容器服务

实践教程

腾讯云

文档目录

- 实践教程
  - 集群
    - 组建集群选型推荐
    - 实现独立集群的 Master 容灾
    - 使用 Private DNS 实现内网访问集群时的私有域名解析
  - 集群迁移
    - 边缘集群迁移至 TKE 标准集群注册节点公网版
  - Serverless 集群
    - 通过 NAT 网关访问外网
    - 通过弹性公网 IP 访问外网
    - 在 Serverless 集群上玩转深度学习
      - 构建深度学习容器镜像
      - 在 TKE Serverless 上运行深度学习
      - 常见问题
        - 公网访问相关
        - 日志采集相关
    - Serverless 集群自定义 DNS 服务
  - 调度
    - 安装 CoScheduling 实现批调度
    - 原生节点提升集群装箱率
  - 安全
    - Pod 安全组
    - 容器镜像签名及验证
    - Pod 使用 CAM 对数据库身份验证
    - 使用 ExternalSecretOperator 导入腾讯云 SSM 凭据
  - 服务部署
    - 合理利用节点资源
      - 概述
      - 设置 Request 与 Limit
      - 资源合理分配
      - 弹性伸缩
    - 应用高可用部署
    - 工作负载平滑升级
    - docker run 参数适配
    - 解决容器内时区不一致问题
    - 容器 coredump 持久化
    - 在 TKE 中使用动态准入控制器
  - 网络
    - DNS 相关
      - TKE DNS 最佳实践
      - CoreDNS 日志仪表盘使用指南
      - 在 TKE 集群中使用 NodeLocal DNS Cache
      - 在 TKE 中实现自定义域名解析
      - 在 TKE 中配置 ExternalDNS
      - 从 kube-dns 切换到 CoreDNS
      - CoreDNS ServiceAccount Token 过期问题解决方案
      - TKE 集群 CoreDNS 使用废弃 API 问题的解决方案
    - 自建 Nginx Ingress 实践教程
      - 快速开始
      - 自定义负载均衡器

启用 CLB 直连
高并发场景优化
高可用配置优化
可观测性集成
接入腾讯云 WAF
安装多个 Nginx Ingress Controller
从 TKE Nginx Ingress 插件迁移到自建 Nginx Ingress
values.yaml 完整配置示例
ingress-nginx 应用部署指南
在 TKE 使用 EnvoyGateway 流量网关
核心应用流量完全无损升级
使用 Network Policy 进行网络访问控制
Nginx 升级最佳实践
Nginx Ingress 高并发实践
Nginx Ingress 最佳实践
在 TKE 上对 Pod 进行带宽限速
TKE 基于弹性网卡直连 Pod 的网络负载均衡
在 TKE 上使用负载均衡直连 Pod
在 TKE 中获取客户端真实源 IP
在 TKE 上使用 Traefik Ingress
发布
使用 CLB 实现简单的蓝绿发布和灰度发布
日志
TKE 日志采集最佳实践
NginxIngress 自定义日志
TKE 使用 logrotate 切割 nginx-ingress 访问日志
使用 CLS 告警异常资源
监控
JVM 接入
MySQL Exporter 接入
自建 Prometheus 监控 TKE 集群
腾讯云 Prometheus 一键关联监控容器服务
一键接入腾讯云应用性能监控 APM
运维
TKE 集群中节点移出再移入操作指引
使用 Ansible 批量操作 TKE 节点
使用集群审计排查问题
为 TKE Ingress 证书续期
使用 cert-manager 签发免费证书
使用 cert-manager 为 DNSPod 的域名签发免费证书
使用 TKE NPDPlus 插件增强节点的故障自愈能力
使用 kubecm 管理多集群 kubeconfig
使用 TKE 审计和事件服务快速排查问题
在 TKE 中自定义 RBAC 授权
清理已注销的腾讯云账号资源
Terraform
使用 Terraform 管理 TKE 集群和节点池
DevOps
在 containerd 集群中使用 Docker 做镜像构建服务
在 TKE 上部署 Jenkins
弹性伸缩
事件驱动弹性伸缩最佳实践（KEDA）
认识 KEDA

在 TKE 上部署 KEDA
定时水平伸缩 (Cron 触发器)
多级服务同步水平伸缩（Workload 触发器）
基于 Prometheus 自定义指标的弹性伸缩
基于 CLB 监控指标的水平伸缩
基于 Apache Pulsar 消息队列的弹性伸缩
基于云原生 API 网关监控指标的水平伸缩
集群弹性伸缩实践
使用 tke-autoscaling-placeholder 实现秒级弹性伸缩
在 TKE 上安装 metrics-server
在 TKE 上使用自定义指标进行弹性伸缩
在 TKE 上利用 HPA 实现业务的弹性伸缩
根据不同业务场景调节 HPA 扩缩容灵敏度
容器化
境外镜像拉取加速
镜像分层最佳实践
成本管理
资源利用率提升工具大全
混合云
IDC 集群添加超级节点用于弹性扩容
AI
在 TKE 上部署 AI 大模型
部署大模型常见问题
在 TKE 上部署满血版 DeepSeek-R1 (SGLang)
使用 TKE 完整部署生产级 Stable Diffusion
使用 TKE 快速部署 ChatGLM
使用 TKE + 超级节点快速体验 Stable Diffusion
TKE Serverless 运行 ChatGLM-6B 微调
在 TKE 上运行基于 NCCL 的 RDMA 分布式训练任务
在 TKE 上使用 AIBrix 进行多节点分布式推理
基于 TKE 部署 Dify 最佳实践
游戏
使用 CLB 为 Pod 分配公网地址映射

实践教程
集群
组建集群选型推荐

最近更新时间：2025-04-30 16:05:12

当您使用腾讯云容器服务 TKE 组建 Kubernetes 集群时，会面对多种配置选项，难以进行选择。本文介绍以下功能选型，进行对比并给出选型建议。您可参考本文，选择更适用于您业务的配置选型。

- Kubernetes 版本
- 容器网络插件：GlobalRouter 及 VPC-CNI
- 运行时组件：Containerd 及 Docker
- Service 转发模式：iptables 及 ipvs
- 集群类型：托管集群及独立集群
- 节点操作系统
- 使用节点池
- 使用启动脚本

Kubernetes 版本

Kubernetes 版本迭代较快，新版本通常包含许多 bug 修复和新功能，而旧版本会逐渐淘汰。建议您在创建集群时，选择当前 TKE 支持的最新版本。后续可通过升级已有 Master 和节点版本，更换迭代产生的新版本。

容器网络插件：GlobalRouter 及 VPC-CNI

网络模式架构

TKE 支持网络模式架构，如需了解更多信息，请参见 如何选择容器服务网络模式 。

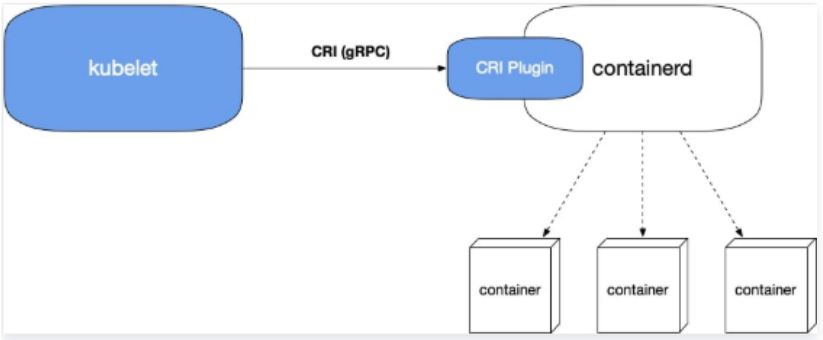
运行时组件：Containerd 及 Docker

运行时架构

TKE 支持以下两种运行时架构：Containerd 和 Docker。由于 Kubernetes 1.24 版本已停止对 Docker 的支持，推荐优先选择 Containerd。更多信息请参见 如何选择 Containerd 和 Docker 。

Containerd 作为运行时架构

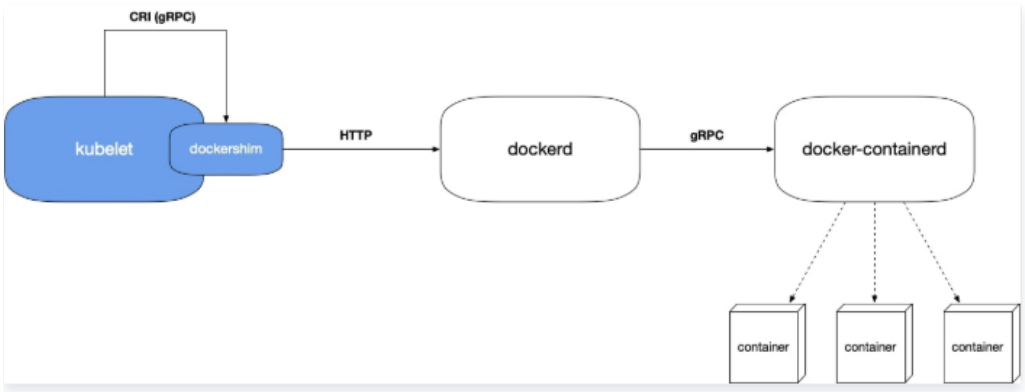
架构图如下：



- 自 Containerd 1.1 之后，支持了 CRI Plugin，即 containerd 自身即可适配 CRI 接口。
- 相比 Docker 方案，调用链少了 dockershim 和 dockerd。

Docker 作为运行时架构

架构图如下：



调用链如下：

1. Kubelet 内置的 dockershim 模块帮助 docker 适配了 CRI 接口。
2. Kubelet 通过 socket 文件自行调用 dockershim。
3. Dockershim 调用 dockerd 接口（Docker HTTP API）。
4. Dockerd 调用 docker-containerd（gRPC）来实现容器的创建与销毁等。

调用链过长原因分析：

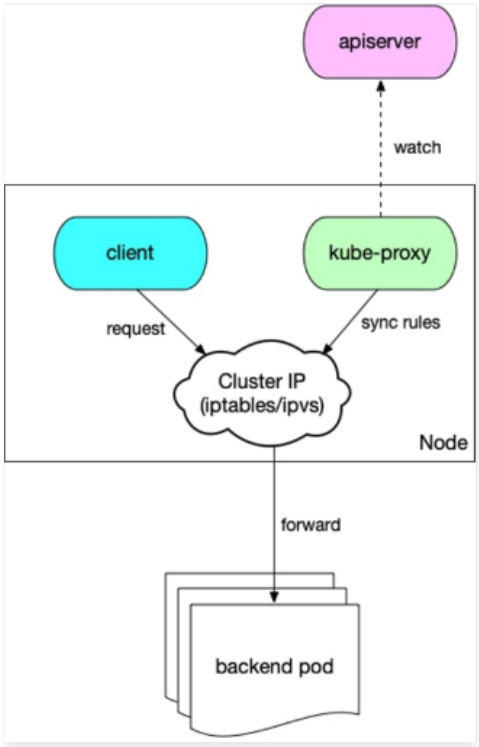
Kubernetes 起初仅支持 Docker，后来引入了 CRI，并将运行时抽象化以支持多种运行时。Docker 与 Kubernetes 存在竞争关系，未在 dockerd 中实现 CRI 接口，故 Kubernetes 需自行在 dockerd 中实现 CRI。Docker 本身内部组件模块化及 CRI 适配。

选型建议

- Containerd 调用链更短，组件更少，更稳定，占用节点资源更少。建议选择 Containerd。
- 当您遇到以下情况时，请选择 Docker 作为运行时组件：
  - 如需使用 Docker in Docker。
  - 如需在 TKE 节点使用 `docker build/push/save/load` 等命令。
  - 如需调用 Docker API。
  - 如需使用 Docker Compose 或 Docker Swarm。

Service 转发模式：iptables 及 ipvs

Service 转发原理图如下所示：



1. 节点上的 kube-proxy 组件 watch apiserver，获取 Service 与 Endpoint，根据转发模式将其转化成 iptables 或 ipvs 规则并写到节点上。
2. 集群内的 client 访问 Service（Cluster IP），会被 iptable 或 ipvs 规则负载均衡到 Service 对应的后端 pod。

转发模式对比

- ipvs 模式性能更高，但存在一些已知未解决的 bug。
- iptables 模式更成熟稳定。

选型建议

对稳定性要求极高且 Service 数量小于2000时，建议选择 iptables，其余场景建议首选 ipvs。

集群类型：托管集群及独立集群

TKE 支持以下两种集群类型：

- **托管集群**：
  - 腾讯云专业团队维护集群 master&etcd，您可专注业务部署。
  - 会率先支持大部分新功能的托管。
  - Master 的计算资源会根据集群规模自动扩容。
  - 用户需要支付集群管理费，定价详情请参见 集群购买指南 。
  - 提供 SLA。
- **独立集群（已停止新建）**：
  - 用户可完全掌控 Master 组件。
  - 用户需要支付 Master 机器相关的资源费用。
  - 不提供 SLA。

选型建议

建议通常情况下选择托管集群，如需完全掌握 Master，例如对 Master 进行个性化定制实现高级功能，则可选择使用独立集群。

节点操作系统

TKE 支持三种发行版操作系统：Tencent Linux、Ubuntu 和 CentOS。其中，Tencent Linux 使用腾讯云团队维护的定制内核 TencentOS-kernel，而 Ubuntu 和 CentOS 则使用 Linux 社区的官方开源内核。推荐您选择 TencentOS 内核系列的公共镜像。

选型建议

建议选择 Tencent Linux 版本的操作系统，该版本操作系统是包含 TencentOS-kernel 内核的腾讯云公共镜像，容器服务 TKE 目前已经支持该镜像并作为缺省选项。

使用节点池

节点池主要用于批量管理节点：

- 节点 Label 与 Taint。
- 节点组件启动参数。
- 节点自定义启动脚本。详情请参见 节点池概述 。

适用场景

- 异构节点分组管理，减少管理成本。
- 使集群更好的支持复杂的调度规则（Label 及 Taint）。
- 频繁扩缩容节点，减少操作成本。
- 节点日常维护，例如版本升级等。

用法举例

部分 IO 密集型业务需要高 IO 机型，为该业务创建节点池、配置机型并统一设置节点 Label 与 Taint，并配置 IO 密集型业务亲和性。选中 Label，使其调度到高 IO 机型的节点（Taint 可以避免其他业务 Pod 调度上来）。
当业务量快速上升时，该 IO 密集型业务也需要更多的计算资源。在业务高峰时段，HPA 功能自动为该业务扩容了 Pod，而节点计算资源不够用，此时节点池的自动伸缩功能自动扩容了节点，守住了流量高峰。

使用启动脚本

组件自定义参数

- 使用自定义 Kubernetes 组件启动参数功能，请通过 在线咨询 进行申请。
- 在集群中设置自定义 Kubernetes 组件参数，详情请参见 自定义 Kubernetes 组件启动参数 。

节点启动配置

您可以在创建集群或新增节点时，或添加已有节点时，设置节点的启动脚本。详情请参见 设置节点的启动脚本 。

实现独立集群的 Master 容灾

最近更新时间：2024-09-03 17:12:51

概述

容器服务 TKE 包含托管集群及独立部署集群。若使用托管集群，则无需关注容灾，托管集群的 Master 由容器服务 TKE 内部维护。若使用独立集群，则 Master 节点由用户自行管理维护。独立集群如需实现容灾，则首先应根据需求规划容灾方案，在创建集群时进行相应配置即可。本文介绍如何实现 TKE 独立集群 Master 的容灾，您可参考本文进行操作。

容灾实现思路

实现容灾应从物理部署层面切入，为避免因一次物理层面的故障导致多台 Master 异常，需将 Master 节点打散部署。可借助 置放群组 来选择将 Master 从物理机、交换机或机架三种维度中其中一种来将 Master 打散，以避免底层硬件或软件故障导致多台 Master 异常。如对容灾要求非常高，还可以考虑将 Master 跨可用区部署，以避免在发生大规模故障时，整个数据中心不可用导致 Master 集体异常的情况。

使用置放群组打散 Master

1. 登录 置放群组控制台，创建置放群组，详情请参见 分散置放群组。如下图所示：

> **注意：**
> 置放群组需与 TKE 独立集群在同一地域。

置放群组层级如下，本文以选择“机架层级”为例：

置放群组层级	说明
物理机层级	独立集群 Master 使用云服务器部署，属于虚拟机，在物理机上运行。一台物理机可能运行有多台虚拟机，如果物理机发生故障，将影响在这台物理机上运行的所有虚拟机。使用这个层级可以将 Master 打散部署到不同物理机上，避免一台物理机故障导致多台 Master 异常。
交换机层级	多台不同物理机可能连接在相同的交换机上，如果交换机发生故障，可能影响多台物理机。使用这个层级可以将 Master 打散部署到连接不同交换机的物理机上，避免交换机故障导致多台 Master 异常。
机架层级	多台不同物理机可能放置在同一个机架上，如果发生机架级别的意外，导致一台机架上多台物理机故障。使用这个层级以将 Master 打散部署到不同机架上的物理机上，避免发生机架级别的意外导致多台 Master 异常。

2. 参考 创建集群 创建 TKE 独立集群。在 “Master&Etcd 节点配置”的“高级设置”中，勾选“将实例添加到分散置放群组” ，并选择已创建的置放群组。如下图所示：

配置完成后，对应 Master 节点就会被打散部署到不同的机架上，实现机架级别的容灾。

Master 跨可用区容灾

如果对容灾要求较高，避免因发生大规模故障时整个数据中心都不可用，导致所有 Master 异常，可选择将 Master 部署在不同可用区中。配置方法如下：在创建集群，选择 “Master&Etcd 节点配置”时，在多个可用区添加机型即可。如下图所示：


Master&Etcd 配置
编辑 删除
可用区 北京二区
节点网络 zone2-1
配置 S3.LARGE8(标准型S3,4核8GB)
系统盘 SSD云硬盘 50GB
数据盘 暂不购买
公网带宽 按使用流量计费 1Mbps
主机名 自动生成
数量 1 台
编辑 删除
可用区 北京三区
节点网络 zone3-1
配置 SA2.LARGE8(标准型SA2,4核8GB)
系统盘 SSD云硬盘 50GB
数据盘 暂不购买
公网带宽 按使用流量计费 1Mbps
主机名 自动生成
数量 1 台
编辑 删除
可用区 北京四区
节点网络 zone4-1
配置 S5.LARGE8(标准型S5,4核8GB)
系统盘 高性能云硬盘 50GB
数据盘 暂不购买
公网带宽 按使用流量计费 1Mbps
主机名 自动生成
数量 1 台

使用 Private DNS 实现内网访问集群时的私有域名解析

最近更新时间：2024-07-22 17:43:41

操作场景

当前集群开启内网访问后，容器服务 TKE 默认通过域名访问集群，您需要在访问机上配置 Host 来进行内网域名解析。如未配置对应的域名解析规则（Host），在访问机上访问对应集群（运行 kubectl get nodes）时将会报错 “no such host”，如下图所示：

```
[root@VM-      -centos ~]# kubectl get nodes
Unable to connect to the server: dial tcp: lookup cls-        .ccs.tencent-cloud.com on        : no such host
```

在实际过程中，配置 Host 行为会增加管理访问机上 Host 的人力成本。因此，我们建议您使用腾讯云全新上线的 私有域解析 Private DNS 服务，使用该服务简单便捷，只需要完成以下三步操作即可。

收费说明

Private DNS 采用按量付费的计费方式。收费项为：私有域名数量 + 解析请求量，以自然日为单位进行结算。了解更多请参见 Private DNS 购买指南 。

支持地域

Private DNS 目前支持的地域未完全覆盖 TKE 支持地域，具体支持地域列表请参见“私有域解析 Private DNS 限制” 开放地域 。
在 Private DNS 不支持的地域使用内网访问集群功能，您仍需手动配置 Host。如需在未支持地域上使用 Private DNS 服务，请 联系我们 。

前提条件

已创建容器集群，并已开启内网访问。详情可参见 创建集群 。

操作步骤

开通 Private DNS

请参见官方文档开通 Private DNS 。

创建私有域

1. 登录 Private DNS 控制台。
2. 单击**新建私有域**，配置以下选项（其他参数使用默认值即可），了解更多请参见 创建私有域文档 。

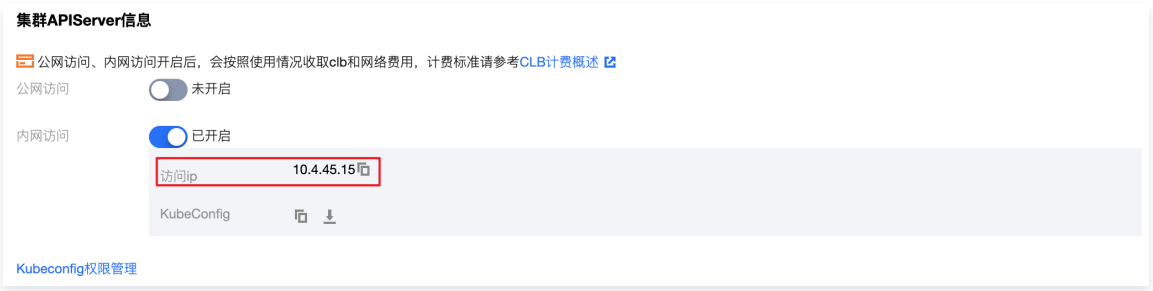
   - **域名**：输入“tencent-cloud.com”（TKE 为集群访问分配的域名）。
   - **关联 VPC**：选择需要访问集群的节点网络 VPC。
3. 单击**确定**即可创建私有域。

配置解析记录

1. 单击上述创建的私有域名称，进入**解析记录**页面。

2. 单击**添加记录**，配置以下选项：

- **主机记录**：输入 TKE 集群访问的次级域名，例如 “cls-{{clsid}}.css”。
- **记录类型**：输入 A。
- **记录值**：输入 TKE 集群内网访问 IP。内网访问 IP 可前往 集群管理 > **集群** > **基本信息**获取，如下图所示：



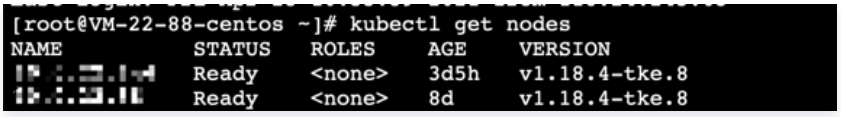
3. 单击右侧操作栏下的**保存**以保存配置。

验证效果

1. 执行以下命令再次访问集群。

```
kubectl get nodes
```

2. 当命令执行结果显示如下图时，说明已成功访问集群并拉取 Node 列表。



集群迁移
边缘集群迁移至 TKE 标准集群注册节点公网版

最近更新时间：2024-08-28 18:00:11

下线公告

腾讯云边缘容器服务 TKE-Edge 于2024年8月28日下线，详情请参见 TKE-Edge 边缘容器服务下线公告 。您可以参考本文将边缘集群迁移至 TKE 标准集群注册节点公网版。

前提条件

- 已有 TKE-Edge 边缘集群（集群 A）：待迁移。如果业务方存在部署测试业务的边缘集群，建议先迁移此测试边缘集群。
- 已创建迁移目标的 TKE 标准集群（集群 B）：集群版本需 ≥1.20，且已开启注册节点公网版。创建 TKE 标准集群并开启注册节点公网版，请参见 创建注册节点（公网版）。
- 集群 A 和 集群 B 建议在同一 VPC 下。

迁移步骤

整体迁移分为5个步骤。如果您在创建过程中遇到任何问题或需要帮助，您可以咨询 在线客服 或 提交工单 来与我们联系。

步骤1：功能验证阶段

准备好迁移目标的集群 B，并确保全流程功能验证正常。集群 B 的创建流程请参见 创建注册节点（公网版）。

步骤2：节点灰度阶段

选取1~2个集群 A 中的节点作为灰度节点，依次进行驱逐和移出，并重新注册到集群 B 中。操作方法如下：

1. 登录 容器服务控制台，选择左侧导航中的集群。
2. 在集群列表中，单击集群 ID，进入集群管理页面。
3. 选择**节点管理 > Worker 节点**，在节点页签中单击节点行右侧的**驱逐**。节点驱逐后，自动将节点内的所有 Pod（不包含 DaemonSet 管理的 Pod）驱逐到集群内其他节点上，并将驱逐的节点设置为封锁状态。

   > **注意：**
   > 带有本地存储的 Pod 被驱逐后数据将丢失，请确认后谨慎操作。

   驱逐操作如下图所示：

4. 在弹出的对话框中，单击**确定**，即可进行驱逐动作，驱逐完成后，请确保节点上没有业务 Pod（不包含 DaemonSet 管理的 Pod）在运行。
5. 驱逐完成后，可再次单击节点行右侧的**移出**，将节点从集群中移出。节点移出后将不再作为集群 A 中的资源参与调度。

   > **注意：**
   > 请确保移出灰度节点后，集群 A 中还有足够的节点支撑业务正常运行。否则建议使用新节点作为灰度节点。

6. 将灰度节点注册到集群 B 中，注册流程请参见 创建注册节点（公网版）。

步骤3：业务灰度阶段

操作方法如下：

1. 选取集群 A 中业务敏感度较低的工作负载作为灰度负载，在集群 B 中进行部署（集群 A 中的工作负载依然保留），确保集群 B 中工作负载部署正常，业务 Pod 正常运行。
2. 将工作负载上游的流量灰度少量到集群 B 的工作负载，观察业务的运行情况。

步骤4：扩大灰度阶段

在步骤2和步骤3的灰度验证无问题后，逐渐扩大步骤2和步骤3的灰度范围，将更多节点和业务迁移至集群 B 中。

步骤5：观察下线阶段

等待所有节点和业务都迁移至集群 B 中后，观察一段时间无问题后可删除集群 A。

Serverless 集群
通过 NAT 网关访问外网

最近更新时间：2024-09-29 16:49:41

操作场景

TKE Serverless 容器服务支持通过配置 NAT 网关 和 路由表 来实现集群内服务访问外网，您可参考本文进行配置。

操作步骤

创建 NAT 网关

1. 登录 NAT 网关控制台 。
2. 选择地区和私有网络，单击**新建**。
3. 参考 创建与管理 NAT 网关，创建与 TKE Serverless 集群同地域、同私有网络 VPC 的 NAT 网关。

创建指向 NAT 网关的路由表

1. 单击左侧导航栏中的 路由表 。
2. 在**路由表**页面，单击**新建**。
3. 在**新建路由表**页面，参考以下信息创建与 TKE Serverless 集群同地域、同 VPC 的路由表。如下图所示：

主要参数信息如下：

- **目的端**：选择需访问的外网 IP 地址，支持配置 CIDR。例如，填写 `0.0.0.0/0` 会转发所有流量到 NAT 网关。
- **下一跳类型**：选择“NAT 网关”。
- **下一跳**：选择在 创建 NAT 网关 步骤中已创建的 NAT 网关。

4. 单击**创建**即可。

关联子网至路由表

完成配置路由后，需选择子网关联到该路由表，被选择子网内的访问 Internet 的流量将指向 NAT 网关。步骤如下：

1. 在“路由表”页面中，选择 创建指向 NAT 网关的路由表 步骤中已创建路由表所在行右侧的**关联子网**。
2. 在弹出的“关联子网”窗口中，勾选需关联子网并单击**确定**即可。

> **说明：**
> 此子网为容器网络，并非 Service CIDR。

完成路由表关联子网后，同 VPC 的资源即可以通过 NAT 网关的外网 IP 访问 Internet。

验证配置

1. 在集群列表页面，单击 Serverless 集群 ID 进入该集群的管理页面。

2. 选择需登录容器所在行右侧的**远程登录**，并执行 ping 命令验证该 Pod 是否可访问外网。返回结果如下所示，则表明已成功访问外网。

```
bash-4.2$ ping qq.com
PING qq.com (203.205.254.157) 56(84) bytes of data.
64 bytes from 203.205.254.157 (203.205.254.157): icmp_seq=1 ttl=45 time=318 ms
64 bytes from 203.205.254.157 (203.205.254.157): icmp_seq=4 ttl=45 time=314 ms
64 bytes from 203.205.254.157 (203.205.254.157): icmp_seq=5 ttl=45 time=311 ms
64 bytes from 203.205.254.157 (203.205.254.157): icmp_seq=6 ttl=45 time=315 ms
```

注意事项

NAT 网关不再自动调整所绑定的 EIP 带宽，若出现镜像拉取超时等问题且 NAT 网关带宽未达上限时，可查询 EIP 带宽是否已达瓶颈并根据实际需求设置 EIP 带宽上限。

通过弹性公网 IP 访问外网

最近更新时间：2023-09-08 19:13:05

目前 TKE Serverless 已经支持在 Pod 中绑定 EIP，只需在 template annotation 中说明即可。详情请参见 Annotation 说明 文档。
与 EIP 相关的 Annotation 标识可参考下列表格：

Annotation Key	Annotation Value 及描述	是否必填
eks.tke.cloud.tencent.com/eip-attributes	表明该 Workload 的 Pod 需要关联 EIP，值为 `""` 时表明采用 EIP 默认配置创建。`""` 内可填写 EIP 云 API 参数 json，实现自定义配置。	如需绑定 EIP，则此项为必填项
eks.tke.cloud.tencent.com/eip-claim-delete-policy	Pod 删除后，EIP 是否自动回收，Never 不回收，默认回收。	否
eks.tke.cloud.tencent.com/eip-id-list	表明使用存量 EIP，仅支持 statefulset。默认销毁 Pod 不会回收 EIP。注意，statefulset pod 的数量最多只能为此 Annotation 中指定 eipId 的数量。	否

1. 如需为 Workload 或 Pod 绑定 EIP 访问公网，最简单的方式就是在对应 Workload 或 Pod 的 `annotation` 下，添加标识 `eks.tke.cloud.tencent.com/eip-attributes: ""` 。示例如下：

```
metadata:
  name: tf-cnn
  annotations:
    eks.tke.cloud.tencent.com/eip-attributes: ""  #需求EIP，配置均为默认
```

2. 运行后执行以下命令查看事件：

```
kubectl describe pod [name]
```

可以发现多了两行跟 EIP 有关的事件，如下图所示，说明成功运行。

```
Events:
  Type    Reason          Age   From               Message
  ----    ------          ----  ----               -------
  Normal  Scheduled       106s  default-scheduler  Successfully assigned default/tf-cnn to eklet-subnet-6rjbxwwb
  Normal  AllocatedEip    95s   eklet              Successfully allocate eip eip-[redacted], ip 43.[redacted]
  Normal  Starting        81s   eklet              Starting pod sandbox eks-j0i3y99u
  Normal  Starting        66s   eklet              Sync endpoints
  Normal  Pulling         64s   eklet              Pulling image "hkccr.ccs.tencentyun.com/carltk/[redacted]:latest"
  Normal  Pulled          64s   eklet              Successfully pulled image "hkccr.ccs.tencentyun.com/carltk/t[redacted]:latest" in 290.700737ms
  Normal  Created         64s   eklet              Created container tf-cnn
  Normal  Started         63s   eklet              Started container tf-cnn
  Normal  AssociatedEip   50s   eklet              Successfully associate eip eip-[redacted]
```

3. 查看 log 文件也发现能正常下载数据集。如下图所示：

```
I0803 07:56:37.621758 140120123275072 dataset_builder.py:400] Generating dataset mnist (/root/tensorflow_datasets/mnist/3.0.1)
I0803 07:56:38.315572 140120123275072 dataset_builder.py:433] Dataset mnist is hosted on GCS. It will automatically be downloaded to your
local data directory. If you'd instead prefer to read directly from our public
GCS bucket (recommended if you're running on GCP), you can instead pass
`try_gcs=True` to `tfds.load` or set `data_dir=gs://tfds-data/datasets`.

Downloading and preparing dataset 11.06 MiB (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to /root/tensorflow_datasets/mnist/3.0.1...
Dl Completed...: 100%|[redacted]| 4/4 [00:02<00:00,  1.92 file/s]
I0803 07:56:40.842971 140120123275072 dataset_info.py:358] Load dataset info from /root/tensorflow_datasets/mnist/3.0.1.incomplete9JL6PD
```

> ⚠ 注意
> **EIP 的申请每天有限额**，不适用于需要多次运行的任务。具体限额请参见 为什么无法申请 EIP 常见问题说明。

在 Serverless 集群上玩转深度学习
构建深度学习容器镜像

最近更新时间：2024-07-22 14:45:41

操作场景

本系列文章将记录在 TKE Serverless 集群部署深度学习的一系列实践，从直接部署 TensorFlow 到后续实现 Kubeflow 的部署，旨在提供一个较完整的容器深度学习实践方案。本文着重介绍自建深度学习容器镜像的搭建，为后面深度学习部署任务提供更方便快捷的完成方式。
因为本文实践任务需要，公有镜像无法满足深度学习部署需求，因此本实践选择自建镜像。
除深度学习框架 TensorFlow-gpu ，该镜像还包含 GPU 训练需要的 cuda、cudnn ，并整合了 TensorFlow 官方提供的深度学习模型——包含了目前 CV、NLP、RS 等领域的 SOTA 模型。模型详情请参见 Model Garden for TensorFlow 。

操作步骤

1. 本文示例通过 docker 容器构建镜像。准备 Dockerfile 文件，示例如下：

```
FROM nvidia/cuda:11.3.1-cudnn8-runtime-ubuntu20.04
RUN apt-get update -y \
 && apt-get install -y python3 \
    python3-pip \
    git \
 && git clone git://github.com/tensorflow/models.git \
 #不需要的组件及时卸载（可选）
 && apt-get --purge remove -y git \
 #删除apt安装用的安装包（可选）
 && rm -rf /var/lib/apt/lists/* \
 #新建存储模型和数据的路径，可作为挂载点（可选）
 && mkdir /tf /tf/models /tf/data
ENV PYTHONPATH $PYTHONPATH:/models
ENV LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/local/cuda-11.3/lib64:/usr/lib/x86_64-linux-gnu
RUN pip3 install --user -r models/official/requirements.txt \
 && pip3 install tensorflow
```

2. 执行以下命令进行部署。

```
docker build -t [name]:[tag] .
```

> **说明**
> 必要的部件，例如 Python、TensorFlow、cuda、cudnn 以及模型库等安装步骤，本文不再赘述。

相关说明

镜像相关

关于基础镜像 nvidia/cuda，CUDA 容器镜像为 CUDA 支持的平台和架构提供了一个易于使用的分发版。此处选择的是 cuda 11.3.1、cudnn 8的组合。更多版本选择可参见 Supported tags 。

环境变量

在进行本文实践时，需要重点关注环境变量 `LD_LIBRARY_PATH` 。
`LD_LIBRARY_PATH` 是动态链接库的安装路径，通常为 libxxxx.so 的格式。在此处主要是为了链接 cuda 和 cudnn。例如 libcudart.so.[version]、libcusolver.so.[version]、libcudnn.so.[version] 等。您可以执行 `ll` 命令进行查看，如下图所示：

```
root@5a949761c669:/usr/local/cuda-11.3/lib64# ll
total 1534336
drwxr-xr-x 1 root root      4096 Jul  2 03:57 ./
drwxr-xr-x 1 root root      4096 Jul  2 03:57 ../
lrwxrwxrwx 1 root root        16 May  4 02:30 libOpenCL.so.1 -> libOpenCL.so.1.0
lrwxrwxrwx 1 root root        18 May  4 02:30 libOpenCL.so.1.0 -> libOpenCL.so.1.0.0
-rw-r--r-- 1 root root     30856 May  4 02:30 libOpenCL.so.1.0.0
lrwxrwxrwx 1 root root        23 May 13 23:26 libcublas.so.11 -> libcublas.so.11.5.1.109
-rw-r--r-- 1 root root 121866104 May 13 23:26 libcublas.so.11.5.1.109
lrwxrwxrwx 1 root root        25 May 13 23:26 libcublasLt.so.11 -> libcublasLt.so.11.5.1.109
-rw-r--r-- 1 root root 263770264 May 13 23:26 libcublasLt.so.11.5.1.109
lrwxrwxrwx 1 root root        21 May  4 02:30 libcudart.so.11.0 -> libcudart.so.11.3.109
-rw-r--r-- 1 root root    619192 May  4 02:30 libcudart.so.11.3.109
lrwxrwxrwx 1 root root        22 May 13 23:30 libcufft.so.10 -> libcufft.so.10.4.2.109
-rw-r--r-- 1 root root 190417864 May 13 23:30 libcufft.so.10.4.2.109
lrwxrwxrwx 1 root root        23 May 13 23:30 libcufftw.so.10 -> libcufftw.so.10.4.2.109
-rw-r--r-- 1 root root    631888 May 13 23:30 libcufftw.so.10.4.2.109
```

根据官方镜像 Dockerfile 源码 执行以下命令：

```
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64
```

其中 `/usr/local/nvidia/lib` 指向 cuda 路径的软连接，为 cuda 准备。而附带 cudnn 的版本只做到了安装 cudnn ，并没有为 cudnn 指定 `LD_LIBRARY_PATH` ，因此可能会导致报错 Warning ，从而使用不了 GPU 资源，报错如下所示：

```
Could not load dynamic library 'libcudnn.so.8'; dlerror: libcudnn.so.8: cannot open shared object file:
No such file or directory
Cannot dlopen some GPU libraries. Please make sure the missing libraries mentioned above are installed
properly if you would like to use GPU...
```

如果出现此类报错，可以尝试手动添加上 cudnn 路径。此处可以执行以下命令运行镜像，查看 libcudnn.so 所在的路径。

```
docker run -it nvidia/cuda:[tag] /bin/bash
```

由源码可知，cudnn 通过 `apt-get install` 命令安装，默认在 `/usr/lib` 下。本文示例中 libcudnn.so.8 的实际路径则是在 `/usr/lib/x86_64-linux-gnu` 下，用冒号在后面补充上。
可能会因为版本系统不同等原因，实际路径有偏差，以源码和实际观察为准。

后续操作

后续操作请参见 在 TKE Serverless 上运行深度学习 文档。

常见问题

在进行本实践过程中遇到的问题，请参见 常见问题 文档进行排查解决。

在 TKE Serverless 上运行深度学习

最近更新时间：2025-07-01 14:42:22

操作场景

本系列文章将记录在 TKE Serverless 上部署深度学习的一系列实践，从直接部署 TensorFlow 到后续实现 Kubeflow 的部署，旨在提供一个较完整的容器深度学习实践方案。

前提条件

本文将在上一篇文档 构建深度学习容器镜像 基础上继续操作，利用自建集群，在 TKE Serverless 上运行深度学习任务。自建镜像已上传到镜像仓库中：`ccr.ccs.tencentyun.com/carltk/tensorflow-model`，无需重新构建，可以直接拉取使用。

操作步骤

创建 TKE Serverless 集群

请参见 创建集群 文档创建 TKE Serverless 集群。

> **说明：**
> 由于需要运行 GPU 训练任务，在创建集群时，请注意选择的容器网络所在区的支持资源，选择支持 GPU 的可用区，如下图所示：

创建 CFS 文件系统（可选）

容器在任务结束后会自动删除，并释放资源。为了实现对模型和数据的持久化存储，建议通过挂载外部存储的方式进行数据存储。目前支持 云硬盘 CBS、文件存储 CFS、对象存储 COS 等方式。在本文示例中，我们将使用 NFS 盘的方式，利用 CFS 实现多读多写的持久化存储。

创建文件存储

1. 登录 文件存储 CFS 控制台，进入**文件系统**页面。
2. 单击**创建**，在新建文件系统页面中，选择文件系统类型，并单击**下一步:详细设置**。
3. 在**详细设置**页面进行相关配置，CFS 类型信息与配置细节可参见 创建文件系统及挂载点 文档。

> **注意：**
> - 文件协议需选择 NFS。
> - 创建的 CFS 地域，需确保与集群在同一地域。

4. 确认无误之后单击**立即购买**并完成付费即可创建文件存储。

获取文件系统挂载信息

1. 在**文件系统**页面，单击需获取子目标路径的文件系统 ID，进入该文件系统详情页。

2. 选择**挂载点信息**页签，从 “Linux下挂载” 获取该文件系统挂载信息。如下图所示：

> **说明**
> 在挂载点详细中需要记住 IPv4 地址，IPv4 将作为 NFS 路径，后续配置挂载时需要，例如 `10.0.0.161:/` 。

创建训练任务

本文任务以 MNIST 手写数字识别数据集，加两层 CNN 为例，相关示范镜像为上一章 自建镜像 ，如需自定义镜像，请参见 深度学习容器镜像构建 文档。以下提供两种创建任务的方式。

控制台操作指引

由于深度学习任务的性质，本文以部署 Job 节点为例。如何部署 Job 请参见 Job 管理 文档。以下提供控制台的部署范例：

1. 在**数据卷（选填）**配置项中，选择 NFS 盘，并输入上述步骤创建的 CFS 名称和 IPv4地址。如下图所示：

2. 在**实例内容器**中的**挂载点**配置项里，选择数据卷，并配置挂载点。如下图所示：

⚠ 注意：

- 因为数据集可能需要联网下载，所以需要配置对集群的外网访问。详情请参见常见问题 公网访问相关 。
- 选择 GPU 型号后，在填写 request 和 limit 时需要为容器分配符合 资源规格 的 CPU 和内存，实际填写并不严格要求精确到个位。在控制台中配置，也可以选择删除默认配置以留空，即为“不限制”，也会有对应的计费规格；更推荐这种做法。
- 容器运行命令 command 继承 Docker 的 CMD 字段，而 CMD 指令首选 exec 形式，不调用 shell 命令。这意味着不会发生正常的 shell 处理。因此命令需要 shell 形式运行，就需要在前面添加 `"sh","-c"` 。在控制台输入多个命令和参数时，每个命令单独一行（以换行为准）。

Kubectl 操作指引

您还可以使用 YAML 文件创建任务。

1. 准备 YAML 文件，示例 gpu_pod.yaml 如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: tf-cnn
  annotations:
    #eks.tke.cloud.tencent.com/cpu: "8"
    #eks.tke.cloud.tencent.com/gpu-count: "1"
    eks.tke.cloud.tencent.com/gpu-type: T4
    #eks.tke.cloud.tencent.com/mem: 32Gi
spec:
  containers:
  - name: tf-cnn
    image: hkccr.ccs.tencentyun.com/carltk/tensorflow-model:latest # 训练任务的镜像
    env:
    - name: MODEL_DIR
      value: /tf/model
    - name: DATA_DIR
      value: /tf/data
    command:
      - "sh"
      - "-c"
      # 触发训练任务的脚本
      - "python3 official/vision/image_classification/mnist_main.py \
        --model_dir=$MODEL_DIR \
        --data_dir=$DATA_DIR \
        --train_epochs=5 \
        --distribution_strategy=one_device \
        --num_gpus=1 \
        --download"
    resources:
      limits:
        #cpu: "8"
        #memory: 32Gi
        nvidia.com/gpu: "1"
      requests:
        #cpu: "8"
        #memory: 32Gi
        nvidia.com/gpu: "1"
    volumeMounts:
    - name: tf-model-cfs
      mountPath: /tf
  volumes:
  - name: tf-model-cfs    #训练结果持久化，保存到CFS
```

```
    nfs:
      path: /             #此处填写CFS保存的根目录
      server: 10.0.1.8    #此处填写之前创建的CFS的IPv4地址
  restartPolicy: OnFailure
```

2. 执行以下命令完成部署。

```
kubectl create -f [yaml_name]
```

> **注意：**
> 除了控制台操作指引中提到的 注意事项，还需注意：
> - 在 YAML 文件中需使用 Annotations 声明资源分配，详情请参见 Annotation 说明。同样需要注意的是不同 GPU 对应不同的 CPU、内存选项，建议按需填写。
> - 此处数据卷使用的是 NFS 。若需使用其他数据卷进行持久化存储，请参见 其他存储卷使用说明。
> - Annotation 可以**只保留** `eks.tke.cloud.tencent.com/gpu-type` ，**不需要其他项**。如果写上了 `/gpu-count` ，那么 `cpu` 和 `mem` 也需要一起写上（本文**不推荐加上其他项**。不加不会影响实际效果，加了之后未按规格填写可能会报 OOM 错误）。
> - 在 GPU 的调度中，对 `nvidia.com/gpu` 而言，**limits 是必须且仅需填写**。如果只写 Annotation ，将出现找不到卡的报错。如果只填 limits ，该值将被作为 request 。如果也写上 request ，二者值必须相等。详情请参见 K8S 文档 调度 GPUs（这里同样**不推荐在 request 和 limits 中加上 cpu 和 memory 的设置**，理由同上）。

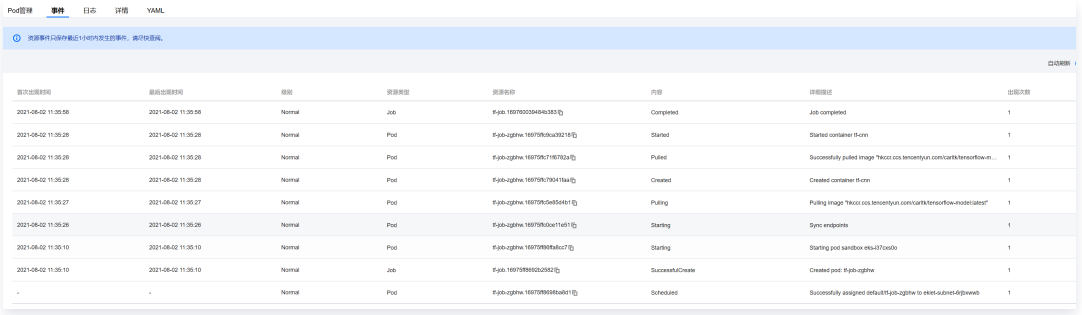
查看运行结果

以下提供控制台和命令行两种方式查看运行结果：

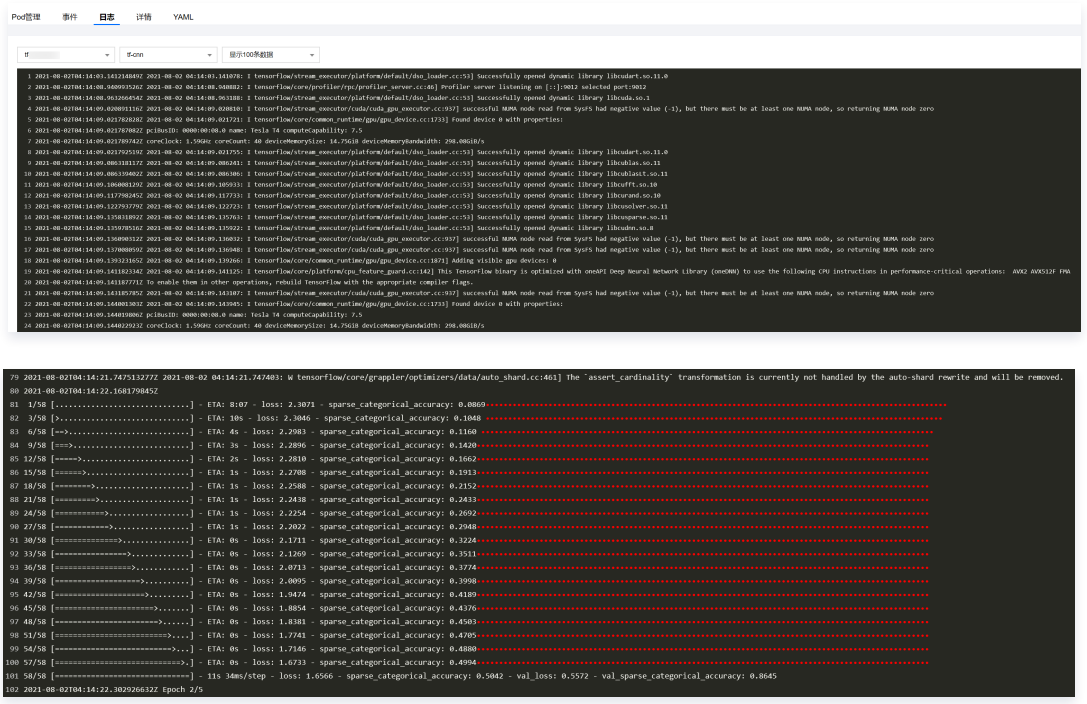
控制台查看

在创建 Job 之后，默认进入 Job 管理页面。您也可以通过以下步骤进入 Job 管理页面：

1. 登录容器服务控制台，选择左侧导航栏中的 集群。
2. 在集群列表中，单击需要查看的事件集群 ID，进入集群管理页面。
3. 选择**工作负载** > **Job**，在 Job 列表中单击上述步骤创建的 Job。
   - 选择**事件**页签查看事件，如下图所示：



- 选择**日志**页签查看日志，如下图所示：



命令行查看

您可以使用命令查看事件或日志：

- **执行以下命令查看事件：**

```
kubectl describe pod [name]
```

如下图所示：

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  98s   default-scheduler  Successfully assigned default/tf-cnn to eklet-subnet-6rjbxwwb
  Normal  Starting   98s   eklet              Starting pod sandbox eks-lv490b0e
  Normal  Starting   82s   eklet              Sync endpoints
  Normal  Pulling    80s   eklet              Pulling image "hkccr.ccs.tencentyun.com/carltk/tensorflow-model:latest"
  Normal  Pulled     80s   eklet              Successfully pulled image "hkccr.ccs.tencentyun.com/carltk/tensorflow-model:latest" in 285.270462ms
  Normal  Created    80s   eklet              Created container tf-cnn
  Normal  Started    79s   eklet              Started container tf-cnn
```

- **执行以下命令持续输出日志：**

```
kubectl logs -f [pod_name]
```

如下图所示：

```
2021-08-02 10:15:19.859055: W tensorflow/core/grappler/optimizers/data/auto_shard.cc:461] The `assert_cardinality` transformation is currently not handled by the auto-shard rewrite and will be removed.
58/58 [==============================] - 10s 35ms/step - loss: 1.5253 - sparse_categorical_accuracy: 0.5300 - val_loss: 0.5346 - val_sparse_categorical_accuracy: 0.8352
Epoch 2/5
58/58 [==============================] - 1s 23ms/step - loss: 0.4301 - sparse_categorical_accuracy: 0.8680 - val_loss: 0.2713 - val_sparse_categorical_accuracy: 0.9221
Epoch 3/5
58/58 [==============================] - 1s 23ms/step - loss: 0.2776 - sparse_categorical_accuracy: 0.9163 - val_loss: 0.1985 - val_sparse_categorical_accuracy: 0.9421
Epoch 4/5
58/58 [==============================] - 1s 23ms/step - loss: 0.2191 - sparse_categorical_accuracy: 0.9341 - val_loss: 0.1620 - val_sparse_categorical_accuracy: 0.9508
Epoch 5/5
58/58 [==============================] - 1s 23ms/step - loss: 0.1864 - sparse_categorical_accuracy: 0.9437 - val_loss: 0.1379 - val_sparse_categorical_accuracy: 0.9587
2021-08-02 10:15:26.437084: W tensorflow/python/util/util.cc:348] Sets are not currently considered sequences, but this may change in the future, so consider avoiding using them.
INFO:tensorflow:Assets written to: /tf/model/saved_model/assets
I0802 10:15:26.888544 140280680830784 builder_impl.py:774] Assets written to: /tf/model/saved_model/assets
2021-08-02 10:15:26.930396: W tensorflow/core/grappler/optimizers/data/auto_shard.cc:461] The `assert_cardinality` transformation is currently not handled by the auto-shard rewrite and will be removed.
9/9 - 0s - loss: 0.1379 - sparse_categorical_accuracy: 0.9587
I0802 10:15:27.101272 140280680830784 mnist_main.py:170] Run stats:
{'accuracy_top_1': 0.9586588740348816, 'eval_loss': 0.13790059089660645, 'loss': 0.18641451001167297, 'training_accuracy_top_1': 0.9436624646186829}
^C
```

因为 TKE Serverless 即用即消的特性，导致如果需要查看日志，必须当且仅当 Pod 处于 Running 状态时才可查看。解决方法请参见常见问题 日志采集相关 。

查看存储

如果已经按照前面的配置 NFS，此时可以前往挂载点，查看 NFS 内存储：

1. 执行以下命令进入相关挂载目录，查看是否存在相关目录。

```
cd /mound_data
```

如下图所示：

```
[root@VM-32-40-centos ~]# cd /mount_data
[root@VM-32-40-centos mount_data]# ll
total 8
drwxr-xr-x 4 root root   34 Aug  2 18:36 data
drwxr-xr-x 3 root root 4096 Jul 21 15:59 dev
drwxr-xr-x 2 root root   51 Jul 21 15:58 etc
drwxr-xr-x 5 root root 4096 Aug  2 18:36 model
drwxr-xr-x 3 root root   16 Jul 21 15:58 var
```

2. 进入 model 目录，查看目录下是否有相关数据。如下图所示：

```
[root@VM-32-40-centos mount_data]# cd model
[root@VM-32-40-centos model]# ll
total 32144
-rw-r--r-- 1 root root      87 Aug  2 18:36 checkpoint
-rw-r--r-- 1 root root 6574829 Aug  2 18:36 model.ckpt-0001.data-00000-of-00001
-rw-r--r-- 1 root root     819 Aug  2 18:36 model.ckpt-0001.index
-rw-r--r-- 1 root root 6574829 Aug  2 18:36 model.ckpt-0002.data-00000-of-00001
-rw-r--r-- 1 root root     819 Aug  2 18:36 model.ckpt-0002.index
-rw-r--r-- 1 root root 6574829 Aug  2 18:36 model.ckpt-0003.data-00000-of-00001
-rw-r--r-- 1 root root     819 Aug  2 18:36 model.ckpt-0003.index
-rw-r--r-- 1 root root 6574829 Aug  2 18:36 model.ckpt-0004.data-00000-of-00001
-rw-r--r-- 1 root root     819 Aug  2 18:36 model.ckpt-0004.index
-rw-r--r-- 1 root root 6574829 Aug  2 18:36 model.ckpt-0005.data-00000-of-00001
-rw-r--r-- 1 root root     819 Aug  2 18:36 model.ckpt-0005.index
drwxr-xr-x 4 root root      80 Aug  2 18:36 saved_model
drwxr-xr-x 3 root root     143 Aug  2 18:36 train
drwxr-xr-x 2 root root      65 Aug  2 18:36 validation
```

3. 进入 data 目录，查看目录下是否有相关数据。如下图所示：

```
[root@VM-32-40-centos mount_data]# cd data
[root@VM-32-40-centos data]# ll
total 0
drwxr-xr-x 3 root root 22 Aug  2 18:36 downloads
drwxr-xr-x 3 root root 18 Aug  2 18:36 mnist
```

相关操作

在 TKE 上使用 GPU 部署深度学习任务

在 TKE 上部署深度学习任务与 TKE Serverless 的部署几乎没有区别。以使用 kubectl 通过 YAML 部署为例，需要进行以下两点改动：

- 创建 TKE 节点时，选择带有 GPU 的节点。详情请参见 新建 GPU 云服务器 。
- 因为节点自带 GPU 资源，因此无需 Annotations 和 Resources。在实践操作中，您可以保留 Annotations，TKE 不会处理这部分。但是，建议注释掉 Resources，因为在某些情况下可能会导致不合理的资源需求。

常见问题

在进行本实践过程中遇到的问题，请参见 常见问题 文档进行排查解决。

常见问题
公网访问相关

最近更新时间：2023-05-17 15:41:01

本文将提供在进行 构建深度学习容器镜像 和 在 TKE Serverless 上运行深度学习 实践时可能遇到的常见问题解答。

容器如何访问公网?

因为任务过程中可能需要下载训练用数据集，所以可能需要进行访问公网操作。而容器初始状态无法访问公网，直接运行带下载数据集的指令将会进行如下报错：

```
W tensorflow/core/platform/cloud/google_auth_provider.cc:184] All attempts to get a Google authentication
bearer token failed, returning an empty token. Retrieving token from files failed with "Not found: Could
not locate the credentials file.". Retrieving token from GCE failed with "Failed precondition: Error
executing an HTTP request: libcurl code 6 meaning 'Couldn't resolve host name', error details: Could not
resolve host: metadata".
E tensorflow/core/platform/cloud/curl_http_request.cc:614] The transmission  of request 0x5b328e0 (URI:
https://www.googleapis.com/storage/v1/b/tfds-data/o/dataset_info%2Fmnist%2F3.0.1?
fields=size%2Cgeneration%2Cupdated) has been stuck at 0 of 0 bytes for 61 seconds and will be aborted....
```

针对上述问题，提供两种访问公网的方式：

- **使用 NAT 网关**：适用于某个 VPC 下的多个实例需要与公网通信。请按照 通过 NAT 网关访问外网 文档进行操作。

  > ⚠ **注意**
  > 创建的 NAT 网关和路由表需要与 TKE Serverless 集群同地域、同私有网络 VPC。

- **使用弹性公网 IP（EIP）**：适用于单个或少量实例需要实现公网互通。请按照 使用弹性公网 IP 访问外网 文档进行操作。

日志采集相关

最近更新时间：2023-10-24 10:31:22

本文将提供在进行 构建深度学习容器镜像 和 在 TKE Serverless 上运行深度学习 实践时可能遇到的常见问题解答。

日志如何进行持久存储?

