 地址栏中填写

http://alertmanager-webhook.uk8s-monitor.svc/wechat/{groupname}/send

其中{groupname}为config.yaml中对应的chatgroups.name的名称

监控相关常见问题

1. 无法采集kube-scheduler的监控信息

监控组件无法采集到kube-scheduler的监控信息，原因可能是prometheus配置了通过https端口(10259)去采集/metrics，但是kube-scheduler未支持通过https端口暴露/metrics接口。

修复方式：依次登录到3台master节点上，修改 /usr/lib/systemd/system/kube-scheduler.service 文件，加上两行参数：--authentication-kubeconfig 和 --authorization-kubeconfig。

```
[Service]
EnvironmentFile=-/etc/kubernetes/config
ExecStart=/usr/local/bin/kube-scheduler \
$KUBE_LOGTOSTDERR \
$KUBE_LOG_LEVEL \
--config=/etc/kubernetes/kube-scheduler.conf \
--authentication-kubeconfig=/etc/kubernetes/kubelet.kubeconfig \
--authorization-kubeconfig=/etc/kubernetes/kubelet.kubeconfig
```

然后执行systemctl restart kube-scheduler。

2. 监控node-exporter服务OOM如何调整

使用命令查看node-exporter重启原因；确定字段status.containerStatuses.lastState.terminated.reason是否是OOM，如果是OOM，需要调整资源。

```
## 其中<pod-name> 换成自己查询到的pod名称；例如：uk8s-monitor-prometheus-node-exporter-sbncp
$ kubectl -n uk8s-monitor get po <pod-name> -o yaml
```

调整node-exportor资源情况；命令中的cpu、memory可以修改成自己想要的资源大小；

```
$ kubectl -n uk8s-monitor set resources daemonset uk8s-monitor-prometheus-node-exporter --limits=cpu=500m,memory=1Gi --
requests=cpu=250m,memory=512Mi
```

验证是否修改成功，查看daemonset的资源是否被设置为修改后数据。

```
$ kubectl -n uk8s-monitor get daemonset uk8s-monitor-prometheus-node-exporter -o yaml
```

使用ELK自建UK8S日志解决方案

下面我们介绍下如何使用Elasticsearch+Filebeat+Kibana来搭建UK8S日志解决方案。

一、部署Elasticsearch

1. 关于Elasticsearch

Elasticsearch（ES）是一个基于Lucene构建的开源、分布式、RESTful接口的全文搜索引擎。Elasticsearch还是一个分布式文档数据库，其中每个字段均可被索引，而且每个字段的数据均可被搜索，ES能够横向扩展至数以百计的服务器存储以及处理PB级的数据。可以在极短的时间内存储、搜索和分析大量的数据。通常作为具有复杂搜索场景情况下的核心发动机

2. 环境要求

Elasticsearch运行时要求vm.max_map_count内核参数必须大于262144，因此开始之前需要确保这个参数正常调整过。

```
sysctl -w vm.max_map_count=262144
```

也可以在ES的的编排文件中增加一个initContainer来修改内核参数，但这要求kublet启动的时候必须添加了--allow-privileged参数，uk8s默认开启了该参数，在后面的示例中采用initContainer的方式。

3. ES节点角色

ES的节点Node可以分为几种角色：

Master-eligible node，是指有资格被选为Master节点的Node，可以统称为Master节点。设置node.master: true

Data node，存储数据的节点，设置方式为node.data: true。

Ingest node，进行数据处理的节点，设置方式为node.ingest: true。

Trible node，为了做集群整合用的。

对于单节点的Node，默认是master-eligible和data，对于多节点的集群，需要根据需求仔细规划每个节点的角色。

4. Elasticsearch部署

为了方便演示，我们把本文所有的对象资源都放置在一个名为 elk 的 namespace 下面，所以我们需要添加创建一个 namespace：

```
kubectl create namespace elk
```

不区分节点角色

这种模式下，集群中的节点不做角色的区分，配置文件请参考elk-cluster.yaml

```
bash-4.4# kubectl apply -f elk-cluster.yaml
deployment.apps/kb-single created
service/kb-single-svc created
statefulset.apps/es-cluster created
service/es-cluster-nodeport created
service/es-cluster created
bash-4.4# kubectl get po -n elk
NAME READY STATUS RESTARTS AGE
es-cluster-0 1/1 Running 0 2m18s
es-cluster-1 1/1 Running 0 2m15s
es-cluster-2 1/1 Running 0 2m12s
kb-single-69ddfc96f5-lr97q 1/1 Running 0 2m18s
bash-4.4# kubectl get svc -n elk
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
es-cluster ClusterIP None <none> 9200/TCP,9300/TCP 2m20s
es-cluster-nodeport NodePort 172.17.177.40 <none> 9200:31200/TCP,9300:31300/TCP 2m20s
kb-single-svc LoadBalancer 172.17.129.82 117.50.40.48 5601:38620/TCP 2m20s
bash-4.4#
```

通过kb-single-svc的EXTERNAL-IP，便可以访问Kibana。

区分节点角色

如果需要区分节点的角色，就需要建立两个StatefulSet部署，一个是Master集群，一个是Data集群。Data集群的存储示例中简单使用了emptyDir，可以根据需要使用localStorage或者hostPath，关于存储的介绍，可以参考Kubernetes官网。这样就可以避免Data节点在本机重启时发生数据丢失而重建索引，但是如果发生迁移的话，如果想保留数据，只能采用共享存储的方案了。具体的编排文件在这里elk-role-cluster.yaml

```
bash-4.4# kubectl apply -f elk-role-cluster.yaml
deployment.apps/kb-single created
service/kb-single-svc created
statefulset.apps/es-cluster created
statefulset.apps/es-cluster-data created
service/es-cluster-nodeport created
service/es-cluster created
bash-4.4# kubectl get po -n elk
NAME READY STATUS RESTARTS AGE
es-cluster-0 1/1 Running 0 53s
es-cluster-1 1/1 Running 0 50s
es-cluster-2 1/1 Running 0 47s
es-cluster-data-0 1/1 Running 0 53s
es-cluster-data-1 1/1 Running 0 50s
es-cluster-data-2 1/1 Running 0 47s
kb-single-69ddfc96f5-lxsn8 1/1 Running 0 53s
bash-4.4# kubectl get statefulset -n elk
NAME READY AGE
```

```
es-cluster 3/3 2m
es-cluster-data 3/3 2m
bash-4.4# kubectl get svc -n elk
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
es-cluster ClusterIP None <none> 9200/TCP,9300/TCP 44s
es-cluster-nodeport NodePort 172.17.63.138 <none> 9200:31200/TCP,9300:31300/TCP 44s
kb-single-svc LoadBalancer 172.17.183.59 117.50.92.74 5601:32782/TCP
```

二、部署FileBeat

在进行日志收集的过程中，我们首先想到的是使用Logstash，因为它是ELK stack中的重要成员，但是在测试过程中发现，Logstash是基于JDK的，在没有产生日志的情况单纯启动Logstash就大概要消耗500M内存，在每个Pod中都启动一个日志收集组件的情况下，使用logstash有点浪费系统资源，因此我们更推荐一个轻量级的日志采集工具Filebeat，经测试单独启动Filebeat容器大约只会消耗12M内存。具体的编排文件可以参考filebeat.yaml，本例采用DaemonSet的方式编排。

```
bash-4.4# kubectl apply -f filebeat.yaml
configmap/filebeat-config created
daemonset.extensions/filebeat created
clusterrolebinding.rbac.authorization.k8s.io/filebeat created
clusterrole.rbac.authorization.k8s.io/filebeat created
serviceaccount/filebeat created
```

编排文件中将filebeat使用到的配置ConfigMap挂载到/home/uk8s-filebeat/filebeat.yaml，实际启动filebeat时使用该自定义配置。有关filebeat的配置可以参见 Configuring Filebeat中相应的说明。

Filebeat命令行参数可以参考 Filebeat Command Reference，本例中使用到的参数说明如下：

- -c, --c FILE

指定Filebeat使用的配置文件，如果不指定则使用默认的配置文件/usr/share/filebeat/filebeat.yaml

- -d, --d SELECTORS

为指定的selectors打开调试模式，selectors是以逗号分隔的列表，-d "*" 表示对所有组件进行调试。在实际生产环境中请关闭该选项，初次配置时打开可以有效排错。

- -e, --e

指定日志输出到标准错误输出，关闭默认的syslog/file输出

三、部署Logstash（可选）

由于Filebeat对message的过滤功能有限，在实际生产环境中通常会结合logstash。这种架构中Filebeat作为日志收集器，将数据发送到Logstash，经过Logstash解析、过滤后，将其发送到Elasticsearch存储，并由Kibana呈献给用户。

1、创建配置文件

创建Logstash的配置文件，可以参考elk-log.conf，更详细的配置信息见Configuring Logstash。大部分Logstash配置文件都可以分为3部分：input, filter 和 output，示例配置文件中指定Logstash从Filebeat获取数据，并输出到Elasticsearch。

2、根据配置文件创建一个名为elk-pipeline-config的ConfigMap，如下：

```
bash-4.4# kubectl create configmap elk-pipeline-config --from-file=elk-log.conf --namespace=elk
configmap/elk-pipeline created
bash-4.4# kubectl get configmap -n elk
NAME DATA AGE
elk-pipeline-config 1 9s
filebeat-config 1 21m
```

3、在K8S集群部署logstash。

编写 logstash.yaml ，在yaml文件中挂载之前创建的ConfigMap。需要注意的是，此处使用了logstash-oss镜像，关于oss和non-oss版本的区别请参考链接。

```
bash-4.4# kubectl apply -f logstash.yaml
deployment.extensions/elk-log-pipeline created
service/elk-log-pipeline created
bash-4.4# kubectl get po -n elk
NAME READY STATUS RESTARTS AGE
elk-log-pipeline-55d64bbcf4-9v49w 1/1 Running 0 50m
```

4、查看logstash是否正常工作，出现如下内容，则表明logstash正常工作

```
bash-4.4# kubectl logs -f elk-log-pipeline-55d64bbcf4-9v49w -n elk
[2019-03-19T08:56:03,631][INFO ][logstash.agent ] Successfully started Logstash API endpoint {:port=>9600}
```

```
...
[2019-03-19T08:56:09,845][INFO ][logstash.inputs.beats ] Beats inputs: Starting input listener {:address=>"0.0.0.0:5044"}
[2019-03-19T08:56:09,934][INFO ][logstash.pipeline ] Pipeline started succesfully {:pipeline_id=>"main", :thread=>"#<Thread:0x77d5c9b5 run>"}
[2019-03-19T08:56:10,034][INFO ][org.logstash.beats.Server] Starting server on port: 5044
```

5、修改 filebeat.yaml的output参数，将其输出指向Logstash

```
items:
- apiVersion: v1
kind: ConfigMap
metadata:
...
data:
filebeat.yml: |
...
output.logstash:
hosts: ["elk-log-pipeline:5044"]
...
```

四、收集应用日志

前面我们已经部署好了Filebeat用于采集应用日志，并将采集到的日志输出到Elasticsearch，下面我们以一个nginx应用为例，来测试日志能否正常采集、索引、展示。

1、部署nginx应用

创建一个Nginx的部署和LoadBalancer服务，这样可以通过eip访问Nginx。配置文件请参考nginx.yaml，我们将Nginx访问日志的输出路径以hostPath的形式挂载到宿主的/var/log/nginx/路径下。

```
bash-4.4# kubectl apply -f nginx.yaml
deployment.apps/nginx-deployment unchanged
service/nginx-cluster configured
bash-4.4# kubectl get svc -n elk nginx-cluster
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
nginx-cluster LoadBalancer 172.17.153.144 117.50.25.74 5680:48227/TCP 19m
bash-4.4# kubectl get po -n elk -l app=nginx
NAME READY STATUS RESTARTS AGE
nginx-deployment-6c858858d5-7tcbx 1/1 Running 0 36m
nginx-deployment-6c858858d5-9xzh8 1/1 Running 0 36m
```

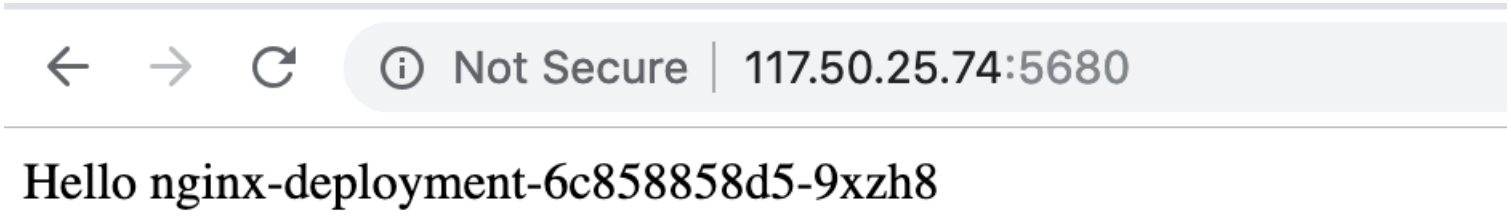
2、Filebeat配置

在之前部署Filebeat时，由于我们已经将/var/log/nginx/加入到inputs.paths中，Filebeat已经可以对nginx的日志实现监控采集。

```
filebeat.modules:
- module: system
filebeat.inputs:
- type: log
paths:
- /var/log/containers/*.log
- /var/log/messages
```

```
- /var/log/nginx/*.log
- /var/log/*
symlinks: true
include_lines: ['hyperkube']
output.logstash:
hosts: ["elk-log-pipeline:5044"]
logging.level: info
index: filebeat-
```

3、通过公网访问nginx服务，产生访问日志



4、通过Kibana验证日志的采集情况

Console Search Profiler Grok Debugger

```
GET _search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_phrase": {
            "source": "/var/log/nginx"
          }
        }
      ]
    }

  }
}
```

```
{
  "took": 3759,
  "timed_out": false,
  "_shards": {
    "total": 11,
    "successful": 11,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 5,
    "max_score": 10.498557,
    "hits": [
      {
        "_index": "filebeat-6.6.0-2019.03.21",
        "_type": "doc",
        "_id": "SsYunmkBbpK7JhSBS0l3",
```

使用UK8S日志插件功能

UK8S 推出了一个新的插件功能，旨在帮助用户快速部署 UK8S 集群所需要的相关插件。

支持UK8S版本：1.14.6 及以后（2019年9月17日之后创建）

ELK 插件安装需要挂载 UDisk，UK8S 支持挂载 UDisk 的地域请查看链接

1. ELK日志插件

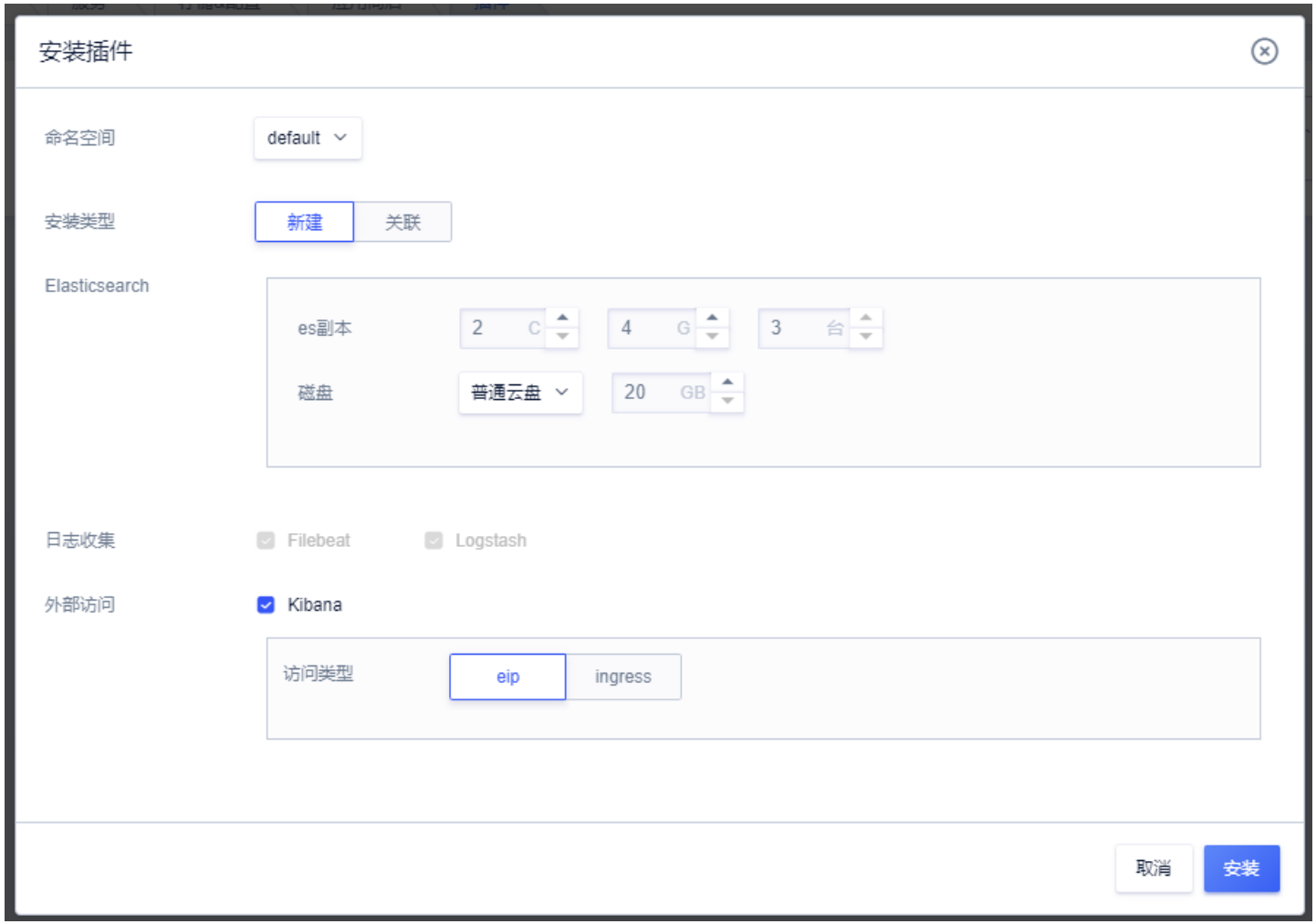
插件-日志ELK是UK8S基于ELK打造的一站式日志服务，包括Elasticsearch（可选）、Logstash、Filebeat 和 Kibana（可选）多个组件，实现了集群内日志的自动采集、过滤和存储，并内置了日志检索功能。

关于 ELK 组件介绍请参考ELK日志解决方案

用户进入集群后可以选择为集群安装完整的一套ELK组件或者安装Filebeat等组件关联用户所使用的UES，通过UK8S插件部署的完整ELK组件可以在UCloud页面进行统一化日志查询展示。

2. 安装完整ELK日志组件

Step 1. 用户在插件页中，点击安装插件进行选择完整安装ELK，在安装过程中用户可以对以下选项进行选择。



可选项：

- Elasticsearch

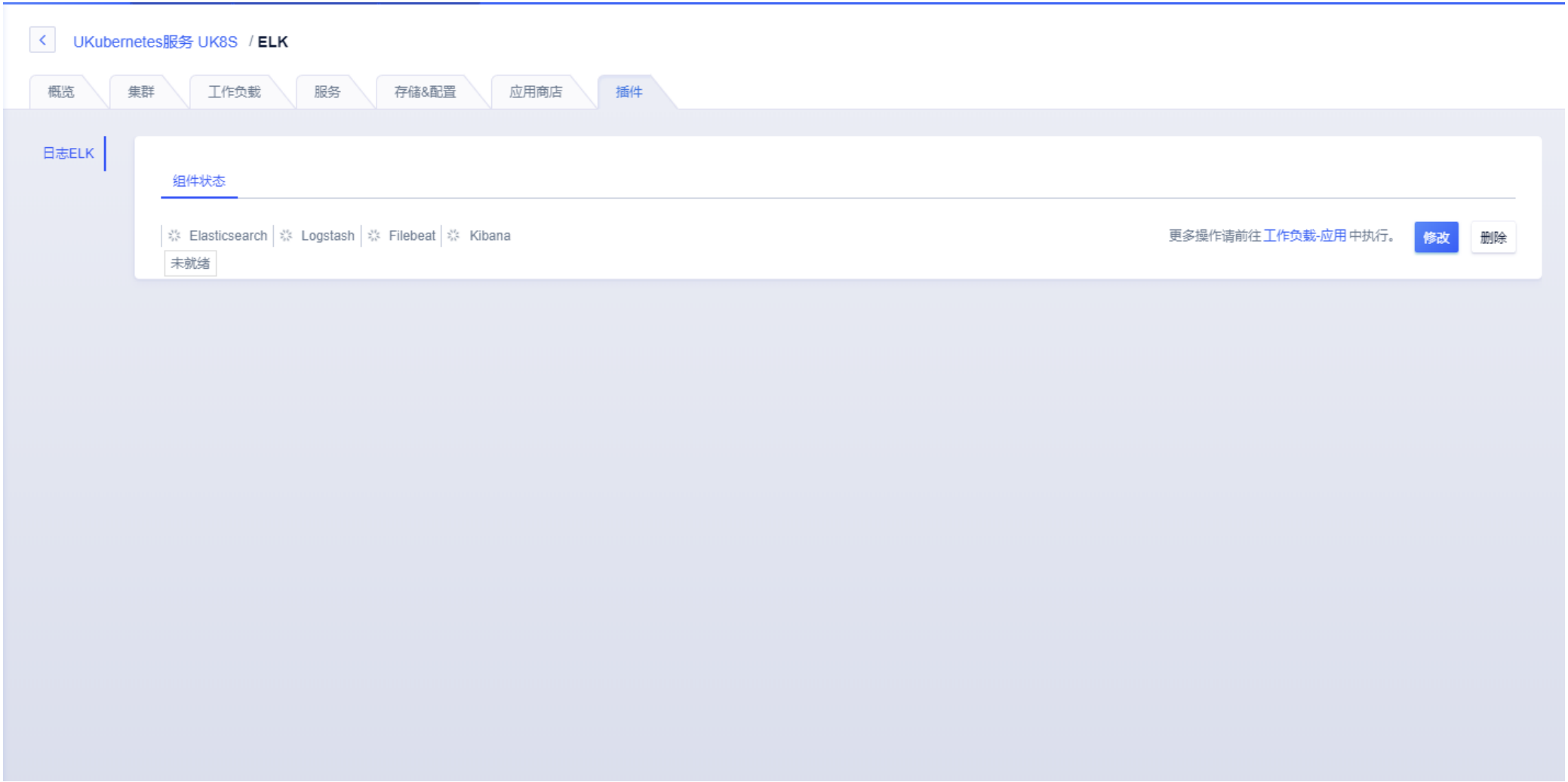
这里 Elasticsearch 我们选择新建，将在 UK8S 集群中进行 Elasticsearch 的部署，选择关联则是进行外挂ES（UES）进行关联，这里我们默认进行集群内安装。

- Kibana

用户可以根据自己的需要进行安装 Kibana 的安装，安装后将可以通过应用中查看对外暴露的 Kibana 进行日志查看，提供了 EIP 和 Ingress 两种方式对外暴露，如集群内部没有安装 Ingress Controller 服务的话 Ingress 方式将无法正常提供外部访问。用户可以不进行安装在 UCloud 控制台进行日志查看，这里我们默认选择安装。

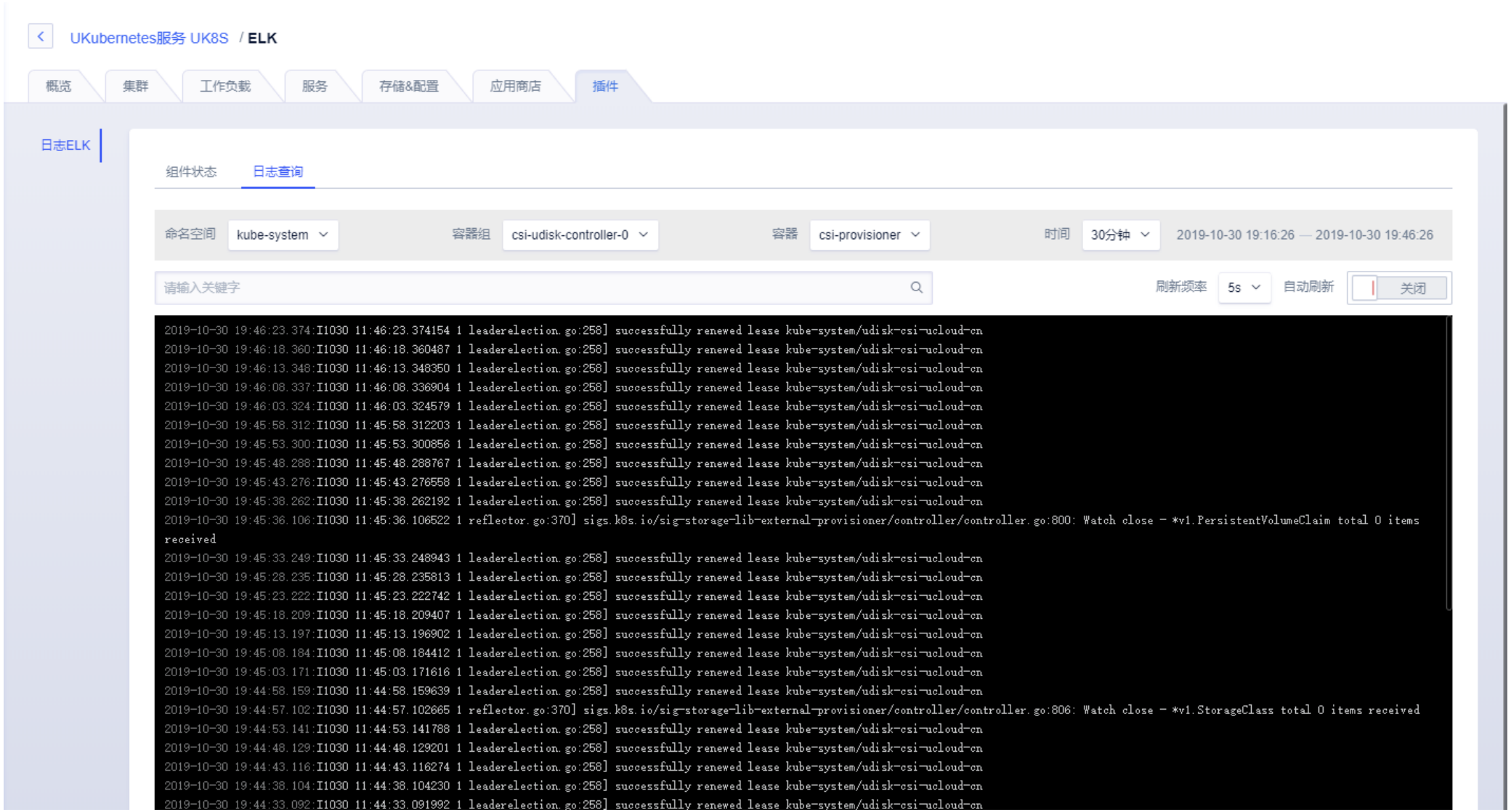
注意：请确保集群中 Node 节点大于 3 台，单台 Node 节点空余资源大于 4C8G，确保足够资源进行安装ELK服务。

Step 2. 点击安装，进入安装页面，可以动过鼠标悬停查看安装状态，或者进入工作负载-应用中查看安装状态，安装可能将持续10分钟左右。



Step 3. 安装完成后可以在控制台看到日志仪表盘，查看组件健康状态、日志信息统计。

Step 4. 点击日志查询tab页可以进行日志查询，提供了具体的容器选择、时间区间和关键字等检索方式，方便用户进行日志查询。



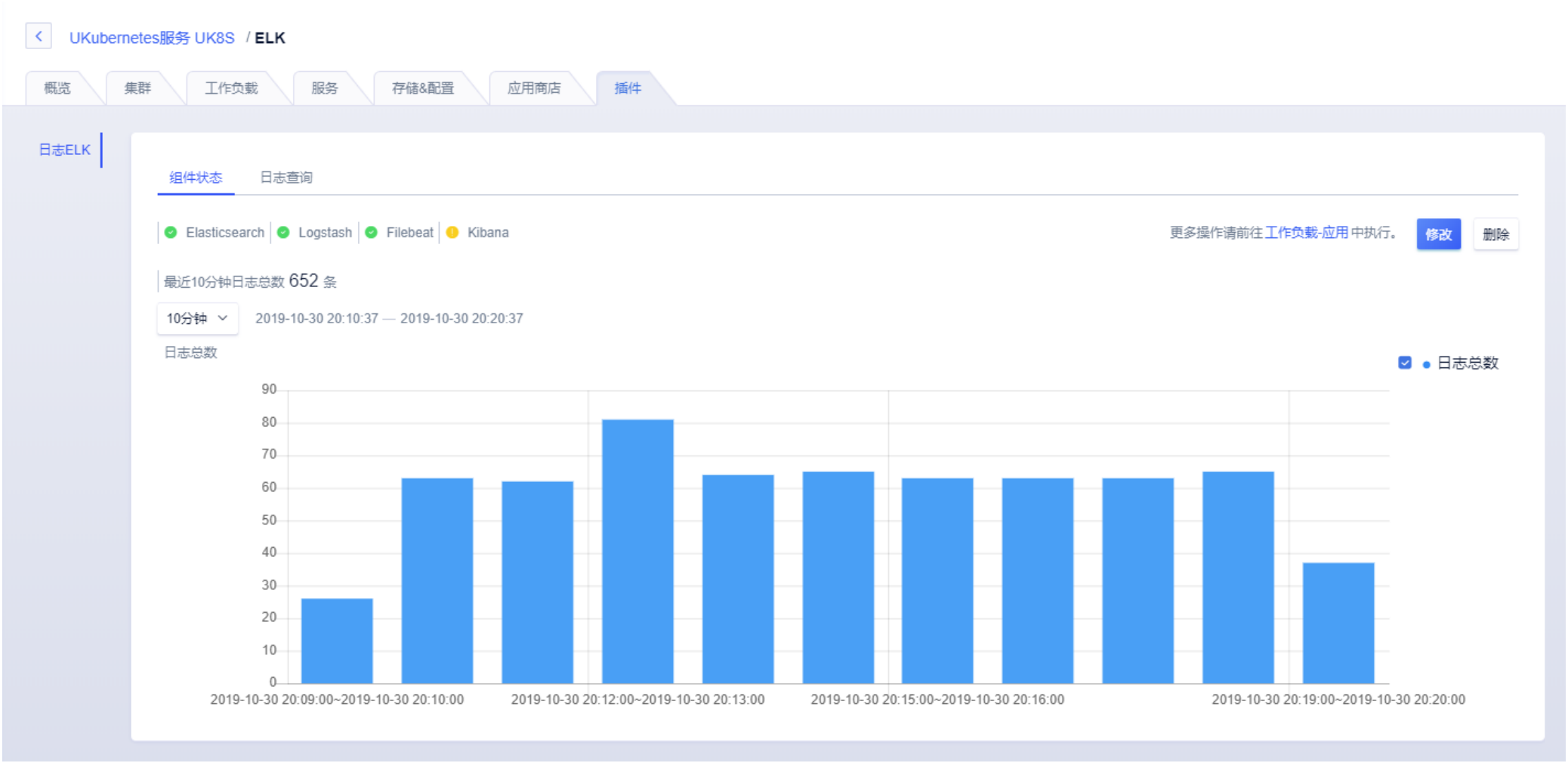
3. 关联UES安装

1. UES是UCloud提供的高可用Elasticsearch集群，使用UK8S关联UES前请创建UES服务，详情请参考UES操作文档，这里我们建立好了一个UES集群，在UES集群详情中的节点信息，选择任意一节点，记录节点IP地址。

2. 用户在插件页中，点击安装插件进行选择关联UES安装ELK，在安装过程中用户可以对以下选项进行选择。

注意:UES需要与UK8S集群属于同一个子网内,用户安装时需要选择具体子网,填写第一步选择节点的IP地址加9200端口号。

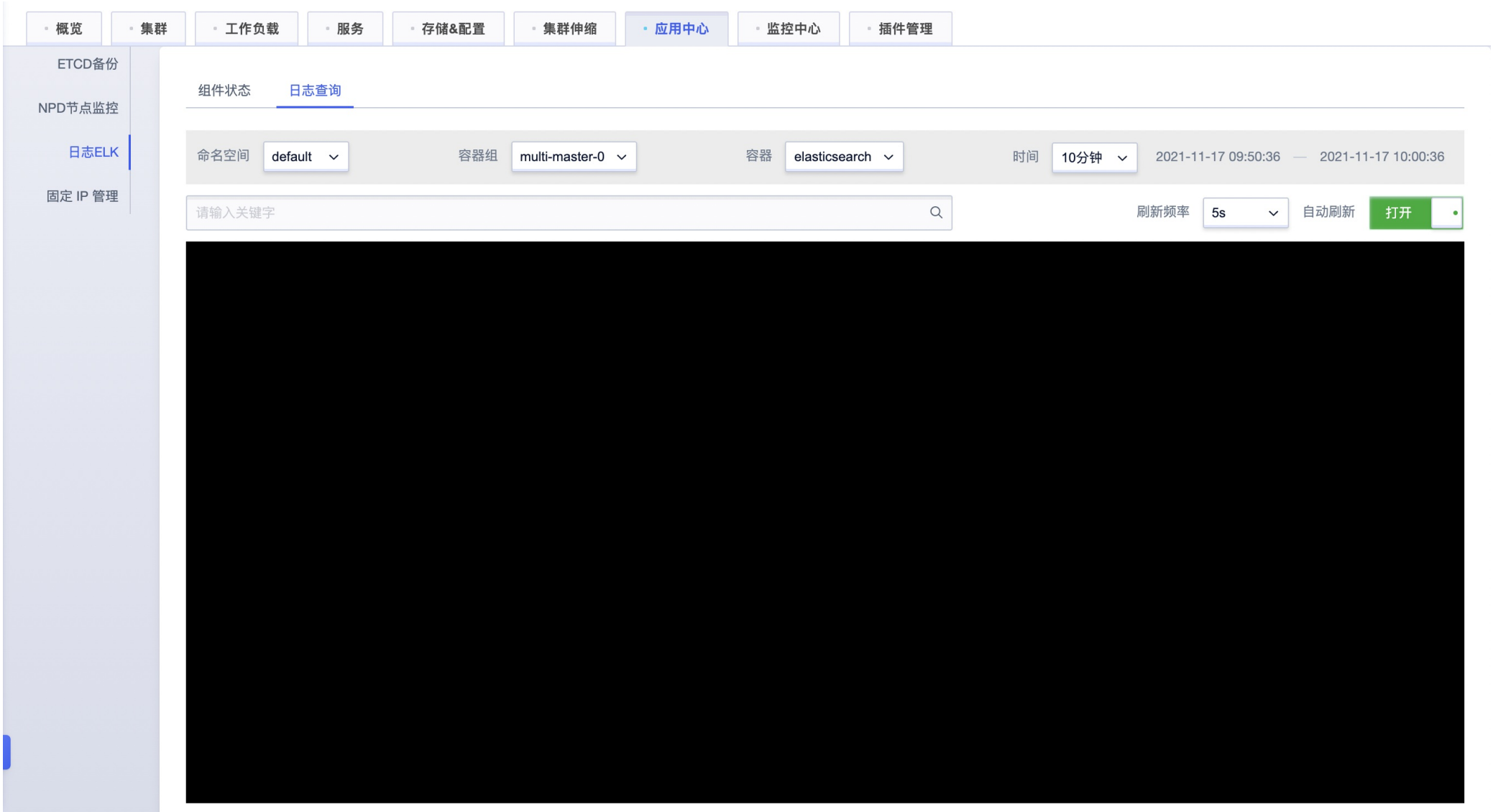
3. 安装完成后可以在UK8S中进行集群内部日志查询,UES集群其他日志查询请至UES集群提供的Kibana进行查询。



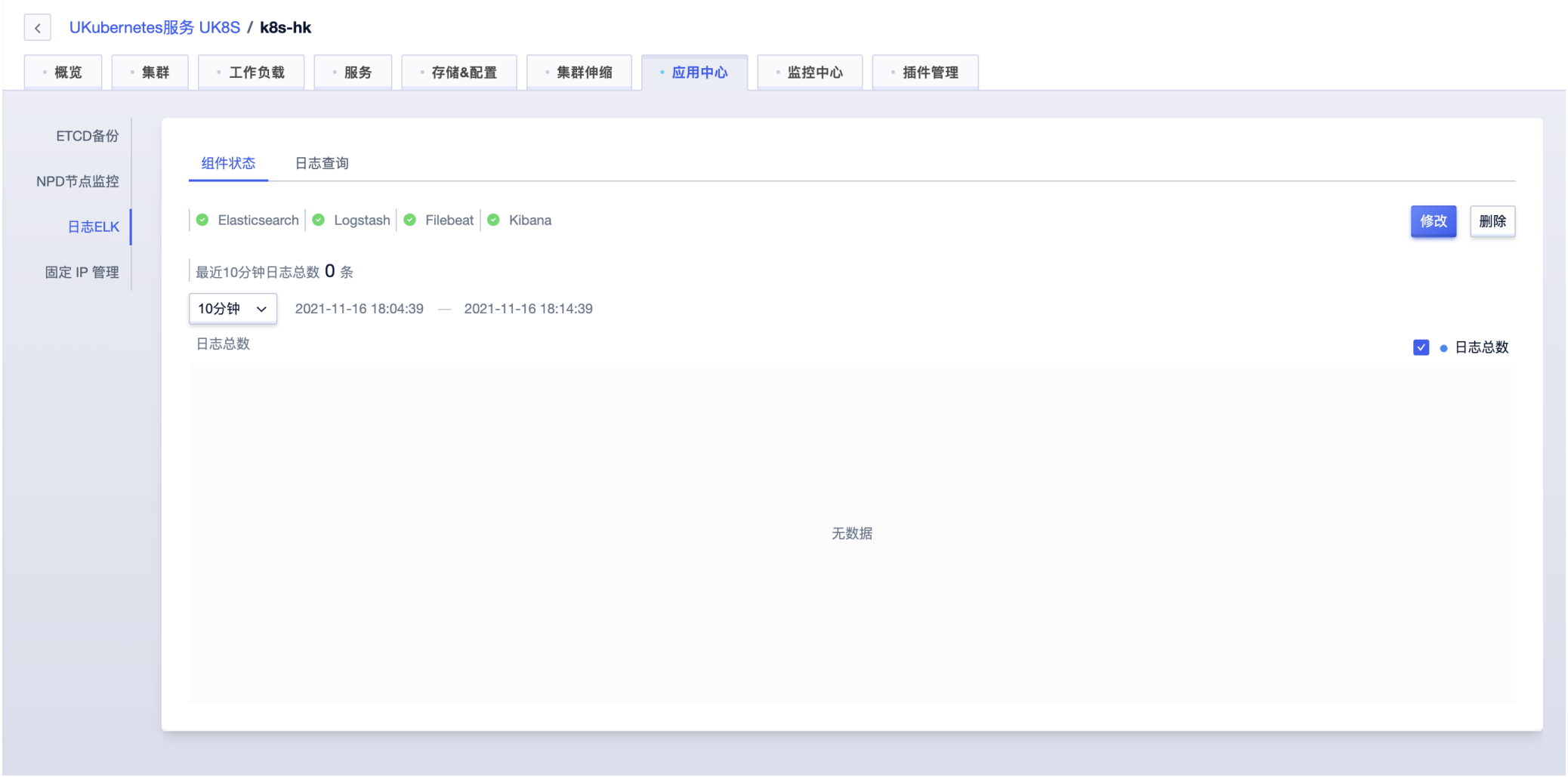
1. 故障现象

在 UK8S 服务控制台，集群应用中心，日志 ELK 页面,开启集群日志插件,使用一段时间后，遇到故障现象如下:

1. 日志 ELK,日志查询页面无最新日志



2. 日志ELK,组件状态页面显示最近10分钟日志总数 0 条



2. 故障排查参考

日志 ELK 默认部署在集群 default 命名空间,如果部署在自定义命名空间，执行命令请替换 default 名称

step 1. 查看 logstash 组件日志, 登录集群master节点，执行命令: `kubectl logs -f uk8s-elk-release-logstash-0 -n default` 可见不断打印如下信息:

```
[2021-11-16T09:55:31,753][INFO ][logstash.outputs.elasticsearch] retrying failed action with response code: 403 ({"type"=>"cluster_block_exception", "reason"=>"index [uk8s-vidxqjoo-kube-system-2021.11.16] blocked by: [FORBIDDEN/12/index read-only / allow delete (api)];"})
[2021-11-16T09:55:31,753][INFO ][logstash.outputs.elasticsearch] Retrying individual bulk actions that failed or were rejected by the previous bulk request. {:count=>1}
```

step 2. 查看ES组件存储卷使用率,登录集群master节点，执行命令:

```
for pod in multi-master-0 multi-master-1 multi-master-2
do
kubectl exec -t -i $pod -- sh -c 'df -h| grep /usr/share/elasticsearch/data' -n default
done
```

可以看到空间磁盘使用率高达 96%

```
/dev/vdb 20G 19G 933M 96% /usr/share/elasticsearch/data
/dev/vdb 20G 19G 939M 96% /usr/share/elasticsearch/data
/dev/vdc 20G 19G 933M 96% /usr/share/elasticsearch/data
```

step 3. 通过ES API 查询索引状态，登录集群 master 节点，执行命令:

```
ES_CLUSTER_IP=`kubectl get svc multi-master | awk 'NR>1 {print $3}'`
curl http://${ES_CLUSTER_IP}:9200/_all/_settings?pretty
```

可以看到返回信息中包含 "read_only_allow_delete": "true" 从这里可以定位故障原因，虽然磁盘没有写满，但是触发了ES的保护机制：

- ES cluster.routing.allocation.disk.watermark.low，控制磁盘使用的低水位线（watermark）默认值85%，超过后，es不会再为该节点分配分片；
- ES cluster.routing.allocation.disk.watermark.high，控制高水位线，默认值90%，超过后，将尝试将分片重定位到其他节点；
- ES cluster.routing.allocation.disk.watermark.flood_stage 控制洪泛水位线。默认值95%，超过后，ES集群将强制将所有索引都标记为只读，导致新增日志无法采集，无法查询最新日志,如需恢复，只能手动将 index.blocks.read_only_allow_delete 改成false.