因为 TKE Serverless 即用即消的特性，导致如果想要查看日志，必须当且仅当 Pod 还在 Running 状态时查看。一旦 Pod 状态变为 Completed ，将会出现如下报错：

```
Error from server (InternalError): Internal error occurred: can not found connection to pod ***
```

以下为您介绍能将日志持久存储的方法：

方式一：重定向

重定向方法最为简单，只需将 `kubectl logs` 输出到终端的 stdout 转向输出到文件中即可持久化存储。执行命令如下：

```
kubectl logs -f tf-cnn >> info.log
```

但使用重定向方法时需要注意，输出流不会流向终端，也就是说在终端上将无法看到日志滚到哪一步。如果在将命令输出保存到文件中的同时，还需要将内容也输出到屏幕，有如下两种方法：

- 使用管道 + `tee` 命令，执行命令如下：

  ```
  kubectl logs -f tf-cnn |tee info.log
  ```

- 使用 `logsave` 命令，也可以做到将命令输出保存到文件的同时将内容页输出到屏幕中，执行命令如下：

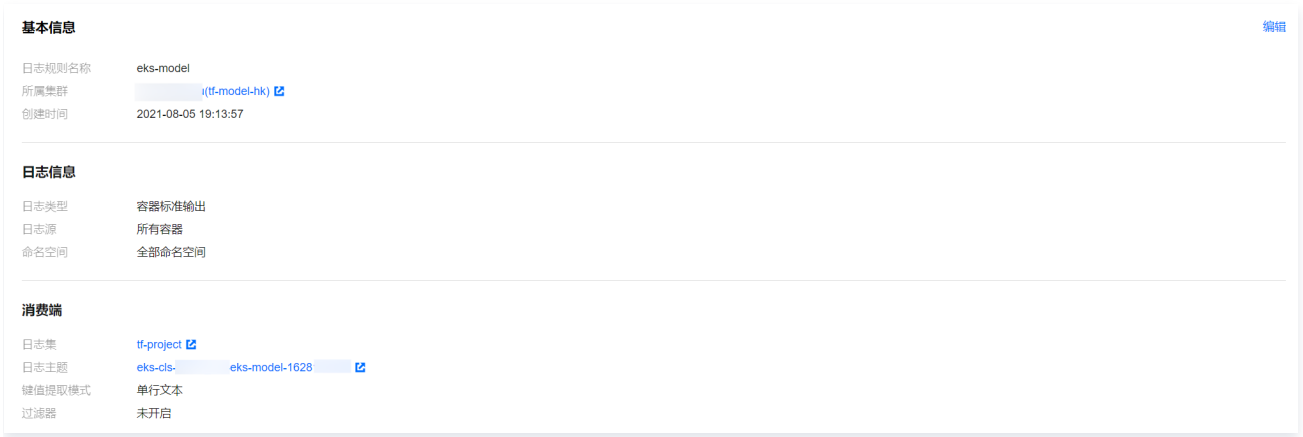
  ```
  logsave [-asv] info.log kubectl logs -f tf-cnn
  ```

  > **说明**
  > `logsave` 相较于 `tee` 的好处是， `logsave` 每次输入会记录下时间，并制造间隔，观感上也更便于查找某一段日志。

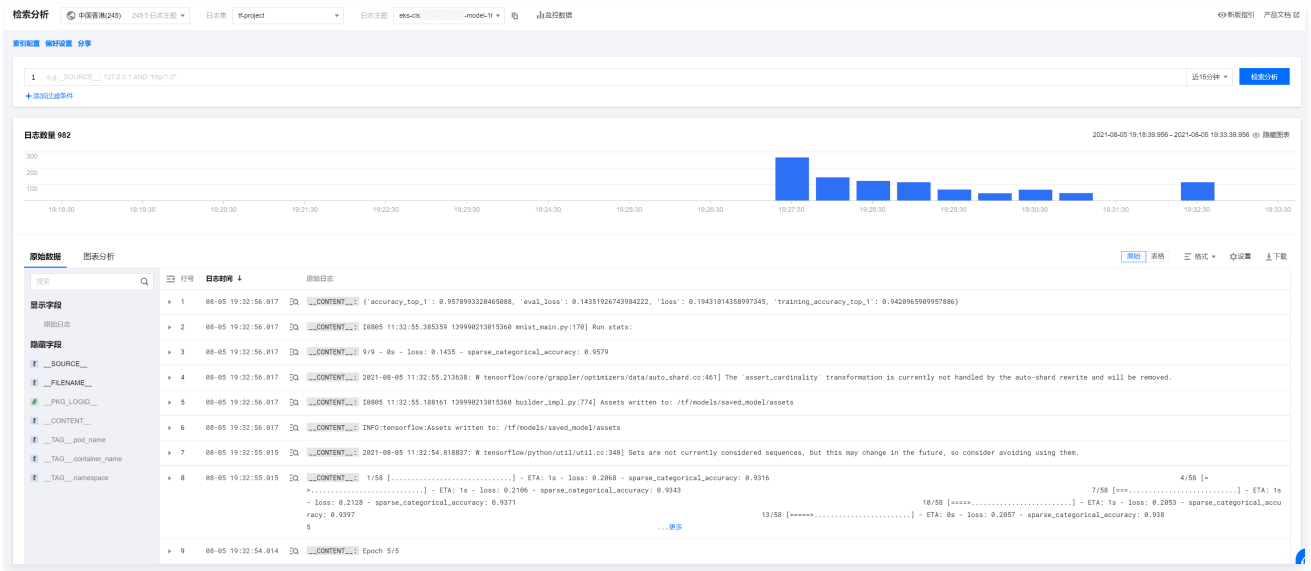
上述三条命令同时存在弊端，因为都是基于 `kubectl logs` 输出的重定向，使用时必须在 Running 状态时运行，起到的作用只是在 Completed 后依然能查看日志。重定向方法可适用于少量的日志，不存在大量的日志输出和检索需求的场景下。如果您的需求不高，那么推荐您使用重定向方法。

方式二：配置日志采集

在 TKE Serverless 集群中，可以通过 自定义资源（CRD）配置日志采集 。
开启日志采集后，新建如下图所示日志规则：



在检索分析端可以看到，时间粒度为毫秒级，而更小尺度也按顺序输出。



> **说明：**
> CRD 配置的日志采集支持通过正则划分原始日志。

配置日志采集可能遇到的问题：

如果选择 CRD 配置日志采集，请选择 Chrome 内核的浏览器（最新版 Edge、Chrome 浏览器）进行配置，而不是旧版 Edge 等。因为前端已经可能不支持旧版内核，会出现日志样例无法正常显示、正则表达式自动生成无法正常框取等问题。

在用 CRD 配置日志采集后，在新建 Pod 时无需其他操作，会自动获取输出的 log 。如果未采集到，考虑是否存在机器组已满的问题。使机器组额度有空余后，重启 cls-provisioner 的 Pod 即可。

Serverless 集群自定义 DNS 服务

最近更新时间：2023-09-08 19:13:06

> **说明**
> DNS Forward 配置的入口将不再开放。此前关于 DNS Forward 配置的参数会同步更新在 CoreDNS 的 Corefile 中，若需要修改集群的 DNS 服务，请参考以下操作，或可参考原生 Kubernetes CoreDNS 的使用方式。

操作场景

本文主要介绍如何通过修改 CoreDNS 配置文件，更改集群的 DNS 服务。

操作前提

已经 创建 Serverless 集群，创建时需要在高级配置中选择**部署 CoreDNS 支持集群内服务发现**，以支持集群内服务发现。

操作指引

默认 Corefile 配置说明

在 Serverless 集群中，部署 CoreDNS 会默认挂载一个 Configmap 作为 CoreDNS 的配置文件，即 Corefile。
CoreDNS 安装时默认的 Corefile 配置如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health :8081
        kubernetes cluster.local in-addr.arpa ip6.arpa {
           pods insecure
           fallthrough in-addr.arpa ip6.arpa
           ttl 30
        }
        prometheus :9153
        forward . 183.60.83.19 183.60.82.98
        cache 30
        loop
        reload
        loadbalance
    }
```

其中各个配置项均采用原生 Kubernetes 的配置，详情见 CoreDNS 。需注意：

- `forward`：183.60.83.19，183.60.82.98 为腾讯云默认 DNS 地址。

自定义配置 Corefile

您可以通过修改 CoreDNS Corefile 的 ConfigMap，以更改服务发现的相关配置。其用法与原生 Kubernetes 使用方式保持一致，详情见 自定义 DNS 服务 。

调度
安装 CoScheduling 实现批调度

最近更新时间：2024-05-23 15:10:52

背景

针对 AI、大数据等多任务协作场景，对调度有 “All-or-Nothing” 的需求，即所有的任务在同一时间被调度。CoScheduling 是一套开源的方案，在 Kubernetes 集群中将一组 Pod（或称为 PodGroup）同时调度到同一个节点上。本文将介绍如何在 TKE 上安装 CoScheduling 实现批调度。

前提条件

- 已创建了 TKE 集群。
- 已安装了 Helm 。
- 已配置了 TKE 集群的 kubeconfig，并具有操作 TKE 集群的权限。详情请参见 连接集群。

使用 Helm 安装

将 CoScheduler 作为第二调度器完成安装

pod 调度时需要指定 schedulerName 为 scheduler-plugins-scheduler。命令示例如下：

```
$ git clone git@github.com:kubernetes-sigs/scheduler-plugins.git
$ cd scheduler-plugins/manifests/install/charts
$ helm install scheduler-plugins as-a-second-scheduler/ --create-namespace --namespace scheduler-plugins
```

验证安装成功

执行如下命令，观察 Pod 运行情况。

```
$ kubectl get deploy -n scheduler-plugins
```

预期输出：

```
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
scheduler-plugins-controller   1/1     1            1           7s
scheduler-plugins-scheduler    1/1     1            1           7s
```

如何使用

PodGroup

PodGroup 是 CoScheduling 组件自定义资源，用来定义最少需要同时调度的 Pod 数。通过设置标签定义 Pod 属于哪一个 PodGroup。以下是 PodGroup 的 CRD 规范示例：

```
# PodGroup CRD spec
apiVersion: scheduling.x-k8s.io/v1alpha1
kind: PodGroup
metadata:
  name: nginx
spec:
  scheduleTimeoutSeconds: 10
  minMember: 3
---
# Add a label `scheduling.x-k8s.io/pod-group` to mark the pod belongs to a group
labels:
  scheduling.x-k8s.io/pod-group: nginx
```

我们将在调度程序中计算正在运行的 pod 和正在等待的 pod（假设但未绑定）的总和，如果总和大于或等于 minMember，则将创建等待 pod。同一 PodGroup 中具有不同优先级的 Pod 可能会导致意外行为，因此需要确保同一 PodGroup 中的 Pod 具有相同的优先级。

示例

假设我们有一个只能容纳3个 nginx pod 的集群。我们创建一个 replicas=6 的 ReplicaSet，并将 minMember 的值设置为3。

```
apiVersion: scheduling.x-k8s.io/v1alpha1
kind: PodGroup
metadata:
  name: nginx
spec:
  scheduleTimeoutSeconds: 10
  minMember: 3
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 6
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
        scheduling.x-k8s.io/pod-group: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            cpu: 3000m
            memory: 500Mi
          requests:
            cpu: 3000m
            memory: 500Mi
```

3个 Pod 将一起被调度，如下：

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
nginx-4jw2m   0/1     Pending   0          55s
nginx-4mn52   1/1     Running   0          55s
nginx-c9gv8   1/1     Running   0          55s
nginx-frm24   0/1     Pending   0          55s
nginx-hsflk   0/1     Pending   0          55s
nginx-qtj5f   1/1     Running   0          55s
```

如果此时修改 minMember 为4，因不满足 PodGroup 定义的 minMember 为3的要求，所有的 nginx pod 都处于 pending 状态：

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
```

```
nginx-4vqrk   0/1     Pending   0          3s
nginx-bw9nn   0/1     Pending   0          3s
nginx-gnjsv   0/1     Pending   0          3s
nginx-hqhhz   0/1     Pending   0          3s
nginx-n47r7   0/1     Pending   0          3s
nginx-n7vtq   0/1     Pending   0          3s
```

原生节点提升集群装箱率

最近更新时间：2023-11-28 17:54:21

操作场景

原生节点专用调度器 可以有效解决集群中装箱率高但利用率低的问题。通过使用原生节点专用调度器的节点放大能力，可以提高节点的装箱率，从而提升整体资源利用率，而无需对业务进行任何修改或重启操作。
然而，放大系数的配置应该如何确定？相应的水位线又应该如何搭配使用，以确保节点放大后的稳定性？这些问题直接关系到功能的稳定性和有效性。此外，放大能力带来的收益和风险具体有哪些？

原生节点放大的利与弊

收益	风险
1. 提高资源利用率：通过虚拟放大，可以更有效地利用节点的计算和存储能力，防止资源被占用后的空闲问题。这有助于降低成本，从而提高整体运行效率。 2. 业务零成本使用：原生节点放大能力调整的是节点的可调度容量，对业务零侵入零改造零迁移。这有助于快速测试新功能，并应用到实际生产环境中。	1. 资源争抢：如果节点上运行的容器都试图使用超分配的资源，这可能导致资源争抢，从而降低系统性能和稳定性。 2. 注意放大过度：如果节点上的工作负载的实际需求超过了可用资源，可能导致业务受损甚至导致系统崩溃和停机。

本文以第一视角的方式提供原生节点放大能力的最佳实践，帮助您充分发挥放大能力的同时降低功能风险。最佳实践主要包括以下五个步骤：

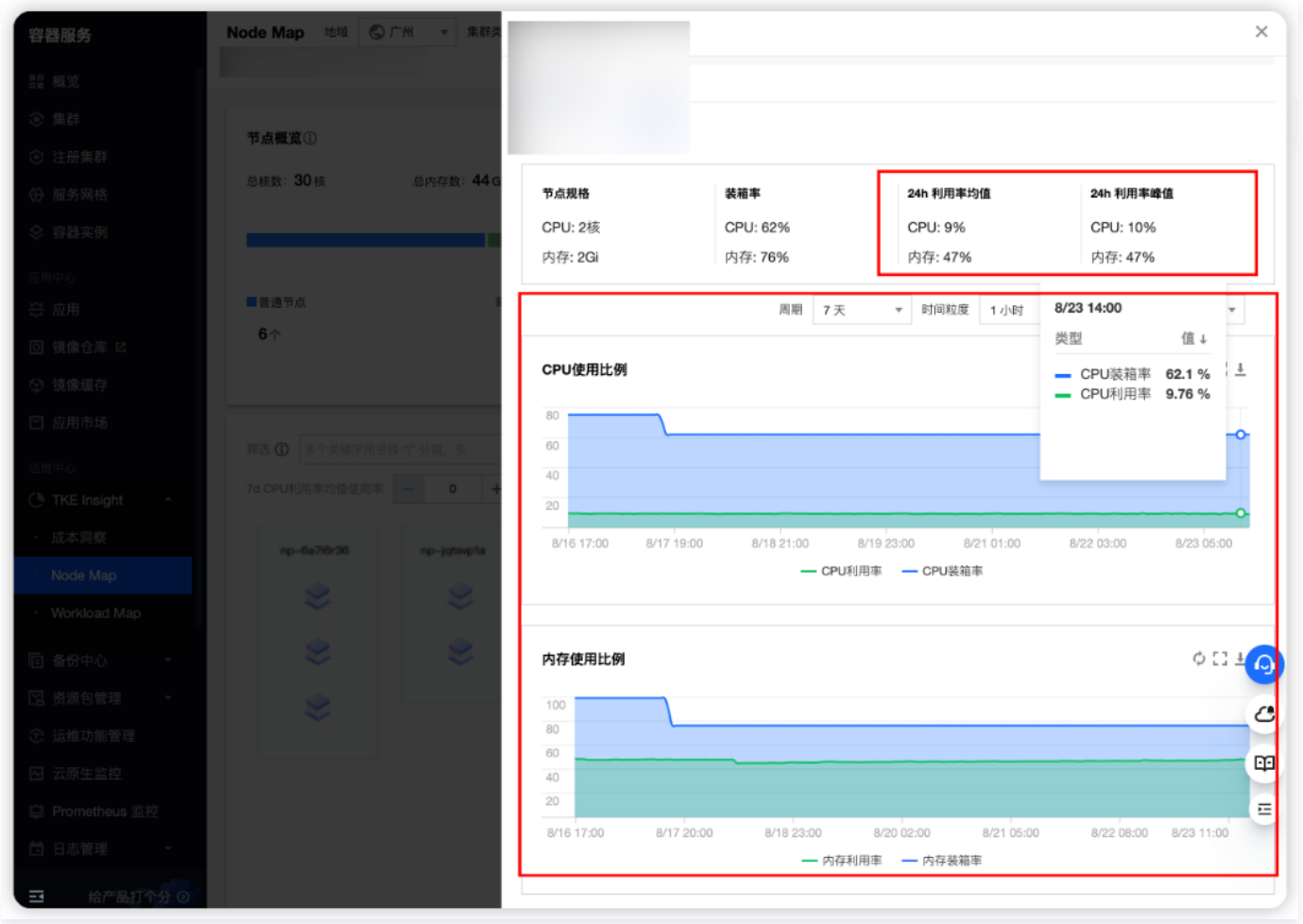
- 步骤1：寻找典型的需要放大的节点，即装箱率高但利用率低的节点。
- 步骤2：确定节点的利用率目标。只有明确目标，才能确定合理的放大系数配置数值。
- 步骤3：根据节点利用率目标和现状确定放大系数和水位线。
- 步骤4：选择目标节点，将这些节点上的 Pod 调度到放大的节点上。
- 步骤5：在步骤4中重新调度 Pod 运行后，可以下线目标节点。

操作步骤

步骤1：观察节点当前装箱率和利用率

> **说明：**
> **装箱率和利用率定义如下：**
> - 装箱率：节点上所有 Pod 的 Request 之和除以节点的真实容量规格。
> - 利用率：节点上所有 Pod 的实际使用量之和除以节点的真实容量规格。

TKE 提供了 TKE Insight，方便您直接查看节点的装箱率和利用率走势图，详情请参见 Node Map 。



装箱率和利用率的关系的分析如下：

1. 装箱率高，利用率高

例如：装箱率90%以上，CPU利用率50%以上，或者内存利用率90%以上。
说明：节点资源使用情况合理，整体安全稳定，但需要注意节点内存可能会出现 OOM（Out of Memory）的情况。
建议：最好配置运行时水位90%，以防止节点稳定性风险。

2. 装箱率高，利用率低

例如：装箱率90%以上，CPU 利用率50%以下（如：10%），或者内存利用率90%以下（30%）。
说明：节点上的 Pod 存在过配的情况，即资源的申请量远大于实际使用量。由于装箱率已经很高，无法调度更多的 Pod，导致节点利用率无法提升。
建议：通过虚拟放大原生节点的规格，让节点的装箱率突破100%的上限，从而调度更多的 Pod，提升节点的利用率。

3. 装箱率低，利用率高

例如：装箱率90%以下（如50%），CPU 利用率50%以上，或者内存利用率90%以上。
说明：节点上的 Pod 普遍存在超卖场景，即 Limit（资源上限）大于 Request（资源需求），或者 Pod 没有配置 Request。
建议：这样配置的 Pod 的 QoS（Quality of Service）等级较低，在节点高负载时可能会重启甚至重新调度。需要确认这种配置的 Pod 是否是低优先级 Pod。此外，还建议再配置运行时水位90%，以防止节点稳定性风险。

4. 装箱率低，利用率低

例如：装箱率90%以下（如50%），CPU 利用率50%以下（如：10%），或者内存利用率90%以下（30%）。
说明：节点没有充分使用。
建议：可以调度更多的 Pod 到该节点上，或者将该节点上的 Pod 驱逐到其他节点上后，下线该节点。另外，也可以考虑更换一个更小规格的节点。

步骤2：定节点利用率目标

在设置合理的节点放大系数和水位线时，需要确定节点的利用率目标，以确保高利用率的同时防止节点出现异常。以下是涉及多个利用率指标的示例：

- 节点 CPU 利用率：根据腾讯内部上云成熟度大规模落地的经验，将节点的 CPU 利用率峰值设定为50%是一个比较理想的目标。

- 节点内存利用率：通过对百万业务规模的分析，发现节点的内存利用率普遍较高，且波动没有 CPU 大。因此，将节点的内存利用率峰值设定为90%是一个比较理想的目标。

根据实际情况和业务需求，您可以根据这些指标来设定节点的利用率目标，以便在放大节点时保持节点的稳定性和高效利用。

步骤3：定放大系数和水位线

在了解节点当前利用率现状和目标后，可以确定节点的放大系数和水位线，以达到目标利用率。放大系数表示节点容量可以放大的倍数。示例如下：

- 假设当前利用率为20%，目标利用率为40%。这意味着还可以放入当前一倍的业务到该节点上，因此节点的放大系数需要配置为2。
- 假设当前利用率为15%，目标利用率为45%。这意味着还可以放入当前三倍的业务到该节点上，因此节点的放大系数需要配置为3。

> **注意：**
> 通常使用峰值来查看当前利用率，以确保在业务波峰时有足够的资源使用。

放大系数的计算公式如下：

- CPU 放大系数 = CPU 目标利用率 / CPU 当前利用率
- 内存放大系数 = 内存目标利用率 / 内存当前利用率

以 步骤1 中的示例为例：
当前 CPU 利用率峰值为10%，内存利用率峰值为47%。假设 CPU 目标利用率为50%，内存目标利用率为90%。
则 CPU 的放大系数为5（注意不要设置得太高，以免 CPU 够用但节点内存出现瓶颈）；内存的放大系数为2。
确定目标利用率后，可以根据目标来设置水位线。例如：

- 调度时水位：建议设置小于等于目标利用率，以允许未达到目标利用率的节点持续调度 Pod。设置得太高可能导致节点过负载。例如，如果目标 CPU 利用率为50%，可以将 CPU 的调度时水位设置为40%。
- 运行时水位：建议设置大于等于目标利用率，以防止利用率过高导致节点过负载。例如，如果目标 CPU 利用率为50%，可以设置 CPU 的运行时水位为60%。

步骤4：往放大的节点调度 Pod

只有将 Pod 调度到放大的节点上，才能提升节点的资源利用率。有两种方式可以实现：
1. 封锁其他节点：将新需要调度的 Pod 仅调度到放大的节点上，阻止其他节点接收新的 Pod 调度请求。
2. 使用 Workload 的标签选择器能力：通过标签选择器，将 Pod 指定调度到放大的节点上。

建议：
在进行节点放大时，最好选择那些容易下线的节点，将这些节点上的 Pod 重新调度到放大的节点上。这些节点可能包括：

- Pod 数量很少的节点。
- 按量计费节点。
- 快到期的包年包月的节点。

如果您指定了节点的 CPU 和内存放大系数，可以通过查看与放大系数相关的 Annotation：`expansion.scheduling.crane.io/cpu`，`expansion.scheduling.crane.io/memory` 来确认。示例如下：

```
kubectl describe node 10.8.22.108
...
Annotations:        expansion.scheduling.crane.io/cpu: 1.5  # CPU 放大系数
                    expansion.scheduling.crane.io/memory: 1.2   # 内存放大系数
...
Allocatable:
  cpu:                1930m # 该节点原始可调度资源量
  ephemeral-storage:  47498714648
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             1333120Ki
  pods:               253
...
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests         Limits
  --------           --------         ------
  cpu                960m (49%)       8100m (419%)   # 该节点 Request 和 Limit 占用量
```

```
  memory             644465536 (47%)  7791050368 (570%)
  ephemeral-storage  0 (0%)           0 (0%)
  hugepages-1Gi      0 (0%)           0 (0%)
  hugepages-2Mi      0 (0%)           0 (0%)
...
```

说明：
当前节点的原始 CPU 可调度量为1930m，节点上所有 Pod 的 CPU 请求总量为960m。在正常情况下，该节点最多只能调度970m的 CPU 资源（1930m - 960m）。然而，通过虚拟放大，该节点的 CPU 可调度量已经增加到2895m（1930m * 1.5），实际剩余的 CPU 可调度资源为1935m（2895m - 960m）。
此时，如果创建一个工作负载，只有一个 Pod，其 CPU 请求量为1500m，如果没有节点放大的能力，则无法将该 Pod 调度到该节点上。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: default
  name: test-scheduler
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector: # 指定节点调度
        kubernetes.io/hostname: 10.8.20.108 # 指定使用被样例中的原生节点
      containers:
      - name: nginx
        image: nginx:1.14.2
        resources:
          requests:
            cpu: 1500m  # 申请量大于放大之前的节点可调度量，但有小于放大之后的节点可调度量
        ports:
        - containerPort: 80
```

该工作负载创建成功：

```
% kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
test-scheduler     1/1     1            1           2m32s
```

再次检查节点的资源占用情况：

```
kubectl describe node 10.8.22.108
...
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests         Limits
  --------           --------         ------
  cpu                2460m (127%)     8100m (419%)  # 该节点 Request 和 Limit 占用量。可以看到，Request 总和超
过了节点原始可调度量，节点规格放大成功。
  memory             644465536 (47%)  7791050368 (570%)
```

```
  ephemeral-storage  0 (0%)           0 (0%)
  hugepages-1Gi      0 (0%)           0 (0%)
  hugepages-2Mi      0 (0%)           0 (0%)
```

步骤5：移除多余节点

在 步骤4 中选择的节点上，当节点上的非 DaemonSet 的 Pod 全部移出后，可以将该节点移出并删除，从而用更少的节点承载相同的业务量。
通过以上方式，可以对集群中的 Pod 进行规整。例如，假设集群中有20个原生节点和20个普通节点，规格相同，整体资源利用率为 CPU 10%和内存 40%。通过将20个原生节点的 CPU 和内存放大一倍，可以将20个普通节点上的 Pod 迁移至原生节点上。这样，原生节点的资源利用率将提高到 CPU 20%和内存 80%。同时，普通节点上将没有 Pod 存在，因此可以将这些普通节点从集群中移除，从而减少了一半的节点规模。

安全
Pod 安全组

最近更新时间：2023-09-08 19:13:06

Pod 安全组将腾讯云 CVM 安全组与 Kubernetes Pod 集成。您可以使用腾讯云 CVM 安全组来定义规则，以允许多种 TKE 节点类型上运行的 Pod 的网络流量进出。目前，我们只支持超级节点，但后续会逐步扩展支持普通节点等。

限制条件

在为 Pod 使用安全组之前，请考虑以下限制条件：

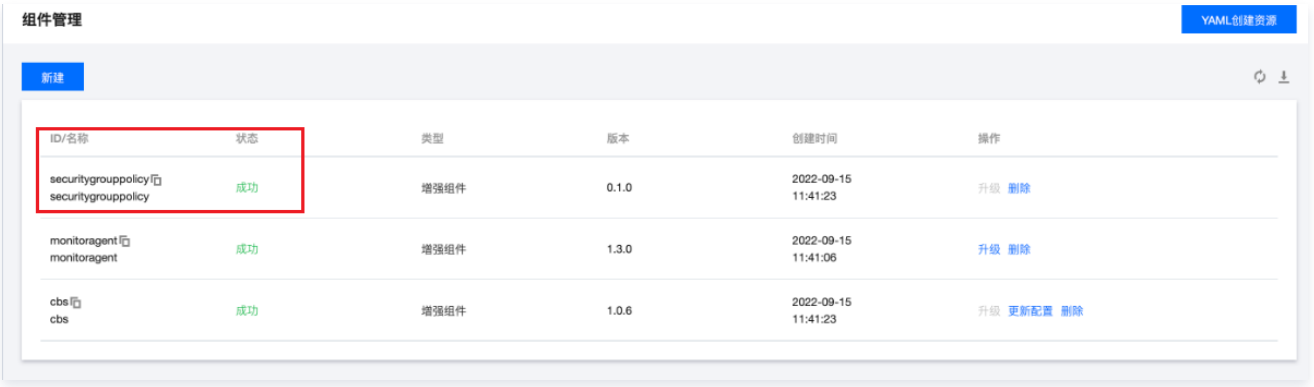
- Pod 必须运行在 TKE 1.20 或更高版本的集群中。
- Pod 的安全组目前仅支持超级节点，其他类型节点后续上线。
- Pod 的安全组不能与双栈集群一起使用。
- 超级节点仅支持部分地域，请参考 超级节点支持地域 。

为 Pod 启用安全组能力

安装扩展组件

1. 登录 容器服务控制台 。
2. 为集群安装 SecurityGroupPolicy（安全组策略）组件。
   - 如果您还没有创建集群，可以在创建集群的时候安装 SecurityGroupPolicy 组件。详情见 通过集群创建页安装 。
   - 如果您需要给已创建好的集群中的 Pod 开启安全组能力，请在组件管理中安装 SecurityGroupPolicy 组件。详情见 通过组件管理页安装 。

3. 在组件管理页面查看组件状态。如组件状态为“成功”，代表组件部署完成。如下图所示：



4. 在超级节点页面，确认您的 TKE 标准集群已包含超级节点，目前仅支持调度到超级节点上的 Pod 开启安全组能力。

部署示例应用程序

要对 Pod 使用安全组，您必须将 SecurityGroupPolicy 部署到您的集群。以下步骤向您展示了如何使用 CloudShell 为 Pod 使用安全组策略。除非另有说明，否则请从同一终端完成所有步骤，因为在以下步骤中使用的变量不会跨终端持续存在。

使用安全组部署示例 Pod

1. 创建一个安全组以与您的 Pod 一起使用。以下步骤可帮助您创建一个简单的安全组，仅用于说明目的。在生产集群中，您的规则可能会有所不同。

   1.1 检索集群的 VPC 和集群安全组的 ID。您在使用时可替换 `my-cluster` 。

```
my_cluster_name=my-cluster
my_cluster_vpc_id=$(tccli tke DescribeClusters --cli-unfold-argument --ClusterIds $my_cluster_name
--filter Clusters[0].ClusterNetworkSettings.VpcId | sed 's/\"//g')
my_cluster_security_group_id=$(tccli vpc DescribeSecurityGroups --cli-unfold-argument --
Filters.0.Name security-group-name --Filters.0.Values tke-worker-security-for-$my_cluster_name --
filter SecurityGroupSet[0].SecurityGroupId | sed 's/\"//g')
```

   1.2 为您的 Pod 创建安全组。您在使用时可替换 `my-pod-security-group` 。记下运行命令后输出中返回的安全组 ID，您将在后面的步骤中使用它。

```
my_pod_security_group_name=my-pod-security-group
tccli vpc CreateSecurityGroup --GroupName "my-pod-security-group" --GroupDescription "My pod
security group"
my_pod_security_group_id=$(tccli vpc DescribeSecurityGroups --cli-unfold-argument --Filters.0.Name
security-group-name --Filters.0.Values my-pod-security-group --filter
SecurityGroupSet[0].SecurityGroupId | sed 's/\"//g')
echo $my_pod_security_group_id
```

   1.3 允许您上一步中创建的 Pod 安全组到集群安全组的 TCP 和 UDP 端口53流量，以允许部署示例中 Pod 可以通过域名访问应用程序。

```
tccli vpc CreateSecurityGroupPolicies --cli-unfold-argument --SecurityGroupId
$my_cluster_security_group_id --SecurityGroupPolicySet.Ingress.0.Protocol UDP --
SecurityGroupPolicySet.Ingress.0.Port 53 --SecurityGroupPolicySet.Ingress.0.SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Action ACCEPT
tccli vpc CreateSecurityGroupPolicies --cli-unfold-argument --SecurityGroupId
$my_cluster_security_group_id --SecurityGroupPolicySet.Ingress.0.Protocol TCP --
SecurityGroupPolicySet.Ingress.0.Port 53 --SecurityGroupPolicySet.Ingress.0.SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Action ACCEPT
```

   1.4 需要允许任何协议和端口从安全组关联的 Pod 到任意安全组关联的 Pod 的入站流量。并且允许安全组关联的 Pod 的任何协议和端口的出站流量。

```
tccli vpc CreateSecurityGroupPolicies --cli-unfold-argument --SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Protocol ALL --
```

```
SecurityGroupPolicySet.Ingress.0.Port ALL --SecurityGroupPolicySet.Ingress.0.SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Action ACCEPT
tccli vpc CreateSecurityGroupPolicies --cli-unfold-argument --SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Egress.0.Protocol ALL --
SecurityGroupPolicySet.Egress.0.Port ALL --SecurityGroupPolicySet.Egress.0.Action ACCEPT
```

2. 创建一个 Kubernetes 命名空间来部署资源。

```
kubectl create namespace my-namespace
```

3. 将 SecurityGroupPolicy 部署到您的集群。

3.1 将以下示例安全策略保存为 `my-security-group-policy.yaml` 。如果您更愿意根据服务账户标签选择 Pod，则可以替换 podSelector 为 serviceAccountSelector，您必须指定一个或另一个选择器。如果指定多个安全组，则所有安全组中的所有规则都会对选定的 Pod 有效。将 `$my_pod_security_group_id` 替换为您在上一步中为 Pod 创建安全组时记下的安全组 ID 。

```
apiVersion: vpcresources.tke.cloud.tencent.com/v1beta1
kind: SecurityGroupPolicy
metadata:
  name: my-security-group-policy
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      app: my-app
  securityGroups:
    groupIds:
      - $my_pod_security_group_id
```

> **注意**
>
> 您为 Pod 指定的一个或多个安全组必须满足以下条件：
>
> - 它们必须存在。
> - 它们必须允许来自集群安全组（for kubelet）的入站请求，允许给 Pod 配置的健康检查可以工作。
> - 您的 CoreDNS pod 的安全组必须允许 Pod 安全组的入站 TCP 和 UDP 端口53流量。
> - 它们必须具有必要的入站和出站规则才能与其他 Pod 进行通信。
>
> 安全组策略仅适用于新调度的 Pod。它们不会影响正在运行的 Pod。如需存量 Pod 生效，则需要您确认存量 Pod 满足上述条件后手动重建。

3.2 部署策略。

```
kubectl apply -f my-security-group-policy.yaml
```

4. 部署示例应用程序使用您在上一步中 podSelector 指定的 my-app 匹配标签。

4.1 将以下内容保存到名为 `sample-application.yaml` 。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  namespace: my-namespace
  labels:
    app: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
```

```
  template:
    metadata:
      labels:
        app: my-app
    spec:
      terminationGracePeriodSeconds: 120
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
      nodeSelector:
        node.kubernetes.io/instance-type: eklet
      tolerations:
      - effect: NoSchedule
        key: eks.tke.cloud.tencent.com/eklet
        operator: Exists
---
apiVersion: v1
kind: Service
metadata:
  name: my-app
  namespace: my-namespace
  labels:
    app: my-app
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

4.2 使用以下命令部署应用程序。当您部署应用程序时，Pod 会优先调度到超级节点上，并且将应用您在上一步中指定的安全组到 Pod 上。

```
kubectl apply -f sample-application.yaml
```

> **注意**
> 如果您没有使用 nodeSelector 优先调度到超级节点，当 Pod 调度到其他节点的时候，安全组是不生效的并且 `kubectl describe pod` 会输出 `security groups is only support super node, node 10.0.0.1 is not super node` 。

5. 查看使用示例应用程序部署的 Pod。此时该终端称为 `TerminalA` 。

```
kubectl get pods -n my-namespace -o wide
```

示例输出如下：

```
NAME                             READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
my-deployment-866ffd8886-9zfrp   1/1     Running   0          85s   10.0.64.10   eklet-subnet-
q21rasu6-8bpgyx9r   <none>           <none>
my-deployment-866ffd8886-b7gzb   1/1     Running   0          85s   10.0.64.3    eklet-subnet-
q21rasu6-8bpgyx9r   <none>           <none>
```

6. 在另一个终端中进入任意 Pod，此终端称为 `TerminalB` 。替换为上一步输出中返回的 Pod ID。

```
kubectl exec -it -n my-namespace my-deployment-866ffd8886-9zfrp -- /bin/bash
```

7. 在终端 `TerminalB` 中确认示例应用程序工作正常。

```
curl my-app
```

示例输出如下：

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

您收到了响应是因为运行应用程序的所有 Pod 都与您创建的安全组关联。该安全组包含规则有：

- 允许与安全组关联的所有 Pod 之间的所有流量。
- 允许 DNS 流量从该安全组出站到您的节点关联的集群安全组，这些节点正在运行 CoreDNS Pod，您的 Pod 会对 `my-app` 进行域名查找。

8. 从 `TerminalA` 中，从集群安全组中删除允许 DNS 通信的安全组规则。

```
tccli vpc DeleteSecurityGroupPolicies --cli-unfold-argument --SecurityGroupId
$my_cluster_security_group_id --SecurityGroupPolicySet.Ingress.0.Protocol UDP --
SecurityGroupPolicySet.Ingress.0.Port 53 --SecurityGroupPolicySet.Ingress.0.SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Action ACCEPT
tccli vpc DeleteSecurityGroupPolicies --cli-unfold-argument --SecurityGroupId
$my_cluster_security_group_id --SecurityGroupPolicySet.Ingress.0.Protocol TCP --
SecurityGroupPolicySet.Ingress.0.Port 53 --SecurityGroupPolicySet.Ingress.0.SecurityGroupId
$my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Action ACCEPT
```

9. 从 `TerminalB`，尝试再次访问应用程序。

```
curl my-app
```

尝试失败，因为 Pod 不能访问 CoreDNS Pod，集群安全组不再允许从与您安全组关联的 Pod 中进行 DNS 通信。
如果您尝试使用 IP 地址访问应用程序，您仍然会收到响应，因为所有端口都允许在具有与其关联的安全组的 pod 之间进行，并且不需要域名查找。

10. 完成试验后，您可以使用以下命令删除创建的示例安全组策略、应用程序和安全组。

```
kubectl delete namespace my-namespace
tccli vpc DeleteSecurityGroup --cli-unfold-argument --SecurityGroupId $my_pod_security_group_id
```

容器镜像签名及验证

最近更新时间：2023-05-17 15:41:01

镜像签名和验签功能可避免中间人攻击和非法镜像的更新及运行，进而实现镜像从分发到部署的全链路一致性。

容器镜像签名

腾讯云容器镜像服务（Tencent Container Registry，TCR）企业版支持开启命名空间级别的镜像自动签名特性，在推送镜像到仓库时自动匹配签名策略并完成加签动作，保障您仓库下的镜像内容可信。相关操作文档请查看 容器镜像签名 。

镜像签名验证

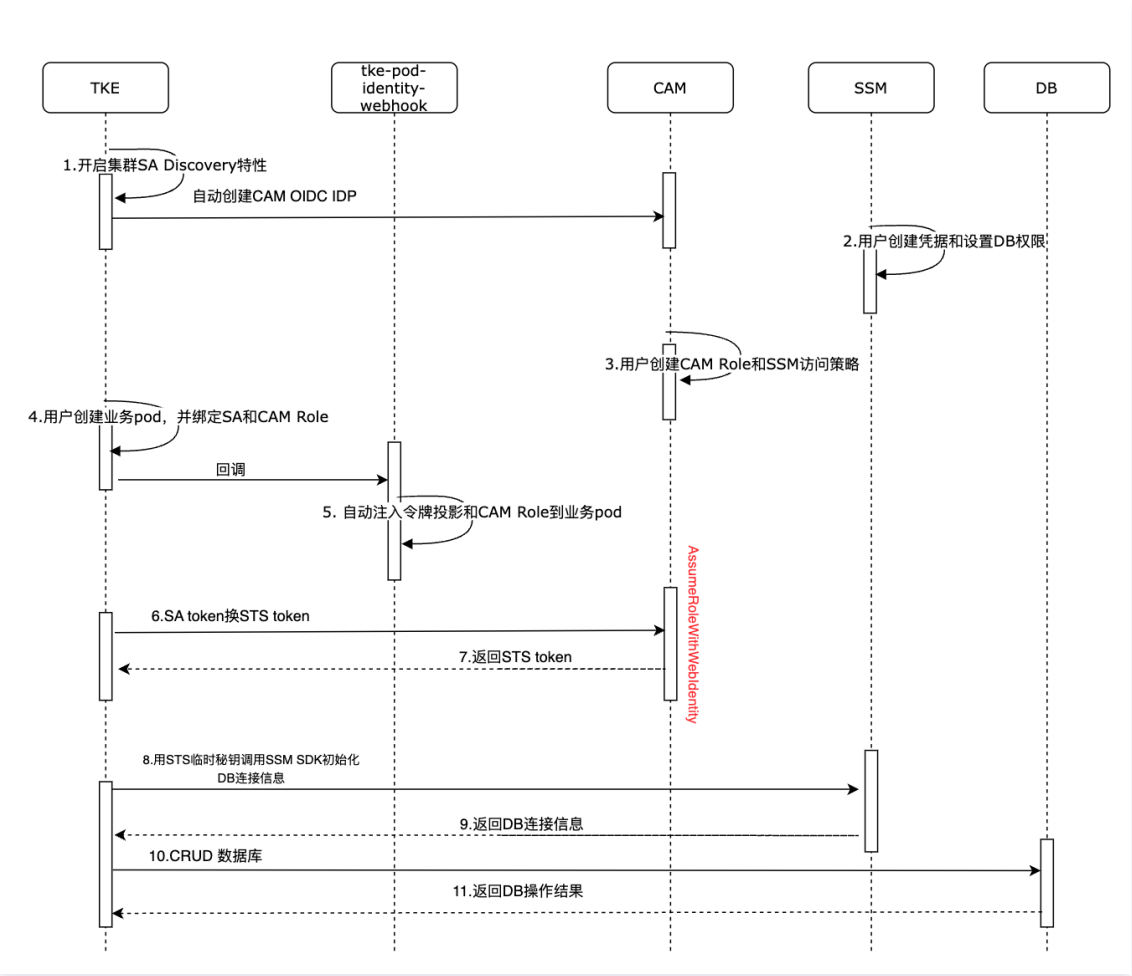
腾讯云容器服务（Tencent Kubernetes Engine，TKE）提供镜像签名验证组件 Cerberus，支持对签名镜像进行可信验证，确保在 TKE 集群中只部署可信授权方签名的容器镜像，降低在容器环境中的镜像安全风险。相关操作文档请查看 Cerberus 组件 。

Pod 使用 CAM 对数据库身份验证

最近更新时间：2024-09-12 11:05:31

使用背景

在腾讯云托管集群中运行容器化的工作负载时，通常需要访问存储在 Kubernetes 集群之外的一个或多个 SQL 或 NoSQL 数据库，但是将 SQL 数据库与 Kubernetes 一起使用时，存在定期轮换凭证和敏感信息传递到 Kubernetes 集群中的问题。为此，借助凭据管理系统（SSM）和腾讯云访问控制管理（CAM）来简化访问腾讯云数据库的整个过程，从而消除验证腾讯云数据库用户名和密钥存在的安全风险；同时凭据管理系统（SSM）定时轮转访问凭证的特性，间接解决人为操作所带来的负担。
本文向您介绍运行在腾讯云容器服务 TKE 上的工作负载如何使用 CAM 对数据库身份验证。在示例中，首先在腾讯云数据库和凭据管理系统（SSM）中分别创建一个数据库实例和数据库凭据；然后开启 OIDC 资源访问控制能力，将创建的 CAM OIDC 提供商作为创建角色的载体，并关联访问腾讯云数据库和凭据管理系统（SSM）的策略；最后利用 Kubernetes 服务账户、腾讯云访问控制管理（CAM）以及凭据管理系统（SSM）安全地连接到腾讯云数据库。整体架构如下图所示：



限制条件

本示例中，假定您已完成以下限制条件：

- 该功能仅支持 TKE 托管集群。
- 集群版本 ≥ v1.20.6-tke.27/v1.22.5-tke.1。
- 业务 Pod 可访问外网。

操作步骤

步骤1：准备托管集群

1. 登录 容器服务控制台，新建集群。

> **说明：**
> - 如果您没有托管集群，您可以使用容器服务控制台创建 TKE 标准集群，详情见 创建集群 。
> - 如果您已有托管集群，请在集群详情页检查集群版本，当集群版本不满足要求时，请升级集群。对运行中的 Kubernetes 集群进行升级，详情见 升级集群 。

2. 执行如下命令，确保您可以通过 kubectl 客户端访问托管集群。

```
kubectl get node
```

返回如下结果，则说明可正常访问集群。

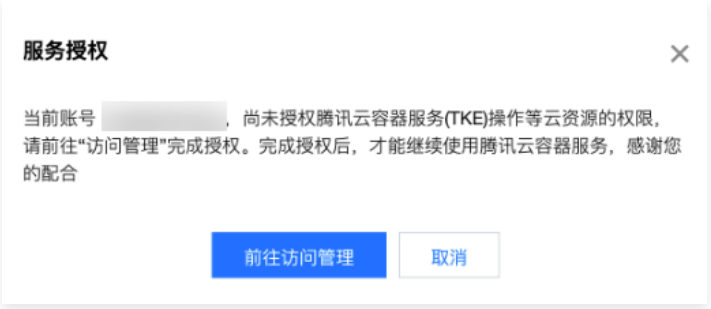
```
NAME          STATUS   ROLES    AGE   VERSION
10.0.4.144    Ready    <none>   24h   v1.22.5-tke.1
```

> **说明：**
> 您可以通过 Kubernetes 命令行工具 Kubectl 从本地客户端机器连接到 TKE 集群。详情见 连接集群 。

步骤2：开启 OIDC 资源访问控制能力

1. 在集群详情页中，单击 ServiceAccountIssuerDiscovery 右侧的 。如下图所示：

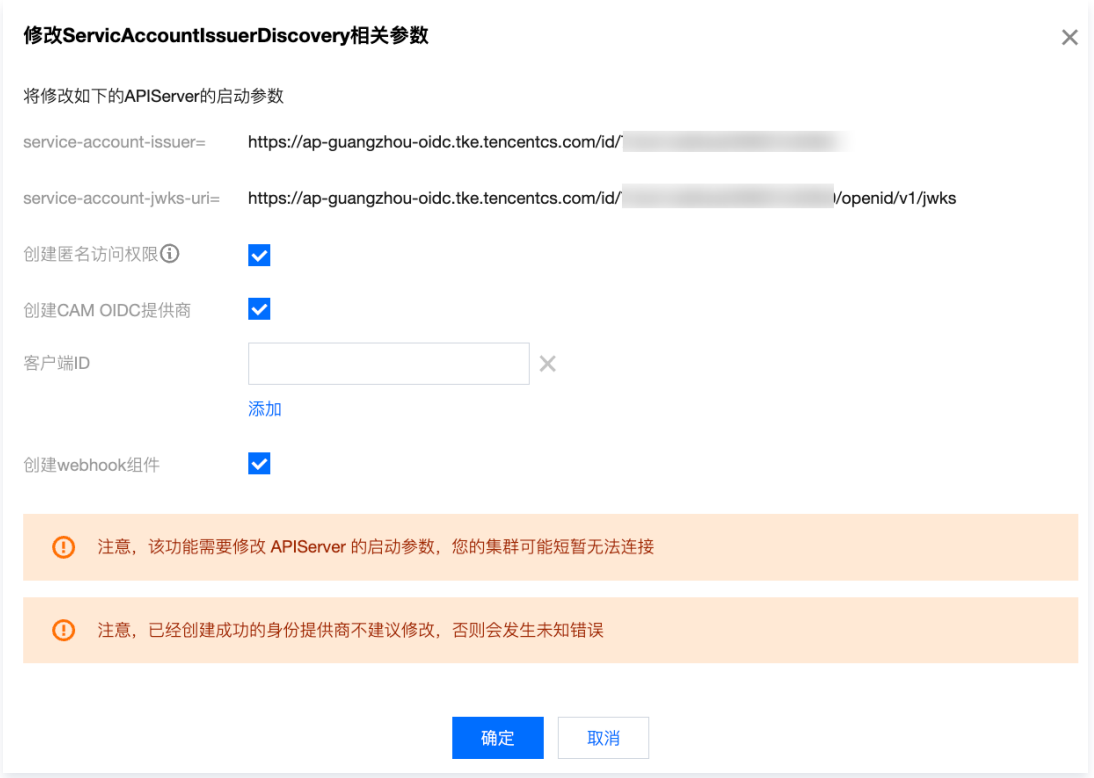
2. 进入**修改 ServiceAccountIssuerDiscovery 相关参数**页面，若系统提示您无法修改相关参数，请先进行服务授权。



在角色管理页面，查看授权策略 QcloudAccessForTKERoleInOIDCConfig，单击**同意授权**。

3. 授权完毕后，勾选“创建 CAM OIDC 提供商”和“创建webhook组件”，并填写客户端 ID，单击**确定**。如下图所示：

> **说明：**
> **客户端 ID** 是选填参数，当不填写时，默认值是 "sts.cloud.tencent.com"，本文示例中创建 CAM OIDC 提供商采用默认值。

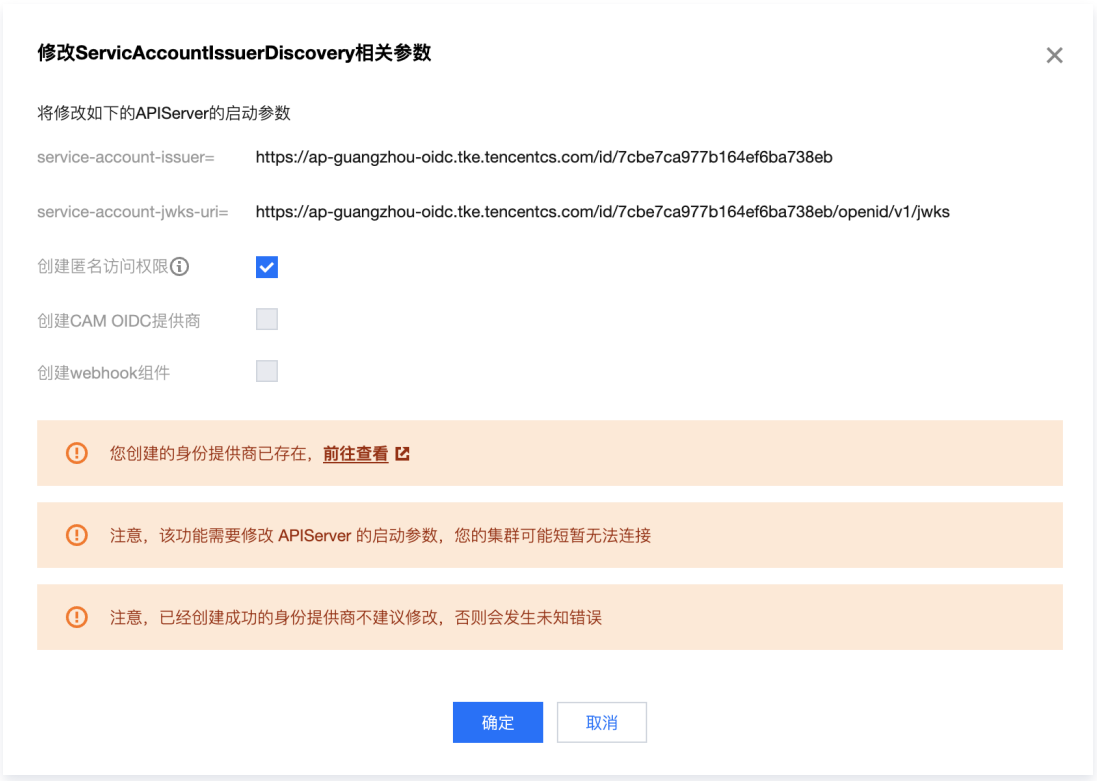


4. 返回集群详情页，当 ServiceAccountIssuerDiscovery 可再次编辑时，表明本次开启 OIDC 资源访问控制结束。

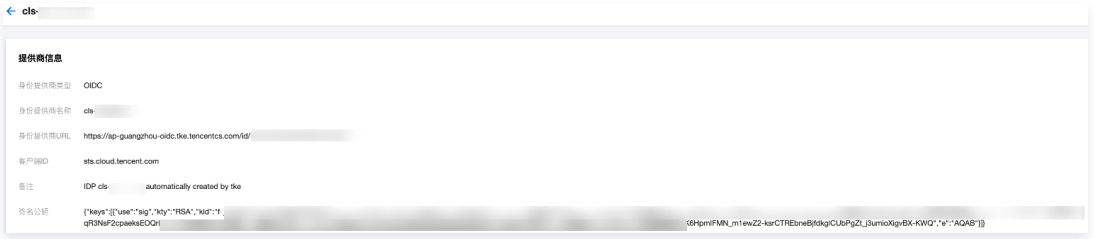
> **注意：**
> "service-account-issuer" 和 "service-account-jwks-uri" 参数值不允许编辑，采用默认规则。

步骤3：检查 CAM OIDC 提供商和 WEBHOOK 组件是否创建成功

1. 在集群详情页中，单击 ServiceAccountIssuerDiscovery 右侧的 。
2. 进入**修改 ServiceAccountIssuerDiscovery 相关参数**页面，系统将提示“您创建的身份提供商已存在，前往查看”。单击**前往查看**。如下图所示：



3. 查看您刚创建的 CAM OIDC 提供商详细信息。如下图所示：



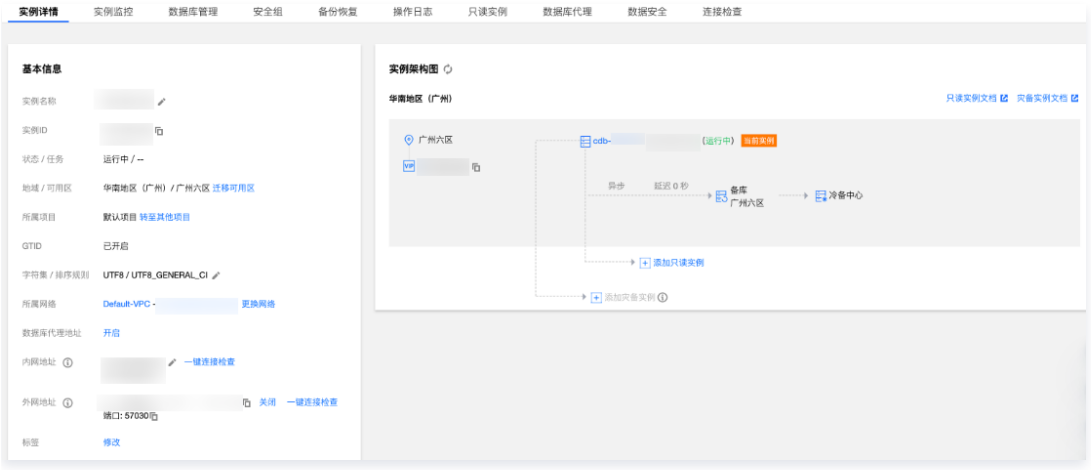
4. 在**集群信息 > 组件管理**中，如在列表看到 pod-identity-webhook 组件状态是“成功”，即表示安装组件成功。如下图所示：

您也可以执行查看命令，以 "pod-identity-webhook" 作为前缀的 Pod 状态是 Running，即表示安装组件成功。

```
kubectl get pod -n kube-system
NAMESPACE     NAME                                    READY   STATUS    RESTARTS   AGE
kube-system   pod-identity-webhook-78c76****-9qrpj    1/1     Running   0          43h
```

步骤4：确认数据库实例

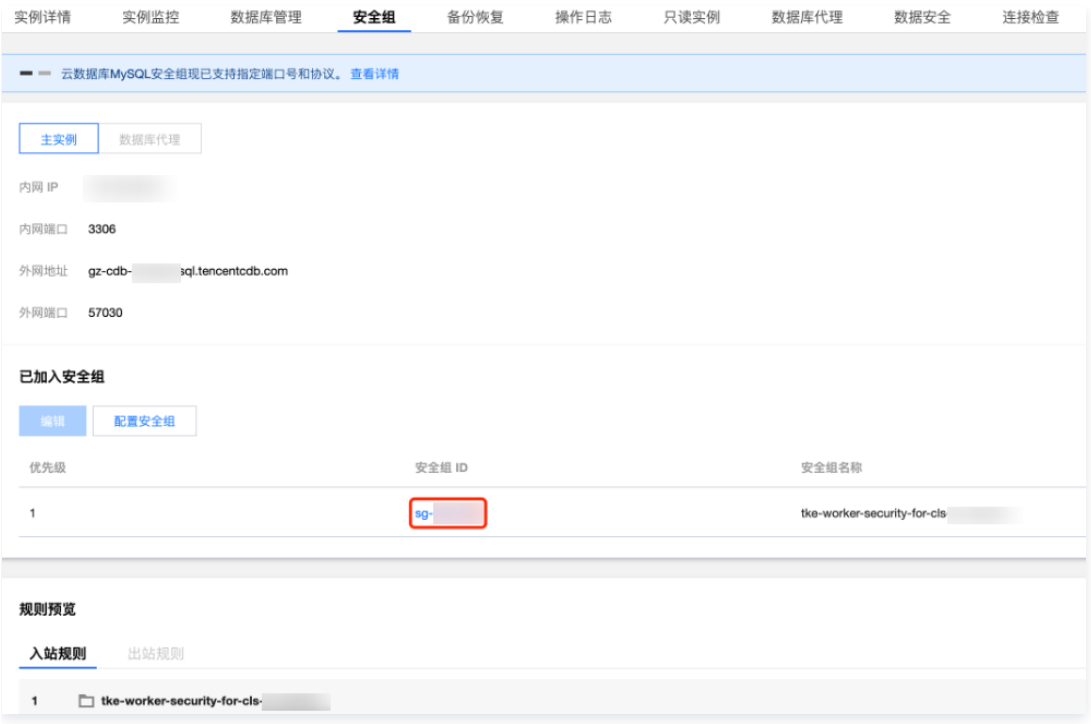
您需要确认是否存在腾讯云数据库实例，若没有腾讯云数据库实例，请您先行创建，并在数据库实例中创建数据库。若已有腾讯云数据库实例，请您跳过数据库创建。
本示例采用腾讯云 MySQL 实例，同时开启 MySQL 实例公网。创建步骤请参考 创建 MySQL 实例 。



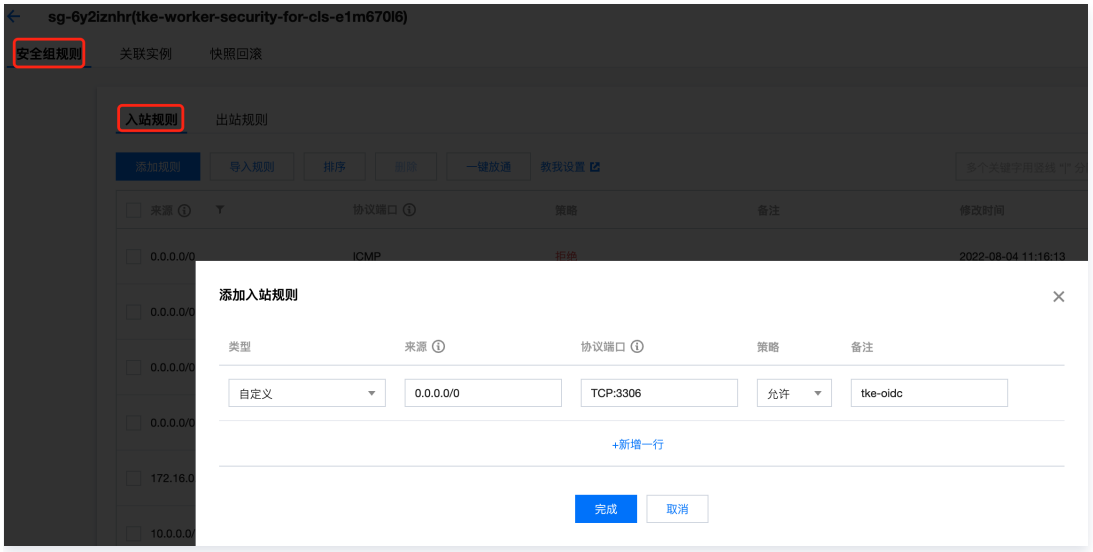
> **注意：**
> - **外网地址**的 value 值标识为 `$db_address` 。
> - **端口**的 value 值标识为 `$db_port` 。

步骤5：更新数据库安全组

托管集群上的 Pod 想要被允许访问腾讯云 MySQL 数据库，需要给腾讯云 MySQL 数据库的安全组添加一些规则。在数据库实例的安全组页面中，修改安全组规则。如下图所示：



为了给 Kubernetes Pod 创建入站规则，您需要单击**安全组 ID**后跳转到安全组实例页面。在安全组实例详情页，选择**安全组规则 > 入站规则 > 添加规则**。在“添加入站规则”弹窗中，进行入站规则的创建。本示例中使用**来源**为 `0.0.0.0/0` ，**协议端口**为 `TCP:3306` 。



步骤6：测试数据库连接性

在安装 MySQL 客户端的实例中，确认您使用用户名 root 和您在创建数据库设置的密码连接到数据库。如果无法连接到数据库，请返回查看是否开启 公网 及是否正确配置 安全组 。

```
mysql -h $db_address  -P $db_port -uroot -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 4238098
Server version: 5.7.36-txsql-log 20211230

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>
```

步骤7：创建数据库、表、数据

为了对数据库连通性和操作权限进行验证，请您先创建个人数据库。

```
MySQL [(none)]> CREATE DATABASE mydb;
Query OK, 1 row affected (0.00 sec)

MySQL [(none)]> CREATE TABLE mydb.user (Id VARCHAR(120), Name VARCHAR(120));
Query OK, 0 rows affected (0.00 sec)

MySQL [(none)]> INSERT INTO mydb.user (Id,Name) VALUES ('123','tke-oidc');
Query OK, 1 row affected (0.01 sec)

MySQL [(none)]> SELECT * FROM mydb.user;
+------+----------+
| Id   | Name     |
+------+----------+
| 123  | tke-oidc |
+------+----------+
1 row in set (0.01 sec)
```

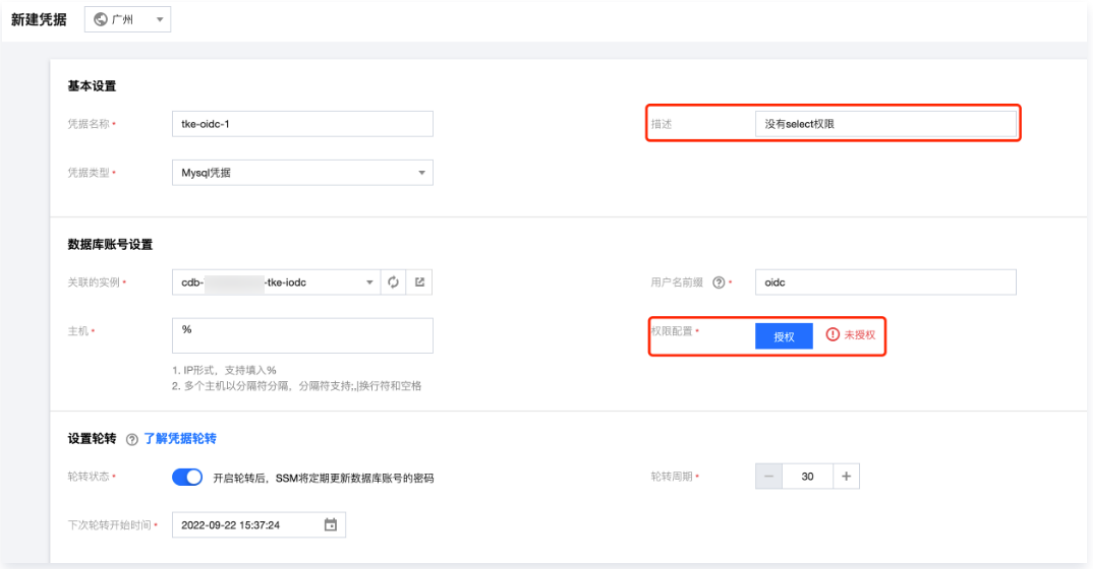
创建完成后，在控制台查看已创建的数据库。如下图所示：

> **注意：**
> **数据库名**的 value 值标识为 `$db_name` 。

步骤8：在凭据管理系统中创建数据库凭据实例

请您检查是否存在数据库凭据。如果不存在数据库凭据，请您在凭据管理系统控制台中创建数据库凭据，开启凭据轮转及选择加密，降低账号的泄露风险与安全威胁。在本文示例中，将创建两个数据库凭证，两者的区别是是否具备对数据库的 select 权限，为了增强可读性，通过**描述** value 值加以区分。

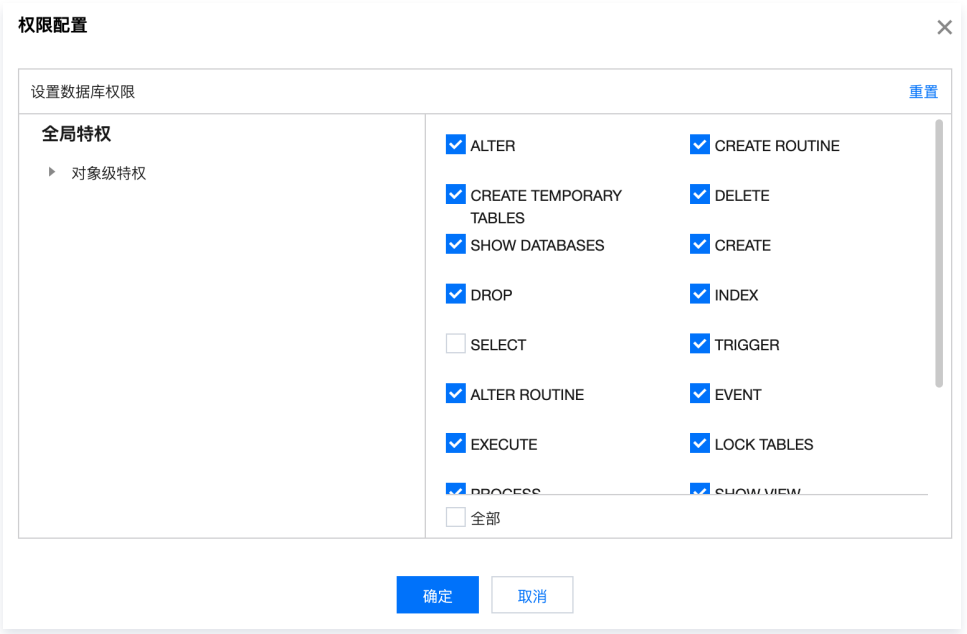
1. 登录 凭据管理系统控制台 。
2. 在**新建凭据**页面，参考如下信息进行数据库账号设置。字段详情可参考 创建数据库凭据 。



- **关联的实例**：选择新建数据库实例或者已存在数据库实例。
- **主机**：是指客户端 IP，不指定时填写 `%` 。
- **权限配置**：根据对数据库实例的操作需求进行授权。

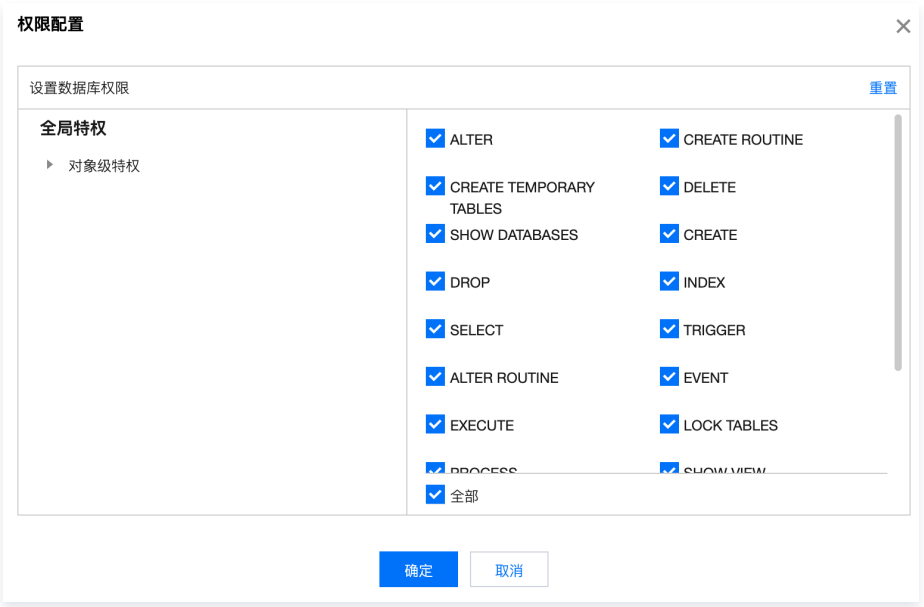
创建第一个数据库凭证

单击授权，在“权限配置”页面，勾选如下权限：



创建第二个数据库凭证

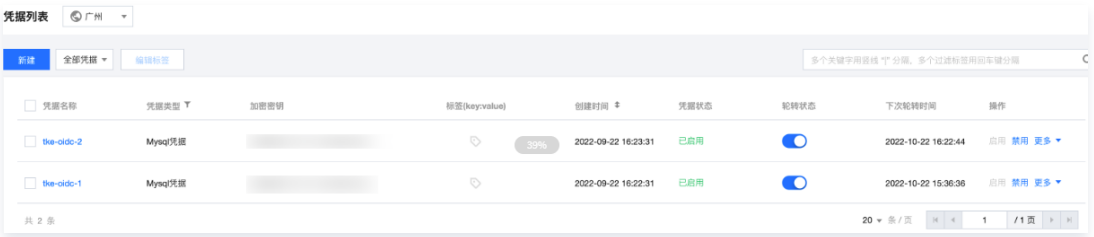
单击授权，在“权限配置”页面，勾选全部权限，如下图所示：



注意：

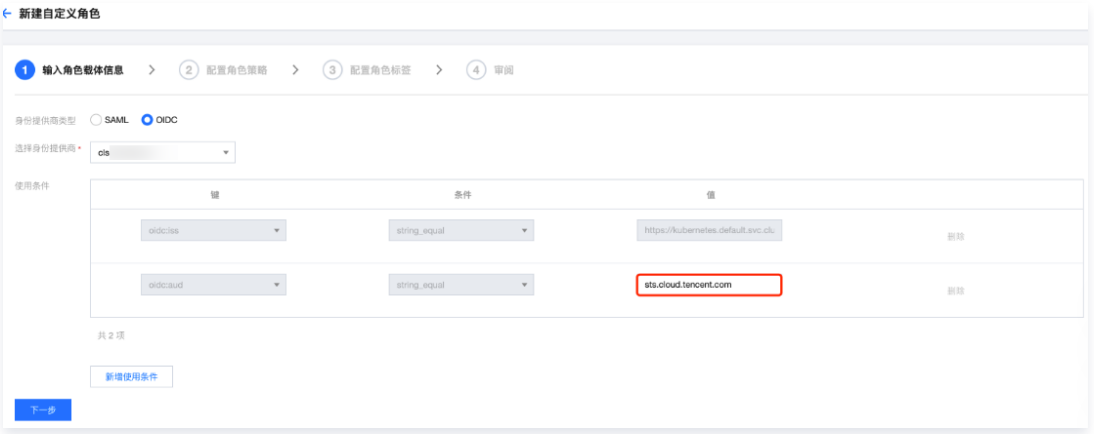
- **凭据名称的 value 值标识为** `$ssm_name`
- **凭据所在地域标识为** `$ssm_region_name`

3. 单击**创建**。在凭据列表页面查看已创建的凭据，如下图所示：

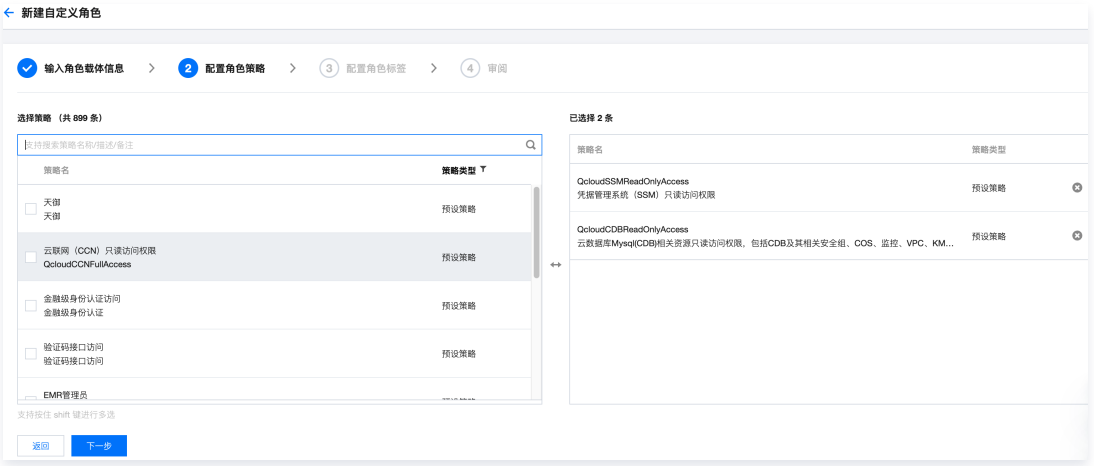


步骤9：创建 CAM 角色并关联访问腾讯云数据库和凭据管理系统的策略

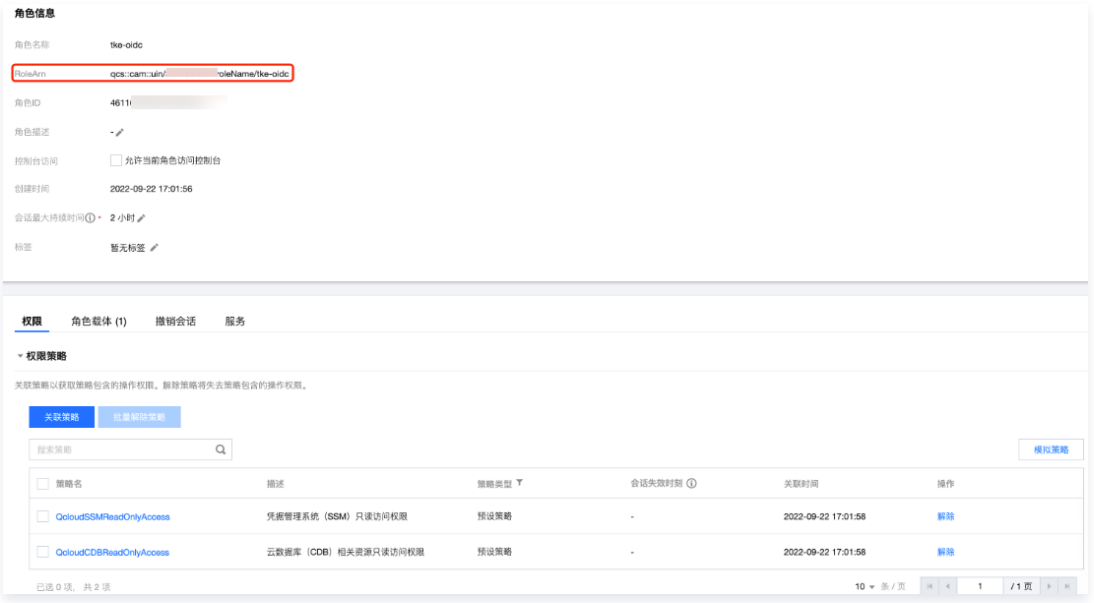
1. 登录 访问管理控制台 。
2. 在角色页中，单击**新建角色 > 身份提供商**。
3. 在**新建自定义角色**页，参考以下信息进行设置。



> **注意：**
> - **oidc:aud** 的 value 值需要和 CAM OIDC 提供商的**客户端 ID** value 值保持一致。
> - **oidc:aud** 的 value 值标识为 `$my_pod_audience` ，当**oidc:aud**的 value 值有多个时，任选其中之一即可。



> **注意：**
> 根据您的业务需求，您可以选择或创建自定义的策略来进行关联。在本示例中，您可以在搜索框中搜索 QcloudSSMReadOnlyAccess 和 QcloudCDBReadOnlyAccess，然后将它们与角色进行关联。



> **注意：**
> RoleArn的 value 值标识为 `$my_pod_role_arn` 。

步骤10：部署示例应用程序

1. 创建一个 Kubernetes 命名空间来部署资源。

```
kubectl create namespace my-namespace
```

2. 将以下内容保存到 **my-serviceaccount.yaml** 中。将 `$my_pod_role_arn` 替换为 RoleArn 的 value 值，将 `$my_pod_audience` 替换为 oidc:aud 的 value 值。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-serviceaccount
  namespace: my-namespace
  annotations:
    tke.cloud.tencent.com/role-arn: $my_pod_role_arn
    tke.cloud.tencent.com/audience: $my_pod_audience
    tke.cloud.tencent.com/token-expiration: "86400"
```

3. 将以下内容保存到**sample-application.yaml**中。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app: my-app
  replicas: 1
  template:
    metadata:
      labels:
        app: my-app
```

```
    spec:
      serviceAccountName: my-serviceaccount
      containers:
        - name: nginx
          image: $image
          ports:
            - containerPort: 80
```

需注意，在本示例中，`$image` 选择 `ccr.ccs.tencentyun.com/tkeimages/sample-application:latest`，该镜像集成了编译的 demo文件，方便进行示例演示。您可以根据自身业务进行填写。

4. 部署示例。

```
kubectl apply -f my-serviceaccount.yaml
kubectl apply -f sample-application.yaml
```

5. 查看使用示例应用程序部署的 Pod。

```
kubectl get pods -n my-namespace
```

示例输出如下：

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6bfd845f47-9zxld   1/1     Running   0          67s
```

6. 查看工作负载环境变量信息。

```
kubectl describe pod nginx-deployment-6bfd845f47-9zxld -n my-namespace
```

示例输出如下：

```
[root@VM-32-127-centos ~]# kubectl describe pod nginx-deployment-6bfd845f47-9zxld -n my-namespace
Name:         nginx-deployment-6bfd845f47-9zxld
Namespace:    my-namespace
Priority:     0
Node:         10.0.32.127/10.0.32.127
Start Time:   Thu, 22 Sep 2022 17:58:55 +0800
Labels:       app=nginx
              pod-template-hash=6bfd845f47
Annotations:  tke.cloud.tencent.com/networks-status:
                [{
                    "name": "tke-bridge",
                    "interface": "eth0",
                    "ips": [
                        "172.24.0.79"
                    ],
                    "mac": "66:16:9c:92:28:08",
                    "default": true,
                    "dns": {}
                }]
Status:       Running
IP:           172.24.0.79
IPs:
  IP:           172.24.0.79
Controlled By:  ReplicaSet/nginx-deployment-6bfd845f47
Containers:
  nginx:
    Container ID:   docker://7b2cfb15cc4fe9b262c24e6fef45191c7628e3765513cb60f68d37332f3afd81
    Image:          ccr.ccs.tencentyun.com/alantlliu/nginx:latest
    Image ID:       docker-pullable://ccr.ccs.tencentyun.com/alantlliu/nginx@sha256:30e1d52127fe952b044d5606952eaa07fb20536cf195620497ad0ade5a100e2c
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Thu, 22 Sep 2022 17:58:57 +0800
    Ready:          True
    Restart Count:  0
    Environment:
      TKE_DEFAULT_REGION:           ap-guangzhou
      TKE_REGION:                   ap-guangzhou
      TKE_PROVIDER_ID:              cls-37hhvfem
      TKE_ROLE_ARN:                 qcs::cam::uin/3321337994:roleName/tke-oidc
      TKE_WEB_IDENTITY_TOKEN_FILE:  /var/run/secrets/cloud.tencent.com/serviceaccount/token
    Mounts:
      /var/run/secrets/cloud.tencent.com/serviceaccount from tke-cam-token (ro)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-7vdtm (ro)
```

步骤11：访问数据库 demo 伪代码实现

1. 确认子账号所有访问 AssumeRoleWithWebIdentity 接口的权限。如果没有权限请联系管理员添加。
2. 确认有访问 AssumeRoleWithWebIdentity 接口的权限后，请参考 凭证管理 中步骤5获取访问 DB + SSM 的临时密钥，详情见 数据库凭据的应用 。
3. 克隆 ssm-rotation-sdk-golang 代码。

```
git clone https://github.com/TencentCloud/ssm-rotation-sdk-golang.git
```

4. 替换 demo 中伪代码实现：

```
package main

import (
    "flag"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "github.com/tencentcloud/ssm-rotation-sdk-golang/lib/db"
    "github.com/tencentcloud/ssm-rotation-sdk-golang/lib/ssm"
    "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/common"
    "log"
    "time"
)

var (
    roleArn, tokenPath, providerId, regionName, saToken string
    secretName, dbAddress, dbName, ssmRegionName        string
    dbPort                                              uint64
    dbConn                                              *db.DynamicSecretRotationDb
```

```
    Header                                                = map[string]string{
        "Authorization":      "SKIP",
        "X-TC-Action":        "AssumeRoleWithWebIdentity",
        "Host":               "sts.internal.tencentcloudapi.com",
        "X-TC-RequestClient": "PHP_SDK",
        "X-TC-Version":       "2018-08-13",
        "X-TC-Region":        regionName,
        "X-TC-Timestamp":     "1659944952",
        "Content-type":       "application/json",
    }
)

type Credentials struct {
    TmpSecretId  string
    TmpSecretKey string
    Token        string
    ExpiredTime  uint64
}

func main() {
    flag.StringVar(&secretName, "ssmName", "", "ssm名称")
    flag.StringVar(&ssmRegionName, "ssmRegionName", "", "ssm地域")
    flag.StringVar(&dbAddress, "dbAddress", "", "数据库地址")
    flag.StringVar(&dbName, "dbName", "", "数据库名称")
    flag.Uint64Var(&dbPort, "dbPort", 0, "数据库端口")
    flag.Parse()

    provider, err := common.DefaultTkeOIDCRoleArnProvider()
    if err != nil {
        log.Fatal("failed to assume role with web identity, err:", err)
    }
    assumeResp, err := provider.GetCredential()
    if err != nil {
        log.Fatal("failed to assume role with web identity, err:", err)
    }

    var credential Credentials
    if assumeResp != nil {
        credential = Credentials{
            TmpSecretId:  assumeResp.GetSecretId(),
            TmpSecretKey: assumeResp.GetSecretKey(),
            Token:        assumeResp.GetToken(),
        }
    }
    log.Printf("secretId:%v,secretey%v,token%v\n", credential.TmpSecretId, credential.TmpSecretKey,
credential.Token)
    DB(credential)
}

func DB(credential Credentials) {
    // 初始化数据库连接
    dbConn = &db.DynamicSecretRotationDb{}
    err := dbConn.Init(&db.Config{
        DbConfig: &db.DbConfig{
            MaxOpenConns:        100,
            MaxIdleConns:        50,
            IdleTimeoutSeconds:  100,
            ReadTimeoutSeconds:  5,
            WriteTimeoutSeconds: 5,
            SecretName:          secretName, // 凭据名
```

```
            IpAddress:           dbAddress,   // 数据库地址
            Port:                dbPort,      // 数据库端口
            DbName:              dbName,      // 可以为空，或指定具体的数据库名
            ParamStr:            "charset=utf8&loc=Local",
        },
        SsmServiceConfig: &ssm.SsmAccount{
            SecretId:  credential.TmpSecretId,  // 需填写实际可用的SecretId
            SecretKey: credential.TmpSecretKey, // 需填写实际可用的SecretKey
            Token:     credential.Token,
            Region:    ssmRegionName, // 选择凭据所存储的地域
        },
        WatchChangeInterval: time.Second * 10, // 多长时间检查一下 凭据是否发生了轮转
    })
    if err != nil {
        fmt.Errorf("failed to init dbConn, err:%v\n", err)
        return
    }
    // 模拟业务处理中，每过一段时间（一般是几毫秒），需要拿到db连接，来操作数据库的场景
    t := time.Tick(time.Second)
    for {
        select {
        case <-t:
            accessDb()
            queryDb()
        }
    }
}

func accessDb() {
    fmt.Println("--- accessDb start")
    c := dbConn.GetConn()
    if err := c.Ping(); err != nil {
        log.Fatal("failed to access db with err:", err)
    }
    log.Println("--- succeed to access db")
}

func queryDb() {
    var (
        id   int
        name string
    )
    log.Println("--- queryDb start")
    c := dbConn.GetConn()
    rows, err := c.Query("select id, name from user where id = ?", 1)
    if err != nil {
        log.Printf("failed to query db with err: ", err)
        log.Fatal(err)
    }
    defer rows.Close()
    for rows.Next() {
        err := rows.Scan(&id, &name)
        if err != nil {
            log.Fatal(err)
        }
        log.Println(id, name)
    }
    err = rows.Err()
    if err != nil {
        log.Fatal(err)
```

```
    }
    log.Println("--- succeed to query db")
}
```

步骤12：测试 demo 示例

基于 部署示例 的部署结果，进入到 nginx 容器：

```
kubectl exec -ti  nginx-deployment-6bfd845f47-9zxld   -n my-namespace  -- /bin/bash
cd /root/
```

将 $ssm_name 和 $ssm_region_name 标识参照 SSM实例 进行替换，将 $db_address、$db_name 和 $db_port 标识参照 数据库实例 进行替换。

```
./demo  --ssmName=$ssm_name --ssmRegionName=$ssm_region_name --dbAddress=$db_address --dbName=$db_name --
dbPort=$db_port
```

在本示例中，当 $ssm_name=tke-oidc-1 时，没有数据库的 select 权限。

在本示例中，当 $ssm_name=tke-oidc-2 时，有数据库的 select 权限。

测试结论

测试表明满足预期的效果。通过 CAM 对托管集群工作负载短暂的身份验证令牌的验证，确保了身份验证的安全性；另外借助凭据管理系统对数据库用户名和密码的轮转和加密特性，使得您不必担心数据库凭据的存储和生命周期问题，这样您在托管集群连接到数据库时无需使用用户名和密码。

pod-identity-webhook 权限说明

权限说明

该组件权限是当前功能实现的最小权限依赖。

权限场景

功能	涉及对象	涉及操作权限
需要查询创建的 pod 上指定的 serviceaccounts 的资源情况。	serviceaccount	list/watch/get
创建组件时需要在 mutatingwebhookconfigurations 的资源注入客户端的证书。	mutatingwebhookconfigurations	get/update

权限定义

```
rules:
  - apiGroups:
      - ""
    resources:
      - serviceaccounts
    verbs:
      - get
      - watch
      - list
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - patch
      - update
  - apiGroups:
      - "admissionregistration.k8s.io"
    resources:
      - "mutatingwebhookconfigurations"
    verbs:
      - get
```

使用 ExternalSecretOperator 导入腾讯云 SSM 凭据

最近更新时间：2024-10-09 15:31:21

ExternalSecretOperator 可以帮助您将统一存储和管理在 腾讯云凭据管理系统（SSM）中的密钥凭据，以 K8S 原生 Secret 对象的形式导入到集群中，并实现密钥数据的自动同步，实现由 SSM 来统一存储和管理密钥的生命周期。

限制条件

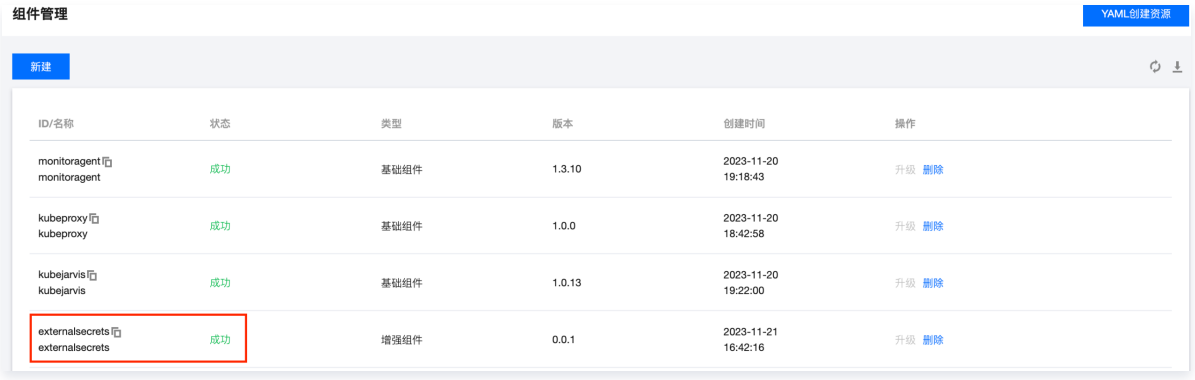
- 使用 ExternalSecrets 组件需要 Kubernetes 版本大于等于1.19。
- 操作系统镜像支持 x86 架构。

启用外部密钥访问能力

安装扩展组件

1. 登录 容器服务控制台 。
2. 为集群安装 ExternalSecrets（外部密钥访问组件）组件。
   - 如果您还没有创建集群，可以在创建集群的时候安装 ExternalSecrets 组件。详情请参见 通过集群创建页安装 。
   - 如果您需要给已创建好的集群开启外部密钥访问能力，请在组件管理中安装 ExternalSecrets 组件。详情请参见 通过组件管理页安装 。

3. 在组件管理页面查看组件状态。如组件状态为“成功”，代表组件部署完成。如下图所示：

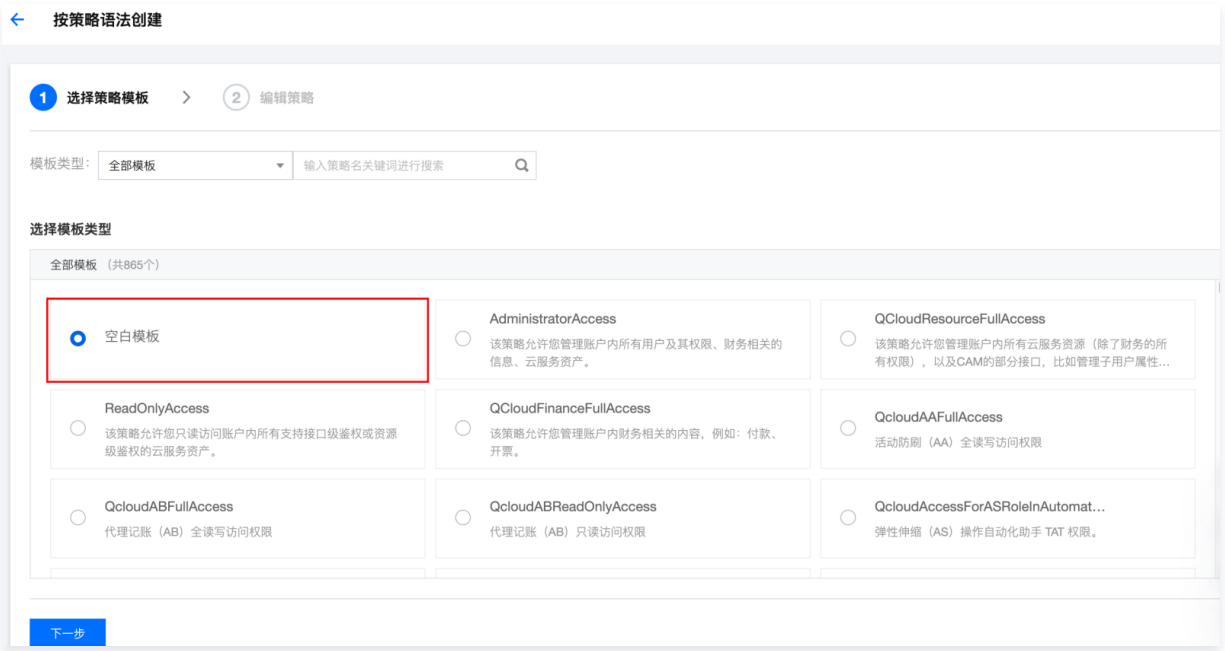


使用方式

方式一：通过 AKSK 授权

步骤1：通过 AKSK 授权方式配置认证信息

1. 登录 腾讯云访问管理控制台，选择左侧导航中的**策略**。
2. 进入**策略**页面，单击**新建自定义策略 > 按策略语法创建**。
3. 在**按策略语法创建**页面，选择**空白模板**，如下图所示：

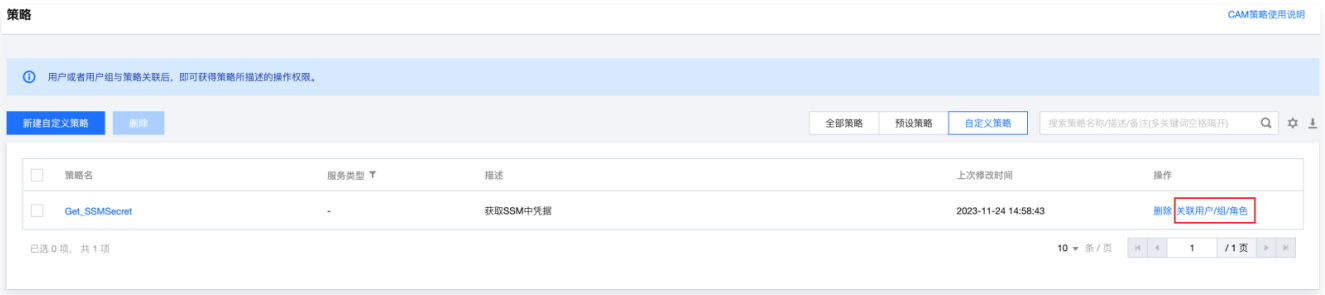


4. 单击**下一步**，进入**编辑策略**页面，在策略内容编辑框中添加以下内容：

```
{
    "statement": [
        {
            "action": [
                "ssm:GetSecretValue"
            ],
            "effect": "allow",
            "resource": [
                "qcs::ssm:$region:uin/$uin:secret/creatorUin/$creatorUin/$secretName"
            ]
        }
    ],
    "version": "2.0"
}
```

> **说明：**
> 授权接口访问特定资源：创建访问控制策略 。

5. 单击**完成**即可添加策略。
6. 在**策略**页面查看已创建的自定义策略，选择**自定义策略 > 关联用户/组/角色**，如下图所示：



在**关联用户/组/角色**页面，选择需要绑定的用户，如下图所示：

7. 单击**确定**。

步骤2：组件使用说明

该组件涉及两种自定义资源（CRD）：SecretStore 用于存放访问凭据，ExternalSecret 用于指定 SecretStore 并存放需要同步的凭据基础信息。通过这种方式，权限和数据得到分离，提高了使用的灵活性。
在 SSM 凭据管理系统中，您需要添加以下凭据：

```
SecretName: hello-test
SecretData: {"name":"jack","password":"123"}
VersionId: v1
```

请参见 腾讯云凭据管理系统文档 以获取详细的创建凭据流程。

> **注意：**
> 以下 secret、SecretStore、ExternalSecret 均在 default 命名空间中。

1. 创建 secret。
   您可以使用以下命令创建 secret：

```
echo -n 'KEYID' > ./accessKeyId
echo -n 'SECRETKEY' > ./accessKeySecret
kubectl create secret generic tencent-credentials --from-file=./accessKeyId --from-
file=./accessKeySecret
```

> **说明：**
> 密钥可前往 访问管理 进行获取。

2. 创建 SecretStore。
   您可以将以下内容保存到 my-secretstore.yaml 文件中：

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
```

```
  name: my-secretstore
spec:
  provider:
    tencent:
      regionID: ap-guangzhou
      auth:
        secretRef:
          accessKeyIDSecretRef:
            name: tencent-credentials
            key: accessKeyId
          accessKeySecretSecretRef:
            name: tencent-credentials
            key: accessKeySecret
```

3. 创建 ExternalSecret。

   您可以将以下内容保存到 my-externalsecret.yaml 文件中：

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: my-externalsecret
spec:
  refreshInterval: 1m
  secretStoreRef:
    kind: SecretStore
    name: my-secretstore
  target:
    name: my-secret-key-to-be-created
    creationPolicy: Owner
  data:
    - secretKey: secret-key-to-be-managed
      remoteRef:
        key: hello-test
        version: v1
        # option
        property: password
```

4. 部署示例，请执行以下命令：

```
kubectl apply -f my-secretstore.yaml
kubectl apply -f my-externalsecret.yaml
```

5. 使用获取的凭据。

   您可以将以下内容保存到 my-pod.yaml 文件中：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: busybox
      command:
        - /bin/sh
        - -c
        - 'echo "Secret value: ${SECRET_KEY_TO_BE_MANAGED}"'
      env:
```

```
        - name: SECRET_KEY_TO_BE_MANAGED
          valueFrom:
            secretKeyRef:
              name: my-secret-key-to-be-created
              key: secret-key-to-be-managed
  restartPolicy: Never
```

然后，使用以下命令部署 Pod 资源：

```
kubectl apply -f my-pod.yaml
```

最后，使用以下命令查看获取的凭据：

```
kubectl logs my-pod
```

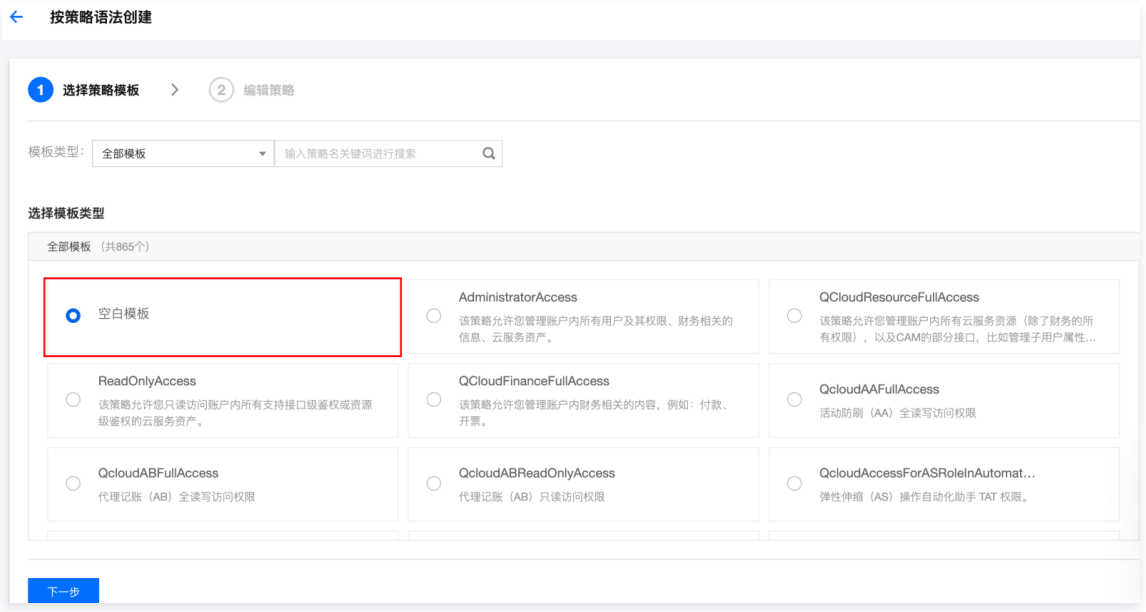
您将看到获取的凭据信息：

```
# ExternalSecret中获取的凭据信息如下所示。
Secret value: 123
```

方式二：通过 AKSK 与角色扮演授权

步骤1：创建获取 SSM 凭据的策略

1. 登录 腾讯云访问管理控制台，选择左侧导航中的**策略**。
2. 进入**策略**页面，单击**新建自定义策略 > 按策略语法创建**。
3. 在**按策略语法创建**页面，选择**空白模板**，如下图所示：



4. 单击**下一步**，进入**编辑策略**页面，在策略内容编辑框中添加以下内容：

```
{
    "statement": [
        {
            "action": [
                "ssm:GetSecretValue"
            ],
            "effect": "allow",
            "resource": [
```

```
                "qcs::ssm:$region:uin/$uin:secret/creatorUin/$creatorUin/$secretName"
            ]
        }
    ],
    "version": "2.0"
}
```

说明：
授权接口访问特定资源：创建访问控制策略 。

5. 单击**完成**即可添加策略。

步骤2：为子账号赋予扮演角色策略

1. 登录 腾讯云访问管理控制台 ，选择左侧导航中的**用户 > 用户列表**。
2. 在**用户列表**页面，单击**新建用户**。新建用户流程详情请参见 新建子用户 。
3. 为创建的子用户赋予扮演角色的策略。详情请参见 为子账号赋予扮演角色策略 。

步骤3：为角色赋予访问 SSM 凭据的策略

1. 在**策略**页面查看已创建的自定义策略，选择**自定义策略 > 关联用户/组/角色**，如下图所示：

2. 在**关联用户/组/角色**页面，选择需要绑定的角色，如下图所示：

3. 单击**确定**。

步骤4：组件使用说明

该组件涉及两种自定义资源（CRD）：SecretStore 用于存放访问凭据，ExternalSecret 用于指定 SecretStore 并存放需要同步的凭据基础信息。通过这种方式，权限和数据得到分离，提高了使用的灵活性。
在 SSM 凭据管理系统中，您需要添加以下凭据：

```
SecretName: hello-test
SecretData: {"name":"jack","password":"123"}
```

```
VersionId: v1
```

请参见 腾讯云凭据管理系统文档 以获取详细的创建凭据流程。

> ⚠ **注意：**
> 以下 secret、SecretStore、ExternalSecret 均在 default 命名空间中。

1. 创建 secret。

   您可以使用以下命令创建 secret：

```
echo -n 'KEYID' > ./accessKeyId
echo -n 'SECRETKEY' > ./accessKeySecret
kubectl create secret generic tencent-credentials --from-file=./accessKeyId --from-
file=./accessKeySecret
```

   > ⓘ **说明：**
   > 密钥可前往 访问管理 进行获取。

2. 创建 SecretStore。

   您可以将以下内容保存到 my-secretstore.yaml 文件中：

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: secretstore-assumerole
spec:
  provider:
    tencent:
      regionID: ap-guangzhou
      role: "qcs::cam::uin/12345:roleName/test-assume-role"
      auth:
        secretRef:
          accessKeyIDSecretRef:
            name: tencent-credentials
            key: accessKeyId
          accessKeySecretSecretRef:
            name: tencent-credentials
            key: accessKeySecret
```

   > ⓘ **说明：**
   > role 字段在 步骤2 中获取。

3. 创建 ExternalSecret。

   您可以将以下内容保存到 my-externalsecret.yaml 文件中：

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: external-secret-assumerole
spec:
  refreshInterval: 1m
  secretStoreRef:
    kind: SecretStore
    name: secretstore-assumerole
  target:
```

```
    name: my-secret-key-to-be-created
    creationPolicy: Owner
  data:
    - secretKey: secret-key-to-be-managed
      remoteRef:
        key: hello-test
        version: v1
        property: password
```

4. 部署示例，请执行以下命令：

```
kubectl apply -f my-secretstore.yaml
kubectl apply -f my-externalsecret.yaml
```

5. 使用获取的凭据。

   您可以将以下内容保存到 my-pod.yaml 中：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: busybox
      command:
        - /bin/sh
        - -c
        - 'echo "Secret value: ${SECRET_KEY_TO_BE_MANAGED}"'
      env:
        - name: SECRET_KEY_TO_BE_MANAGED
          valueFrom:
            secretKeyRef:
              name: my-secret-key-to-be-created
              key: secret-key-to-be-managed
  restartPolicy: Never
```

   然后，使用以下命令部署 Pod 资源：

```
kubectl apply -f my-pod.yaml
```

   最后，使用以下命令查看获取的凭据：

```
kubectl logs my-pod
```

   您将看到获取的凭据信息：

```
# ExternalSecret中获取的凭据信息如下所示。
Secret value: 123
```

方式三：通过 TKE OIDC 授权

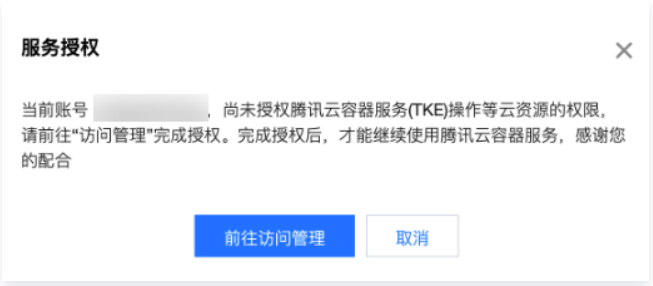
步骤1：开启 OIDC 资源访问控制能力

1. 登录 容器服务控制台，选择左侧导航中的集群。

2. 在**集群管理**页面，选择集群 ID，进入集群的基本信息页面。
3. 在集群基本信息中，单击 ServiceAccountIssuerDiscovery 右侧的 。如下图所示：

> **说明：**
> 如果您需要体验 ServiceAccountIssuerDiscovery 功能，请 提交工单 进行申请。

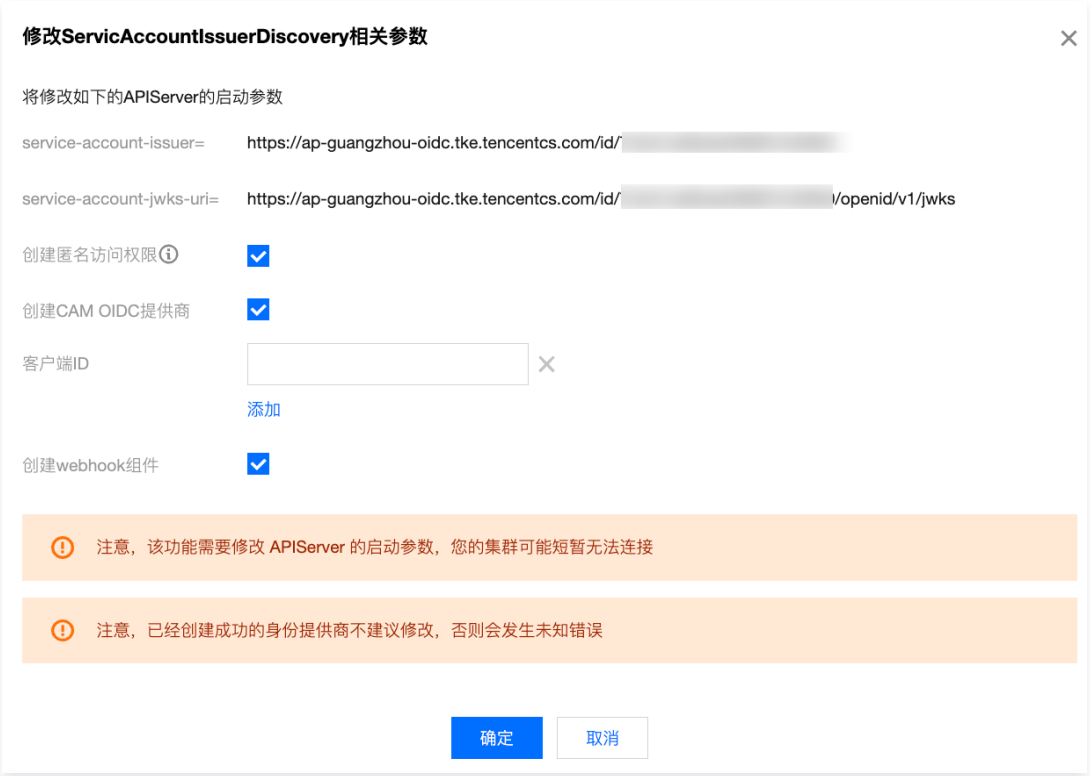
4. 进入**修改 ServiceAccountIssuerDiscovery 相关参数**页面，若系统提示您无法修改相关参数，请先进行服务授权。



在角色管理页面，查看授权策略 QcloudAccessForTKERoleInOIDCConfig，单击**同意授权**。

5. 授权完毕后，勾选“创建 CAM OIDC 提供商”和“创建webhook组件”，并填写客户端 ID，单击**确定**。如下图所示：

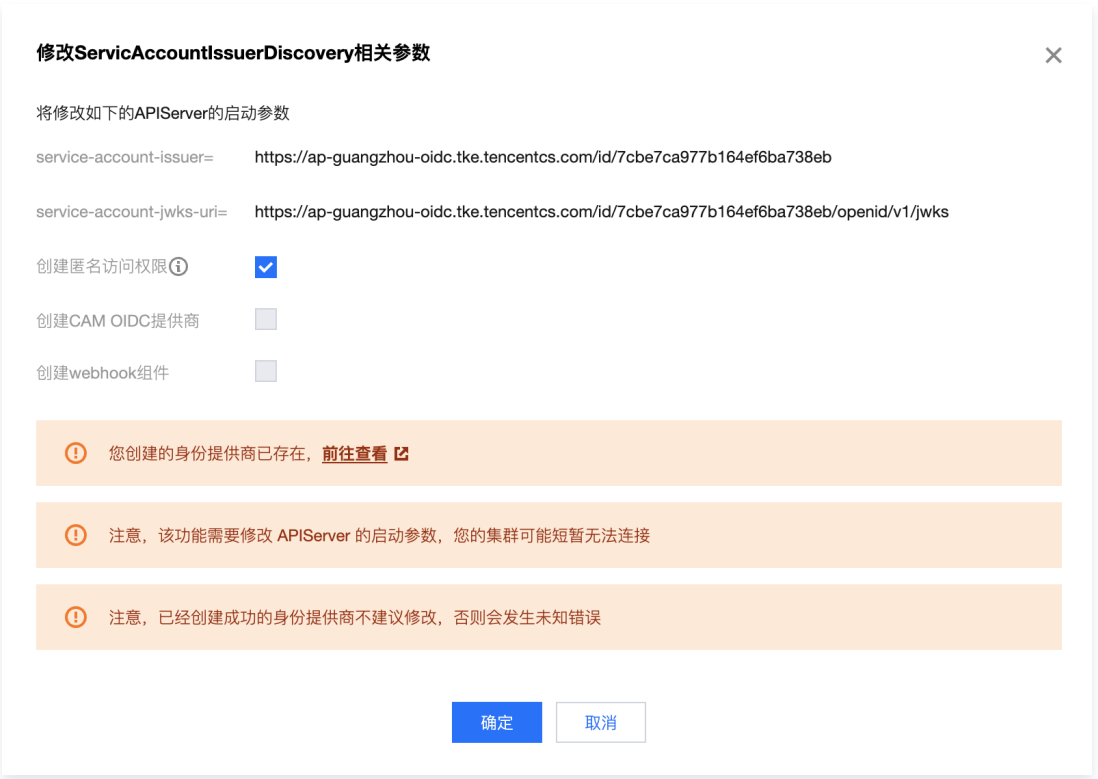
> **说明**
> **客户端 ID** 是选填参数，当不填写时，默认值是 "sts.cloud.tencent.com"，本文示例中创建 CAM OIDC 提供商采用默认值。



6. 返回集群详情页，当 ServiceAccountIssuerDiscovery 可再次编辑时，表明本次开启 OIDC 资源访问控制结束。

> **注意**
> "service-account-issuer" 和 "service-account-jwks-uri" 参数值不允许编辑，采用默认规则。

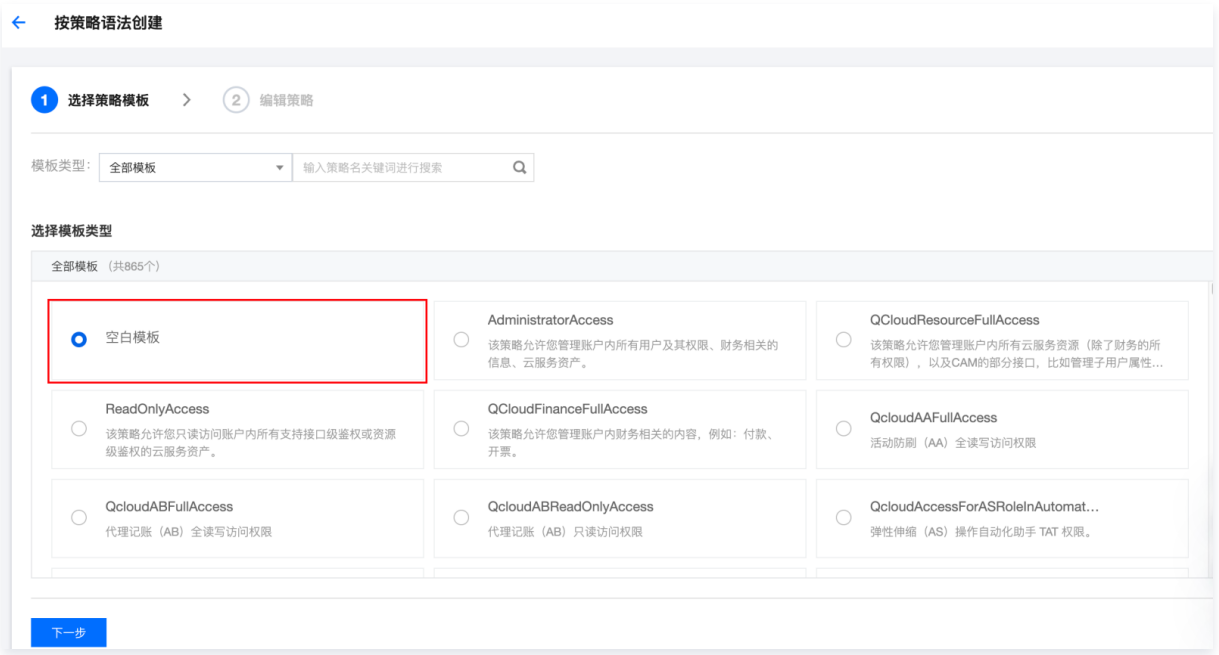
7. 进入**修改 ServiceAccountIssuerDiscovery 相关参数**页面，系统将提示“您创建的身份提供商已存在，前往查看”。单击**前往查看**。如下图所示：



8. 在**集群信息 > 组件管理**中，如在列表看到 pod-identity-webhook 组件状态是“成功”，即表示安装组件成功。如下图所示：

步骤2：创建获取 SSM 凭据的策略

1. 登录 腾讯云访问管理控制台，选择左侧导航中的**策略**。
2. 进入**策略**页面，单击**新建自定义策略 > 按策略语法创建**。
3. 在**按策略语法创建**页面，选择“空白模板”，如下图所示：



4. 单击**下一步**，进入**编辑策略**页面，在策略内容编辑框中添加以下内容。创建访问控制策略，详情请参见 创建访问控制策略 。

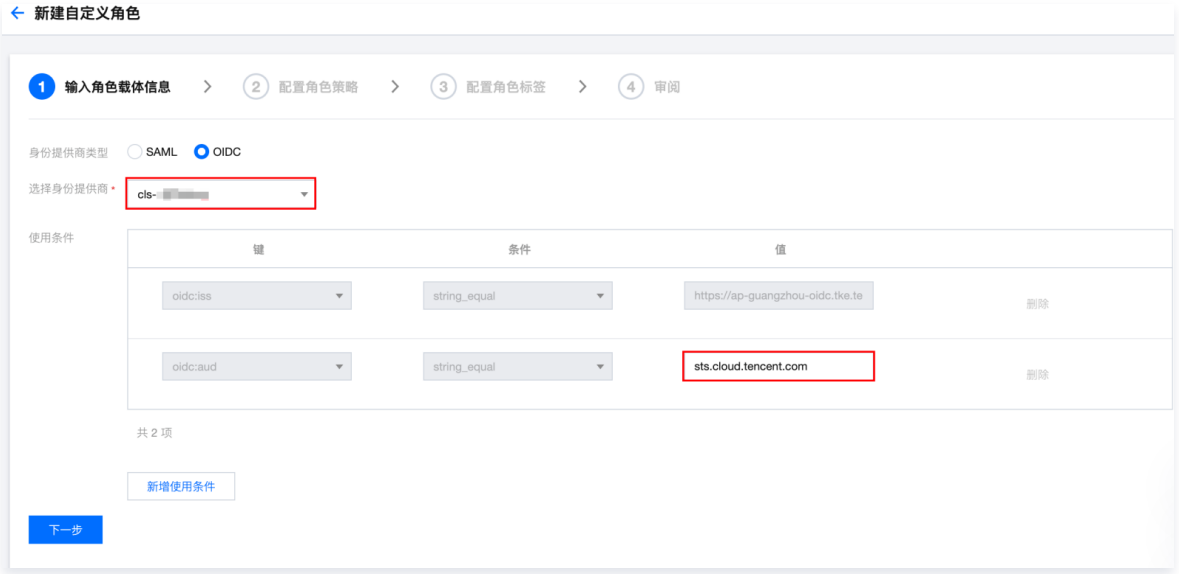
```
{
    "statement": [
        {
            "action": [
                "ssm:GetSecretValue"
            ],
            "effect": "allow",
            "resource": [
                "qcs::ssm:$region:uin/$uin:secret/creatorUin/$creatorUin/$secretName"
            ]
        }
    ],
    "version": "2.0"
}
```

> **说明：**
> 授权接口访问特定资源： 创建访问控制策略 。

5. 单击**完成**即可添加策略。

步骤3：新建 OIDC 角色

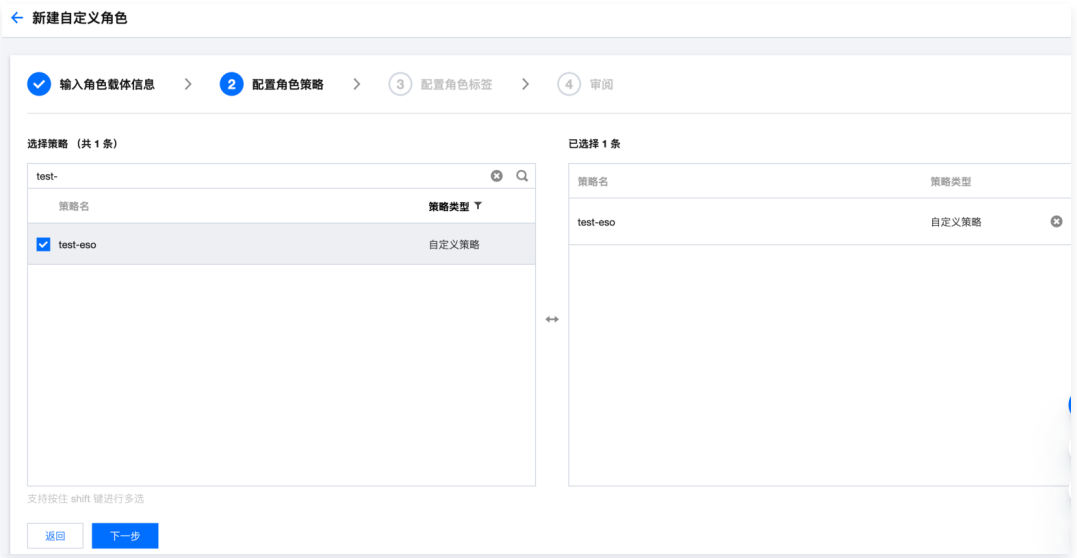
1. 登录 腾讯云访问管理控制台 ，选择左侧导航中的**角色**。
2. 在**角色**页面，选择**新建角色 > 身份提供商**。
3. 在**新建自定义角色**页面，参考以下信息进行设置。



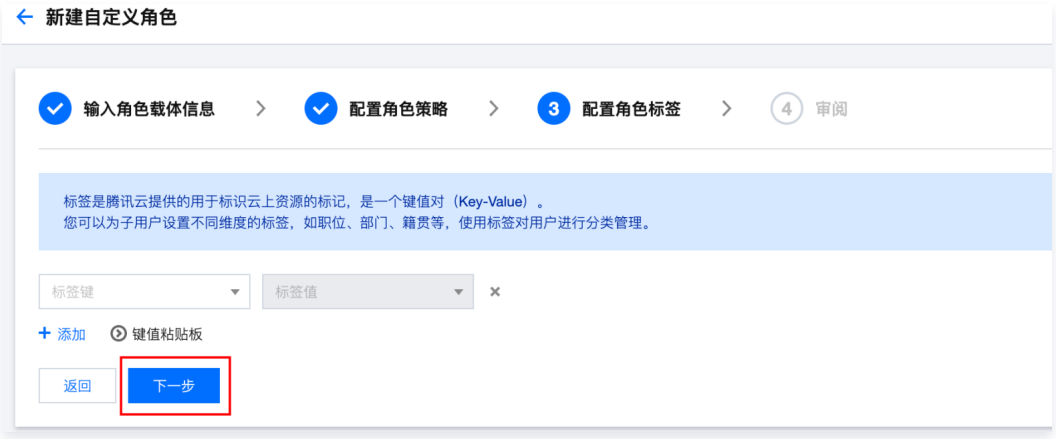
- **身份提供商类型**：选择 OIDC。
- **选择身份提供商**：选择本次为哪个身份提供商创建角色。
- **使用条件**：填写 oidc:aud 的 value 值。

> **注意：**
> - **身份提供商**的 value 值标识为 $my_provider_id。
> - **oidc:aud** 的 value 值需要和 CAM OIDC 提供商的**客户端 ID** value 值保持一致。
> - **oidc:aud** 的 value 值标识为 `$my_pod_audience` ，当**oidc:aud**的 value 值有多个时，任选其中之一即可。

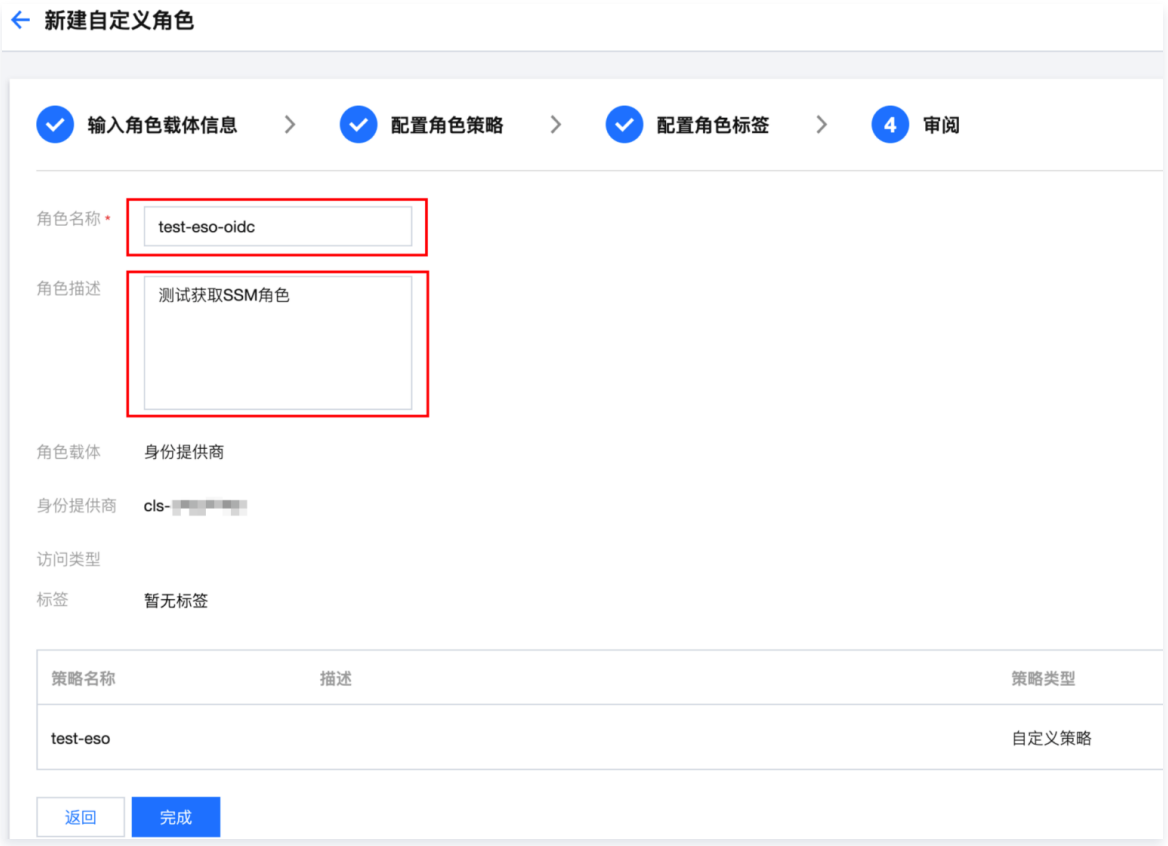
4. 单击**下一步**，进入**配置角色策略**页面，选择在 步骤2 中创建并获取 SSM 的策略，如下图所示：



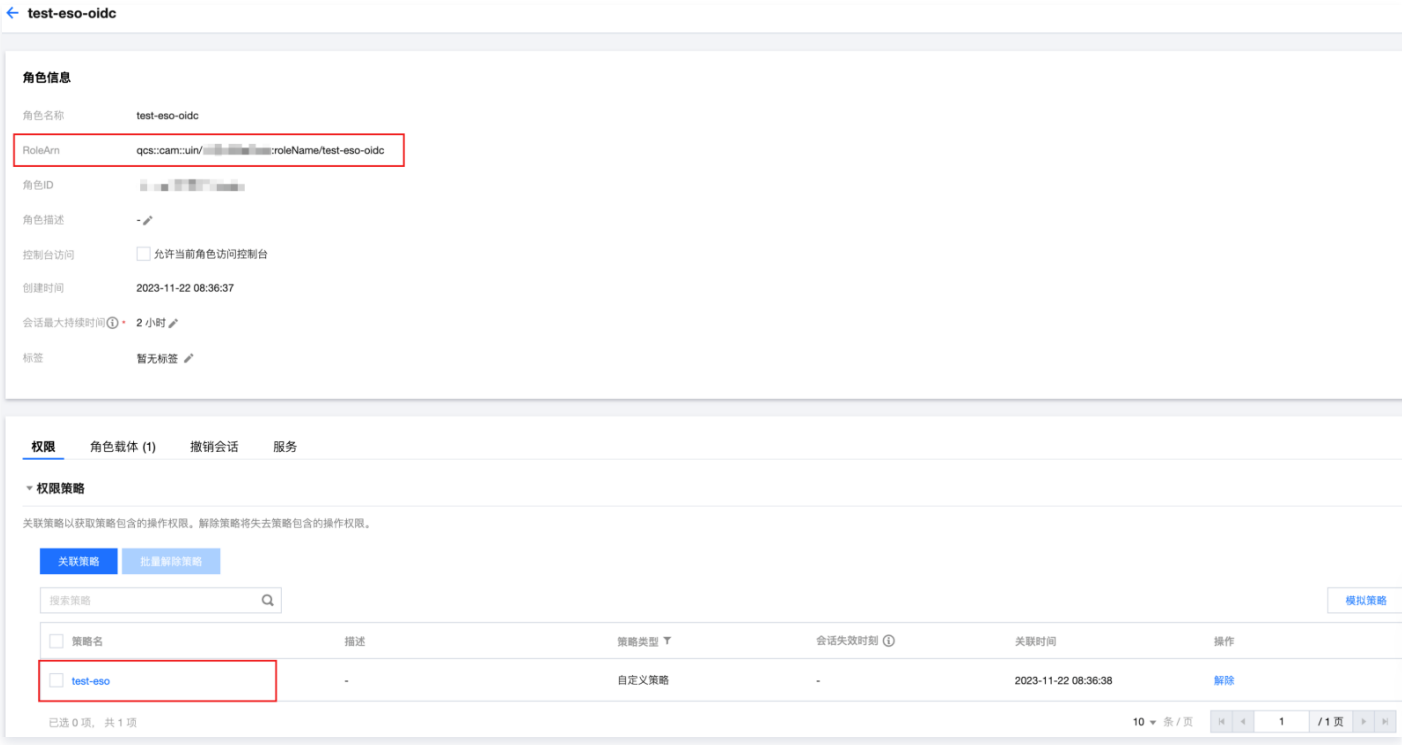
5. 单击**下一步**，进入**配置角色标签**页面，若不需要设置标签可直接下一步，如下图所示：



6. 单击**下一步**，进入**审阅**页面，编辑**角色名称**、**角色描述**，如下图所示：



7. 单击**完成**。角色创建完成后，进入角色详情页，可以查看 OIDC 角色的 RoleArn 与该角色拥有的权限，如下图所示：



> ⚠ **注意：**
> **RoleArn 的 value 值标识为** `$my_pod_role_arn` **。**

步骤4：组件使用说明

该组件涉及两种自定义资源（CRD）：SecretStore 用于存放访问凭据，ExternalSecret 用于指定 SecretStore 并存放需要同步的凭据基础信息。通过这种方式，权限和数据得到分离，提高了使用的灵活性。
在 SSM 凭据管理系统中，您需要添加以下凭据：