3. 参考处理方式

3.1 ES PVC 扩容

日志 ELK 默认部署在集群 default 命名空间,如果部署在自定义命名空间，执行命令请替换 default 名称

Step 1. 登录集群 Master 节点，执行命令:kubectl get pvc -n default 查看 PVC，其中如下名字的 PVC 是 ES 使用的

```
multi-master-multi-master-0
multi-master-multi-master-1
multi-master-multi-master-2
```

执行 kubectl edit pvc {pvc-name} -n default，将 spec.resource.requests.storage 的值调大，保存后退出，大概在一分钟左右，PV、PVC 以及容器内的文件系统就完成了在线扩容，详细操作参考UDisk 动态扩容。

扩容后确认 PV/PVC 状态：kubectl get pv | grep multi-master && kubectl get pvc | grep multi-master

Step 2. 解除 ES 索引只读模式

```
ES_CLUSTER_IP=`kubectl get svc multi-master | awk 'NR>1 {print $3}'`
curl -H "Content-Type: application/json" -XPUT http://${ES_CLUSTER_IP}:9200/_all/_settings -d '{ "index.blocks.read_only_allow_delete": false }'
```

Step 3. 确认 ES集群状态

```
ES_CLUSTER_IP=`kubectl get svc multi-master | awk 'NR>1 {print $3}'`
curl http://${ES_CLUSTER_IP}:9200/_cat/allocation?pretty
curl http://${ES_CLUSTER_IP}:9200/_cat/health
curl http://${ES_CLUSTER_IP}:9200/_all/_settings | jq
```

3.2 调整 ES 配置

如果目前 ES PVC 容量非常大，按照 ES 默认配置 90% 存储依然剩余大量空余空间，可以调大 ES 参数阈值，解除索引只读模式，将 ES 集群恢复至正常状态

```
ES_CLUSTER_IP=`kubectl get svc multi-master | awk 'NR>1 {print $3}'`

curl -H "Content-Type: application/json" -XPUT http://${ES_CLUSTER_IP}:9200/_cluster/settings -d '{
"persistent": {
"cluster.routing.allocation.disk.watermark.low": "90%",
"cluster.routing.allocation.disk.watermark.high": "95%",
"cluster.routing.allocation.disk.watermark.flood_stage": "97%",
"cluster.info.update.interval": "1m"
}
```

```
}'
curl -H "Content-Type: application/json" -XPUT http://${ES_CLUSTER_IP}:9200/_all/_settings -d '{
"index.blocks.read_only_allow_delete": false
}'
```

3.3 ES 相关参数说明:

- cluster.routing.allocation.disk.watermark.low,控制磁盘使用的低水位线(watermark),它默认为 85%,这意味着 Elasticsearch 不会将分片分配存储空间使用率超过 85% 的节点。它还可以设置为绝对字节值(如 500MB),以防止 Elasticsearch 在可用空间少于指定数量时分配分片。此设置对新创建索引的主分片没有影响,特别是对以前从未分配过的任何分片。
- cluster.routing.allocation.disk.watermark.high,控制高水位线,它默认为 90%,这意味着 Elasticsearch 将尝试将分片从存储使用率高于 90% 的节点重新定位。它还可以设置为绝对字节值(类似于低水位线),以便在分片的可用空间小于指定数量时将其重新定位到远离节点的位置。此设置影响所有分片的分配,无论以前是否分配。
- cluster.routing.allocation.disk.watermark.flood_stage,控制洪泛水位线。它默认为 95%,一旦有一个 ES 节点存储空间超过了洪泛阶段 Elasticsearch 将对索引块强制执行只读设置 index.blocks.read_only_allow_delete: true 这是防止节点耗尽存储空间的最后手段。一旦有足够的空间允许索引操作继续,则必须手动调整 index.blocks.read_only_allow_delete: false 取消索引只读属性。

参考:Elasticsearch 官方文档

使用GPU节点

可通过以下方式在UK8s中使用GPU云主机作为集群节点：

- 创建集群
- 新增Node节点
- 添加已有主机

镜像说明

在UK8s集群中使用高性价比显卡的云主机机型（如高性价比显卡3、高性价比显卡4、高性价比显卡5、高性价比显卡6）作为节点时，需使用标准镜像Ubuntu 20.04-高性价比。

- 高性价比显卡支持可用区
  - 华北二A
  - 上海二B
  - 北京二B

标准镜像名	适用显卡	Nvidia驱动版本	CUDA版本
Ubuntu 20.04-高性价比	高性价比显卡（如高性价比显卡3/4/5/6等）	550.120	12.4
Ubuntu 20.04	非高性价比显卡（如T4、V100S、P40等）	550.90.12	12.4
Centos 7.6	非高性价比显卡（如T4、V100S、P40等）	450.80.02	11.0

创建集群

创建集群时，在Node节点配置中，选择机型为“GPU型G”，然后选择具体的GPU卡型及配置。

注：如果选择了高性价比显卡，需要在节点镜像中使用标准镜像Ubuntu 20.04-高性价比。

新增Node节点

在已有集群中新增Node节点时，选择机型为“GPU型G”，然后选择具体的GPU卡型及配置。

添加已有主机

将已创建的GPU云主机添加进已有集群，选择合适的节点镜像。

使用说明

1. 默认情况下，容器之间不共享 GPU，每个容器可以请求一个或多个 GPU。无法请求 GPU 的一小部分。

2. 集群的 Master 节点暂不支持 GPU 机型。
3. UK8S提供的标准镜像中，已安装nvidia驱动，并且，集群中默认安装了nvidia-device-plugin组件，GPU资源添加到集群后可以被自动识别和注册。
4. 如何验证GPU节点的正常使用：
    1. 查看节点是否具有nvidia.com/gpu的资源。

```
Allocatable:
  cpu:                15800m
  ephemeral-storage:  18903225108
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             30751578907
  nvidia.com/gpu:     1
  pods:               110
  ucloud.cn/uni:      16
```

    2. 运行如下示例使用nvidia.com/gpu资源类型请求 NVIDIA GPU，并查看日志结果是否正确。

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
name: gpu-pod
spec:
restartPolicy: Never
containers:
- name: cuda-container
image: uhub.service.ucloud.cn/uk8s/cuda-sample:vectoradd-cuda10.2
resources:
limits:
nvidia.com/gpu: 1 # requesting 1 GPU
tolerations:
```

```
- key: nvidia.com/gpu
operator: Exists
effect: NoSchedule
EOF
```

```
$ kubectl logs gpu-pod
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

5. GPU云主机NCCL TOPO文件透传pod

   如果在GPU pod内NCCL性能测试没有达到理想值，考虑从虚机上把topology.xml文件透传到pod内；具体操作如下：

   > 前提：您的node为8卡高性价比显卡6/高性价比显卡6pro/A800/等GPU

   1. 确认GPU node `/var/run/nvidia-topologyd/` 路径下是否存在 `virtualTopology.xml`文件
      - 若存在执行第2步
      - 若不存在请咨询技术支持，将提供您该文件，拷贝文件保存至GPU node的 `/var/run/nvidia-topologyd/virtualTopology.xml`后执行第2步
   2. gpu-pod.yaml添加以下内容

```
```

```
containers:
volumeMounts:
- mountPath: /var/run/nvidia-topologyd
name: topologyd
readOnly: true
volumes:
- name: topologyd
hostPath:
path: /var/run/nvidia-topologyd
type: Directory
```

插件升级

将 nvidia-device-plugin 升级到最新版本，以解决 GPU 节点不稳定的情况。

升级方法

- 方法一：使用 `kubectl set image` 将 `nvidia-device-plugin-daemonset` 的镜像版本更改为 `v0.14.1`：

```
$ kubectl set image daemonset nvidia-device-plugin-daemonset -n kube-system nvidia-device-plugin-ctr=uhub.service.ucloud.cn/uk8s/nvidia-k8s-device-plugin:v0.14.1
daemonset.apps/nvidia-device-plugin-daemonset image updated
```

- 方法二：更改 `nvidia-device-plugin-daemonset` 的 yaml 文件：

1. 输入如下指令：

```
$ kubectl edit daemonset nvidia-device-plugin-daemonset -n kube-system
```

2. 此时会得到 nvidia-device-plugin-daemonset 的配置，找到 spec.template.spec.containers.image 后，可以看到目前镜像信息：

```
- image: uhub.service.ucloud.cn/uk8s/nvidia-k8s-device-plugin:1.0.0-beta4
```

3. 更改镜像为 uhub.service.ucloud.cn/uk8s/nvidia-k8s-device-plugin:v0.14.1，随后保存。

裸金属云主机绑核

目前裸金属默认支持了绑核，在某些场景下绑核提高GPU效率；

通过删除裸金属节点文件"/var/lib/kubelet/cpu_manager_state"，且默认配置Kubelet如下参数来支持绑核功能；相关官方文档可参考Topology Manager Policy，CPU Mangaer Policy

```
--cpu-manager-policy=static
--topology-manager-policy=best-effort
```

节点是否配置了支持绑核的参数，可登陆节点使用命令做参数检测：

```
ps -aux|grep kubelet|grep topology-manager-policy
```

验证绑核成功

1. 创建测试 Pod 前，需要注意以下几点：

   - 确保 limits 和 requests 中的 CPU、Memory、GPU 数量是一致的，并且 CPU 需要是整数。
   - 可以将 spec.nodeName 设置为裸金属云主机的 ip 地址，确保 Pod 被调度到该节点上。

   现在我们来创建 Pod：

```
apiVersion: v1
kind: Pod
metadata:
name: dcgmproftester
spec:
nodeName: "10.60.159.170" # 这里替换为裸金属节点的ip
restartPolicy: OnFailure
containers:
- name: dcgmproftester12-1
image: uhub.service.ucloud.cn/uk8s/dcgm:3.3.0
command: ["/usr/bin/dcgmproftester12"]
args: ["--no-dcgm-validation", "-t 1004", "-d 3600"] # 这里 -d 为运行时间
resources: # 这里的数值需要根据不同机器配置进行修改
limits: ## limit 与 request保持一致
nvidia.com/gpu: 1
memory: 10Gi
cpu: 10
requests:
```

```
nvidia.com/gpu: 1
memory: 10Gi
cpu: 10
securityContext:
capabilities:
add: ["SYS_ADMIN"]
```

2. 等待 Pod 状态为 Running 之后，通过 ssh 进入裸金属节点内。

3. 输入指令 crictl ps 来获取当前节点内容器列表。根据创建时间或者容器名称找到我们刚刚创建的容器的 ID。然后输入指令 crictl inspect <容器ID> | grep pid，获取进程的 pid。

4. 输入指令 taskset -c -p <pid> 来获取 CPU 亲和性信息：

```
root@10-60-173-208:/home/ubuntu# taskset -c -p 9070
pid 9070's current affinity list: 1-5,65-69
```

我们看到 Pod 亲和的 CPU 范围为 1-5,65-69 而不是完整的 CPU 列表，证明绑 CPU 成功。

5. 现在输入指令 nvidia-smi 来获取 gpu 信息，检查 GPU 亲和性：

```
root@10-60-173-208:/home/ubuntu# nvidia-smi
Tue Nov  5 15:38:57 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.90.07              Driver Version: 550.90.07      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4090        On  |   00000000:01:00.0 Off |                  Off |
| 45%   27C    P8             5W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 4090        On  |   00000000:24:00.0 Off |                  Off |
| 45%   26C    P8             4W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   2  NVIDIA GeForce RTX 4090        On  |   00000000:41:00.0 Off |                  Off |
```

```
| 44%   27C    P8              5W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                         |                  N/A |
+-----------------------------------------+-------------------------+----------------------+
|   3  NVIDIA GeForce RTX 4090        On  |   00000000:61:00.0 Off |                  Off |
| 88%   68C    P2            449W /  450W |     790MiB /  24564MiB |     98%      Default |
|                                         |                         |                  N/A |
+-----------------------------------------+-------------------------+----------------------+
|   4  NVIDIA GeForce RTX 4090        On  |   00000000:81:00.0 Off |                  Off |
| 45%   27C    P8             10W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                         |                  N/A |
+-----------------------------------------+-------------------------+----------------------+
|   5  NVIDIA GeForce RTX 4090        On  |   00000000:A1:00.0 Off |                  Off |
| 44%   27C    P8             14W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                         |                  N/A |
+-----------------------------------------+-------------------------+----------------------+
|   6  NVIDIA GeForce RTX 4090        On  |   00000000:C1:00.0 Off |                  Off |
| 45%   27C    P8             11W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                         |                  N/A |
+-----------------------------------------+-------------------------+----------------------+
|   7  NVIDIA GeForce RTX 4090        On  |   00000000:E1:00.0 Off |                  Off |
| 44%   27C    P8              4W /  450W |       2MiB /  24564MiB |      0%      Default |
|                                         |                         |                  N/A |
+-----------------------------------------+-------------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    3   N/A  N/A      9110      C   /usr/bin/dcgmproftester12                   782MiB |
+---------------------------------------------------------------------------------------+
```

上图中的 Process 框中记录了哪一个 GPU 在运行，图中为 GPU2 在运行。证明绑 GPU 成功。

6. 通过指令 `nvidia-smi topo -m` 来检查 GPU 和 CPU 是否在同一个 NUMA 节点上：

```
root@10-60-173-208:/home/ubuntu# nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      SYS     SYS     SYS     SYS     SYS     SYS     SYS     24-31,88-95     3               N/A
GPU1    SYS      X      SYS     SYS     SYS     SYS     SYS     SYS     16-23,80-87     2               N/A
GPU2    SYS     SYS      X      SYS     SYS     SYS     SYS     SYS     8-15,72-79      1               N/A
GPU3    SYS     SYS     SYS      X      SYS     SYS     SYS     SYS     0-7,64-71       0               N/A
GPU4    SYS     SYS     SYS     SYS      X      SYS     SYS     SYS     56-63,120-127   7               N/A
GPU5    SYS     SYS     SYS     SYS     SYS      X      SYS     SYS     48-55,112-119   6               N/A
GPU6    SYS     SYS     SYS     SYS     SYS     SYS      X      SYS     40-47,104-111   5               N/A
GPU7    SYS     SYS     SYS     SYS     SYS     SYS     SYS      X      32-39,96-103    4               N/A
```

我们可以看出 GPU3 的 CPU Affinity 为 `0-7,64-71`。在第 4 步中我们得到进程的 CPU Affinity list 为 `1-5,65-69`。这证明了 GPU 和 CPU 对应了同一个 NUMA 节点。

7. Best-effort 策略补充说明:

使用 `best-effort` 策略时，需要提前了解使用的裸金属配置以决定 Pod 申请 CPU 和 GPU 的参数设置。例如现在有一台裸金属云主机配置如下：

CPU	GPU	内存（不影响绑核）	NUMA 节点数量
128	8	1024	8

> CPU 和 NUMA 参数可以通过指令 `lscpu` 获取。GPU 参数可以通过指令 `nvidia-smi topo -m` 获取。

通过指令 `lscpu` 我们可以得知 NUMA 节点和 CPU 核心的关系：

```
...
NUMA node0 CPU(s): 0-7,64-71
NUMA node1 CPU(s): 8-15,72-79
NUMA node2 CPU(s): 16-23,80-87
NUMA node3 CPU(s): 24-31,88-95
NUMA node4 CPU(s): 32-39,96-103
NUMA node5 CPU(s): 40-47,104-111
NUMA node6 CPU(s): 48-55,112-119
NUMA node7 CPU(s): 56-63,120-127
...
```

通过指令 `nvidia-smi topo -m` 我们可以得知 NUMA 节点和 GPU 的关系：

```
root@10-60-173-208:/home/ubuntu# nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      SYS     SYS     SYS     SYS     SYS     SYS     SYS     24-31,88-95     3               N/A
GPU1    SYS      X      SYS     SYS     SYS     SYS     SYS     SYS     16-23,80-87     2               N/A
GPU2    SYS     SYS      X      SYS     SYS     SYS     SYS     SYS     8-15,72-79      1               N/A
GPU3    SYS     SYS     SYS      X      SYS     SYS     SYS     SYS     0-7,64-71       0               N/A
GPU4    SYS     SYS     SYS     SYS      X      SYS     SYS     SYS     56-63,120-127   7               N/A
GPU5    SYS     SYS     SYS     SYS     SYS      X      SYS     SYS     48-55,112-119   6               N/A
GPU6    SYS     SYS     SYS     SYS     SYS     SYS      X      SYS     40-47,104-111   5               N/A
GPU7    SYS     SYS     SYS     SYS     SYS     SYS     SYS      X      32-39,96-103    4               N/A
```

可以看出每个 NUMA 节点包含了 16 核 CPU 和 1 个 GPU。为了可以确保 CPU 和 GPU 都亲和相同的 NUMA 节点，我们的配置需要保证 **GPU 亲和的 NUMA 节点数量等于 CPU 亲和的 NUMA 节点数量，否则可能导致亲和节点不一致**。下面是能够实现亲和的情况：

GPU	CPU
1	1 ~ 16
n	> 16 * (n-1), 且 ≤ 16 * n

> 这里的 16 和 1 是根据上文的方法查看配置得到的。不同机器的配置是不一样的，需要自行查看。

除了以上 GPU/CPU 配比，其他情况均无法达到 NUMA 亲和的效果。

具体可参考官方文档

GPU插件

1. 介绍

UK8S 使用开源组件 HAMi 实现GPU共享，包括：

- 显存划分
- 算力划分
- 错误隔离

2. 部署

⚠ 安装前请检查系统需满足以下要求：

- **NVIDIA drivers**: 版本需不低于 `440`。
- **Kubernetes 版本**: 不低于 `1.16`。
- **glibc**: 版本需在 `2.17` 及以上，但低于 `2.3.1`。

2.1 将需要由 HAMi 调度的 GPU 节点打上标签：

```
kubectl label nodes xxx.xxx.xxx.xxx gpu=on
```

2.2 helm安装

⚠ helm版本要求3.0以上，安装后通过以下指令查看helm版本

```
helm version
```

2.3 获取chart

下载chart压缩包，并且解压

```
wget https://docs.ucloud.cn/uk8s/yaml/gpu-share/hami.tar.gz
tar -xzf hami.tar.gz
rm hami.tar.gz
```

2.4 安装hami

```
helm install hami ./hami -n kube-system
```

检查安装结果

```
kubectl get po -A
```

安装成功时，以下输出显示 Pod 运行状态：

```
hami-device-plugin-l4jj4 2/2 Running 0 45s
```

```
hami-scheduler-59c7f4b6ff-7g565 2/2 Running 0 3m54s
```

2.5 使用方法

通过YAML 文件来创建 Pod，注意 resources.limit 中，除了传统的 nvidia.com/gpu 外，增加了 nvidia.com/gpumem 和 nvidia.com/gpucores 等来指定显存大小和 GPU 算力。

- **nvidia.com/gpu**: 请求的 vGPU 数量，例如 1。
- **nvidia.com/gpumem**: 请求的显存大小，例如 3000M。
- **nvidia.com/gpumem-percentage**: 显存请求百分比，例如 50 表示请求 50% 的显存。
- **nvidia.com/gpucores**: 每个 vGPU 的算力占实际显卡的百分比。
- **nvidia.com/priority**: 优先级，0 表示高优先级，1 表示低优先级，默认为 1。
    - 对于高优先级任务，如果与其他高优先级任务共享 GPU 节点，则其资源利用率不会受到 resourceCores 的限制。换言之，当 GPU 节点上仅有高优先级任务时，它们可利用节点上所有可用的资源。
    - 对于低优先级任务，如果该任务是唯一占用 GPU 的任务，则其资源利用率同样不受 resourceCores 的限制。这意味着在无其他任务共享 GPU 的情况下，低优先级任务也可以利用节点上所有可用的资源。

3. 监控

3.1. 若未开启监控中心

开启 监控中心

3.2. 若已开启监控中心

⚠ 如果监控中心版本 1.0.6 > version >= 1.0.5-3 或者 version > 1.0.6 ,则默认安装了下面部署文件,请跳过3.2.1中的部署内容。

3.2.1. 部署 Dcgm-Exporter

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/gpu-share/dcgm-exporter.yaml
```

3.2.2. 在 uk8s 添加监控目标

在uk8s-监控中心-监控目标之中,按照图示的方式,在 uk8s 中添加监控目标。

3.3 Grafana监控

在 uk8s 添加监控目标登陆Grafana后,你需要先下载 json 文件 --> 选择左侧导航栏 '+' 号 --> Import --> 第二个输入框粘贴下载的 json 内容 --> Load。

以下是Grafana监控HAMi的示意图：

3.4 监控指标

除了DCGM插件（在GPU监控文档之中具体描述）之外的指标，HAMi还支持以下指标：

- **Device_memory_desc_of_container**：表示容器中设备内存的实时使用情况。用于监控每个容器的设备（如 GPU）内存消耗。
- **Device_utilization_desc_of_container**：表示容器设备的实时利用率。用于监控容器内部设备的使用情况，如 GPU 的工作负载。
- **HostCoreUtilization**：表示宿主机上核心的实时利用率。用于监控宿主机的 CPU 核心使用情况，可能包括容器或虚拟化的多个工作负载。
- **HostGPUMemoryUsage**：表示宿主机上 GPU 设备的实时内存使用情况。用于监控宿主机中使用 GPU 的各个容器或任务所消耗的内存。
- **vGPU_device_memory_limit_in_bytes**：表示某个容器的 vGPU（虚拟 GPU）设备内存的限制，单位是字节。这是该容器可以使用的最大 GPU 内存量。
- **vGPU_device_memory_usage_in_bytes**：表示某个容器的 vGPU 设备内存的实际使用量，单位是字节。用于监控该容器的 vGPU 内存使用情况。

4. 测试

4.1. Node GPU 资源验证

在测试环境中，物理 GPU 数量为 1，但由于 HAMi 默认配置扩容比例为 10 倍，因此理论上可以在 Node 上查看到 `1 * 10 = 10` 个 GPU。

验证指令

执行以下命令获取 Node 的 GPU 资源量：

```
kubectl get node xxx -oyaml | grep capacity -A 8
```

得到的输出如下：

```
capacity:
```

```
cpu: "16"
ephemeral-storage: 102687672Ki
hugepages-1Gi: "0"
hugepages-2Mi: "0"
memory: 32689308Ki
nvidia.com/gpu: "10"
pods: "110"
ucloud.cn/uni: "16"
```

4.2. GPU 显存测试

以下是 gpu-mem-test.yaml 的配置内容：

```
apiVersion: v1
kind: Pod
metadata:
name: gpu-pod
spec:
containers:
- name: ubuntu-container
image: uhub.service.ucloud.cn/library/ubuntu:trusty-20160412
command: ["bash", "-c", "sleep 86400"]
resources:
limits:
nvidia.com/gpu: 1 # 请求1个vGPU
```

```
nvidia.com/gpumem: 3000 # 每个vGPU申请3000MiB显存(可选,整数类型)
nvidia.com/gpucores: 30 # 每个vGPU的算力为30%实际显卡的算力(可选,整数类型)
```

使用上述配置创建 Pod,Pod 应能正常启动,验证步骤如下:

```
kubectl get po
```

得到的输出:

```
NAME READY STATUS RESTARTS AGE
gpu-pod 1/1 Running 0 48s
```

进入 Pod 并执行 `nvidia-smi` 命令,查看 GPU 信息,可以验证到显存的限制为资源中申请的 `3000M`。

4.3. 多个 Pod 单个 GPU 利用率测试

测试命令

首先创建 `hami-npod-1gpu.yml`,内容如下,请替换其中的 GPU 节点 IP,用于指定运行的 GPU 节点。

该配置创建了n个 Pod,(通过修改yaml之中的副本数量控制)建议测试完成后手动删除。

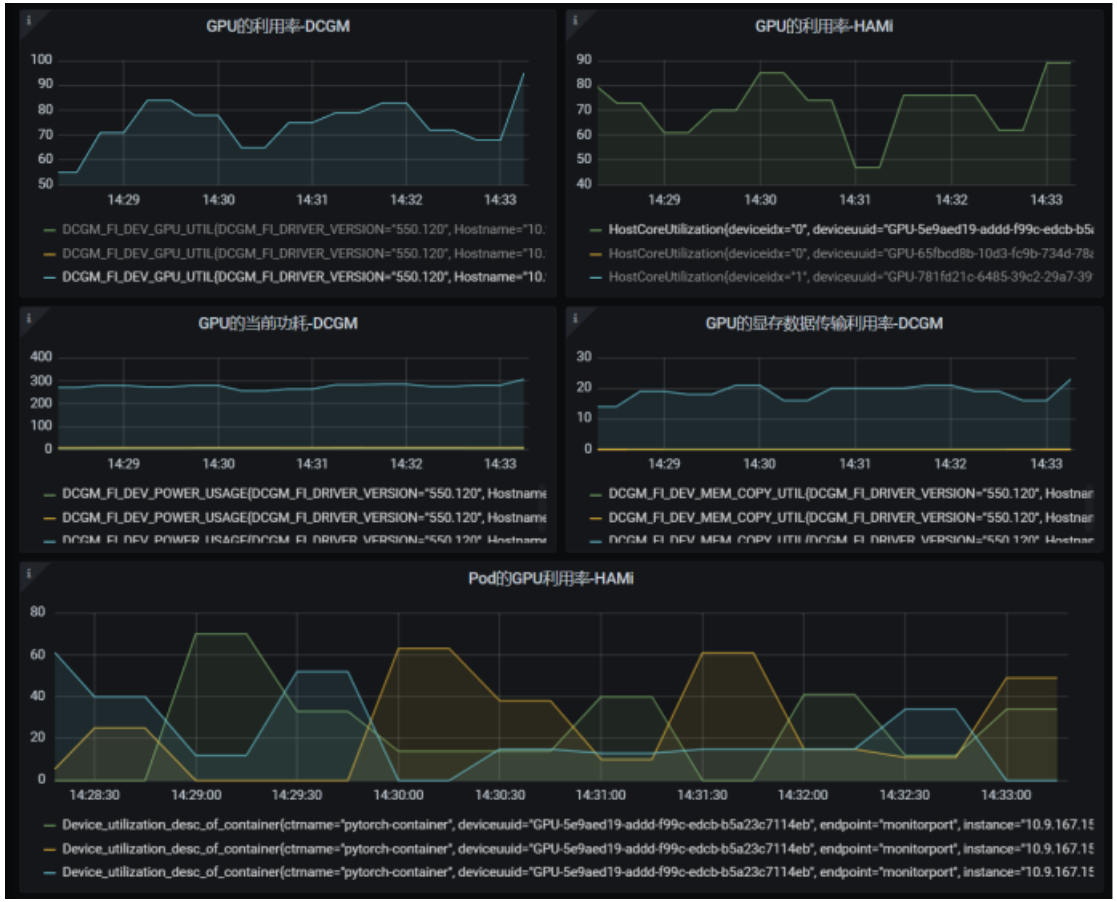
```
apiVersion: apps/v1
kind: Deployment
metadata:
name: hami-npod-1gpu
```

```
spec:
replicas: 3 # 创建三个相同的 Pod,可根据需要修改数量
selector:
matchLabels:
app: pytorch
template:
metadata:
labels:
app: pytorch
spec:
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: kubernetes.io/hostname
operator: In
values:
- xxx.xxx.xxx.xxx # 请替换为实际 GPU 节点 IP
containers:
- name: pytorch-container
image: uhub.service.ucloud.cn/gpu-share/gpu_pytorch_test:latest
command: ["/bin/sh", "-c"]
args: ["cd /app/pytorch_code && python3 2.py"]
resources:
```

```
limits:
nvidia.com/gpu: 1
nvidia.com/gpumem: 3000
nvidia.com/gpucores: 25
```

测试结果

通过监控可知，3个pod在长期的平均算力消耗符合限制，短期下存在波动。



集群GPU监控

1. 介绍

Uk8s 使用开源组件 Dcgm-Exporter 用于获取 GPU 相关监控指标，主要包含：

- GPU 卡利用率
- 容器 GPU 资源利用率

GPU监控与当前版本的GPU共享插件存在兼容问题，目前暂不支持同时开启

2. 部署

2.1. 未开启监控中心

开启 监控中心 即可在 Grafana 页面查看 Dashboard `NVIDIA/DCGM/Exporter/Node`、`NVIDIA/DCGM/Exporter/Container`。

2.2. 已开启监控中心

⚠ 如果监控中心版本 1.0.6 > version >= 1.0.5-3 或者 version > 1.0.6 ，默认安装了下面部署文件，请跳过下面部署内容，否则需要进行下面部署。

2.2.1. 部署 Dcgm-Exporter

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/gpu-share/dcgm-exporter.yaml
```

2.2.2. 部署 NVIDIA/DCGM/Exporter/Node Dashboard

登陆Grafana后，你需要先下载 json 文件 --> 选择左侧导航栏 '+' 号 --> Import --> 第二个输入框粘贴下载的 json 内容 --> Load

2.2.3. 部署 NVIDIA/DCGM/Exporter/Container Dashboard

> ⚠ 官方的图表没有容器相关信息，如果你需要查看容器的 GPU 相关信息，需要导入 Uk8s 自制的 Dashboard。

登陆Grafana后，你需要先下载 json 文件 --> 选择左侧导航栏 '+' 号 --> Import --> 第二个输入框粘贴下载的 json 内容 --> Load

3. 测试

你可以通过下面命令快速启动一个 GPU Pod。该 Pod 会运行一段时间结束。随后你可以在 Grafana 的 `NVIDIA/DCGM/Exporter/Container` Dashboard 中查看该 Pod 的 GPU 使用情况。

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: Pod
```

```
metadata:
name: dcgmproftester
spec:
restartPolicy: OnFailure
containers:
- name: dcgmproftester12
image: uhub.service.ucloud.cn/uk8s/dcgm:3.3.0
command: ["/usr/bin/dcgmproftester12"]
args: ["--no-dcgm-validation", "-t 1004", "-d 120"]
resources:
limits:
nvidia.com/gpu: 1
securityContext:
capabilities:
add: ["SYS_ADMIN"]

EOF
```

4. Dashboard 图表

Dashboard	Grafana图表	作用
NVIDIA/DCGM/Exporter/Node	GPU Temperature	GPU 卡温度

NVIDIA/DCGM/Exporter/Node	GPU Power Usage	GPU 功耗
NVIDIA/DCGM/Exporter/Node	GPU SM Clocks	GPU 时钟频率
NVIDIA/DCGM/Exporter/Node	GPU Utilization	GPU 利用率
NVIDIA/DCGM/Exporter/Node	Tensor Core Utilization	Tensor Pipes 平均处于 Active 状态的周期分数
NVIDIA/DCGM/Exporter/Node	GPU Framebuffer Mem Used	GPU 显存使用量
NVIDIA/DCGM/Exporter/Node	GPU XID Error	GPU 掉卡
NVIDIA/DCGM/Exporter/Container	GPU Utilization	容器 GPU 利用率
NVIDIA/DCGM/Exporter/Container	GPU Framebuffer Mem	容器 GPU 显存使用量&剩余量
NVIDIA/DCGM/Exporter/Container	GPU Memory Usage	容器 GPU 显存使用率

5. 监控规则

我们新版本监控(version >= 1.0.10)默认配置了 GPU掉卡 的告警规则，针对老版本需进行手动部署

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/gpu-share/gpu-xid.yaml
```

如果有新增告警规则的需求，可以通过下面命令更改告警规则。

```
kubectl -n uk8s-monitor edit prometheusrule uk8s-dcgm
```

6. DCGM 常见指标

6.1. 利用率

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_GPU_UTIL	Gauge	%	GPU 利用率。
DCGM_FI_DEV_MEM_COPY_UTIL	Gauge	%	GPU 内存带宽利用率。
DCGM_FI_DEV_ENC_UTIL	Gauge	%	GPU 编码器利用率。
DCGM_FI_DEV_DEC_UTIL	Gauge	%	GPU 解码器利用率。

6.2. 显存

在 GPU 里，显卡内存（显存）也被称为帧缓存。

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_FB_FREE	Gauge	MiB	GPU 帧缓存剩余量。
DCGM_FI_DEV_FB_USED	Gauge	MiB	GPU 帧缓存剩余量。

6.3. 频率

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_SM_CLOCK	Gauge	MHz	GPU SM 时钟频率。
DCGM_FI_DEV_MEM_CLOCK	Gauge	MHz	GPU 内存时钟频率。

6.4. 剖析

指标名称	指标类型	指标单位	指标含义
DCGM_FI_PROF_GR_ENGINE_ACTIVE	Gauge	%	在一个时间间隔内，Graphics 或 Compute 引擎处于 Active 的时间占比。
DCGM_FI_PROF_SM_ACTIVE	Gauge	%	在一个时间间隔内，至少一个线程束在一个 SM（Streaming Multiprocessor）上处于 Active 的时间占比。该值统计的是所有 SM 的均值。
DCGM_FI_PROF_SM_OCCUPANCY	Gauge	%	在一个时间间隔内，驻留在 SM 上的线程束与该 SM 最大可驻留线程束的比例。该值统计的是所有 SM 的均值。
DCGM_FI_PROF_PIPE_TENSOR_ACTIVE	Gauge	%	单位时间内 Tensor Pipes 平均处于 Active 状态的周期分数。
DCGM_FI_PROF_DRAM_ACTIVE	Gauge	%	内存拷贝活跃周期分数（一个周期内有一次 DRAM 指令则该周期为 100%）。
DCGM_FI_PROF_PIPE_FP64_ACTIVE	Gauge	%	单位时间内 F64 Pipes 平均处于 Active 状态的周期分数。
DCGM_FI_PROF_PIPE_FP32_ACTIVE	Gauge	%	单位时间内 F32 Pipes 平均处于 Active 状态的周期分数。
DCGM_FI_PROF_PIPE_FP16_ACTIVE	Gauge	%	单位时间内 F16 Pipes 平均处于 Active 状态的周期分数。
DCGM_FI_PROF_NVLINK_RX_BYTES	Counter	B/s	通过 NVLink 接收的数据流量。

DCGM_FI_PROF_NVLINK_TX_BYTES	Counter	B/s	通过 NVLink 传输的数据流量。
DCGM_FI_PROF_PCIE_RX_BYTES	Counter	B/s	通过 PCIe 总线接收字节数。
DCGM_FI_PROF_PCIE_TX_BYTES	Counter	B/s	通过 PCIe 总线传输字节数。
DCGM_FI_DEV_PCIE_REPLAY_COUNTER	Counter	次	GPU PCIe 总线的重试次数。
DCGM_FI_DEV_NVLINK_BANDWIDTH_TOTAL	Counter	-	GPU 所有通道的 NVLink 带宽计数器总数。

6.5. 温度和功率

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_GPU_TEMP	Gauge	℃	GPU 当前温度
DCGM_FI_DEV_MEMORY_TEMP	Gauge	℃	GPU 显存当前温度
DCGM_FI_DEV_POWER_USAGE	Gauge	W	GPU 当前使用功率
DCGM_FI_DEV_TOTAL_ENERGY_CONSUMPTION	Counter	mJ	GPU 启动以来的总能耗

6.6. XID 错误&违规

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_XID_ERRORS	Gauge	-	最近发生的错误代码

DCGM_CUSTOM_XID_ERRORS_TOTAL_COUNTER	Counter	-	发生错误代码总数
DCGM_FI_DEV_POWER_VIOLATION	Counter	µs	因功率上限而导致违规的累积持续时间
DCGM_FI_DEV_THERMAL_VIOLATION	Counter	µs	因热限制导致违规的累积持续时间
DCGM_FI_DEV_SYNC_BOOST_VIOLATION	Counter	µs	因同步提升限制而导致违规的累积持续时间
DCGM_FI_DEV_BOARD_LIMIT_VIOLATION	Counter	µs	因电路板限制而导致违规的累积持续时间
DCGM_FI_DEV_LOW_UTIL_VIOLATION	Counter	µs	因低利用率限制导致违规的累积持续时间
DCGM_FI_DEV_RELIABILITY_VIOLATION	Counter	µs	因电路板可靠性限制导致违规的累积持续时间

6.7. 停用的内存页面

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_RETIRED_SBE	Counter	个	因单 bit 错误而停用的内存页面
DCGM_FI_DEV_RETIRED_DBE	Counter	个	因双 bit 错误而停用的内存页面

6.8. 其他

指标名称	指标类型	指标单位	指标含义
DCGM_FI_DEV_VGPU_LICENSE_STATUS	Gauge	-	vGPU 许可证状态
DCGM_FI_DEV_UNCORRECTABLE_REMAPPED_ROWS	Counter	-	因无法纠正的错误而重新映射的行数
DCGM_FI_DEV_CORRECTABLE_REMAPPED_ROWS	Counter	-	因可纠正的错误而重新映射的行数

DCGM_FI_DEV_ROW_REMAP_FAILURE	Gauge	-	重新映射行是否失败

NodePort 相关参数修改

1. API Server NodePort Range 修改

UK8S 集群中，APIServer 相关参数，保存在 Master 节点 /etc/kubernetes/apiserver 文件中，请逐台在 Master 节点上，进行如下操作。

备份该配置文件，修改该文件中 --service-node-port-range **至需要的参数。**

```
KUBE_API_ARGS=" --... \
--... \
--service-node-port-range=30000-32767 \
--... \
--..."
```

修改完成后，通过 systemctl restart kube-apiserver，重启 APIServer

⚠ 如果集群中的 Service 已经分配了目标 NodePort Range 之外的端口，修改之后通过 kubectl get event -A 查看集群事件，会有类似以下 warning 事件产生，warning 事件不影响该端口使用，如希望避免该 warning，需要重建该 Service。

```
# kubectl get event -A
NAMESPACE LAST SEEN TYPE REASON OBJECT MESSAGE
```

```
uk8s-monitor 79s Warning PortOutOfRange service/grafana Port 34840 is not within the port range 30000-32767; please recreate service
uk8s-monitor 78s Warning PortNotAllocated service/grafana Port 34840 is not allocated; repairing
```

2. 修改节点 ip_local_port_range

查看当前系统开放端口范围，命令如下：

```
# cat /proc/sys/net/ipv4/ip_local_port_range
12000 65535
```

如需修改，请更改 `/etc/sysctl.conf` 文件中 `net.ipv4.ip_local_port_range` 字段

```
net.ipv4.ip_local_port_range = 32768 60999
```

修改完成后，执行 `sysctl -p` 生效配置，并再次验证：

```
# cat /proc/sys/net/ipv4/ip_local_port_range
32768 60999
```