```
SecretName: hello-test
SecretData: {"name":"jack","password":"123"}
VersionId: v1
```

请参见 腾讯云凭据管理系统文档 以获取详细的创建凭据流程。

> ⚠ **注意：**
> 以下 ServiceAccount、SecretStore、ExternalSecret 均在 default 命名空间中。

1. 创建 ServiceAccount。
   您可以将以下内容保存到 my-serviceaccount.yaml 中：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-serviceaccount
  annotations:
    tke.cloud.tencent.com/role-arn: $my_pod_role_arn
    tke.cloud.tencent.com/audience: $my_pod_audience
    tke.cloud.tencent.com/providerID: $my_provider_id
```

> **说明：**
> - 将 $my_pod_role_arn 替换为 RoleArn 的 value 值。
> - 将 $my_pod_audience 替换为 oidc:aud 的 value 值。
> - 将 $my_provider_id 替换为“身份提供商”。

2. 创建 SecretStore。
   您可以将以下内容保存到 my-secretstore.yaml 中：

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: secretstore-tkeoidc
spec:
  provider:
    tencent:
      regionID: ap-guangzhou
      auth:
        serviceAccountRef:
          name: my-serviceaccount
```

3. 创建 ExternalSecret。
   您可以将以下内容保存到 my-externalsecret.yaml 中：

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: external-secret-tkeoidc
spec:
  refreshInterval: 1h
  secretStoreRef:
    kind: SecretStore
    name: secretstore-tkeoidc
  target:
    name: my-secret-key-to-be-created
    creationPolicy: Owner
  data:
    - secretKey: secret-key-to-be-managed
      remoteRef:
        key: hello-test
        version: v1
        # option
        property: password
```

4. 部署示例，请执行以下命令：

```
kubectl apply -f my-serviceaccount.yaml
kubectl apply -f my-secretstore.yaml
kubectl apply -f my-externalsecret.yaml
```

5. 查看目标 Secret 是否创建成功，请执行以下命令：

```
kubectl get secret my-secret-key-to-be-created -o yaml
```

> **注意：**

在没有关闭同步刷新的前提下，可以修改 SSM 凭据管理系统中的密钥内容，等到刷新时间到达后，目标 secret 会完成同步。

服务部署

合理利用节点资源

概述

最近更新时间：2024-10-31 19:14:52

将已容器化的业务部署至 Kubernetes 的过程并不复杂，但若业务用于正式生产环境，则需结合业务场景和部署环境进行方案选型及配置调优。例如，设置容器的 Request 与 Limit、使部署的服务达到高可用、配置健康检查、弹性伸缩、更好地进行资源调度、选择持久化存储、对外暴露服务等。
您可参考以下文档，结合实际情况进行 Kubernetes 服务部署与配置调优：

- 设置 Request 与 Limit
- 资源合理分配
- 弹性伸缩

设置 Request 与 Limit

最近更新时间：2023-09-08 19:13:06

容器的 request 及 limit 需根据服务类型、需求及场景进行灵活设置。本文结合实际生产经验进行分析总结，您可参考下文并进行相应的配置调整。

Request 工作原理

Request 的值并不代表给容器实际分配的资源大小，而是用于提供给调度器。调度器会检测每个节点可用于分配的资源（节点可分配资源 = 节点资源总额 – 已调度到节点上的 Pod 内容器 request 之和），同时记录每个节点已经被分配的资源（节点上所有 Pod 中定义的容器 request 之和）。如发现节点剩余的可分配资源已小于当前需被调度的 Pod 的 request，则该 Pod 就不会被调度到此节点。反之，则会被调度到此节点。
若不配置 request，调度器就无法感知节点资源使用情况，无法做出合理的调度决策，可能会造成调度不合理，引起节点状态混乱。建议给所有容器设置 request，使调度器可感知节点资源情况，以便做出合理的调度决策。集群的节点资源能够被合理的分配使用，避免因资源分配不均而导致发生故障。

设置 request 与 limit 默认值

可使用 LimitRange 来设置 namespace 的 request 与 limit 默认值，也可设定 request 与 limit 的最大值与最小值。示例如下：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
  namespace: test
spec:
  limits:
  - default:
      memory: 512Mi
      cpu: 500m
    defaultRequest:
      memory: 256Mi
      cpu: 100m
    type: Container
```

重要线上应用配置

节点资源不足时，会触发自动驱逐，删除低优先级的 Pod 以释放资源使节点自愈。Pod 优先级**由低到高**排序如下：

1. 未设置 request 及 limit 的 Pod。
2. 设置 request 值不等于 limit 值的 Pod。
3. 设置 request 值等于 limit 值的 Pod。

建议重要线上应用设置 request 值等于 limit 值，此类 Pod 优先级较高，在节点故障时不易被驱逐导致线上业务受到影响。

提高资源利用率

如应用设置了较高的 request 值，而实际占用资源远小于设定值，会导致节点整体的资源利用率较低。除对时延非常敏感的业务外，敏感的业务本身并不期望节点利用率过高，影响网络包收发速度。
建议对非核心，并且资源非长期占用的应用，适当减少 request 以提高资源利用率。若您的服务支持水平扩容，则除 CPU 密集型应用外，单副本的 request 值通常可设置为不大于1核。例如，coredns 设置为0.1核，即100m即可。

避免 request 与 limit 值过大

若您的服务使用单副本或少量副本，且 request 及 limit 的值设置过大，使服务可分配到足够多的资源去支撑业务。则某个副本发生故障时，可能会给业务带来较大影响。当 Pod 所在节点发生故障时，由于 request 值过大，且集群内资源分配的较为碎片化，其余节点无足够可分配资源满足该 Pod 的 request，则该 Pod 无法实现漂移，无法自愈，会加重对业务的影响。
建议尽量减小 request 及 limit，通过增加副本的方式对您的服务支撑能力进行水平扩容，使系统更加灵活可靠。

避免测试 namespace 消耗过多资源

若生产集群有用于测试的 namespace，如不加以限制，则可能导致集群负载过高，影响生产业务。可以使用 `ResourceQuota` 限制测试 namespace 的 request 与 limit 的总大小。示例如下：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-test
  namespace: test
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

资源合理分配

最近更新时间：2024-10-31 19:14:52

设置 request 能够使 Pod 调度到有足够资源的节点上，但无法做到更细致的控制。本文介绍通过亲和性、污点与容忍，使 Pod 能够被调度到合适的节点上，让资源得到充分的利用。

使用亲和性

- 对节点有特殊要求的服务可使用节点亲和性（Node Affinity）部署，以便调度到符合要求的节点。例如，让 MySQL 调度到高 IO 的机型以提升数据读写效率。
- 需进行关联的服务可使用节点亲和性（Node Affinity）部署。例如，让 Web 服务与其 Redis 缓存服务都部署在同一可用区，可实现低延时。
- 可使用节点亲和性（Node Affinity）将 Pod 进行打散调度，避免单点故障或流量过于集中导致的一些问题。

使用污点与容忍

使用污点（Taint）与容忍（Toleration）可优化集群资源调度：

- 通过给节点打污点，来给某些应用预留资源，避免其他 Pod 调度到此节点。
- 需使用预留资源的 Pod 加上容忍，结合节点亲和性使 Pod 调度到预留节点，即可使用预留资源。

弹性伸缩

最近更新时间：2023-09-08 19:13:06

本文结合实际生产经验介绍如何在业务中结合弹性伸缩使资源得到充分利用，您可参考下文并进行相应的配置调整。

应对流量突发型业务

通常业务会有高峰和低谷，为了更合理的利用资源，可为服务定义 HPA，实现根据 Pod 的资源实际使用情况来对服务进行自动扩缩容。在业务高峰期时自动扩容 Pod 数量来支撑服务，在业务低谷时自动缩容 Pod 释放资源，以供其他服务使用。例如，夜间线上业务低峰，自动缩容释放资源以供大数据类的离线任务运行。
使用 HPA 前，需安装 resource metrics（metrics.k8s.io）或 custom metrics（custom.metrics.k8s.io），使 hpa controller 通过查询相关 API 获取到服务资源的占用情况，即 K8S 先获取服务的实际资源占用情况（指标数据）。早期 HPA 使用 resource metrics 获取指标数据，后推出的 custom metrics 可通过更灵活的指标来控制扩缩容。Kubernetes 官方相关实现为 metrics-server，而社区通常使用基于 prometheus 的 实现 prometheus-adapter，云厂商托管的 Kubernetes 集群通常集成自身的实现。例如容器服务，实现了 CPU、内存、硬盘、网络等维度的指标，可在网页端可视化创建 HPA，最终转化为 Kubernetes 的 yaml。示例如下：

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta2
    kind: Deployment
    name: nginx
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Pods
    pods:
      metric:
        name: k8s_pod_rate_cpu_core_used_request
      target:
        averageValue: "100"
        type: AverageValue
```

节约成本

HPA 能够实现 Pod 水平扩缩容，但如果节点资源不足，则扩容出的 Pod 状态仍会 Pending。如果提前准备好大量节点，使资源冗余，即使不会发生 Pod Pending 问题，但成本可能过高。通常云厂商托管的 Kubernetes 集群均会实现 cluster-autoscaler，即根据资源使用情况，动态增删节点，使计算资源能够被最大化弹性使用，并通过按量计费的计费模式节约成本。例如，容器服务中的伸缩组，及包含伸缩组功能的拓展特性（节点池）。

使用垂直伸缩

对于无法适配水平伸缩的单体应用，或不确定最佳 request 与 limit 超卖比的应用，可尝试使用 VPA 来进行垂直伸缩。即自动更新 request 与 limit，并重启 pod。该特性容易导致服务出现短暂的不可用，不建议在生产环境中大规模使用。

应用高可用部署

最近更新时间：2024-12-05 21:17:32

高可用性（High Availability，HA）是指应用系统无中断运行的能力，通常可通过提高该系统的容错能力来实现。一般情况下，通过设置 `replicas` 给应用创建多个副本，可以适当提高应用容错能力，但这并不意味着应用就此实现高可用性。本文为部署应用高可用的实践教程，通过以下方式实现高可用性。您可结合实际情况，选择多种方式进行部署：

- 将业务工作负载打散调度
- 使用置放群组从物理层面实现容灾
- 使用 PodDisruptionBudget 避免驱逐导致服务不可用
- 使用 preStopHook 和 readinessProbe 保证服务平滑更新不中断

将业务工作负载打散调度

1. 使用反亲和性避免单点故障

Kubernetes 的设计理念为假设节点不可靠，节点越多，发生软硬件故障导致节点不可用的几率就越高。因此，我们通常需要为应用部署多个副本，并根据实际情况调整 `replicas` 的值。该值如果为1，就必然存在单点故障。该值如果大于1但所有副本都调度到同一个节点，仍将无法避免单点故障。
为了避免单点故障，我们需要有合理的副本数量，还需要让不同副本调度到不同的节点。可以利用反亲和性来实现，示例如下：

```
affinity:
 podAntiAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   - weight: 100
     labelSelector:
       matchExpressions:
       - key: k8s-app
         operator: In
         values:
         - kube-dns
     topologyKey: kubernetes.io/hostname
```

示例相关配置如下：

- **requiredDuringSchedulingIgnoredDuringExecution**
  此为反亲和性硬性条件，强调 Pod 调度时必须要满足该条件。当不存在满足该条件的节点时，Pod 将不会调度到任何节点（Pending）。如果不使用这种硬性条件，也可以使用 `preferredDuringSchedulingIgnoredDuringExecution` 来指示调度器尽量满足反亲和性条件。当不存在满足该条件的节点时，Pod 也可以调度到某个节点。
- **labelSelector.matchExpressions**
  标记该服务对应 Pod 中 labels 的 key 与 values。
- **topologyKey**
  本示例中使用 `kubernetes.io/hostname`，表示避免 Pod 调度到同一节点。如果您有更高的要求，例如避免调度到同一个可用区的节点，实现异地多活，则可以使用 `failure-domain.beta.kubernetes.io/zone`。但通常情况下，同一个集群的节点都在一个地域。如果节点跨地域，即使使用专线，时延也会很大。如果想避免调度到同一个地域的节点，则可以使用 `failure-domain.beta.kubernetes.io/region`。

2. 使用 topologySpreadConstraints

topologySpreadConstraints 特性在 Kubernetes v1.18 中默认启用，建议在 v1.18 及以上版本的集群中使用 `topologySpreadConstraints` 来打散 Pod 的分布以提高服务可用性。
将 Pod 最大程度上均匀的打散调度到各个节点上：
例如：将所有 nginx 的 Pod 严格均匀打散调度到不同节点上，不同节点上 nginx 的副本数量最多只能相差 1 个，若有节点因其他因素无法调度更多的 Pod（如资源不足），剩余的 nginx 副本 Pending。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: nginx
```

```
    qcloud-app: nginx
  name: nginx
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: nginx
      qcloud-app: nginx
  template:
    metadata:
      labels:
        k8s-app: nginx
        qcloud-app: nginx
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        whenUnsatisfiable: DoNotSchedule
        topologyKey: topology.kubernetes.io/region
        labelSelector:
          matchLabels:
            k8s-app: nginx
      containers:
      - image: nginx
        name: nginx
        resources:
          limits:
            cpu: 500m
            memory: 1Gi
          requests:
            cpu: 250m
            memory: 256Mi
      dnsPolicy: ClusterFirst
```

说明：

- topologyKey：与 podAntiAffinity 中配置类似。
- labelSelector：与 podAntiAffinity 中配置类似，只是这里可以支持选中多组 Pod 的 label。
- maxSkew：必须是大于零的整数，表示能容忍不同拓扑域中 Pod 数量差异的最大值。这里的 1 意味着只允许相差 1 个 Pod。
- whenUnsatisfiable：指示不满足条件时如何处理。DoNotSchedule 表示不调度（保持 Pending），类似强反亲和；ScheduleAnyway 表示要调度，类似弱反亲和，将 Pod 尽量均匀的打散调度到各个节点上，不强制 (DoNotSchedule 改为 ScheduleAnyway)：

```
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    whenUnsatisfiable: ScheduleAnyway
    topologyKey: topology.kubernetes.io/region
    labelSelector:
      matchLabels:
        k8s-app: nginx
```

若集群节点支持跨可用区，可将 Pod 尽量均匀的打散调度到各个可用区以实现更高级别的高可用（topologyKey 改为 topology.kubernetes.io/zone）：

```
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: ScheduleAnyway
```

```
      labelSelector:
        matchLabels:
          k8s-app: nginx
```

更进一步地，可以将 Pod 尽量均匀的打散调度到各个可用区的同时，在可用区内部各节点也尽量打散：

```
  spec:
    topologySpreadConstraints:
    - maxSkew: 1
      whenUnsatisfiable: ScheduleAnyway
      topologyKey: topology.kubernetes.io/zone
      labelSelector:
        matchLabels:
          k8s-app: nginx
    - maxSkew: 1
      whenUnsatisfiable: ScheduleAnyway
      topologyKey: kubernetes.io/hostname
      labelSelector:
        matchLabels:
          k8s-app: nginx
```

使用置放群组从物理层面实现容灾

当云服务器底层硬件或软件故障时，可能导致多台节点同时异常，即使利用反亲和性将 Pod 打散到不同节点上，可能仍无法避免业务异常。可使用 置放群组 将节点从物理机、交换机或机架三种物理层面其中一种进行打散，以避免底层硬件或软件故障造成节点批量异常。操作步骤如下：

1. 登录 置放群组控制台 创建置放群组，根据实际需求从物理机层级、交换机层级和机架层级中选一种作为节点的打散策略。详情请参见 分散置放群组 。

> ⚠ **注意：**
> 置放群组需与 TKE 独立集群在同一地域。

2. 批量添加节点，勾选高级设置中的“将实例添加到分散置放群组”，并选择已创建的置放群组。详情请参见 新增节点 。如下图所示：

3. 在**节点列表**中为该批节点编辑相同的 label 进行标识，这些节点是置放群组中某个同一批次添加的节点。如下图所示：

> ⚠ **注意：**
> 置放群组的策略仅对同一批次的节点生效，即需为每一批次的节点增加 label 并指定不同的值来进行标识。

4. 给需要部署的工作负载的 Pod 指定节点亲和性，指定部署在这一批节点上，同时也指定 Pod 反亲和，将 Pod 在这批节点中尽量打散调度。YAML 示例如下：

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "placement-set-uniq"
          operator: In
          values:
          - "rack1"
```

```
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - nginx
        topologyKey: kubernetes.io/hostname
```

使用 PodDisruptionBudget 避免驱逐导致服务不可用

驱逐节点是一种有损操作，该操作的过程如下：

1. 封锁节点（设置为不可调度，避免新的 Pod 调度上来）。
2. 删除该节点上的 Pod。
3. ReplicaSet 控制器检测到 Pod 减少，会重新创建一个 Pod，调度到新的节点上。

该过程是先删除再创建，并非滚动更新。因此在更新过程中，如果一个服务的所有副本都在被驱逐的节点上，则可能导致该服务不可用。通常，节点被驱逐导致服务不可用的情况会有以下两种：

1. 服务存在单点故障，所有副本都在同一个节点，驱逐该节点时可能造成服务不可用。针对此情况，可参考 使用反亲和性避免单点故障 进行处理。
2. 服务在多个节点，但这些节点被同时驱逐，造成该服务涉及的所有副本同时被删，可能造成服务不可用。针对此情况，可通过配置 PDB（PodDisruptionBudget）来避免所有副本同时被删除。示例如下：

示例1

保证驱逐时 zookeeper 至少有两个副本可用。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

示例2

保证驱逐时 zookeeper 最多有一个副本不可用，相当于逐个删除并在其他节点重建。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: zookeeper
```

更多内容请参考官方文档 Specifying a Disruption Budget for your Application 。

使用 preStopHook 和 readinessProbe 保证服务平滑更新不中断

如果服务不做配置优化，默认情况下更新服务期间可能会产生部分流量异常，请参考以下步骤进行部署。

服务更新场景

通常情况下，服务更新场景会包含以下几种：

- 手动调整服务的副本数量。
- 手动删除 Pod 触发重新调度。
- 驱逐节点（主动或被动驱逐，会先删除 Pod 并在其它节点重建）。
- 触发滚动更新（例如，修改镜像 tag 升级程序版本）。
- HPA（HorizontalPodAutoscaler）自动对服务进行水平伸缩。
- VPA（VerticalPodAutoscaler）自动对服务进行垂直伸缩。

服务更新过程连接异常的原因

滚动更新时，Service 对应的 Pod 会被创建或销毁，Service 对应的 Endpoint 也会新增或移除相应的 `Pod IP:Port`，kube-proxy 会根据 Service 的 Endpoint 中的 `Pod IP:Port` 列表更新节点上的转发规则，而 kube-proxy 更新节点转发规则的动作并不是及时的。
转发规则更新不及时这一现象的出现，主要由于 Kubernetes 的设计理念中各个组件的逻辑是解耦的，它们各自使用 Controller 模式，listAndWatch 感兴趣的资源并做出相应的行为，使得从 Pod 创建或销毁到 Endpoint 更新再到节点上的转发规则更新的整个过程是异步的。
当转发规则没有及时更新时，服务更新期间就有可能发生部分连接异常。以下通过分析 Pod 创建和销毁到规则更新期间的两种情况，寻找服务更新期间部分连接异常发生的原因：

- 情况一：Pod 被创建，但启动速度较慢，Pod 还未完全启动就被 Endpoint Controller 加入到 Service 对应 Endpoint 的 `Pod IP:Port` 列表，kube-proxy watch 到更新也同步更新了节点上的 Service 转发规则（iptables/ipvs），如果此时有请求就可能被转发到还没完全启动完全的 Pod，此时 Pod 还无法正常处理请求，就会导致连接被拒绝。
- 情况二：Pod 被销毁，但是从 Endpoint Controller watch 到变化并更新 Service 对应 Endpoint 再到 kube-proxy 更新节点转发规则这期间是异步的，存在时间差，在这个时间差内 Pod 可能已经完全被销毁了，但转发规则还未更新，就会造成新的请求依旧还能被转发到已经被销毁的 Pod，导致连接被拒绝。

平滑更新

- 针对 情况一，可以给 Pod 中的 container 添加 readinessProbe（就绪检查）。通常是容器完全启动后监听一个 HTTP 端口，kubelet 发送就绪检查探测包，若正常响应则说明容器已经就绪，并将容器状态修改为 Ready。当 Pod 中所有容器都 Ready 时，该 Pod 才会被 Endpoint Controller 加入 Service 对应 Endpoint 中的 `IP:Port` 列表，kube-proxy 再更新节点转发规则，完成更新后即使立刻有请求被转发到的新的 Pod，也能够确保正常处理连接，避免连接异常。
- 针对 情况二，可以给 Pod 中的 container 添加 preStop hook，使 Pod 真正销毁前先 sleep 等待一段时间，留出时间给 Endpoint controller 和 kube-proxy 更新 Endpoint 和转发规则，这段时间 Pod 处于 Terminating 状态，即便在转发规则更新完全之前有请求被转发到 Terminating 的 Pod，依然可以被正常处理，因为 Pod 还在 sleep 没有被真正销毁。

Yaml 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      component: nginx
  template:
    metadata:
      labels:
        component: nginx
    spec:
      containers:
      - name: nginx
        image: "nginx"
        ports:
```

```
      - name: http
        hostPort: 80
        containerPort: 80
        protocol: TCP
      readinessProbe:
        httpGet:
          path: /healthz
          port: 80
          httpHeaders:
          - name: X-Custom-Header
            value: Awesome
        initialDelaySeconds: 15
        timeoutSeconds: 1
      lifecycle:
        preStop:
          exec:
            command: ["/bin/bash", "-c", "sleep 30"]
```

更多参考资料请前往 Kubernetes 官网 Container probes 及 Container Lifecycle Hooks 。

工作负载平滑升级

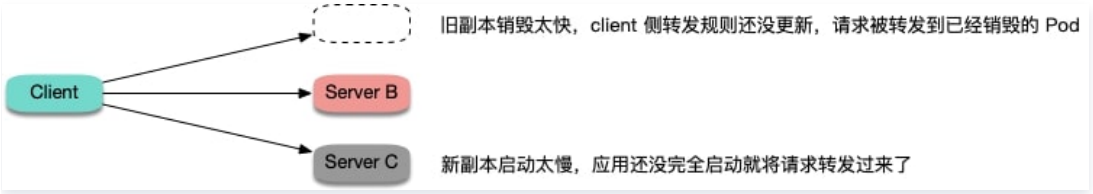
最近更新时间：2023-05-17 15:41:02

解决了服务单点故障和驱逐节点时导致的可用性降低问题后，我们还需要考虑一种可能导致可用性降低的场景，那就是滚动更新。为什么服务正常滚动更新也可能影响服务的可用性呢？可能存在以下原因。

业务有损滚动更新

假如集群内存在服务间调用：

当 server 端发生滚动更新时：



可能发生以下两种情况：

- **情况1：**旧的副本很快销毁，而 client 所在节点 kube-proxy 还没更新完转发规则，仍然将新连接调度给旧副本，造成连接异常，可能会报“connection refused”（进程停止过程中，不再接受新请求）或“no route to host”（容器已经完全销毁，网卡和 IP 已不存在）。
- **情况2：**新副本启动，client 所在节点 kube-proxy 很快 watch 到了新副本，更新了转发规则，并将新连接调度给新副本，但容器内的进程启动很慢（如 Tomcat 这种 java 进程），还在启动过程中，端口还未监听，无法处理连接，也造成连接异常，通常会报 “connection refused” 的错误。

最佳实践

- 针对**情况1**，可以给 container 加 preStop，让 Pod 真正销毁前先 sleep 等待一段时间，等待 client 所在节点 kube-proxy 更新转发规则，然后再真正去销毁容器。这样能保证在 Pod Terminating 后还能继续正常运行一段时间，这段时间如果因为 client 侧的转发规则更新不及时导致还有新请求转发过来，Pod 还是可以正常处理请求，避免了连接异常的发生。听起来感觉有点不优雅，但实际效果还是比较好的，分布式的世界没有银弹，我们只能尽量在当前设计现状下找到并实践能够解决问题的最优解。
- 针对**情况2**，可以给 container 加 ReadinessProbe（就绪检查），让容器内进程真正启动完成后才更新 Service 的 Endpoint，然后 client 所在节点 kube-proxy 再更新转发规则，让流量进来。这样能够保证等 Pod 完全就绪了才会被转发流量，也就避免了连接异常的发生。
  yaml 示例：

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 80
    httpHeaders:
    - name: X-Custom-Header
      value: Awesome
  initialDelaySeconds: 10
  timeoutSeconds: 1
lifecycle:
  preStop:
    exec:
      command: ["/bin/bash", "-c", "sleep 10"]
```

docker run 参数适配

最近更新时间：2024-04-08 17:30:51

本文介绍将已经在本地的 Docker 中调试完毕的容器，迁移到腾讯云容器服务平台中运行时，如何将 Docker run 命令中的参数与腾讯云容器控制台的参数对应。本文以创建一个简单的 GitLab 服务为例。

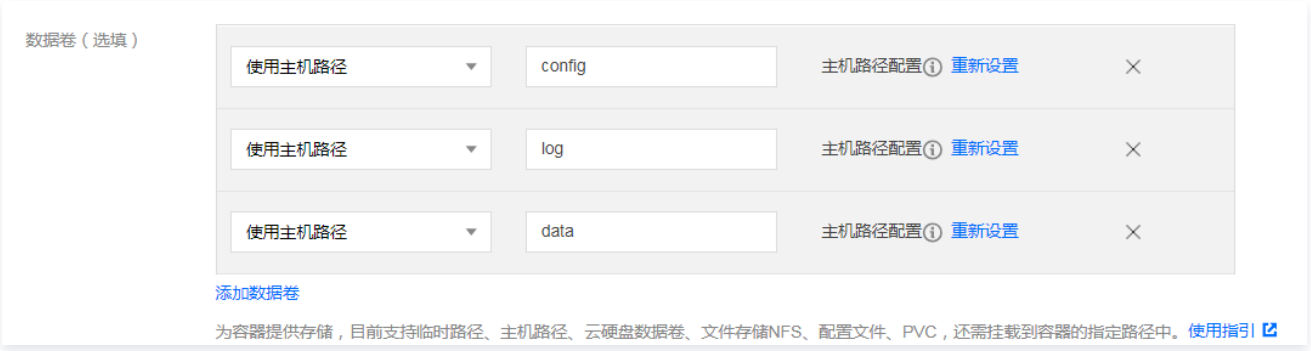
gitlab 容器的参数示例

您可以使用以下 docker run 命令创建出一个简单的 GitLab 容器：

```
docker run \
-d \
-p 20180:80 \
-p 20122:22 \
--restart always \
-v /data/gitlab/config:/etc/gitlab \
-v /data/var/log/gitlab:/var/log/gitlab \
-v /data/gitlab/data:/var/opt/gitlab \
--name gitlab \
gitlab/gitlab-ce:8.16.7-ce.0
```

- `-d`：容器在后台运行。容器平台都是以后台的形式来运行容器，所以本参数不需要在容器控制台指定。
- `-p`：指定端口映射。这里映射了两个端口，容器端口分别是80和22，对外暴露的端口可自行定义，对应到控制台，添加两条端口映射规则，并填写对应的容器端口和服务端口。由于 GitLab 需要提供外网访问，采用了**提供公网访问**访问方式。如下图所示：

- `--restart`：本参数用于指定在容器退出时，是否重启容器。容器平台创建的所有容器退出时，都会重启容器，所以本参数不需要在容器控制台指定。
- `-v`：本参数用于指定容器卷。上面的命令指定了三个卷，对应到容器控制台，我们也需要添加三个**数据卷**，并在**实例内容器**里将这三个卷挂载到容器里。首先我们创建三个卷。如下图所示：



在实例内容器里面，将三个卷分别挂载到容器里。如下图所示：

这里要注意的是，数据卷类型选择的是 `使用主机路径` ，所以容器运行过程中，在容器中生产的数据会被保存到容器所在的节点上，如果容器被调度到其他的节点上，那么数据就丢失了。您可以使用 `云硬盘` 类型数据卷，容器的数据会保存到云硬盘中，即使容器被调度到其他的节点，容器卷的数据也不会丢。

- `--name` ：容器运行的名字。这个参数，对应到容器控制台就是服务名，当然容器名也可以跟服务名用相同的名字。

其它参数

以下为执行 docker run 时，其它常见的参数：

- `-i` ：交互式执行容器。容器控制台只支持后台运行容器，本参数不支持。
- `-t` ：分配虚拟终端，本参数不支持。
- `-e` ：容器运行的环境变量。例如用户执行以下的 docker run 命令：

```
docker run -e FOO='foo' -e BAR='bar' --name=container_name container_image
```

这里用户希望为容器添加两个环境变量，在容器控制台创建服务时，容器的高级设置里可添加容器的环境变量。变量名和变量值分别为：

- 变量名：FOO，变量值：foo。
- 变量名：BAR，变量值：bar。

Command 和 Args

您可以在 docker run 命令中指定进程的命令和参数。例如：

```
docker run --name=kubedns gcr.io/google_containers/kubedns-amd64:1.7 /kube-dns --domain=cluster.local. --
dns-port=10053 -v 2
```

指定了容器进程的命令为 `/kube-dns` ，并指定了三个参数： `--domain=cluster.local.` 、 `--dns-port=10053` 和 `-v 2` 。在控制台中参数设置如下图所示：


运行命令
/kube-dns
新增
控制容器运行的输入命令，查看详情
运行参数
--domain=cluster.local.
--dns-port=10053
-v 2
新增
传递给容器运行命令的输入参数，查看详情

解决容器内时区不一致问题

最近更新时间：2024-08-23 16:20:41

操作场景

腾讯云容器服务（TKE）集群中容器系统时间默认为 UTC 协调世界时间（Universal Time Coordinated），与节点本地所属时区 CST（上海时间）相差8个小时。在容器使用过程中，当需要获取系统时间用于日志记录、数据库存储等相关操作时，容器内时区不一致问题将会带来一系列困扰。
默认时间不支持直接以集群为单位进行修改，但可在单个容器内进行修改。本文提供了容器内时区不一致问题的多种解决方案，请选择合适的方案进行操作：

- 方案1：Dockerfile 中创建时区文件（推荐）
- 方案2：挂载主机时区配置到容器

操作环境

本文中所有操作步骤均在 TKE 集群节点上完成，相关操作环境如下所示，请根据实际情况结合文档解决问题：

角色	地域	配置	操作系统	Kubernetes 版本信息
节点	华南地区（广州）	• CPU：1核 • 内存:：1GB • 带宽：1 Mbps • 系统盘：50 GB（普通云硬盘）	CentOS Linux 7（Core）	1.16.3

问题定位

1. 参考 使用标准登录方式登录 Linux 实例（推荐），登录目标节点。
2. 执行以下命令，查看本地时间。

```
date
```

返回结果如下图所示：

```
[root@VM_6_12_centos ~]# date
Tue Mar  3 16:23:53 CST 2020
```

3. 依次执行以下命令，查看容器内 CentOS 系统默认时区。

```
docker run -it centos /bin/sh
```

```
date
```

返回结果如下图所示：

```
[root@VM_6_12_centos ~]# docker run -it centos /bin/sh
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
8a29a15cefae: Pull complete
Digest: sha256:fe8d824220415eed5477b63addf40fb06c3b049404242b31982106ac204f6700
Status: Downloaded newer image for centos:latest
sh-4.4# date
Tue Mar  3 08:24:29 UTC 2020
sh-4.4#
```

对比发现，本地时间与容器内时区不一致。

4. 执行以下命令，退出容器。

```
exit
```

操作步骤

方案1：Dockerfile 中创建时区文件（推荐）

在构建基础镜像或在基础镜像的基础上制作自定义镜像时，在 Dockerfile 中创建时区文件即可解决单一容器内时区不一致问题，且后续使用该镜像时，将不再受时区问题困扰。

1. 执行以下命令，新建 Dockerfile.txt 文件。

```
vim Dockerfile.txt
```

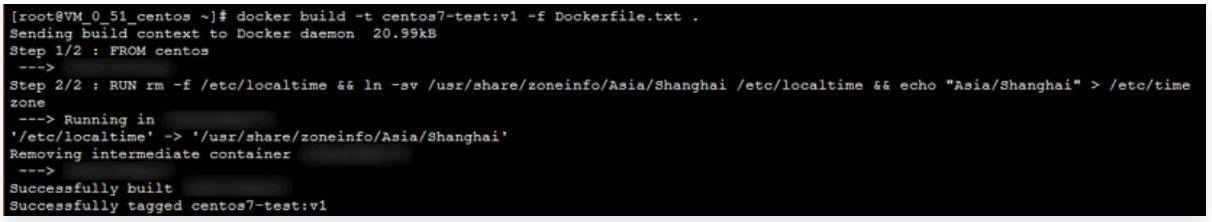
2. 按 i 切换至编辑模式，写入以下内容，配置时区文件。

```
FROM centos
RUN rm -f /etc/localtime \
&& ln -sv /usr/share/zoneinfo/Asia/Shanghai /etc/localtime \
&& echo "Asia/Shanghai" > /etc/timezone
```

3. 按 **Esc**，输入 **:wq**，保存文件并返回。
4. 执行以下命令，构建容器镜像。

```
docker build -t centos7-test:v1 -f Dockerfile.txt .
```

返回结果如下图所示：



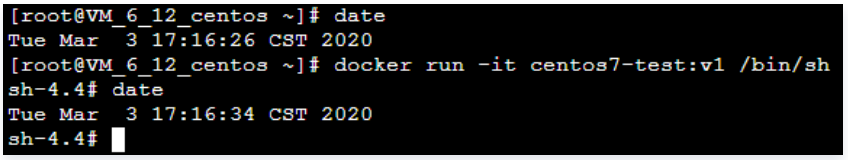
5. 依次执行以下命令，启动容器镜像并查看容器内时区。

```
date
```

```
docker run -it centos7-test:v1 /bin/sh
```

```
date
```

此时，容器内时区已与本地时间一致。如下图所示：



6. 执行以下命令，退出容器。

```
exit
```

方案2：挂载主机时区配置到容器

解决容器内时区不一致问题，还可以通过挂载主机时间配置到容器的方式进行解决。该方式可以在容器启动时进行设置，也可以在 YAML 文件中使用主机路径挂载数据卷到容器。

容器启动时挂载主机时间配置到容器

挂载主机时间到容器内覆盖配置时，有以下两种选择：

- **方式1：挂载本地 `/etc/localtime`：需确保该主机时区配置文件存在且时区正确。**
- **方式2：挂载本地 `/usr/share/zoneinfo/Asia/Shanghai`：当本地 `/etc/localtime` 不存在或者时区不正确时，可选择直接挂载该配置文件。**

请根据实际情况，选择以下方式，进行挂载主机时间配置到容器：

方式1

挂载本地 `/etc/localtime`：

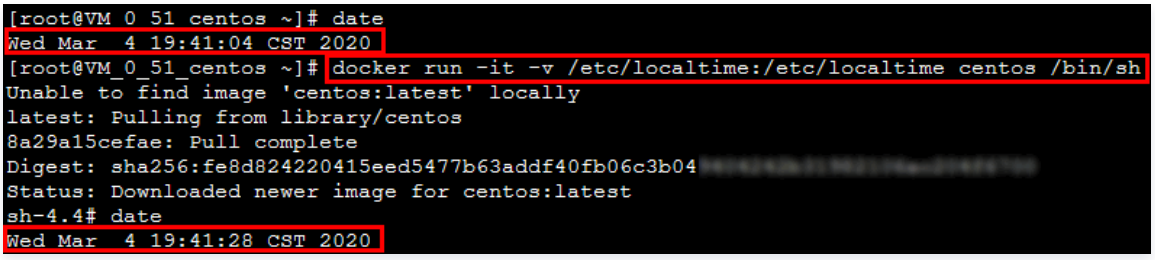
1. **依次执行以下命令，查看本地时间并挂载本地 `/etc/localtime` 到容器内。**

```
date
```

```
docker run -it -v /etc/localtime:/etc/localtime centos /bin/sh
```

```
date
```

返回结果如下图所示，容器内时区已与本地时间一致：



2. **执行以下命令，退出容器。**

```
exit
```

方式2

挂载本地 `/usr/share/zoneinfo/Asia/Shanghai`：

1. **依次执行以下命令，查看本地时间并挂载本地 `/usr/share/zoneinfo/Asia/Shanghai` 到容器内。**

```
date
```

```
docker run -it -v /usr/share/zoneinfo/Asia/Shanghai:/etc/localtime centos /bin/sh
```

```
date
```

返回结果如下图所示，容器内时区已与本地时间一致：

2. **执行以下命令，退出容器。**

```
exit
```

YAML 文件使用主机路径挂载数据卷到容器

本节内容以 `mountPath:/etc/localtime` 为例，介绍在 YAML 文件中如何通过数据卷挂载主机时区配置到容器内，解决容器内时区不一致的问题。

1. 在节点上执行以下命令，创建 pod.yaml 文件。

```
vim pod.yaml
```

2. 按 i 切换至编辑模式，写入以下内容。

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  namespace: default
spec:
  restartPolicy: OnFailure
  containers:
  - name: nginx
    image: nginx-test
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: date-config
      mountPath: /etc/localtime
    command: ["sleep", "60000"]
  volumes:
  - name: date-config
    hostPath:
      path: /etc/localtime
```

3. 按 Esc，输入 :wq，保存文件并返回。
4. 执行以下命令，新建该 Pod。

```
kubectl create -f  pod.yaml
```

返回结果如下图所示：

```
[root@VM_6_5_centos ~]# kubectl create -f  pod.yaml
pod/test created
```

5. 依次执行以下命令，查看该容器内时区。

```
date
```

```
kubectl exec -it test date
```

返回结果如下图所示，与本地系统时区一致即为成功：

```
[root@VM_6_5_centos ~]# date
Wed Mar  4 11:56:27 CST 2020
[root@VM_6_5_centos ~]# kubectl exec -it test date
Wed Mar  4 11:56:31 CST 2020
[root@VM_6_5_centos ~]#
```

容器 coredump 持久化

最近更新时间：2024-11-25 17:15:02

操作场景

容器有时会在发生异常后无法正常工作，业务日志中若无足够的信息来定位问题原因，则需要结合 coredump 来进一步分析，本文将介绍如何使容器产生 coredump 并保存。

> **注意：**
> 本文仅适用于容器服务 TKE 集群。

前提条件

已登录 容器服务控制台 。

操作步骤

开启 coredump

1. 在节点上执行以下命令，为节点设置 core 文件的存放路径格式：

```
# 在节点上执行
echo "/tmp/cores/core.%h.%e.%p.%t" > /proc/sys/kernel/core_pattern
```

主要参数信息如下：
- %h：主机名（在 Pod 内主机名即 Pod 的名称），推荐。
- %e：程序文件名，推荐。
- %p：进程 ID，可选。
- %t：coredump 的时间，可选。

最终生成的 core 文件完整路径如下所示：

```
/tmp/cores/core.nginx-7855fc5b44-p2rzt.bash.36.1602488967
```

2. 节点完成配置后，无需更改容器原有配置，将以继承的方式自动生效。如需在多个节点上批量执行，则请对应实际情况进行操作：
   - 对于存量节点，请参见 使用 Ansible 批量操作 TKE 节点 。
   - 对于增量节点，请参见 设置节点的启动脚本 。

启用 COS 扩展组件

为了避免容器重启后丢失 core 文件，需要为容器挂载 volume。由于为每个 Pod 单独挂载云盘的成本太高，所以将组件挂载至 COS 对象存储。具体操作步骤请参见 安装 COS 扩展组件 。

创建存储桶

登录 对象存储控制台 ，手动创建 COS 存储桶，用于存储容器 coredump 生成的 core 文件，本文以创建自定义名称为 coredump 的存储桶为例。具体操作步骤请参见 创建存储桶 。

创建 Secret

可通过以下3种方式创建可以访问对象存储的 Secret，请按需选择：
- 若通过控制台使用对象存储，可参见 创建可以访问对象存储的 Secret 。
- 若通过 YAML 文件使用对象存储，可参见 创建可以访问对象存储的 Secret 。
- 若使用 kubectl 命令行工具创建 Secret，可参考以下代码片段：

```
kubectl create secret generic cos-secret -n kube-system  --from-
literal=SecretId=AKI*****************lV --from-literal=SecretKey=paQ9***************sZF
```

> **注意：**
> 注意替换 SecretId、SecretKey 以及命名空间。

创建 PV 和 PVC

使用 COS 插件需要手动创建 PV 和创建 PVC，并完成绑定。

创建 PV

1. 在目标集群详情页面，选择左侧菜单栏中的**存储** > **PersistentVolume**，进入 “PersistentVolume” 页面。
2. 单击**新建**进入 “新建PersistentVolume” 页面，参考以下信息创建 PV。如下图所示：

主要参数信息如下：

- **来源设置**：选择**静态创建**。
- **Secret**：选择已在 创建 Secret 中创建的 Secret，本文以 coredump 为例（kube-system 命名空间下）。
- **存储桶列表**：选中已创建的用于存储 coredump 文件的存储桶。
- **存储桶子目录**：此处指定根目录，如果需要指定子目录，请提前在存储桶中创建。

3. 单击**创建PersistentVolume**即可。

创建 PVC

1. 在目标集群详情页，选择左侧菜单栏中的**存储** > **PersistentVolumeClaim**，进入 “PersistentVolumeClaim” 页面。

2. 单击**新建**进入“新建PersistentVolumeClaim” 页面，参考以下信息创建 PVC。如下图所示：

主要参数信息如下：

- **命名空间**：要与需要挂载存储 COS 的 PVC 的容器所在命名空间相同，如果有多个命名空间，可以创建多对 PV 与 PVC。
- **PersistentVolume**：选择在 创建 PV 中已创建的 PV 的名称。

3. 单击**创建PersistentVolumeClaim**即可。

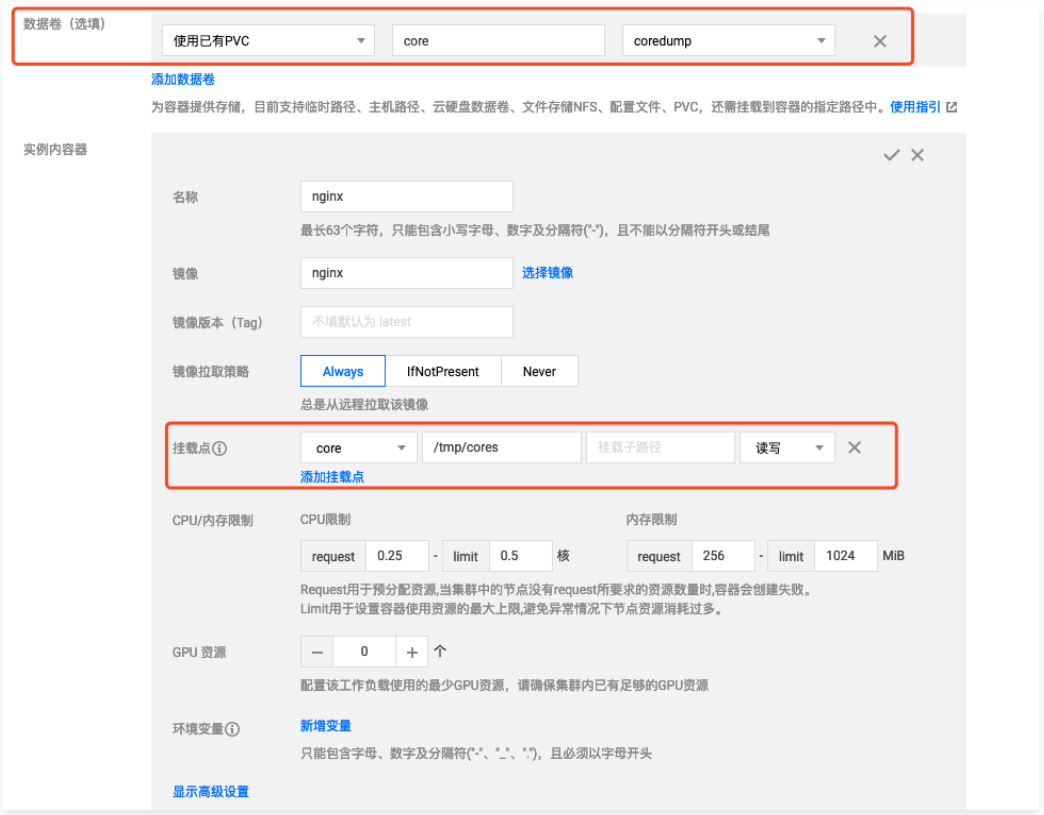
挂载 COS 存储

通过控制台创建 Pod 使用 PVC

> **说明：**
> 本步骤以创建工作负载 Deployment 为例。

1. 在目标集群详情页，选择左侧菜单栏中的**工作负载** > **Deployment**，进入 “Deployment” 页面。

2. 单击**新建**进入“新建Workload” 页面，参考 创建 Deployment 进行创建，并设置数据卷挂载。如下图所示：



主要参数信息如下：

- **数据卷**：添加在 创建 PVC 中已创建的 PVC。
- **挂载点**：单击**添加挂载点**，进行挂载点设置。选择为该步骤中所添加的数据卷 “core”。引用**数据卷**中声明的 PVC，挂载至目标路径，本文以 `/tmp/cores` 为例。

3. 单击**创建Workload**即可。

通过 YAML 创建 Pod 使用 PVC

通过 YAML 创建 Pod，示例如下：

```
containers:
- name: pod-cos
  command: ["tail", "-f", "/etc/hosts"]
  image: "centos:latest"
  volumeMounts:
  - mountPath: /tmp/cores
    name: core
volumes:
- name: core
  persistentVolumeClaim:
    # Replaced by your pvc name.
    claimName: coredump
```