ETCD 备份

ETCD 备份插件是 UK8S 提供的 ETCD 数据库备份功能，ETCD 备份可以有效的帮助集群进行故障恢复操作，可根据插件设置进行定时备份、指定存储等功能。

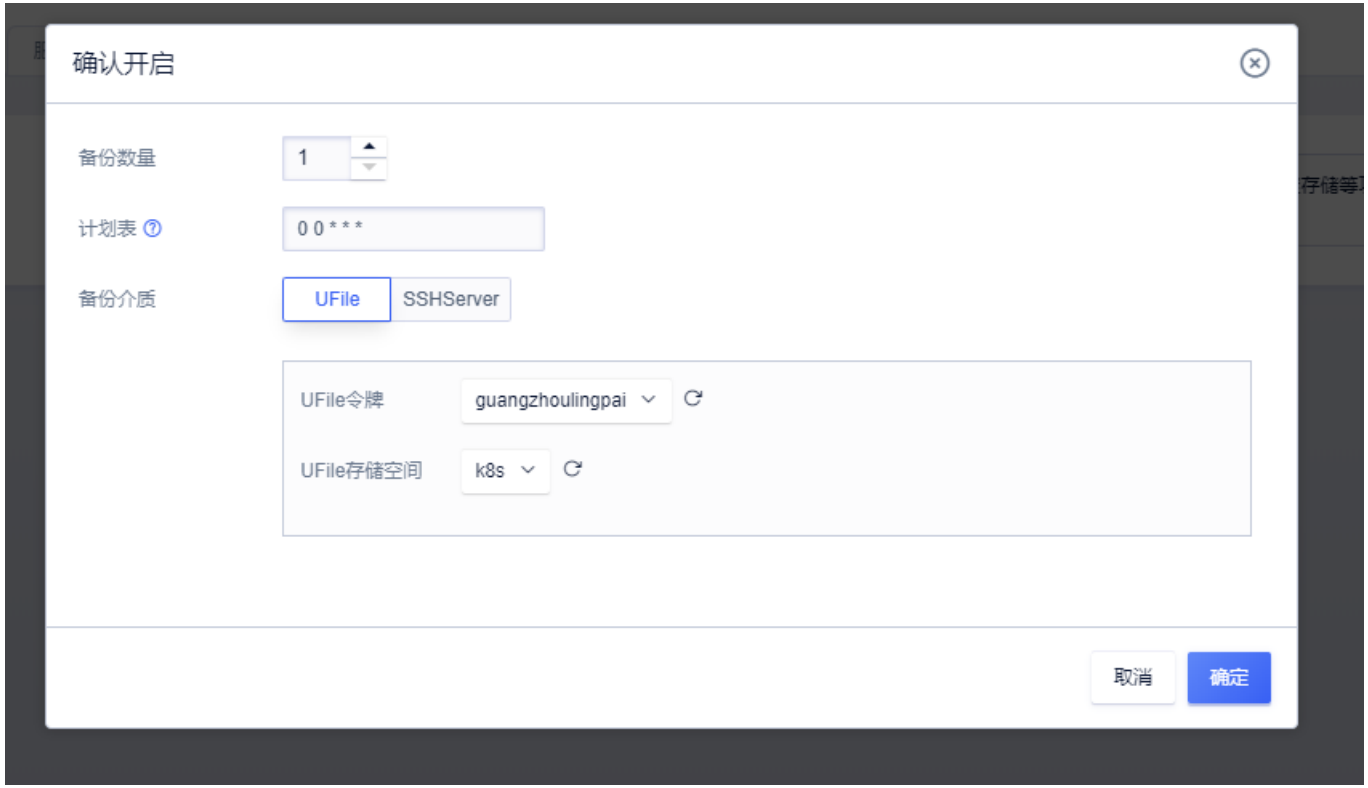
1. 首次操作

如您是首次安装插件提示您需要保存密钥操作，请在任一 Master 节点上执行以下命令，执行完成后刷新页面即可。

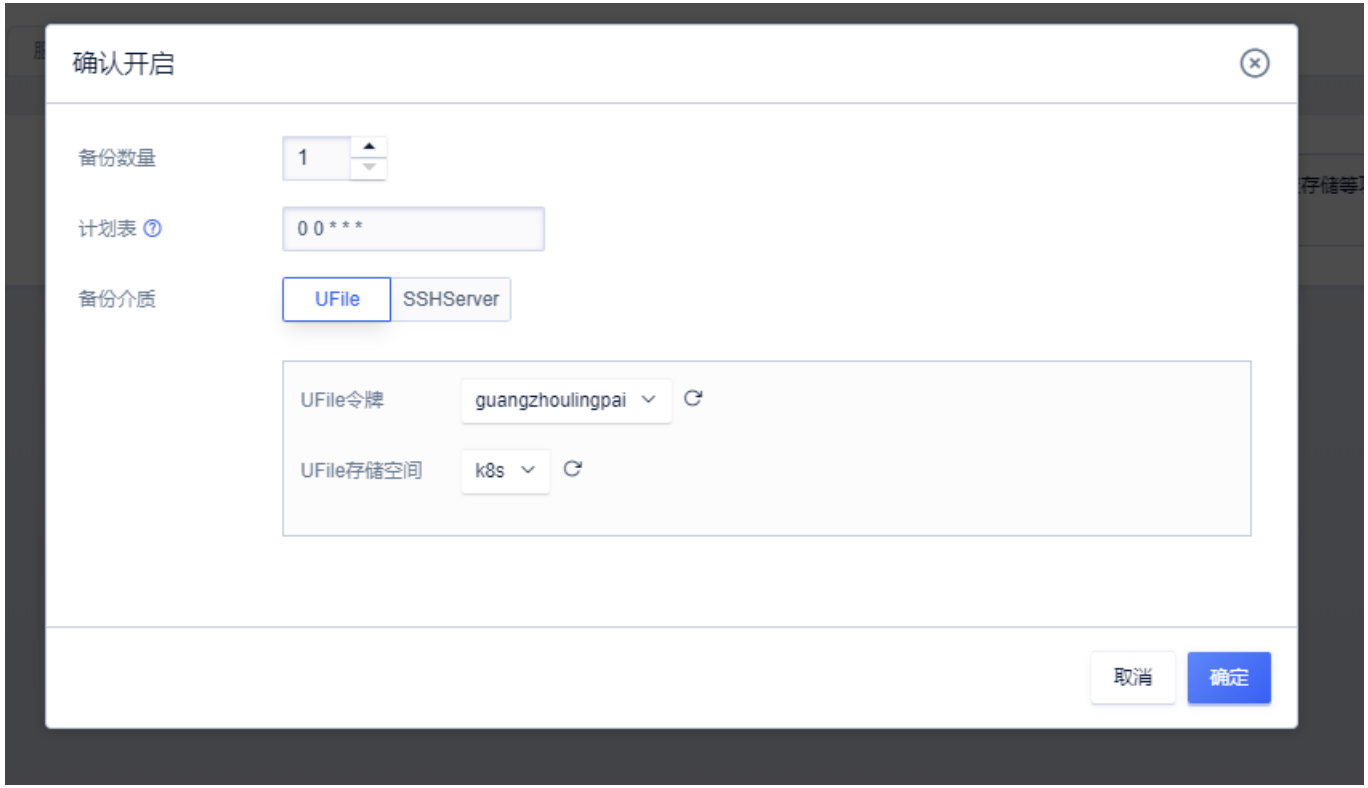
```
kubectl create secret generic uk8s-etcd-backup-cert -n kube-system --from-file /etc/kubernetes/ssl/etcd.pem --from-file /etc/kubernetes/ssl/etcd-key.pem
```

2. 备份操作

开启ETCD备份，备份系统会根据创建的设定进行备份数据留存，如下图



1. 选择希望保留的备份数量，备份系统会根据该数字保持存储空间内ETCD备份数量的上限。
2. 添加定时备份时间，这里使用的时间表语法可以参考本篇 **4. 针对计划表语法说明**。
3. 备份介质我们提供了 UFile 和 SSHServer 两种介质。
4. 选择 UFile 需要创建对应的 UFile 令牌和存储空间，**UFile令牌需要赋予上传、下载、删除、文件列表。**
5. 如使用 SSHServer 需要填写具体的可以执行SSH的主机，并确保填写的账号拥有该主机目录的读写权限。



3. 恢复操作

本篇仅讨论 ETCD 的恢复操作，如需恢复 APIServer、Controller Manager 等其他 K8S 核心组件，请参考UK8S 核心组件故障恢复。

⚠ ETCD恢复操作，建议您仔细阅读文档后谨慎操作。

当误删除 UK8S 集群 Master 节点时，集群恢复较为麻烦，需要联系 UK8S 技术人员进行支持；对于节点故障而非误删节点的情况，处理起来相对简单一些。下面分别进行说明。

3.1 Master 节点未删除，但节点存在故障

3.1.1 ETCD 集群仍然可用

此时，出现故障的 ETCD 节点数量小于总数的一半（UK8S 的 ETCD 节点数量默认为 3，所以此时出现故障的 ETCD 节点数量为 1 ）。这种情况下，不需要借助 ETCD 备份即可进行集群恢复，操作步骤如下：

1. 首先对出现故障的节点进行修复，如需重装系统，则建议安装 CentOS 7.6 64 位操作系统或者联系技术支持人员安装 UK8S 定制镜像；
2. 登录到一个健康节点（本例使用节点 10.23.17.200），进行如下操作从 ETCD 集群中删除故障节点；

```
# 注意后续操作中需要替换相关参数，与你当前集群相匹配

# 配置 ETCDCTL 相关参数
export ETCDCTL_API=3
export ETCDCTL_CACERT=/etc/kubernetes/ssl/ca.pem
export ETCDCTL_CERT=/etc/kubernetes/ssl/etcd.pem
export ETCDCTL_KEY=/etc/kubernetes/ssl/etcd-key.pem
```

```
# 替换 IP 地址为你的 ETCD 集群节点 IP
export ETCDCTL_ENDPOINTS=10.23.17.200:2379,10.23.207.11:2379,10.23.97.19:2379

# 执行如下命令查看集群状态
etcdctl endpoint health

# 输出如下，可以看到节点 10.23.97.19 处于故障状态
...
10.23.17.200:2379 is healthy: successfully committed proposal: took = 15.028244ms
10.23.207.11:2379 is healthy: successfully committed proposal: took = 15.712076ms
10.23.97.19:2379 is unhealthy: failed to commit proposal: context deadline exceeded
Error: unhealthy cluster

# 执行以下命令查看节点 ID
etcdctl member list

# 输出如下，可以获知故障节点 10.23.97.19 的 ID 为 45b4ced6b6a33ef5，etcd name 为 etcd2
1d3f61116d4f3128, started, etcd3, https://10.23.207.11:2380, https://10.23.207.11:2379
45b4ced6b6a33ef5, started, etcd2, https://10.23.97.19:2380, https://10.23.97.19:2379
8a190c41d92119cb, started, etcd1, https://10.23.17.200:2380, https://10.23.17.200:2379

# 执行以下命令删除故障节点
etcdctl member remove 45b4ced6b6a33ef5
```

3. 登录到一个健康节点（本例使用节点 10.23.17.200），拷贝文件到已完成修复的故障节点（本例中是 10.23.97.19）；

```
# 拷贝 etcd 相关程序
scp /usr/bin/etcd /usr/bin/etcdctl 10.23.97.19:/usr/bin
# 拷贝 etcd 配置文件
scp -r /etc/etcd 10.23.97.19:/etc/etcd
# 拷贝 kubernetes 配置文件和 etcd 证书
scp -r /etc/kubernetes 10.23.97.19:/etc/kubernetes
# 拷贝etcd 服务文件
scp /usr/lib/systemd/system/etcd.service 10.23.97.19:/usr/lib/systemd/system/etcd.service
```

4. 登录到完成修复的故障节点(本例中是 10.23.97.19),修改 etcd 相关配置;

```
# 备份原配置文件
cp /etc/etcd/etcd.conf /etc/etcd/etcd.conf.bak

# 清空原配置
echo '' >/etc/etcd/etcd.conf

# 设置参数
# 替换为故障节点的 etcd name
export FAILURE_ETCD_NAME=etcd2
# 替换为使用的健康节点的 IP
export NORMAL_ETCD_NODE_IP=10.23.17.200
# 替换为故障节点的 IP
export FAILURE_ETCD_NODE_IP=10.23.97.19
# 执行以下命令生成新配置
```

```
cat /etc/etcd/etcd.conf.bak | while read LINE; do
if [[ $LINE == "ETCD_INITIAL_CLUSTER="* ]]; then
echo $LINE >>/etc/etcd/etcd.conf
elif [[ $LINE == "ETCD_INITIAL_CLUSTER_STATE="* ]]; then
echo 'ETCD_INITIAL_CLUSTER_STATE="existing"' >>/etc/etcd/etcd.conf
elif [[ $LINE == "ETCD_NAME="* ]]; then
echo 'ETCD_NAME='$FAILURE_ETCD_NAME >>/etc/etcd/etcd.conf
else
echo $LINE | sed "s/$NORMAL_ETCD_NODE_IP/$FAILURE_ETCD_NODE_IP/g" >>/etc/etcd/etcd.conf
fi
done

# 如有遗留数据则删除旧的 etcd 数据
rm -rf /var/lib/etcd
# 新建数据目录
mkdir -p /var/lib/etcd/default.etcd
```

5. 登录到一个健康节点(本例使用节点 10.23.17.200),将故障节点重新加入 etcd 集群;

```
# 注意后续操作中需要替换相关参数,与你当前集群相匹配
export ETCDCTL_API=3
export ETCDCTL_CACERT=/etc/kubernetes/ssl/ca.pem
export ETCDCTL_CERT=/etc/kubernetes/ssl/etcd.pem
export ETCDCTL_KEY=/etc/kubernetes/ssl/etcd-key.pem
# 替换 IP 地址为你的 etcd 集群节点 IP
```

```
export ETCDCTL_ENDPOINTS=10.23.17.200:2379,10.23.207.11:2379,10.23.97.19:2379
# 替换为故障节点的 etcd name
export FAILURE_ETCD_NAME=etcd2
# 替换为故障节点的 IP
export FAILURE_ETCD_NODE_IP=10.23.97.19
# 执行以下命令将故障节点重新加入 etcd 集群
etcdctl member add $FAILURE_ETCD_NAME --peer-urls=https://$FAILURE_ETCD_NODE_IP:2380
```

6. 登录到故障节点，启动 etcd；

```
# 执行以下命令启动 etcd，如无报错则启动成功
systemctl enable --now etcd
# 启动成功后，可以参照步骤 2 查看 etcd 集群状态。
```

3.1.2 ETCD 集群不可用

etcd 集群不可用即有一半及以上的 ETCD 节点出现故障。极端情况下，如果所有节点都因故无法正常登入系统，则需联系支持人员尝试从故障节点中恢复出 ETCD 和 UK8S 相关的文件或者依据数据库数据重新构建相关文件；这里仅考虑至少有一个 ETCD 节点可以正常登入系统的情况，此时可使用 UK8S ETCD 备份插件保存的数据进行集群恢复，操作步骤如下：

1. 保留一个可以正常登录的节点不作处理，对其它故障节点进行修复，如需重装系统，则建议安装 CentOS 7.6 64 位操作系统或者联系技术支持人员安装 UK8S 定制镜像；
2. 如果修复故障节点时未造成文件系统数据丢失，则可跳过该步骤，否则应登录到保留节点（本例为节点 10.23.166.234）拷贝文件到已完成修复的故障节点（本例中是 10.23.172.172，10.23.95.6）；

```
# 拷贝 etcd 相关程序
scp /usr/bin/etcd /usr/bin/etcdctl 10.23.172.172:/usr/bin
```

```
scp /usr/bin/etcd /usr/bin/etcdctl 10.23.95.6:/usr/bin
# 拷贝 etcd 配置文件
scp -r /etc/etcd 10.23.172.172:/etc/etcd
scp -r /etc/etcd 10.23.95.6:/etc/etcd
# 拷贝 kubernetes 配置文件和 etcd 证书
scp -r /etc/kubernetes 10.23.172.172:/etc/kubernetes
scp -r /etc/kubernetes 10.23.95.6:/etc/kubernetes
# 拷贝etcd 服务文件
scp /usr/lib/systemd/system/etcd.service 10.23.172.172:/usr/lib/systemd/system/etcd.service
scp /usr/lib/systemd/system/etcd.service 10.23.95.6:/usr/lib/systemd/system/etcd.service
```

3. 如果修复故障节点时未造成文件系统数据丢失，则可跳过该步骤，否则应在完成步骤 2 后分别登录到故障节点（本例中是 10.23.172.172，10.23.95.6）修改 ETCD 相关配置，以下为其中一个故障节点（本例是10.23.172.172）的操作，其它故障节点应执行相同操作（注意修改参数）；

```
# 备份原配置文件
cp /etc/etcd/etcd.conf /etc/etcd/etcd.conf.bak

# 清空原配置
echo '' >/etc/etcd/etcd.conf

# 设置参数
# 替换为故障节点的 etcd name，这里 10.23.172.172 对应 etcd2
export FAILURE_ETCD_NAME=etcd2
# 替换为保留节点的 IP
export RETAIN_ETCD_NODE_IP=10.23.166.234
# 替换为故障节点的 IP
```

```
export FAILURE_ETCD_NODE_IP=10.23.172.172
# 执行以下命令生成新配置
cat /etc/etcd/etcd.conf.bak | while read LINE; do
if [[ $LINE == "ETCD_INITIAL_CLUSTER="* ]]; then
echo $LINE >>/etc/etcd/etcd.conf
elif [[ $LINE == "ETCD_NAME="* ]]; then
echo 'ETCD_NAME='$FAILURE_ETCD_NAME >>/etc/etcd/etcd.conf
else
echo $LINE | sed "s/$RETAIN_ETCD_NODE_IP/$FAILURE_ETCD_NODE_IP/g" >>/etc/etcd/etcd.conf
fi
done
```

4. 在所有 ETCD 节点上执行以下命令；

```
# 停止 etcd 服务
systemctl stop etcd
# 如有遗留数据则删除旧的 etcd 数据
rm -rf /var/lib/etcd
# 新建数据目录
mkdir /var/lib/etcd
```

5. 从 UFile 或备份服务器获取 ETCD 备份文件并上传至所有 ETCD 节点，分别在各节点上执行以下命令从备份文件恢复 ETCD 数据（注意参数要匹配节点信息）；

```
# 本例备份文件已经保存到 /root/uk8s-f1wymalx-backup-etcd-3.3.17-2020-02-11T03-56-12.db.tar.gz 路径
# 解压缩获取备份数据
```

```
tar zxvf uk8s-f1wymalx-backup-etcd-3.3.17-2020-02-11T03-56-12.db.tar.gz
# 从备份数据恢复，注意调整 ETCD_NAME 和 NODE_IP 匹配节点信息
export ETCD_NAME=etcd2
export NODE_IP=10.23.172.172
export ETCDCTL_API=3
etcdctl --name=$ETCD_NAME --endpoints=https://$NODE_IP:2379 --cert=/etc/kubernetes/ssl/etcd.pem --key=/etc/kubernetes/ssl/etcd-key.pem --
cacert=/etc/kubernetes/ssl/ca.pem --initial-cluster-token=etcd-cluster --initial-advertise-peer-urls=https://$NODE_IP:2380 --initial-
cluster=etcd1=https://10.23.95.6:2380,etcd2=https://10.23.172.172:2380,etcd3=https://10.23.166.234:2380 --data-dir=/var/lib/etcd/default.etcd/ snapshot
restore uk8s-f1wymalx-backup-etcd-3.3.17-2020-02-11T03-56-12.db
```

6. 在所有 ETCD 节点完成步骤 5 后，在所有 ETCD 节点执行以下命令启动 ETCD；

```
# 注意最开始执行该命令时有可能 hang 住，这是正常现象
# 继续在其它节点上执行该命令，当超过半数的节点都启动后该命令就会正常退出
systemctl enable --now etcd
```

7. 登录任一 etcd 节点查看集群状态；

```
export ETCDCTL_API=3
export ETCDCTL_CACERT=/etc/kubernetes/ssl/ca.pem
export ETCDCTL_CERT=/etc/kubernetes/ssl/etcd.pem
export ETCDCTL_KEY=/etc/kubernetes/ssl/etcd-key.pem

# 替换 IP 地址为你的 etcd 集群节点 IP
export ETCDCTL_ENDPOINTS=10.23.95.6:2379,10.23.172.172:2379,10.23.166.234:2379
```

```
# 执行如下命令查看集群状态
etcdctl endpoint health
```

3.2 误删除etcd节点

3.2.1 etcd 集群仍然可用

此时，误删除的 etcd 节点数量小于总数的一半（UK8S 的 etcd 节点数量默认为 3，所以此时误删的 etcd 节点数量为 1 ），etcd 集群仍然可以正常访问。这种情况下，不需要借助 etcd 备份即可进行集群恢复，操作步骤如下：

1. 联系技术支持人员根据误删除节点的信息创建一个配置和 IP 与被删节点相同的虚拟机；
2. 参照 ***etcd 节点未删除，但节点存在故障*** 章节下 ***etcd 集群仍然可用*** 子章节，将步骤 1 中创建的虚拟机作为 *已修复的故障节点* 进行集群恢复；
3. etcd 集群修复完成后，联系技术支持人员更改数据库中误删除的虚拟机的 UHost ID 为步骤 1 中创建的虚拟机的 UHost ID。

3.2.2 etcd 集群不可用

etcd 集群不可用即有一半及以上的 etcd 节点被删除。极端情况下，如果所有节点都被删除，则需联系支持人员尝试依据数据库数据重新构建相关文件；这里仅考虑至少有一个 etcd 节点可以正常登入系统的情况，此时可使用 UK8S etcd 备份插件保存的数据进行集群恢复，操作步骤如下：

1. 联系技术支持人员根据所有误删除的节点的信息创建配置和 IP 与被删节点一一对应的虚拟机；
2. 参照 ***etcd 节点未删除，但节点存在故障*** 章节下 ***etcd 集群不可用*** 子章节，将步骤 1 中创建的虚拟机作为 *已修复的故障节点* 进行集群恢复；
3. etcd 集群修复完成后，联系技术支持人员更改数据库中误删除的虚拟机的 UHost ID 为步骤 1 中创建的虚拟机的 UHost ID。

4. 针对计划表语法说明

针对计划表语法使用和CronTab一致的语法，下面列举几种常用语法，详细语法请参考链接

Crontab格式(前5位为时间选项,这里我们只用到了前5位)

```
<分钟> <小时> <日> <月份> <星期> <命令>
```

每天一次,0点0分执行

```
0 0 * * *
```

每周一次,0点0分执行

```
0 0 * * 0
```

每月一次,0点0分执行

```
0 0 1 * *
```

配置自定义DNS服务

本文主要介绍如何在UK8S集群中,使用自定义的DNS服务。

从Kubernetes 1.11起,CoreDNS取代kube-dns成为默认的DNS方案,UK8S目前支持的Kubernetes版本>=1.11,因此本文主要介绍如何修改CoreDNS的配置以达到使用自定义DNS服务的目的。

简介

CoreDNS是一个模块化、插件式的DNS服务器，其配置文件信息保存在Corefile内。UK8S集群管理员可通过修改ConfigMap，来配置自定义DNS服务。

在UK8S中，CoreDNS的默认Corefile配置信息如下:

```
apiVersion: v1
kind: ConfigMap
data:
Corefile: |
.:53 {
errors
health
ready
template ANY AAAA {
rcode NOERROR
}
kubernetes cluster.local in-addr.arpa ip6.arpa {
pods insecure
upstream
fallthrough in-addr.arpa ip6.arpa
ttl 30
}
prometheus :9153
forward . /etc/resolv.conf {
policy sequential
}
```

```
cache 30
loop
reload
loadbalance
}
metadata:
name: coredns
namespace: kube-system
```

Corefile的配置信息包含以下CoreDNS的插件：

- errors: 错误日志会以标准输出的方式打印到容器日志；
- health: CoreDNS的健康状况；
- kubernetes: kubernetes插件是CoreDNS中用来替代kube-dns的模块，将service的域名转为IP的工作由该插件完成，其中常用的参数作用如下：
  - **pods POD-MODES**: 用于设置Pod的A记录处理模式，如**1-2-3-4.ns.pod.cluster.local. in A 1.2.3.4**。**pods disabled**为默认值，表示不为pod提供dns记录；**pods insecure**会一直返回对应的A记录，而不校验ns；**pods verified**会校验ns是否正确，如果该ns下有对应的pod，则返回A记录。
  - **upstream [ADDRESS..]**: 定义用于解析外部hosts的上游dns服务器。如果不指定，则CoreDNS会自行处理，例如使用后面会介绍到的proxy插件。
  - **fallthrough [ZONE..]**: 如果指定此选项，则DNS查询将在插件链上传递，该插件链可以包含另一个插件来处理查询，例如**in-addr.arpa**。
- prometheus: CoreDNS对外暴露的监控指标，默认为http://localhost:9153/metrics。
- forward [from to]: 任何不属于Kubernetes集群内部的域名，其DNS请求都将指向forword指定的 DNS 服务器地址。**from**一般为"."，代表所有域名，**to**可以为多个，如 111.114.114.114 8.8.8.8。需要注意的是，新版本的CoreDNS已forward插件替代proxy插件，不过使用方法是一致的，如果你的集群是proxy，建议改为forward插件，性能更好。
- reload: 允许自动加载变化了的Corefile，建议配置，这样CoreDNS可以实现热更新。

- template: 模版可以使用go的语法编写动态响应，这里我们写死了AAAA类的查询固定返回NOTERROR以规避这个issues中的问题

其他选项的意义请查看Kubernetes官方文档，下面我们举例说明下如何修改ConfigMap。

示例

为特殊域配置DNS服务器

假设我们有一个ucloudk8s的服务域，自建的私有DNS Server地址为10.9.10.8，则集群管理员可以执行kubectl edit configmap/coredns -n kube-system中添加如下所示的一段规则，这是个独立的**Server Block**,我们可以在Corefile里面为不同的域配置不同的**Server Block**。

```
ucloudk8s.com:53 {
errors
template ANY AAAA {
rcode NOERROR
}
cache 30
forward . 10.9.10.8
}
```

并且，我们不希望使用/etc/resolv.conf里配置的DNS服务器作为上游服务器，而是指向自建的DNS Server，只需要直接修改之前提到的upstream和proxy选项即可

```
upstream 10.9.10.8
forward . 172.16.0.1
```

修改完毕后的configmap如下：

```
apiVersion: v1
kind: ConfigMap
data:
Corefile: |
.:53 {
errors
health
kubernetes cluster.local in-addr.arpa ip6.arpa {
pods insecure
upstream 10.9.10.8
fallthrough in-addr.arpa ip6.arpa
}
prometheus :9153
forward . 10.9.10.8
cache 30
loop
reload
loadbalance
}
ucloudk8s.com:53 {
errors
template ANY AAAA {
rcode NOERROR
}
```

```
cache 30
forward . 10.9.10.8
}
metadata:
name: coredns
namespace: kube-system
```

验证

我们在同VPC下的某台云主机中(请勿在UK8S Node节点中操作)安装一个DNS服务器，来验证自定义DNS是否正常工作。

通过Docker安装DNS服务器

```
docker run -d -p 53:53/tcp -p 53:53/udp --cap-add=NET_ADMIN --name dns-server andyshinn/dnsmasq:2.75
```

进入容器

```
docker exec -it dns-server /bin/sh
```

配置上游DNS服务器

```
vi /etc/resolv.dnsmasq


nameserver 114.114.114.114
```

```
nameserver 8.8.8.8
```

配置本地解析规则

```
vi /etc/dnsmasqhosts

110.110.110.110 baidu.com
```

修改dnsmasq配置文件，指定使用上述两个我们自定义的配置文件，修改下述两个选项，并重启容器。

```
vi /etc/dnsmasq.conf

resolv-file=/etc/resolv.dnsmasq
addn-hosts=/etc/dnsmasqhosts

docker restart dns-server
```

修改CoreDNS的configmap，添加如下规则。

```
baidu.com:53{
errors
template ANY AAAA {
rcode NOERROR
```

```
}
cache 30
forward . 10.9.10.8(测试时需修改成你的DNS地址)
}
```

进入K8S在容器内测试，使用dig命名测试，可以看到解析地址为110.110.110.110，符合预期。

```
bash-4.4# dig baidu.com

; <<>> DiG 9.12.3-P4 <<>> baidu.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39140
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;baidu.com. IN A

;; ANSWER SECTION:
baidu.com. 5 IN A 110.110.110.110

;; Query time: 4 msec
```

```
;; SERVER: 172.17.0.2#53(172.17.0.2)
;; WHEN: Mon May 27 09:11:50 UTC 2019
;; MSG SIZE rcvd: 63
```

常见问题

修改了CoreFile，但解析不成功

首先确认CoreFile是否包含了reload插件，如果没有包含，需要添加reload，并且重建CoreDNS，使其可以加载到最新的DNS规则。

其次，通过''kubectl logs COREDNS-POD-NAME -n kube-system''查看CoreDNS日志，确认CoreDNS是否正常工作，以及DNS配置是否加载成功。

如果依然不能正常工作，在容器或Pod内执行''dig @YOUR-DNS-SERVER-ADDRESS YOUR-DOMAIN''，确认您的DNS服务器是否正常工作。

如以上操作皆无效，请联系UCloud技术支持协助处理。

改变 CoreDNS 部署方式

UK8S 默认为 CoreDNS 服务配置了 Pod 反亲和性，两个服务副本需要分散在两个不同节点上。这会导致您在开启节点自动扩缩容之后，集群缩容只能缩到最小两节点。

如果您希望集群可以缩容至单节点，可以按如下方式修改 CoreDNS 部署方式。

1. 执行 `kubectl edit deploy coredns -n kube-system`
2. 将 podAntiAffinity 配置按如下方式替换

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
name: coredns
namespace: kube-system
spec:
template:
spec:
affinity:
podAntiAffinity:
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 100
podAffinityTerm:
topologyKey: kubernetes.io/hostname
labelSelector:
matchExpressions:
- key: k8s-app
operator: In
values:
- kube-dns
```

Docker 和 Containerd 容器引擎

在 UK8S 中新建 V1.19 以后的 Kubernetes 版本，其容器引擎已经默认更换为 Containerd，1.19 之前的Kubernetes 则为 Docker，这两种容器引擎都有各自的命令工具来管理镜像和容器。

为了让低于 1.19 的 UK8S 集群能够升级到 1.19+，UK8S 也推出容器运行时升级功能，帮助客户平滑升级。

请注意，采用 containerd 运行时的节点，通过 containerd 调用 CNI 网络插件进行 Pod IP 分配，如另行安装 docker，将会覆盖 containerd 的相关配置文件，导致分配Pod IP失败。因此**请勿自行另外安装 docker，避免导致节点不可用。**

Docker 及 Cotainerd 命令对比

镜像管理命令对比

镜像相关功能	Docker	Containerd
显示本地镜像列表	docker images	crictl images
下载镜像	docker pull	crictl pull
上传镜像	docker push	无
删除本地镜像	docker rmi	crictl rmi
查看镜像详情	docker inspect IMAGE-ID	crictl inspecti IMAGE-ID

容器管理命令对比

容器相关功能	Docker	Containerd
显示容器列表	docker ps	crictl ps
创建容器	docker create	crictl create

启动容器	docker start	crictl start
停止容器	docker stop	crictl stop
删除容器	docker rm	crictl rm
查看容器详情	docker inspect	crictl inspect
attach	docker attach	crictl attach
exec	docker exec	crictl exec
logs	docker logs	crictl logs
stats	docker stats	crictl stats

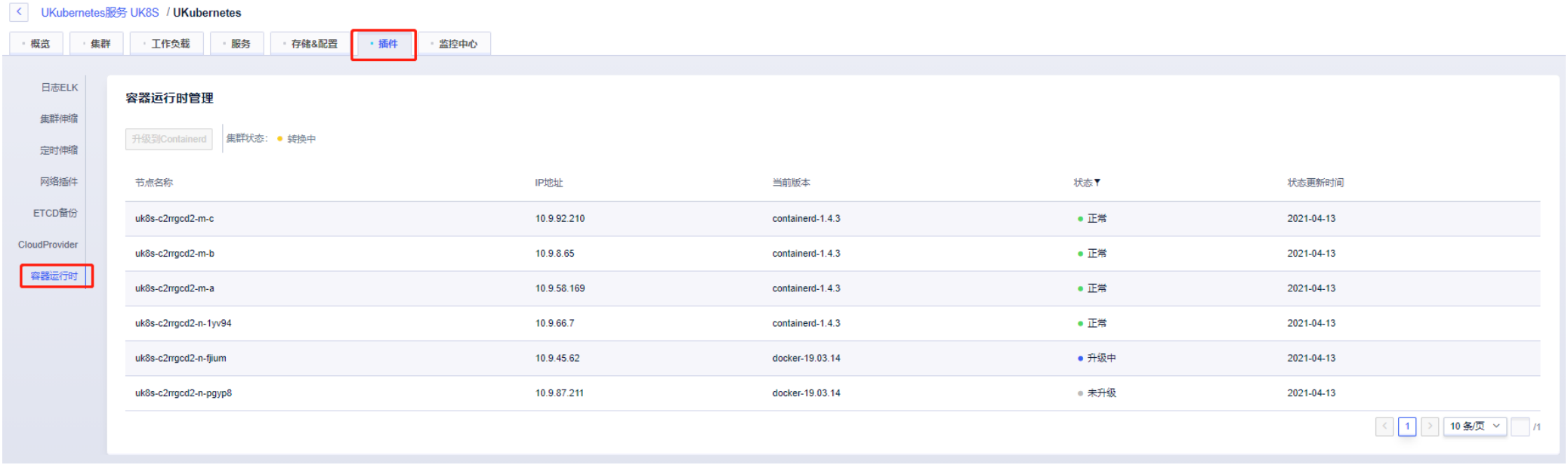
Pod命令对比

POD 相关功能	**Docker**	**Containerd**
显示 POD 列表	无	crictl pods
查看 POD 详情	无	crictl inspectp
运行 POD	无	crictl runp
停止 POD	无	crictl stopp

容器运行时升级

在控制台集群管理页面的插件页，选择「容器进行时」分页（该分页目前仅针对从老版本升级至 1.19 及以上版本的集群展示），即可对集群升级前存量节点进行容器运行时升级，集群升级后新增节点默认为 containerd 运行时。

运行时升级将会按照先升级所有 Master 节点、再升级 Node 节点的顺序逐台进行，升级时节点将处于不可用状态，其上的 Pod 将会被驱逐到其他节点，业务可能会受影响。建议在执行此操作时保证集群资源充裕，并在业务低峰期时操作。



Containerd 常见问题

1. containerd-shim进程泄露

- 目前发现使用docker的k8s集群，有可能会遇到containerd-shim进程泄露，尤其是在频繁创建删除Pod或者Pod频繁重启的情况下。
- 此时甚至有可能导致docker inspect某个容器卡住进一步导致kubelet PLEG timeout 异常。此时以coredns Pod为例，说明如何查看是否存在containerd-shim进程泄露。如下示例，正常情况下，一个containerd-shim进程会有一个实际工作的子进程。子进程消失时，containerd-shim进程会自动退出。如果containerd-shim进程没有子进程，则说明存在进程泄露。

遇到containerd-shim进程泄露的情况，可以按照如下方式进行处理

- 确认泄露的进程id，执行kill pid。注意，此时无需加-9参数，一般情况下简单的kill就可以处理。
- 确认containerd-shim进程退出后，可以观察docker及kubelet是否恢复正常。
- 注意，由于kubelet此时可能被docker卡住，阻挡了很多操作的执行，当docker恢复后，可能会有大量操作同时执行，导致节点负载瞬时升高，可以在操作前后分别重启一遍kubelet及docker。

```
[root@xxxx ~]# docker ps |grep coredns-8f7c8b477-snmpq
ee404991798d uhub.service.ucloud.cn/uk8s/coredns "/coredns -conf /etc..." 4 minutes ago Up 4 minutes k8s_coredns_coredns-8f7c8b477-snmpq_kube-system_26da4954-3d8e-4f67-902d-28689c45de37_0
b592e7f9d8f2 uhub.service.ucloud.cn/google_containers/pause-amd64:3.2 "/pause" 4 minutes ago Up 4 minutes k8s_POD_coredns-8f7c8b477-snmpq_kube-system_26da4954-3d8e-4f67-902d-28689c45de37_0
[root@xxxx ~]# ps aux |grep ee404991798d
root 10386 0.0 0.2 713108 10628 ? Sl 11:12 0:00 /usr/bin/containerd-shim-runc-v2 -namespace moby -id ee404991798d70cb9c3c7967a31b3bc2a50e56b072f2febf604004f5a3382ce2 -address /run/containerd/containerd.sock
```

```
root 12769 0.0 0.0 112724 2344 pts/0 S+ 11:16 0:00 grep --color=auto ee404991798d
[root@xxxx ~]# ps -ef |grep 10386
root 10386 1 0 11:12 ? 00:00:00 /usr/bin/containerd-shim-runc-v2 -namespace moby -id
ee404991798d70cb9c3c7967a31b3bc2a50e56b072f2febf604004f5a3382ce2 -address /run/containerd/containerd.sock
root 10421 10386 0 11:12 ? 00:00:00 /coredns -conf /etc/coredns/Corefile
root 12822 12398 0 11:17 pts/0 00:00:00 grep --color=auto 10386
```

2. 1.19.5 集群kubelet连接containerd失败

在1.19.5集群中，有可能出现节点not ready的情况，查看kubelet日志，发现有大量Error while dialing dial unix:///run/containerd/containerd.sock相关的日志。这是1.19.5版本中一个已知bug，当遇到containerd重启的情况下，kubelet会失去与containerd的连接，只有重启kublet才能恢复。具体可以查看k8s官方issue。
如果您遇到此问题，重启kubelet即可恢复。同时目前uk8s集群已经不支持创建1.19.5版本的集群，如果您的集群版本为1.19.5，可以通过升级集群的方式，升级到1.19.10。

kube-proxy模式选择

kube-proxy是kubernetes中的关键组件，其主要功能是在Service和其后端Pod之间（Endpoint）进行负载均衡。kube-proxy 有三种运行模式，每种都有不同的实现技术：userspace、iptables或者IPVS。

userspace模式由于性能问题已经不推荐使用。这里主要介绍iptables和IPVS两种模式的比较及选择。

如何选择

- 对于集群规模较大，特别是Service数量可能超过1000的，推荐选择IPVS。（详见后续测试数据）

- 对于集群规模中等，Service数量不多的，推荐选择iptables。
- 如果客户端会出现大量并发短链接，目前建议选择iptables，原因见下方备注。

备注：在使用IPVS模式的kubernetes集群中进行滚动更新，期间如果有一个客户端在短时间内（两分钟）内发送大量短链接，客户端端口会被复用，导致node收到的来自于该客户端的请求报文网络五元组相同，触发IPVS复用Connection，有可能导致报文被转发到了一个已经销毁的Pod上，导致业务异常。

官方issue：https://github.com/kubernetes/kubernetes/issues/81775

如何切换

请参考kube-proxy模式切换

iptables模式

iptables是一个Linux内核功能，是一个高效的防火墙，并提供了数据包处理和过滤方面的能力。它可以在核心数据包处理管线上用Hook挂接一系列的规则。iptables模式中kube-proxy在NAT pre-routing Hook中实现它的NAT和负载均衡功能。这种方法简单有效，依赖于成熟的内核功能，并且能够和其它跟 iptables 协作的应用融洽相处。

IPVS模式

IPVS是一个用于负载均衡的Linux内核功能。IPVS模式下，kube-proxy使用IPVS负载均衡代替了iptables。IPVS的设计思路就是用来为大量服务进行负载均衡的，它有一套优化过的API，使用优化的查找算法，而不是从列表中查找规则，在大规模场景下相对IPVS性能更好。

模式对比

无论是iptables模式还是IPVS模式，转发性能都与Service及对应的Endpoint数量有关，原因是Node上iptables或IPVS转发规则的数量与svc和ep的数目成正比。

IPVS和iptables转发性能主要差异体现在TCP三次握手连接建立的过程，因此在大量短连接请求的场景下，两种模式的性能差异尤为突出。

在Service和Endpoint的数量较少的情况下(Service数十到数百，Endpoint数百到数千)，iptables模式转发性能要略优于IPVS。

随着Service和Endpoint的数量逐渐提升，iptables模式转发性能明显下降，IPVS模式转发性能则相对稳定。

Service数量1000左右，Endpoint数量到20000左右时，iptables模式转发性能开始低于IPVS，随着Service和Endpoint的数量继续增大（Service数千，Endpoint数万），IPVS模式性能略微下降，iptables模式性能则大幅下降。

测试用例

我们使用了2台Node作为测试节点，一台节点KubeProxy使用iptables模式，记为N1；另一台KubeProxy使用IPVS模式，记为N2。

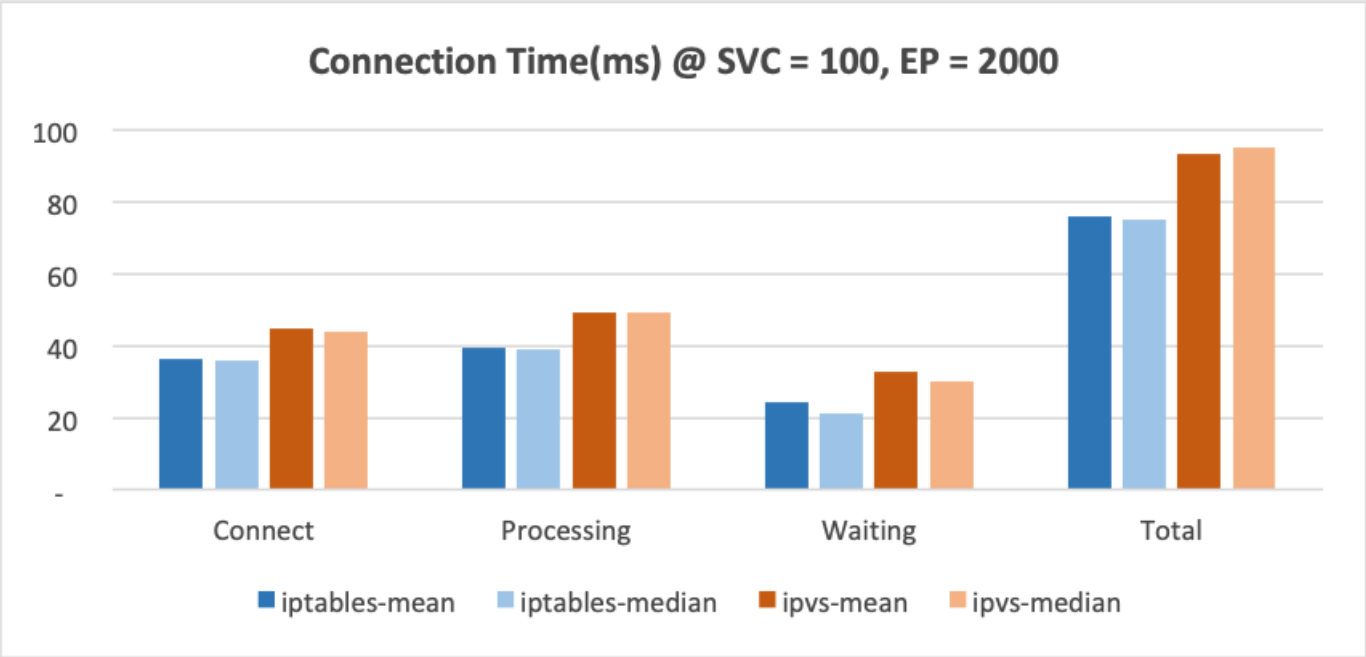
在N1和N2上准备好压测客户端ab，并发连接数1000，一共需要完成10000次短连接请求。

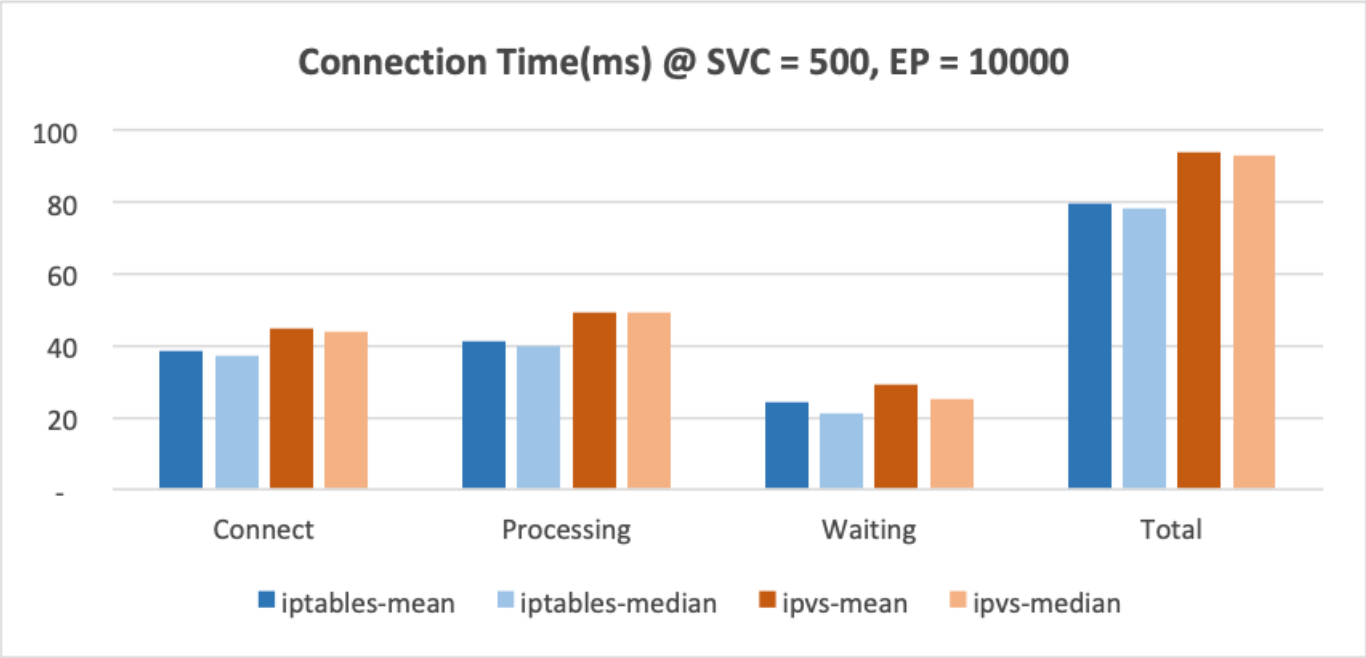
在N1和N2上分别但不同时执行测试命令，观察ab返回的结果：

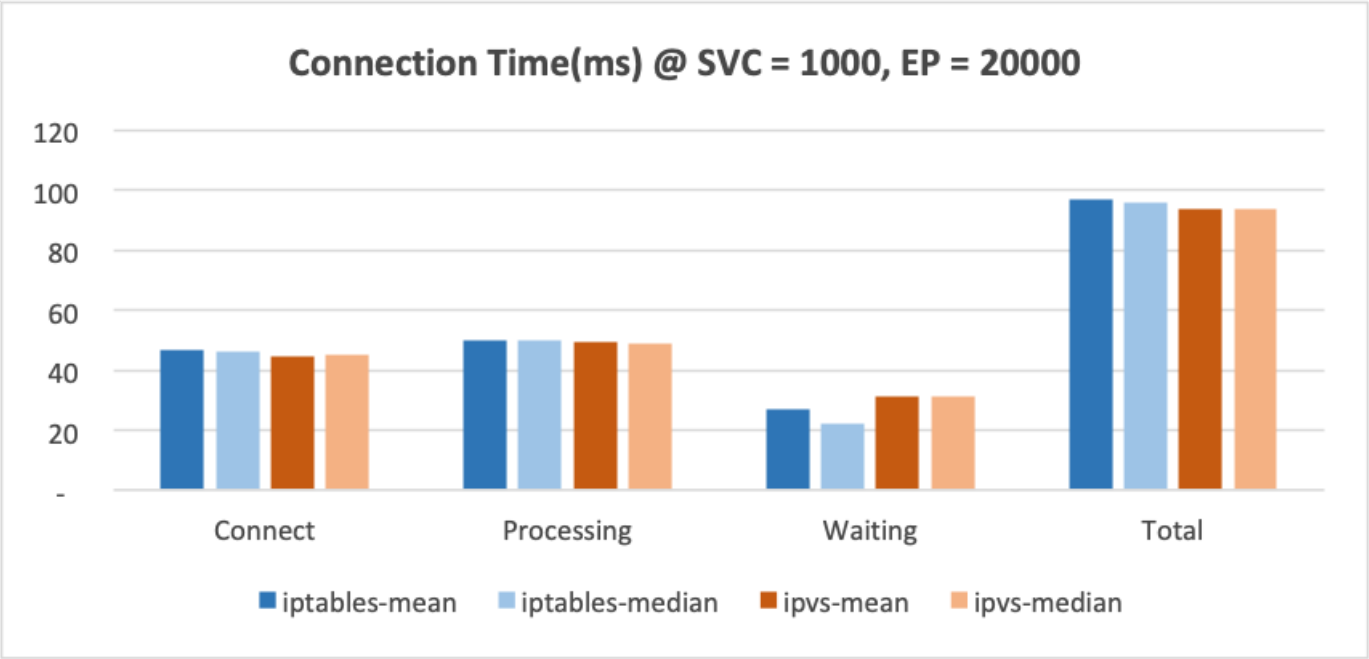
```
Connection Times (ms)
min mean[+/-sd] median max
Connect: 1 38 8.4 38 59
Processing: 10 41 9.7 40 67
Waiting: 1 28 9.0 28 56
Total: 51 79 7.5 78 101
```

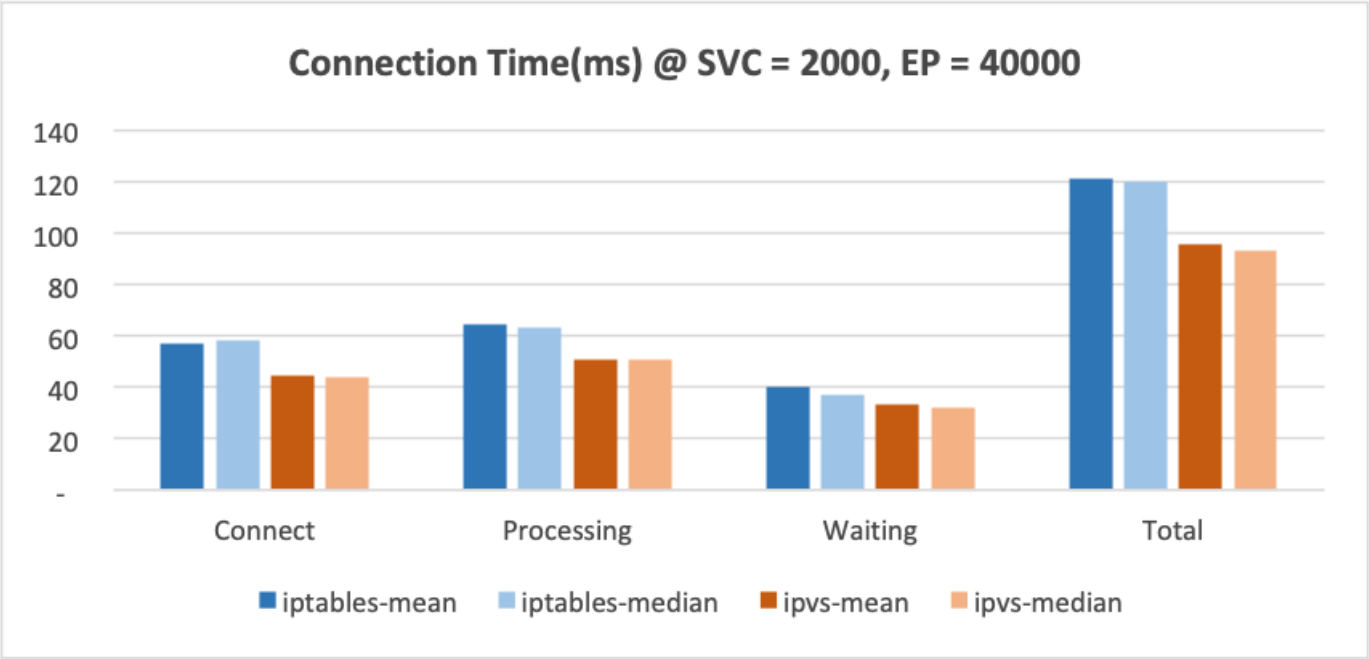
不断变化Service数量,100，500，1000，2000，3000，4000，观察结果采集数据。

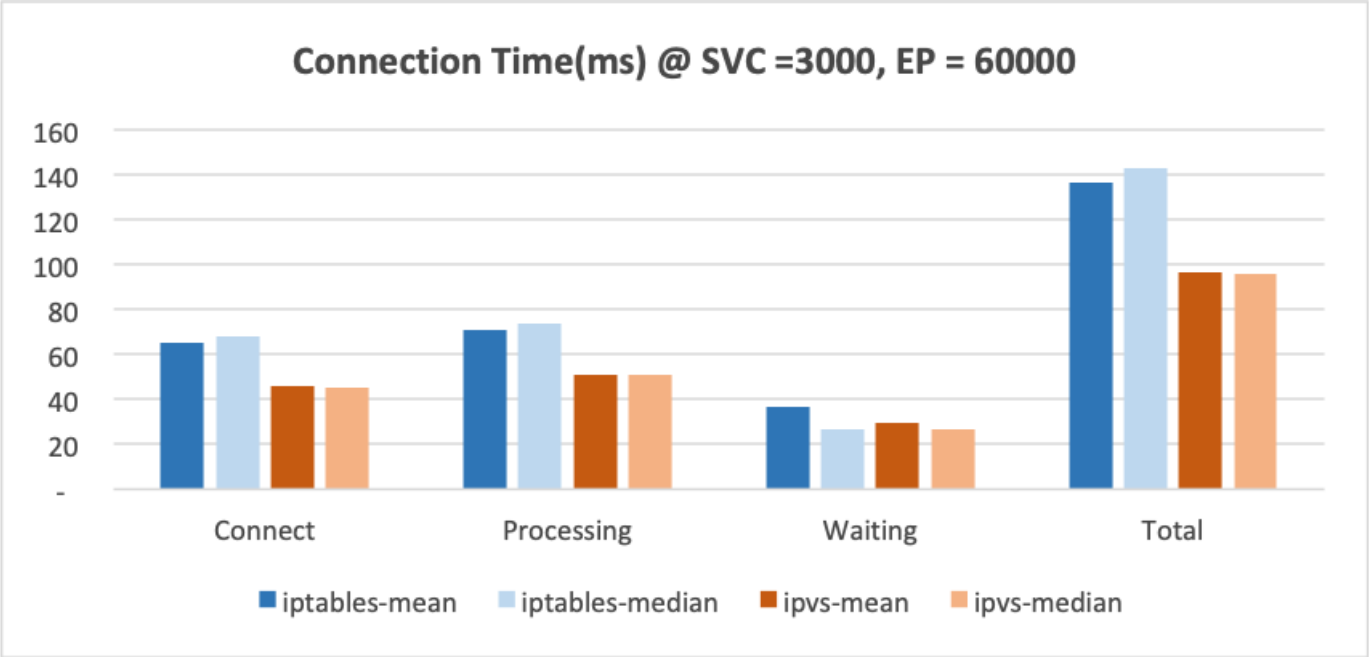
以下为UK8S团队针对IPVS和iptables进行的性能测试数据。

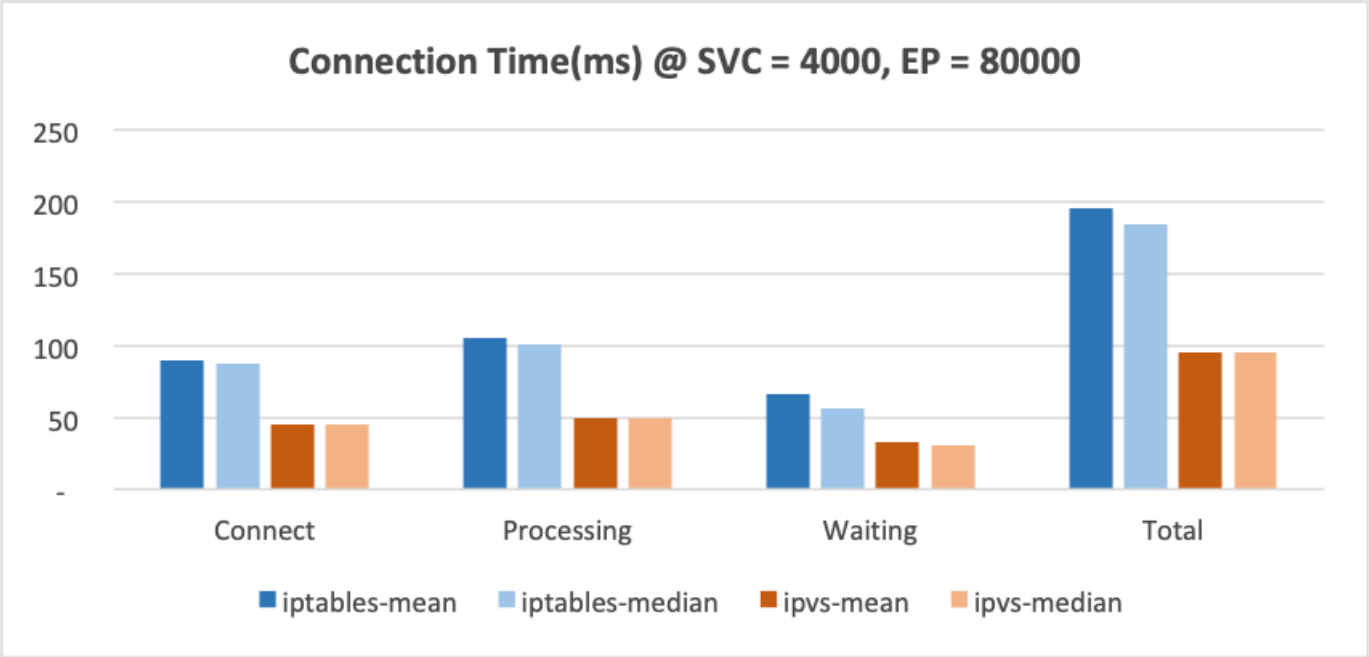












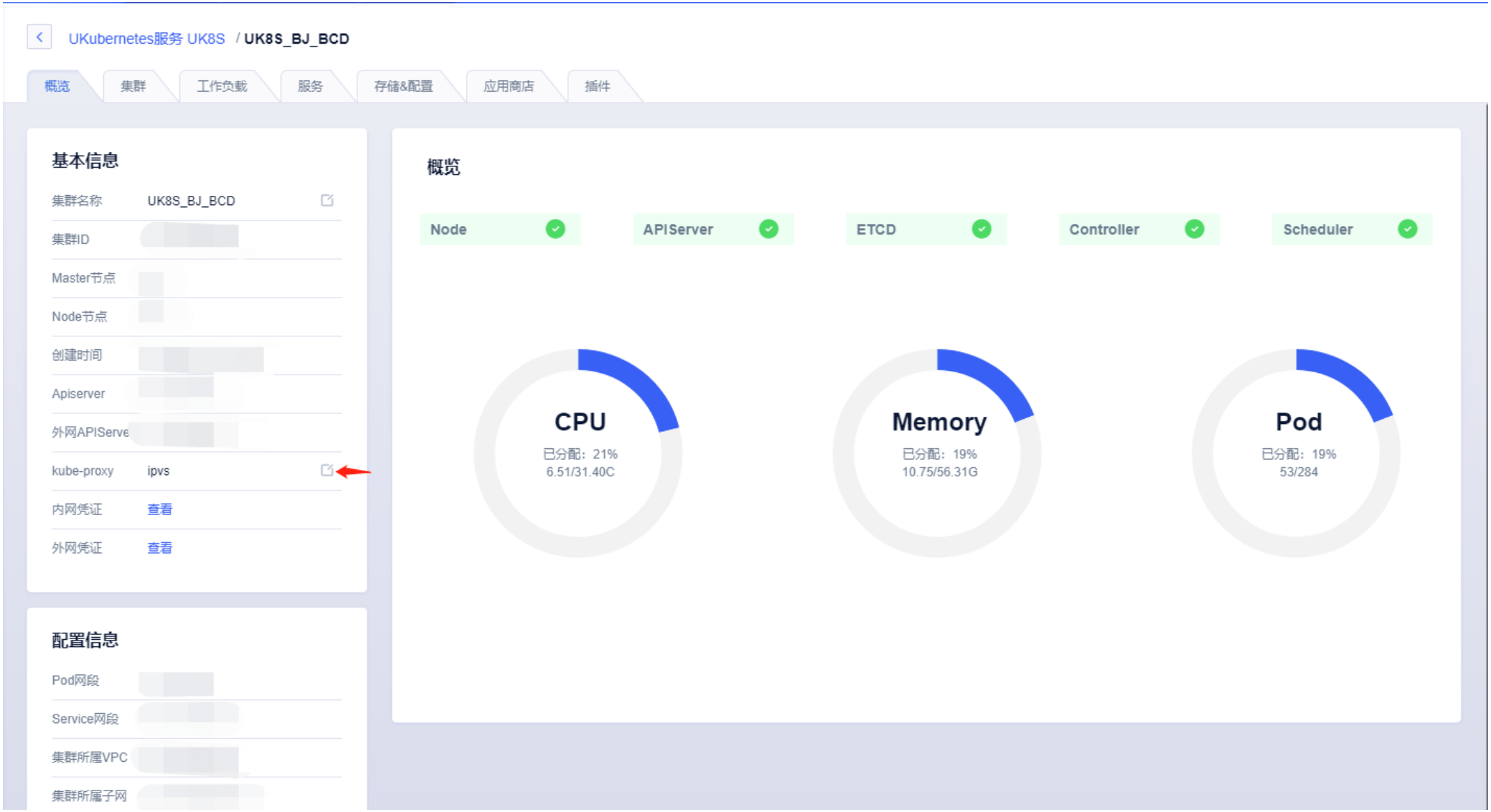
可以看出，在Service数量为100和500时，iptables转发性能要优于IPVS；Service数量达到1000时，两者大体持平；Service数量继续增大，IPVS的性能优势则越发明显。

kube-proxy模式切换

在UK8S集群创建时我们会对集群创建生成默认的kube-proxy模式，在集群中新增的Node节点都会依赖这个集群默认kube-proxy模式进行新增，您可以参考以下操作进行集群kube-proxy模式的切换。

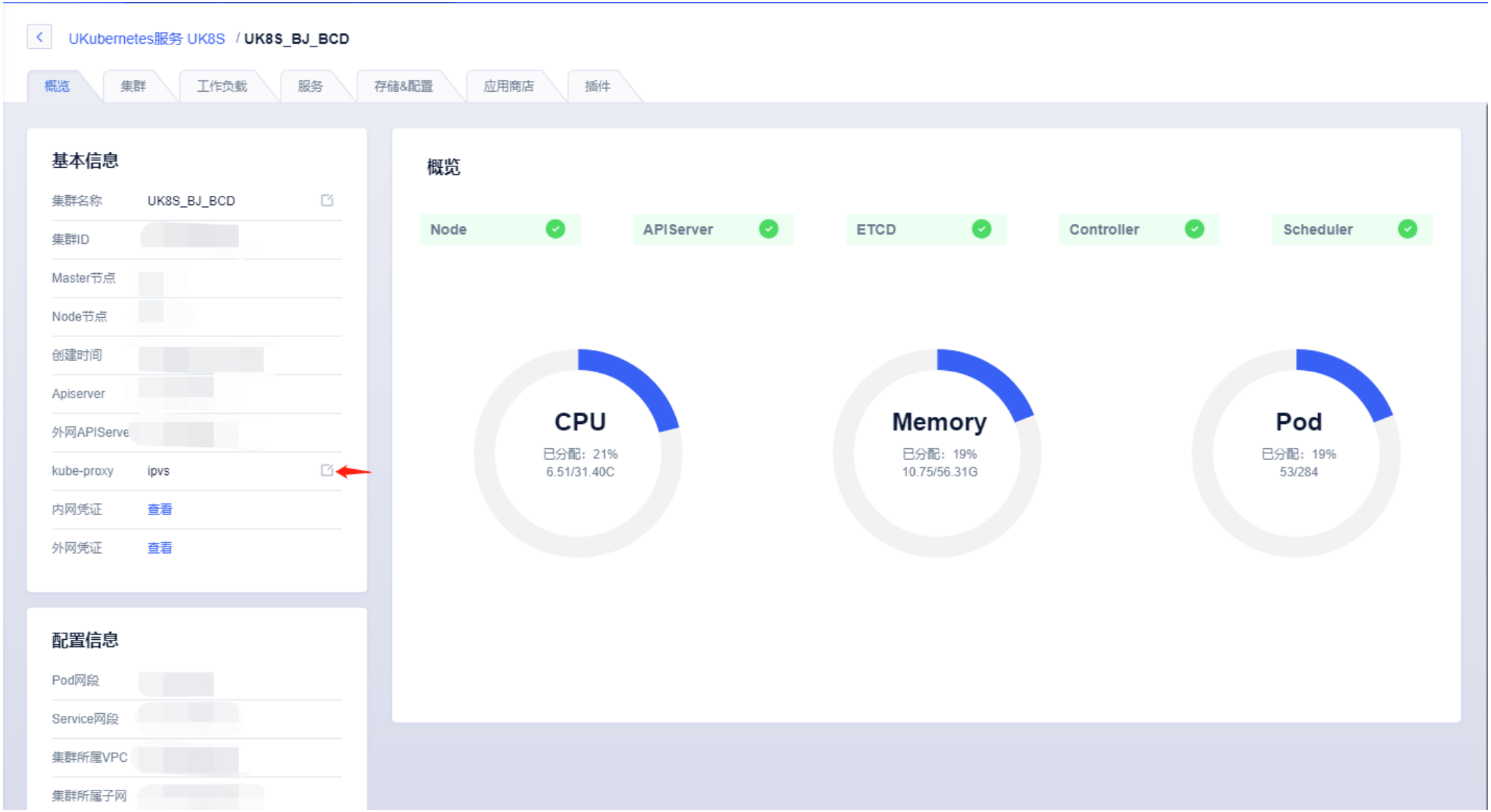
kube-proxy模式切换操作

1. UK8S集群是您通过UCloud控制台创建的独立的集群，UK8S集群创建时会根据您选择的kube-proxy模式进行创建。已如下集群为例，集群kube-proxy模式默认是IPVS。



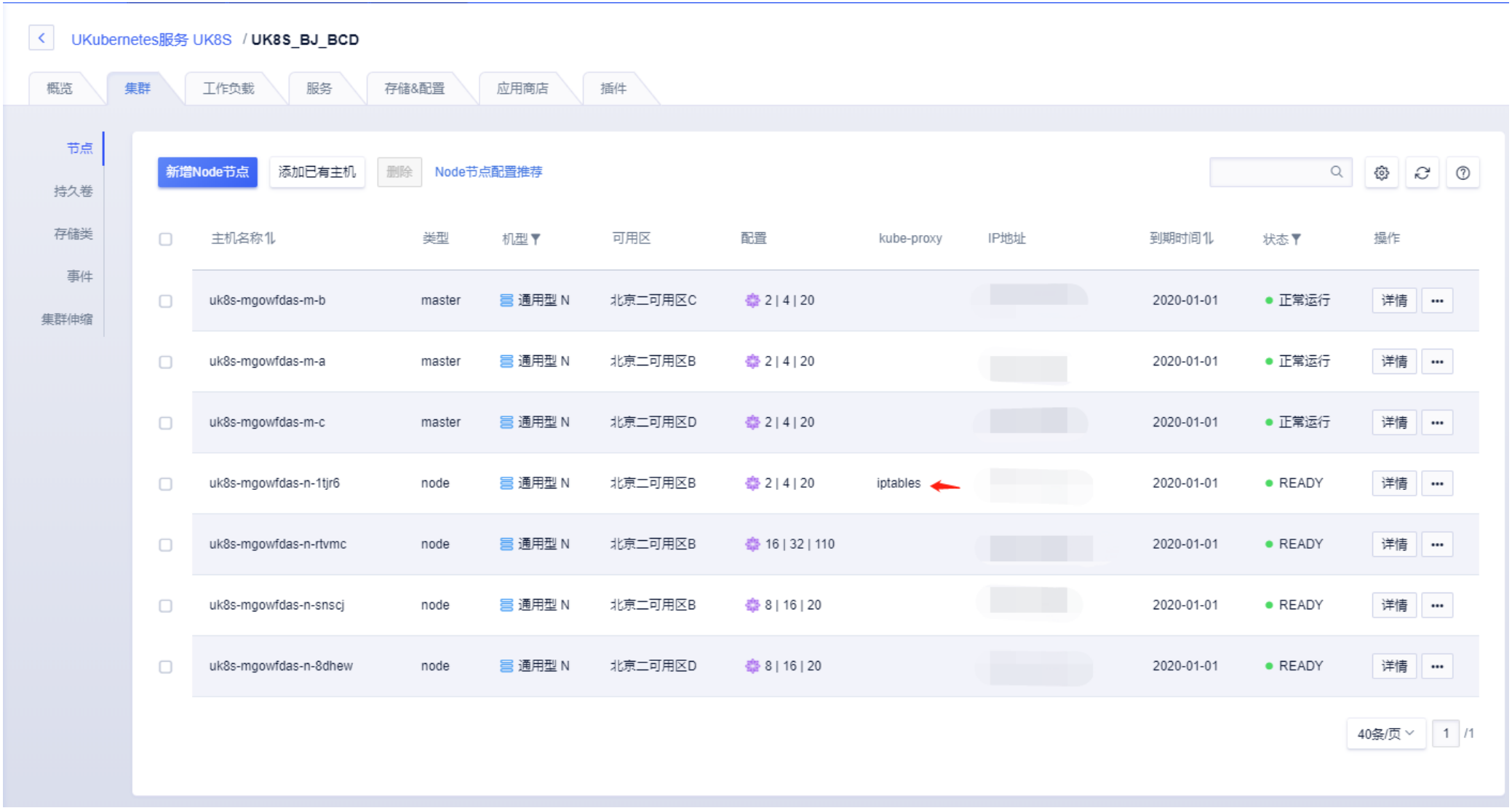
2. 创建完成后可以在集群概览中对kube-proxy模式进行修改，此修改只针对修改后的节点添加，对于集群原有节点不会修改。我们在此修改为iptables模式。

3. 用户可以在集群节点中查看到节点的kube-proxy模式，针对集群的kube-proxy模式进行查看，如没有可以在页面的设置中点击展示该字段。



注:2019年12月17日前创建的节点默认没有节点的kube-proxy信息。

4. 切换集群节点的kube-proxy模式我们采用新增新节点，新节点ready后删除一个老节点的循环操作进行集群全部Node节点的kube-proxy模式切换。切换过程中老节点的Pod会调度到新的节点上。循环操作直至集群全部节点替换完成。



注意事项

1. 请注意您的集群是否是多可用区的集群，如果是的话新增的Node和原Node最好保持在同一可用区。
2. 如果您是针对Node节点的IP已经做了ACL策略或者对节点IP有要求，或是对于Node节点付费存在异议，请不要使用此方案，进入每个Node节点进行kubelet参数修改，如需要此种方式调整集群的kube-proxy模式请联系UCloud售后团队获取操作文档。
3. K8S集群的节点最好统一使用一种kube-proxy模式，如果进行了切换，请将集群所有Node进行切换。
4. Master节点不在K8S调度范围内，不用进行切换。

node-problem-detector 资源修改

由于 node-problem-detector 默认给的配置资源太低，可能会导致以下问题

- 系统出现僵尸进程
- node 节点读 IO 变高
- npd 进程会出现报错：`Timeout when running plugin "./config/plugin/network_problem.sh": state -signal`

需要调整 node-problem-detector 的资源配置，操作步骤如下

1. 获取 DaemonSet 的名称

```
kubectl get ds -n kube-system |grep node-problem-detector
node-problem-detector 1 1 1 1 1 <none> 236d
```

找到 DaemonSet 的名称，一般叫做 node-problem-detector，或者叫做 ack-node-problem-detector-daemonset

2. 编辑 DaemonSet 资源，修改 resources 配置

通过 edit 命令修改资源的配置，执行下面命令之后，会进入 vim 模式

```
kubectl edit ds node-problem-detector -n kube-system
```

修改 resource 对应的内容为 100m 和 100Mi ，保存退出即可，node-problem-detector 对应的 pod 会自动重启

```
resources:
limits:
cpu: 100m
memory: 100Mi
requests:
cpu: 100m
memory: 100Mi
```

3. 验证 pod 正常运行

```
kubectl get pod -n kube-system |grep node-problem-detector
```

确认 pod 正常运行

常见问题:

1. 修改会影响我们的应用服务吗?

不会，node-problem-detector 的作用是检查 node 节点是否存在异常，修改 node-problem-detector 不会影响应用服务

2. 哪些集群需要修改?

如果集群是在 2022 年 3月 12 日之前创建的，且在集群的 应用中心 -> NPD节点监控 开启了Node-Problem-Detector节点监控插件，就有可能出现这个问题。

我们已经在近期的发布中修改了该问题，后面新创建的集群不会出现。

UK8S 核心组件故障恢复

1. APIServer、Controller Manager、Scheduler 组件的故障恢复

APIServer、Controller Manager、Scheduler 是 Kubernetes 的核心管理组件，在 UK8S 集群中，默认配置三台 Master 节点，每台 Master 节点上均部署安装了这些核心组件，各个组件通过负载均衡对外提供服务，确保集群的高可用。

当某个组件出现故障时，请逐台登录三台 Master 节点，通过 `systemctl status ${PLUGIN_NAME}` 确认组件状态，如组件不可用，可通过以下步骤进行恢复：

```
# 将一台健康 Master 节点的内网 IP 配置为环境变量，便于从健康节点拷贝相关文件
export IP=10.23.17.200

# 从健康节点拷贝 APIServer、Controller Manager、Scheduler 组件二进制安装包
## 1.16 及以下 UK8S 版本，K8S 组件统一安装在 hyperkube 文件中
scp root@IP:/usr/local/bin/hyperkube /usr/local/bin/hyperkube
## 1.17 及以后 UK8S 版本，K8S 组件以独立二进制文件形式安装
scp root@IP:/usr/local/bin/{kube-apiserver,kube-controller-manager,kube-scheduler} /usr/local/bin/

# 拷贝 APIServer、Controller Manager、Scheduler 组件服务文件
scp root@IP:/usr/lib/systemd/system/{kube-apiserver.service,kube-controller-manager.service,kube-scheduler.service} /usr/lib/systemd/system/

# 拷贝 APIServer、Controller Manager、Scheduler 组件配置文件
```

```
scp root@IP:/etc/kubernetes/{apiserver,controller-manager,kube-scheduler.conf} /etc/kubernetes/

# 拷贝 kubectl 二进制文件
scp root@IP:/usr/local/bin/kubectl /usr/local/bin/kubectl

# 拷贝 kubeconfig
scp -r root@IP:~/.kube ~/

# 修改 APIServer 配置参数
vim /etc/kubernetes/apiserver # 将 advertise-address 参数配置为故障节点 IP

# 启用服务
systemctl enable --now kube-apiserver kube-controller-manager kube-scheduler

# 配置 APIServer 负载均衡器的内外网 IP（仅在开启外网 APIServer 功能情况下需要配置外网 IP）
scp root@IP:/etc/sysconfig/network-scripts/ifcfg-lo:internal /etc/sysconfig/network-scripts/ifcfg-lo:internal
scp root@IP:/etc/sysconfig/network-scripts/ifcfg-lo:external /etc/sysconfig/network-scripts/ifcfg-lo:external
systemctl restart network
```

2. Kubelet、Kube-proxy 的故障恢复

Kubelet、Kube-proxy 部署在每个 Master / Node 节点上，分别负责节点注册及流量转发。

注:2020.6.12 以前创建的 UK8S 集群中,Master 节点上默认不安装 Kubelet,不能通过 kubectl get node 显示。

```
# 将一台健康节点的内网 IP 配置为环境变量,便于从健康节点拷贝相关文件
export IP=10.23.17.200

# 从健康节点拷贝 Kubelet、Kube-proxy 组件二进制安装包
## 1.16 及以下 UK8S 版本,K8S 组件统一安装在 hyperkube 文件中,如在上一环节中已执行过此操作可忽略
scp root@IP:/usr/local/bin/hyperkube /usr/local/bin/hyperkube
## 1.17 及以后 UK8S 版本,K8S 组件以独立二进制文件形式安装
scp root@IP:/usr/local/bin/{kubelet,kube-proxy} /usr/local/bin/

# 准备目录
mkdir -p /opt/cni/net.d
mkdir -p /opt/cni/bin
mkdir -p /var/lib/kubelet

# 配置文件拷贝、服务文件
scp root@$IP:/etc/kubernetes/{kubelet,kubelet.conf,kube-proxy.conf,ucloud} /etc/kubernetes/
scp root@$IP:/usr/lib/systemd/system/{kubelet.service,kube-proxy.service} /usr/lib/systemd/system/
scp root@$IP:/etc/kubernetes/set-conn-reuse-mode.sh /etc/kubernetes/
scp root@$IP:/etc/rsyslog.conf /etc/
scp root@$IP:/opt/cni/bin/{cnivpc,loopback,host-local} /opt/cni/bin/
scp root@$IP:/opt/cni/net.d/10-cnivpc.conf /opt/cni/net.d/
```

```
# 修改配置参数
# 修改 --node-ip、--hostname-override 为待修复节点 IP
# 修改 --node-labels 中 topology.kubernetes.io/zone、failure-domain.beta.kubernetes.io/zone 为待修复节点可用区(cn-bj2-02)
# 修改 --node-labels 中 UHostID、node.uk8s.ucloud.cn/resource_id 为待修复节点资源 ID(uhost-xxxxxxxx)
vim /etc/kubernetes/kubelet

# 禁用swap
swapoff -a

# 启用服务
systemctl enable --now kubelet kube-proxy
```

3. 容器引擎的恢复

3.1 Docker 容器引擎

```
# 将一台健康 Master 节点的内网 IP 配置为环境变量,便于从健康节点拷贝相关文件
export IP=10.23.17.200

# 准备目录
mkdir -p /data/docker
rm -rf /var/lib/docker
ln -s /data/docker /var/lib/docker
```

```
# 安装包下载及安装
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-19.03.14-3.el7.x86_64.rpm
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/containerd.io-1.4.3-3.2.el7.x86_64.rpm
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-cli-19.03.14-3.el7.x86_64.rpm
yum install *.rpm -y

# 拷贝配置及服务文件
scp root@$IP:/usr/lib/systemd/system/docker.service /usr/lib/systemd/system/
scp root@$IP:/etc/docker/daemon.json /etc/docker/

# 启用服务
systemctl enable --now docker
```

3.2 Containerd 容器引擎

```
# 将一台健康 Master 节点的内网 IP 配置为环境变量,便于从健康节点拷贝相关文件
export IP=10.23.17.200

# 准备目录
mkdir -p /etc/containerd
mkdir -p /data/containerd
mkdir -p /data/log/pods
ln -s /data/containerd /var/lib/containerd
```

```
ln -s /data/log/pods /var/log/pods

# 安装包下载及安装
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/containerd.io-1.4.3-3.2.el7.x86_64.rpm
yum install containerd.io-1.4.3-3.2.el7.x86_64.rpm

# 拷贝配置文件
scp root@$IP:/etc/containerd/{config.toml,containerd.toml} /etc/containerd/
scp root@$IP:/usr/lib/systemd/system/containerd.service /usr/lib/systemd/system/
scp root@$IP:/usr/local/bin/crictl /usr/local/bin/
scp root@$IP:/etc/crictl.yaml /etc/

# 启用服务
systemctl start containerd
```

概述

镜像仓库概述

镜像库用于存储、分发Docker镜像，你可以将您的应用打包成Docker镜像，并Push到镜像仓库，需要使用时，再从镜像仓库Pull到本地。

UK8S支持的镜像库类型

UK8S支持各类公有及私有镜像库，如：

1. DockerHub官方镜像库(默认)
2. UHub(UCloud镜像仓库)
3. 用户私有镜像库

使用帮助

1. UHub产品文档

2. 在UK8S中使用UHub

在UK8S中使用UHub

本文主要说明如何在UK8S中使用UHub或你自己搭建的私有容器镜像来创建应用。

Kubernetes支持为Pod指定Secret来拉取私有仓库中的镜像，下面我们演示如何使用从UHub中拉取镜像来创建一个Nginx应用；

一、生成秘钥Secret

镜像仓库中关于账号与密码的信息是配置在Kuernetes中的Secret资源中，可以创建一个docker-registry类型的Secret保存账号密码；使用以下命令创建Secret，注意将其中的大写字母值替换为你自己的信息；

- MYSECRET：Secret资源名称，改为自己定义名称；这里以mysecret为例
- YOUR_UCLOUD_USERNAME@EMAIL.COM：镜像仓库登陆账号，可以是控制台账号，也可以是镜像仓库独立账号
- YOUR_UHUB_PASSWORD：镜像仓库登陆账号对应的密码

```
# kubectl create secret docker-registry <MYSECRET> \
--docker-server=uhub.service.ucloud.cn \
--docker-username=<YOUR_UCLOUD_USERNAME@EMAIL.COM> \
--docker-password=<YOUR_UHUB_PASSWORD>
```

其中uhub.service.ucloud.cn是UCloud提供的仓库域名，如果自己搭建的仓库，请调整为自己的仓库域名；

二、查看生成的秘钥信息

我们看到一个名为mysecret的秘钥已经生成；

```
# kubectl get secret
NAME TYPE DATA AGE
default-token-sfv7s kubernetes.io/service-account-token 3 8d
mysecret kubernetes.io/dockerconfigjson 1 3h
```

三、在Pod中使用

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
labels:
app: nginx
spec:
containers:
- name: nginx
image: uhub.service.ucloud.cn/ucloud/nginx:1.9.2
imagePullSecrets:
```

```
- name: mysecret # 这里就是上面创建的Secret名称
```

使用上述的yaml文件创建一个Nginx应用

```
# kubectl create -f pod.yml
```

五、查看Pod信息

通知打印的日志，我们可以看到Kubernetes成功地从UHub拉取镜像。

```
# kubectl describe pods/nginx
.....
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Scheduled 1min default-scheduler Successfully assigned default/nginx to 10.25.95.46
Normal Pulling 1min kubelet, 10.25.95.46 pulling image "uhub.service.ucloud.cn/ucloud/nginx:1.9.2"
Normal Pulled 1min kubelet, 10.25.95.46 Successfully pulled image "uhub.service.ucloud.cn/ucloud/nginx:1.9.2"
Normal Created 1min kubelet, 10.25.95.46 Created container
Normal Started 1min kubelet, 10.25.95.46 Started container
```

镜像及镜像仓库常见问题

如何在 UK8S 中 Build 镜像?