相关文档

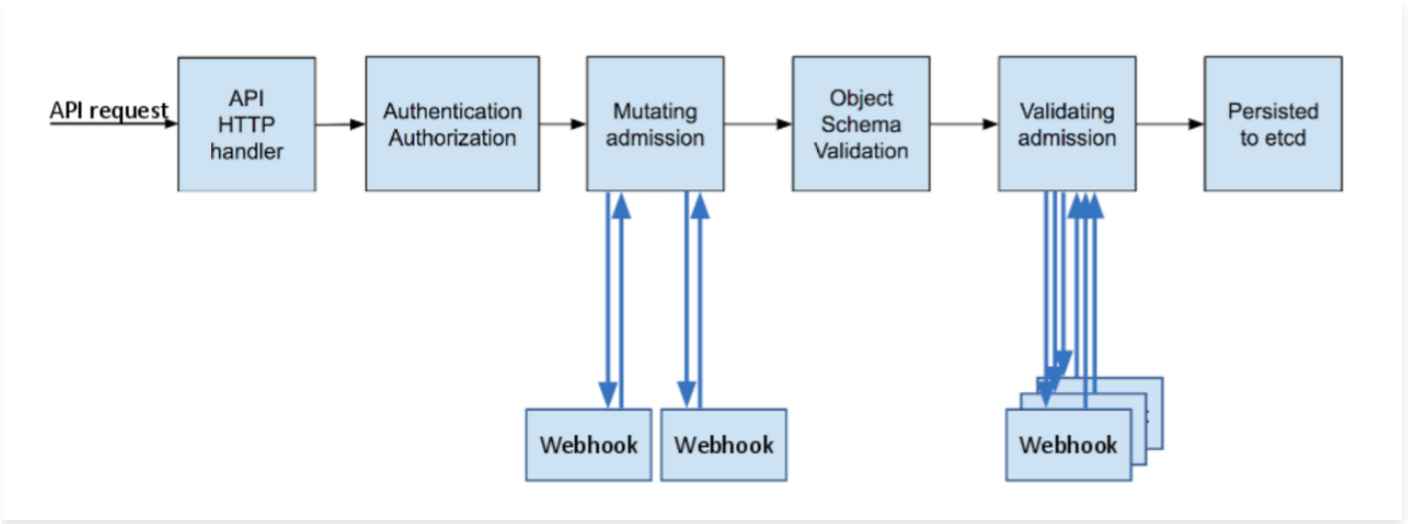
使用对象存储 COS

在 TKE 中使用动态准入控制器

最近更新时间：2023-05-17 15:41:03

操作场景

动态准入控制器 Webhook 在访问鉴权的过程中可以更改请求对象或完全拒绝该请求，其调用 Webhook 服务的方式使其独立于集群组件。动态准入控制器具有很大的灵活性，可便捷地进行众多自定义准入控制。下图为动态准入控制在 API 请求调用链的位置，如需了解更多信息，请前往 Kubernetes 官网 。



由图可知，动态准入控制分为执行及验证两个阶段。首先执行 Mutating 阶段，该阶段可对到达请求进行修改，然后执行 Validating 阶段来验证到达的请求是否被允许，两个阶段可单独或组合使用。
本文将在容器服务 TKE 中实现一个简单的动态准入控制调用示例，您可结合实际需求参考本文进行操作。

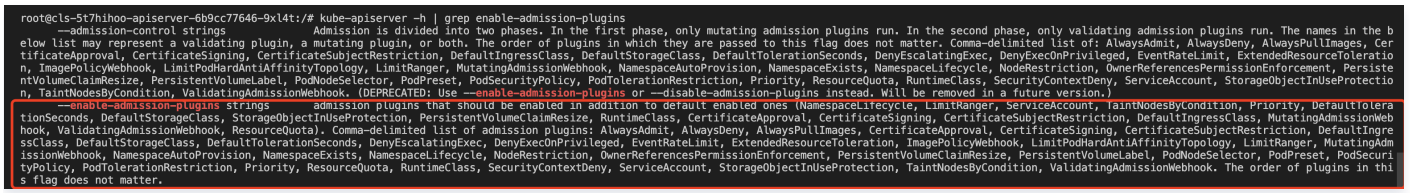
操作步骤

查看及验证插件

TKE 现有集群版本（1.10.5及以上）已默认开启了 validating admission webhook 和 mutating admission webhook API。若您的集群版本低于1.10.5，则可执行以下命令验证当前集群是否开启插件。

```
kube-apiserver -h | grep enable-admission-plugins
```

返回结果如已包含 `MutatingAdmissionWebhook` 和 `ValidatingAdmissionWebhook`，则说明当前集群已开启动态准入控制器插件。如下图所示：



签发证书

为确保动态准入控制器调用可信任的 Webhook 服务端，须通过 HTTPS 调用 Webhook 服务（TLS 认证），则需为 Webhook 服务端颁发证书，并且在注册动态准入控制 Webhook 时为 `caBundle` 字段（`ValidatingWebhookConfiguration` 和 `MutatingAdmissionWebhook` 资源清单中的 `caBundle` 字段）绑定受信任的颁发机构证书（CA）来核验 Webhook 服务端的证书是否可信任。本文介绍了 制作自签证书 及 使用 K8S CSR API 签发证书 两种推荐的颁发证书方法。

> **注意**
> 当 `ValidatingWebhookConfiguration` 和 `MutatingAdmissionWebhook` 使用 `clientConfig.service` 配置时（Webhook 服务在集群内），为服务器端颁发的证书域名必须为 `<svc_name>.<svc_namespace>.svc` 。

方法1：制作自签证书

制作自签证书的方法不依赖于 K8S 集群，比较独立，类似于为网站制作自签证书。目前有很多工具可制作自签证书，本文以使用 Openssl 为例。具体步骤如下：

1. 执行以下命令，生成密钥位数为2048的 `ca.key` 。

```
openssl genrsa -out ca.key 2048
```

2. 执行以下命令，依据 `ca.key` 生成 `ca.crt` 。"webserver.default.svc" 为 Webhook 服务端在集群中的域名， `-days` 参数用于设置证书有效时间。

```
openssl req -x509 -new -nodes -key ca.key -subj "/CN=webserver.default.svc" -days 10000 -out ca.crt
```

3. 执行以下命令，生成密钥位数为2048的 `server.key` 。

```
openssl genrsa -out server.key 2048
```

4. 创建用于生成证书签名请求（CSR）的配置文件 `csr.conf` 。示例如下：

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn

[ dn ]
C = cn
ST = shaanxi
L = xi'an
O = default
OU = websever
CN = webserver.default.svc

subjectAltName = @alt_names

[ alt_names ]
DNS.1 = webserver.default.svc

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
subjectAltName=@alt_names
```

5. 执行以下命令，基于配置文件 `csr.conf` 生成证书签名请求。

```
openssl req -new -key server.key -out server.csr -config csr.conf
```

6. 执行以下命令，使用 `ca.key` 、 `ca.crt` 和 `server.csr` 颁发生成服务器证书（x509签名）。

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
 -CAcreateserial -out server.crt -days 10000 \
 -extensions v3_ext -extfile csr.conf
```

7. 执行以下命令，查看 Webhook server 端证书。

```
openssl x509  -noout -text -in ./server.crt
```

生成的证书及密钥文件说明如下：

- `ca.crt`：为颁发机构证书。
- `ca.key`：为颁发机构证书密钥，用于服务端证书颁发。
- `server.crt`：为颁发的服务端证书。
- `server.key`：为颁发的服务端证书密钥。

方法2：使用 K8S CSR API 签发证书

可使用 K8S 的证书颁发机构系统来下发证书，执行以下脚本可使用 K8S 集群根证书和根密钥签发一个可信任的证书用户。

> **注意**
> 用户名需为 Webhook 服务在集群中的域名。

```
USERNAME='webserver.default.svc' # 设置需要创建的用户名为 Webhook 服务在集群中的域名
# 使用 Openssl 生成自签证书 key
openssl genrsa -out ${USERNAME}.key 2048
# 使用 Openssl 生成自签证书 CSR 文件，CN 代表用户名，O 代表组名
openssl req -new -key ${USERNAME}.key -out ${USERNAME}.csr -subj "/CN=${USERNAME}/O=${USERNAME}"
# 创建 Kubernetes 证书签名请求（CSR）
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
   name: ${USERNAME}
spec:
   request: $(cat ${USERNAME}.csr | base64 | tr -d '\n')
   usages:
   - digital signature
   - key encipherment
   - server auth
EOF
# 证书审批允许信任
kubectl certificate approve ${USERNAME}
# 获取自签证书 CRT
kubectl get csr ${USERNAME} -o jsonpath={.status.certificate} > ${USERNAME}.crt
```

- `${USERNAME}`.crt：为 Webhook 服务端证书。
- `${USERNAME}`.key：为 Webhook 服务端证书密钥。

使用示例

本文将使用 `ValidatingWebhookConfiguration` 资源在 TKE 中实现一个动态准入 Webhook 调用示例。为了确保可访问性，示例代码 Fork 自 原代码库，示例代码实现了一个简单的动态准入 Webhook 请求和响应的接口，具体接口格式请参见 Webhook 请求和响应 。示例代码可在 示例代码 中获取，本文将使用其作为 Webhook 服务端代码。

1. 对应实际使用颁发证书方法，准备 `caBundle` 内容。
   - 若颁发证书使用方法1，则执行以下命令，使用 `base64` 编码 `ca.crt` 生成 `caBundle` 字段内容。

     ```
     cat ca.crt | base64 --wrap=0
     ```

   - 若颁发证书使用方法2，集群的根证书即为 `caBundle` 字段内容。获取步骤如下：

     1.1.1 登录容器服务控制台，选择左侧导航栏中的 集群 。

     1.1.2 在“集群管理”页面，选择集群 ID。

1.1.3 在集群“基本信息”页面，查看“集群APIServer信息”模块的 “Kubeconfig”，并在 Kubeconfig 中的 `clusters.cluster[].certificate-authority-data` 字段进行获取，该字段已进行 `base64` 编码，无需处理。

2. 复制生成的 `ca.crt`（颁发机构证书）、`server.crt`（HTTPS 证书）及 `server.key`（HTTPS 密钥）到项目主目录。如下图所示：

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# ls
admission.yaml  app  ca.crt  controller.yaml  Dockerfile  pod.yaml  server.crt  server.key
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control#
```

3. 修改项目中的 Dockerfile，添加三个证书文件到容器工作目录。如下图所示：

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# cat Dockerfile
FROM node:12.18.2
WORKDIR /usr/src/app
COPY app/package*.json ./
RUN npm install
COPY app .
COPY ca.crt .
COPY server.crt .
COPY server.key .
EXPOSE 8443
USER 1000:1000
CMD [ "npm", "start" ]
```

4. 执行以下命令，构建 Webhook 服务端镜像。

```
docker build -t webserver .
```

5. 部署一个域名为 “weserver.default.svc” 的 Webhook 后端服务，修改适配后的 `controller.yaml` 如下所示：

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# cat controller.yaml
apiVersion: v1
kind: Service
metadata:
  name: webserver
spec:
  ports:
  - port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    run: webserver
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: webserver
  name: webserver
spec:
  containers:
  - image: webserver
    name: webserver
    ports:
    - containerPort: 8443
    imagePullPolicy: IfNotPresent
    livenessProbe:
      httpGet:
        port: 8443
        path: /hc
        scheme: HTTPS
    readinessProbe:
      httpGet:
        port: 8443
        path: /hc
        scheme: HTTPS
    resources:
      limits:
        cpu: 100m
        memory: 128Mi
      requests:
        cpu: 10m
        memory: 12Mi
    securityContext:
      runAsNonRoot: true
      readOnlyRootFilesystem: true

root@VM-0-12-ubuntu:~/hello-dynamic-admission-control#
```

6. 注册创建类型为 `ValidatingWebhookConfiguration` 的资源，修改适配项目中的 `admission.yaml` 文件。如下图所示：
本示例配置的 Webhook 触发规则为：当创建 `pods` 类型、API 版本 “v1” 时触发调用，`clientConfig` 配置对应上述在集群中创建的 Webhook 后端服务，`caBundle` 字段内容为证书颁发方法一获取的 `ca.crt` 内容。

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# cat admission.yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "webserver.default.svc"
webhooks:
- name: "webserver.default.svc"
  rules:
  - apiGroups:   [""]
    apiVersions: ["v1"]
    operations:  ["CREATE"]
    resources:   ["pods"]
    scope:       "Namespaced"
  clientConfig:
    service:
      namespace: "default"
      name: "webserver"
    caBundle: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURFekNDQWZ1Z0F3SUJBZ0lKQU52Mkg3WlYxb1lFTUEwR0NTcUdTSWIzRFFFQkN3VUFNQ0F4SGpBY0JnTlYKQkFNTUZYZGxZbk5sY25abGNpNWtaV1poZFd4MExuTjJZekFlRncweU1ERXhNVGN3TkRVeU1URmFGdzAwT0RBMApNRFF3TkRVeU1URmFNQ0F4SGpBY0JnTlZCQU1NRlhkbFluTmxjblpsY2k1a1pXWmhkV3gwTG5OMll6Q0NBU0l3DQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBAMEVOUXhDWDA2ZVBjVFlyendwTGtqZHMyWU5pSmRiM3pVVEhpZWJTUzVWWkRxWApPT0NWWUt4VnNSUkRwRWFNMDFxR0NiNWJnc2xwTXpHZk5FRUF1K2dLSVM1VGFsTkRhVVhacWRHT0xRRUNBd0VBCkFhTlFNRTR3SFFZRFZSME9CQllFRkZiWkVzeEhaSGdseXVqaWFPZ2w5eS9rT1Nrck1COEdBMVVkSXdRWU1CYUEKRkZiWkVzeEhaSGdseXVqaWFPZ2w5eS9rT1Nrck1Bd0dBMVVkRXdRRk1BTUJBZjh3RFFZSktvWklodmNOQVFFTApCUUFEZ2dFQkFIVzI4c01RWFNFUnZYMFJDbG1vT3oydFFhVkVhaWd2aU1SR09SZithd3dUNWpWdktNRHhvREtRCnhRUHh6enhhWjFNRDd4QmRUTlQ2VFArZWNTeC9qdEh2RkpPQk10NHQrdHRlb3h5T1lYVWZjdNTHRyb2dUYW0KeVE4b0JZN3ppOWtkaGt5cnhmanEraFdZWis5d0hxWFE2U2NFaktzeWxRMjVsdllGNG1FOVB0ZFNXcFpURWtmegpFSklFUjBPZnZxd2F4VHBISEN1NkpucjNDM2NtMDJpWkgyYStMWWVDZk1EOTU2YTJqbU4xWCs2cWg0VGVGaHRmCjJCZnNHVUtyVkpJOGUwc01PcUVUV2YrbEtTZ3VQS0FzWStKRXhRbmhzbWJHcjZNNk5MNjR6Zzh5dVl6aVY4UnYKRDc3M0V4a0tzVXdrMU9SQ1hySm4rWWtXWk53VlJpQT0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo="
    admissionReviewVersions: ["v1"]
    sideEffects: None
    timeoutSeconds: 5
```

7. 注册好后创建一个 Pod 类型且 API 版本为 “v1” 的测试资源。如下图所示：

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# cat pod.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
  name: hello-pod
spec:
  containers:
  - name: hello
    image: alpine
    command: ['tail', '-f', '/dev/null']
```

8. 测试代码已打印请求日志，查看 Webhook 服务端日志即可查看动态准入控制器触发了 webhook 调用。如下图所示：

```
{
  kind: 'AdmissionReview',
  apiVersion: 'admission.k8s.io/v1',
  request: {
    uid: '31ce0418-ba2e-4daf-a6f4-7e97454d06d1',
    kind: { group: '', version: 'v1', kind: 'Pod' },
    resource: { group: '', version: 'v1', resource: 'pods' },
    requestKind: { group: '', version: 'v1', kind: 'Pod' },
    requestResource: { group: '', version: 'v1', resource: 'pods' },
    name: 'hello-pod',
    namespace: 'default',
    operation: 'CREATE',
    userInfo: { username: '100015757548-1600947194', groups: [Array] },
    object: {
      kind: 'Pod',
      apiVersion: 'v1',
      metadata: [Object],
      spec: [Object],
      status: [Object]
    },
    oldObject: null,
    dryRun: false,
    options: { kind: 'CreateOptions', apiVersion: 'meta.k8s.io/v1' }
  }
}
```

9. 此时查看创建的测试 pod 已成功创建，由于测试 Webhook 服务端代码已具备 `allowed: true` 配置项，即可创建成功该测试 pod。如下图所示：

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# cat app/app.js
const bodyParser = require('body-parser');
const express = require('express');
const fs = require('fs');
const https = require('https');

const app = express();
app.use(bodyParser.json());
const port = 8443;

const options = {
  ca: fs.readFileSync('ca.crt'),
  cert: fs.readFileSync('server.crt'),
  key: fs.readFileSync('server.key'),
};

app.get('/hc', (req, res) => {
  res.send('ok');
});

app.post('/', (req, res) => {
  if (
    req.body.request === undefined ||
    req.body.request.uid === undefined
  ) {
    res.status(400).send();
    return;
  }
  console.log(req.body); // DEBUGGING
  const { request: { uid } } = req.body;
  res.send({
    apiVersion: 'admission.k8s.io/v1',
    kind: 'AdmissionReview',
    response: {
      uid,
      allowed: true,
    },
  });
});

const server = https.createServer(options, app);

server.listen(port, () => {
  console.log(`Server running on port ${port}/`);
});
```

如需进一步验证，将 “allowed” 改为 “false” 后重复上述步骤重新构建 Webserver 服务端镜像，并重新部署 `controller.yaml` 和

`admission.yaml` 资源。当再次尝试创建 pods 资源时请求被动态准入拦截，则说明配置的动态准入策略已生效。如下图所示：

```
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control# kubectl apply -f pod.yaml
Error from server: error when creating "pod.yaml": admission webhook "webserver.default.svc" denied the request without explanation
root@VM-0-12-ubuntu:~/hello-dynamic-admission-control#
```

总结

本文主要介绍了动态准入控制器 Webhook 的概念和作用、如何在 TKE 集群中签发动态准入控制器所需的证书，并使用简单示例演示如何配置和使用动态准入 Webhook 功能。

参考资料

- Kubernetes Dynamic Admission Control by Example
- Dynamic Admission Control

网络
DNS 相关
TKE DNS 最佳实践

最近更新时间：2025-06-04 18:05:02

总述

DNS 作为 Kubernetes 集群中服务访问的第一环节，其稳定性和性能至关重要，如何以更优的方式配置和使用 DNS，涉及到方方面面，本文档将总结这些最佳实践。

> **警告：**
> CoreDNS 依赖集群中的 kube-system 命名空间下名称为 kube-dns 的 Service 服务，请不要删除或修改该 Service，否则会导致节点无法访问集群内 DNS 服务。

选择最佳 CoreDNS 版本

下表列出了随各个版本 TKE 集群默认部署的 CoreDNS 版本：

TKE Version	CoreDNS version
v1.24	1.8.4
v1.22	1.8.4
v1.20	1.8.4
v1.18	1.7.0
v1.16	1.6.2
v1.14	1.6.2

由于历史原因，可能会有 v1.18 及以上版本的集群仍然部署 1.6.2 版本的 CoreDNS，如果当前 CoreDNS 版本不满足需求，可以按如下指引手动升级：

- 升级到1.7.0
- 升级到1.8.4

配置合适的 CoreDNS 副本数

1. TKE 默认设置 CoreDNS 副本数为2，且配置了 `podAntiAffinity` 使两副本部署在不同节点。
2. 针对节点数大于80的集群，建议安装 NodeLocal DNSCache，详情请参见 在 TKE 集群中使用 NodeLocal DNS Cache 。
3. 一般根据集群内业务访问 DNS 的 QPS 来确定 CoreDNS 合理的副本数，也可以根据节点数以及总核数来确定，在安装 NodeLocal DNSCache 后，建议 CoreDNS 最大副本数为10，可以按照如下方式配置：
   副本数 = min (max (ceil (QPS/10000), ceil (集群节点数/8)), 10)
   示例：
   - 集群节点数为10，DNS 服务请求 QPS 为22000，则副本数为3。
   - 集群节点数为30，DNS 服务请求 QPS 为15000，则副本数为4。
   - 集群节点数为100，DNS 服务请求 QPS 为50000，则副本数为10（已部署 NodeLocal DNSCache）。
4. 可以通过在控制台 安装 DNSAutoScaler 组件，来实现自动调整 CoreDNS 副本数（要注意提前配置好平滑升级），组件的默认配置如下：

```
data:
  ladder: |-
 {
   "coresToReplicas":
   [
     [ 1, 1 ],
     [ 128, 3 ],
```

```
      [ 512,4 ],
    ],
    "nodesToReplicas":
    [
      [ 1, 1 ],
      [ 2, 2 ]
    ]
  }
```

使用 NodeLocal DNSCache

在 TKE 集群中部署 NodeLocal DNSCache 可以提升服务发现的稳定性和性能，其通过在节点上作为 DaemonSet 运行 DNS 缓存代理来提高集群 DNS 性能。
关于更多 NodeLocal DNSCache 的介绍及如何在 TKE 集群中部署 NodeLocal DNSCache 的具体步骤，详情请参见 在 TKE 集群中使用 NodeLocal DNS Cache 。

CoreDNS 的上游 DNS

腾讯云 TKE 集群的节点及 CoreDNS 默认将 VPCDNS 配置为上游解析服务。当集群内业务访问外部域名时，DNS 请求会通过上游 VPCDNS 进行解析。CoreDNS 默认启用 cache 插件以缓存解析结果，降低对上游 DNS 依赖。但在爬虫等高并发场景下，若业务需频繁解析大量外部域名，需注意 VPCDNS 的 QPS 限制，避免因超出配额导致解析失败。
参考文档：Private DNS 产品使用限制 。

配置 CoreDNS 平滑升级

当重启节点或者升级 CoreDNS 时，可能导致 CoreDNS 部分副本在一段时间不可用，可以通过以下配置，最大程度保证 DNS 服务整体的可用性，实现平滑升级。

配置会话保持

kube-proxy 为 iptables 模式，无需配置会话保持

iptables 模式下，kube-proxy 会在同步 iptables 规则后及时清理遗留的 conntrack 表项，不存在会话保持问题，无需配置。

kube-proxy 为 IPVS 模式，需要配置 IPVS UDP 协议的会话保持超时时间

IPVS 模式下，内核默认打开会话保持特性，这会导致在 CoreDNS 升级或重启期间，业务侧的 DNS 解析请求，在 5分钟 (300s) 内概率性超时。如果业务自身没有 UDP 服务，可以降低 IPVS UDP 协议的会话保持超时时间，以此来减少解析超时的持续时间。

1. 集群版本大于等于1.18，kube-proxy 提供参数 `--ipvs-udp-timeout` ，该参数默认值为5分钟 (300s) ，推荐配置为 `--ipvs-udp-timeout=10s` ，以便将解析超时的持续时间缩短为10s，请按如下方式配置 kube-proxy DaemonSet：

```
spec:
  containers:
  - args:
    - --kubeconfig=/var/lib/kube-proxy/config
    - --hostname-override=$(NODE_NAME)
    - --v=2
    - --proxy-mode=ipvs
    - --ipvs-scheduler=rr
    - --nodeport-addresses=$(HOST_IP)/32
    - --ipvs-udp-timeout=10s
    command:
    - kube-proxy
    name: kube-proxy
```

2. 集群版本小于等于1.16，kube-proxy 不支持该参数，可以使用 `ipvsadm` 工具批量在节点侧修改：

```
yum install -y ipvsadm
ipvsadm --set 900 120 10
```

3. 配置完成后，可以按如下方式验证：

```
ipvsadm -L --timeout
Timeout (tcp tcpfin udp): 900 120 10
```

> ⚠ **注意：**
> 在配置完成后，需要等待5分钟，再继续后面的步骤；如果业务有使用 UDP 服务，请 提交工单 来寻求帮助。

配置 CoreDNS 优雅退出

已经收到退出信号的副本，可以通过配置 lameduck 使其能在一段时间内继续提供服务，按如下方式配置 CoreDNS 的 configmap（仅展示 CoreDNS 1.6.2版本的部分配置，其它版本配置参见 手动升级 CoreDNS ）：

```
        .:53 {
            health {
                lameduck 30s
            }
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
        }
```

配置 CoreDNS 服务就绪确认

新副本启动后，需确认其服务就绪，再加入 DNS 服务的后端列表。

1. 打开 ready 插件，按如下方式配置 CoreDNS 的 configmap（仅展示 CoreDNS 1.6.2版本的部分配置，其它版本配置参见 手动升级 CoreDNS ）：

```
        .:53 {
            ready
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
        }
```

2. 为 CoreDNS 增加配置 ReadinessProbe：

```
readinessProbe:
  failureThreshold: 5
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
  initialDelaySeconds: 30
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 5
```

配置 CoreDNS 使用 UDP 访问上游 DNS

当 CoreDNS 需要与上游 DNS Server 通信时，它将默认使用客户端请求的协议（UDP 或者 TCP），而 TKE 中 CoreDNS 的上游默认为 VPC 内的 DNS 服务，该服务对 TCP 的支持在性能上比较有限，因此推荐做如下配置，显式指定 UDP（尤其在安装了 NodeLocal DNSCache 时）：

```
        .:53 {
            forward . /etc/resolv.conf {
                prefer_udp
            }
        }
```

配置 CoreDNS 过滤 HINFO 请求

VPC 内的 DNS 服务不支持 HINFO 类型的 DNS 请求，因此推荐做如下配置，在 CoreDNS 侧过滤此类请求（尤其在安装了 NodeLocal DNSCache 时）：

```
        .:53 {
            template ANY HINFO . {
                rcode NXDOMAIN
            }
        }
```

配置 CoreDNS 对 IPv6 类型的 AAAA 记录查询返回域名不存在

当业务不需要做 IPv6 的域名解析时，可以通过该配置降低通信成本：

```
        .:53 {
            template ANY AAAA {
                rcode NXDOMAIN
            }
        }
```

> **注意：**
> IPv4/IPv6 双栈集群不能做此配置。

配置 CoreDNS 上游 DNS 服务地址

当前节点默认的 DNS 上游地址为腾讯云公共 DNS 服务（183.60.83.19，183.60.82.98）。当业务有自定义上游的场景时，可以通过配置指定，以配置 Google 公共 DNS 上游（8.8.8.8）为例。

```
        .:53 {
            forward . 8.8.8.8
        }
```

配置自定义域名解析

详情请参见 在 TKE 中实现自定义域名解析 。

手动升级

升级到1.7.0

1. 编辑 coredns configmap：

   ```
   kubectl edit cm coredns -n kube-system
   ```

   修改为以下内容：

   ```
       .:53 {
           template ANY HINFO . {
               rcode NXDOMAIN
   ```

```
        }
        errors
        health {
            lameduck 30s
        }
        ready
        kubernetes cluster.local. in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            prefer_udp
        }
        cache 30
        reload
        loadbalance
    }
```

2. 编辑 coredns deployment：

```
kubectl edit deployment coredns -n kube-system
```

替换镜像为：

```
image: ccr.ccs.tencentyun.com/tkeimages/coredns:1.7.0
```

升级到1.8.4

> ⚠ **注意：**
> 社区中 CoreDNS 1.7.0 版本是一个 backwards-incompatible release，在这个版本中删除了kubernetes插件的 **upstream** 和 **resyncperiod** 的option（详见社区 pr），如果用户需要从 1.7.0 以下的版本（例如 1.6.2 版本）升级到 1.8.4 版本，需要进行相关配置的兼容性修改。

CoreDNS 升级配置检查

CoreDNS 升级前，需要对 **corefile 配置**以及 CoreDNS 的 **deployment 配置**进行检查，详情如下。
corefile 配置检查：

- kubernetes 插件兼容性检查
  - 检查 kubernetes 插件是否设置了 upstream，如果设置 upstream，且未配置 server 参数，可以直接删除。

```
apiVersion: v1
data:
  Corefile: |2-
        .:53 {
            template ANY HINFO . {
                rcode NXDOMAIN
            }
            errors
            health {
                lameduck 30s
            }
            ready
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                fallthrough in-addr.arpa ip6.arpa
                upstream
            }
            prometheus :9153
            forward . /etc/resolv.conf {
                prefer_udp
            }
            cache 30
            reload
            loadbalance
        }
```

  - 如果有设置 upstream server 参数，需要将 upstream server 参数迁移到 forward 插件部分。

- 检查 kubernetes 插件是否设置resyncperiod，如果设置，直接删除。
- health 和 ready 插件检查
  - 检查 health 插件的 lameduck 参数配置时间不要超过 30s。
  - 检查是否配置 health 和 ready 插件，这两个插件是必选配置。

deployment 配置检查：

- 确认 coredns 的 deployment 中配置 livenessprobe，并和 corefile 文件的 health 端口配置保持一致。
- 确认 coredns 的 deployment 中配置 readinessprobe，并和 corefile 文件的 ready 端口配置保持一致。

CoreDNS 升级流程

完成 corefile 以及 deployment 配置检查后，可进行升级。

1. 编辑 coredns clusterrole：

```
kubectl edit clusterrole system:coredns
```

请在 1.6.2 版本 coredns 的 clusterrole 下增加以下内容：

```
- apiGroups:
  - discovery.k8s.io
  resources:
  - endpointslices
  verbs:
  - list
  - watch
```

旧版本1.6.2版本内容

```
rules:
- apiGroups:
  - '*'
  resources:
  - endpoints
  - services
  - pods
  - namespaces
  verbs:
  - list
  - watch
```

新版本1.8.4内容

```
rules:
- apiGroups:
  - '*'
  resources:
  - endpoints
  - services
  - pods
  - namespaces
  verbs:
  - list
  - watch
```

```
- apiGroups:
  - discovery.k8s.io
  resources:
  - endpointslices
  verbs:
  - list
  - watch
```

编辑 coredns configmap：

```
kubectl edit cm coredns -n kube-system
```

如果用户有自定义 corefile 配置，请按照**上述检查注意事项**修改，其中标准模板如下：

```
    .:53 {
        template ANY HINFO . {
            rcode NXDOMAIN
        }
        errors
        health {
            lameduck 30s
        }
        ready
        kubernetes cluster.local. in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            prefer_udp
        }
        cache 30
        reload
        loadbalance
    }
```

2. 编辑 coredns deployment

```
kubectl edit deployment coredns -n kube-system
```

替换镜像为升级镜像如下：

```
image: ccr.ccs.tencentyun.com/tkeimages/coredns:1.8.4
```

升级后检查

- 确保 coredns pod running 且处于 ready 状态：

```
kubectl get pod -n kube-system -l k8s-app=kube-dns -o wide
```

- 确保 coredns 无错误日志：

```
kubectl logs ${coredns-pod-name} -n kube-system | grep -i error
```

- 确保 coredns pod 可以正常解析内外部域名：

```
nslookup kubernetes.default.svc.cluster.local ${coredns-pod-ip}
nslookup www.qq.com ${coredns-pod-ip}
```

业务配置

除了 DNS 服务的最佳实践外，在业务侧，也可以做适当的优化配置，来提升 DNS 的使用体验。

1. 默认情况下，Kubernetes 集群中的域名解析往往需要经过多次请求才能解析到。查看 pod 内 的 `/etc/resolv.conf` 可以知道 `ndots` 选项默认为 5。例如，在 debug 命名空间查询 `kubernetes.default.svc.cluster.local` 这个 service：
   - 域名中有 4 个 `.` ，小于 5，尝试拼接上第一个 search 进行查询，即 `kubernetes.default.svc.cluster.local.debug.svc.cluster.local` ，查不到该域名。
   - 继续尝试 `kubernetes.default.svc.cluster.local.svc.cluster.local` ，查不到该域名。
   - 继续尝试 `kubernetes.default.svc.cluster.local.cluster.local` ，仍然查不到该域名。
   - 尝试不加后缀，即 `kubernetes.default.svc.cluster.local` ，查询成功，返回响应的 ClusterIP。
2. 上面一个简单的 service 域名解析需要经过 4 轮解析才能成功，集群中充斥着大量无用的 DNS 请求。因此需要根据业务配置的访问方式来为其设置合理的 ndots 来降低查询次数：

```
spec:
  dnsConfig:
    options:
    - name: ndots
      value: "2"
  containers:
  - image: nginx
    imagePullPolicy: IfNotPresent
    name: diagnosis
```

3. 同时，可以优化业务访问服务的域名配置：
   - Pod 访问本命名空间的 Service，使用 `<service-name>` 访问。
   - Pod 访问其它命名空间的 Service，使用 `<service-name>.<namespace-name>` 访问。
   - Pod 访问外部域名，使用 FQDN 类型域名访问，在域名最后添加 `.` 以减少无效搜索。
4. glibc 的 resolver 库访问一个 name server 的超时时间默认为5秒，针对 /etc/resolv.conf 中列出的一组 name server，默认最多尝试（attempts）2次，如 /etc/resolv.conf 中配置两个 name server，当所有 name server 都不可用时，总超时时间为20秒，然而，这对于许多业务来说过于保守。可以根据业务实际需要，为 Pod 设置合理的 DNS 超时配置，以降低超时时间，避免 DNS 服务短时不可用导致业务吞吐量的显著下降，以下是一个示例：

```
spec:
  dnsConfig:
    options:
    - name: timeout
      value: "1"
    - name: attempts
      value: "2"
  containers:
  - image: nginx
    imagePullPolicy: IfNotPresent
    name: diagnosis
```

相关内容

配置介绍

- **errors**
  输出错误信息。
- **health**
  上报健康状态，用于配置健康检查，如 `livenessProbe` ，默认监听 8080 端口，路径为 `http://localhost:8080/health` 。

> ⚠ **注意：**
> 如果有多 Server 块，health 只能配置一次，或者配置在不同端口。

```
com {
  whoami
  health :8080
}

net {
  erratic
  health :8081
}
```

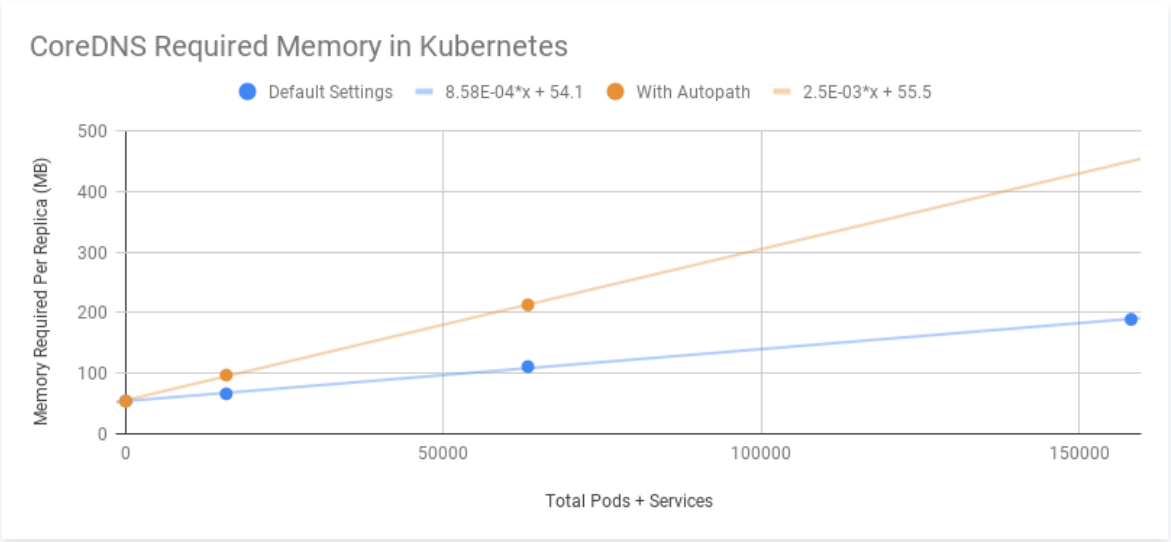
- **lameduck**
  用于配置优雅退出的时间，实现方式是 hook 在 CoreDNS 收到退出信号时，在其中执行 sleep，以保证时限内可以继续提供服务。
- **ready**
  上报插件状态，用于配置服务就绪检查，如 `readinessProbe`，默认监听 8181 端口，路径为 `http://localhost:8181/ready`。
- **kubernetes**
  Kubernetes 插件，支持集群内服务解析。
- **prometheus**
  metrics 数据接口，用于获取监控数据，路径为 `http://localhost:9153/metrics`。
- **forward（proxy）**
  将无法处理的请求转发到上游 DNS 服务器。默认使用宿主机的 `/etc/resolv.conf` 配置。
  - 根据 forward aaa bbb 的配置，内部会维护一个 udns 的列表 [aaa,bbb]
  - 当有请求到来时，根据预设的策略（random|round_robin|sequential，默认 random）在列表 [aaa,bbb] 中找一个 udns 发请求，如果失败，则找出下一个 udns 进行尝试，同时针对失败的 udns 启动周期性的健康监测，直到其变为健康，停止健康监测。
  - 在健康监测的过程中，如果连续几次（默认两次）监测失败，则将该 udns 状态置为 down，后面从列表中选 udns 时将跳过状态为 down 的 udns。
  - 当所有的 udns 都 down 时，随机选一个 udns 做转发。
    因此，可以认为 coredns 有在多个 upstream 间智能切换的能力，forward 列表里只要有一个可用的 udns，则请求可以成功。
- **cache**
  DNS 缓存。
- **reload**
  热加载 Corefile，修改 ConfigMap 后，会在两分钟内加载新配置。
- **loadbalance**
  提供基于 DNS 的负载均衡功能，随机响应记录的顺序。

CoreDNS 资源占用

内存

- 主要取决于集群内 Pod 数和 Service 数。
- 受打开缓存大小的影响。
- 受 QPS 的影响。

以下数据来自于 CoreDNS 官方：
MB required (default settings) = (Pods + Services) / 1000 + 54



CPU

主要受 QPS 的影响。
以下数据来自于 CoreDNS 官方：
单副本 CoreDNS，运行节点规格：2 vCPUs, 7.5 GB memory。

Query Type	QPS	Avg Latency (ms)	Memory Delta (MB)
external	6733	12.02	+5
internal	33669	2.608	+5

CoreDNS 日志仪表盘使用指南

最近更新时间：2024-09-24 11:34:32

TKE 容器服务部署了 CoreDNS 以提供集群内的域名服务解析。由于网络故障或者 CoreDNS 负载压力过大等多种原因，可能会出现 DNS 请求异常、请求延迟高以及多副本 CoreDNS 请求不均衡等多种问题，从而影响用户正常业务的 DNS 请求。为了快速排查 DNS 异常，发现潜在的业务和安全隐患，TKE 基于 CoreDNS 的 log 插件和 CLS 日志平台构建了全面的 CoreDNS 日志能力。本文将指导您如何在 TKE 集群中启用 CoreDNS 日志，并利用相应的仪表盘功能进行问题排查。

前置条件

1. 集群需要开启日志服务。
2. CoreDNS 相关的 Corefile 配置中需要添加 log 插件。

> **说明：**
> 请按照下面的方式添加 Corefile 配置中的 log 插件，编辑 kube-system 下的 coredns 名称的 configmap。
>
> ```
> data:
>  Corefile: |2-
>        .:53 {
>            template ANY HINFO . {
>                rcode NXDOMAIN
>            }
>            log  #请在此处添加 log 插件
>            errors
>            health {
>                lameduck 30s
>            }
>            ready
>            kubernetes cluster.local. in-addr.arpa ip6.arpa {
>                pods insecure
>                fallthrough in-addr.arpa ip6.arpa
>            }
>            prometheus :9153
>            forward . /etc/resolv.conf {
>                prefer_udp
>            }
>            cache 30
>            reload
>            loadbalance
>        }
> kind: ConfigMap
> ```
>
> 保存配置后退出，Corefile 会被自动 reload；如果 Corefile 没有配置 reload，需要重建 CoreDNS 以使配置生效。

3. 确保集群 CoreDNS 版本 >= 1.8.4。如果需要升级 CoreDNS 版本到 1.8.4，请参见 升级到 1.8.4 。

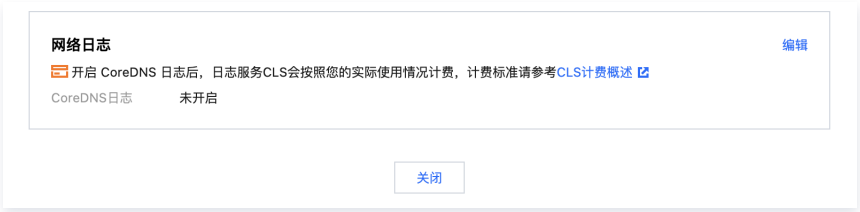
开启 CoreDNS 日志

1. 登录 容器服务控制台 ，选择左侧导航栏中的**运维功能管理**。
2. 选择您需要开启 CoreDNS 日志的集群，单击集群右侧的设置。如下图所示：

3. 在**设置功能**页面，单击日志采集右侧的**编辑**。
4. 勾选**开启日志采集**，单击**确定**。如下图所示：

> **注意：**
> 未执行 前置条件 第2步，将无法进行开启操作。

5. 单击**网络日志**右侧的**编辑**。如下图所示：



6. 勾选**开启 CoreDNS 日志**，并输入以下信息：

   - 日志所在地域：选择 cls 日志集存储地域。
   - 日志集：选择 cls 日志集名称。如果没有合适的日志集，可以**新建日志集**。
   - 日志主题：可以选择自动创建日志主题，或者选择已有的日志主题。
7. 单击**确定**，完成 CoreDNS 日志开启。

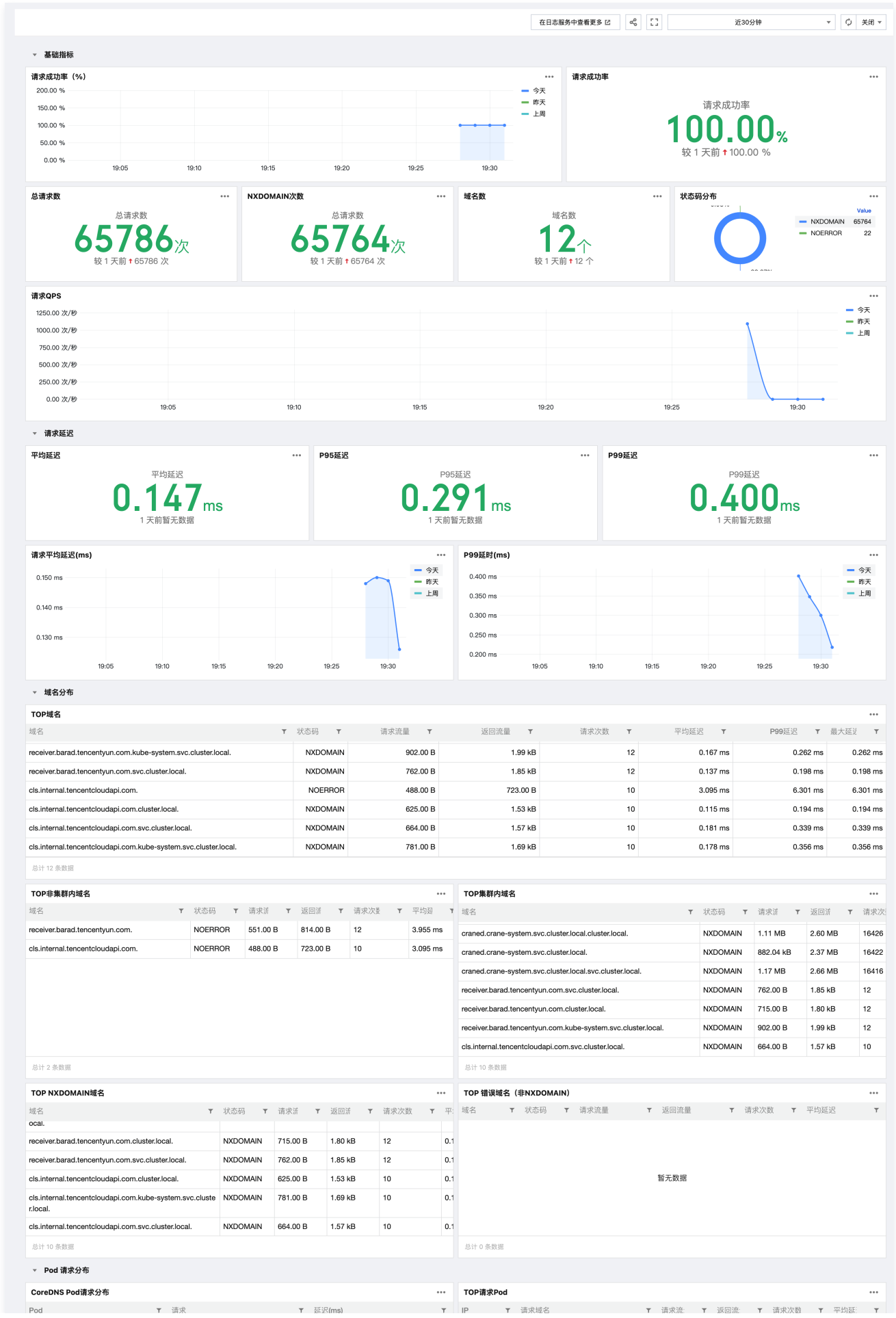
单击日志主题链接，进入 CLS 页面进行日志检索等操作。日志索引字段含义如下：

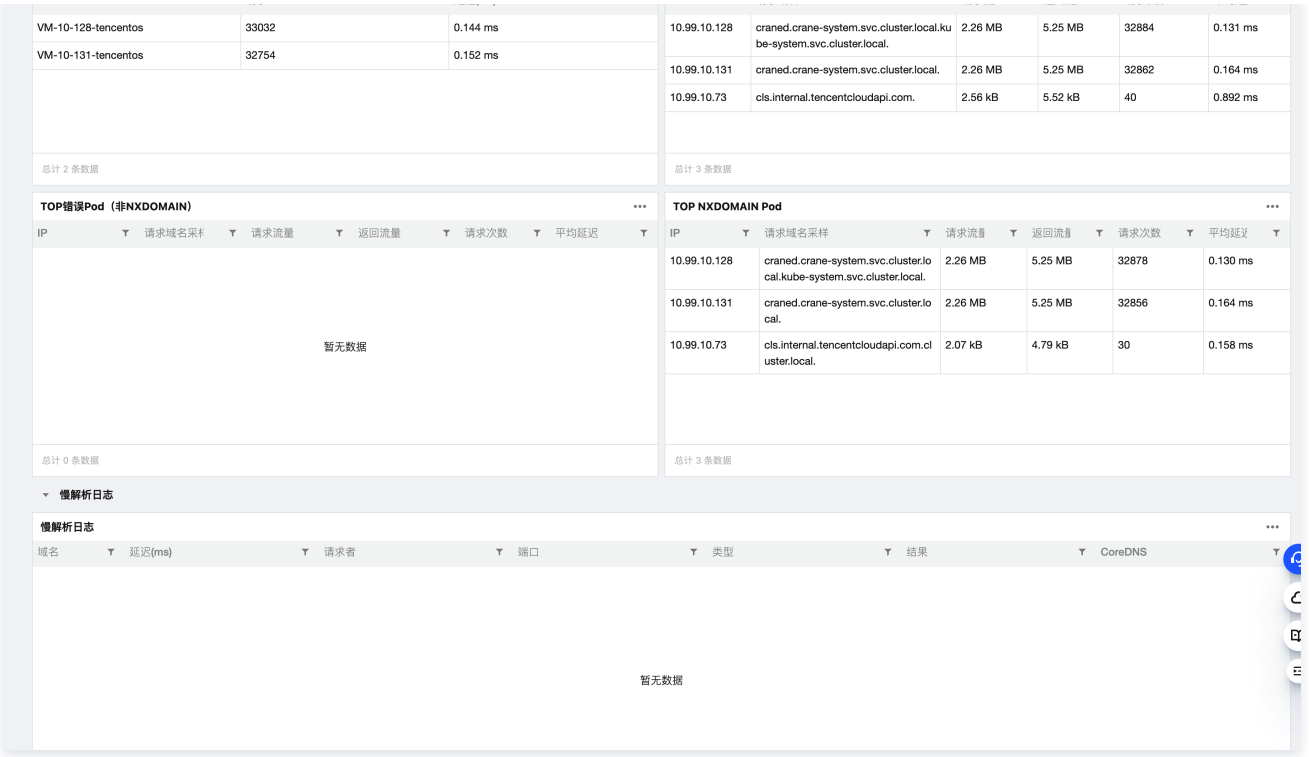
字段名称	字段含义	示例
class	请求类别。	IN
do	查询中是否设置了 "DNSSEC OK"（DNS 安全扩展确认）。	false
duration	响应时间（单位秒）。	0.000098921
id	请求 ID，标识特定的 DNS 请求和响应。	30008
level	日志级别。	INFO
name	DNS 请求中查询的目标域名。	craned.crane-system.svc.cluster.local.
port	发送 DNS 请求的客户端端口。	50424
proto	使用协议。	udp
rcode	响应代码。	NXDOMAIN
remote	客户端 IP 地址。	10.99.10.128
rflags	响应报文中的标志字段，用于表示 DNS 查询的状态和结果。	qr,aa,rd
rsize	限制 DNS 响应的最大值。	162
size	限制 DNS 请求的最大值。	69
bufsize	DNS 请求和响应的内部缓冲区大小。	65535
type	请求类型。	A

在日志管理中使用 CoreDNS 仪表盘

1. 登录 容器服务控制台，选择左侧导航栏中的**日志管理 > CoreDNS 日志**。
2. 进入 CoreDNS 日志页面，选择地域，集群类型，和您需要查看的集群。如下图所示：

3. 查看仪表盘数据。如下图所示：


在日志服务中查看更多
近30分钟
关闭
基础指标
请求成功率（%）
200.00 %
150.00 %
100.00 %
50.00 %
0.00 %
19:05
19:10
19:15
19:20
19:25
19:30
今天
昨天
上周
请求成功率
请求成功率
100.00%
较 1 天前 ↑ 100.00 %
总请求数
总请求数
65786次
较 1 天前 ↑ 65786 次
NXDOMAIN次数
总请求数
65764次
较 1 天前 ↑ 65764 次
域名数
域名数
12个
较 1 天前 ↑ 12 个
状态码分布
Value
NXDOMAIN 65764
NOERROR 22
请求QPS
1250.00 次/秒
1000.00 次/秒
750.00 次/秒
500.00 次/秒
250.00 次/秒
0.00 次/秒
请求延迟
平均延迟
平均延迟
0.147ms
1 天前暂无数据
P95延迟
P95延迟
0.291ms
1 天前暂无数据
P99延迟
P99延迟
0.400ms
1 天前暂无数据
请求平均延迟(ms)
0.150 ms
0.140 ms
0.130 ms
P99延时(ms)
0.400 ms
0.350 ms
0.300 ms
0.250 ms
0.200 ms
域名分布
TOP域名
域名 状态码 请求流量 返回流量 请求次数 平均延迟 P99延迟 最大延迟
receiver.barad.tencentyun.com.kube-system.svc.cluster.local. NXDOMAIN 902.00 B 1.99 kB 12 0.167 ms 0.262 ms 0.262 ms
receiver.barad.tencentyun.com.svc.cluster.local. NXDOMAIN 762.00 B 1.85 kB 12 0.137 ms 0.198 ms 0.198 ms
cls.internal.tencentcloudapi.com. NOERROR 488.00 B 723.00 B 10 3.095 ms 6.301 ms 6.301 ms
cls.internal.tencentcloudapi.com.cluster.local. NXDOMAIN 625.00 B 1.53 kB 10 0.115 ms 0.194 ms 0.194 ms
cls.internal.tencentcloudapi.com.svc.cluster.local. NXDOMAIN 664.00 B 1.57 kB 10 0.181 ms 0.339 ms 0.339 ms
cls.internal.tencentcloudapi.com.kube-system.svc.cluster.local. NXDOMAIN 781.00 B 1.69 kB 10 0.178 ms 0.356 ms 0.356 ms
总计 12 条数据
TOP非集群内域名
域名 状态码 请求流 返回流 请求次数 平均延
receiver.barad.tencentyun.com. NOERROR 551.00 B 814.00 B 12 3.955 ms
cls.internal.tencentcloudapi.com. NOERROR 488.00 B 723.00 B 10 3.095 ms
总计 2 条数据
TOP集群内域名
域名 状态码 请求流 返回流 请求次
craned.crane-system.svc.cluster.local.cluster.local. NXDOMAIN 1.11 MB 2.60 MB 16426
craned.crane-system.svc.cluster.local. NXDOMAIN 882.04 kB 2.37 MB 16422
craned.crane-system.svc.cluster.local.svc.cluster.local. NXDOMAIN 1.17 MB 2.66 MB 16416
receiver.barad.tencentyun.com.svc.cluster.local. NXDOMAIN 762.00 B 1.85 kB 12
receiver.barad.tencentyun.com.cluster.local. NXDOMAIN 715.00 B 1.80 kB 12
receiver.barad.tencentyun.com.kube-system.svc.cluster.local. NXDOMAIN 902.00 B 1.99 kB 12
cls.internal.tencentcloudapi.com.svc.cluster.local. NXDOMAIN 664.00 B 1.57 kB 10
总计 10 条数据
TOP NXDOMAIN域名
域名 状态码 请求流 返回流 请求次数 平
ocal.
receiver.barad.tencentyun.com.cluster.local. NXDOMAIN 715.00 B 1.80 kB 12 0.1
receiver.barad.tencentyun.com.svc.cluster.local. NXDOMAIN 762.00 B 1.85 kB 12 0.1
cls.internal.tencentcloudapi.com.cluster.local. NXDOMAIN 625.00 B 1.53 kB 10 0.1
cls.internal.tencentcloudapi.com.kube-system.svc.cluster.local. NXDOMAIN 781.00 B 1.69 kB 10 0.1
cls.internal.tencentcloudapi.com.svc.cluster.local. NXDOMAIN 664.00 B 1.57 kB 10 0.1
总计 10 条数据
TOP 错误域名（非NXDOMAIN）
域名 状态码 请求流量 返回流量 请求次数 平均延迟
暂无数据
总计 0 条数据
Pod 请求分布
CoreDNS Pod请求分布
Pod 请求 延迟(ms)
TOP请求Pod
IP 请求域名 请求流 返回流 请求次数 平均延



- **请求成功率**：统计所有 DNS 响应正常数目（NOERROR 和 NXDOMAIN）占总请求的比例。用户可以根据此指标，发现当前 CoreDNS 是否存在解析失败的情况。
- **域名数**：当前 CoreDNS 服务中响应的域名总数。
- **请求 QPS**：反应一段时间周期内，CoreDNS 服务的 QPS 性能情况（请求次/秒）。用户可以根据此时序图，定位 CoreDNS 相应的性能问题。
- **平均延迟/P95 延迟/P99 延迟**：通过最近1w个请求的延迟，反应 CoreDNS 服务的平均延迟/P95和 P99 延迟，用于定位 CoreDNS 响应慢的问题。
- **CoreDNS Pod 请求分布**：多副本 CoreDNS 的情况下，此表格可以展示每个副本的请求分布数以及平均延迟，用于定位 CoreDNS 副本请求不均的问题。
- **慢解析日志**：当 DNS 请求的处理时间超过特定阈值时，CoreDNS 会在慢解析日志中记录相关信息。通过分析慢解析日志，用户可以发现哪些类型的请求最耗时，然后针对这些问题进行优化。

关闭 CoreDNS 日志

如果您不再需要 CoreDNS 日志采集，可以通过下面的方式关闭 CoreDNS 日志采集能力：

1. 登录 容器服务控制台，选择左侧导航栏中的**运维功能管理**。
2. 选择您需要关闭 CoreDNS 日志的集群，单击集群右侧的**设置**。
3. 在**设置功能**页面，单击**网络日志**右侧的**编辑**。如下图所示：

4. 取消选择**开启 CoreDNS 日志**。如下图所示：

5. **单击确定。如果是自动创建的日志主题，这里会提示有关联的日志主题，如果您不再需要该日志主题，请单击跳转到 CLS 控制台删除相应的日志主题。否则关联的日志主题会一直保存，产生相应的计费。**

在 TKE 集群中使用 NodeLocal DNS Cache

最近更新时间：2025-03-27 19:50:53

应用场景

在用户业务采用 Kubernetes 标准服务发现机制的情况下，若 CoreDNS 请求的 QPS 过高，可能导致 DNS 查询延迟增加和负载不均，从而对业务性能和稳定性产生不利影响。
针对这种场景，可以通过部署 NodeLocal DNS Cache，降低 CoreDNS 请求压力，提升集群内 DNS 解析性能以及稳定性。本文将详细介绍如何在 TKE 集群安装并使用 NodeLocal DNS Cache。

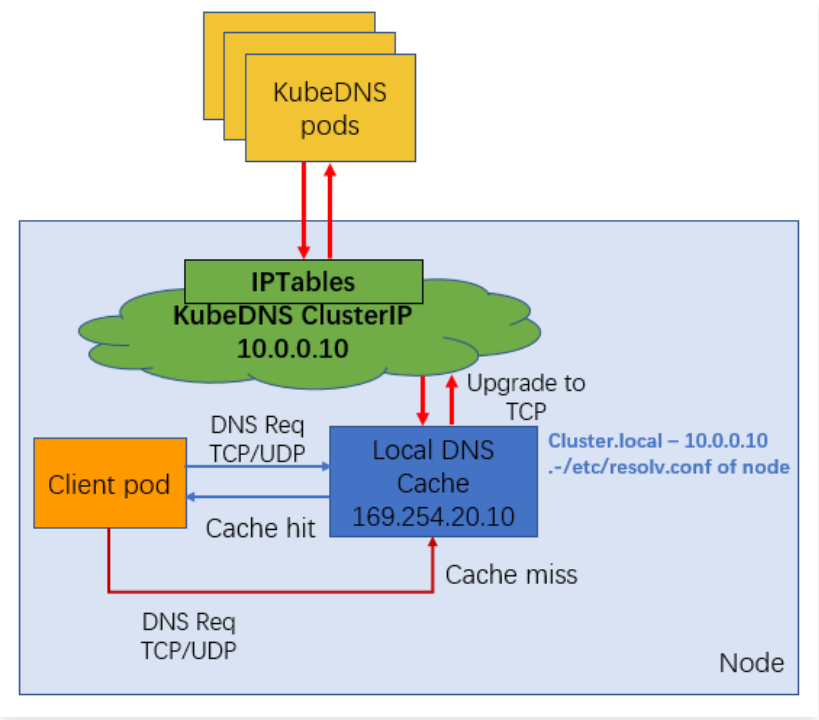
使用限制

- 暂不支持部署在超级节点上的 Pod。
- 暂不支持网络模式采用 Cilium Overlay，以及独立网卡模式的 Pod。
- NodeLocal DNS Cache 当前仅作为 CoreDNS 的缓存代理使用，不支持配置其他插件。如果有需要，请直接配置 CoreDNS。

原理介绍

社区方案

社区版本 NodeLocal DNS Cache 通过 DaemonSet 在集群的每个节点上部署一个 hostNetwork 的 Pod，该 Pod 名称为 node-local-dns，可以缓存本节点上 Pod 的 DNS 请求。如果存在 cache misses，该 Pod 将会通过 TCP 链接请求上游 kube-dns 服务进行获取。原理图如下所示：

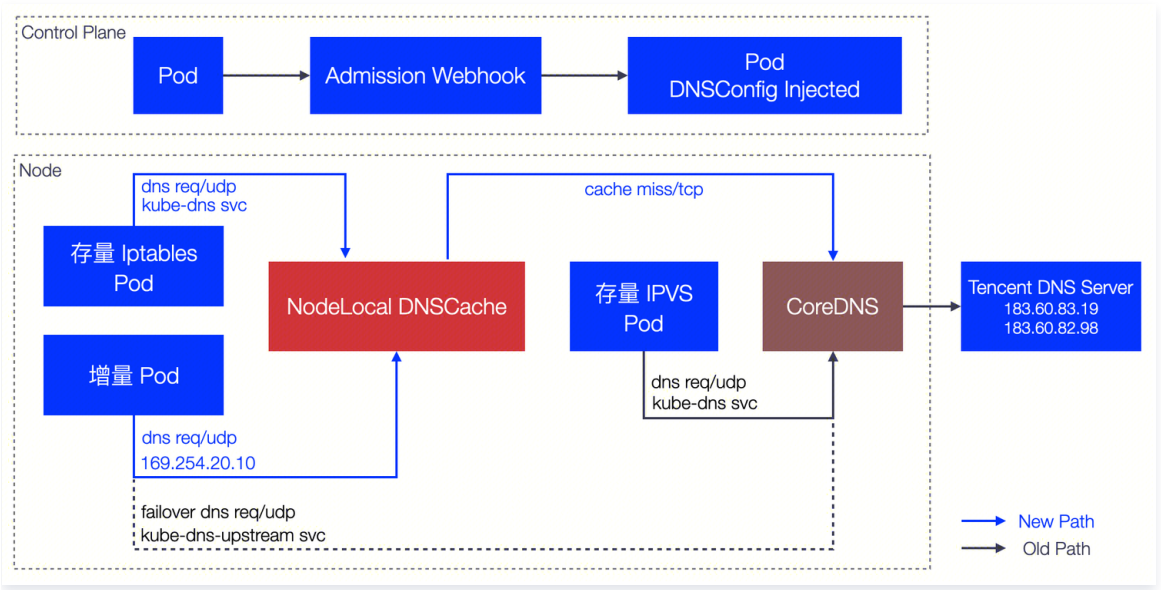


在 kube-proxy 采用不同的转发模式下，支持效果有差异。

- iptables 模式下，部署 NodeLocal DNS Cache 后，存量 Pod 和增量 Pod 均可以无感自动切换访问本地 DNS Cache。
- ipvs 模式下，增量和存量 Pod 均无法实现 DNS Cache 的无感切换。如果在 ipvs 模式下想使用 NodeLocal DNS Cache 服务，可以采用以下两种方式：
  - 方式1：修改 kubelet 参数 `--cluster-dns`，指向 `169.254.20.10`，然后重启 kubelet 服务。此操作方式存在业务中断的风险。
  - 方式2：修改 Pod 的 DNSConfig，指向新的 `169.254.20.10` 地址，使用本地的 DNS Cache 处理 DNS 解析。

TKE 中 NodeLocal DNS Cache 方案

TKE 上的 NodeLocal DNS Cache 方案对社区版本在 ipvs 模式下的缺陷进行了增强，针对增量 Pod 会自动配置 DNSConfig，具备本地 DNS 缓存能力。**不过当前仍然无法支持存量 Pod 自动切换，需要用户显式操作（重建 Pod 或手动配置 DNSConfig）。**
工作原理：



控制台安装 NodeLocal DNS Cache

您可以通过 TKE 的组件管理部署安装 Nodelocal DNS Cache，详情请参见 操作步骤 。安装完成后，返回**组件管理**列表页，检查 localdns 组件状态置为**成功**，如下图所示：

使用 NodeLocal DNS Cache

iptables 集群和 ipvs 集群下，对 NodeLocal DNS Cache 的使用方式不同，具体描述如下：

iptables 集群

- **存量 Pod**：用户无需任何操作，存量 Pod 可以直接使用本地 DNS Cache 能力解析 DNS 请求。
- **增量 Pod**：用户无需任何操作，新建 Pod 可以直接使用本地 DNS Cache 能力解析 DNS 请求。

ipvs 集群

针对 ipvs 集群，TKE 会将 DNSConfig 配置动态注入到新建的 Pod 中，同时会将 dnsPolicy 配置为 None，避免手动配置 Pod YAML。自动注入的配置如下：

```
dnsConfig:
  nameservers:
  - 169.254.20.10
  - 10.23.1.234
  options:
  - name: ndots
```

```
      value: "3"
    - name: attempts
      value: "2"
    - name: timeout
      value: "1"
  searches:
  - <pod所在命名空间>.svc.cluster.local
  - svc.cluster.local
  - cluster.local
dnsPolicy: None
```

注意：

如果您需要相应的 Pod 能够自动注入 DNSConfig，请确保满足以下条件：

1. 请在 Pod 所在的命名空间打上 Label 标签：`localdns-injector=enabled`。

   例如：如果您需要 default 命名空间中的新建 Pod 自动注入 DNSConfig，请配置：

   ```
   kubectl label namespace default localdns-injector=enabled
   ```

2. 保证 Pod 不在 **kube-system** 和 **kube-public** 命名空间，这两个命名空间下的 Pod 不会自动注入 DNSConfig。
3. 保证 Pod label 不包含 `localdns-injector=disabled`，包含此 label 的 Pod 不会被注入 DNSConfig。
4. 新建 Pod 网络配置非 hostNetwork，需要配置 DNSPolicy 为 ClusterFirst；如果 Pod 网络为 hostNetwork，需要配置 DNSPolicy 为 ClusterFirstWithHostNet。
5. 暂不支持 GR 网络模式。

- **存量 Pod**：存量 Pod 暂时无法做到无感切换。如果需要存量 Pod 使用本地 DNS Cache 代理能力，用户需要重建 Pod。重建后，Pod 会自动注入 DNSConfig，从而使用本地 DNS Cache 能力解析 DNS 请求。
- **增量 Pod**：当满足上述注意事项后，新建 Pod 会自动注入 DNSConfig 配置，访问本节点 `169.254.20.10:53`，使用本地 DNS Cache 能力解析 DNS 请求。

ipvs 集群的注意事项

在 Kubernetes 集群使用 IPVS kube-proxy 模式的网络环境下，由于 IPVS 的四层负载均衡机制特性限制，目前缺乏从 CoreDNS 到 NodeLocal DNS 的透明劫持方案，所以在此场景下必须通过 PodSpec 中的 DNSConfig 显式配置 DNS 服务器列表（不管是自动注入还是用户主动配置）。为了优先使用 NodeLocalDNS，并在其故障时降级到备用 CoreDNS 服务节点。目前推荐的配置方案（包括自动注入）中的 servers 列表如下。

```
dnsConfig:
  servers:
    - "169.254.20.10"  # NodeLocal DNS虚拟IP
    - "10.23.1.234"    # CoreDNS ClusterIP（每个集群不同）
```

客户端解析逻辑。标准 DNS 客户端库（如 glibc）默认采用顺序查询机制。优先向 dnsConfig.servers 列表中的首个 DNS 服务器发起请求。仅在首节点响应超时或返回 SERVFAIL 时，自动切换至列表中的后续服务器。
异常场景下的影响。当 NodeLocal DNS 实例发生故障时，以示例配置为例。客户端将经历单次 DNS 查询超时 1s，然后超时后自动降级到备用 CoreDNS 服务节点。极端情况下可能导致最大延迟接近单个查询超时周期。

验证 NodeLocal DNS Cache

NodeLocal DNS Cache 成功开启后，可以在节点上验证 Pod 访问 CoreDNS 服务是否通过了本地 DNS Cache 进行解析。以下是分别验证 iptables 集群和 ipvs 集群的 NodeLocal DNS Cache 开启效果的方法。

说明：

如果您想通过日志验证节点上 NodeLocal DNS Cache 是否代理了本节点的 DNS 请求，需要修改 kube-system 命名空间下 node-local-dns 的 ConfigMap 配置，在对应的 Corefile 配置中添加 log 日志能力。如下图所示：

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  Corefile: |
    cluster.local:53 {
        errors
        log
        cache {
                success 9984 30
                denial 9984 5
        }
        reload
        loop
        bind __PILLAR__LOCAL__DNS__
```

iptables 集群验证

在 iptables 集群中，需要验证存量 Pod 以及增量 Pod 是否可以自动通过本地 NodeLocal DNS Cache 代理 Pod 的 DNS 请求。

存量 Pod

1. 登录存量 Pod。
2. 使用 nslookup 命令解析 kube-dns 的 svc。如下图所示：

```
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] i
nstead.
/ # nslookup kube-dns.kube-system.svc
nslookup: can't resolve '(null)': Name does not resolve

Name:      kube-dns.kube-system.svc
Address 1: 10.23.1.110 kube-dns.kube-system.svc.cluster.local
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 3a:4c:b7:ad:1c:5b brd ff:ff:ff:ff:ff:ff
    inet 10.99.21.3/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2402:4e00:1207:5916::9b8f:90ce:846a/128 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::384c:b7ff:fead:1c5b/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

3. 检查本节点上 node-cache Pod 日志。如下图所示：

```
[INFO] 10.99.21.3:50709 - 62853 "A IN kube-dns.kube-system.svc.default.svc.cluster.local. udp 68 false 512" NXDOMAIN qr,a
a,rd 161 0.002424868s
[INFO] 10.99.21.3:50709 - 63675 "AAAA IN kube-dns.kube-system.svc.default.svc.cluster.local. udp 68 false 512" NXDOMAIN q
r,aa,rd 161 0.0037218s
[INFO] 10.99.21.3:48989 - 38909 "A IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NXDOMAIN qr,aa,rd 153
 0.00215277s
[INFO] 10.99.21.3:48989 - 39939 "AAAA IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NXDOMAIN qr,aa,rd
153 0.003375326s
[INFO] 10.99.21.3:49403 - 2811 "A IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR qr,aa,rd 110 0.002
135301s
[INFO] 10.99.21.3:49403 - 3664 "AAAA IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR qr,aa,rd 149 0.
```

可以确认存量 Pod 对 kube-dns 的解析请求通过了本节点上的 NodeLocal DNS Cache 服务。

增量 Pod

1. 登录新建 Pod。
2. 使用 nslookup 命令解析 kube-dns 的 svc。如下图所示：

```
/ # nslookup kube-dns.kube-system.svc
nslookup: can't resolve '(null)': Name does not resolve

Name:      kube-dns.kube-system.svc
Address 1: 10.23.1.110 kube-dns.kube-system.svc.cluster.local
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether fa:e9:bd:9e:f3:df brd ff:ff:ff:ff:ff:ff
    inet 10.99.10.29/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2402:4e00:1207:5958::9b67:446c:2ab9/128 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::f8e9:bdff:fe9e:f3df/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

3. 检查本节点上 node-cache Pod 日志。如下图所示：

```
[INFO] 10.99.10.29:35234 - 52850 "AAAA IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NXDOMAIN qr,aa,rd
 153 0.000454866s
[INFO] 10.99.10.29:53379 - 11046 "AAAA IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR qr,aa,rd 149
0.000248217s
[INFO] 10.99.10.29:53379 - 10135 "A IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR qr,aa,rd 110 0.0
002900195
[INFO] 10.99.10.29:35234 - 52022 "A IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NXDOMAIN qr,aa,rd 15
3 0.009849625s
```

可以确认新增 Pod 对 kube-dns 的解析请求通过了本节点上的 NodeLocal DNS Cache 服务。

ipvs 集群验证

在 ipvs 集群中，存量 Pod 暂时无法自动切换使用本地 DNS Cache，需要验证增量 Pod 是否可以自动通过本地 NodeLocal DNS Cache 代理 Pod 的 DNS 请求。操作步骤如下：

1. 将需要的命名空间添加 label：`localdns-injector=enabled`
2. 在需要的命名空间中，新建 Pod，确认 Pod 注入了 DNSConfig 配置。如下图所示：

```
  dnsConfig:
    nameservers:
    - 169.254.20.10
    - 10.0.3.209
    options:
    - name: ndots
      value: "3"
    - name: attempts
      value: "2"
    - name: timeout
      value: "1"
    searches:
    - dodia.svc.cluster.local
    - svc.cluster.local
    - cluster.local
  dnsPolicy: None
```

3. 登录新建 Pod。
4. 使用 nslookup 命令解析 kube-dns 的 svc。如下图所示：

```
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] i
nstead.
/ # nslookup kube-dns.kube-system.svc
nslookup: can't resolve '(null)': Name does not resolve

Name:      kube-dns.kube-system.svc
Address 1: 10.0.3.106 kube-dns.kube-system.svc.cluster.local
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether a2:ea:db:b2:ed:08 brd ff:ff:ff:ff:ff:ff
    inet 10.99.10.2/32 scope global eth0
       valid_lft forever preferred_lft forever
/ #
```

5. 检查本节点上 node-cache Pod 日志。如下图所示：

```
[INFO] 10.99.10.2:50375 - 45281 "AAAA IN kube-dns.kube-system.svc.dodia.svc.cluster.local. udp 66 false 512" NXDOMAIN qr,
aa,rd 159 0.000953497s
[INFO] 10.99.10.2:50375 - 44477 "A IN kube-dns.kube-system.svc.dodia.svc.cluster.local. udp 66 false 512" NXDOMAIN qr,aa,
rd 159 0.001436562s
[INFO] 10.99.10.2:50240 - 24282 "A IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NXDOMAIN qr,aa,rd 153
 0.000706419s
[INFO] 10.99.10.2:50240 - 25235 "AAAA IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NXDOMAIN qr,aa,rd
153 0.000955811s
[INFO] 10.99.10.2:35072 - 55625 "A IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR qr,aa,rd 110 0.00
0513704s
[INFO] 10.99.10.2:35072 - 56505 "AAAA IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR qr,aa,rd 149 0
.00060428s
```

可以确认 ipvs 集群中新增 Pod 对 kube-dns 的解析请求通过了本节点上的 NodeLocal DNS Cache 服务。

卸载 NodeLocal DNS Cache

前置检查

1. 在 NodeLocal DNS 卸载场景下，由于原节点级 DNS 代理流量将直接回退至 CoreDNS 服务端，存在引发 CoreDNS 级联故障的风险。所以需要对 CoreDNS 进行提前扩容，同时尽量在业务低峰期进行 NodeLocalDNS 的卸载操作。
2. 在 IPVS 网络模式下，在卸载之前有一些额外的检查项需要进行。
   2.1 禁用自动注入机制。清理所有命名空间和工作负载的配置注入标签：localdns-injector=disabled。确保后续新增工作负载不被自动注入 DNSConfig。
   2.2 存量工作负载DNS配置清理。遍历检查所有 Deployment/StatefulSet/DaemonSet 等控制器配置，确保 spec.template.spec.dnsConfig.servers 字段中移除 169.254.20.10 地址。验证 Pod 实例的 /etc/resolv.conf 文件不再包含 NodeLocal DNS 虚拟 IP。
   2.3 如果存在修改配置 kubelet 的启动参数 --cluster-dns 的情况。则需要更新 kubelet 将 DNS Server 重新制定 CoreDNS 的 ClusterIP 作为集群默认 DNS 服务地址。

控制台卸载

1. 登录 容器服务控制台，在左侧导航栏中选择**集群**。
2. 在集群列表中，单击目标集群 ID，进入集群详情页。
3. 选择左侧菜单栏中的**组件管理**，在组件管理页面单击需要删除组件所在行右侧的**删除**，如下图所示：

相关问题

关于 prefer_udp 相关配置

问题描述

在 TKE 集群中，CoreDNS 使用腾讯云默认的 DNS 服务（183.60.83.19/183.60.82.98）作为上游 DNS。腾讯云默认的 DNS 服务支持在 VPC 内进行私有域解析的 DNS 请求，但目前仅支持 UDP 协议，不支持 TCP 协议。然而，NodeLocal DNS 默认会通过 TCP 方式连接到 CoreDNS。如果 CoreDNS 未配置 prefer_udp，它将默认通过 TCP 方式访问上游的腾讯云默认 DNS 服务，这将导致一定几率的域名解析失败。

解决方案

1. **新创建的 TKE 集群**：CoreDNS 已经默认配置了 prefer_udp，用户无需处理。
2. **存量集群**：如果用户已经部署了 NodeLocal DNS Cache 组件，建议用户配置 CoreDNS 服务的相关 Corefile，添加 prefer_udp 并 reload 配置。示例如下：

```
apiVersion: v1
data:
  Corefile: |2-
        .:53 {
            template ANY HINFO . {
                rcode NXDOMAIN
            }
            log
            errors
            health {
                lameduck 30s
            }
            ready
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                fallthrough in-addr.arpa ip6.arpa
            }
            prometheus :9153
            forward . /etc/resolv.conf {
                prefer_udp
            }
            cache 30
            reload
            loadbalance
        }
kind: ConfigMap
```

3. **未安装 NodeLocal DNS Cache 的集群**：在安装 NodeLocal DNS Cache 组件时，会强制判断 CoreDNS Corefile 是否添加了 prefer_udp 配置。用户需要手动配置后，才能继续安装 NodeLocal DNS Cache 组件。

关于 kube-proxy 版本适配问题

问题描述

TKE 集群中低版本的 kube-proxy 存在 iptables（legacy/nftable）多后端问题，触发条件如下：

1. 集群 kube-proxy 代理模式为 iptables。
2. 对应不同版本的 k8s 集群，集群中的 kube-proxy 版本小于下面的版本号。

TKE 集群版本	问题修复版本
1.24	升级 kube-proxy 到 v1.24.4-tke.5 及以上
1.22	升级 kube-proxy 到 v1.22.5-tke.11 及以上
1.20	升级 kube-proxy 到 v1.20.6-tke.31 及以上
1.18	升级 kube-proxy 到 v1.18.4-tke.35 及以上
1.16	升级 kube-proxy 到 v1.16.3-tke.34 及以上
1.14	升级 kube-proxy 到 v1.14.3-tke.28 及以上
1.12	升级 kube-proxy 到 v1.12.4-tke.32 及以上
1.10	升级 kube-proxy 到 v1.10.5-tke.20 及以上

此时，如果客户部署了 NodeLocal DNS Cache 组件，会概率性触发多后端问题，导致集群内 service 服务无法正常访问。

解决方案

1. 如果用户现有集群配置符合上述触发条件，建议用户将 kube-proxy 版本升级到最新版本。
2. 当前 TKE 集群安装 NodeLocal DNS Cache 组件时，会对集群的 kube-proxy 进行判断，如果版本不符合条件，会禁止用户安装组件。此时请主动升级 kube-proxy 版本到最新版本。

具体 kube-proxy 最新版本请参见 TKE Kubernetes Revision 版本历史 。

关于操作系统版本适配问题

问题描述

NodeLocalDNS 会通过 hostNetwork 的方式部署，在节点创建 dummy 网卡，并绑定固定 IP（169.254.20.10）。NodeLocalDNS 服务进程会直接监听该 dummy 网卡 IP 的53端口，对外提供服务。但是在 Ubuntu 20.04系统中，系统服务 (named) 会占用所有网卡的53端口，导致 NodeLocalDNS 的服务进程因端口占用问题而无法启动。

问题解决方案

1. 直接通过 `sudo service named stop` 关闭 named 系统服务或者通过 sudo apt purge bind9 移除并通过 systemctl status named 来确保 named 服务被关闭。
2. 目前 Ubuntu Server 20.04.1 LTS 64bit | img-22trbn9x 存在该问题，您可以使用其他镜像来规避该问题。

在 TKE 中实现自定义域名解析

最近更新时间：2023-05-17 15:41:03

操作场景

在使用容器服务或 Serverless 容器服务时，可能会有解析自定义内部域名的需求，例如：

- 在集群外自建了集中存储服务，需要将集群中的监控或日志数据采集通过固定内部域名发送到存储服务。
- 传统业务在进行容器化改造过程中，部分服务的代码配置了用固定域名调用内部其他服务，且无法修改配置，即无法使用 Kubernetes 的 Service 名称进行调用。

方案选择

本文将介绍以下3种在集群中使用自定义域名解析的方案示例：

方案	优势
方案1：使用 CoreDNS Hosts 插件配置任意域名解析	简单直观，可以添加任意解析记录。
方案2：使用 CoreDNS Rewrite 插件指向域名到集群内服务	无需提前知道解析记录的 IP 地址，但要求解析记录指向的地址必须部署在集群中。
方案3：使用 CoreDNS Forward 插件将自建 DNS 设为上游 DNS	可以管理大量的解析记录，记录的管理都在自建 DNS 中，增删记录无需修改 CoreDNS 配置。

> **说明**
> 方案1和方案2，每次添加解析记录都需要修改 CoreDNS 配置文件（无需重启）。请根据自身需求评估并选择具体方案。

方案示例

方案1：使用 CoreDNS Hosts 插件配置任意域名解析

1. 执行以下命令，修改 CoreDNS 的 configmap。示例如下：

```
kubectl edit configmap coredns -n kube-system
```

2. 修改 hosts 配置，将域名加入 hosts，示例如下：

```
hosts {
      192.168.1.6      harbor.example.com
      192.168.1.8      es.example.com
      fallthrough
}
```

> **说明**
> 将 `harbor.example.com` 指向192.168.1.6；`es.example.com` 指向192.168.1.8。

完整配置示例如下：

```
apiVersion: v1
data:
      Corefile: |2-
        .:53 {
            errors
            health
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
```

```
                fallthrough in-addr.arpa ip6.arpa
            }
            hosts {
                192.168.1.6     harbor.example.com
                192.168.1.8     es.example.com
                fallthrough
            }
            prometheus :9153
            forward . /etc/resolv.conf
            cache 30
            reload
            loadbalance
        }
kind: ConfigMap
metadata:
      labels:
        addonmanager.kubernetes.io/mode: EnsureExists
      name: coredns
      namespace: kube-system
```

方案2：使用 CoreDNS Rewrite 插件指向域名到集群内服务

如果需要使用自定义域名的服务部署在集群中，可以使用 CoreDNS 的 Rewrite 插件，将指定域名解析到某个 Service 的 ClusterIP。

1. 执行以下命令，修改 CoreDNS 的 configmap。示例如下：

```
kubectl edit configmap coredns -n kube-system
```

2. 执行以下命令，加入 Rewrite 配置。示例如下：

```
rewrite name es.example.com es.logging.svc.cluster.local
```

> **说明**
> 将 `es.example.com` 指向部署在 `logging` 命名空间下的 `es` 服务，如有多个域名可添加多行。

完整配置示例如下：

```
apiVersion: v1
data:
      Corefile: |2-
        .:53 {
            errors
            health
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
            rewrite name es.example.com es.logging.svc.cluster.local
            prometheus :9153
            forward . /etc/resolv.conf
            cache 30
            reload
            loadbalance
        }
kind: ConfigMap
metadata:
      labels:
```

```
        addonmanager.kubernetes.io/mode: EnsureExists
    name: coredns
    namespace: kube-system
```

方案3：使用 CoreDNS Forward 插件将自建 DNS 设为上游 DNS

1. 查看 forward 配置。forward 默认配置如下所示，指非集群内域名通过 CoreDNS 所在节点 `/etc/resolv.conf` 文件中配置的 nameserver 解析。

```
forward . /etc/resolv.conf
```

2. 配置 forward，将 `/etc/resolv.conf` 显式替换为自建的 DNS 服务器地址。示例如下：

```
forward . 10.10.10.10
```

完整配置示例如下：

```
apiVersion: v1
data:
   Corefile: |2-
     .:53 {
         errors
         health
         kubernetes cluster.local. in-addr.arpa ip6.arpa {
             pods insecure
             upstream
             fallthrough in-addr.arpa ip6.arpa
         }
         prometheus :9153
         forward . 10.10.10.10
         cache 30
         reload
         loadbalance
     }
kind: ConfigMap
metadata:
   labels:
     addonmanager.kubernetes.io/mode: EnsureExists
   name: coredns
   namespace: kube-system
```

3. 将自定义域名的解析记录配置到自建 DNS。建议将节点上 `/etc/resolv.conf` 中的 nameserver 添加到自建 DNS 的上游，因为部分服务依赖腾讯云内部 DNS 解析，如果未将其设为自建 DNS 的上游，可能导致部分服务无法正常工作。本文以 BIND 9 为例修改配置文件，将上游 DNS 地址写入 forwarders 中。示例如下：

> ⚠ **注意**
> 自建 DNS Server 和请求源不在同个 Region，可能会导致部分不支持跨域访问的腾讯域名失效。

```
options {
     forwarders {
             183.60.83.19;
             183.60.82.98;
     };
     ...
```

参考文档

- CoreDNS Hosts 插件文档
- CoreDNS Rewrite 插件文档
- CoreDNS Forward 插件文档

在 TKE 中配置 ExternalDNS

最近更新时间：2024-10-31 09:49:01

本文介绍如何在腾讯云容器服务集群中配置 ExternalDNS。

什么是 External DNS

ExternalDNS 将公开的 Kubernetes Service 和 Ingress 与 DNS 提供商同步。
受 Kubernetes 集群内部 DNS 服务器 Kubernetes DNS 的启发，ExternalDNS 使 Kubernetes 资源可通过公共 DNS 服务器发现。与 KubeDNS 一样，它从 Kubernetes API 中检索资源列表（Service、Ingress 等），以确定所需的 DNS 记录列表。然而，与 KubeDNS 不同的是，它本身并不是一个 DNS 服务器，而只是用于对接其他 DNS 提供商。更多请查看 ExternalDNS Readme 。

操作步骤

配置 API 密钥 的 CAM 权限

在腾讯云 访问管理控制台，获取 API 密钥的 SecretId 和 SecretKey 信息，确保当前的用户的 CAM 权限拥有以下策略：

```
{
    "version": "2.0",
    "statement": [
        {
            "effect": "allow",
            "action": [
                "dnspod:ModifyRecord",
                "dnspod:DeleteRecord",
                "dnspod:CreateRecord",
                "dnspod:DescribeRecordList",
                "dnspod:DescribeDomainList"
            ],
            "resource": [
                "*"
            ]
        },
        {
            "effect": "allow",
            "action": [
                "privatedns:DescribePrivateZoneList",
                "privatedns:DescribePrivateZoneRecordList",
                "privatedns:CreatePrivateZoneRecord",
                "privatedns:DeletePrivateZoneRecord",
                "privatedns:ModifyPrivateZoneRecord"
            ],
            "resource": [
                "*"
            ]
        }
    ]
}
```

部署 ExternalDNS 服务

配置 PrivateDNS 或 DNSPod

腾讯 DNS 解析 DNSPod 向全网域名提供免费的智能解析服务，拥有海量处理能力、灵活扩展性和安全能力。为您的站点提供稳定、安全、快速的解析体验。
Private DNS 是基于腾讯云私有网络 VPC 的私有域名解析及管理服务，为您提供安全、稳定、高效的内网智能解析服务。支持在私有网络中快速构建 DNS 系统，满足定制化解析需求。

- 如果您想在腾讯云的环境中使用内网的 DNS 服务：
  - 配置下列 YAML 文件中参数：`--tencent-cloud-zone-type=private`
  - 在 PrivateDNS 控制台创建 DNS 域名。DNS 域名记录中将会包含 DNS 记录。
- 如果您想在腾讯云的环境中使用公网的 DNS 服务：

- 配置下列 YAML 文件中参数：`--tencent-cloud-zone-type=public`
- 在 DNSPod 控制台 创建 DNS 域名。DNS 域名记录中将会包含 DNS 记录。

在 Kubernetes 集群中部署相关资源对象

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: external-dns
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: external-dns
rules:
- apiGroups: [""]
  resources: ["services","endpoints","pods"]
  verbs: ["get","watch","list"]
- apiGroups: ["extensions","networking.k8s.io"]
  resources: ["ingresses"]
  verbs: ["get","watch","list"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: external-dns-viewer
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: external-dns
subjects:
- kind: ServiceAccount
  name: external-dns
  namespace: default
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: external-dns
data:
  tencent-cloud.json: |
    {
      "regionId": "ap-shanghai", # 必填项，集群所在地域的 ID
      "secretId": "******",
      "secretKey": "******",
      "vpcId": "vpc-******",     # 必填项，集群所在 VPC 的 ID
      "internetEndpoint": false # 腾讯云API入口。如果需要在非腾讯云的环境部署，改为true，走公网访问。
    }
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: external-dns
spec:
  strategy:
    type: Recreate
  selector:
```

```
    matchLabels:
      app: external-dns
  template:
    metadata:
      labels:
        app: external-dns
    spec:
      containers:
      - args:
        - --source=service
        - --source=ingress
        - --domain-filter=external-dns-test.com # 将使 ExternalDNS 仅看到与提供的域匹配的托管区域，省略以处理所有可用的托管区域
        - --provider=tencentcloud
        - --policy=sync # 设置“upsert-only”将阻止 ExternalDNS 删除任何记录
        - --tencent-cloud-zone-type=private # 仅管理私有托管区域。设置“public”以使用公网 DNS 服务
        - --tencent-cloud-config-file=/etc/kubernetes/tencent-cloud.json
        image: ccr.ccs.tencentyun.com/tke-market/external-dns:v1.1.0
        imagePullPolicy: Always
        name: external-dns
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        volumeMounts:
        - mountPath: /etc/kubernetes
          name: config-volume
          readOnly: true
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      serviceAccount: external-dns
      serviceAccountName: external-dns
      terminationGracePeriodSeconds: 30
      volumes:
      - configMap:
          defaultMode: 420
          items:
          - key: tencent-cloud.json
            path: tencent-cloud.json
          name: external-dns
        name: config-volume
```

使用示例

创建名为 nginx 的 Service，示例如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  annotations:
    external-dns.alpha.kubernetes.io/hostname: nginx.external-dns-test.com # 公网域名地址
    external-dns.alpha.kubernetes.io/internal-hostname: nginx-internal.external-dns-test.com # 内网域名地址
    external-dns.alpha.kubernetes.io/ttl: "600"
spec:
  type: LoadBalancer
  ports:
  - port: 80
    name: http
```

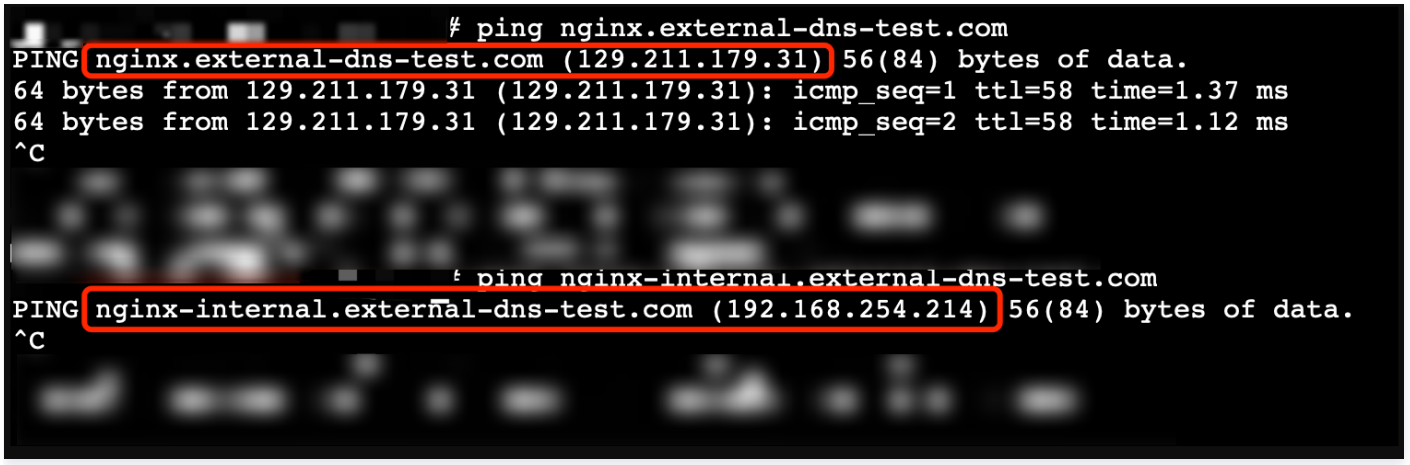
```
    targetPort: 80
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        ports:
        - containerPort: 80
          name: http
```

- `nginx.external-dns-test.com` 将记录服务的 LoadBalancer VIP。
- `nginx-internal.external-dns-test.com` 将记录服务的 ClusterIP。所有的 DNS 记录的 TTL 都是 600。

执行验证

名为 "nginx" 的 Service 的 ClusterIP 为 `192.168.254.214`，LoadBalancer VIP 为 `129.211.179.31`，如下图所示：

当您在与集群位于同一个 VPC 内的节点上，ping 名为 "nginx" 的 Service 的 annotation 域名声明时，会自动解析成 ClusterIP 和 LoadBalancer VIP。



从 kube-dns 切换到 CoreDNS

最近更新时间：2023-12-01 09:59:22

低版本的 kube-dns 存在一些潜在问题，例如：

1. 依赖库 miekg/dns 存在 bug，导致 kube-dns 在处理 /etc/resolv.conf 中的某些特定 option 时会发生 panic。
2. 依赖库 client-go 的版本较低，不支持 周期性刷新 token 的功能，导致 kube-dns 在访问 kube-apiserver 时无法进行鉴权。因此，我们建议您将集群中的 kube-dns 切换到 CoreDNS。

本文档提供了一种尽可能平滑、对业务无感知的方式，来完成将集群中的 kube-dns 切换到 CoreDNS 的过程。

前置说明

- 集群 Kubernetes 版本不低于1.12。
- 在切换到 CoreDNS 之前，请根据集群的 Kubernetes 版本选择最适合的 CoreDNS 版本。详情请参见 选择最佳 CoreDNS 版本 。
- 如果集群的 kube-proxy 正在使用 IPVS 模式，在 kube-dns 缩容阶段，由于 IPVS UDP 会话超时，可能会导致 DNS 解析失败的概率性问题。为了缩短解析失败的持续时间，使切换过程尽可能平滑，请配置 IPVS UDP 会话保持的超时时间。由于 kube-dns 不具备像 CoreDNS 一样的优雅退出的能力，针对 kube-dns 切换到 CoreDNS 的场景，建议将超时时间配置为5秒，配置方法详情请参见 配置会话保持 。完成配置后，请等待5分钟后再继续进行后续步骤。

> **说明：**
> 关于 ipvs udp 会话超时问题，TencentOS 3.1 中0009.23及以上的内核合并了社区 expire_nodest_conn 特性，能快速删除已有连接，减少解析超时的持续时间，用户无需再配置 ipvs UDP 会话保持超时时间。具体特性请参见 ipvs: queue delayed work to expire no destination connections if expire_nodest_conn=1 。

操作步骤

准备 CoreDNS 资源文件

基于以下 CoreDNS 资源模板，根据集群 Kubernetes 版本以及 CoreDNS 版本处理 # 标注的内容后，将其保存到 switch2coredns.yaml 文件：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:coredns
rules:
  - apiGroups:
      - '*'
    resources:
      - endpoints
      - services
      - pods
      - namespaces
    verbs:
      - list
      - watch
  # 如果集群 k8s 版本大于等于 1.20，则需要增加对 endpointslice 的权限，反之则不需要，从此开始
  - apiGroups:
      - discovery.k8s.io
    resources:
      - endpointslices
    verbs:
      - list
      - watch
  # 到此结束
```

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
    kubernetes.io/cluster-service: "true"
  name: coredns
  namespace: kube-system

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    addonmanager.kubernetes.io/mode: EnsureExists
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:coredns
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:coredns
subjects:
  - kind: ServiceAccount
    name: coredns
    namespace: kube-system

---
apiVersion: v1
data:
  Corefile: |2-
        .:53 {
            template ANY HINFO . {
                rcode NXDOMAIN
            }
            errors
            health {
                lameduck 30s
            }
            ready
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                # 如果 CoreDNS 版本小于 v1.7.0，则需要 upstream 选项，反之则不需要，CoreDNS 版本确定参见本文档前置
说明
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
            prometheus :9153
            forward . /etc/resolv.conf {
                prefer_udp
            }
            cache 30
            reload
            loadbalance
        }
kind: ConfigMap
metadata:
  labels:
```

```
    addonmanager.kubernetes.io/mode: EnsureExists
  name: coredns
  namespace: kube-system

---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
  name: coredns
  namespace: kube-system
spec:
  # 副本数先设置成0，后面的步骤会调整
  replicas: 0
  selector:
    matchLabels:
      k8s-app: kube-dns
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      annotations:
        seccomp.security.alpha.kubernetes.io/pod: docker/default
      labels:
        k8s-app: kube-dns
      name: coredns
      namespace: kube-system
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: k8s-app
                    operator: In
                    values:
                      - kube-dns
              topologyKey: kubernetes.io/hostname
      containers:
        - args:
            - -conf
            - /etc/coredns/Corefile
          # 集群所在地域不同，{HOST} 也不同，可直接采用当前 kube-dns 镜像所使用的域名替换这里的 {HOST}，比如香港地域域
名为：hkccr.ccs.tencentyun.com
          # 使用要切换到的 CoreDNS 版本来替换 {VERSION}，CoreDNS 版本确定参见本文档前置说明
          image: {HOST}/tkeimages/coredns:{VERSION}
          name: coredns
          livenessProbe:
            failureThreshold: 5
            httpGet:
              path: /health
              port: 8080
              scheme: HTTP
```

```
        initialDelaySeconds: 60
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 5
      ports:
        - containerPort: 53
          name: dns
          protocol: UDP
        - containerPort: 53
          name: dns-tcp
          protocol: TCP
        - containerPort: 9153
          name: metrics
          protocol: TCP
      readinessProbe:
        failureThreshold: 5
        httpGet:
          path: /ready
          port: 8181
          scheme: HTTP
        initialDelaySeconds: 30
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 5
      resources:
        limits:
          memory: 170M
        requests:
          cpu: 100m
          memory: 30M
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          add:
            - NET_BIND_SERVICE
          drop:
            - all
        readOnlyRootFilesystem: true
      volumeMounts:
        - mountPath: /etc/coredns
          name: config-volume
          readOnly: true
  dnsPolicy: Default
  serviceAccountName: coredns
  tolerations:
    - effect: NoSchedule
      key: node-role.kubernetes.io/master
    - key: CriticalAddonsOnly
      operator: Exists
  volumes:
    - configMap:
        items:
          - key: Corefile
            path: Corefile
        name: coredns
        optional: true
      name: config-volume
```

迁移配置

如果您在当前集群中对 kube-dns 进行了一些自定义配置，例如自定义上游等，那么您需要将这些配置等效地迁移到 CoreDNS。请参考以下示例完成迁移：

kube-dns 自定义配置

```
apiVersion: v1
data:
  federations: |
    {"foo" : "foo.feddomain.com"}
  stubDomains: |
    {"abc.com" : ["1.2.3.4"], "my.cluster.local" : ["2.3.4.5"]}
  upstreamNameservers: |
    ["8.8.8.8", "8.8.4.4"]
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
```

迁移到 CoreDNS 的相应配置

根据 CoreDNS 版本处理 # 标注的内容后，将您的自定义配置写入上一步准备好的 switch2coredns.yaml 文件：

```
apiVersion: v1
data:
  Corefile: |2-
        .:53 {
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                # 如果 CoreDNS 版本小于 v1.7.0，则需要 upstream 选项，反之则不需要，CoreDNS 版本确定参见本文档前置
说明
                upstream  8.8.8.8 8.8.4.4
                fallthrough in-addr.arpa ip6.arpa
            }
            federation cluster.local {
               foo foo.feddomain.com
            }
            forward . 8.8.8.8 8.8.4.4
        }
        abc.com:53 {
            forward . 1.2.3.4
        }
        my.cluster.local:53 {
            forward . 2.3.4.5
        }
kind: ConfigMap
metadata:
  labels:
    addonmanager.kubernetes.io/mode: EnsureExists
  name: coredns
  namespace: kube-system
```

部署 CoreDNS

将上述 switch2coredns.yaml 完善后，执行以下命令部署 CoreDNS：

> **注意：**
> 此时 CoreDNS Deployment 的副本数为0，不会实际部署 Pod。

```
kubectl apply -f switch2coredns.yaml
```

执行切换

切换的总体思路为：逐步扩容 CoreDNS，缩容 kube-dns，直到所有 kube-dns 副本都被 CoreDNS 副本替代。执行切换的步骤如下：

> **注意：**
> 由于 kube-dns 和 CoreDNS 副本之间配置了 Pod 反亲和，因此如果集群资源有限，例如节点数不大于2，可以考虑先缩容 kube-dns，再扩容 CoreDNS，类似于原地腾挪，但这种方案在缩容时会带来服务容量的降低。

1. 扩容一个 CoreDNS 副本：

```
kubectl scale deployment coredns -n kube-system --replicas=1
```

2. 等待该 CoreDNS Pod 正常运行并变为 Ready 状态。如果长时间无法 Ready，请执行以下命令查看 CoreDNS Pod 的日志以诊断问题。

```
kubectl logs $(COREDNS_POD_NAME) -n kube-system
```

3. 等待 CoreDNS Pod Ready 后，进入一个业务 Pod 或创建一个测试 Pod（包含 nslookup 工具），将 nameserver 指向该 CoreDNS Pod 的 IP 地址，并测试系统域名、业务域名（注意命名空间）、外部域名等的解析是否正常。

```
nslookup kubernetes.default $(COREDNS_POD_IP)
nslookup $(业务域名) $(COREDNS_POD_IP)
nslookup www.baidu.com $(COREDNS_POD_IP)
```

4. 检查该 CoreDNS Pod 是否已添加到 kube-dns Service 的后端列表。

```
kubectl get endpoints kube-dns -n kube-system -o jsonpath='{.subsets[*].addresses[*].ip}{"\n"}' | grep
$(COREDNS_POD_IP)
```

5. 缩容一个 kube-dns 副本，假设 kube-dns 原先副本数为 N：

```
kubectl scale deployment kube-dns -n kube-system --replicas=N-1
```

6. 确认业务侧没有频繁的 DNS 解析报错，通过监控系统（如果具备）确认 DNS 服务整体 QPS 水平稳定，持续观察5分钟。
7. 重复执行步骤1-6，每扩容一个 CoreDNS Pod 就缩容一个 kube-dns pod，直到 CoreDNS 副本数达到原先 kube-dns 副本数，而 kube-dns 副本数变为0，即完成整体切换。
8. 切换完成后，观察业务持续72小时无问题，再清理 kube-dns 相关资源：

```
kubectl delete deployment kube-dns -n kube-system
kubectl delete cm kube-dns -n kube-system
kubectl delete serviceaccount kube-dns -n kube-system
```

回滚

如在切换过程中遇到不符合预期的行为，您可以通过扩容 kube-dns Deployment 到原来的副本数，缩容 CoreDNS Deployment 到0副本的方式实现回滚。

CoreDNS ServiceAccount Token 过期问题解决方案

最近更新时间：2024-04-12 14:58:21

问题影响

在 Kubernetes 1.22 及以上版本中，Pod 不再使用 ServiceAccount 绑定的 Secret Token，而是使用 kubelet 周期性申请的 Token。该 Token 默认有效期为一年，Token 过期后不会自动更新，详情请参见 1.21 CHANGELOG 。
TKE 服务中 CoreDNS 1.6.2 及以下版本不会自动加载 Token，导致运行时间超过1年的 CoreDNS 访问 apiserver 鉴权失败，无法获取到 service/endpoint 最新的变更事件，导致用户可能受到的影响如下：
1. 业务使用 service 做服务发现，新建的 service 域名无法正常解析。
2. 业务使用 headless service 做服务发现，workload 发生变更后，headless service 域名不会解析到最新 pod ip 地址，会影响到存量业务访问。

问题原因

CoreDNS 1.6.2 版本使用了 v11.0.0 版本的 client-go，不会重新加载 Token，导致运行时间超过1年的 coredns，访问 apiserver 鉴权失败，无法获取到 service/endpoint 最新的变更事件。

修复方案

针对这些集群，您需要将 CoreDNS 升级到1.8.4版本，可彻底解决该问题，具体操作方式请参见 CoreDNS 升级到 1.8.4 。

> **注意：**
> **CoreDNS 升级影响如下：**
> 1. iptables 模式下，kube-proxy 会在同步 iptables 规则后及时清理遗留的 conntrack 表项，CoreDNS 升级不会影响用户集群。
> 2. ipvs 模式下，内核默认打开会话保持特性，这会导致在 CoreDNS 升级或重启期间，业务侧的 DNS 解析请求，在5分钟（300s）内概率性超时。如果业务自身没有 UDP 服务，可以降低 IPVS UDP 协议的会话保持超时时间，以此来减少解析超时的持续时间。
>
> 具体详情请参见 配置 CoreDNS 平滑升级 。

临时修复

如果您短期内无法升级 CoreDNS 版本，可以通过重建 CoreDNS 的 pod 临时修复，CoreDNS 会默认加载 1 年的有效期，具体操作方式如下（**此方案仅为临时修复方案，请尽快通过升级方式彻底解决**）。

TKE 集群

- 针对 TKE 集群，当 kube-proxy 是 iptables 模式时，可以通过重新部署 CoreDNS，触发 CoreDNS 滚动更新修复问题。操作步骤如下：
  1.1 选择**集群** > **工作负载** > **Deployment**，在 kube-system ns 下选择 coredns，单击**设置更新策略**。如下图所示：

配置更新策略如下：

1.2 单击**重新部署**。如下图所示：

- 当 kube-proxy 是 ipvs 模式时，如果希望 CoreDNS 实现平滑升级，不出现域名解析超时或者尽量减少服务不可用时间，可以通过配置 CoreDNS 平滑升级，具体指引请参见 配置 CoreDNS 平滑升级 。

TKE Serverless 集群

针对 TKE Serverless 集群，操作步骤如下：

1. 选择**集群 > 工作负载 > Deployment**，在 kube-system ns 下选择 coredns，单击**设置更新策略**。如下图所示：

配置更新策略如下：

2. 单击**重新部署**。如下图所示：

常见问题

1. 为什么做这次修复变更?

低版本 CoreDNS 存在 Token 过期隐患，会导致用户 service 访问异常。

2. 这个隐患会有什么影响?

当证书到期后，CoreDNS 将无法连接到 apiserver，这会导致集群内的 pod 的 service 服务发现出现异常。具体表现为：

1. 业务使用 service 做服务发现，新建的 service 域名无法正常解析。
2. 业务使用 headless service 做服务发现，workload 发生变更后，headless service 域名不会解析到最新 pod ip 地址，会影响到存量业务访问。

3. 如何进行平滑升级?

具体指引请参见 配置 CoreDNS 平滑升级 。

TKE 集群 CoreDNS 使用废弃 API 问题的解决方案

最近更新时间：2024-11-14 17:25:32

问题影响

在 TKE 集群中，如果 CoreDNS 启动时间较早，集群升级到 1.26 后，CoreDNS 可能仍在监听已废弃的 v1beta1 版 EndpointSlice 资源。由于该资源在 Kubernetes 1.26 版本中被移除，导致无法获取到最新的 Service/Endpoint 变更事件，对业务造成以下影响：

- **Service 域名解析失败：**使用 Service 进行服务发现的业务，新建的 Service 域名无法正常解析。
- **Headless Service 解析异常：**使用 Headless Service 进行服务发现的业务，当工作负载发生变更后，Headless Service 域名不会解析到最新的 Pod IP 地址，影响现有业务的正常访问。

潜在问题场景

1. **从 1.20 版本集群升级到 1.26 版本：**创建 1.20 版本的集群后，将集群控制面逐步升级到 1.26。
2. **CoreDNS 在 1.20 版本集群中重启或扩容后升级：**在集群处于 1.20 版本时，CoreDNS 发生重启或扩容，然后将集群控制面升级到 1.26。

问题原因

在 TKE 1.20 版本的集群中，CoreDNS 使用的是 1.8.4 版本，该版本监听的是 EndpointSlice (v1beta1) 资源。由于 Kubernetes 在 1.25 版本中移除了 EndpointSlice(v1beta1) API，当集群升级到 1.26 后，CoreDNS 仍继续监听已被移除的资源，导致无法获取 Service/Endpoint 的最新事件。
当集群运行在 1.22、1.24 版本时，CoreDNS 的日志中会出现以下警告，表明 CoreDNS 正在监听已废弃的 API：

```
W0703 06:28:15.718073       1 warnings.go:70] discovery.k8s.io/v1beta1 EndpointSlice is deprecated in v1.21+, unavailable in v1.25+; use discovery.k8s.io/v1 EndpointSlice
```

CoreDNS 滚动更新的注意事项

- 发布前，注意检查集群相关配置。
  - 确保集群中可调度节点数量大于 CoreDNS 的实例数。默认情况下，CoreDNS 有两个实例，请确保在重启 CoreDNS 前，保证集群至少有三个调用资源充足的节点。
  - 当集群 kubeproxy 处于 ipvs 模式下时，内核默认打开会话保持特性，这会导致在 CoreDNS 升级或重启期间，业务侧的 DNS 解析请求，在5分钟（300s）内概率性超时。如果业务自身没有 UDP 服务，可以降低 IPVS UDP 协议的会话保持超时时间，以此来减少解析超时的持续时间。详情请参见 配置 CoreDNS 平滑升级 。
- 发布前，注意 corefile 配置。
  - health 插件配置。lameduck 时间配置不要大于30秒。
  - forward 插件配置。如果 forward 到 /etc/resolv.conf，要注意是否修改过集群节点的 /etc/resolv.conf 内容。如果集群内节点的 /etc/resolv.conf 不一致，在 CoreDNS 重启漂移到其他节点后，导致 CoreDNS 的 DNS 上游地址出现变化。
- 发布后，注意 CoreDNS 的 Pod 状态。
  - 确认所有 CoreDNS 的 Pod 都滚动更新完成，状态都是 Ready 的，没有 PENDING 状态的 Pod。
  - 如果 Pod 处于 PENDING 状态，检查 Event 信息，解决因资源不足或节点污点等各种原因，导致新的 CoreDNS 无法启动的问题。

修复方案

重建 CoreDNS，让 CoreDNS 监听 `EndpointSlices (v1)` 资源。

TKE 集群

- 针对 TKE 集群，当 kube-proxy 是 iptables 模式时，可以通过重新部署 CoreDNS，触发 CoreDNS 滚动更新修复问题。操作步骤如下：
  1.1 选择**集群** > **工作负载** > **Deployment**，在 kube-system ns 下选择 coredns，单击**设置更新策略**。如下图所示：

配置更新策略如下：

1.2 单击**重新部署**。如下图所示：

- 若希望 CoreDNS 实现平滑升级，避免域名解析超时或尽量减少服务不可用时间，可以参考以下步骤：
  - 配置 CoreDNS 平滑升级：请参见 配置 CoreDNS 平滑升级 进行详细配置。
  - 执行滚动更新：按照文档指引，完成配置后，对 CoreDNS 进行滚动更新。
- **CoreDNS 重启之后，可以观察 Pod 内的日志信息，日志中不会再出现使用废弃 API 的警告日志。**

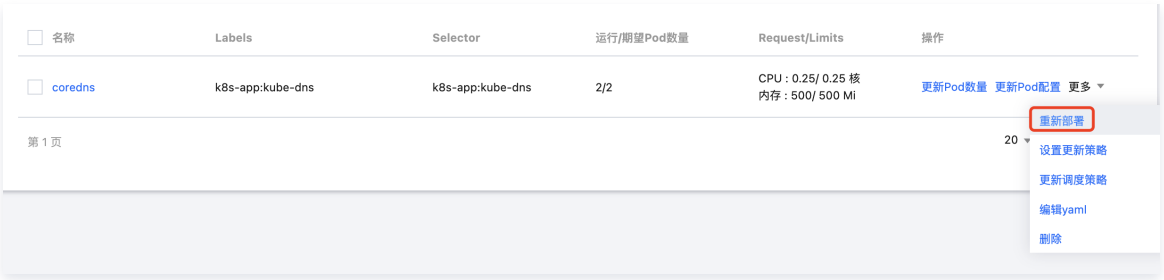
TKE Serverless 集群

针对 TKE Serverless 集群，操作步骤如下：

1. 选择**集群 > 工作负载 > Deployment**，在 kube-system ns 下选择 coredns，单击**设置更新策略**。如下图所示：

配置更新策略如下：

2. 单击**重新部署**。如下图所示：



结语

通过上述步骤，可以有效解决 CoreDNS 使用废弃 API 导致的服务发现问题，确保业务的正常运行。如有任何疑问或需要协助，请及时联系腾讯云客服或技术支持。

自建 Nginx Ingress 实践教程
快速开始

最近更新时间：2024-09-14 16:10:21

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

概述

Nginx Ingress Controller 是基于高性能 NGINX 反向代理实现的 Kubernetes Ingress 控制器，也是最常用的开源 Ingress 实现。本文介绍如何在 TKE 环境中自建 Nginx Ingress Controller，主要使用 helm 进行安装，提供一些 `values.yaml` 配置指引。

前提条件

- 创建了 TKE 集群。
- 安装了 helm。
- 配置了 TKE 集群的 kubeconfig，且有权限操作 TKE 集群。详情请参见 连接集群。

使用 helm 安装

添加 helm repo：

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

> **说明：**
> 如果 Helm 命令所在的机器无法连接到 GitHub，将添加失败，您可以参考 常见问题：连不上 GitHub 导致安装失败 来解决此问题。

查看默认配置：

```
helm show values ingress-nginx/ingress-nginx
```

Nginx Ingress 依赖的镜像在 `registry.k8s.io` 这个 registry 下，国内网络环境无法拉取，可替换为 docker hub 中的 mirror 镜像。
准备 `values.yaml`：

```
controller: # 以下配置将依赖镜像替换为了 docker hub 上的 mirror 镜像以保证在国内环境能正常拉取
  image:
    registry: docker.io
    image: k8smirror/ingress-nginx-controller
  admissionWebhooks:
    patch:
      image:
        registry: docker.io
        image: k8smirror/ingress-nginx-kube-webhook-certgen
  defaultBackend:
    image:
      registry: docker.io
      image: k8smirror/defaultbackend-amd64
  opentelemetry:
    image:
      registry: docker.io
      image: k8smirror/ingress-nginx-opentelemetry
```

> **说明：**
> 配置中的 mirror 镜像均使用 image-porter 长期自动同步，可放心安装和升级。

安装：

```
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  -f values.yaml
```

> **说明：**
> 后续如果需要修改 values 配置，或者升级版本，都可以通过执行这个命令来更新 Nginx Ingress Controller。

查看流量入口（CLB VIP 或域名）：

```
$ kubectl get services -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)
AGE
ingress-nginx-controller             LoadBalancer   xxx.xx.xxx.xxx   xxx.xx.xx.xxx
80:30683/TCP,443:32111/TCP   53s
ingress-nginx-controller-admission   ClusterIP      xxx.xx.xxx.xxx   <none>          443/TCP
53s
```

> **说明：**
> `LoadBalancer` 类型 Service 的 `EXTERNAL-IP` 就是 CLB 的 VIP 或域名，可以配置 DNS 解析。如果是 VIP，则配 A 记录；如果是 CLB 域名，则配置 CNAME 记录。

常见问题

连不上 GitHub 导致安装失败如何处理?

问题描述

`ingress-nginx` 的 Helm chart 仓库地址在 GitHub。如果 Helm 命令所在的环境无法连接到 GitHub，就无法下载 chart 包，`helm repo add` 操作也会失败。

解决方案

如果遇到这个问题，您可以在能连上 GitHub 的机器上下载 chart 包，然后拷贝到 Helm 命令所在机器上。详细步骤如下：

1. 下载 chart 包，代码示例如下：

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm fetch ingress-nginx/ingress-nginx
```

> **说明：**
> 如果需要下载指定版本的 chart，可以给 fetch 子命令添加 `--version` 参数，例如：
>
> ```
> helm fetch ingress-nginx/ingress-nginx --version 4.7.5
> ```

2. 查看下载的 chart 包：

```
$ ls
ingress-nginx-4.11.2.tgz
```

3. 将下载的 chart 包拷贝到 Helm 命令所在的机器上，安装命令将 chart 名称替换成压缩包文件路径即可：

```
helm upgrade --install ingress-nginx ingress-nginx-4.11.2.tgz \
  --namespace ingress-nginx --create-namespace \
  -f values.yaml
```

版本与升级

Nginx Ingress 的版本需要与 Kubernetes 集群版本能够兼容，可参考官方 Supported Versions table 确认当前集群版本能否支持最新的 nginx ingress，如果不支持，安装的时候需指定 chart 版本。
例如当前的 TKE 集群版本是 1.24，chart 版本最高只能到 `4.7.*`，通过以下命令检查有哪些可用版本：

```
$ helm search repo ingress-nginx/ingress-nginx --versions | grep 4.7.
ingress-nginx/ingress-nginx     4.7.5           1.8.5           Ingress controller for Kubernetes using
NGINX a...
ingress-nginx/ingress-nginx     4.7.3           1.8.4           Ingress controller for Kubernetes using
NGINX a...
ingress-nginx/ingress-nginx     4.7.2           1.8.2           Ingress controller for Kubernetes using
NGINX a...
ingress-nginx/ingress-nginx     4.7.1           1.8.1           Ingress controller for Kubernetes using
NGINX a...
ingress-nginx/ingress-nginx     4.7.0           1.8.0           Ingress controller for Kubernetes using
NGINX a...
```

可以看到 `4.7.*` 版本最高是 `4.7.5`，安装的时候需加上版本号：

```
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
  --version 4.7.5 \
  --namespace ingress-nginx --create-namespace \
  -f values.yaml
```

> ⚠ **注意：**
> TKE 集群升级前，先检查当前 Nginx Ingress 版本能否兼容升级后的集群版本，如果不能兼容，先升级 Nginx Ingress（用上面的命令指定 chart 版本号）。