UK8S节点Docker配置文件经过修改,且1.19及以上集群默认安装containerd,因此不推荐在UK8S中直接Build镜像。可以通过 部署CI/CD,或者在集群外 UHost 中安装docker进行Build镜像。

怎么在UK8S集群中拉取Uhub以外的镜像?

UK8S使用VPC网络实现内网互通,拉取Uhub镜像不受影响,拉取外网镜像时需要对VPC的子网配置网关,需要在UK8S所在的区域下进入VPC产品,对具体子网配置NAT网关,使集群节点可以通过NAT网关拉取外网镜像,具体操作详见VPC创建NAT网关 。

为什么我的 UHub 登陆失败了?

1. 请确认是在公网还是 UCloud 内网进行登陆的(如果 ping uhub 的 ip 为 106.75.56.109 则表示是通过公网拉取)
2. 如果在公网登陆,请在 UHub 的 Console 页面确认外网访问选项打开
3. 确认是否使用独立密码登陆,UHub 独立密码是和用户绑定的,而不是和镜像库绑定的

UHub 下载失败(慢)

1. ping uhub.service.ucloud.cn（如果ip为106.75.56.109 则表示是通过公网拉取，有限速）
2. curl https://uhub.service.ucloud.cn/v2/ 查看是否通，正常会返回 UNAUTHORIZED 或 301
3. systemctl show --property=Environment docker 查看是否配置了代理
4. 在拉镜像节点执行iftop -i any -f 'host <uhub-ip>'命令，同时尝试拉取 UHub 镜像，查看命令输出（uhub-ip替换为步骤1中得到的ip）
5. 对于公网拉镜像的用户，还需要在 Console 页面查看外网访问是否开启

拉取自建镜像库证书错误

升级为新版Docker之后，可能会遇到自建镜像库签名证书的问题。具体报错信息为x509: certificate relies on legacy Common Name field, use SANs or temporarily enable Common Name matching with GODEBUG=x509ignoreCN=0

该报错为go在1.11版本引入的一个新特性，表明https证书不符合SANs标准，该特性在1.15版本默认启用，而GODEBUG=x509ignoreCN=0 标志也在1.17版本被正式移除。

如果遇到相同的报错，可以按照如下方案进行处理

1. 更换符合标准的签名证书，此方案为推荐方案，并且go版本升级为1.17之后，该方案为唯一可行方案。
2. 针对Docker节点，按照如下步骤临时允许忽略检查，注意此方案仅可作为临时方案使用。
   2.1 修改 /usr/lib/systemd/system/docker.service 在 [Service] 结构下面增加 Environment=GODEBUG=x509ignoreCN=0
   2.2 执行 systemctl daemon-reload
   2.3 执行 systemctl restart docker
3. 针对Containerd节点，按照如下步骤临时允许忽略检查，注意此方案仅可作为临时方案使用。
   3.1 修改 /usr/lib/systemd/system/containerd.service 在 [Service] 结构下面增加 Environment=GODEBUG=x509ignoreCN=0
   3.2 执行 systemctl daemon-reload
   3.3 执行 systemctl restart containerd

推理常用基础镜像

Uhub提供了一系列的推理相关基础容器镜像，用户可以基于这些基础镜像构建自定义镜像，用于部署推理服务。

镜像列表：

镜像名称	tag信息	源仓库
pytorch/pytorch	2.4.0-cuda11.8-cudnn9-runtime 2.4.0-cuda11.8-cudnn9-runtime	pytorch/pytorch
nvida/pytorch	24.07-py3 24.06-py3	nvcr.io/nvidia/pytorch
vllm/vllm-openai	v0.5.3 v0.5.2	vllm/vllm-openai
jupyter/tensorflow-notebook	x86_64-ubuntu-22.04	jupyter/tensorflow-notebook
nvidia/cuda	12.5.1-cudnn-runtime-ubuntu20.04 12.5.1-cudnn-runtime-ubuntu22.04	nvcr.io/nvidia/cuda

如何获取

上述列表中所有镜像已经同步仓库：ucloudai，镜像地址规则如下所示：

```
uhub.service.ucloud.cn/ucloudai/<镜像名称>:tag
```

例如：下载vllm/vllm-openai镜像的v0.5.3版本，可使用如下命令：

```
docker pull uhub.service.ucloud.cn/ucloudai/vllm/vllm-openai:v0.5.3
```

Pod 容忍节点异常时间调整

1. 原理说明

Kubernetes 集群节点处于异常状态之后需要有一个等待时间，才会对节点上的 Pod 进行驱逐。那么针对部分关键业务，是否可以调整这个时间，便于在节点发生异常时及时将 Pod 驱逐并在别的健康节点上重建？

要解决这个问题，我们首先要了解 Kubernetes 在节点异常时驱逐 Pod 的机制。

在 Kubernetes 1.13 及以后的版本中默认开启了 TaintBasedEvictions 及 TaintNodesByCondition 这两个 feature gate，节点及其上 Pod 的生命周期管理将通过节点的 Condition 和 Taint 来进行，Kubernetes 会不断地检查所有节点状态，设置对应的 Condition，根据 Condition 为节点设置对应的 Taint，再根据 Taint 来驱逐节点上的 Pod。

同时在创建 Pod 时会默认为 Pod 添加相应的 tolerationSeconds 参数，指定当节点出现异常（如 NotReady）时 Pod 还将在这个节点上运行多长的时间。

那么，节点发生异常到 Pod 被驱逐的时间，就取决于两个参数：1. 节点实际异常到被判断为不健康的时间；2. Pod 对节点不健康的容忍时间。

Kubernetes 集群中默认节点实际异常到被判断为不健康的时间为 40s，Pod 对节点 NotReady 的容忍时间为 5min，也就是说，节点实际异常 5min40s（340s）后，节点上的 Pod 才会发生驱逐。

2. 调整节点被标记为不健康的时间

ControllerManager 参数 --node-monitor-grace-period 控制了在将一个节点标记为不健康之前允许其无响应的时长上限，该参数默认值为 40s，且必须比 Kubelet 的 nodeStatusUpdateFrequency 参数（Kubelet 向主控节点汇报节点状态的时间间隔）大 N 倍；这里 N 指的是 kubelet 发送节点状态的重试次数。

如需修改该参数，请逐台在三台 **Master** 节点上进行如下操作：

1. 在 ControllerManager 配置文件/etc/kubernetes/controller-manager 中添加参数 --node-monitor-grace-period=20s，将节点被标记为不健康的容忍时间调整为 20s，修改前请做好配置文件备份；
2. 执行 systemctl restart kube-controller-manager 重启 ControllerManager；
3. 执行 systemctl status kube-controller-manager 确认 ControllerManager 状态为 active。

3. 调整 Pod 对节点不健康的容忍时长

在创建 Pod 时，如无特别指定，节点控制器会为 Pod 添加如下污点：

```
tolerations:
- key: "node.kubernetes.io/unreachable"
operator: "Exists"
effect: "NoExecute"
tolerationSeconds: 300
- key: "node.kubernetes.io/not-ready"
operator: "Exists"
effect: "NoExecute"
tolerationSeconds: 300
```

这种自动添加的容忍度意味着在其中一种问题（NotReady / UnReachable）被检测到时 Pod 默认能够继续停留在当前节点运行 5 分钟。

注:当 DaemonSet 中的 Pod 被创建时，针对 unreachable / not-ready 污点自动添加的 NoExecute 的容忍度将不会指定 tolerationSeconds,保证出现相应问题时 DaemonSet 中的 Pod 永远不会被驱逐。

3.1 调整默认容忍时长

Kubernetes 为 Pod 自动添加的针对 unreachable / not-ready 污点的容忍时长由 APIServer 中的相应参数控制,如需修改请逐台在三台 **Master** 节点上进行如下操作:

1. 在 APIServer 配置文件/etc/kubernetes/apiserver 中添加参数 --default-not-ready-toleration-seconds=100 及 --default-unreachable-toleration-seconds=100,将对污点 NotReady:NoExecute 及 Unreachable:NoExecute 的容忍时长(以秒记,默认为 300)调整为 100s,修改前请做好配置文件备份;
2. 执行 systemctl restart kube-apiserver 重启 APIServer
3. 执行 systemctl status kube-apiserver 确认 APIServer 状态为 active。

3.2 调整现有 Pod 容忍时长

以通过 Deployment 创建的 Pod 为例,我们需要通过 kubectl patch 命令修改现有 Deployment 中的 Toleration 参数。

首先,创建 patch 文件 tolerationseconds.yaml,示例如下:

```
spec:
template:
spec:
tolerations:
- key: "node.kubernetes.io/unreachable"
operator: "Exists"
effect: "NoExecute"
```

```
# 调整 Pod 对污点 Unreachable:NoExecute 的容忍时长为 100s
tolerationSeconds: 100
- key: "node.kubernetes.io/not-ready"
operator: "Exists"
effect: "NoExecute"
# 调整 Pod 对污点 NotReady:NoExecute 的容忍时长为 100s
tolerationSeconds: 100
```

再执行 `kubectl patch deploy your-deployment --patch "$(cat tolerationseconds.yaml)"` 命令，对 Deployment 进行修改。修改完成后，会发现该 Deployment 控制的 Pod 中相应的污点容忍时长已经被修改。

⚠ 该操作会引发 Deployment 对所有 Pod 进行重建，请在业务低谷时进行。

4. 参考文档

1. 污点和容忍度
2. kube-apiserver
3. kube-controller-manager

Pod 常见故障处理

在Kubernetes中发布应用时，我们经常会遇到Pod出现异常的情况，如Pod长时间处于Pending状态，或者反复重启，下面介绍下Pod 的各种异常状态及处理思路。

1. 常见错误

状态	状态说明	处理办法
Error	Pod 启动过程中发生错误。	一般是由于容器启动命令、参数配置错误所致，请联系镜像制作者
NodeLost	Pod 所在节点失联。	检查 Pod 所在节点的状态
Unkown	Pod 所在节点失联或其他未知异常。	检查 Pod 所在节点的状态
Pending	Pod 等待被调度。	资源不足等原因导致，通过 kubectl describe 命令查看 Pod 事件
Terminating	Pod 正在被销毁。	可增加 --fore参数强制删除
CrashLoopBackOff	容器退出，Kubelet 正在将它重启。	一般是由于容器启动命令、参数配置错误所致
ErrImageNeverPull	策略禁止拉取镜像。	拉取镜像失败，确认imagePullSecret是否正确
ImagePullBackOff	正在重试拉取。	镜像仓库与集群的网络连通性问题
RegistryUnavailable	连接不到镜像仓库。	联系仓库管理员
ErrImagePull	拉取镜像出错。	联系仓库管理员，或确认镜像名是否正确
RunContainerError	启动容器失败。	容器参数配置异常

PostStartHookError	执行 postStart hook 报错。	postStart 命令有误
NetworkPluginNotReady	网络插件还没有完全启动。	cni 插件异常，可检查cni状态

2. 常见命令

当我们发现 Pod 处于 上述状态时，可以使用以下命令来快速定位问题：

1. 获取 Pod 状态

```
kubectl -n ${NAMESPACE} get pod -o wide
```

2. 查看 Pod 的 yaml 配置

```
kubectl -n ${NAMESPACE} get pod ${POD_NAME} -o yaml
```

3. 查看 Pod 事件

```
kubectl -n ${NAMESPACE} describe pod ${POD_NAME}
```

4. 查看 Pod 日志

```
kubectl -n ${NAMESPACE} logs ${POD_NAME} ${CONTAINER_NAME}
```

5. 登录 Pod

```
kubectl -n ${NAMESPACE} exec -it ${POD_NAME} /bin/bash
```

3. UK8S对Node上发布的容器有限制吗？如何修改？

UK8S 为保障生产环境 Pod 的运行稳定，每个 Node 限制了 Pod 数量为 110 个，用户可以通过登陆 Node 节点"vim /etc/kubernetes/kubelet.conf" 修改 `maxpods:110`，然后执行 `systemctl restart kubelet` 重启 kubelet 即可。

4. 为什么我的容器一起来就退出了？

1. 查看容器log，排查异常重启的原因
2. pod是否正确设置了启动命令，启动命令可以在制作镜像时指定，也可以在pod配置中指定
3. 启动命令必须保持在前台运行，否则k8s会认为pod已经结束，并重启pod。

5. Docker 如何调整日志等级

1. 修改/etc/docker/daemon.json 文件，增加一行配置"debug": true
2. systemctl reload docker 加载配置，查看日志
3. 如果不再需要查看详细日志，删除debug配置，重新reload docker即可

6. 为什么节点已经异常了，但是 Pod 还处在 Running 状态

1. 这是由于k8s的状态保护造成的，在节点较少或异常节点很多的情况下很容易出现

2. 具体可以查看文档 https://kubernetes.io/zh/docs/concepts/architecture/nodes/#reliability

7. 节点宕机了 Pod 一直卡在 Terminating 怎么办

1. 节点宕机超过一定时间后(一般为 5 分钟),k8s 会尝试驱逐 pod,导致 pod 变为 Terminating 状态
2. 由于此时 kubelet 无法执行删除pod的一系列操作,pod 会一直卡在 Terminating
3. 类型为 daemonset 的 pod,默认在每个节点都有调度,因此 pod 宕机不需要考虑此种类型 pod,k8s 也默认不会驱逐该类型的 pod
4. 类型为 depolyment 和 replicaset 的 pod,当 pod 卡在 termanting 时,控制器会自动拉起对等数量的 pod
5. 类型为 statefulset 的 pod,当 pod 卡在 termanting 时,由于 statefulset 下属的 pod 名称固定,必须等上一个 pod 彻底删除,对应的新 pod 才会被拉起,在节点宕机情况下无法自动拉起恢复
6. 对于使用 udisk-pvc 的 pod,由于 pvc 无法卸载,会导致新起的 pod 无法运行,请按照本文 pvc 相关内容(#如何查看pvc对应的udisk实际挂载情况),确认相关关系

8. Pod 异常退出了怎么办?

1. `kubectl describe pods pod-name -n ns` 查看 pod 的相关 event 及每个 container 的 status,是 pod 自己退出,还是由于 oom 被杀,或者是被驱逐
2. 如果是 pod 自己退出,`kubectl logs pod-name -p -n ns` 查看容器退出的日志,排查原因
3. 如果是由于 oom 被杀,建议根据业务重新调整 pod 的 request 和 limit 设置(两者不宜相差过大),或检查是否有内存泄漏
4. 如果 pod 被驱逐,说明节点压力过大,需要检查时哪个 pod 占用资源过多,并调整 request 和 limit 设置
5. 非 pod 自身原因导致的退出,需要执行dmesg查看系统日志以及journalctl -u kubelet查看 kubelet 相关日志。

9. 为什么在 K8S 节点 Docker 直接起容器网络不通

1. UK8S 使用 UCloud 自己的 CNI 插件,而直接用 Docker 起的容器并不能使用该插件,因此网络不通。
2. 如果需要长期跑任务,不建议在 UK8S 节点用 Docker 直接起容器,应该使用 pod
3. 如果只是临时测试,可以添加--network host 参数,使用 hostnetwork 的模式起容器

10. Pod的时区问题

在 Kubernetes 集群中运行的容器默认使用格林威治时间，而非宿主机时间。如果需要让容器时间与宿主机时间一致，可以使用 "hostPath" 的方式将宿主机上的时区文件挂载到容器中。

大部分linux发行版都通过 "/etc/localtime" 文件来配置时区，我们可以通过以下命令来获取时区信息：

```
# ls -l /etc/localtime
lrwxrwxrwx. 1 root root 32 Oct 15 2015 /etc/localtime -> ../usr/share/zoneinfo/Asia/Shanghai
```

通过上面的信息，我们可以知道宿主机所在的时区为Asia/Shanghai，下面是一个Pod的yaml范例，说明如何将容器内的时区配置更改为Asia/Shanghai，和宿主机保持一致。

```
apiVersion: app/v1
kind: Pod
metadata:
name: nginx
labels:
name: nginx
spec:
containers:
- name: nginx
image: nginx
imagePullPolicy: "IfNotPresent"
resources:
requests:
cpu: 100m
```

```
memory: 100Mi
ports:
- containerPort: 80
volumeMounts:
- name: timezone-config
mountPath: /etc/localtime
volumes:
- name: timezone-config
hostPath:
path: /usr/share/zoneinfo/Asia/Shanghai
```

如果容器之前已经创建了，只需要在 yaml 文件中加上 `volumeMounts` 及 `volumes` 参数，再使用 `kubectl apply` 命令更新即可。

权限管理实践

本文主要通过一个例子来介绍如何基于 Kubernetes 的 RBAC 实现授权决策，允许集群管理员通过 Kubernetes API 动态配置策略，让非集群管理员具有某个 namespace 下的所有权限，并可通过 Dashboard 或者 kubectl 来管理该 ns 下的资源。

如果要更加深入地了解和掌握 RBAC，可以查看官方文档。

K8S里面有两类用户，

- Service Account，Kubernetes 中一种用于非人类用户的账号，在 Kubernetes 集群中提供不同的身份标识。详细可参考官方文档
- 普通用户(user)，K8S本身不并管理user，而是交由外部独立服务管理，不能通过K8SAPI来创建user;

目前是通过kubectl和Dashboard来管理集群，Service account已经足够满足要求，而且可以在Kubernetes中直接管理。因此这里不介绍如何使用user这个对象来管理集群。

1. 创建NS

```
kubectl create ns pre
```

上面的示例创建了一个名为"pre"的命名空间，用于部署预发布的服务。

2. 创建Service Account

```
kubectl create sa mingpianwang -n pre
```

在pre的命名空间下创建一个名为"mingpianwang"的Service account，给到某个特定的用户使用。

3. 赋予权限

由于我们已经预先说明，需要给mingpianwang这个用户赋予pre 这个命名空间下的所有权限，即admin权限。

重点来了，RoleBinding对象是可以引用一个ClusterRole对象的，然后这个ClusterRole所拥有的权限只会在这个NS下面有效。这一点允许管理员在整个集群范围内首先定义一组通用的角色，然后再在不同的名字空间中复用这些角色。

我们先看下集群内默认的ClusterRole有哪些，执行get clusterrole命名可以看到，有admin、cluster-admin、edit等角色，那我们可以直接使用admin这个clusterrole角色，通过rolebinding的方式赋予”mingpianwang“这个用户。

```
[root@10-9-149-7 ~]# kubectl get clusterrole
NAME AGE
admin 4h53m
cluster-admin 4h53m
edit 4h53m
```

示例的yaml如下，我们只要执行下kubectl apply -f rolebinding.yaml 即可。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
name: kubernetes-dashboard-minimal
```

```
namespace: pre
roleRef:
apiGroup: rbac.authorization.k8s.io
kind: ClusterRole
name: admin
subjects:
- kind: ServiceAccount
name: mingpianwang
namespace: pre
```

当然，我们也可以创建一个Namespace级别的role，并将这个角色绑定到ServiceAccount。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
namespace: pre
name: admin
rules:
- apiGroups: [""] # "" indicates the core API group
resources: ["*"]
verbs: ["*"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
```

```
metadata:
name: kubernetes-dashboard-minimal
namespace: pre
roleRef:
apiGroup: rbac.authorization.k8s.io
kind: Role
name: admin
subjects:
- kind: ServiceAccount
name: mingpianwang
namespace: pre
```

只是这个role不能复用到其他Namespace，一般只有在做精细化权限管理的时候，我们才会创建Role对象，比如一个只能查看pod 名称为test-pod的Role。其他场景下，我们推荐集群管理员使用ClusterRole。

4. 访问Dashboard

在 Kubernetes 1.22 之前的版本中，Kubernetes 会以Secret 形式为ServciceAccount 提供一个长期的有效的静态令牌，Kubernetes v1.22 及更高版本中，需要用户自己配置；详细的可参考官方文档手动获取 ServiceAccount 凭据

Kubernetes 1.22 之前的版本

就要获取到”mingpianwang“的token，其实就是secret了。通过下面的方式来获取，最后的token复制下来就可以了。

```
bash-4.4# kubectl describe sa/mingpianwang -n pre
Name: mingpianwang
Namespace: pre
Labels: <none>
Annotations: <none>
Image pull secrets: <none>
Mountable secrets: mingpianwang-token-4l8xj
Tokens: mingpianwang-token-4l8xj
Events: <none>
bash-4.4# kubectl describe secret/mingpianwang-token-4l8xj -n pre
Name: mingpianwang-token-4l8xj
Namespace: pre
Labels: <none>
Annotations: kubernetes.io/service-account.name: mingpianwang
kubernetes.io/service-account.uid: d7bb847d-7621-11e9-9679-5254007e7ba9

Type: kubernetes.io/service-account-token

Data
====
ca.crt: 1359 bytes
namespace: 5 bytes
token: eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9/....
```

Kubernetes 1.22 之后的版本

如果你需要为 ServiceAccount 获得一个 API 令牌，你可以创建一个新的、带有特殊注解 kubernetes.io/service-account.name 的 Secret 对象。详细可参考官方文档手动获取一个长久的Token

```
bash-4.4# kubectl -n pre apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
name: mingpianwang-secret
annotations:
kubernetes.io/service-account.name: mingpianwang
type: kubernetes.io/service-account-token
EOF
```

可以通过下面的命令来查看 Secret：

```
bash-4.4# kubectl -n pre describe secrets/mingpianwang-secret
Name: mingpianwang-secret
Namespace: pre
Labels: <none>
Annotations: kubernetes.io/service-account.name: mingpianwang
kubernetes.io/service-account.uid: 0c3e9a16-6962-45ee-9e6e-e7f0107cd9a8

Type: kubernetes.io/service-account-token
```

```
Data
====
ca.crt: 1359 bytes
namespace: 3 bytes
token: ...
```

这里将 token 的内容抹去了。当你删除一个与某 Secret 相关联的 ServiceAccount 时，Kubernetes 的控制面会自动清理该 Secret 中长期有效的令牌。

可以使用以下命令查看 ServiceAccount:

```
kubectl -n pre get serviceaccount mingpianwang -o yaml
```

在 ServiceAccount API 对象中看不到 mingpianwang-secret Secret，.secrets 字段，因为该字段只会填充自动生成的 Secret。

登陆Dashboard

复制到登录框，我们发现可以登录到Dashboard首页，不过需要注意的是，由于这个账号只有pre这个命名空间的权限，而Dashboard默认是default，所以进去之后会报一堆错咯，没关系，只要将左侧的NS改为pre即可。

5. 通过 kubectl 管理集群

由于我们还需要支持 kubectl 命令行管理 NS，因此还需要为 mingpianwang 生成kubeconfig，一个用户还好，多个用户就很麻烦了，因此这里我们使用一个自动生成 kubeconfig 的脚本，代码如下：

```
#!/bin/bash -e
# Usage ./k8s-service-account-kubeconfig.sh ( namespace ) ( service account name )
TEMPDIR=$( mktemp -d )
trap "{ rm -rf $TEMPDIR ; exit 255; }" EXIT
SA_SECRET=$( kubectl get sa -n $1 $2 -o jsonpath='{.secrets[0].name}' )
# Pull the bearer token and cluster CA from the service account secret.
BEARER_TOKEN=$( kubectl get secrets -n $1 $SA_SECRET -o jsonpath='{.data.token}' | base64 -d )
kubectl get secrets -n $1 $SA_SECRET -o jsonpath='{.data.ca\.crt}' | base64 -d > $TEMPDIR/ca.crt
CLUSTER_URL=$( kubectl config view -o jsonpath='{.clusters[0].cluster.server}' )
KUBECONFIG=kubeconfig

kubectl config --kubeconfig=$KUBECONFIG \
set-cluster \
$CLUSTER_URL \
--server=$CLUSTER_URL \
--certificate-authority=$TEMPDIR/ca.crt \
--embed-certs=true

kubectl config --kubeconfig=$KUBECONFIG \
set-credentials $2 --token=$BEARER_TOKEN

kubectl config --kubeconfig=$KUBECONFIG \
set-context registry \
--cluster=$CLUSTER_URL \
```

```
--user=$2 \
--namespace=$1

kubectl config --kubeconfig=$KUBECONFIG \
use-context registry

echo "kubeconfig written to file \"$KUBECONFIG\""
```

直接在master节点执行sh kubeconfig.sh pre mingpianwang，即可自动生成一个kubeconfig文件，将这个kubeconfig文件分发给使用者，让其复制到~/.kube/config下即可，而且默认NS就是pre，get nodes等操作都是不被允许的。

自动生成kubeconfig的源代码在这里，generator kubeconfig，我们只是加了一个默认NS，这样不需要在执行kubectl命令的时候追加-n pre。

在控制台使用集群审计功能

审计功能使得集群管理员能够回答以下问题：

- 发生了什么？
- 什么时候发生的？
- 谁触发的？
- 活动发生在哪个（些）对象上？
- 在哪观察到的？
- 它从哪触发的？
- 活动的后续处理行为是什么？

> 审计日志记录功能会增加 API server 的内存消耗，因为需要为每个请求存储审计所需的某些上下文。此外，内存消耗取决于审计日志记录的配置。

您可以通过UK8S的控制台来方便地开启集群审计功能，并将审计日志上传到ES中以方便进行查询。

准备

在开始前，您需要准备一个ES服务，以保存集群的审计日志。我们支持两种方式的ES：

- 直接使用UK8S提供的日志ELK。（推荐）
- 使用外部ES，可以是您自建的或是我们的UES服务。需要确保该ES跟UK8S集群在同一个VPC中。

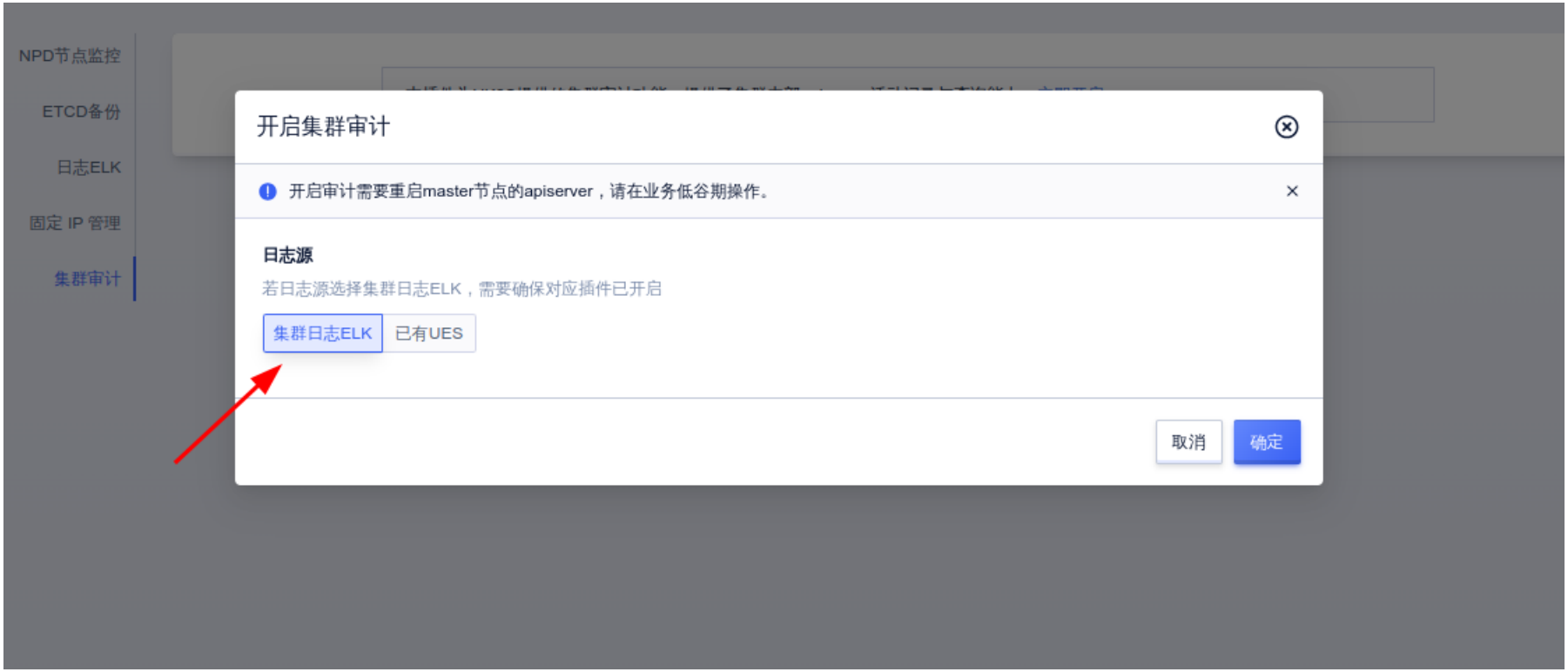
对于第一种方式，您需要打开控制台的应用中心 / 日志ELK功能。

开启集群审计

⚠ 开启集群审计时，会重启集群的APIServer，请在业务低谷期操作！

您可以直接到应用中心 / 集群审计页面开启审计：

在开启时，如果您打算直接将审计日志上传到集群内部ELK，直接选择集群日志ELK即可。注意这需要您的集群开启了日志ELK插件功能：



如果您需要使用外链ES，则选择已有UES，这需要您填入外部ES的子网和IP地址。如果您的ES开启了认证，还需要填入认证信息：

点击确定，我们会启动一个异步任务，分别在3个master节点上做下面的事情：

- 上传审计策略文件到/etc/kubernetes/yaml/audit-policy.yaml。具体策略文件见下一节。
- 修改APIServer配置文件，增加审计相关参数：
  - `--audit-log-path=/var/log/kubernetes/audit.log`
  - `--audit-policy-file=/etc/kubernetes/yaml/audit-policy.yaml`
  - `--audit-log-maxage=7`
  - `--audit-log-maxbackup=10`

    - `--audit-log-maxsize=1000`
- 重启APIServer，以让审计开始生效。
- 启动一个filebeat服务，将审计日志上传到上面配置的ES中。

该过程会顺序执行，需要消耗几分钟时间来完成，请耐心等待。

审计策略

在默认情况下，我们会为您准备好一个审计策略文件。它适用于大多数情况：

```
apiVersion: audit.k8s.io/v1
kind: Policy
omitStages:
- "RequestReceived"
rules:
# 集群中包含大量以下低风险请求，建议不做审计（不记录日志）
- level: None
users:
- admin
- kubelet
- system:apiserver
- system:kube-controller-manager
- system:kube-proxy
- system:serviceaccount:kube-system:cni-vpc-ipamd
- system:unsecured
```

```
verbs:
- get
- list
- watch

# system:node 用户组对于 node 节点的 get 请求
- level: None
userGroups: ["system:nodes"]
verbs: ["get"]
resources:
- group: "" # core
resources: ["nodes", "nodes/status"]

# 系统组件在 kube-system namespace 下对于 endpoints 的 get/update 请求
- level: None
users:
- system:kube-controller-manager
- system:kube-scheduler
- system:serviceaccount:kube-system:endpoint-controller
verbs: ["get", "update"]
namespaces: ["kube-system"]
resources:
- group: "" # core
resources: ["endpoints"]
```

```
# cluster-autoscaler 集群伸缩组件在 kube-system namespace 下对 configmap、endpoint 的 get/update 请求
- level: None
users: ["cluster-autoscaler"]
verbs: ["get", "update"]
namespaces: ["kube-system"]
resources:
- group: "" # core
resources: ["configmaps", "endpoints"]

# 以下只读 URL
- level: None
nonResourceURLs:
- /healthz*
- /version
- /swagger*

# event 事件
- level: None
resources:
- group: "" # core
resources: ["events"]

# 集群插件有大量对lease的操作,不做审计
- level: None
resources:
```

```
- group: "coordination.k8s.io"
resources: ["leases"]

# cloudprovider对endspoints的操作
- level: None
users: ["system:serviceaccount:kube-system:cloudprovider-ucloud"]
resources:
- group: ""
resources: ["endpoints"]

# kubelet、system:node-problem-detector 和 system:nodes 对于节点的 update 和 patch 请求，等级设置为 Request，记录元数据和请求的消息体
- level: Request
users: ["kubelet", "system:node-problem-detector", "system:serviceaccount:kube-system:node-problem-detector"]
verbs: ["update","patch"]
resources:
- group: "" # core
resources: ["nodes/status", "pods/status"]
- level: Request
userGroups: ["system:nodes"]
verbs: ["update","patch"]
resources:
- group: "" # core
resources: ["nodes/status", "pods/status"]

# 对于可能包含敏感信息或二进制文件的 Secrets，ConfigMaps，tokenreviews 接口的日志等级设为 Metadata
```

```
- level: Metadata
resources:
- group: "" # core
resources: ["secrets", "configmaps", "serviceaccounts/token"]
- group: authentication.k8s.io
resources: ["tokenreviews"]

# 对于一些返回体比较大的 get, list, watch 请求，设置为 Request
- level: Request
verbs: ["get", "list", "watch"]
resources:
- group: "" # core
- group: "admissionregistration.k8s.io"
- group: "apiextensions.k8s.io"
- group: "apiregistration.k8s.io"
- group: "apps"
- group: "authentication.k8s.io"
- group: "authorization.k8s.io"
- group: "autoscaling"
- group: "batch"
- group: "certificates.k8s.io"
- group: "extensions"
- group: "metrics.k8s.io"
- group: "networking.k8s.io"
- group: "node.k8s.io"
```

```
- group: "policy"
- group: "rbac.authorization.k8s.io"
- group: "scheduling.k8s.io"
- group: "storage.k8s.io"

# 对已知 Kubernetes API 默认设置为 RequestResponse
- level: RequestResponse
resources:
- group: "" # core
- group: "admissionregistration.k8s.io"
- group: "apiextensions.k8s.io"
- group: "apiregistration.k8s.io"
- group: "apps"
- group: "authentication.k8s.io"
- group: "authorization.k8s.io"
- group: "autoscaling"
- group: "batch"
- group: "certificates.k8s.io"
- group: "extensions"
- group: "metrics.k8s.io"
- group: "networking.k8s.io"
- group: "node.k8s.io"
- group: "policy"
- group: "rbac.authorization.k8s.io"
- group: "scheduling.k8s.io"
```

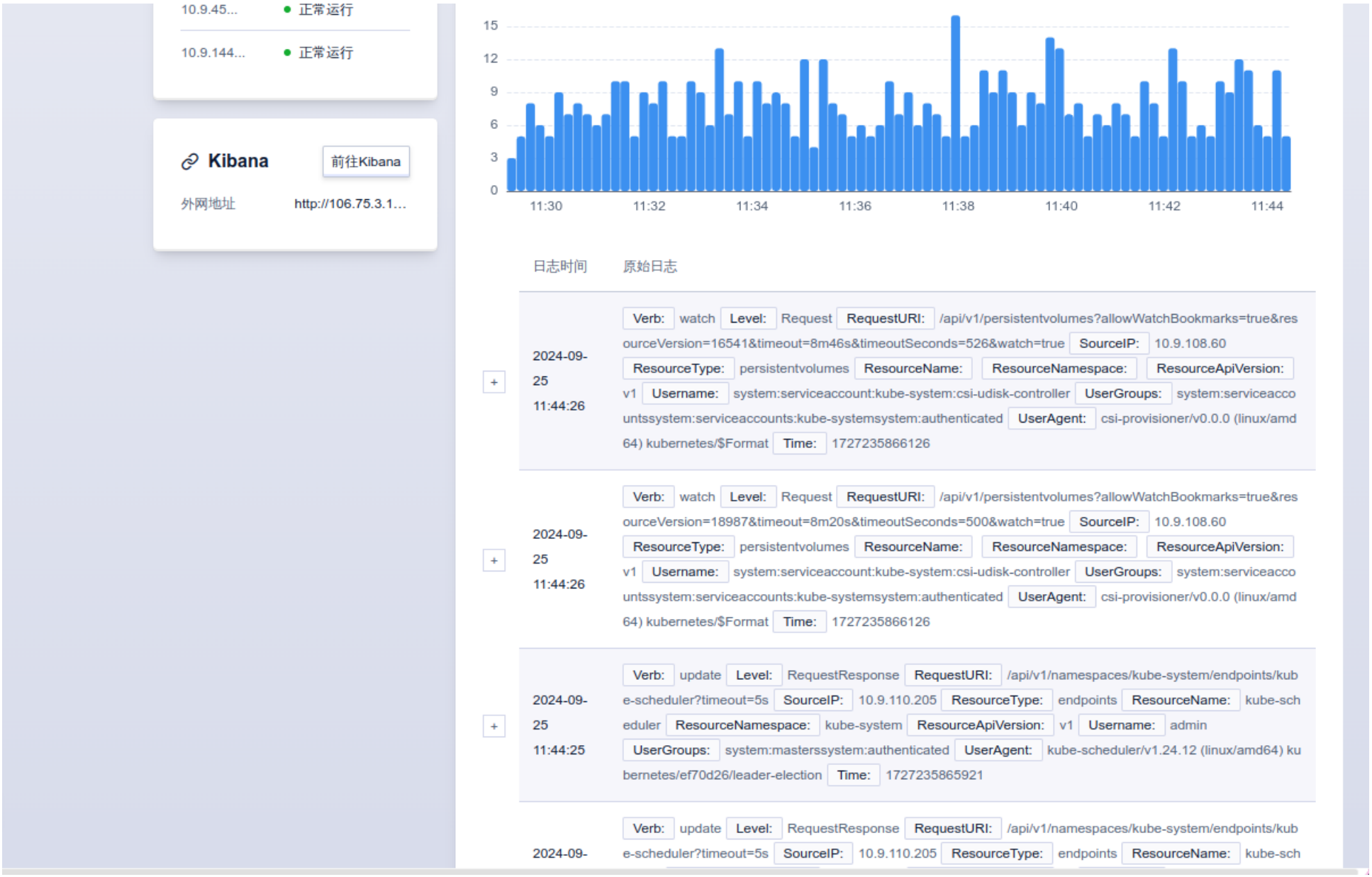
```
- group: "storage.k8s.io"

# 对于其他请求都设置为 Metadata
- level: Metadata
```

如果您对默认策略不满意，想要手动更改，请将策略文件依次上传到3个master节点的/etc/kubernetes/yaml/audit-policy.yaml中，并依次执行systemctl restart kube-apiserver重启APIServer服务。