使用 Ingress

Nginx Ingress 实现了 Kubernetes 的 Ingress API 定义的标准能力，Ingress 的基础用法请参见 Kubernetes 官方文档。
必须指定 `ingressClassName` 为 Nginx Ingress 实例所使用的 IngressClass（默认为 `nginx`）：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx
                port:
                  number: 80
```

除此之外，Nginx Ingress 还有很多其它特有的功能，通过 Ingress 注解来扩展 Ingress 的功能，请参见 Nginx Ingress Annotations。

更多自定义

如果需要对 Nginx Ingress 进行更多的自定义，可参考以下文档，根据自己需求合并 `values.yaml` 配置，values.yaml 完整配置示例 提供了合并后的 `values.yaml` 完整配置示例。
另外您也可以将 `values.yaml` 拆成多个文件维护，执行安装或更新命令时，用多个 `-f` 参数指定多个配置文件即可：

```
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  -f image-values.yaml \
  -f prom-values.yaml \
  -f logrotate-values.yaml \
  -f autoscaling-values.yaml
```

- 自定义负载均衡器
- 启用 CLB 直连
- 高并发场景优化
- 高可用配置优化
- 可观测性集成
- 接入腾讯云 WAF
- 安装多个 Nginx Ingress Controller
- 从 TKE Nginx Ingress 插件迁移到自建 Nginx Ingress
- values.yaml 完整配置示例

自定义负载均衡器

最近更新时间：2024-09-14 16:10:21

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

概述

默认安装会自动创建出一个公网 CLB 来接入流量，但您也可以利用 TKE 的 Service 注解对 Nginx Ingress Controller 的 CLB 进行自定义，本文为您介绍自定义的方法。

使用内网 CLB

例如改成内网 CLB，在 `values.yaml` 中的示例代码如下：

```
controller:
  service:
    annotations:
      service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: 'subnet-xxxxxx' # 内网 CLB 需指定 CLB 实例所在的子网 ID
```

使用已有 CLB

您也可以直接在 CLB 控制台 根据自身需求创建一个 CLB（例如自定义实例规格、运营商类型、计费模式、带宽上限等），然后在 `values.yaml` 中用注解复用这个 CLB，详情请参见 Service 使用已有 CLB。

```
controller:
  service:
    annotations:
      service.kubernetes.io/tke-existed-lbid: 'lb-xxxxxxxx' # 指定已有 CLB 的实例 ID
```

> **注意：**
> 在 CLB 控制台创建 CLB 实例时，选择的 VPC 需与集群一致。

公网和内网 IP 同时接入

有时需要让 Nginx Ingress 同时使用公网和内网的 IP 来接入流量，可以通过以下两种方案实现：

方案一：双 Service

配置 Nginx Ingress 使用两个 Service，默认创建一个公网 CLB Service，如果还需要一个内网 CLB 的 Service，可以配置 internal service：

```
controller:
  service:
    internal:
      enabled: true # 创建内网 CLB Service
      annotations:
        service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: "subnet-xxxxxxxx" # 配置内网 CLB 的子网
```

方案二：内网 CLB 绑 EIP

使用内网 CLB，然后在 CLB 控制台给 CLB 绑定一个 EIP。详情请参见 CLB 官方文档 内网负载均衡实例绑定 EIP。

> **说明：**

> 该功能是 CLB 的内测功能，需要 提交工单 申请开通。

CLB 跨域绑定

如果需要使用其他地域或 VPC 的 CLB 来接入流量，可以利用 CLB 的 跨地域绑定2.0 和 TKE 的 Service 跨域绑定 能力来实现，需要满足以下前提条件：

1. 账号是带宽上移类型。
2. 两个 VPC 通过云联网打通。
3. 开通了 CLB 的跨地域绑定2.0 功能（ 提交工单 申请开通）。

然后将 CLB 的 ID、所在地域和 VPC 信息配置在注解里：

```
controller:
  service:
    annotations:
      service.cloud.tencent.com/cross-region-id: "ap-guangzhou"   # 如果CLB在其它地域，指定下CLB所在地域
      service.cloud.tencent.com/cross-vpc-id: "vpc-xxx" # 指定CLB所在VPC
      service.kubernetes.io/tke-existed-lbid: "lb-xxx" # 如果使用已有CLB，指定下CLB ID
```

启用 CLB 直连

最近更新时间：2024-09-14 16:10:21

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

概述

流量从 CLB 转发到 Nginx Ingress 的链路可以直连，即不通过 NodePort 通信。这种方式可以带来更好的性能，并且可以实现获取真实源 IP 的需求。
如果您使用的是 TKE Serverless 集群，或者您能确保所有 Nginx Ingress Pod 都调度到超级节点上，那么这段链路本身就是直连的，无需进行任何额外操作。
在其他情况下，这段链路中间默认会通过 NodePort 通信。如果您希望启用直连，可以参考以下步骤（根据您的集群环境选择适用的步骤）。

> **说明：**
> 请参见 使用 LoadBalancer 直连 Pod 模式 Service。

GlobalRouter + VPC-CNI 网络模式启用直连

如果集群网络模式是 GlobalRouter，且启用了 VPC-CNI：

建议为 Nginx Ingress 声明用 VPC-CNI 网络，同时启用 CLB 直连，`values.yaml` 配置方法：

```
controller:
  podAnnotations:
    tke.cloud.tencent.com/networks: tke-route-eni # 声明使用 VPC-CNI 网络
  resources: #  resources 里声明使用弹性网卡
    requests:
      tke.cloud.tencent.com/eni-ip: "1"
    limits:
      tke.cloud.tencent.com/eni-ip: "1"
  service:
    annotations:
      service.cloud.tencent.com/direct-access: "true" # 启用 CLB 直通
```

GlobalRouter 网络模式启用直连

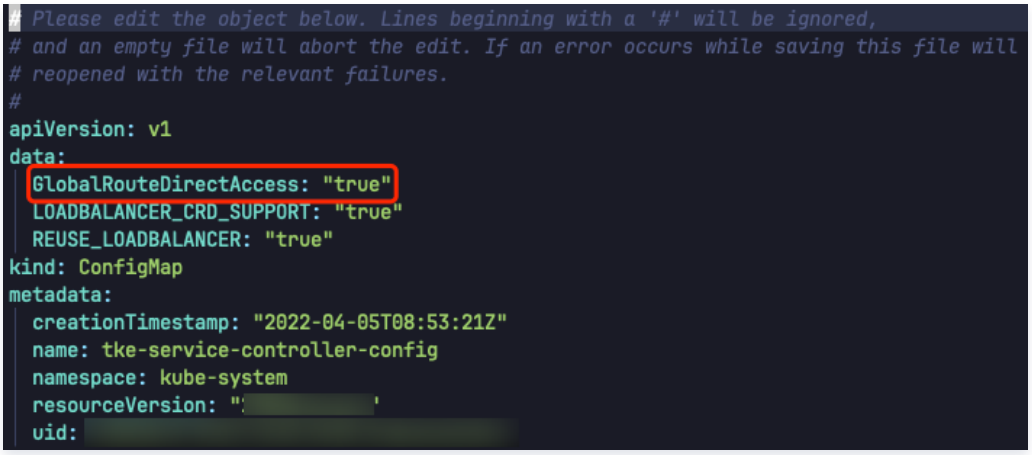
如果集群网络是 GlobalRouter，但没有启用 VPC-CNI，建议为集群开启 VPC-CNI，详情见 GlobalRouter + VPC-CNI 网络模式启用直连 启用 CLB 直连。
如果不希望开启 VPC-CNI，且腾讯云账号是带宽上移类型（请参见 账号类型说明），可以根据以下步骤启用直连，但是需接受 使用限制 。

> **注意：**
> 请确认您的账号满足上述条件，并接受使用限制。

1. 修改 configmap 开启 GlobalRouter 集群维度的直连能力：

```
kubectl edit configmap tke-service-controller-config -n kube-system
```

将 `GlobalRouteDirectAccess` 置为 true：



2. 配置 `values.yaml` 启用 CLB 直连：

```
controller:
  service:
    annotations:
      service.cloud.tencent.com/direct-access: "true" # 启用 CLB 直通
```

VPC-CNI 网络模式启用直连

如果集群网络本身就是 VPC-CNI，直接配置 `values.yaml` 启用 CLB 直连即可：

```
controller:
  service:
    annotations:
      service.cloud.tencent.com/direct-access: "true" # 启用 CLB 直通
```

高并发场景优化

最近更新时间：2025-03-21 10:13:22

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

操作场景

本文介绍如何针对高并发场景对 Nginx Ingress 进行配置调优。

操作指南

调大 CLB 规格和带宽

高并发场景的流量吞吐需求较高，对 CLB 的转发性能要求也较高，可以在 CLB 控制台 手动创建一个 CLB，实例规格选择性能容量型，按需选择型号，并将带宽上限调高（注意 VPC 要与 TKE 集群一致）。
CLB 创建好后，配置 nginx ingress 以复用这个 CLB 作为流量入口，详情请参见 自定义负载均衡器 。

调优内核参数与 Nginx 配置

针对高并发场景调优内核参数和 nginx 自身的配置，`values.yaml` 配置方法：

```
controller:
  extraInitContainers:
    - name: sysctl
      image: busybox
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: true
      command:
        - sh
        - -c
        - |
          sysctl -w net.core.somaxconn=65535 # 调大链接队列，防止队列溢出
          sysctl -w net.ipv4.ip_local_port_range="1024 65535" # 扩大源端口范围，防止端口耗尽
          sysctl -w net.ipv4.tcp_tw_reuse=1 # TIME_WAIT 复用，避免端口耗尽后无法新建连接
          sysctl -w fs.file-max=1048576 # 调大文件句柄数，防止连接过多导致文件句柄耗尽
  config:
    # nginx 与 client 保持的一个长连接能处理的请求数量，默认100，高并发场景建议调高，但过高也可能导致 nginx ingress 扩容后负载不均。
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#keep-alive-requests
    keep-alive-requests: "1000"
    # nginx 与 upstream 保持长连接的最大空闲连接数 （不是最大连接数），默认 320，在高并发下场景下调大，避免频繁建联导致 TIME_WAIT 飙升。
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#upstream-keepalive-connections
    upstream-keepalive-connections: "2000"
    # 每个 worker 进程可以打开的最大连接数，默认 16384。
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#max-worker-connections
    max-worker-connections: "65536"
```

其中大部分 `net.*` 参数是 namespace 级别参数，仅影响当前容器命名空间，但也有部分参数是 `unnamespaced` 参数，会影响当前宿主机。在不同 Linux 版本间可能也有差异，具体配置前请查阅官方文档。

> **说明：**

> 请参见 Nginx Ingress 高并发实践。

日志轮转

Nginx Ingress 默认会将日志打印到容器标准输出，日志由容器运行时自动管理，在高并发场景可能会导致 CPU 占用较高。
解决方案是将 Nginx Ingress 的日志输出到日志文件中，并使用 sidecar 对日志文件做自动轮转处理，以避免磁盘空间被日志填满。
`values.yaml` 配置方法：

```
controller:
  config:
    # nginx 日志落盘到日志文件，避免高并发下占用过多 CPU
    access-log-path: /var/log/nginx/nginx_access.log
    error-log-path: /var/log/nginx/nginx_error.log
  extraVolumes:
    - name: log # controller 挂载日志目录
      emptyDir: {}
  extraVolumeMounts:
    - name: log # logratote 与 controller 共享日志目录
      mountPath: /var/log/nginx
  extraContainers: # logrotate sidecar 容器，用于轮转日志
    - name: logrotate
      image: imroc/logrotate:latest # https://github.com/imroc/docker-logrotate
      imagePullPolicy: IfNotPresent
      env:
        - name: LOGROTATE_FILE_PATTERN # 轮转的日志文件 pattern，与 nginx 配置的日志文件路径相匹配
          value: "/var/log/nginx/nginx_*.log"
        - name: LOGROTATE_FILESIZE # 日志文件超过多大后轮转
          value: "100M"
        - name: LOGROTATE_FILENUM # 每个日志文件轮转的数量
          value: "3"
        - name: CRON_EXPR # logrotate 周期性运行的 crontab 表达式，这里每分钟一次
          value: "*/1 * * * *"
        - name: CROND_LOGLEVEL # crond 日志级别，0~8，越小越详细
          value: "8"
      volumeMounts:
        - name: log
          mountPath: /var/log/nginx
```

高可用配置优化

最近更新时间：2024-09-14 16:10:21

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

概述

本文介绍 Nginx Ingress 的高可用部署配置方法。

调高副本数

配置自动扩缩容：

```
controller:
  autoscaling:
    enabled: true
    minReplicas: 10
    maxReplicas: 100
    targetCPUUtilizationPercentage: 50
    targetMemoryUtilizationPercentage: 50
    behavior: # 快速扩容应对流量洪峰，缓慢缩容预留 buffer 避免流量异常
      scaleUp:
        stabilizationWindowSeconds: 300
        policies:
          - type: Percent
            value: 900
            periodSeconds: 15 # 每 15s 最多允许扩容 9 倍于当前副本数
      scaleDown:
        stabilizationWindowSeconds: 300
        policies:
          - type: Pods
            value: 1
            periodSeconds: 600 # 每 10 分钟最多只允许缩掉 1 个 Pod
```

如果希望固定副本数，直接配置 `replicaCount`：

```
controller:
  replicaCount: 50
```

打散调度

使用拓扑分布约束将 Pod 打散以支持容灾，避免单点故障：

```
controller:
  topologySpreadConstraints: # 尽量打散的策略
    - labelSelector:
        matchLabels:
          app.kubernetes.io/name: '{{ include "ingress-nginx.name" . }}'
          app.kubernetes.io/instance: '{{ .Release.Name }}'
          app.kubernetes.io/component: controller
      topologyKey: topology.kubernetes.io/zone
      maxSkew: 1
      whenUnsatisfiable: ScheduleAnyway
    - labelSelector:
        matchLabels:
```

```
          app.kubernetes.io/name: '{{ include "ingress-nginx.name" . }}'
          app.kubernetes.io/instance: '{{ .Release.Name }}'
          app.kubernetes.io/component: controller
      topologyKey: kubernetes.io/hostname
      maxSkew: 1
      whenUnsatisfiable: ScheduleAnyway
```

调度专用节点

通常 Nginx Ingress Controller 的负载跟流量成正比，由于 Nginx Ingress Controller 作为网关的重要性，可以考虑将其调度到专用的节点或者超级节点，避免干扰业务 Pod 或被业务 Pod 干扰。
调度到指定节点池：

```
controller:
  nodeSelector:
    tke.cloud.tencent.com/nodepool-id: np-********
```

> **说明：**
> 超级节点的效果更好，所有 Pod 独占虚拟机，不会相互干扰。如果使用的是 Serverless 集群，则不需要配这里的调度策略，只会调度到超级节点。

合理设置 request limit

如果 Nginx Ingress 不是调度到超级节点，需合理设置 request 和 limit，既要确保有足够的资源，也要避免使用过多资源导致节点负载过高：

```
controller:
  resources:
    requests:
      cpu: 500m
      memory: 512Mi
    limits:
      cpu: 1000m
      memory: 1Gi
```

如果使用的是超级节点或 Serverless 集群，只需要定义 requests，即声明每个 Pod 的虚拟机规格：

```
controller:
  resources:
    requests:
      cpu: 1000m
      memory: 2Gi
```

可观测性集成

最近更新时间：2025-06-13 09:23:42

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

概述

本文介绍如何配置 Nginx Ingress 来集成监控和日志系统以提升可观测性，包括与腾讯云上托管的 Prometheus、Grafana 和 CLS 这些产品的集成；也包括与自建的 Prometheus 和 Grafana 的集成。

集成 Prometheus 监控

如果您使用了 腾讯云 Prometheus 监控服务关联 TKE 集群，或者是安装了 Prometheus Operator 来监控集群，都可以启用 ServiceMonitor 来采集 Nginx Ingress 的监控数据，`values.yaml` 配置方法：

```
commonLabels:
  prom_id: prom-xxx # 通过这个 label 指定 Prometheus 实例的 ID，以便被 Prometheus 实例识别到 ServiceMonitor
controller:
  metrics:
    enabled: true # 专门创建一个 service 给 Prometheus 用作 Nginx Ingress 的服务发现
    serviceMonitor:
      enabled: true # 下发 ServiceMonitor 自定义资源，启用监控采集规则
```

集成 Grafana 监控面板

如果您使用了 腾讯云 Prometheus 监控服务关联 TKE 集群 且关联了 腾讯云 Grafana 服务，可以直接在 Prometheus 集成中心安装 Nginx Ingress 的监控面板：

如果是自建的 Grafana，直接将 Nginx Ingress 官方提供的 Grafana Dashboards 中两个监控面板（JSON 文件）导入 Grafana 即可。

集成 CLS 日志服务

以下内容将指导您如何将 Nginx Ingress Controller 的 access log 采集到 CLS，并结合 CLS 的仪表盘分析日志。

1. 在 `values.yaml` 中配置 Nginx 访问日志的格式，同时设置时区以便时间戳能展示当地时间（增强可读性）：

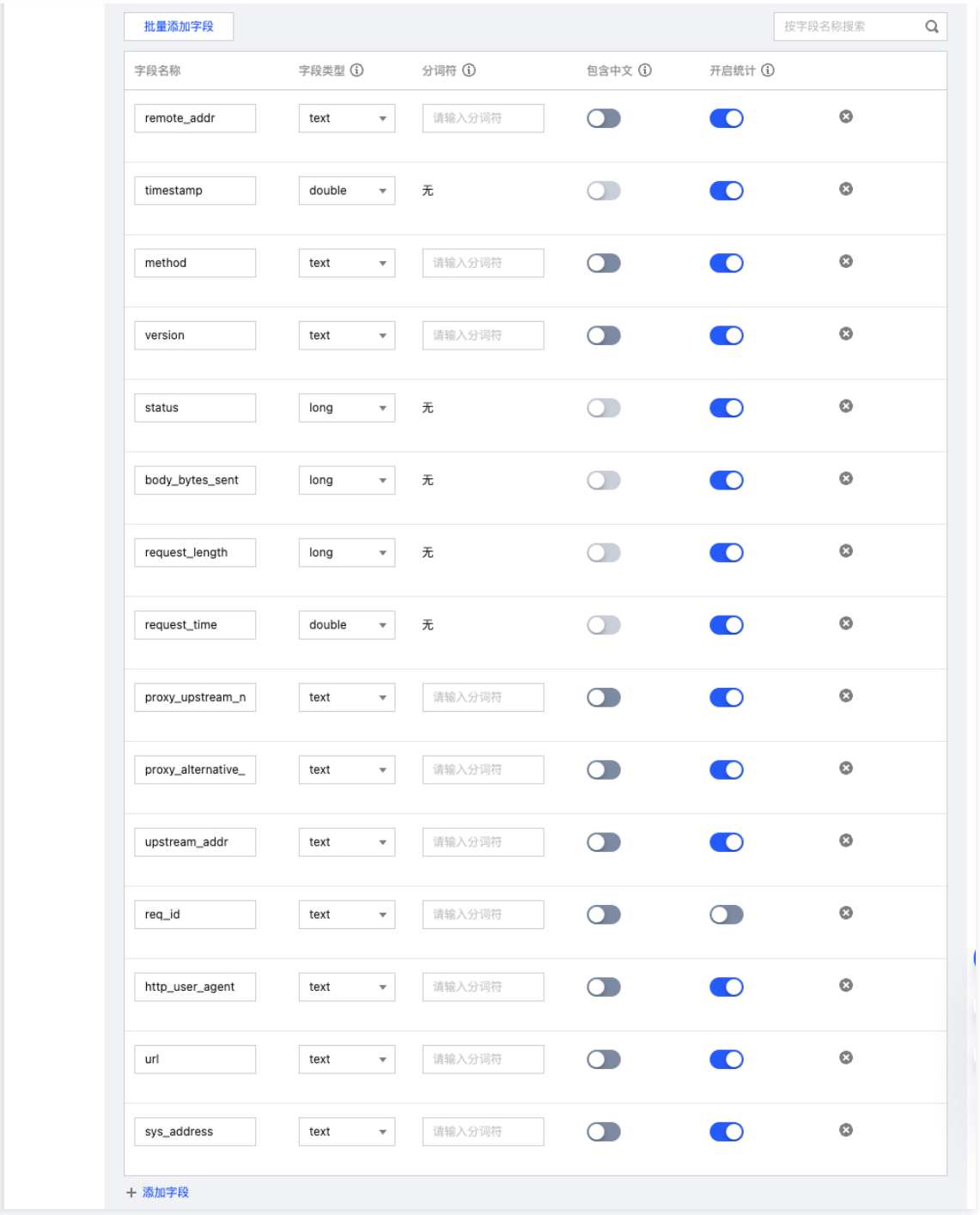
```
controller:
  config:
    log-format-upstream:
```

```
      $remote_addr - $remote_user [$time_local] "$request"
      $status $body_bytes_sent "$http_referer" "$http_user_agent"
      $request_length $request_time [$proxy_upstream_name] [$proxy_alternative_upstream_name]
$upstream_addr
      $upstream_response_length $upstream_response_time $upstream_status $req_id $host
  extraEnvs:
    - name: TZ
      value: Asia/Shanghai
```

2. 确保集群启用了日志采集功能，详情请参见 开启日志采集。
3. 为 Nginx Ingress Controller 准备好 CLS 日志集和日志主题，如果没有，请前往 CLS 控制台 根据自己的需求来创建，并记录日志主题的 ID。
4. 为日志主题开启索引：
   - 进入 日志主题 的**索引配置**页面，单击**编辑**：

   - 启用索引，全文分词符为 `@&?|#()='",;:<>[]{}/ \n\t\r\\` ：

   - 批量添加索引字段（需与下图中配置保持一致）：



- 高级设置：

5. 创建 TKE 日志采集规则（根据实际情况二选一）：

> **注意：**
> - 必须替换的配置项是 `topicId`，即日志主题 ID，表示采集的日志将会发送到该 CLS 日志主题里。
> - 根据自己实际情况选择配置采集标准输出还是日志文件，Nginx Ingress 默认是将日志输出到标准输出，您也可以选择将日志落盘到日志文件，详情请参见 日志轮转 。

- 采集标准输出：

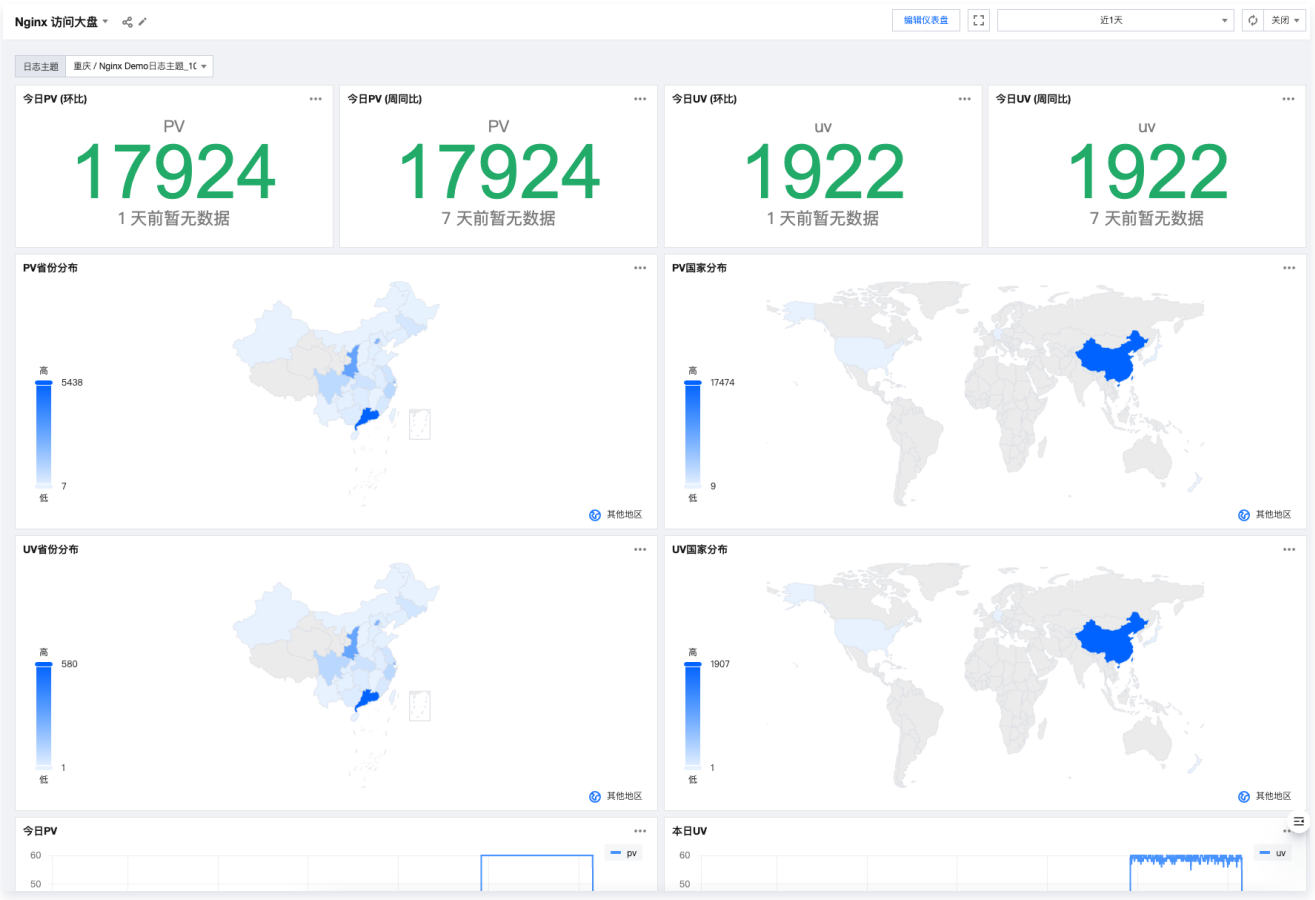
```
apiVersion: cls.cloud.tencent.com/v1
kind: LogConfig
metadata:
  name: ingress-nginx-controller # 日志采集规则名称，如果是多个 nginx ingress 实例，这里不能冲突
spec:
  clsDetail:
    topicId: "********-****-****-****-************" # 替换日志主题的 ID
    logType: fullregex_log
    extractRule:
      beginningRegex: (\S+)\s-\s(\S+)\s\[([^\]]+)\]\s\"(\w+)\s(\S+)\s([^\"]+)\"\s(\S+)\s(\S+)\s\"
([^\"]*)\"\s\"([^\"]*)\"\s(\S+)\s(\S+)\s\[([^\]]*)\]\s\
[([^\]]*)\]\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)
      logRegex: (\S+)\s-\s(\S+)\s\[([^\]]+)\]\s\"(\w+)\s(\S+)\s([^\"]+)\"\s(\S+)\s(\S+)\s\"
([^\"]*)\"\s\"([^\"]*)\"\s(\S+)\s(\S+)\s\[([^\]]*)\]\s\
[([^\]]*)\]\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)
      keys:
        - remote_addr
        - remote_user
        - timestamp
        - method
        - url
        - version
        - status
        - body_bytes_sent
        - http_referer
        - http_user_agent
        - request_length
        - request_time
        - proxy_upstream_name
        - proxy_alternative_upstream_name
        - upstream_addr
        - upstream_response_length
        - upstream_response_time
        - upstream_status
        - req_id
        - host
  inputDetail:
    type: container_stdout
    containerStdout:
      namespace: ingress-nginx # nginx ingress 所在命名空间
      workload:
        kind: deployment
        name: ingress-nginx-controller # 选中 nginx ingress controller 的 deployment 名称
```

- 采集日志文件：

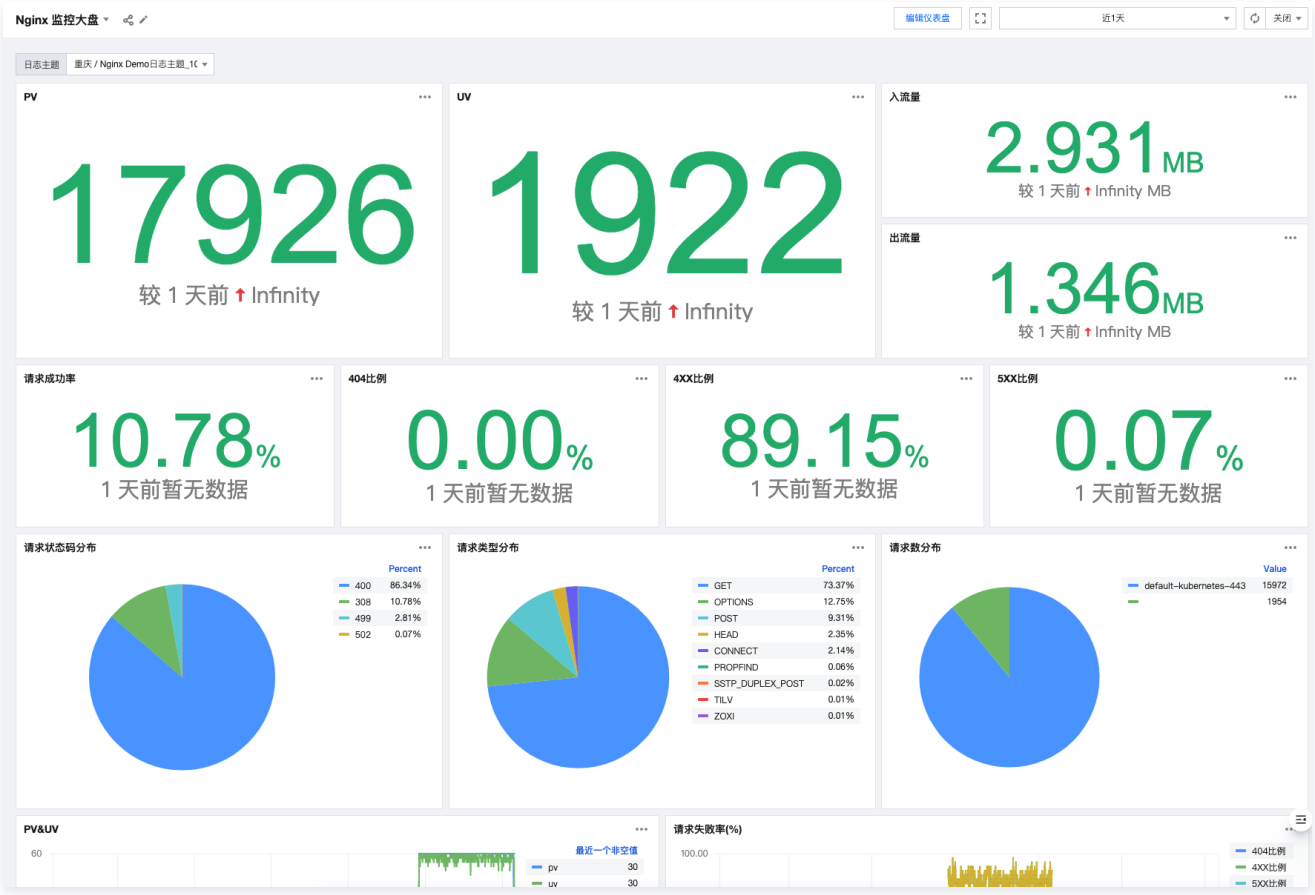
```
apiVersion: cls.cloud.tencent.com/v1
kind: LogConfig
metadata:
  name: ingress-nginx-controller # 日志采集规则名称，如果是多个 nginx ingress 实例，这里不能冲突
spec:
```

```
  clsDetail:
    topicId: "********-****-****-****-************" # 替换日志主题的 ID
    logType: fullregex_log
    extractRule:
      beginningRegex: (\S+)\s-\s(\S+)\s\[([^\]]+)\]\s\"(\w+)\s(\S+)\s([^\"]+)\"\s(\S+)\s(\S+)\s\"
([^\"]*)\"\s\"([^\"]*)\"\s(\S+)\s(\S+)\s\[([^\]]*)\]\s\
[([^\]]*)\]\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)
      logRegex: (\S+)\s-\s(\S+)\s\[([^\]]+)\]\s\"(\w+)\s(\S+)\s([^\"]+)\"\s(\S+)\s(\S+)\s\"
([^\"]*)\"\s\"([^\"]*)\"\s(\S+)\s(\S+)\s\[([^\]]*)\]\s\
[([^\]]*)\]\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)\s(\S+)
      keys:
        - remote_addr
        - remote_user
        - timestamp
        - method
        - url
        - version
        - status
        - body_bytes_sent
        - http_referer
        - http_user_agent
        - request_length
        - request_time
        - proxy_upstream_name
        - proxy_alternative_upstream_name
        - upstream_addr
        - upstream_response_length
        - upstream_response_time
        - upstream_status
        - req_id
        - host
  inputDetail:
    type: container_file
    containerFile:
      namespace: ingress-nginx # nginx ingress 所在命名空间
      workload:
        kind: deployment
        name: ingress-nginx-controller # 选中 nginx ingress controller 的 deployment 名称
      container: controller
      logPath: /var/log/nginx
      filePattern: nginx_access.log
```

6. 测试 Ingress 请求，产生日志数据。
7. 进入**日志服务控制台**的 检索分析 页面，选择 Nginx Ingress 所使用的日志主题，确认日志能够被正常检索。
8. 如果一切正常，可以使用**日志服务**的 Nginx 访问大盘 和 Nginx 监控大盘 两个预置仪表盘并选择 Nginx Ingress 所使用的日志主题来展示 Nginx 访问日志的分析面板：
   - Nginx 访问大盘


Nginx 访问大盘
今日PV (环比)
PV
17924
1 天前暂无数据
今日PV (周同比)
PV
17924
7 天前暂无数据
今日UV (环比)
uv
1922
1 天前暂无数据
今日UV (周同比)
uv
1922
7 天前暂无数据
PV省份分布
PV国家分布
UV省份分布
UV国家分布
今日PV
本日UV

- Nginx 监控大盘


Nginx 监控大盘
PV
17926
较 1 天前 Infinity
UV
1922
较 1 天前 Infinity
入流量
2.931 MB
较 1 天前 Infinity MB
出流量
1.346 MB
较 1 天前 Infinity MB
请求成功率
10.78%
1 天前暂无数据
404比例
0.00%
1 天前暂无数据
4XX比例
89.15%
1 天前暂无数据
5XX比例
0.07%
1 天前暂无数据
请求状态码分布
请求类型分布
请求数分布
PV&UV
请求失败率(%)

9. 在**日志服务**的 Nginx 访问大盘 和 Nginx 监控大盘 可以直接通过面板来设置监控告警规则，详情请参见 监控告警概述 。

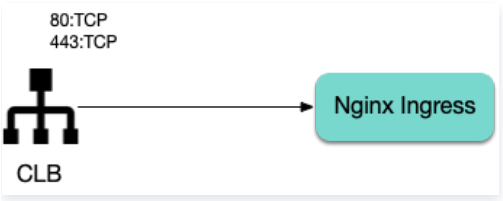
接入腾讯云 WAF

最近更新时间：2024-09-14 16:10:21

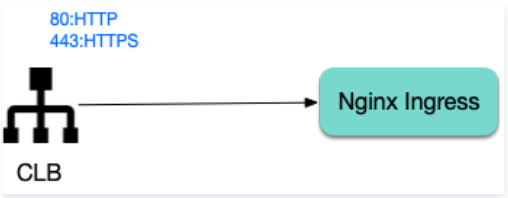
> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

背景

腾讯云 WAF （Web 应用防火墙）支持接入腾讯云负载均衡（CLB），但需要使用七层的监听器（HTTP/HTTPS）：



而 Nginx Ingress 默认使用四层的 CLB 监听器：



本文为您介绍将 Nginx Ingress 所使用的 CLB 监听器改为七层监听器。

使用 specify-protocol 注解

TKE 的 Service 支持使用 `service.cloud.tencent.com/specify-protocol` 这个注解来修改 CLB 监听器协议，详情请参见 Service 扩展协议 。
`values.yaml` 配置示例：

```
controller:
  service:
    annotations:
      service.cloud.tencent.com/specify-protocol: |
        {
          "80": {
            "protocol": [
              "HTTP"
            ],
            "hosts": {
              "a.example.com": {},
              "b.example.com": {}
            }
          },
          "443": {
            "protocol": [
              "HTTPS"
            ],
            "hosts": {
              "a.example.com": {
                "tls": "cert-secret-a"
              },
              "b.example.com": {
                "tls": "cert-secret-b"
              }
            }
```

```
        }
      }
```

- 实际使用的 Ingress 规则中涉及的域名，也需要在注解中的 `hosts` 字段进行配置。
- HTTPS 监听器需要证书，先在 我的证书 中创建好证书，然后在 TKE 集群中创建 Secret（需在 Nginx Ingress 所在的命名空间），Secret 的 Key 为 `qcloud_cert_id`，Value 为对应的证书 ID，然后在注解里引用 secret 名称。
- `targetPorts` 需要将 https 端口指向 nginx ingress 的 80 (http)，避免 CLB 的 443 流量转到 nginx ingress 的 443 端口（会导致双重证书，转发失败）。
- 不需要 HTTP 流量可以将 `enableHttp` 置为 false。

注意：

如果需要将 HTTP 的流量重定向到 HTTPS，可以在 CLB 控制台找到 nginx ingress 使用的 CLB 实例（实例 ID 可通过查看 nginx ingress controller 的 service 的 yaml 获取），在实例页面手动配置下重定向规则：

操作步骤

1. 在 我的证书 里上传证书并复制证书 ID。
2. 在 nginx ingress 所在 namespace 创建对应的证书 secret（引用证书 ID）：

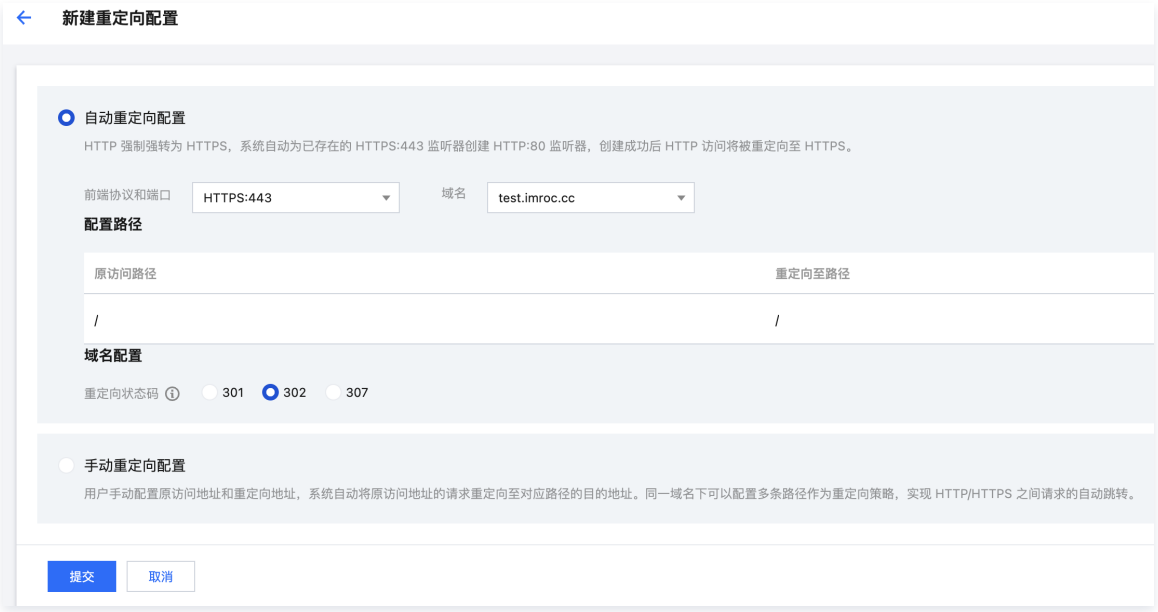
```
apiVersion: v1
kind: Secret
metadata:
  name: cert-secret-test
  namespace: ingress-nginx
stringData: # 用 stringData 就不需要手动 base64 转码
  # highlight-next-line
  qcloud_cert_id: E2pcp0Fy
type: Opaque
```

3. 配置 `values.yaml`：

```
controller: # 以下配置将依赖镜像替换为了 docker hub 上的 mirror 镜像以保证在国内环境能正常拉取
  image:
    registry: docker.io
    image: k8smirror/ingress-nginx-controller
  admissionWebhooks:
    patch:
      image:
        registry: docker.io
        image: k8smirror/ingress-nginx-kube-webhook-certgen
  defaultBackend:
    image:
      registry: docker.io
      image: k8smirror/defaultbackend-amd64
  opentelemetry:
    image:
      registry: docker.io
      image: k8smirror/ingress-nginx-opentelemetry
  service:
    enableHttp: false
```

```
    targetPorts:
      https: http
    annotations:
      service.cloud.tencent.com/specify-protocol: |
        {
          "80": {
            "protocol": [
              "HTTP"
            ],
            "hosts": {
              "test.example.com": {}
            }
          },
          "443": {
            "protocol": [
              "HTTPS"
            ],
            "hosts": {
              "test.example.com": {
                "tls": "cert-secret-test"
              }
            }
          }
        }
```

4. 如果需要，将 HTTP 自动重定向到 HTTPS，去 CLB 控制台配置下重定向规则：



5. 部署测试应用和 Ingress 规则：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
```

```
  selector:
    app: nginx
  type: NodePort
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:latest
          name: nginx
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: nginx
  rules:
    - host: test.example.com
      http:
        paths:
          - backend:
              service:
                name: nginx
                port:
                  number: 80
            path: /
            pathType: Prefix
```

6. 配置 hosts 或域名解析后，测试功能是否正常：

```
❯ curl -v http://[redacted]cc
*   Trying [redacted]...
* Connected to [redacted] port 80
> GET / HTTP/1.1
> Host: [redacted]
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 302 Moved Temporarily
< Server: stgw
< Date: Sat, 13 Apr 2024 03:47:41 GMT
< Content-Type: text/html
< Content-Length: 137
< Connection: keep-alive
< Location: HTTPS://[redacted]c/
<
<html>
<head><title>302 Found</title></head>
<body>
<center><h1>302 Found</h1></center>
<hr><center>stgw</center>
</body>
</html>
```

```
❯ curl https://[redacted]
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is succe
working. Further configuration is required.</p>
```

配置 WAF

Nginx Ingress 配置好后，确认对应的 CLB 监听器已经改为了 HTTP/HTTPS，则已满足 Nginx Ingress 接入 WAF 的前提条件，可以根据 WAF 官方文档 的指引来进行配置，最终完成 Nginx Ingress 的 WAF 接入。

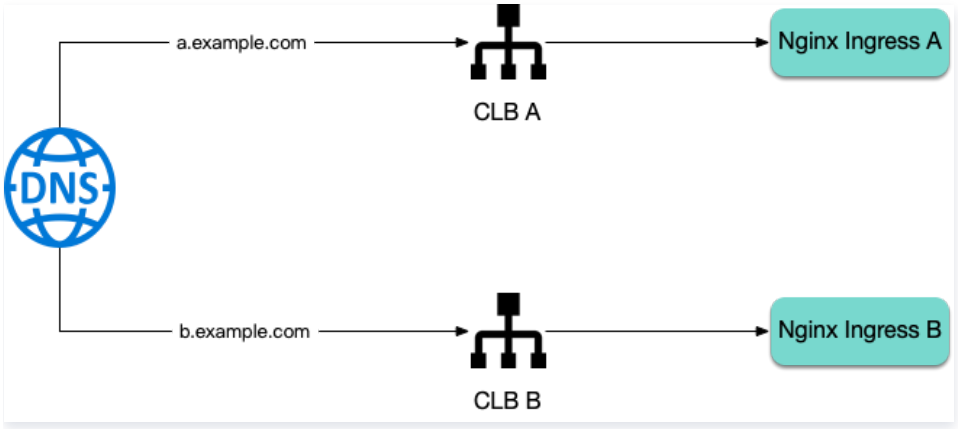
安装多个 Nginx Ingress Controller

最近更新时间：2024-09-14 16:10:21

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

概述

如果您需要部署多个 Nginx Ingress Controller，即希望不同的 Ingress 规则使用不同的流量入口：



您可以为集群部署多个 Nginx Ingress Controler，不同的 Ingress 指定不同的 ingressClassName 来实现。
本文介绍安装多个 Nginx Ingress Controller 的配置方法。

配置方法

如果要安装多个 Nginx Ingress Controller，需要在 `values.yaml` 指定 ingressClass（注意不要冲突）：

```
controller:
  ingressClass: prod
  ingressClassResource:
    name: prod
    controllerValue: k8s.io/ingress-prod
```

> **说明：**
> 三个字段需同时改。

另外，多实例的 release 名称也不能与已安装的相同，**即便是 namespace 不同，release 名称也不能相同**（避免 ClusterRole 冲突），示例代码如下：

```
helm upgrade --install prod ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  -f values.yaml
```

在创建 Ingress 资源时也要指定对应的 `ingressClassName`：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: prod
  rules:
```

```
  - http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: nginx
              port:
                number: 80
```

从 TKE Nginx Ingress 插件迁移到自建 Nginx Ingress

最近更新时间：2024-09-14 16:10:21

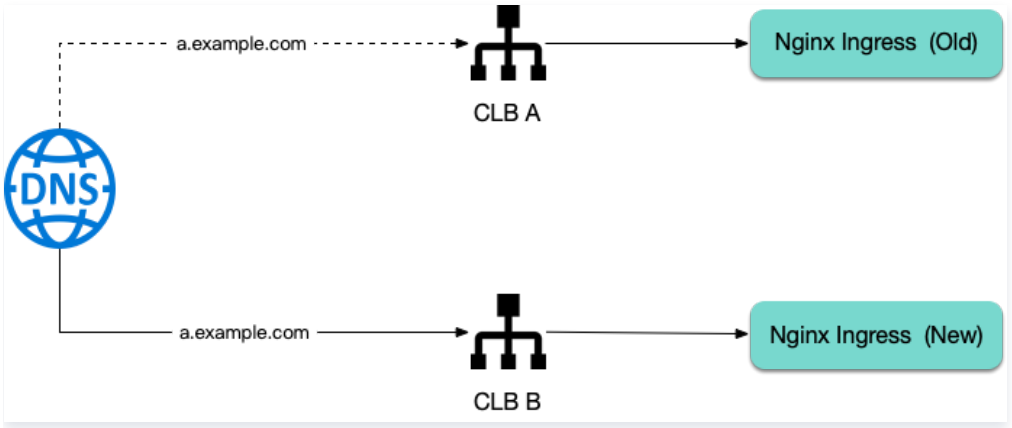
> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

迁移的好处

Nginx Ingress 提供的功能和配置都是非常多和灵活，可以满足各种使用场景，自建可以解锁 Nginx Ingress 的全部功能，并根据自己需求，对配置进行自定义，还能够及时更新版本。

迁移思路

使用本文中自建的方法创建一套新的 Nginx Ingress 实例以及 Ingress 规则，两套流量入口共存，最后修改 DNS 指向新的入口地址，完成平滑迁移。



确认已安装的 Nginx Ingress 相关信息

1. 先确认已安装的 Nginx Ingress 实例的 IngressClass 名称，例如：

```
$ kubectl get deploy -A | grep nginx
kube-system            extranet-ingress-nginx-controller           1/1     1            1
216d
```

在本例中，只有一个实例，Deployment 名称是 `extranet-ingress-nginx-controller`，IngressClass 是 `-ingress-nginx-controller` 之前的部分，即 `extranet` 。

2. 确认当前使用的 nginx ingress 的镜像版本：

```
$ kubectl -n kube-system get deploy extranet-ingress-nginx-controller -o yaml | grep image:
        image: ccr.ccs.tencentyun.com/tkeimages/nginx-ingress-controller:v1.9.5
```

本例中的镜像版本是 `v1.9.5` 。

3. 确认当前使用的 chart 版本：

```
$ helm search repo ingress-nginx/ingress-nginx --versions  | grep 1.9.5
ingress-nginx/ingress-nginx     4.9.0           1.9.5           Ingress controller for Kubernetes
using NGINX a...
```

本例中 chart 版本为 `4.9.0` ，请记住这个版本，后续在使用 helm 安装新版渲染时需要指定这个 chart 版本。

准备 values.yaml

下面配置 `values.yaml` ，确保 helm 新创建的 Nginx Ingress 实例和 TKE 插件创建 Nginx Ingress 实例不要共用同一个 IngressClass：

```
controller:
  ingressClass: extranet-new # 新 IngressClass 名称，避免冲突
  ingressClassResource:
    name: extranet-new
    enabled: true
    controllerValue: k8s.io/extranet-new
```

安装新的 Nginx Ingress Controller

```
helm upgrade --install new-extranet-ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  --version 4.9.0 \
  -f values.yaml
```

- 避免在 release 名称加上 `-controller` 后缀导致其与已有的 Nginx Ingress Deployment 名称相同，如果有同名的 ClusterRole 存在会导致 helm 安装失败。
- version 指定在前面步骤得到的 chart 版本（即当前 nginx ingress 实例版本对应的 chart 版本）。

获取到新的 Nginx Ingress 的流量入口：

```
$ kubectl -n ingress-nginx get svc
NAME                                          TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)                      AGE
new-extranet-ingress-nginx-controller         LoadBalancer   172.16.165.100   43.136.214.239
80:31507/TCP,443:31116/TCP   9m37s
```

`EXTERNAL-IP` 是新的流量入口，后续用该入口验证是否正常转发。

复制 Ingress 资源

将使用旧 IngressClass 的 Ingress 资源的 YAML 文件保存下来，并修改其名称（例如添加后缀 `-new`）。然后，将修改后的 YAML 文件应用到集群中。这样，新旧 Nginx Ingress 实例的转发规则将保持一致，确保流量进入任意入口时效果相同。

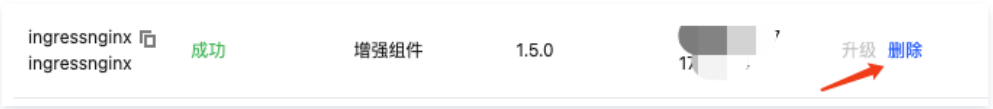
切换 DNS

至此，新旧 Nginx Ingress 共存，无论通过哪个流量入口都能正常转发。
接下来修改域名的 DNS 解析，指向新 Nginx Ingress 流量入口，在 DNS 解析完全生效前，两边流量入口均能正常转发，无论通过哪个流量入口都能正常转发，这个过程会非常平滑，生产环境的流量不受影响。

删除旧 NginxIngress 实例和插件

1. 等到所有旧的 Nginx Ingress 实例完全没有流量的时候，前往 TKE 控制台删除 Nginx Ingress 实例：

2. 在**组件管理**中删除 `ingressnginx`，彻底完成迁移。


ingressnginx
ingressnginx
成功
增强组件
1.5.0
升级
删除

values.yaml 完整配置示例

最近更新时间：2024-09-14 16:10:21

> **注意：**
> ingress-nginx 是一个由社区维护的开源项目。本文内容仅供参考，不提供官方支持。如在使用过程中遇到问题，建议查阅 ingress-nginx 社区的最新官方文档以获取帮助。

以下是比较完整的 `values.yaml` 配置示例，您可复制此示例，并根据自己需求来进行修改：

```
controller:
  extraInitContainers:
    - name: sysctl
      image: busybox
      securityContext:
        privileged: true
      imagePullPolicy: IfNotPresent
      command:
        - sh
        - -c
        - |
          sysctl -w net.core.somaxconn=65535 # 调大链接队列，防止队列溢出
          sysctl -w net.ipv4.ip_local_port_range="1024 65535" # 扩大源端口范围，防止端口耗尽
          sysctl -w net.ipv4.tcp_tw_reuse=1 # TIME_WAIT 复用，避免端口耗尽后无法新建连接
          sysctl -w fs.file-max=1048576 # 调大文件句柄数，防止连接过多导致文件句柄耗尽
  config:
    # nginx 与 client 保持的一个长连接能处理的请求数量，默认100，高并发场景建议调高，但过高也可能导致 nginx ingress 扩容后负载不均。
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#keep-alive-requests
    keep-alive-requests: "1000"
    # nginx 与 upstream 保持长连接的最大空闲连接数 （不是最大连接数），默认 320，在高并发下场景下调大，避免频繁建联导致 TIME_WAIT 飙升。
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#upstream-keepalive-connections
    upstream-keepalive-connections: "2000"
    # 每个 worker 进程可以打开的最大连接数，默认 16384。
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#max-worker-connections
    max-worker-connections: "65536"
    log-format-upstream: $remote_addr - $remote_user [$time_local] "$request"
      $status $body_bytes_sent "$http_referer" "$http_user_agent"
      $request_length $request_time [$proxy_upstream_name] [$proxy_alternative_upstream_name]
$upstream_addr
      $upstream_response_length $upstream_response_time $upstream_status $req_id $host
    # nginx 日志落盘到日志文件，避免高并发下占用过多 CPU
    access-log-path: /var/log/nginx/nginx_access.log
    error-log-path: /var/log/nginx/nginx_error.log
  extraEnvs:
    - name: TZ
      value: Asia/Shanghai
  extraVolumes:
    - name: log
      emptyDir: {}
  extraVolumeMounts:
    - name: log
      mountPath: /var/log/nginx
  extraContainers:
```

```
    - name: logrotate
      image: imroc/logrotate:latest
      imagePullPolicy: Always
      env:
        - name: LOGROTATE_FILE_PATTERN # 轮转的日志文件pattern，与 nginx 配置的日志文件路径相匹配
          value: "/var/log/nginx/nginx_*.log"
        - name: LOGROTATE_FILESIZE # 日志文件超过多大后轮转
          value: "100M"
        - name: LOGROTATE_FILENUM # 每个日志文件轮转的数量
          value: "3"
        - name: CRON_EXPR # logrotate 周期性运行的 crontab 表达式，这里每分钟一次
          value: "*/1 * * * *"
        - name: CROND_LOGLEVEL # crond 日志级别，0~8，越小越详细
          value: "8"
      volumeMounts:
        - name: log
          mountPath: /var/log/nginx
  podAnnotations:
    tke.cloud.tencent.com/networks: tke-route-eni # 声明使用 VPC-CNI 网络
  resources: #  resources 里声明使用弹性网卡
    requests:
      tke.cloud.tencent.com/eni-ip: "1"
    limits:
      tke.cloud.tencent.com/eni-ip: "1"
  service:
    annotations:
      service.cloud.tencent.com/direct-access: "true" # 启用 CLB 直通
  autoscaling:
    enabled: true
    minReplicas: 1
    maxReplicas: 10
    targetCPUUtilizationPercentage: 50
    targetMemoryUtilizationPercentage: 50
    behavior: # 快速扩容应对流量洪峰，缓慢缩容预留 buffer 避免流量异常
      scaleUp:
        stabilizationWindowSeconds: 300
        policies:
          - type: Percent
            value: 900
            periodSeconds: 15 # 每 15s 最多允许扩容 9 倍于当前副本数
      scaleDown:
        stabilizationWindowSeconds: 300
        policies:
          - type: Pods
            value: 1
            periodSeconds: 600 # 每 10 分钟最多只允许缩掉 1 个 Pod
  topologySpreadConstraints: # 尽量打散的策略
    - labelSelector:
        matchLabels:
          app.kubernetes.io/name: '{{ include "ingress-nginx.name" . }}'
          app.kubernetes.io/instance: "{{ .Release.Name }}"
          app.kubernetes.io/component: controller
      topologyKey: topology.kubernetes.io/zone
      maxSkew: 1
      whenUnsatisfiable: ScheduleAnyway
    - labelSelector:
        matchLabels:
          app.kubernetes.io/name: '{{ include "ingress-nginx.name" . }}'
          app.kubernetes.io/instance: "{{ .Release.Name }}"
          app.kubernetes.io/component: controller
```

```
      topologyKey: kubernetes.io/hostname
      maxSkew: 1
      whenUnsatisfiable: ScheduleAnyway
  image:
    registry: docker.io
    image: k8smirror/ingress-nginx-controller
  admissionWebhooks:
    patch:
      image: # 默认的镜像在境内无法拉取，可替换为 docker hub 上的 mirror 镜像
        registry: docker.io
        image: k8smirror/ingress-nginx-kube-webhook-certgen
  defaultBackend:
    image: # 默认的镜像在境内无法拉取，可替换为 docker hub 上的 mirror 镜像
      registry: docker.io
      image: k8smirror/defaultbackend-amd64
  opentelemetry:
    image: # 默认的镜像在境内无法拉取，可替换为 docker hub 上的 mirror 镜像
      registry: docker.io
      image: k8smirror/ingress-nginx-opentelemetry
```

ingress-nginx 应用部署指南

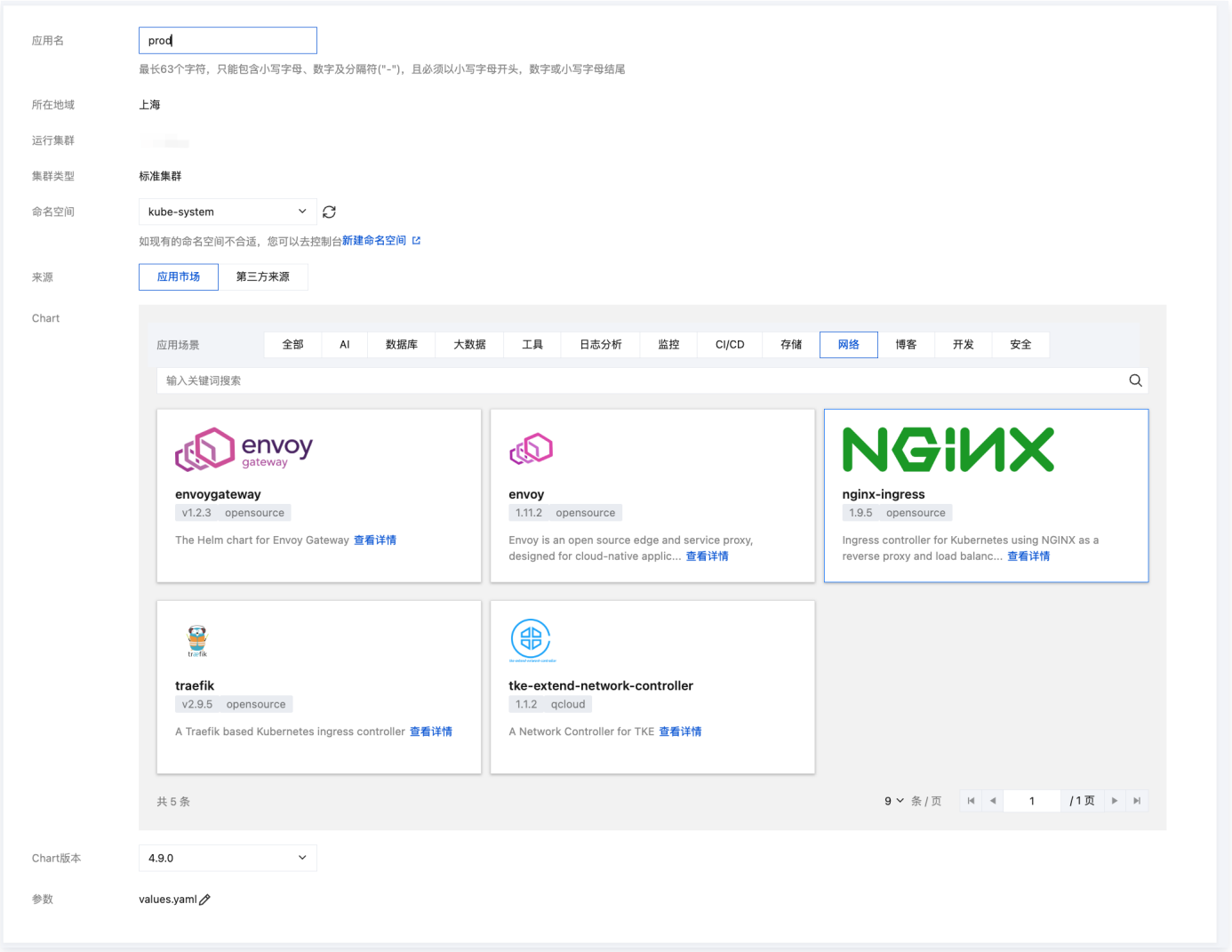
最近更新时间：2025-03-06 16:50:32

本指南将详细介绍如何在腾讯云容器服务 TKE 控制台的应用市场中部署社区版 ingress-nginx，并提供其关键参数的配置方法。

> **注意：**
> ingress-nginx 应用基于开源的 ingress-nginx 构建，腾讯云容器服务不提供 SLA 保障。

通过控制台创建应用

1. 登录 容器服务控制台，选择左侧导航栏中的 运维中心 > 应用市场。
2. 在应用管理页面上方，选择需创建应用的集群及地域，并单击**新建**。
3. 在**新建应用**页面，选择 nginx-ingress 应用。如下图所示：



4. 选择 Chart 版本，推荐版本如下：

Chart 版本	ingress-nginx 应用版本	推荐的 TKE 版本
4.2.5	v1.3.1	1.20，1.22，1.24
4.9.0	v1.9.5	1.26，1.28，1.30

5. 单击参数 Values.yaml，进入参数配置界面，按需调整相关参数。

参数配置说明

工作负载类型

支持以下两种工作负载类型：

- Deployment（默认）
- Daemonset

```
controller:
  # -- Use a `DaemonSet` or `Deployment`
  kind: Deployment
```

资源设置

通过设置 requests 和 limits 为 ingress-nginx 分配资源。

```
controller:
  resources:
    ##  limits:
    ##    cpu: 100m
    ##    memory: 90Mi
    requests:
      cpu: 100m
      memory: 90Mi
```

ingressClass 设置

默认的 ingressClass 为 nginx，可以通过设置不同值实现多个 ingress-nginx 应用的部署。

```
controller:
  ...
  ingressClass: prod
  ingressClassResource:
    name: prod
    controllerValue: k8s.io/ingress-prod
```

在创建 Ingress 资源时，需要显式指定对应的 ingressClassName。

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: prod
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: example
                port:
                  number: 80
```

调度设置

支持以下调度方式：

- nodeSelector
- tolerations（容忍）
- affinity（亲和性）

示例配置如下：

```
controller:
  # nodeSelector
  nodeSelector:
    kubernetes.io/os: linux
  tolerations: []
  #  - key: "key"
  #    operator: "Equal|Exists"
  #    value: "value"
  #    effect: "NoSchedule|PreferNoSchedule|NoExecute(1.6 only)"
  affinity: {}
  # # An example of preferred pod anti-affinity, weight is in the range 1-100
  # podAntiAffinity:
  #   preferredDuringSchedulingIgnoredDuringExecution:
  #   - weight: 100
  #     podAffinityTerm:
  #       labelSelector:
  #         matchExpressions:
  #         - key: app.kubernetes.io/name
  #           operator: In
  #           values:
  #           - '{{ include "ingress-nginx.name" . }}'
  #         - key: app.kubernetes.io/instance
  #           operator: In
  #           values:
  #           - '{{ .Release.Name }}'
  #         - key: app.kubernetes.io/component
  #           operator: In
  #           values:
  #           - controller
  #       topologyKey: kubernetes.io/hostname

  # # An example of required pod anti-affinity
  # podAntiAffinity:
  #   requiredDuringSchedulingIgnoredDuringExecution:
  #   - labelSelector:
  #       matchExpressions:
  #       - key: app.kubernetes.io/name
  #         operator: In
  #         values:
  #         - '{{ include "ingress-nginx.name" . }}'
  #       - key: app.kubernetes.io/instance
  #         operator: In
  #         values:
  #         - '{{ .Release.Name }}'
  #       - key: app.kubernetes.io/component
  #         operator: In
  #         values:
  #         - controller
  #     topologyKey: kubernetes.io/hostname
```

请求入口配置

公网入口

请注意，启用公网入口会创建公网 CLB 资源。

```
controller:
  ...
  service:
    enabled: true
    external:
      enabled: true
    annotations:
      service.cloud.tencent.com/direct-access: "true" # 开启直连 Pod 模式
      service.cloud.tencent.com/enable-grace-shutdown: "true"   # 表示使用优雅停机
      service.cloud.tencent.com/enable-grace-shutdown-tkex: "true"
    type: LoadBalancer
```

内网入口

请注意，启用内网入口会创建内网 CLB 资源。

```
controller:
  ...
  service:
    ...
    internal:
      enabled: true
      annotations:
        service.cloud.tencent.com/direct-access: "true" # 开启直连 Pod 模式
        service.cloud.tencent.com/enable-grace-shutdown: "true"   # 表示使用优雅停机
        service.cloud.tencent.com/prevent-loopback: "true" # 规避回环
        service.cloud.tencent.com/enable-grace-shutdown-tkex: "true"
        service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: "subnet-xxx" # VPC 子网 ID
      type: "LoadBalancer"
```

> **说明：**
> 如果希望使用已有的 CLB，可以通过如下注解实现：`service.kubernetes.io/tke-existed-lbid: <LoadBalanceId>`。更多关于 Service 的配置信息，请参见 Service 使用已有 CLB。

弹性伸缩 HPA

通过 Deployment 工作负载类型部署时，可以配置以下弹性伸缩策略。

```
controller:
  ...
  autoscaling:
    enabled: true
    annotations: {}
    minReplicas: 1
    maxReplicas: 11
    targetCPUUtilizationPercentage: 50
    targetMemoryUtilizationPercentage: 50
    behavior: {}
    # scaleDown:
    #   stabilizationWindowSeconds: 300
    #   policies:
    #   - type: Pods
    #     value: 1
    #     periodSeconds: 180
    # scaleUp:
    #   stabilizationWindowSeconds: 300
```

```
    #   policies:
    #   - type: Pods
    #     value: 2
    #     periodSeconds: 60
```

TKE 还支持基于自定义指标的弹性伸缩，示例如下。更多 TKE 支持的自定义指标请参见 自动伸缩指标说明 。

```
controller:
  ...
  autoscalingTemplate:
  - pods:
      metric:
        name: k8s_pod_rate_cpu_core_used_limit
      target:
        averageValue: "80"
        type: AverageValue
    type: Pods
```

在 TKE 使用 EnvoyGateway 流量网关

最近更新时间：2025-03-07 18:49:32

什么是 Envoy Gateway?