关于如何配置审计策略，请参考文档：audit policy。

查看审计日志

开启审计之后，您可以在控制台简单地查询审计日志：



我们支持一些简单维度的过滤操作。但是支持的字段有限，请以控制台为准。

如果您希望使用更加复杂的过滤条件，建议到ES或是Kibana中进行操作。

关闭集群审计

⚠ 关闭集群审计时，会重启集群的APIServer，请在业务低谷期操作！

如果您不想使用集群审计了，例如觉得审计导致APIServer内存变得过高。可以在控制台进行关闭操作。直接在审计页面点击关闭应用即可：

关闭审计会分别在3个master节点做下面的事情：

- 移除用于上传日志的filebeat服务。
- 修改APIServer配置文件，删除审计相关参数。
- 移除审计策略文件。

- 重启APIServer。
- 移除审计日志文件/var/log/kubernetes/audit.log。

该过程会顺序执行，需要消耗几分钟时间来完成，请耐心等待。

手动开启 APIServer 审计功能

⚠ 该文档展示的是如何手动开启审计，如无特殊情况，请使用我们控制台的集群审计应用。

1. 审计策略

通过编辑文件 /etc/kubernetes/audit-policy.yaml，设定自己的审计策略

```
apiVersion: audit.k8s.io/v1
kind: Policy
omitStages:
- "RequestReceived"
rules:
# 集群中包含大量以下低风险请求，建议不做审计（不记录日志）
# kube-proxy 的 watch 请求
- level: None
users: ["system:kube-proxy"]
verbs: ["watch"]
resources:
- group: "" # core
```

```
resources: ["endpoints", "services", "services/status"]
# 在 kube-system namespace 下对 configmap 的 get 请求
- level: None
users: ["system:unsecured"]
namespaces: ["kube-system"]
verbs: ["get"]
resources:
- group: "" # core
resources: ["configmaps"]
# kubelet 对于 node 节点的 get 请求
- level: None
users: ["kubelet"] # legacy kubelet identity
verbs: ["get"]
resources:
- group: "" # core
resources: ["nodes", "nodes/status"]
# system:node 用户组对于 node 节点的 get 请求
- level: None
userGroups: ["system:nodes"]
verbs: ["get"]
resources:
- group: "" # core
resources: ["nodes", "nodes/status"]
# 系统组件在 kube-system namespace 下对于 endpoints 的 get/update 请求
- level: None
```

```
users:
- system:kube-controller-manager
- system:kube-scheduler
- system:serviceaccount:kube-system:endpoint-controller
verbs: ["get", "update"]
namespaces: ["kube-system"]
resources:
- group: "" # core
resources: ["endpoints"]
# apiserver 对于 namespace 的 get 请求
- level: None
users: ["system:apiserver"]
verbs: ["get"]
resources:
- group: "" # core
resources: ["namespaces", "namespaces/status", "namespaces/finalize"]
# cluster-autoscaler 集群伸缩组件在 kube-system namespace 下对 configmap、endpoint 的 get/update 请求
- level: None
users: ["cluster-autoscaler"]
verbs: ["get", "update"]
namespaces: ["kube-system"]
resources:
- group: "" # core
resources: ["configmaps", "endpoints"]
# HPA 通过 controller manager 获取 metrics 信息的请求
```

```
- level: None
users:
- system:kube-controller-manager
verbs: ["get", "list"]
resources:
- group: "metrics.k8s.io"
# 以下只读 URL
- level: None
nonResourceURLs:
- /healthz*
- /version
- /swagger*
# event 事件
- level: None
resources:
- group: "" # core
resources: ["events"]

# kubelet、system:node-problem-detector 和 system:nodes 对于节点的 update 和 patch 请求，等级设置为 Request，记录元数据和请求的消息体
- level: Request
users: ["kubelet", "system:node-problem-detector", "system:serviceaccount:kube-system:node-problem-detector"]
verbs: ["update","patch"]
resources:
- group: "" # core
resources: ["nodes/status", "pods/status"]
```

```
- level: Request
userGroups: ["system:nodes"]
verbs: ["update","patch"]
resources:
- group: "" # core
resources: ["nodes/status", "pods/status"]

# 对于可能包含敏感信息或二进制文件的 Secrets，ConfigMaps，tokenreviews 接口的日志等级设为 Metadata
- level: Metadata
resources:
- group: "" # core
resources: ["secrets", "configmaps", "serviceaccounts/token"]
- group: authentication.k8s.io
resources: ["tokenreviews"]

# 对于一些返回体比较大的 get, list, watch 请求，设置为 Request
- level: Request
verbs: ["get", "list", "watch"]
resources:
- group: "" # core
- group: "admissionregistration.k8s.io"
- group: "apiextensions.k8s.io"
- group: "apiregistration.k8s.io"
- group: "apps"
- group: "authentication.k8s.io"
```

```
- group: "authorization.k8s.io"
- group: "autoscaling"
- group: "batch"
- group: "certificates.k8s.io"
- group: "extensions"
- group: "metrics.k8s.io"
- group: "networking.k8s.io"
- group: "node.k8s.io"
- group: "policy"
- group: "rbac.authorization.k8s.io"
- group: "scheduling.k8s.io"
- group: "storage.k8s.io"

# 对已知 Kubernetes API 默认设置为 RequestResponse
- level: RequestResponse
resources:
- group: "" # core
- group: "admissionregistration.k8s.io"
- group: "apiextensions.k8s.io"
- group: "apiregistration.k8s.io"
- group: "apps"
- group: "authentication.k8s.io"
- group: "authorization.k8s.io"
- group: "autoscaling"
- group: "batch"
```

```
  - group: "certificates.k8s.io"
  - group: "extensions"
  - group: "metrics.k8s.io"
  - group: "networking.k8s.io"
  - group: "node.k8s.io"
  - group: "policy"
  - group: "rbac.authorization.k8s.io"
  - group: "scheduling.k8s.io"
  - group: "storage.k8s.io"

  # 对于其他请求都设置为 Metadata
  - level: Metadata
```

1.1 阶段（omitStages）

阶段	含义
RequestReceived	此阶段对应审计处理器接收到请求后，并且在委托给 其余处理器之前生成的事件
ResponseStarted	在响应消息的头部发送后，响应消息体发送前生成的事件。只有长时间运行的请求（例如 watch）才会生成这个阶段
ResponseComplete	当响应消息体完成并且没有更多数据需要传输的时候
Panic	当 panic 发生时生成

1.2 审计级别（level）

级别	含义
None	符合这条规则的日志将不会记录
Metadata	记录请求的元数据（请求的用户、时间戳、资源、动词等等），但是不记录请求或者响应的消息体
Request	记录事件的元数据和请求的消息体，但是不记录响应的消息体。这不适用于非资源类型的请求
RequestResponse	记录事件的元数据，请求和响应的消息体。这不适用于非资源类型的请

2. 审计日志配置

分别登录 3 台 Master 节点，在 APIServer 配置文件 `/etc/kubernetes/apiserver` 中添加以下参数，并通过 `systemctl restart kube-apiserver` 重启 APIServer：

```
# 指定用来写入审计事件的日志文件路径。不指定此标志会禁用日志后端
--audit-log-path=/var/log/audit.log
# 指定审计策略配置文件
--audit-policy-file=/etc/kubernetes/audit-policy.yaml
# 定义保留旧审计日志文件的最大天数
--audit-log-maxage=7
# 定义要保留的审计日志文件的最大数量
--audit-log-maxbackup=10
# 定义审计日志文件的最大大小（兆字节）
--audit-log-maxsize=1000
```

3. 参考

- Kubernetes 官方文档 - 审计

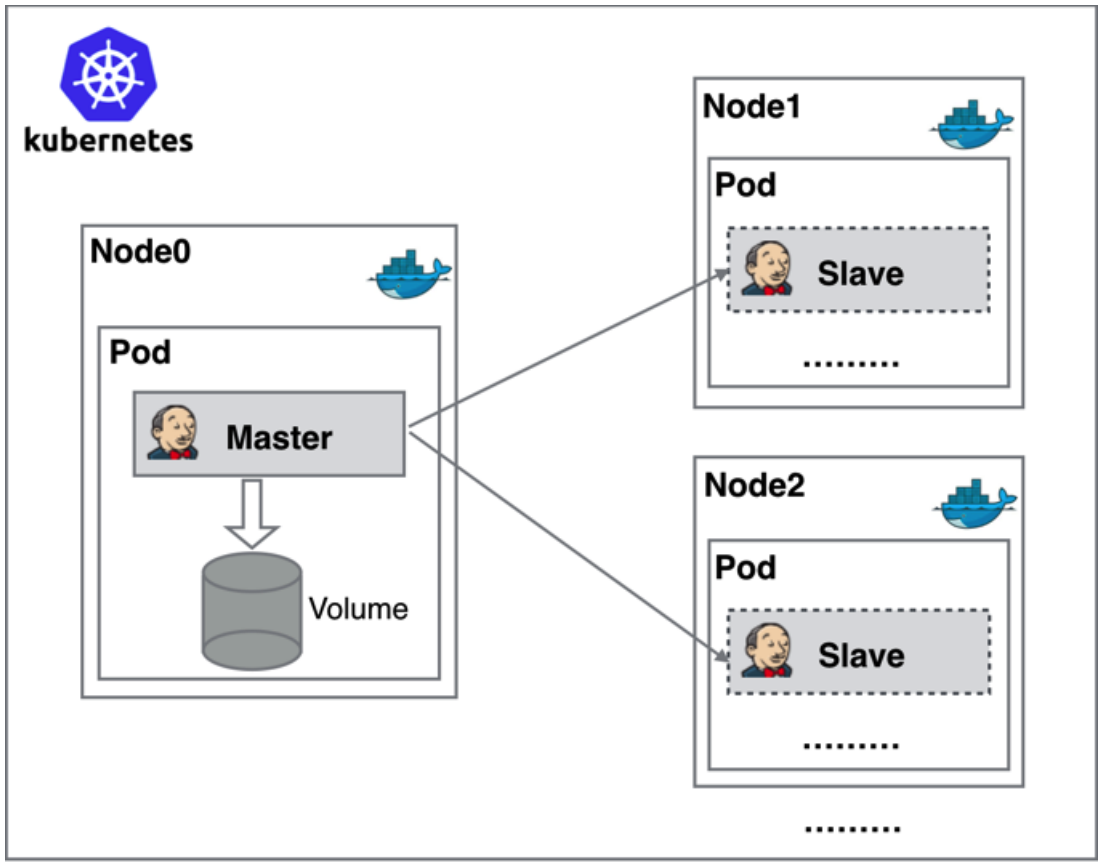
基于Jenkins的CI/CD实践

- 适用集群版本 1.14~1.18

一、概要

提到K8S环境下的CI/CD，可以使用的工具有很多，比如Jenkins、Gitlab CI、新兴的drone等，考虑到大多公司在VM环境下都采用 Jenkins 集群来搭建符合需求的 CI/CD 流程，这里先给介绍大家下Kubernetes+Jenkins的CI/CD方案。

1、Jenkins架构



Jenkins Master 和 Jenkins Slave 以 Pod 形式运行在 Kubernetes 集群的 Node 上，Master是常驻服务，所有的配置数据都存储在一个 Volume 中，Slave 不是一直处于运行状态，它会按照需求动态的创建并自动删除。

2、工作原理

当 Jenkins Master 接受到 Build 请求时，会根据配置的 Label 动态创建一个运行在 Pod 中的 Jenkins Slave 并注册到 Master 上，当运行完 Job 后，这个 Slave 会被注销并且这个 Pod 也会自动删除，恢复到最初状态。

3、优势

相对于部署在虚拟机环境下的Jenkins 一主多从架构，将Jenkins部署到K8S会带来以下好处：

- **服务高可用:**当 Jenkins Master 出现故障时，Kubernetes 会自动创建一个新的 Jenkins Master 容器，并且将 Volume 分配给新创建的容器，保证数据不丢失，从而达到集群服务高可用。
- **动态伸缩:** 合理使用资源，每次运行 Job 时，会自动创建一个 Jenkins Slave，Job 完成后，Slave 自动注销并删除容器，资源自动释放，而且 Kubernetes 会根据每个资源的使用情况，动态分配 Slave 到空闲的节点上创建，降低出现因某节点资源利用率高，还排队等待在该节点的情况。
- **扩展性好：** 当 Kubernetes 集群的资源严重不足而导致 Job 排队等待时，可以很容易的添加一个 Kubernetes Node 到集群中，从而实现扩展。

二、部署Jenkins

1、为了管理方便，我们把需要创建的资源都部署在一个名为 jenkins 的 namespace 下面，所以我们需要添加创建一个 namespace：

```
kubectl create namespace jenkins
```

2、声明一个PVC对象，后面我们要将Jenkins容器的 /var/jenkins_home 目录挂载到了这个名为PVC对象上面。

- 如果您使用的k8s版本大于等于1.14，且没有使用快杰云主机，请部署。

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cicd/yaml_jenkins_jenkins-pvc.yaml
```

- 如果您使用的k8s版本小于1.14，请部署。

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cicd/yaml_jenkins_jenkins-pvc-1.13.yaml
```

3、以Deployment方式部署Jenkins master,为了演示方便，我们还使用LoadBalancer类型的service将其暴露到外网。

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cicd/yaml_jenkins_jenkins.yaml
```

4、等到服务启动成功后，我们就可以根据LoadBalancer的IP（即EXTERNAL-IP），访问 jenkins 服务了，并根据提示信息进行安装配置。

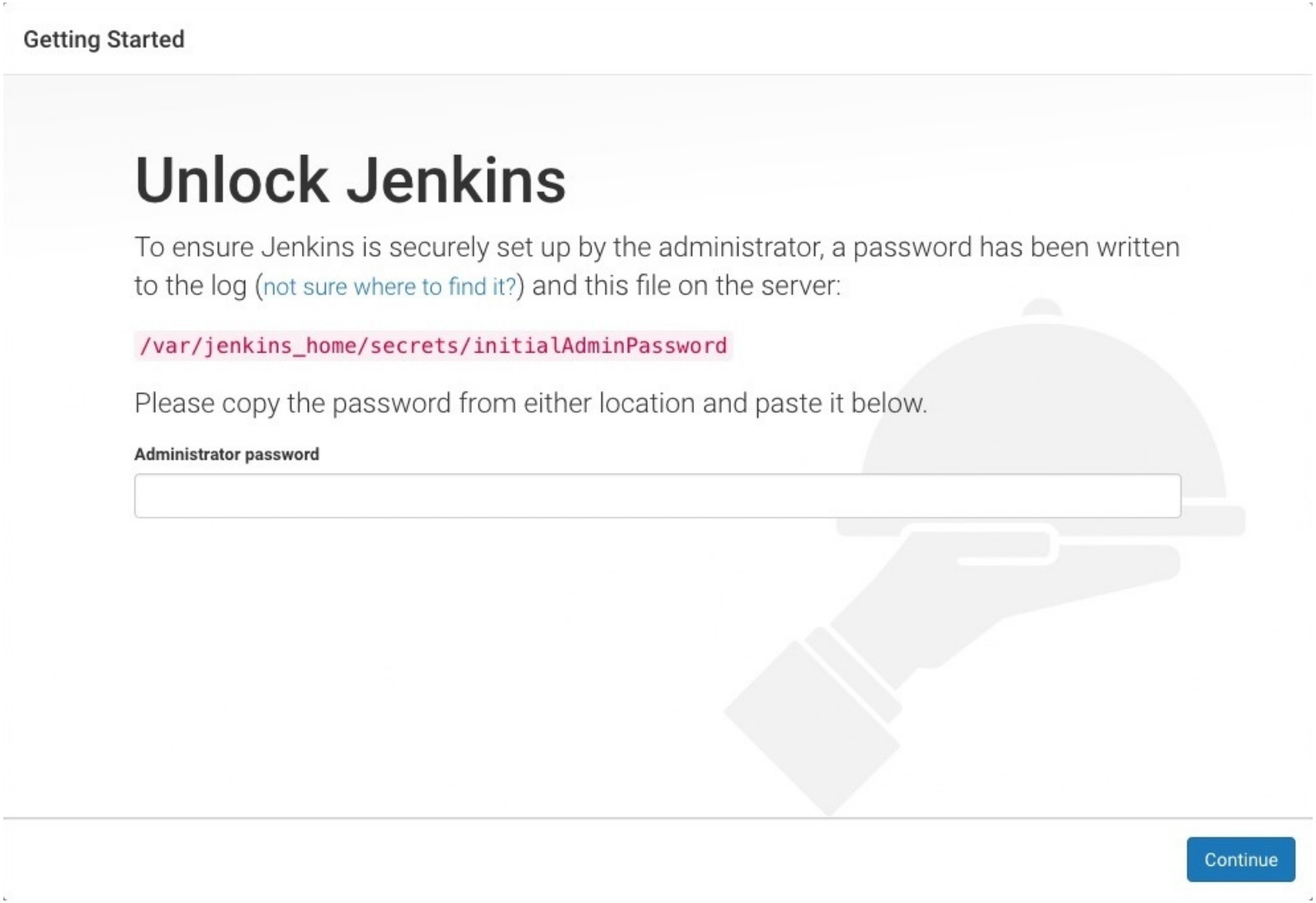
```
bash-4.4# kubectl get svc -n jenkins
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT AGE
jenkins LoadBalancer 172.17.201.210 106.75.98.80 8080:33651/TCP,50000:43748/TCP 4d21h
```

5、创建一个名为jenkins2的ServiceAccount，并且为其赋予特定的权限，后面配置Jenkins-Slave我们会用到。

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cicd/yaml_jenkins_jenkins-rbac.yaml
```

三、安装Kubernetes插件

1、前面我们已经获取到Jenkins的外网IP地址，我们直接在浏览器输入EXTERNAL-IP:8080，即可打开Jenkins页面，提示需要输入初始化密码：



2、我们通过kubectl log获取jenkins容器的日志来获取初始化密码

```
kubectl logs jenkins-deployment-66b865dbd-xvmdz -n jenkins
```

命令行-uk8s

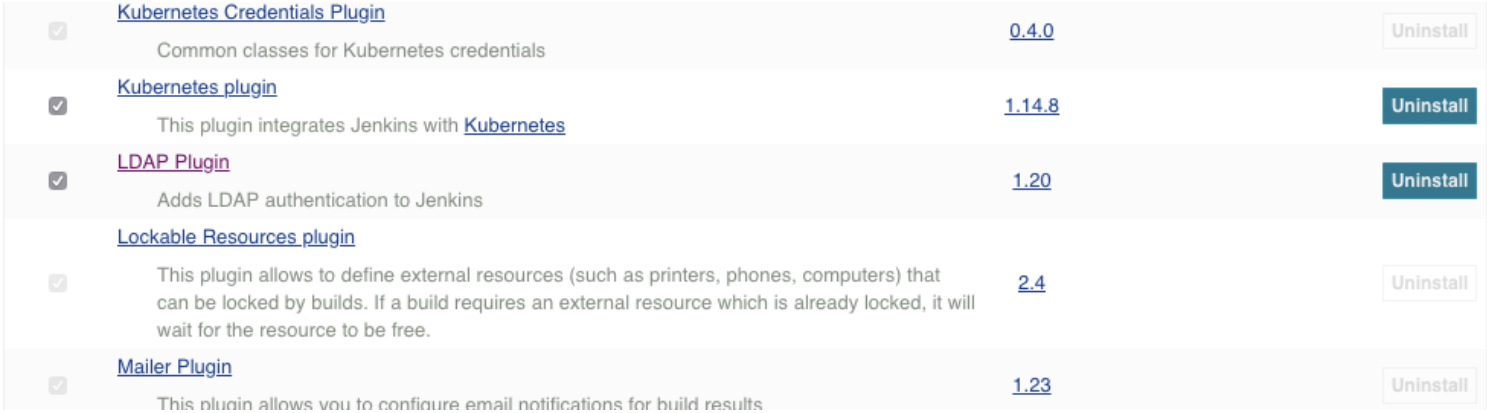
https://dws.ucloud.cn/terminal/uk8s/cn-bj2/org-loccjs/uk8s-krdctz

```
Mar 04, 2019 12:35:57 PM jenkins.InitReactorRunner$1 onAttained
INFO: Prepared all plugins
Mar 04, 2019 12:35:57 PM jenkins.InitReactorRunner$1 onAttained
INFO: Started all plugins
Mar 04, 2019 12:35:57 PM jenkins.InitReactorRunner$1 onAttained
INFO: Augmented all extensions
Mar 04, 2019 12:35:58 PM jenkins.InitReactorRunner$1 onAttained
INFO: Loaded all jobs
Mar 04, 2019 12:35:58 PM hudson.model.AsyncPeriodicWork$1 run
INFO: Started Download metadata
Mar 04, 2019 12:35:59 PM jenkins.util.groovy.GroovyHookScript execute
INFO: Executing /var/jenkins_home/init.groovy.d/tcp-slave-agent-port.groovy
Mar 04, 2019 12:35:59 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.StaticWebApplicationContext@14608a4: display name [Roo
 WebApplicationContext]; startup date [Mon Mar 04 12:35:59 UTC 2019]; root of context hierarchy
Mar 04, 2019 12:35:59 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext
14608a4]: org.springframework.beans.factory.support.DefaultListableBeanFactory@445e198a
Mar 04, 2019 12:35:59 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSin
letons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@445e
98a: defining beans [authenticationManager]; root of factory hierarchy
Mar 04, 2019 12:36:00 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.StaticWebApplicationContext@18623816: display name [Ro
t WebApplicationContext]; startup date [Mon Mar 04 12:36:00 UTC 2019]; root of context hierarchy
Mar 04, 2019 12:36:00 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext
18623816]: org.springframework.beans.factory.support.DefaultListableBeanFactory@69932bbd
Mar 04, 2019 12:36:00 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSin
letons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@6993
bbd: defining beans [filter,legacy]; root of factory hierarchy
Mar 04, 2019 12:36:00 PM jenkins.install.SetupWizard init
INFO:
*************************************************************
*************************************************************
*************************************************************
Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:
4805e19fc13e479288d0dd3ca45134a7
This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
*************************************************************
```

```
************************************************************
************************************************************
```

3、选择推荐安装，添加完管理员帐号admin，即可进入到 jenkins 主界面。

4、接下来安装jenkins依赖插件清单——kubernets plugin，让他能够动态的生成 Slave 的 Pod。点击 Manage Jenkins -> Manage Plugins -> Available -> Kubernetes plugin 勾选安装即可。



安装插件相对较慢，请耐心等待，并且由于是在线安装，集群需要开通外网，请开启natgw来使node节点通外网

四、配置Jenkins

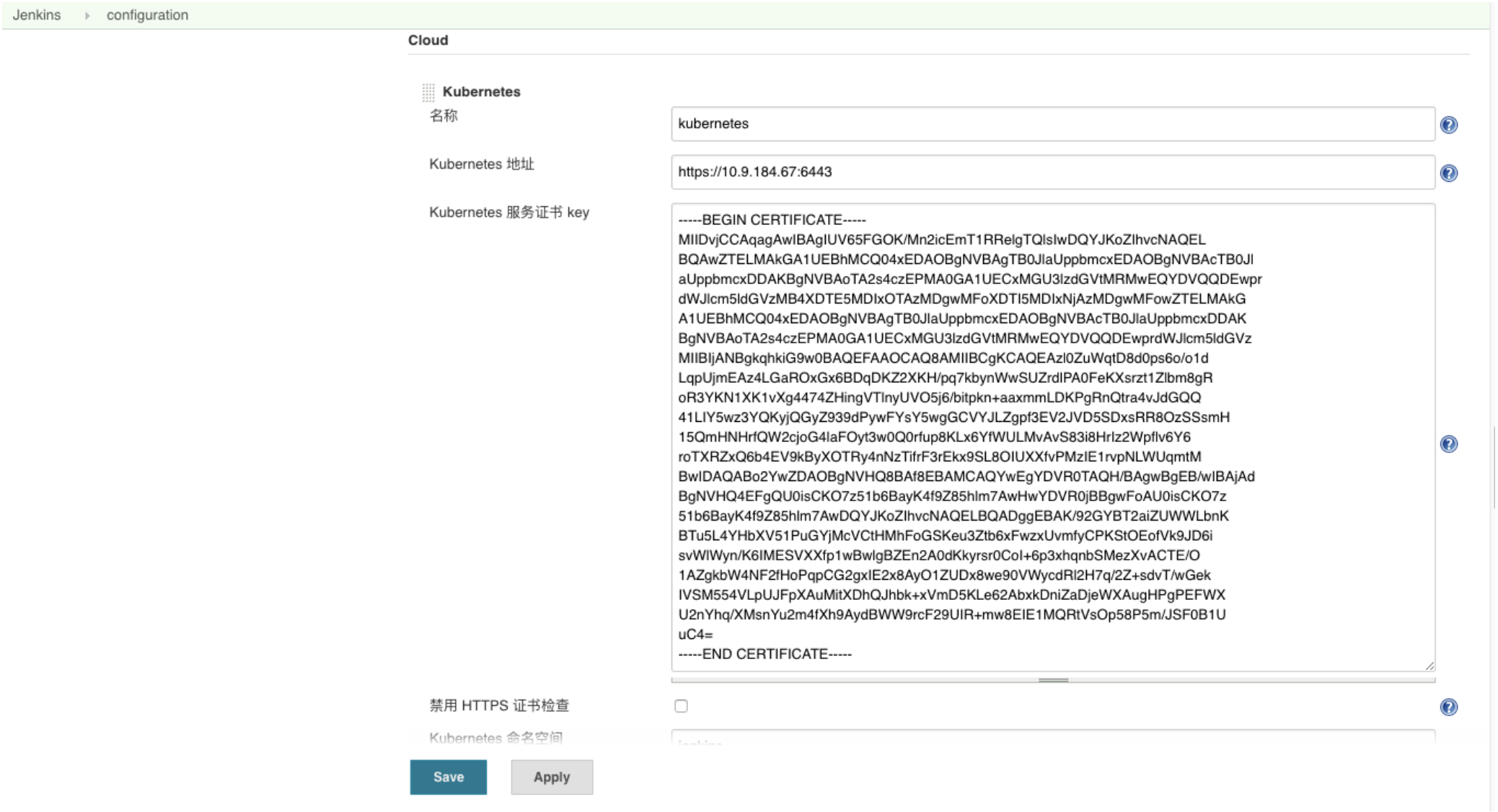
接下来将进入最重要的一个步骤，在Kubernetes插件安装完毕后，我们需要配置Jenkins和Kubernetes参数，使Jenkins连接到UK8S集群，并能调用Kubernetes API 动态创建Jenkins Slave，执行构建任务。

首先点击 Manage Jenkins —> Configure System，进入到系统设置页面。滚动到页面最下方，然后点击Add a new cloud —> 选择 Kubernetes，开始填写 Kubernetes 和 Jenkins 配置信息。

1、输入UK8S Apiserver地址，以及服务证书key。

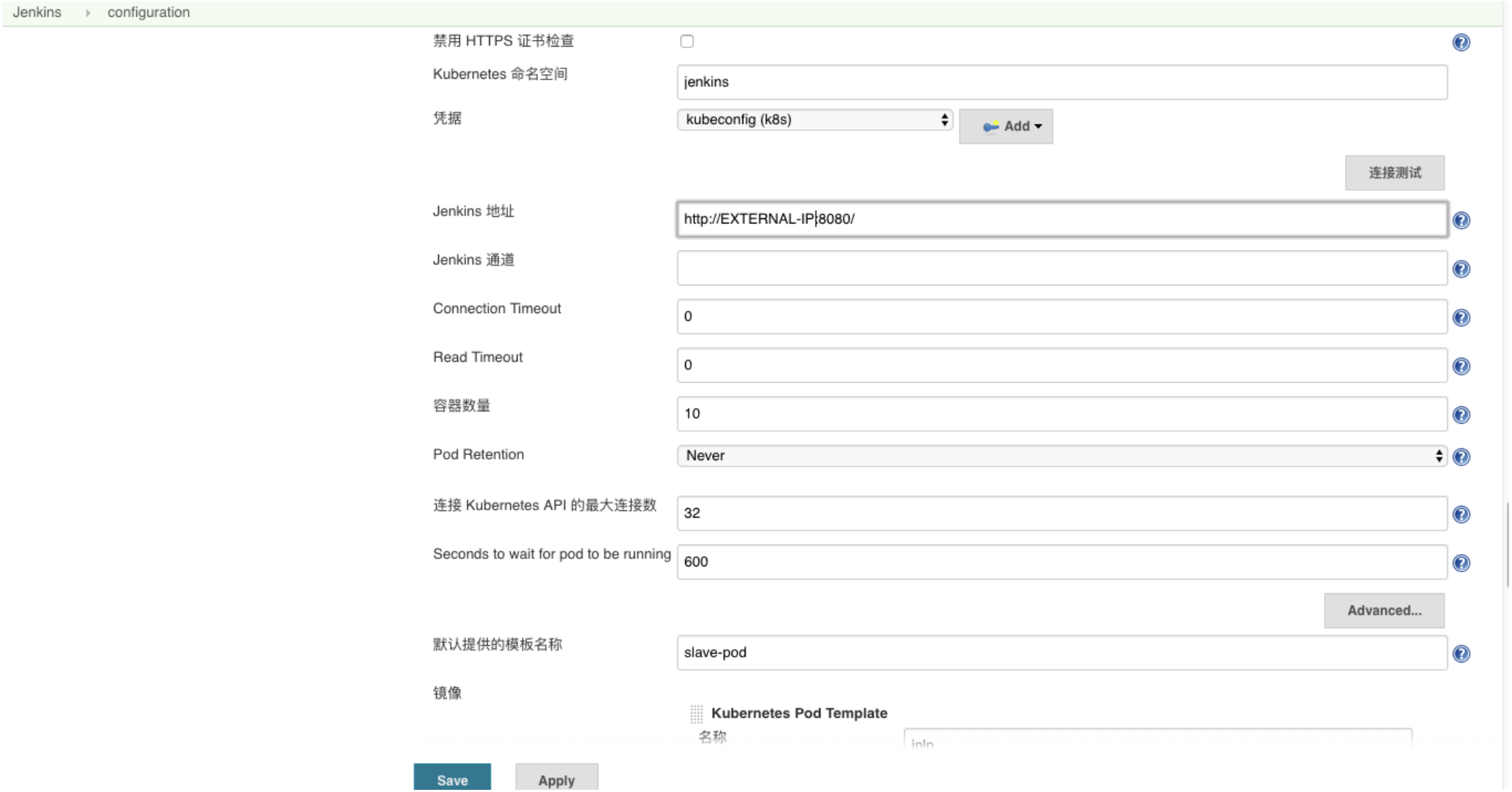
以上两个参数信息，可以在UK8S集群详情页的内网或外网集群凭证中获取。"服务证书key"为集群凭证中的certificate-authority-data字段内容，进行base64解码，将解码后的内容复制

到输入框即可。



2、填写集群Namespace、上传凭据、Jenkins地址

Namespace此处填写之前创建Namespace即可，此处为jenkins。凭证处，点击”Add“，凭证类型选择"Secret file"，将UK8S集群详情页全部内容复制下来，保存为kubeconfig上传。

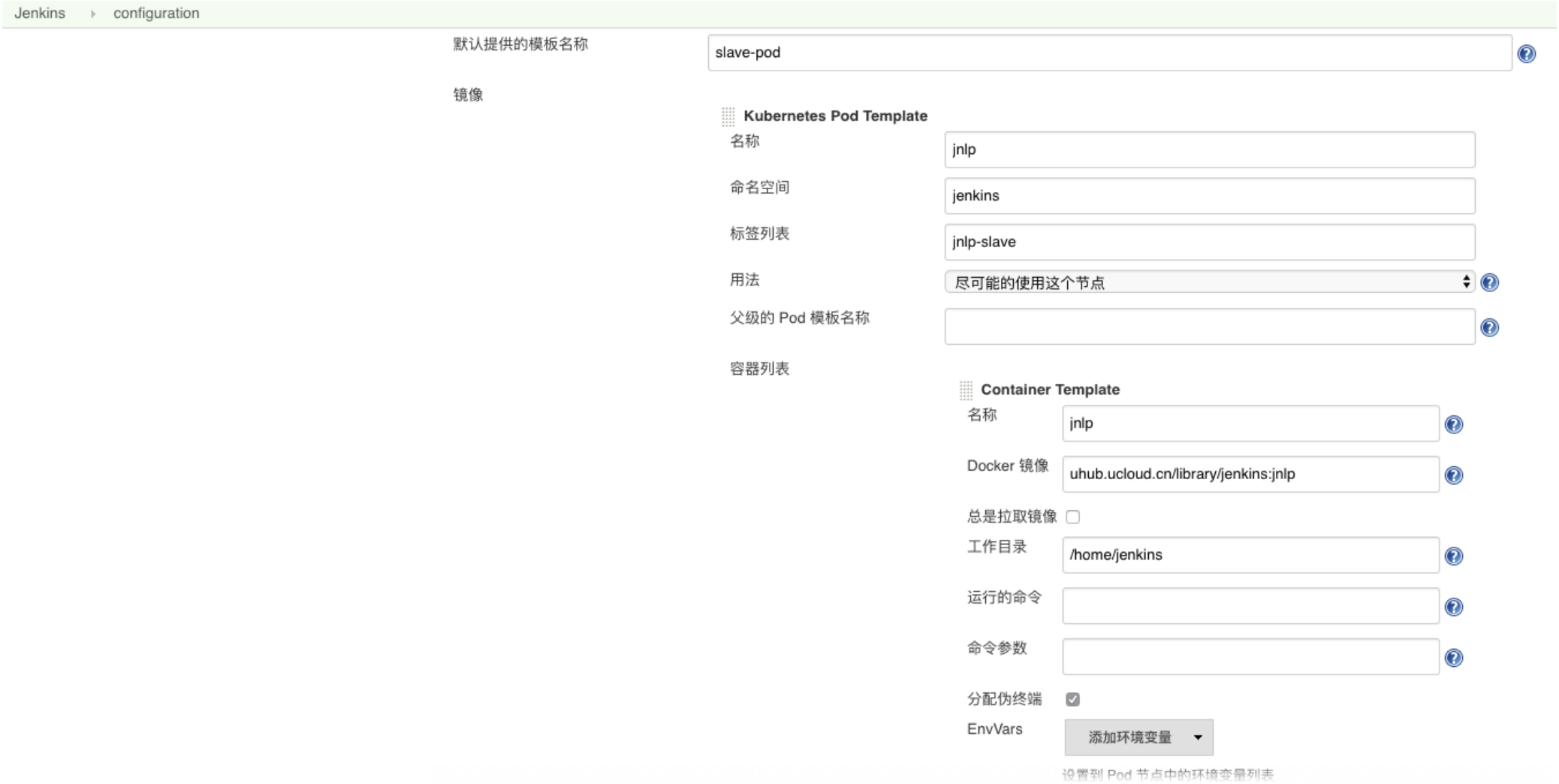


3、点击”连接测试“，如果出现 Connection test successful 的提示信息证明 Jenkins 已经可以和 Kubernetes 系统正常通信了

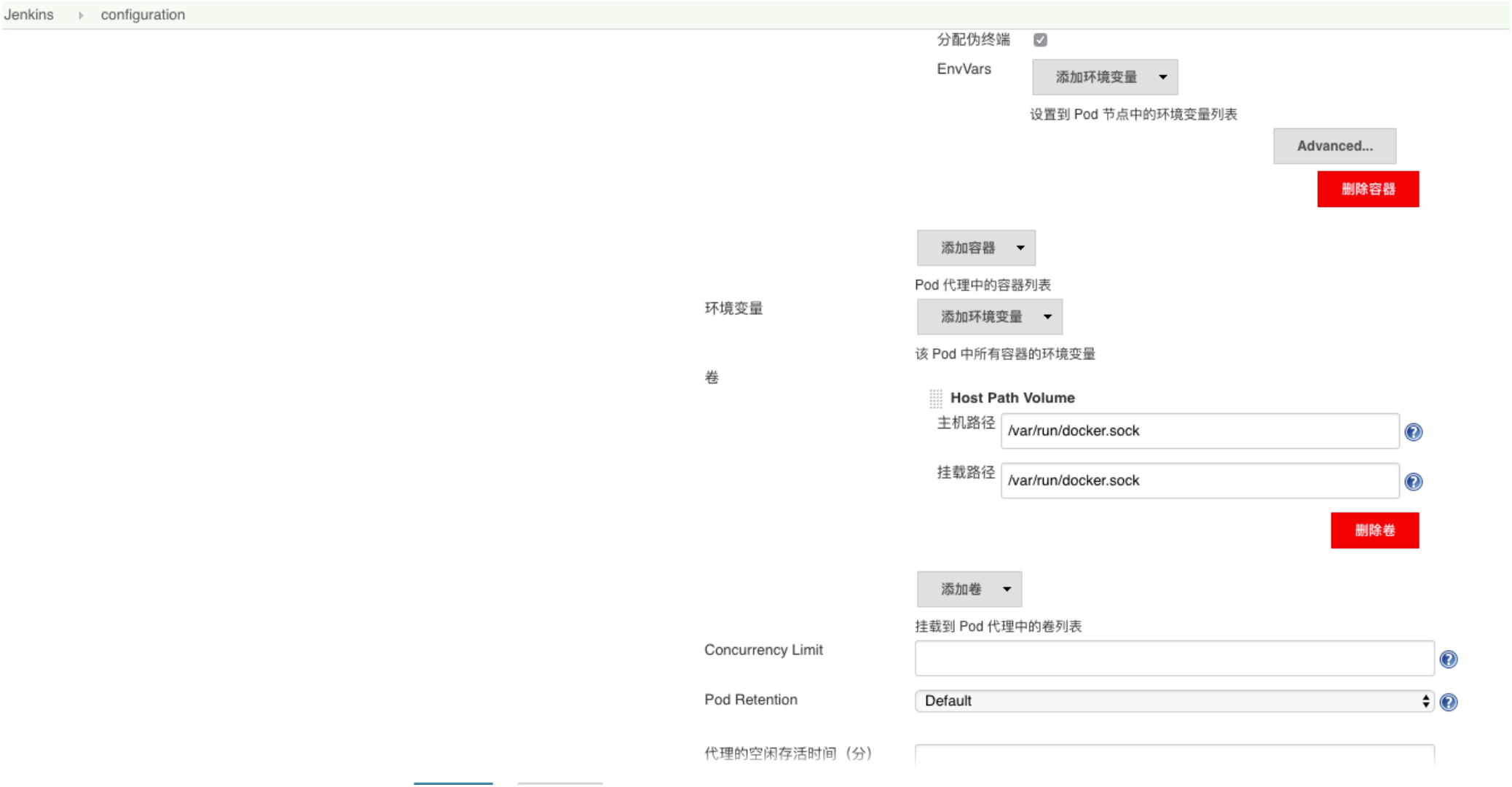
4、接下来，我们点击”添加Pod模板“，这个Pod模板即Jenkins-slave pod的模板。

- namespace，我们这里填 ”jenkins“

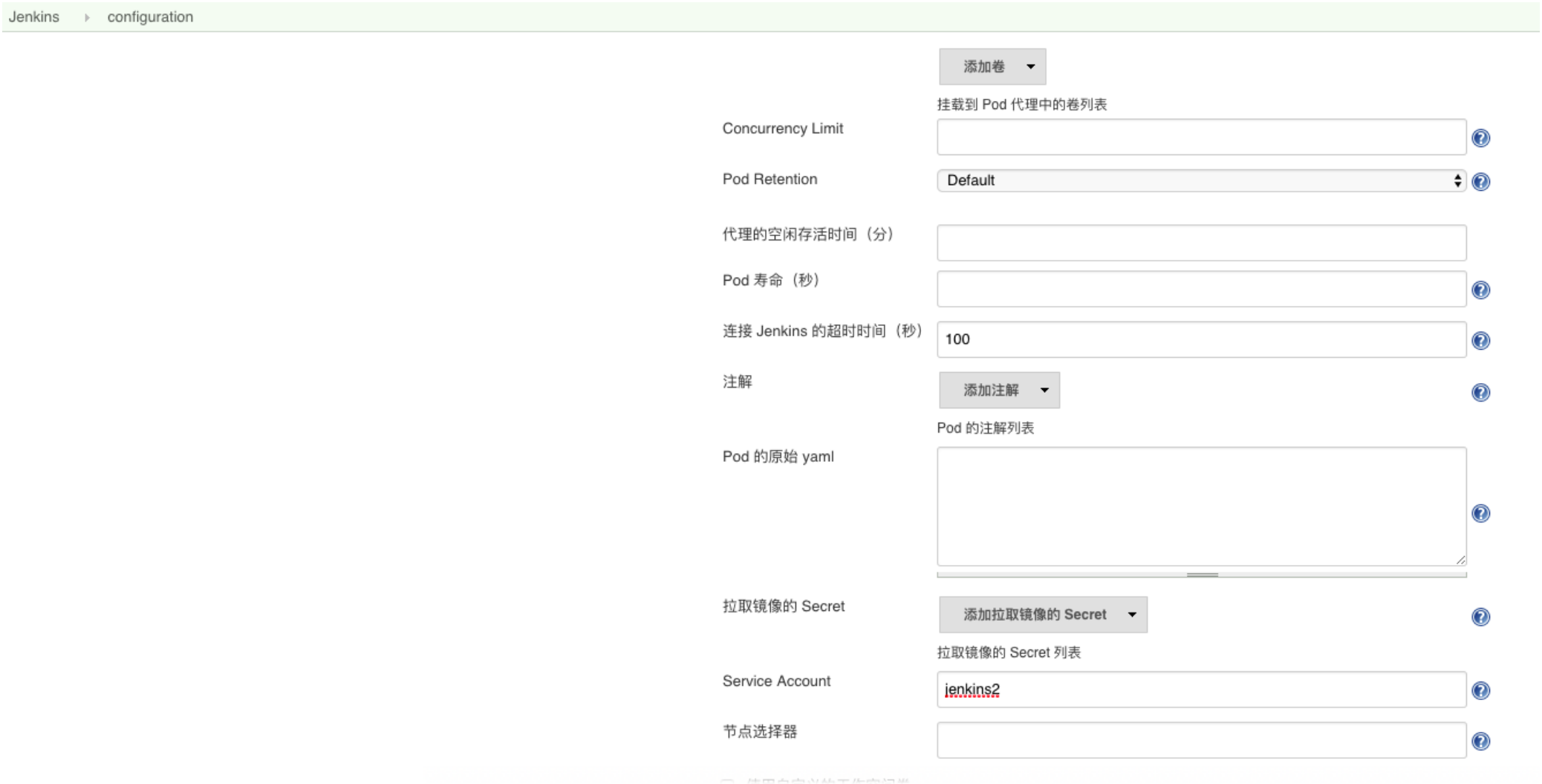
- 标签列表，这里我们填 ”jnlp-slave“，这个标签我们在后面创建Jobs会用到，非常重要。
- 用法，选择 ”尽可能使用这个节点“
- Docker镜像，填写”uhub.service.ucloud.cn/library/jenkins:jnlp“，这个容器镜像是我们CI/CD的运行环境。
- 工作目录，填写”/home/jenkins“



选择添加卷，主机路径和挂载路径都填写为”/var/run/docker.sock“，使得jenkins-slave可以使用宿主机的Docker，让我们可以在容器中进行镜像Build等操作。



点击最下方的Advanced，Service Account 输入jenkins2，这是我们之前创建的SA。

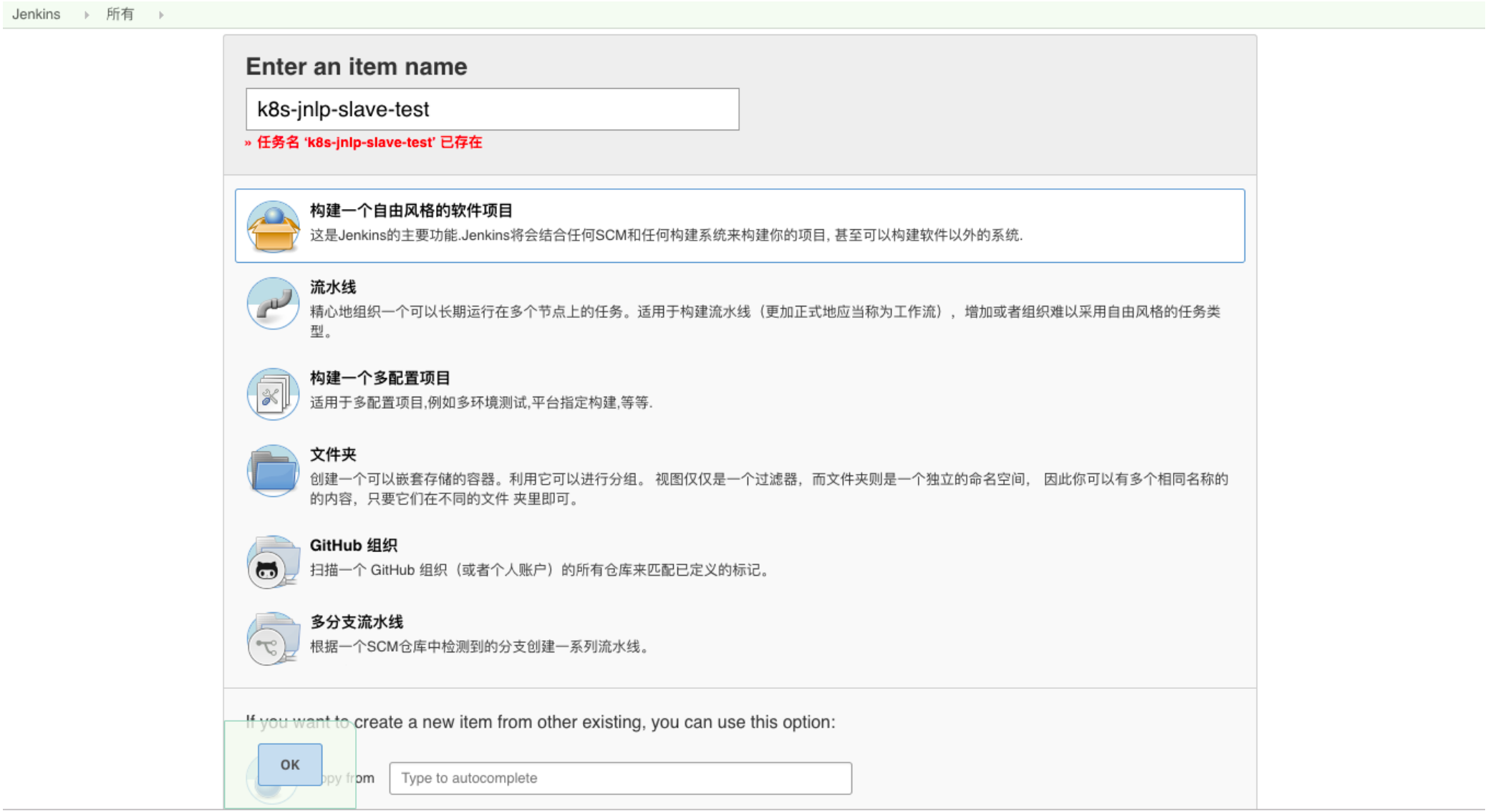


其他几个参数由于只是演示，我们都使用默认值，在实际使用的时候，请自行选择合理的参数。到这里我们的 Kubernetes Plugin 插件就算配置完成了。

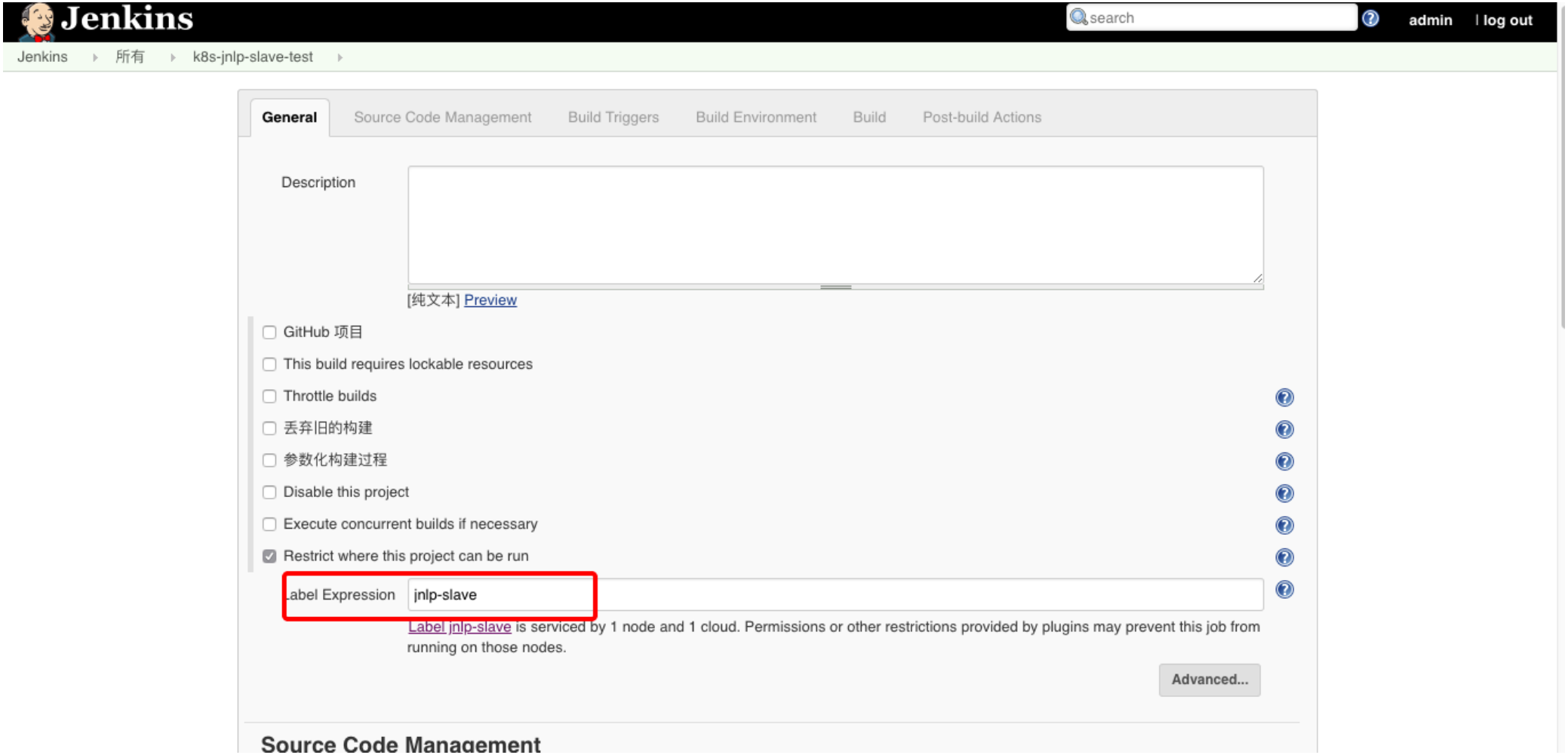
五、运行一个简单任务

Kubernetes 插件的配置工作完成了，接下来我们就来添加一个 Job 任务，看是否能够在 Slave Pod 中执行，任务执行完成后看 Pod 是否会被销毁。

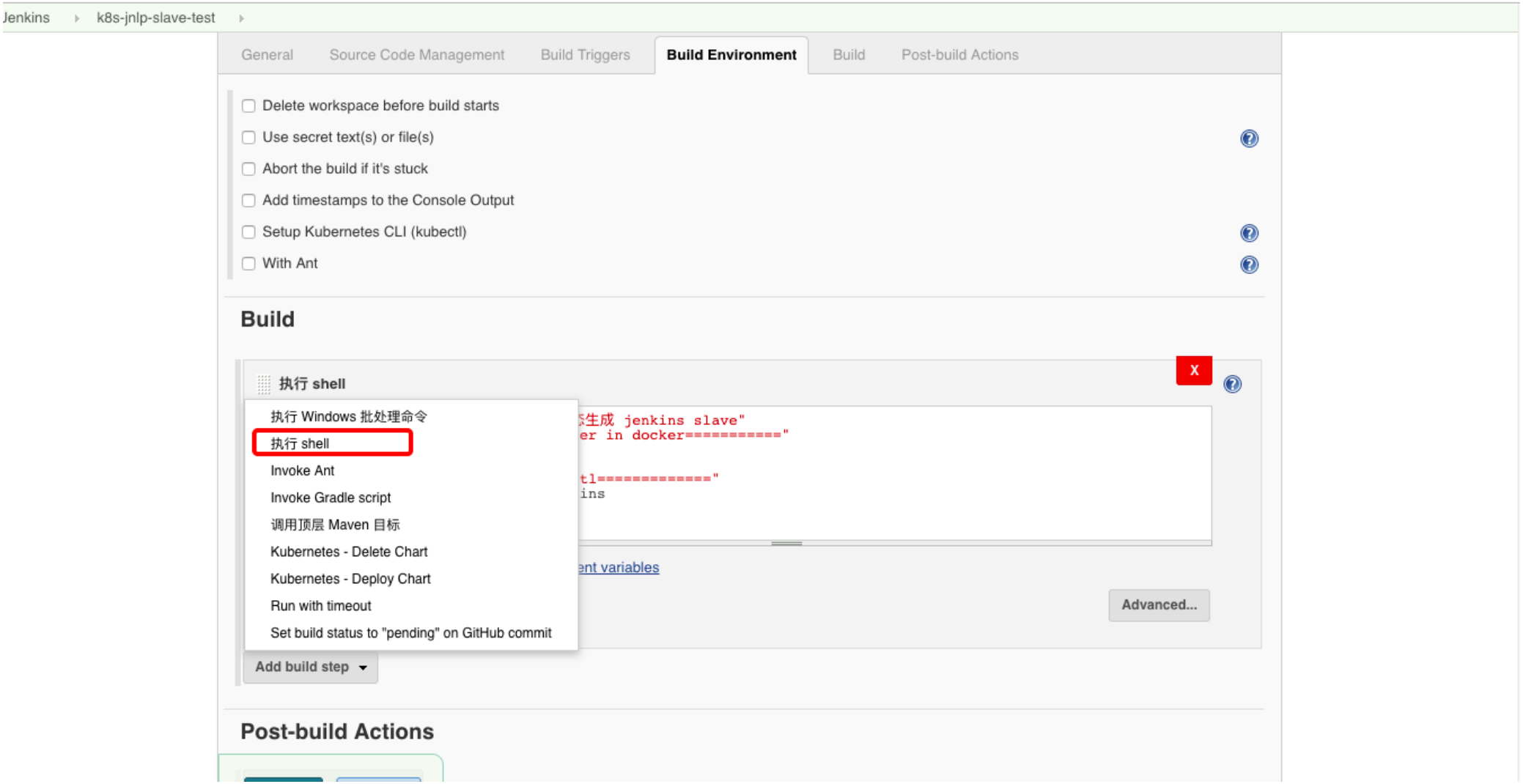
1、在 Jenkins 首页点击create new jobs，创建一个测试的任务，输入任务名称，然后我们选择 Freestyle project 类型的任务，点击OK。



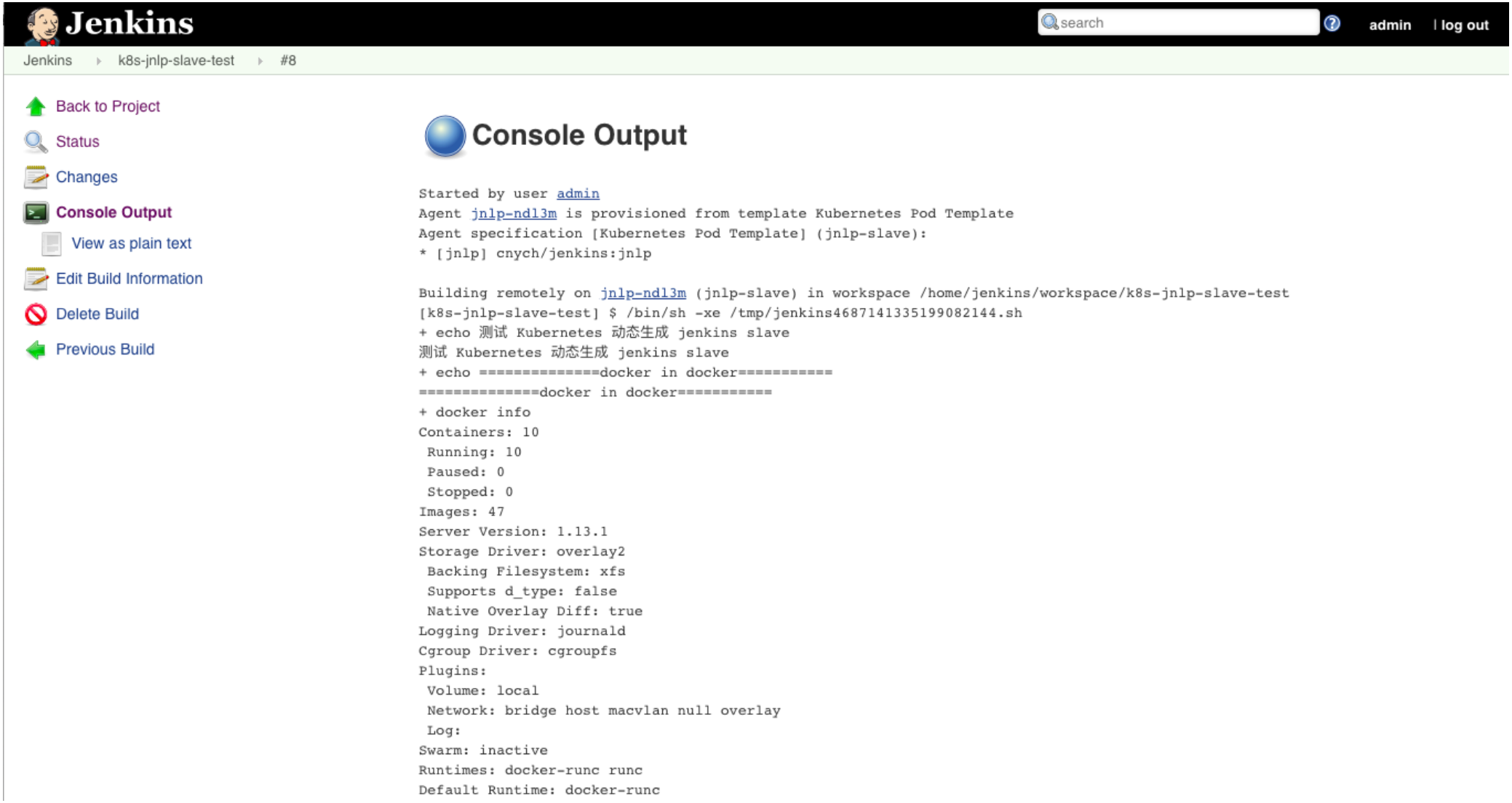
2、在任务配置页，最下面的 Label Expression 这里要填入jnlp-slave，就是前面我们配置的 Slave Pod 中的 Label，这两个地方必须保持一致



3、在任务配置页的 Build 区域，选择Execute shell，输入一个简单的测试命令，并点击保存。



4、点击查看Console output，查看任务运行情况。



到这里我们就完成了使用 Kubernetes 动态生成 Jenkins Slave 的方法。

六、运行一个pipeline任务

1. pipeline介绍

Pipeline，简单来说，就是一套运行在 Jenkins 上的工作流（流水线）框架，将原来独立运行于单个或者多个节点的任务连接起来，实现单个任务难以完成的复杂流程编排和可视化的工作。Jenkins Pipeline 有几个核心概念：

- **Node：**节点，一个 Node 就是一个 Jenkins 节点，是执行 Step 的具体运行环境，比如我们之前动态运行的 Jenkins Slave 就是一个 Node 节点。
- **Stage：**阶段，一个 Pipeline 可以划分为若干个 Stage，每个 Stage 代表一组操作，比如：Build、Test、Deploy，Stage 是一个逻辑分组的概念，可以跨多个 Node。
- **Step：**步骤，Step 是最基本的操作单元，可以是打印一句话，也可以是构建一个 Docker 镜像，由各类 Jenkins 插件提供，比如命令：sh ‘make’，就相当于我们平时 shell 终端中执行 make 命令一样。

2. 创建pipeline 任务

Pipeline 有两种创建方法，一是直接在 Jenkins 的 Web UI 界面中输入脚本，二是通过创建一个 Jenkinsfile 脚本文件放入项目源码库中，这里为了方便演示，我们使用在 Web UI 界面中输入脚本的方式来运行Pipeline。

1、点击”new item“，输入Job名称，选择Pipeline，点击"OK"。

2、在最下方的pipeline 脚本部分，输入以下脚本内容，并点击保存

```
node('jnlp-slave') {
stage('Clone') {
echo "1.Clone Stage"
}
stage('Test') {
echo "2.Test Stage"
}
stage('Build') {
echo "3.Build Stage"
}
stage('Deploy') {
echo "4. Deploy Stage"
}
}
```

上面的脚本内容中，我们给 node 添加了一个 jnlp-slave 标签，指定这个pipeline的4个stage，都运行在jenkins的slave节点中。

3、任务创建好之后，点击”立即构建“，我们可以通过kubectl命令发现UK8S集群中正启动一个新的pod用于构建任务。

```
bash-4.4# kubectl get po -n jenkins
NAME READY STATUS RESTARTS AGE
jenkins-deployment-6f9d84f745-lcs67 1/1 Running 0 5d2h
jnlp-0qn7x 0/1 ContainerCreating 0 1s
```

4、回到 Jenkins 的 Web UI 界面中查看 本次构建历史的 Console Output，也可以类似如下的信息，表明构建成功

```
Console Output
Started by user kukkazhang
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Still waiting to schedule task
‘jnlp-7m9dl’ is offline
Agent jnlp-7m9dl is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (jnlp-slave):
* [jnlp] uhub.service.ucloud.cn/library/jenkins:jnlp

Running on jnlp-7m9dl in /home/jenkins/workspace/testhelloworld
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Clone)
[Pipeline] echo
1.Clone Stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
```

```
....
....
6. Deploy Stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

七、在UK8S中部署应用

上面我们已经知道了如何在 Jenkins Slave 中构建Pipeline任务，那么如何通过Jenkins来部署一个原生的 Kubernetes 应用呢？

一般而言，在Kubernetes中部署一个业务的流程大致如下：

- 1.编写代码
- 2.测试
- 3.编写 Dockerfile
- 4.构建 Docker 镜像
- 5.推送 Docker 镜像到仓库
- 6.编写 Kubernetes YAML 文件
- 7.更改 YAML 文件中的 Docker 镜像 TAG
- 8.利用 kubectl 工具部署应用

这是我们人肉部署应用的流程，现在我们要做的是把上面这些流程放入 Jenkins 中来自动帮我们完成，从测试到更新 YAML 文件属于 CI 流程，后面部署属于 CD 的范畴。下面我们现在要来编写一个 Pipeline 的脚本，帮我们自动完成以上工作。

开始之前的准备工作

为了演示方便，我们准备了一个简单的helloworld程序，并将业务代码、dockerfile、yaml 放置 github代码仓库,分支:jenkins-cicd,接下来我们来逐步编写Pipeline脚本。

1、clone代码，我们将git commit的记录作为后面构建的镜像 tag，让镜像tag和git commit记录对应起来，方便后续排查问题。

```
stage('Clone') {
echo "1.Clone Stage"
git branch: 'jenkins-cicd', url: "https://github.com/ucloud/uk8s-demo.git"
script {
build_tag = sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
}
}
```

2、编写测试用例，这里涉及到业务逻辑，我们选择略过。

```
stage('Test') {
echo "2.Test Stage"
}
```

3、构建镜像，镜像tag便是我们之前在clone代码阶段定义的build_tag，注意将"jenkins_k8s_cicd"更换成您自己的uhub仓库名。

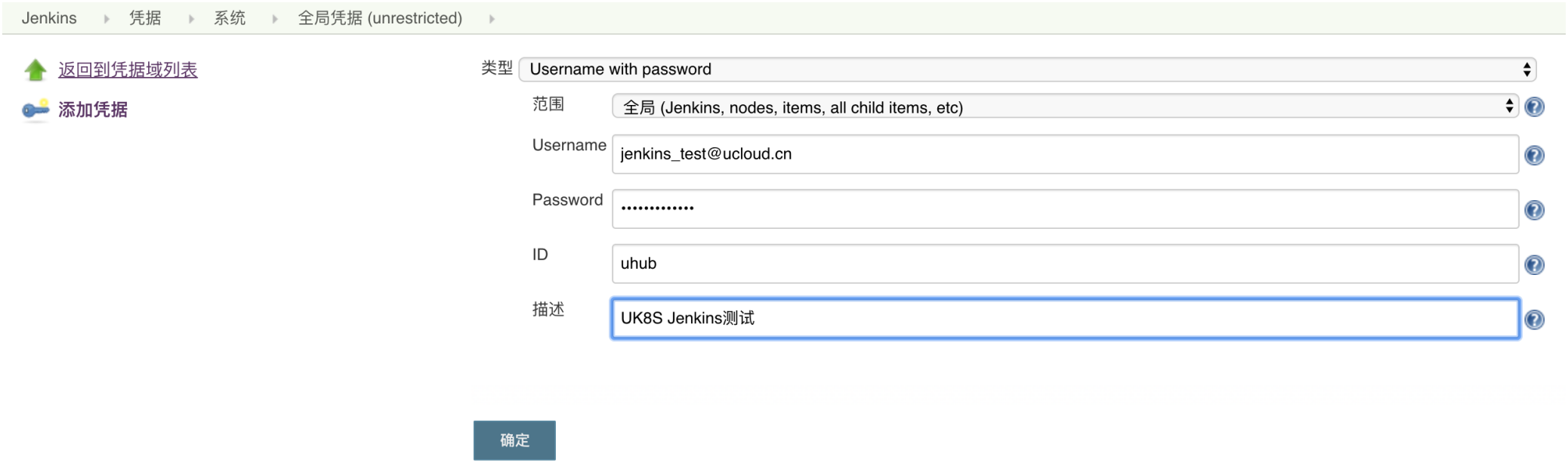
```
stage('Build') {
echo "3.Build Docker Image Stage"
sh "docker build -t uhub.service.ucloud.cn/jenkins_k8s_cicd/jenkins_k8s_cicd:${build_tag} ."
}
```

4、将镜像推送到镜像仓库。我们选择将镜像推送到Uhub的私人仓库中去，因此我们还需要登录到uhub，注意将"jenkins_k8s_cicd"更换成您自己的uhub仓库名。

```
stage('Push') {
echo "4.Push Docker Image Stage"
withCredentials([usernamePassword(credentialsId: 'uhub', passwordVariable: 'uhubPassword', usernameVariable: 'uhubUser')]) {
echo "${uhubPassword}"
echo "${uhubUser}"
sh "docker login -u ${uhubUser} -p ${uhubPassword} uhub.service.ucloud.cn"
sh "docker push uhub.service.ucloud.cn/jenkins_k8s_cicd/jenkins_k8s_cicd:${build_tag}"
}
}
```

为了保证账户安全，上面的脚本中，我们用了Jenkins的一个"凭据"功能。在首页点击 Credentials -> Stores scoped to Jenkins 下面的 Jenkins -> Global credentials (unrestricted) -> 左侧的 Add Credentials：添加一个 Username with password 类型的认证信息。

输入 uhub 的用户名和密码，ID 部分我们输入uhub，注意，这个值非常重要，需要与Pipeline 中的脚本保持一致。



5、更新yaml文件，这里我们只是将yaml中的镜像tag更换成最新构建出的镜像tag。

```
stage('YAML') {
echo "5. Change YAML File Stage"
sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yml"
}
```

6、应用发布。我们直接使用kubectl apply命令来更新应用，还记得我们之前创建的名为jenkins2的ServiceAccount吗?能成功发布应用，还有赖我们为jenkins2配置的权限呢。

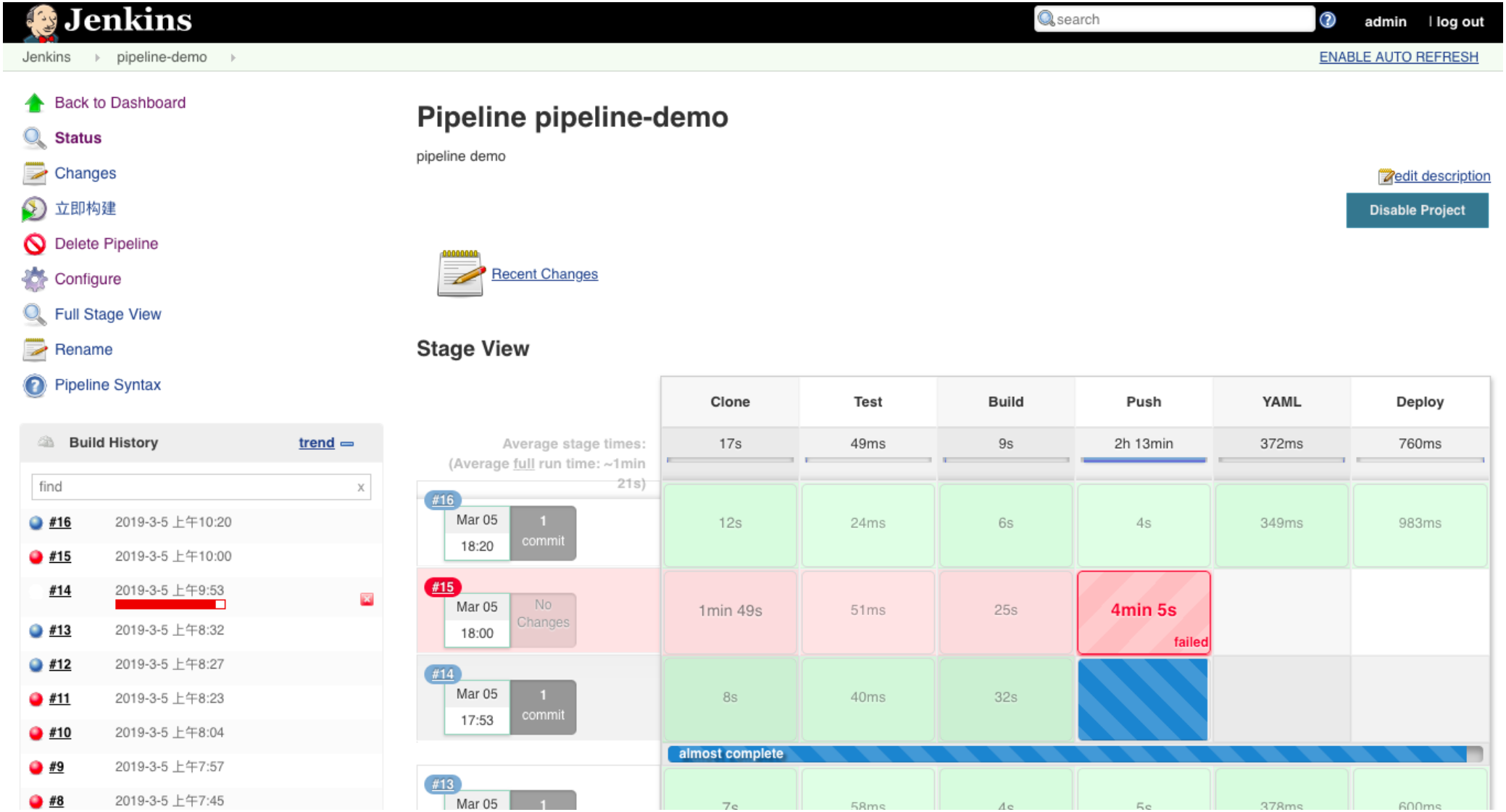
```
stage('Deploy') {
echo "6. Deploy Stage"
sh "kubectl apply -f k8s.yml"
```

```
}
```

7、上面我们把pipeline中的每个Stage都讲述了一遍，下面我们把6个stage的脚本合并到一起，创建一个新的流水线任务，体验下完整的应用发布流程吧。

```
node('jnlp-slave') {
stage('Clone') {
echo "1.Clone Stage"
git branch: 'jenkins-cicd', url: 'https://github.com/ucloud/uk8s-demo.git'
script {
build_tag = sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
}

}
stage('Test') {
echo "2.Test Stage"
}
stage('Build') {
echo "3.Build Docker Image Stage"
sh "docker build -t uhub.service.ucloud.cn/jenkins_k8s_cicd/jenkins_k8s_cicd:${build_tag} ."

}
stage('Push') {
echo "4.Push Docker Image Stage"
withCredentials([usernamePassword(credentialsId: 'uhub', passwordVariable: 'uhubPassword', usernameVariable: 'uhubUser')]) {
echo "${uhubPassword}"
```

```
echo "${uhubUser}"
sh "docker login -u ${uhubUser} -p ${uhubPassword} uhub.service.ucloud.cn"
sh "docker push uhub.service.ucloud.cn/jenkins_k8s_cicd/jenkins_k8s_cicd:${build_tag}"
}

}
stage('YAML') {
echo "5. Change YAML File Stage"
sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yml"
}
stage('Deploy') {
echo "6. Deploy Stage"
sh "kubectl apply -f k8s.yml"
}
}
```


Jenkins
search
admin
log out
Jenkins
pipeline-demo
ENABLE AUTO REFRESH
Back to Dashboard
Status
Changes
立即构建
Delete Pipeline
Configure
Full Stage View
Rename
Pipeline Syntax
Build History
trend
find
#16 2019-3-5 上午10:20
#15 2019-3-5 上午10:00
#14 2019-3-5 上午9:53
#13 2019-3-5 上午8:32
#12 2019-3-5 上午8:27
#11 2019-3-5 上午8:23
#10 2019-3-5 上午8:04
#9 2019-3-5 上午7:57
#8 2019-3-5 上午7:45
Pipeline pipeline-demo
pipeline demo
edit description
Disable Project
Recent Changes
Stage View
Clone
Test
Build
Push
YAML
Deploy
Average stage times:
(Average full run time: ~1min 21s)
17s
49ms
9s
2h 13min
372ms
760ms
#16
Mar 05
18:20
1 commit
12s
24ms
6s
4s
349ms
983ms
#15
Mar 05
18:00
No Changes
1min 49s
51ms
25s
4min 5s
failed
#14
Mar 05
17:53
1 commit
8s
40ms
32s
almost complete
#13
Mar 05
7s
58ms
4s
5s
378ms
600ms

基于Jenkins的CI/CD实践(kaniko版本)

- 适用集群版本 1.14~1.22

前言

在之前的方案中，我们介绍了Docker+Jenkins实现容器镜像构建、业务部署的方案，该方案需要直接挂载docker socket文件到Jenkins slave容器中。由于UK8S 1.20以后的版本将采用Containerd,因此该方案不再适用。

这篇文章中，我们介绍基于Kaniko+Jenkins的CICD方案，Kaniko是谷歌开源的一款用来构建容器镜像的工具。与docker不同，Kaniko 并不依赖于Docker daemon进程，完全是在用户空间根据Dockerfile的内容逐行执行命令来构建镜像，这就使得在一些无法获取 docker daemon 进程的环境下也能够构建镜像，比如在标准的Kubernetes Cluster上。关于kaniko

一、部署Jenkins

1、为了管理方便，我们把需要创建的资源都部署在一个名为 jenkins 的 namespace 下面，所以我们需要添加创建一个 namespace：

```
kubectl create namespace jenkins
```

2、声明一个PVC对象，后面我们要将Jenkins容器的 /var/jenkins_home 目录挂载到了这个名为PVC对象上面。

- 使用SSD云盘作为存储，请部署。

```
kubectl apply -f https://docs.ucloud.cn/uk8s/yaml/cicd/yaml_jenkins_jenkins-pvc.yaml
```

3、以Deployment方式部署Jenkins master,为了演示方便，我们还使用LoadBalancer类型的service将其暴露到外网。

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
name: jenkins-deployment
namespace: jenkins
spec:
replicas: 1
selector:
matchLabels:
app: jenkins
template:
metadata:
labels:
app: jenkins
spec:
securityContext:
fsGroup: 1000
containers:
- name: jenkins
image: uhub.service.ucloud.cn/ucloud/jenkins:2.326
env:
- name: JENKINS_UC
value: https://mirrors.tuna.tsinghua.edu.cn/jenkins/updates/
- name: JENKINS_UC_DOWNLOAD
value: https://mirrors.tuna.tsinghua.edu.cn/jenkins/
ports:
```

```
- containerPort: 8080
name: web
protocol: TCP
- containerPort: 50000
name: agent
protocol: TCP
volumeMounts:
- name: jenkins-strorage
mountPath: /var/jenkins_home
volumes:
- name: jenkins-strorage
persistentVolumeClaim:
claimName: jenkins-pvc-claim
---
apiVersion: v1
kind: Service
metadata:
name: jenkins
namespace: jenkins
labels:
app: jenkins
spec:
type: LoadBalancer
ports:
- name: web
```

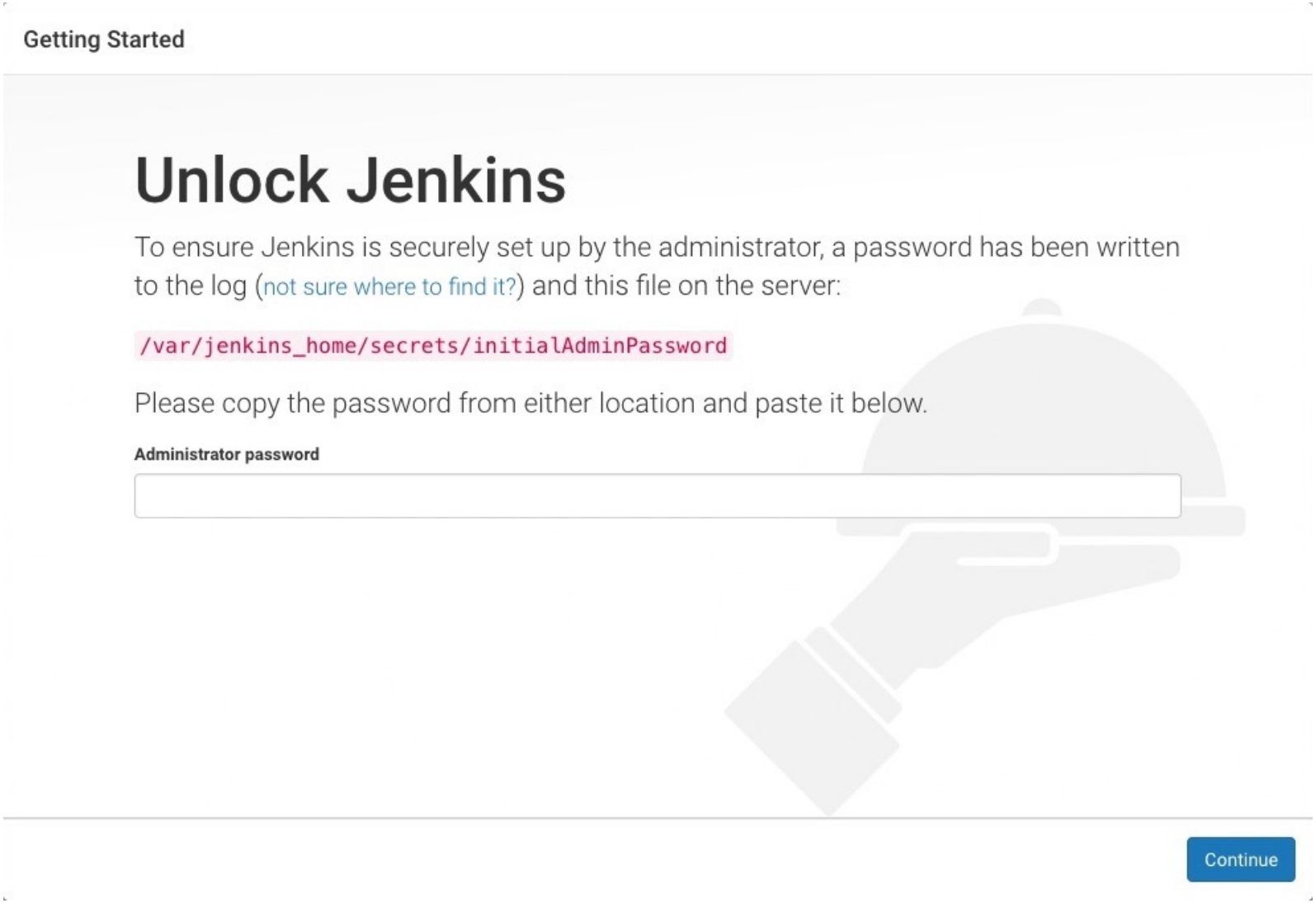
```
port: 8080
targetPort: web
- name: agent
port: 50000
targetPort: agent
selector:
app: jenkins
```

4、等到服务启动成功后，我们就可以根据LoadBalancer的IP（即EXTERNAL-IP），访问 jenkins 服务了，并根据提示信息进行安装配置。

```
bash-4.4# kubectl get svc -n jenkins
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT AGE
jenkins LoadBalancer 172.17.201.210 106.75.98.80 8080:33651/TCP,50000:43748/TCP 4d21h
```

三、安装Kubernetes插件

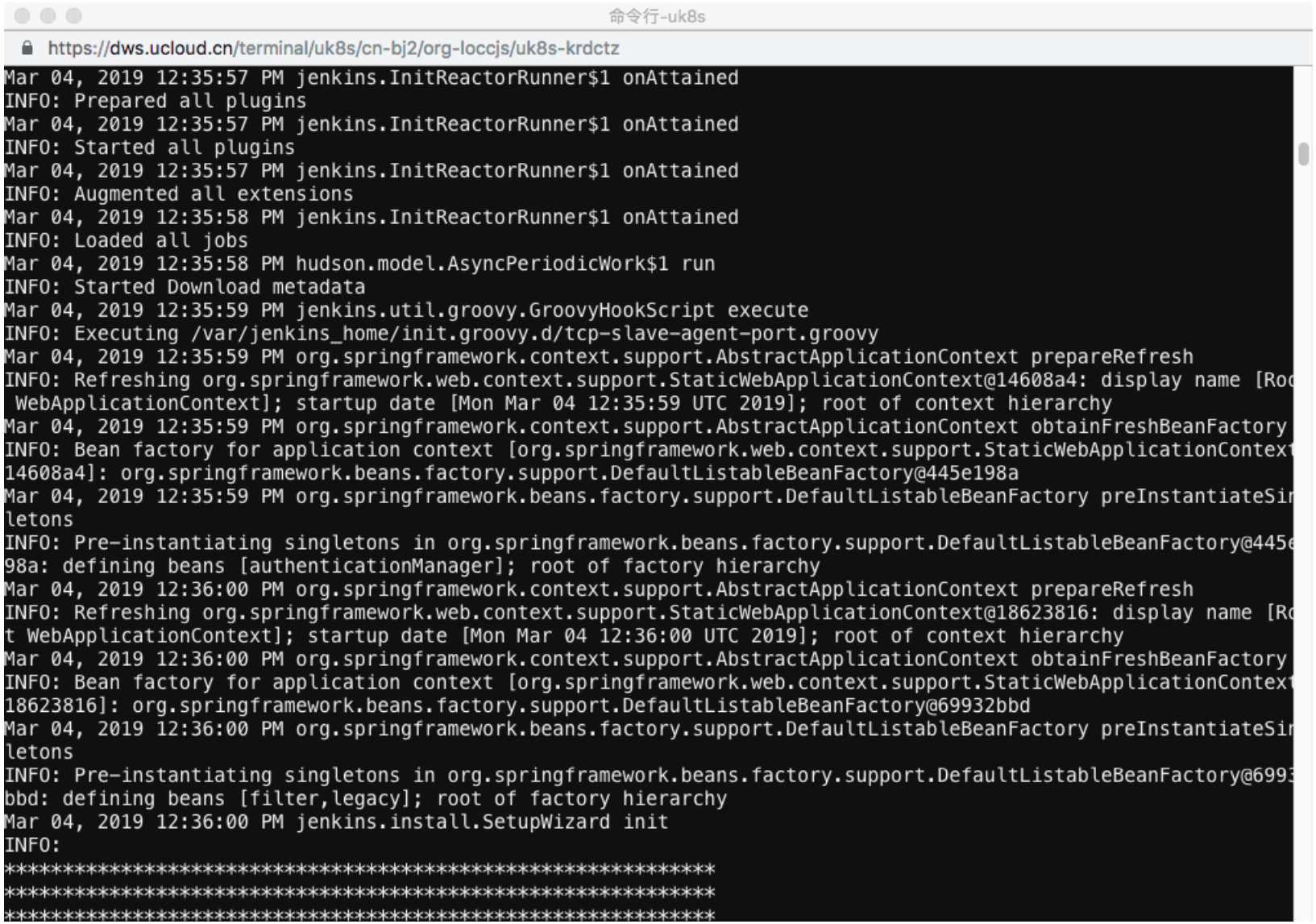
1、前面我们已经获取到Jenkins的外网IP地址，我们直接在浏览器输入EXTERNAL-IP:8080，即可打开Jenkins页面，提示需要输入初始化密码：



2、我们通过kubectl log获取jenkins容器的日志来获取初始化密码

```
kubectl -n jenkins get pod
```

kubectl -n jenkins logs jenkins-deployment-xxxxxxx

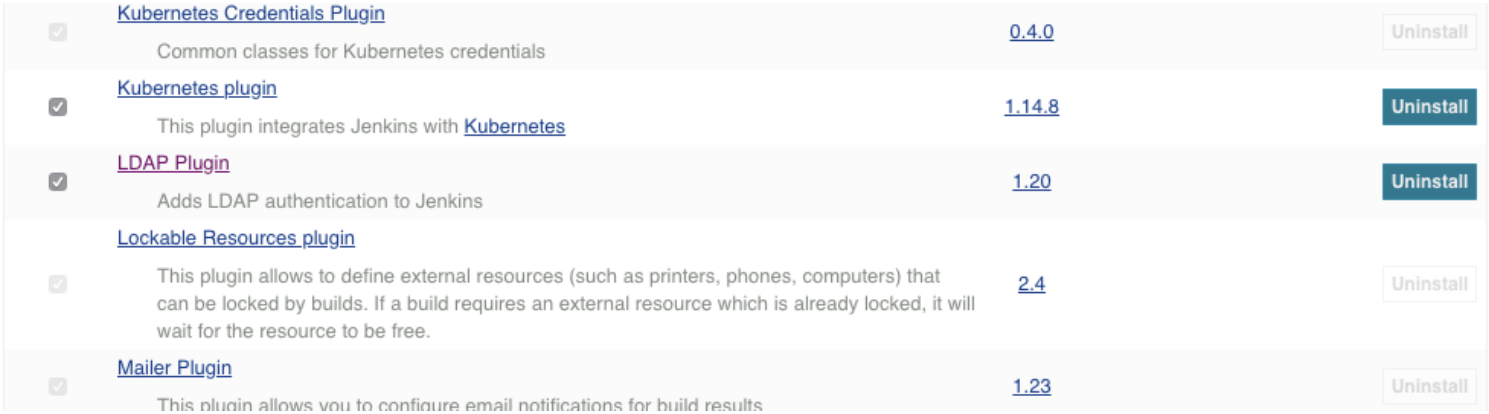


```
Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:
4805e19fc13e479288d0dd3ca45134a7
This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
************************************************************
************************************************************
************************************************************
```

3、进入安装插件页面，在配置之前，先开启代理兼容，访问 http://exter-ip:8080/configureSecurity/ ，勾选 **启用代理兼容**。这一步可能会出现报错，多尝试几次即可。

4、选择推荐安装，安装完毕后，添加管理员帐号，即可进入到 jenkins 主界面。

5、接下来安装jenkins依赖插件清单——kubernets plugin，让他能够动态的生成 Slave 的 Pod。点击 Manage Jenkins -> Manage Plugins -> Available -> Kubernetes plugin 勾选安装即可。



安装插件相对较慢，请耐心等待，并且由于是在线安装，集群需要开通外网，请开启natgw来使node节点通外网

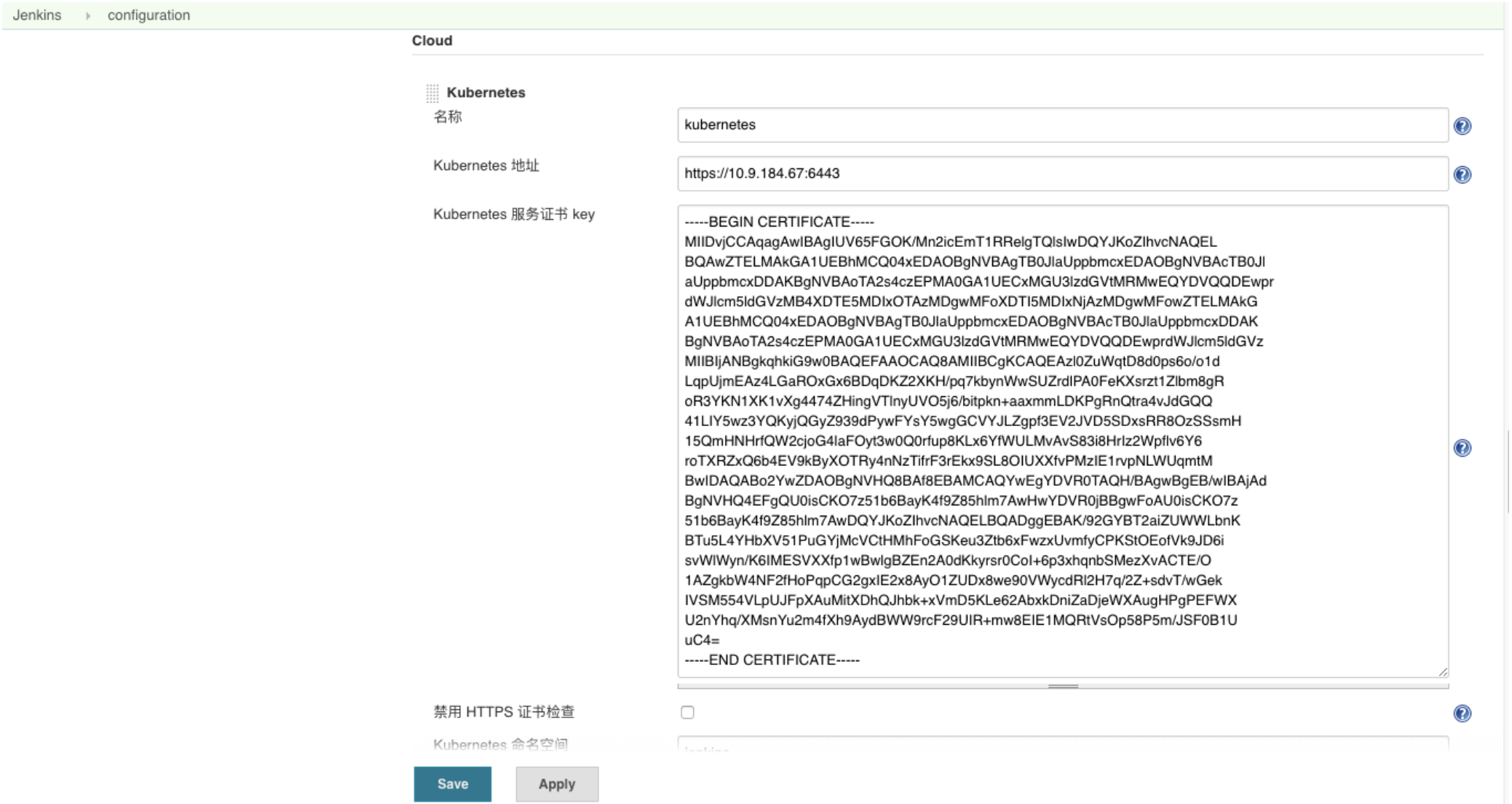
四、配置Jenkins

接下来将进入最重要的一个步骤，在Kubernetes插件安装完毕后，我们需要配置Jenkins和Kubernetes参数，使Jenkins连接到UK8S集群，并能调用Kubernetes API 动态创建Jenkins Slave，执行构建任务。

首先点击 Manage Jenkins —> Configure System，进入到系统设置页面。滚动到页面最下方，然后点击Add a new cloud —> 选择 Kubernetes，开始填写 Kubernetes 和 Jenkins 配置信息。

1、输入UK8S Apiserver地址，以及服务证书key。

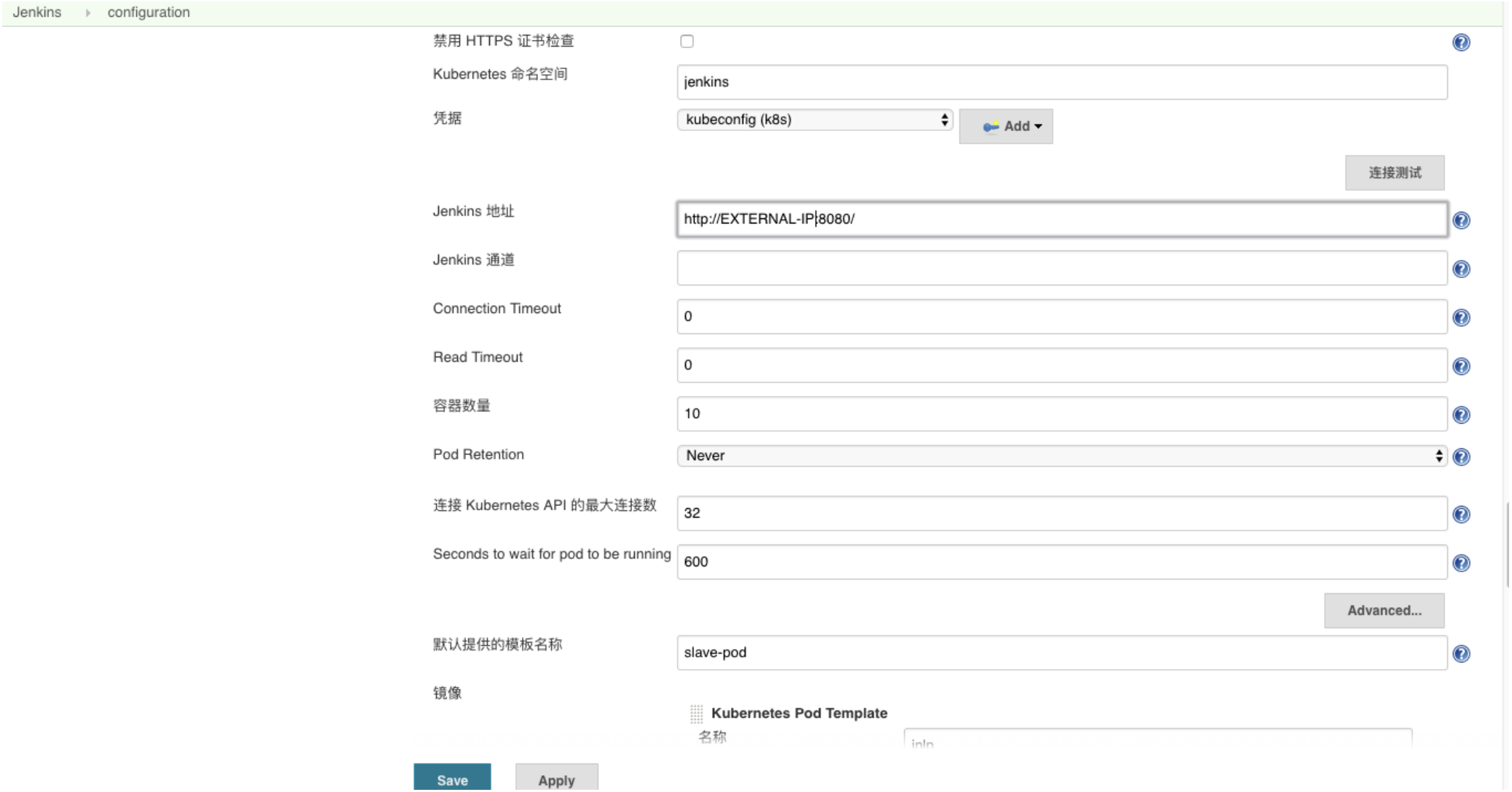
以上两个参数信息，可以在UK8S集群详情页的**内网凭证**中获取。"服务证书key"为集群凭证中的certificate-authority-data字段内容，进行base64解码，将解码后的内容复制到输入框即可。

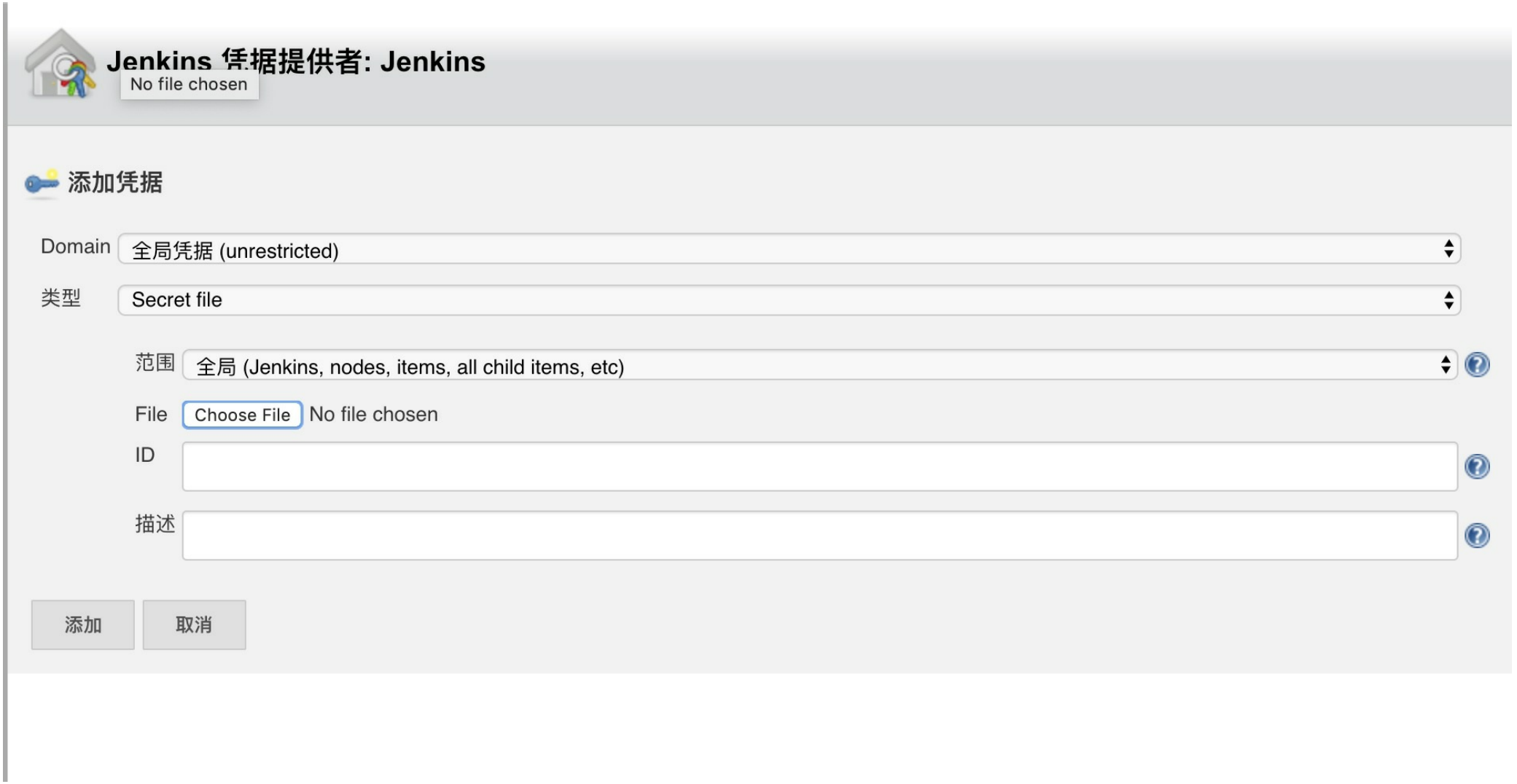


2、填写集群Namespace、上传凭据、Jenkins地址

- Namespace此处填写之前创建Namespace即可，此处为jenkins。

- 凭证处，点击”Add“，凭证类型选择"Secret file"，将UK8S集群详情页 内网凭证内容复制下来，保存为kubeconfig上传。
- Jenkins地址为 `kubectl -n jenkins get svc` 获取到的地址。





3、点击”连接测试“,如果出现 Connection test successful 的提示信息证明 Jenkins 已经可以和 Kubernetes 系统正常通信了

五、为kaniko配置凭证

创建一个配置kaniko推送到uhub镜像的凭证，找一台有安装docker的云主机，登录一次uhub：

```
docker login uhub.service.ucloud.cn -u user@ucloud.cn
```

登入成功之后，会生成一个config.json的文件，使用该文件创建一个secret供kaniko容器使用。

- centos环境中文件位置：/root/.docker/config.json
- ubuntu环境中文件位置：/home/ubuntu/.docker/config.json

把config.json拷贝到master节点根目录执行以下命令创建secret

```
kubectl -n jenkins create secret generic regcred --from-file=config.json
```

到此配置结束。

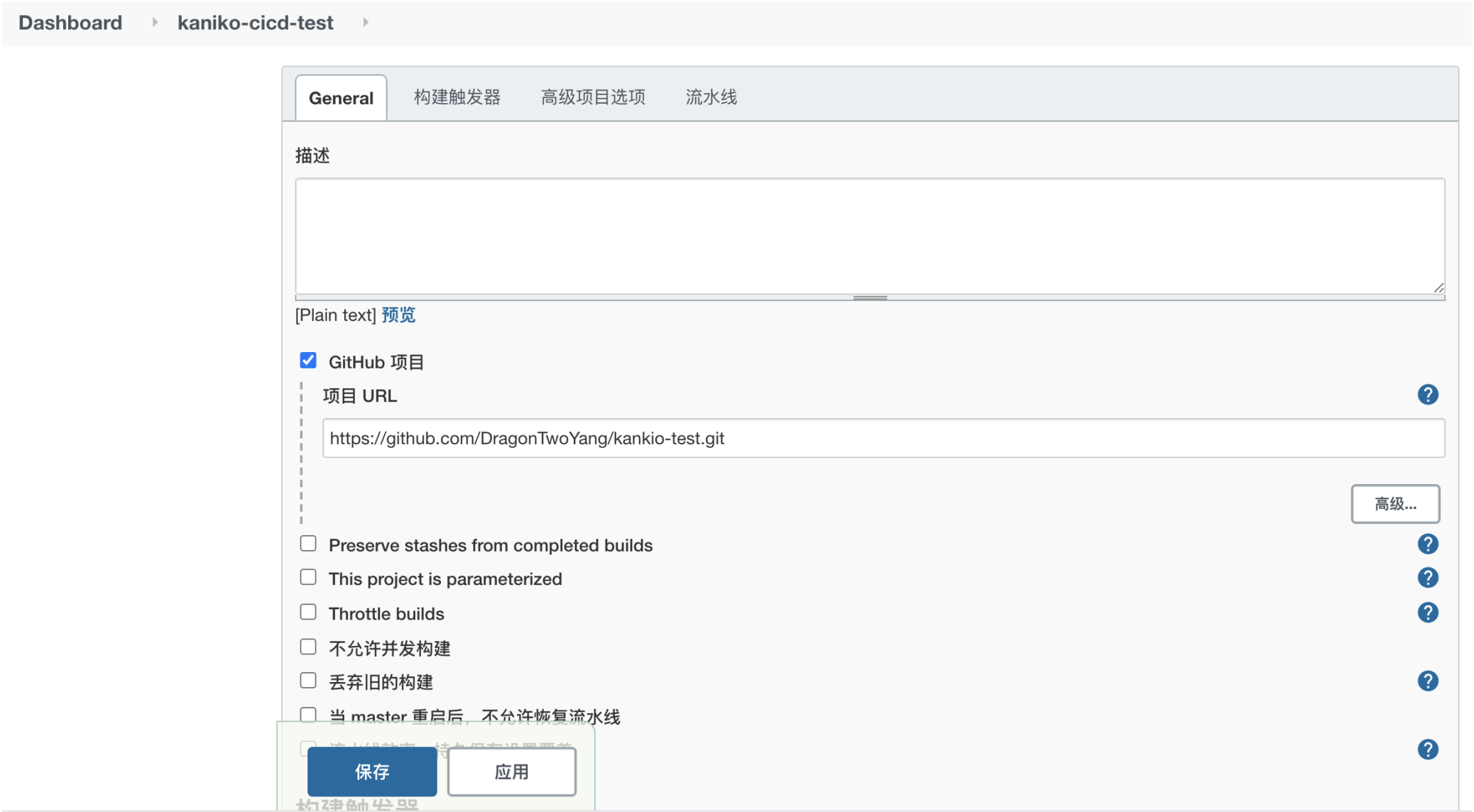
六、创建Job

Kubernetes 插件的配置工作完成了，接下来我们就来添加一个 Job 任务，看是否能够在 Slave Pod 中执行，任务执行完成后看 Pod 是否会被销毁。

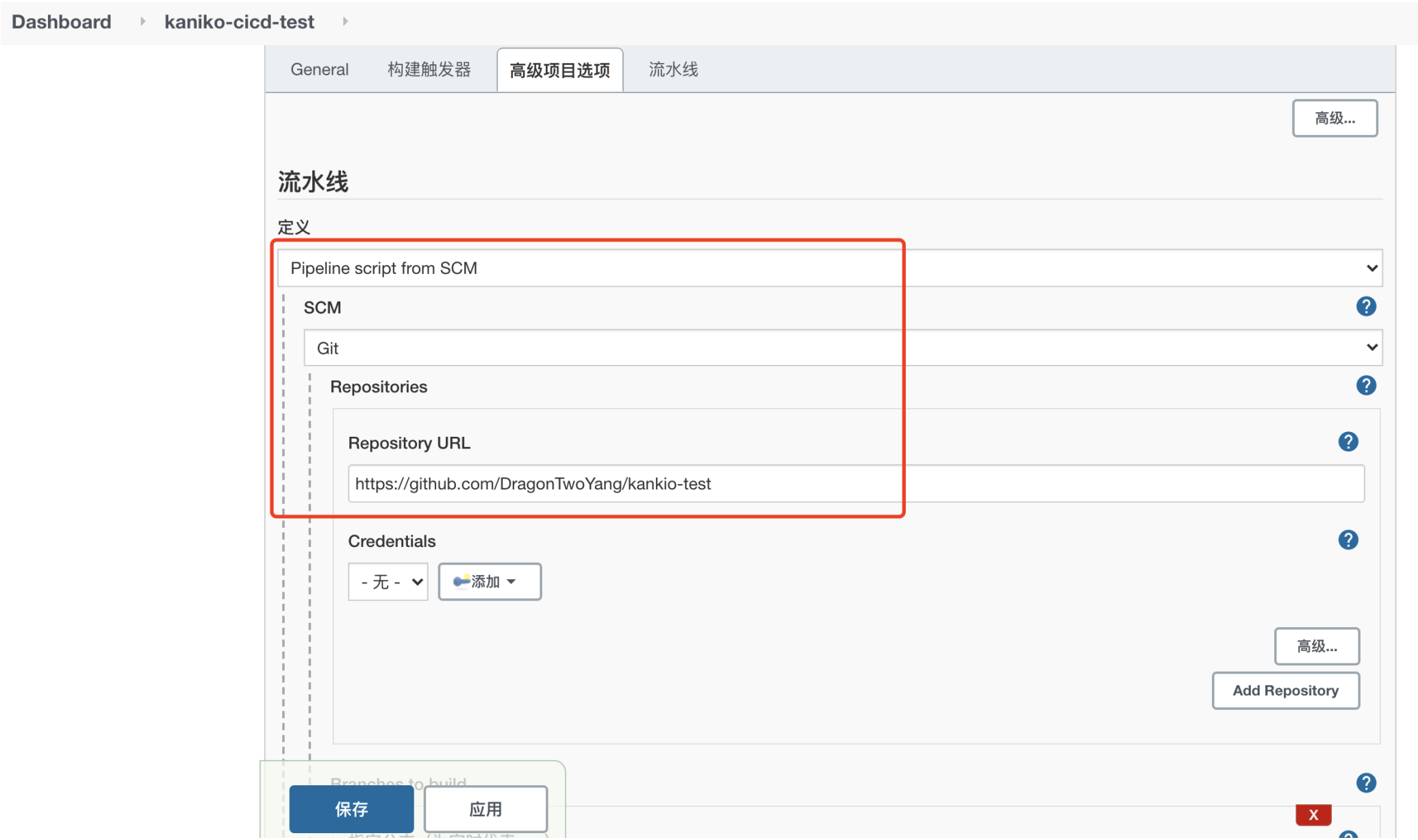
为了方便使用，我们提供了一个golang项目的ci/cd jenkins-kaniko-cicd 里面包含了完整的编译，构建镜像，部署流程。原方案中关于Slave Pod以及CICD的配置信息都配置在Jenkinsfile中，你可以根据项目自身需求更改Jenkinsfile文件。

1、在 Jenkins 首页点击create new jobs，创建一个测试的任务，输入任务名称，然后我们选择“流水线”类型的任务，点击OK。

2、在任务配置页，这里我们选择GitHub项目，输入kaniko的任务地址。



3、在任务配置页的流水线区域，选择Pipeline script from SCM，并选择 Master 分支以及 Jenkinsfile的文件路径。。


Dashboard
kaniko-cicd-test
General
构建触发器
高级项目选项
流水线
高级...
流水线
定义
Pipeline script from SCM
SCM
Git
Repositories
Repository URL
https://github.com/DragonTwoYang/kankio-test
Credentials
- 无 -
添加
高级...
Add Repository
保存
应用

亲和性实践

Kubernetes 提供了多种节点分配使用的方法，常用的有以下4种：

- 节点筛选器(nodeSelector)
- 节点亲和与反亲和性(nodeAffinity)
- Pod亲和与反亲和性(podAffinity)
- 节点隔离/限制

1. 标签

在做节点选择的时候很多时候用到了 Kubernetes 的 Label 功能，这里我们分别展示给节点及 Pod Label 的方法。

1. 给 10.10.10.10 节点增加 `disktype=ssd` 标签

```
# kubectl label nodes 10.10.10.10 disktype=ssd
```

2. 给 Pod 增加 `disktype=ssd` 标签，同理可以针对 Deployment、Service 等对象进行增加标签操作

```
# kubectl label po unginx-7db67b8c69-zcxmm disktype=ssd
```

2. 节点筛选器（nodeSelector）

节点筛选器，用于创建 Pod（及 Deployment、StatefulSet 等控制器时），将 Pod 分配给相应的节点。

实例 yaml 创建了一个 Pod 对象，在 `spec.nodeSelector` 下以 Map 形式增加改容器部署的节点的限制条件，nodeSelector 会筛选拥有 `disktype: ssd` 的 Node 节点进行 Pod 部署。

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
labels:
env: test
spec:
containers:
- name: nginx
image: nginx
imagePullPolicy: IfNotPresent
nodeSelector:
disktype: ssd
```

3. 节点亲和与反亲和性（nodeAffinity）

使用节点亲和性与反亲和性，需要在 Deployment `spec.template.spec` 下增加 `affinity` 字段，节点亲和分为**硬匹配和软匹配** 两种。

3.1 硬匹配

在以下 yaml 示例中，requiredDuringSchedulingIgnoredDuringExecution 可以理解为**排除不具备指定 Label 的节点**，如果节点不含有 ucloud=yes 则 Pod 不会被分配到该节点。

nodeSelectorTerms 下提供了 matchExpressions（匹配表达式）和 matchFields（匹配字段），选择使用其一，我们这里使用了 matchExpressions，下面的表达式中 Key 和 Values对应，operator 的可选参数有In、NotIn、Exists、DoesNotExist、Gt、Lt，这里可以设置 NotIn、DoesNotExist 进行反亲和的设置，也就是如果这里写入了 operator: NotIn 的时候，Pod 将分配在没有 ucloud=yes Label 的节点上。

```
apiVersion: apps/v1
kind: Deployment
metadata:
creationTimestamp: null
labels:
run: ucloud
name: ucloud
spec:
replicas: 3
selector:
matchLabels:
run: ucloud
template:
metadata:
creationTimestamp: null
labels:
run: ucloud
spec:
```

```
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: ucloud
operator: In
values:
- "yes"
containers:
- image: nginx
name: ucloud
ports:
- containerPort: 80
resources: {}
```

3.2 软匹配

和 requiredDuringSchedulingIgnoredDuringExecution 对应还有 preferredDuringSchedulingIgnoredDuringExecution，这里称为软匹配，这个参数为对应节点进行打分，降低不具备 Label 的节点的选中几率。

这里的 weight 字段在1-100范围内。对于满足所有调度要求的每个节点，调度程序将通过迭代此字段的元素计算总和，并在节点与对应的节点匹配时将「权重」添加到总和 MatchExpressions，然后将该分数与节点的其他优先级函数的分数组合。总得分最高的节点是最优选的。

```
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 1
```

```
preference:
matchExpressions:
- key: another-node-label-key
operator: In
values:
- another-node-label-value
```

3.3 说明

1. 如果同时指定 nodeSelector 和 nodeAffinity，节点必须满足全部条件，Pod 才会被调度到该节点。
2. 如果指定了多个 nodeSelectorTerms 关联 nodeAffinity 类型，如果能满足其中一个 nodeSelectorTerms，则 Pod 就可以调度到这个节点。
3. 如果指定了多个 matchExpressions 关联的 nodeSelectorTerms，则只有在节点满足所有 matchExpressions 要求情况下才能将该容器调度到节点上。
4. 如果删除或更改已有容器节点的 Label，Kubernetes 不会主动删除该容器，亲和的调度选择仅在调度 Pod 时起作用。

CVE-2019-5736

UK8S已于2019年2月14日15:00修复runc容器逃逸漏洞，并通过攻防测试。本文主要介绍2019年2月14日之前创建的集群修复办法。

漏洞详情

runc被曝存在容器逃逸漏洞。该漏洞允许恶意容器(以最少的用户交互)覆盖host上的runc文件，从而在host上以root权限执行代码。在下面两种情况下，通过用户交互可以在容器中以root权限执行任意代码：1.使用攻击者控制的镜像创建新容器。2.进入到攻击者之前具有写入权限的现有容器中(docker exec)。

影响范围

** 2019年2月14日15:00之前创建的集群**

此批集群的Docker版本为1.13.1，runc版本<1.0-rc6，存在安全隐患，需要修复。

2019年2月14日15:00之后创建的集群已修复该漏洞，并已通过攻防测试无需担心。

由于UK8S为单租户模式，如果之前未在UK8S集群内部署未经审查的第三方镜像，该漏洞无法被黑客被利用，但为了业务安全，建议尽早修复。

修复方案

方案一

对于测试用集群，建议删除后重新创建，新版本UK8S集群已修复该漏洞（CVE-2019-5736）；

方案二

仅升级runc版本，该方案为热升级方案，理论上不会导致业务中断，具体方案如下：

1.备份原有runc，UK8S的runc位于/usr/libexec/docker路径下。

```
mv /usr/libexec/docker/docker-runc-current /usr/libexec/docker/docker-runc-current.$(date -Iseconds)
```

2.下载修复的runc版本（对应的容器版本为1.13.1，内核版本为4.x），并替换原有的runc。

```
wget https://github.com/rancher/runc-cve/releases/download/CVE-2019-5736-build3/runc-v1.13.1-amd64
mv runc-v1.13.1-amd64 /usr/libexec/docker/docker-runc-current

##内核版本为3.x
wget https://github.com/rancher/runc-cve/releases/download/CVE-2019-5736-build3/runc-v1.13.1-amd64-no-memfd_create
```

3.配置可执行权限

```
chmod +x /usr/libexec/docker/docker-runc-current
```

4.测试新版本runc是否正常工作

```
/usr/libexec/docker/docker-runc-current -v
docker run -it --rm ubuntu echo OK
```

方案三

升级Docker版本。将已有集群的Docker版本升级到18.09.2或以上。该方案会导致容器和业务中断，请谨慎操作。

如在该漏洞过程中需要协助，请联系UK8S团队协助处理。

参考链接

https://www.openwall.com/lists/oss-security/2019/02/11/2

https://kubernetes.io/blog/2019/02/11/runc-and-cve-2019-5736/

HTTP/2漏洞升级说明

Go语言HTTP/2漏洞

- 发布时间:2019年8月13日
- 更新时间:2019年8月26日
- 漏洞等级:Important
- CVE编号:CVE-2019-9512 CVE-2019-9514

漏洞详情

Netflix、Google及CERT/CC 近日揭露了有关HTTP/2的8个安全漏洞,其中Go语言的net/http库存在着CVE-2019-9512与CVE-2019-9514两个安全漏洞,造成任何基于HTTP或HTTPS接听器的程序发生服务中断,而且波及K8s的所有版本及组件。

Go语言已经发布Go1.12.9与Go1.11.13来修补这两个漏洞,而K8s则是基于Go的修补发布K8s v1.15.3、K8s v1.14.6与K8s v1.13.10,建议K8s用户应尽快升级到最新版本。

官方参考文档

影响范围

2019年8月28日18:00之前创建的集群

此批集群的K8S版本低于官方给出的修复完成版本的编号。

2019年8月28日18:00之后创建的集群已修复该漏洞，并已通测试。

修复方案

UK8S根据官方提供的修补发布版本制作了k8s v1.13.10和v1.14.6升级安装包。

v1.13.10下载地址：http://uk8s.cn-bj.ufileos.com/1.13.10/k8s.tgz

v1.14.6下载地址：http://uk8s.cn-bj.ufileos.com/1.14.6/k8s.tgz

1. 下载安装包文件到服务器上，可使用命令

```
wget http://uk8s.cn-bj.ufileos.com/1.13.10/k8s.tgz
```

如您开通的是1.14.5请将wget的下载地址替换为1.14.6的下载地址

2. 解压下载完成的安装包

```
tar zxvf k8s.tgz
```

3. 执行更新脚本

master节点执行

```
chmod +x 1.13.10/uk8supgrade.sh
sh 1.13.10/uk8supgrade.sh master
```

node节点执行

```
chmod +x 1.13.10/uk8supgrade.sh
sh 1.13.10/uk8supgrade.sh node
```

如您开通的是1.14.5请将sh的执行路径更改为1.14.6的解压路径

注意事项

1. 升级操作需要对集群内所有master和node节点进行执行，master节点需要一个节点执行成功后再执行第二个节点的升级操作，可以通过**kubectl get cs**查看系统服务是否变为Healthy，变为Healthy后再切换到第二台执行。
2. 如果集群为内网使用或者没有设置外网网关，可以在集群内部开通一台带有外网IP的虚拟机，通过scp的方式复制到每一个节点，然后切换节点执行更新操作。
3. 更新1个节点大约2分钟左右，请您耐心等待。
4. 如升级过程中，升级脚本执行报错，请立即与我们联系，我们将协助您完成升级操作。

批量安装方法

批量操作方法需要节点使用密码相同，如密码不同建议使用手动部署。

master节点需要一台健康后更新第二台，故不能使用此批量更新文档。

1. 安装pssh工具

```
yum install pssh -y
```

2. 创建node节点列表文件

```
vim hosts.txt
```

格式为user@ip,例如：

```
root@10.10.10.10
root@10.10.10.11
root@10.10.10.12
root@10.10.10.13
```

3. 执行

```
pscp.pssh -Ah hosts.txt k8s.tgz /root/
pssh -Ah hosts.txt -x '-o StrictHostKeyChecking=no' -i 'sudo tar zxvf k8s.tgz'
pssh -Ah hosts.txt -x '-o StrictHostKeyChecking=no' -i 'sudo chmod +x 1.13.10/uk8supgrade.sh'
pssh -Ah hosts.txt -t 0 -x '-o StrictHostKeyChecking=no' -i 'sudo sh 1.13.10/uk8supgrade.sh node'
```

如您开通的是1.14.5请将相关的路径更改为1.14.6的路径。

CVE-2021-30465

漏洞描述

Runc 是一个CLI工具，用于根据OCI规范生成和运行容器，该工具被广泛的应用于各种容器环境中，如Kubernetes。此次漏洞触发条件为攻击者将目标挂载路径设置为一个容器Volume在主机上目录的软链接，以此来获取主机上的挂载点。由于挂载的源路径是攻击者可以控制的目录，攻击者可以将源路径中的子目录软链接到主机根目录上，并通过条件竞争TOCTTOU（Time Of Check To Time Of Use）的特定手段在一定条件让恶意容器中的指定目录挂载到主机目录。

影响版本

Docker 20.10.7 以下版本

Containerd 1.4.6, 1.5.2以下版本

Runc 1.0-rc95 以下版本

目前UK8S节点使用的默认Docker及Containerd版本均在影响范围内

漏洞自查

您可以采用以下任意一种方式确认节点使用的运行时版本。

1、直接执行`kubectl get nodes -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.status.nodeInfo.containerRuntimeVersion}{"\n"}{end}'`

2、登录节点，对于Docker节点，执行`docker version`。对于Containerd节点，执行`crictl version`

⚠ 2020年7月9日之前创建的集群，Master节点默认不加入节点，可以跳过Master节点检查。

修复建议

1、确认业务部署方式，使用受信的容器，并且在容器挂载目录中，不存在主机内其它路径的软链接。

2、自2022年1月14日起，1.17及以上版本的集群，新增节点及新建集群均会默认采用修复版本的Docker及Containerd。

3、由于兼容性问题，1.17以下集群版本已经不再维护，建议您尽快升级到1.17及以上的版本，或者通过检查业务部署，避免该漏洞。

4、1.17及以上版本集群的旧节点，您可以通过添加新节点，驱逐旧节点Pod，删除旧节点的方式进行升级。**请您务必确认Pod在新节点运行正常后，再删除旧节点。**

5、由于Master节点无法通过增删节点的方式进行升级，请按照手动升级方案进行升级。

手动升级方案

1. 请按照漏洞自查步骤，检查节点是否需要升级。
2. 执行 `kubectl drain --ignore-daemonsets <node>` 驱逐节点上所有pod，手动驱逐Pod。某些情况下，有可能需要增加`--delete-emptydir-data`参数。等待Pod驱逐完成后，执行后续升级操作。
3. 按照本文升级Docker及Containerd小节，根据您的节点类型及节点运行时是Docker还是Containerd。选择对应的升级方式进行操作。
4. 对于Docker节点 执行`docker version`。对于Containerd节点，执行`crictl version`，确认当前节点运行时版本号。
5. 【可选】对于工作节点，而非Master节点，等待升级完成后，可以执行`kubectl uncordon <node>`，将节点重新加入集群。

升级Docker及Containerd

⚠ 如果您错误地在Containerd节点上安装Docker，将导致节点不可用。

Centos7

Docker节点

```
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-rootless-extras-20.10.11-3.el7.x86_64.rpm
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-20.10.11-3.el7.x86_64.rpm
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/containerd.io-1.4.12-3.1.el7.x86_64.rpm
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-cli-20.10.11-3.el7.x86_64.rpm
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-scan-plugin-0.9.0-3.el7.x86_64.rpm
yum install -y fuse-overlayfs slirp4netns
rpm -U containerd.io-1.4.12-3.1.el7.x86_64.rpm docker-ce-20.10.11-3.el7.x86_64.rpm docker-ce-rootless-extras-20.10.11-3.el7.x86_64.rpm docker-scan-plugin-0.9.0-3.el7.x86_64.rpm docker-ce-cli-20.10.11-3.el7.x86_64.rpm
```

Containerd节点

```
wget https://download.docker.com/linux/centos/7/x86_64/stable/Packages/containerd.io-1.4.12-3.1.el7.x86_64.rpm
rpm -U containerd.io-1.4.12-3.1.el7.x86_64.rpm
```

Ubuntu 20.04

Docker节点

```
wget https://download.docker.com/linux/ubuntu/dists/focal/pool/stable/amd64/docker-ce_20.10.11~3-0~ubuntu-focal_amd64.deb
```

```
wget https://download.docker.com/linux/ubuntu/dists/focal/pool/stable/amd64/containerd.io_1.4.12-1_amd64.deb
wget https://download.docker.com/linux/ubuntu/dists/focal/pool/stable/amd64/docker-ce-cli_20.10.11~3-0~ubuntu-focal_amd64.deb
dpkg -i docker-ce_20.10.11~3-0~ubuntu-focal_amd64.deb containerd.io_1.4.12-1_amd64.deb docker-ce-cli_20.10.11~3-0~ubuntu-focal_amd64.deb
```

Containerd节点

```
wget https://download.docker.com/linux/ubuntu/dists/focal/pool/stable/amd64/containerd.io_1.4.12-1_amd64.deb
dpkg -i containerd.io_1.4.12-1_amd64.deb
```

Ubuntu 18.04

Docker节点

```
wget https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/docker-ce_20.10.11~3-0~ubuntu-bionic_amd64.deb
wget https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/containerd.io_1.4.12-1_amd64.deb
wget https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/docker-ce-cli_20.10.11~3-0~ubuntu-bionic_amd64.deb
dpkg -i docker-ce_20.10.11~3-0~ubuntu-bionic_amd64.deb containerd.io_1.4.12-1_amd64.deb docker-ce-cli_20.10.11~3-0~ubuntu-bionic_amd64.deb
```

Containerd节点

```
wget https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/containerd.io_1.4.12-1_amd64.deb
dpkg -i containerd.io_1.4.12-1_amd64.deb
```

漏洞原型链接

1、https://github.com/opencontainers/runc/security/advisories/GHSA-c3xm-pvg7-gh7r