Envoy Gateway 是基于 Envoy 实现 Gateway API 的 Kubernetes 网关。通过定义 `Gateway API` 中的 `Gateway`、`HTTPRoute` 等资源，您可以管理 Kubernetes 的南北向流量。

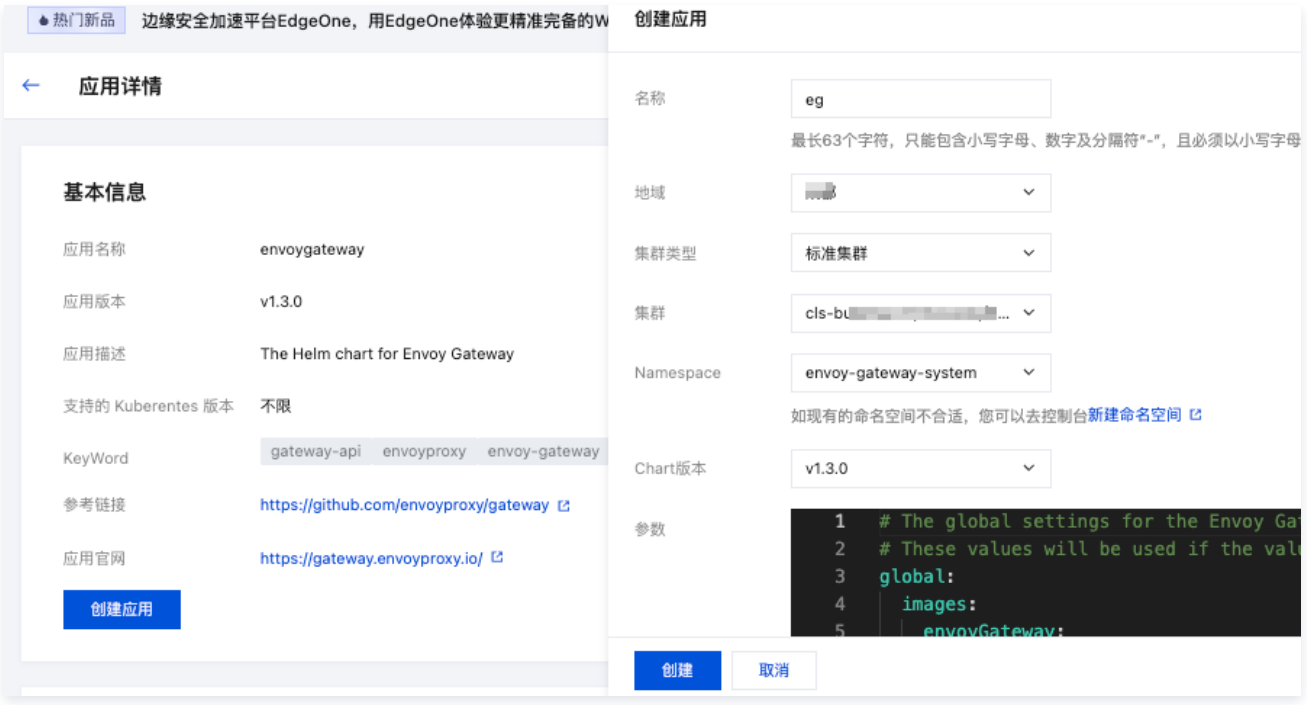
为什么要用 Gateway API?

Kubernetes 提供了 `Ingress API` 来接入七层南北向流量，但功能较弱，每种实现都带有不同的 annotation 来增强 Ingress 的能力，灵活性和扩展性较差。社区推出了 `Gateway API` 作为更好的解决方案，解决 Ingress API 的痛点，同时统一了四七层南北向流量，并支持服务网格的东西向流量（参考 GAMMA），各个云厂商以及开源代理软件都在积极适配 `Gateway API`，可参考 Gateway API 的实现列表，其中 Envoy Gateway 是一个很流行的实现。

安装 Envoy Gateway

方法一：通过应用市场安装

1. 在目标 TKE 集群中新建一个名为 `envoy-gateway-system` 的命名空间。操作详情请参见 创建命名空间。
2. 在 TKE 应用市场 搜索并选择 `envoygateway`。
3. 单击**创建应用**，选择好目标 TKE 集群，名称可写 `eg`，命名空间选 `envoy-gateway-system`。



4. 根据需求配置参数完后，单击**创建**即可将 Envoy Gateway 安装到集群中。

方法二：按照 Envoy Gateway 官方文档安装

参考 Envoy Gateway Installation。

配置 kubectl 访问集群

Envoy Gateway 使用的是 Gateway API 而不是 Ingress API，在 TKE 控制台无法直接创建，可通过 kubectl 命令进行创建，请参考 连接集群 这篇文档配置 kubectl。

创建 GatewayClass

类似 `Ingress` 需要指定 `IngressClass`，Gateway API 中每个 `Gateway` 都需要引用一个 `GatewayClass`，`GatewayClass` 相当于是网关实例除监听器外的配置（如部署方式、网关 Pod 的 template、副本数量、关联的 Service 等），所以先创建一个 `GatewayClass`：

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: eg
spec:
  controllerName: gateway.envoyproxy.io/gatewayclass-controller
```

说明：
`GatewayClass` 是 non-namespaced 资源，无需指定命名空间。

创建 Gateway

每个 `Gateway` 对应一个 CLB，在 `Gateway` 上声明端口相当于在 CLB 上创建响应协议的监听器：

说明：
Gateway 的所有字段参考 API Specification: Gateway 。

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: test-gw
  namespace: test
spec:
  gatewayClassName: eg
  listeners:
    - name: http
      protocol: HTTP
      port: 8080
      allowedRoutes:
        namespaces:
          from: All
```

说明：
`Gateway` 可以指定命名空间，可以被 `HTTPRoute` 等路由规则跨命名空间引用。

`Gateway` 创建后，`EnvoyGateway` 会自动为其创建一个 LoadBalancer 类型的 Service，也就是一个 CLB。在 TKE 上，LoadBalancer 类型的 Service 默认是一个公网 CLB，如果要自定义，可参考常见问题中的 如何自定义 CLB 。

说明：
Gateway 通过 LoadBalancer 类型的 Service 对外暴露流量，所以 CLB 只会用到四层监听器（TCP/UDP），七层流量也是先进入 CLB 四层监听器，转发给 EnvoyGateway 的 Pod，再由 EnvoyGateway 解析四七层流量并根据配置规则进行转发。

如何获取 `Gateway` 对应的 CLB 地址呢？可以通过 `kubectl get gtw` 查看：

```
$ kubectl get gtw test-gw -n test
NAME      CLASS   ADDRESS          PROGRAMMED   AGE
test-gw   eg      139.155.64.52    True         358d
```

其中 `ADDRESS` 就是 CLB 的地址（IP 或域名）。

创建 HTTPRoute

`HTTPRoute` 用于定义 HTTP 转发规则（七层流量），也是 Gateway API 中最常用的转发规则，类似 Ingress API 中的 `Ingress` 资源。
下面给出一个示例：

> **说明：**
> HTTPRoute 的所有字段参考 API Specification: HTTPRoute 。

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: nginx
  namespace: test
spec:
  parentRefs:
  - name: test-gw
    namespace: test
  hostnames:
  - "test.example.com"
  rules:
  - backendRefs:
    - name: nginx
      port: 80
```

> **注意：**
> 1. `parentRefs` 中指定要引用 `Gateway`（CLB），表示将该规则应用到这个 `Gateway` 中。
> 2. `hostnames` 定义转发规则使用的的域名，确保该域名解析到 `Gateway` 对应的 CLB，这样可以通过域名访问集群内的服务。
> 3. `backendRefs` 定义该条转发规则对应的后端 Service。

创建 TCPRoute 和 UDPRoute

`TCPRoute` 和 `UDPRoute` 用于定义 TCP 和 UDP 转发规则（四层流量），类似 `LoadBalancer` 类型的 `Service` 。
下面是 `TCPRoute` 的示例：

```
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: foo
spec:
  parentRefs:
  - namespace: test
    name: test-gw
    sectionName: foo
  rules:
  - backendRefs:
    - name: foo
      port: 6000
```

> **注意：**
> 1. `parentRefs` 指定要引用的 `Gateway`（CLB），表示将该 TCP 要监听到这个 `Gateway` 中。通常只使用 `Gateway` 中的一个端口，所以指定 `sectionName` 来指定使用哪个监听器暴露。
> 2. `backendRefs` 定义该条转发规则对应的后端 Service。

下面是 `UDPRoute` 的示例，与 `TCPRoute` 类似：

```
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: UDPRoute
metadata:
  name: bar
```

```
spec:
  parentRefs:
  - namespace: test
    name: test-gw
    sectionName: bar
  rules:
  - backendRefs:
    - name: bar
      port: 6000
```

引用的 `Gateway` 示例：

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: test-gw
  namespace: test
spec:
  gatewayClassName: eg
  listeners:
  - name: foo
    protocol: TCP
    port: 6000
    allowedRoutes:
      namespaces:
        from: All
  - name: bar
    protocol: UDP
    port: 6000
    allowedRoutes:
      namespaces:
        from: All
```

常见问题

如何自定义 CLB?

可通过创建 `EnvoyProxy` 自定义资源来自定义，下面是示例：

```
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: proxy-config
  namespace: test
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyDeployment:
        replicas: 1
        container:
          resources:
            requests:
              tke.cloud.tencent.com/eni-ip: "1"
            limits:
              tke.cloud.tencent.com/eni-ip: "1"
        pod:
          annotations:
            tke.cloud.tencent.com/networks: tke-route-eni
```

```
    envoyService:
      annotations:
        service.kubernetes.io/tke-existed-lbid: lb-5nhlk3nr
        service.cloud.tencent.com/direct-access: "true"
```

以上示例中：

- 显式声明使用 VPC-CNI 网络模式且启用 CLB 直连 Pod。
- 使用已有 CLB，指定了 CLB 的 ID。

相应的，`GatewayClass` 中需引用该 `EnvoyProxy` 配置：

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: eg
spec:
  controllerName: gateway.envoyproxy.io/gatewayclass-controller
  parametersRef:
    group: gateway.envoyproxy.io
    kind: EnvoyProxy
    name: proxy-config
    namespace: test
```

更多 CLB 相关的自定义可参考 Service Annotation 说明 。

举几个常见的自定义例子：

1. 通过 `service.cloud.tencent.com/specify-protocol` 注解来修改监听器协议为 HTTPS 并正确引用 SSL 证书，以便让 CLB 能够接入 腾讯云 WAF 。
2. 通过 `service.kubernetes.io/qcloud-loadbalancer-internal-subnetid` 注解指定 CLB 内网 IP，实现自动创建内网 CLB 来接入流量。
3. 通过 `service.kubernetes.io/service.extensiveParameters` 注解自定义自动创建的 CLB 更多属性，如指定运营商、带宽上限、实例规格、网络计费模式等。

多个 HTTPRoute 如何复用同一个 CLB?

通常一个 `Gateway` 对象就对应一个 CLB，只要不同 `HTTPRoute` 的 `parentRefs` 引用的是同一个 `Gateway` 对象，那么它们就会复用同一个 CLB。

> **注意：**
> 如果多个 `HTTPRoute` 复用同一个 CLB，确保它们定义的 HTTP 规则不要冲突，否则可能转发行为可能不符预期。

下面给个示例，第一个 `HTTPRoute`，引用 Gateway `test-gw`，使用域名 `test1.example.com`：

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: test1
  namespace: test
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: test-gw
      namespace: test
  hostnames:
    - "test1.example.com"
  rules:
    - backendRefs:
        - group: ""
          kind: Service
          name: test1
```

```
          port: 80
```

第二个 `HTTPRoute`，也引用 Gateway `test-gw`，域名则使用 `test2.example.com`：

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: test2
  namespace: test
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: test-gw
      namespace: test
  hostnames:
    - "test2.example.com"
  rules:
    - backendRefs:
        - group: ""
          kind: Service
          name: test2
          port: 80
```

如何实现四七层共用同一个 CLB?

使用 TKE 自带的 `LoadBalancer` 类型的 `Service`，可以实现多个 `Service` 复用同一个 CLB，也就是多个四层端口（TCP/UDP）复用同一个 CLB；使用 TKE 自带的 `Ingress`（CLB Ingress），无法与任何其它 `Ingress` 和 `LoadBalancer` 类型的 `Service` 复用同一个 CLB。所以，如果需要实现四七层共用同一个 CLB，直接使用 TKE 自带的 CLB Service 和 CLB Ingress 无法实现，而如果你安装了 `EnvoyGateway` 的话就可以实现。
下面给个示例，首先 `Gateway` 的监听器声明四层和七层的端口：

> **注意：**
> 使用 `name` 给每个监听器取个名字，方便后续通过 `sectionName` 引用。

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: test-gw
  namespace: test
spec:
  gatewayClassName: eg
  listeners:
  - name: http
    protocol: HTTP
    port: 80
    allowedRoutes:
      namespaces:
        from: All
  - name: https
    protocol: HTTPS
    port: 443
    allowedRoutes:
      namespaces:
        from: All
    tls:
      mode: Terminate
      certificateRefs:
```

```
      - kind: Secret
        group: ""
        name: https-cert
  - name: tcp-6000
    protocol: TCP
    port: 6000
    allowedRoutes:
      namespaces:
        from: All
  - name: udp-6000
    protocol: UDP
    port: 6000
    allowedRoutes:
      namespaces:
        from: All
```

`HTTPRoute` 里使用 `Gateway` 里的七层监听器（80 和 443）：

> ⚠ **注意：**
> 使用 `sectionName` 指定具体要绑定的监听器。

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: test
  namespace: test
spec:
  parentRefs:
  - name: test-gw
    namespace: test
    sectionName: http
  - name: test-gw
    namespace: test
    sectionName: https
  hostnames:
  - "test.example.com"
  rules:
  - backendRefs:
    - group: ""
      kind: Service
      name: nginx
      port: 80
```

`TCPRoute` 和 `UDPRoute` 里使用 `Gateway` 里的四层监听器（TCP/6000 和 UDP/6000）：

> ⚠ **注意：**
> 与 `HTTPRoute` 一样，使用 `sectionName` 指定具体要绑定的监听器。

```
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: foo
spec:
  parentRefs:
  - namespace: test
    name: test-gw
    sectionName: tcp-6000
```

```
  rules:
  - backendRefs:
    - name: foo
      port: 6000
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: UDPRoute
metadata:
  name: foo
spec:
  parentRefs:
  - namespace: test
    name: test-gw
    sectionName: udp-6000
  rules:
  - backendRefs:
    - name: foo
      port: 6000
```

如何实现自动重定向?

通过配置 `HTTPRoute` **的** `filters` **可实现自动重定向，下面给出示例。**
路径前缀 `/api/v1` **替换成** `/apis/v1` **：**

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: redirect-api-v1
  namespace: test
spec:
  hostnames:
  - test.example.com
  parentRefs:
  - name: test-gw
    namespace: test
    sectionName: https
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /api/v1
    filters:
    - type: RequestRedirect
      requestRedirect:
        path:
          type: ReplacePrefixMatch
          replacePrefixMatch: /apis/v1
        statusCode: 301
```

> **说明：**
> `http://test.example.com/api/v1/pods` **会被重定向到** `http://test.example.com/apis/v1/pods` **。**

以 `/foo` **开头的统一重定向到** `/bar` **：**

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: redirect-api-v1
```

```
  namespace: test
spec:
  hostnames:
  - test.example.com
  parentRefs:
  - name: test-gw
    namespace: test
    sectionName: https
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /foo
    filters:
    - type: RequestRedirect
      requestRedirect:
        path:
          type: ReplaceFullPath
          replaceFullPath: /bar
        statusCode: 301
```

说明：

`https://test.example.com/foo/cayenne` 和 `https://test.example.com/foo/paprika` 都会被重定向到 `https://test.example.com/bar` 。

HTTP 重定向到 HTTPS：

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: redirect-https
  namespace: test
spec:
  hostnames:
  - test.example.com
  parentRefs:
  - name: test-gw
    namespace: test
    sectionName: http
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    filters:
    - type: RequestRedirect
      requestRedirect:
        scheme: https
        statusCode: 301
        port: 443
```

说明：

`http://test.example.com/foo` 会被重定向到 `https://test.example.com/foo` 。

如何配置 HTTPS 或 TLS?

将证书和密钥存储在 Kubernetes 的 Secret 中：

> **说明：**
> 如果不想手动管理证书，希望证书自动签发，可考虑使用 `cert-manager` 来自动签发。参考 使用 cert-manager 为 DNSPod 的域名签发免费证书。

```
apiVersion: v1
kind: Secret
metadata:
  name: test-cert
  namespace: test
type: kubernetes.io/tls
data:
  tls.crt: xxx
  tls.key: xxx
```

在 `Gateway` 的 listeners 中配置 TLS（HTTPS 或 TLS 协议），`tls` 字段里引用证书 Secret：

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: test-gw
  namespace: test
spec:
  gatewayClassName: eg
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    allowedRoutes:
      namespaces:
        from: All
    tls:
      mode: Terminate
      certificateRefs:
      - kind: Secret
        group: ""
        name: test-cert
  - name: tls
    protocol: TLS
    port: 9443
    allowedRoutes:
      namespaces:
        from: All
    tls:
      mode: Terminate
      certificateRefs:
      - kind: Secret
        group: ""
        name: test-cert
```

如何修改 HTTP Header?

在 `HTTPRoute` 中使用 `RequestHeaderModifier` 这个 filter 可以修改 HTTP 请求的 Header。
以下是修改请求 Header 的例子：

> **说明：**
> 对路径以 `/foo` 开头的请求修改 Header。

增加Header

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  namespace: test
  name: foo-add-header
spec:
  hostnames:
  - test.example.com
  parentRefs:
  - name: test-gw
    namespace: test
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /foo
    filters:
    - type: RequestHeaderModifier
      requestHeaderModifier:
        add:
        - name: my-header-name
          value: my-header-value
    backendRefs:
    - name: foo
      port: 8080
```

修改Header

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  namespace: test
  name: foo-set-header
spec:
  hostnames:
  - test.example.com
  parentRefs:
  - name: test-gw
    namespace: test
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /foo
    filters:
    - type: RequestHeaderModifier
      requestHeaderModifier:
        set:
        - name: my-header-name
          value: my-header-value
    backendRefs:
    - name: foo
```

```
      port: 8080
```

删除Header

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  namespace: test
  name: foo-set-header
spec:
  hostnames:
  - test.example.com
  parentRefs:
  - name: test-gw
    namespace: test
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /foo
    filters:
    - type: RequestHeaderModifier
      requestHeaderModifier:
        remove: ["x-request-id"]
    backendRefs:
    - name: foo
      port: 8080
```

如果要改响应的 Header 也是类似的，`RequestHeaderModifier` 改成 `ResponseHeaderModifier` 即可：

```
  filters:
  - type: ResponseHeaderModifier
    responseHeaderModifier:
      add:
      - name: X-Header-Add-1
        value: header-add-1
      - name: X-Header-Add-2
        value: header-add-2
      - name: X-Header-Add-3
        value: header-add-3
```

探索更多用法

Gateway API 非常强大，可实现很多复杂的功能，如基于权重、header、cookie 等特征的路由、灰度发布、流量镜像、URL 重定向与重写、TLS 路由、GRPC 路由等，更详细的用法参考 Gateway API 官方文档 。
EnvoyGateway 也支持了 Gateway API 之外的一些特有的高级能力，可参考 EnvoyGateway 官方文档 。

核心应用流量完全无损升级

最近更新时间：2024-04-10 15:07:21

业务场景

对于某些核心的应用（如流量网关），我们不希望升级过程中出现任何故障。在升级这种核心应用的时候，我们采取极其保守的策略，**宁愿操作繁琐，也要确保风险完全可控**。这意味着需要先确保待升级的 Pod 完全被摘流，然后再手动重建 Pod 触发升级，持续经现网流量灰度一段时间后，如果一切正常，再逐步扩大灰度范围。如果期间发现任何问题，我们可以回滚已升级的副本。

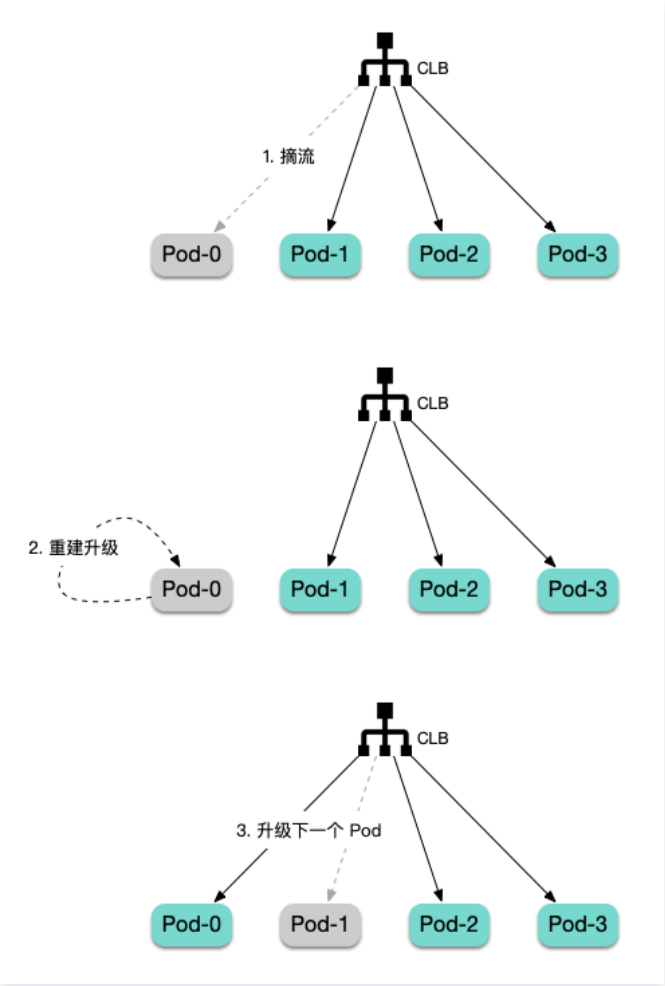
升级导致故障的可能情况

下面列举一些在升级时可能导致故障的情况：

1. 应用的优雅终止逻辑可能不够完善，导致在 Pod 停止时部分存量连接异常。
2. 存量长连接迟迟不断开，可能超过 `terminationGracePeriodSeconds`，导致 Pod 停止时，存量连接异常关闭。
3. 新版应用可能引入隐藏 BUG 或不兼容改动，这些问题可能在健康检查中无法被探测到，等上线后一段时间才被动发现。
4. CLB 摘流和 Pod 停止两个过程异步并行执行，在极端的场景下，Pod 已经开始进入优雅终止流程，不再接收增量连接，但 CLB 还没有来得及摘流（改变权重），导致个别新连接被调度到正在停止的 Pod 而无法得到处理。

流量完全无损升级的方法

在 TKE 环境中，可以使用 StatefulSet 来部署核心应用，并配合 TKE 的 Service 注解来实现对指定序号的 Pod 提前摘流，再结合 StatefulSet 的 `OnDelete` 更新策略对 Pod 进行手动重建更新来实现流量完全无损的升级。以下是大致的升级流程：



操作步骤

创建 StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  replicas: 10
  podManagementPolicy: Parallel
  selector:
    matchLabels:
      app: nginx
  serviceName: nginx
  template:
    metadata:
      annotations:
        tke.cloud.tencent.com/networks: tke-route-eni # GlobalRouter 与 VPC-CNI 混用时显式声明使用 VPC-CNI
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.25.3
          imagePullPolicy: IfNotPresent
          name: nginx
          ports:
            - containerPort: 80
              name: http
              protocol: TCP
          resources: # GlobalRouter 与 VPC-CNI 混用时显式声明使用 VPC-CNI
            limits:
              tke.cloud.tencent.com/eni-ip: "1"
            requests:
              tke.cloud.tencent.com/eni-ip: "1"
  updateStrategy:
    type: OnDelete # 重要：使用手动重建触发升级而非滚动更新

---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  annotations:
    service.cloud.tencent.com/direct-access: "true" # 重要：启用 CLB 直连 Pod
  name: nginx
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

说明：

- StatefulSet 的 `updateStrategy` 使用 `OnDelete`。
- 确保 Pod 使用 VPC-CNI 网络或调度到超级节点（方便启用 CLB 直连）。
- 加 Service 注解启用 CLB 直连。

逐个升级

1. 首先将 StatefulSet 所使用的镜像替换为新的镜像（升级后期望使用的镜像版本）。

```
kubectl set image statefulset/nginx nginx=nginx:1.25.4
```

2. 将 StatefulSet 副本数加1，因为副本重建的过程中副本数会少1个，提前增加1个副本可避免升级过程中 Pod 的平均负载过高导致业务异常：

```
kubectl scale --replicas=11 statefulset/nginx
```

3. 为 Service 添加注解：

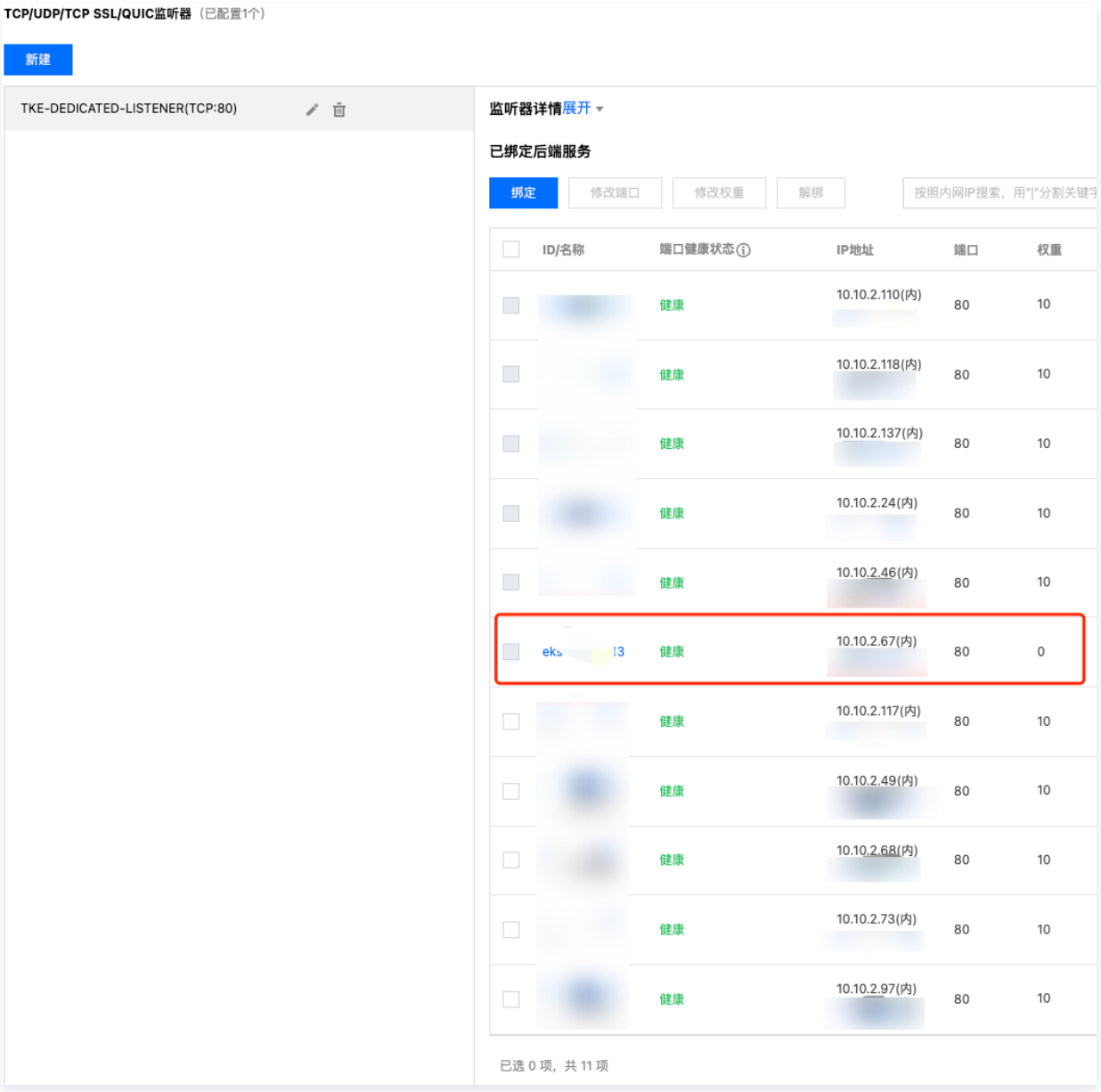
```
    service.cloud.tencent.com/lb-rs-weight: '{"defaultWeight":10,"groups":[{"key":
{"proto":"TCP","port":80},"statefulSets":[{"name":"nginx","weights":[{"weight":0,"podIndexes":
[0]}]}]}]}'
```

   - `key` 里填入 Service 里声明的端口和协议，如果有多个端口，需在 `groups` 数组里再加一份配置（只有 `key` 不同），示例如下：

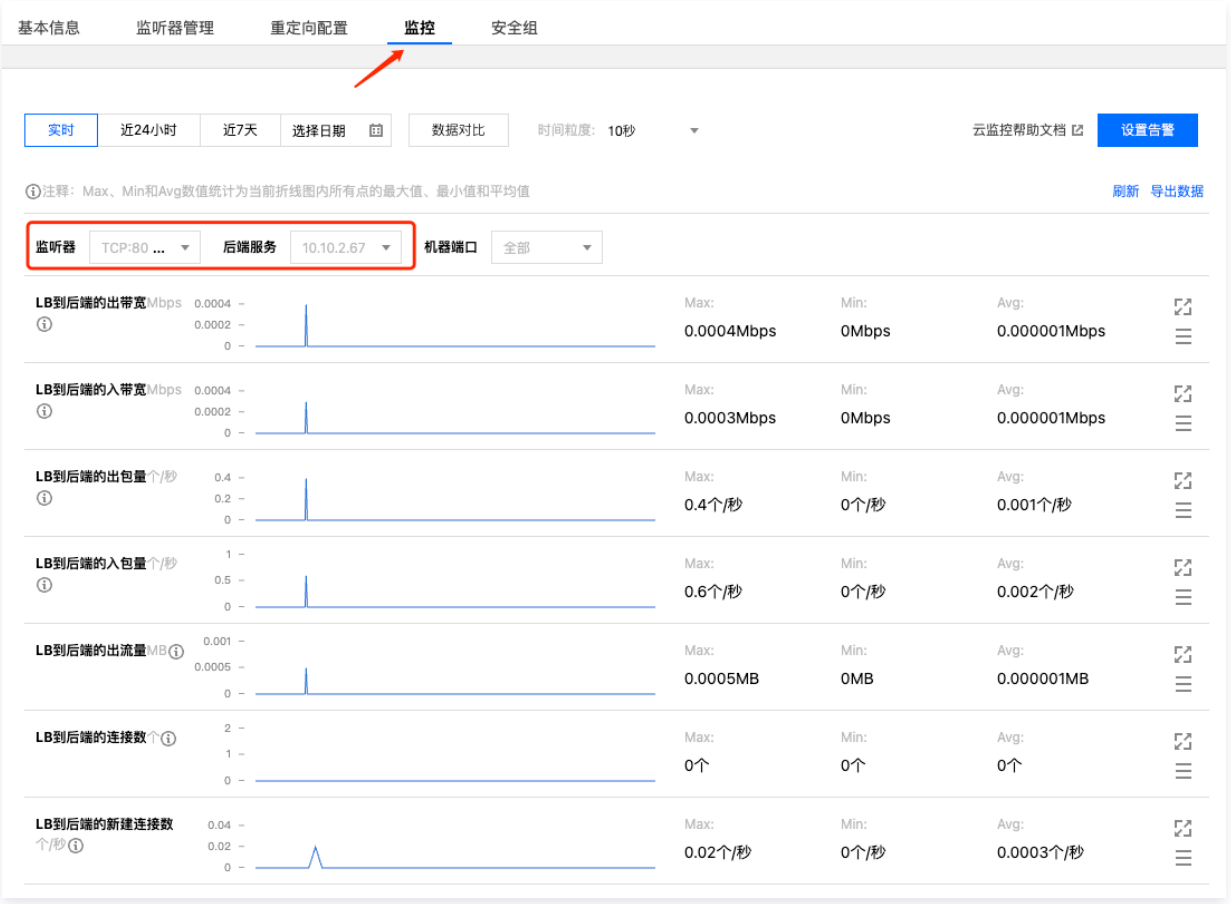
```
    service.cloud.tencent.com/lb-rs-weight: '{"defaultWeight":10,"groups":[{"key":
{"proto":"TCP","port":80},"statefulSets":[{"name":"nginx","weights":[{"weight":0,"podIndexes":
[0]}]}]},{"key":{"proto":"TCP","port":8080},"statefulSets":[{"name":"nginx","weights":
[{"weight":0,"podIndexes":[0]}]}]}]}'
```

   - `statefulSets` 里的 `name` 填入 StatefulSet 的名称。
   - `podIndexes` 里填入接下来计划升级的 Pod 序号，一般从0开始。
   - `weight` 置为0，即将该序号的 Pod 从 CLB 后端摘流（增量连接不再调度过去，等待存量连接结束）。

4. 在 CLB 控制台查看对应 CLB 的监听器绑定的后端服务，确认将要升级的 Pod IP 的流量权重为0：



5. 等待存量连接和流量完全归零，可在 CLB 的监控页面确认（输入对应的监听器和后端服务进行过滤）：



6. 删除计划升级的 Pod，触发重建升级：

```
kubectl delete pod nginx-0
```

7. 观察升级后的 Pod 运行状态、镜像和健康状态，以确保它们都符合预期（可以通过现网流量验证一段时间，如果发现异常，可以回滚镜像版本并再次按照步骤 3~5 回滚）：

```
$ kubectl get pod -o wide nginx-0
NAME      READY   STATUS    RESTARTS   AGE   IP           NODE         NOMINATED NODE   READINESS
GATES
nginx-0   1/1     Running   0          86s   10.10.2.28   10.10.11.3   <none>           1/1
$ kubectl get pod nginx-0 -o yaml | grep image:
  - image: nginx:1.25.4
    image: docker.io/library/nginx:1.25.4
```

8. 修改 Service 注解中的 `podIndexes`，准备升级下一个 Pod：

```
    service.cloud.tencent.com/lb-rs-weight: '{"defaultWeight":10,"groups":[{"key":
{"proto":"TCP","port":80},"statefulSets":[{"name":"nginx","weights":[{"weight":0,"podIndexes":
[1]}]}]}]}'
```

9. 循环 3~7 的步骤，直到倒数第二个副本升级完成。
10. 删除 `service.cloud.tencent.com/lb-rs-weight` 这个 Service 注解，恢复所有 Pod 的流量权重。
11. 恢复 StatefulSet 的副本数（缩掉最后一个冗余的副本）：

```
kubectl scale --replicas=10 statefulset/nginx
```

至此，完成升级。

分批升级

如果副本数较多，逐个升级可能会显得过于繁琐。在这种情况下，您可以选择分批升级，即从每次升级1个 Pod 变为每次升级多个 Pod，操作步骤与 逐个升级 基本一致，主要的区别在于每次操作的 Pod 数量：

- 提前扩的副本数以及每次删除重建的副本数从1个变为多个。
- `service.cloud.tencent.com/lb-rs-weight` 注解里的 `podIndexes` 从1个变为多个。例如，如果每次升级4个 Pod，您需要在注解中指定这4个 Pod：

```
    service.cloud.tencent.com/lb-rs-weight: '{"defaultWeight":10,"groups":[{"key":
{"proto":"TCP","port":80},"statefulSets":[{"name":"nginx","weights":[{"weight":0,"podIndexes":
[0,1,2,3]}]}]}]}'
```

使用 Network Policy 进行网络访问控制

最近更新时间：2023-05-17 15:41:04

Network Policy 简介

Network Policy 是 Kubernetes 提供的一种资源，用于定义基于 Pod 的网络隔离策略。描述了一组 Pod 是否可以与其他组 Pod，以及其他 network endpoints 进行通信。

使用场景

在腾讯云容器服务 TKE 中，Pod Networking 的功能是由基于 IaaS 层私有网络 VPC 的高性能容器网络实现，而 service proxy 功能是由 kube-proxy 所支持的 ipvs/iptables 两种模式提供。TKE 通过 Network Policy 扩展组件提供网络隔离能力。

在 TKE 上启用 NetworkPolicy 扩展组件

目前 TKE 集群的扩展组件市场已提供 NetworkPolicy 扩展组件，支持一键安装与部署。具体操作步骤可参见 NetworkPolicy 说明 。

NetworkPolicy 配置示例

> **说明**
> 资源对象的 apiVersion 可能因为您集群的 Kubernetes 版本不同而不同，您可通过 `kubectl api-versions` 命令查看当前资源对象的 apiVersion。

- nsa namespace 下的 Pod 可互相访问，而不能被其他任何 Pod 访问。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npa
  namespace: nsa
spec:
  ingress:
  - from:
    - podSelector: {}
  podSelector: {}
  policyTypes:
  - Ingress
```

- nsa namespace 下的 Pod 不能被任何 Pod 访问。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npa
  namespace: nsa
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

- nsa namespace 下的 Pod 只在 6379/TCP 端口可以被带有标签 app: nsb 的 namespace 下的 Pod 访问，而不能被其他任何 Pod 访问。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npa
  namespace: nsa
```

```
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          app: nsb
    ports:
    - protocol: TCP
      port: 6379
  podSelector: {}
  policyTypes:
  - Ingress
```

- nsa namespace 下的 pod 可以访问 CIDR 为14.215.0.0/16的 network endpoint 的5978/TCP 端口，而不能访问其他任何 network endpoints（此方式可以用来为集群内的服务开访问外部 network endpoints 的白名单）。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npa
  namespace: nsa
spec:
  egress:
  - to:
    - ipBlock:
        cidr: 14.215.0.0/16
    ports:
    - protocol: TCP
      port: 5978
  podSelector: {}
  policyTypes:
  - Egress
```

- default namespace 下的 Pod 只在80/TCP 端口可以被 CIDR 为14.215.0.0/16的 network endpoint 访问，而不能被其他任何 network endpoints 访问。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npd
  namespace: default
spec:
  ingress:
  - from:
    - ipBlock:
        cidr: 14.215.0.0/16
    ports:
    - protocol: TCP
      port: 80
  podSelector: {}
  policyTypes:
  - Ingress
```

NetworkPolicy 扩展组件功能测试

运行 K8S 社区针对 `NetworkPolicy` 的 e2e 测试，结果如下：

NetworkPolicy Feature	是否支

	持
should support a 'default-deny' policy	支持
should enforce policy to allow traffic from pods within server namespace based on PodSelector	支持
should enforce policy to allow traffic only from a different namespace, based on NamespaceSelector	支持
should enforce policy based on PodSelector with MatchExpressions	支持
should enforce policy based on NamespaceSelector with MatchExpressions	支持
should enforce policy based on PodSelector or NamespaceSelector	支持
should enforce policy based on PodSelector and NamespaceSelector	支持
should enforce policy to allow traffic only from a pod in a different namespace based on PodSelector and NamespaceSelector	支持
should enforce policy based on Ports	支持
should enforce multiple, stacked policies with overlapping podSelectors	支持
should support allow-all policy	支持
should allow ingress access on one named port	支持
should allow ingress access from namespace on one named port	支持
should allow egress access on one named port	不支持
should enforce updated policy	支持
should allow ingress access from updated namespace	支持
should allow ingress access from updated pod	支持
should deny ingress access to updated pod	支持
should enforce egress policy allowing traffic to a server in a different namespace based on PodSelector and NamespaceSelector	支持
should enforce multiple ingress policies with ingress allow-all policy taking precedence	支持
should enforce multiple egress policies with egress allow-all policy taking precedence	支持
should stop enforcing policies after they are deleted	支持
should allow egress access to server in CIDR block	支持
should enforce except clause while egress access to server in CIDR block	支持
should enforce policies to check ingress and egress policies can be controlled independently based on PodSelector	支持

NetworkPolicy 扩展组件功能测试（旧版）

在 k8s 集群中部署大量的 Nginx 服务，通过 ApacheBench 工具压测固定的一个服务，对比开启和不开启 kube-router 场景下的 QPS，衡量 kube-router 带来的性能损耗。

测试环境

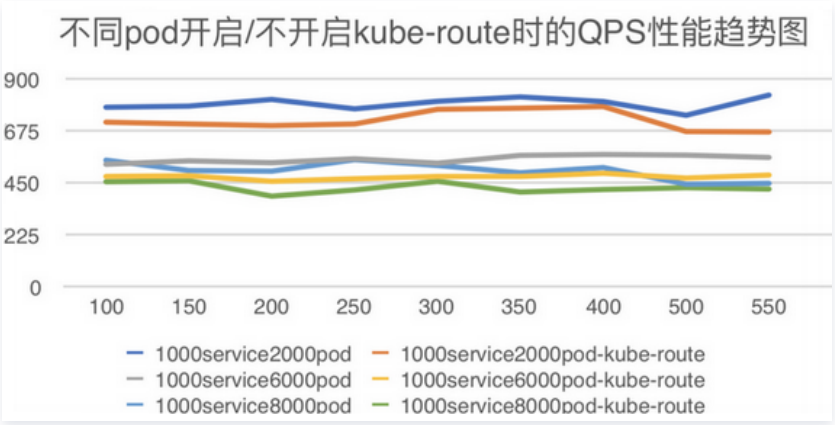
- VM 数量：100
- VM 配置：2核4G
- VM OS：Ubuntu
- k8s：1.10.5
- kube-router version：0.2.0

测试流程

1. 部署1个 service，对应两个 Pod（Nginx），作为测试组。
2. 部署1000个 service，每个分别对应 2/6/8 个 Pod（Nginx），作为干扰组。
3. 部署 NetworkPolicy 规则，使得所有 Pod 都被选中，以便产生足够数量的 iptables 规则：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npd
  namespace: default
spec:
  ingress:
  - from:
    - ipBlock:
        cidr: 14.215.0.0/16
    ports:
    - protocol: TCP
      port: 9090
  - from:
    - ipBlock:
        cidr: 14.215.0.0/16
    ports:
    - protocol: TCP
      port: 8080
  - from:
    - ipBlock:
        cidr: 14.215.0.0/16
    ports:
    - protocol: TCP
      port: 80
  podSelector: {}
  policyTypes:
  - Ingress
```

4. 使用 ab 压测测试组的服务，记录 QPS。得出性能曲线如下：



- 图例中：
  - 1000service2000pod、1000service6000pod、1000service8000pod 为 pod 未开启 kube-route
  - 1000service2000pod-kube-route、1000service6000pod-kube-route、1000service8000pod-kube-route 为 pod 已开启 kube-route
- X轴：ab 并发数
- Y轴：QPS

测试结论

Pod 数量从2000增长到8000，开启 kube-router 时的性能比不开启时要下降10% - 20%。

相关说明

腾讯云提供的 kube-router 版本

NetworkPolicy 扩展组件基于社区的 Kube-Router 项目，在该组件的开发过程中，腾讯云 PaaS 团队积极建设社区，持续贡献了一些 feature support 及 bug fix，提交 PR 均已被社区合并，列表如下：

- processing k8s version for NPC #488
- Improve health check for cache synchronization #498
- Make the comments of the iptables rules in NWPLCY chains more accurate and reasonable #527
- Use ipset to manage multiple CIDRs in a network policy rule #529
- Add support for 'except' feature of network policy rule#543
- Avoid duplicate peer pods in npc rules variables #634
- Support named port of network policy #679

Nginx 升级最佳实践

最近更新时间：2023-04-26 18:02:27

由于 Nginx Controller 各个版本支持的 Kubernetes 版本范围比较窄，所以用户会遇到 Nginx Controller 升级的相关问题。本文向您介绍通过新建实例实现过渡的方案，以帮助您顺利升级 Nginx Controller。

迁移升级示例

集群与 Nginx 实例相关信息

- 当前 Kubernetes 集群版本：1.20
- 当前 Nginx Addon 版本：1.2.0（Nginx Addon 是 Nginx Controller 的安装工具，您可以在插件列表查看该插件与版本）
- 当前 Nginx Controller 版本：v0.49.3（Nginx Controller 是实际运行的社区组件，您可以在 Nginx Controller 的工作负载中查询到镜像版本）
- 当前 Nginx Ingress Class：prod-0-49-3

> **说明：**
> 集群如果升级到 1.22，有许多旧版本的社区 CRD 定义被废弃，Nginx Controller 版本低于v1.0.0将无法正常工作。所以大部分用户在这个集群版本升级时，需要对 Nginx Controller 进行升级。

升级过程

安装新版本 Nginx Controller 实例

参考 Nginx 的安装文档，安装v1.1.3版本的 Nginx，同时建议开启日志、监控。假定文档中的新实例 Ingress Class 为 prod-1-1-3 安装新版本 Nginx Controller 实例之后。参考原实例配置，调整 Workload 规格或 HPA，避免新实例出现容量问题。

迁移 Ingress 资源

将所有旧 Nginx Controller 相关资源复制到新实例中。
这里以其中一个 Ingress 资源为例，说明迁移过程：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: prod-0-49-3
  name: access
  namespace: default
spec:
  rules:
  - host: www.exsample.com
    http:
      paths:
      - backend:
          serviceName: access-server
          servicePort: 8080
        path: /
        pathType: ImplementationSpecific
status:
  loadBalancer:
    ingress:
    - ip: 172.17.99.37
```

迁移注意事项：

1. 请保留原有 Ingress 不变，复制一个新的 Ingress 实例。请确保实例名称不能出现冲突。
2. 注意将新实例的 “kubernetes.io/ingress.class” 修改为 prod-1-1-3。

3. 请注意从 Kubernetes 1.22 版本开始，“apiVersion: extensions/v1beta1”已被删除。升级到 Nginx 版本1.1.3的同时，请将资源升级到“apiVersion: networking.k8s.io/v1”。
4. 请注意，extensions/v1beta1 和 networking.k8s.io/v1 在 backend 内的字段格式有变化，您需要做出一定的调整。
5. 在将资源添加到集群后，Nginx Controller 会自动更新 status 的内容并提供新的 IP 地址。

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: prod-1-1-3
  name: access-new
  namespace: default
spec:
  rules:
  - host: www.exsample.com
    http:
      paths:
      - backend:
          service:
            name: access-server
            port:
              number: 8080
        path: /
        pathType: ImplementationSpecific
status:
  loadBalancer:
    ingress:
    - ip: 172.17.22.11
```

测试、流量切换与回滚方案

1. 在本地通过 Host 绑定进行测试：

```
172.17.22.11 www.exsample.com
```

2. 测试通过之后，您可以更新 DNS 解析至新的 IP，以验证新的入口访问是否正常。
3. 如果在更新 DNS 解析配置后发现数据面流量异常，您可以通过回滚 DNS 解析配置的方式，将域名解析切换至旧实例，以恢复正常的流量。
4. 直到流量切换操作之前，在新旧版本并行存在的期间，旧版本的 Nginx Ingress 实例不会有任何变更。新实例的创建和测试都不会受到影响，这是一个相对安全可控的升级方案。

清理资源

在全部流量迁移至新实例之后，您可以按照以下步骤进行清理：

1. 可以先清理旧 Nginx Ingress 实例相关的 Ingress 资源。
2. 在控制台中销毁 Nginx Ingress 实例。

相关文档

安装 Nginx-ingress 实例

Nginx Ingress 高并发实践

最近更新时间：2023-11-01 11:03:21

概述

Nginx Ingress Controller 基于 Nginx 实现 Kubernetes Ingress API。Nginx 是一款高性能网关，在实际生产环境运行时，需要对参数进行调优，以保证其充分发挥高性能的优势。在 TKE 上部署 Nginx Ingress 中的部署 YAML 已经包含 Nginx 部分性能方面的参数优化。本文将介绍针对 Nginx Ingress 全局配置与内核参数调优的方法及其原理，让 Nginx Ingress 更好的适配高并发业务场景。

内核参数调优

您可通过 调高连接队列的大小 、扩大源端口范围 、TIME_WAIT 复用 或 调大最大文件句柄数 的方式对 Nginx Ingress 进行内核参数调优，并可使用 initContainers 方式设置内核参数，详情请参见 配置示例 。

调高连接队列的大小

在高并发环境下，如果连接队列过小，则可能导致队列溢出，使部分连接无法建立。进程监听 socket 的连接队列大小受限于内核参数 `net.core.somaxconn` ，调整 somaxconn 内核参数的值即可增加 Nginx Ingress 连接队列。
进程调用 listen 系统监听端口时会传入一个 backlog 参数，该参数决定 socket 连接队列大小，且其值不大于 somaxconn 取值。Go 程序标准库在 listen 时，默认直接读取 somaxconn 作为队列大小，但 Nginx 监听 socket 时并不会读取 somaxconn，而是读取 `nginx.conf` 。在 `nginx.conf` 中的 listen 端口配置项中，可以通过 backlog 参数配置连接队列大小，来决定 Nginx listen 端口的连接队列大小。配置示例如下：

```
server {
    listen  80  backlog=1024;
    ...
```

如果未配置 backlog 值，则该值默认为511。backlog 参数详细说明如下：

```
backlog=number
   sets the backlog parameter in the listen() call that limits the maximum length for the queue of
pending connections. By default, backlog is set to -1 on FreeBSD, DragonFly BSD, and macOS, and to 511 on
other platforms.
```

在默认配置下，即便 somaxconn 的值配置超过511，但 Nginx 所监听端口的连接队列最大只有511，因此在高并发环境下可能导致连接队列溢出。
而 Nginx Ingress 不同，Nginx Ingress Controller 会自动读取 somaxconn 的值作为 backlog 参数，并写到生成的 nginx.conf 中，因此 Nginx Ingress 的连接队列大小只取决于 somaxconn 的大小，该取值在 TKE 中默认为4096。在高并发环境下，建议执行以下命令，将 somaxconn 设为 65535：

```
sysctl -w net.core.somaxconn=65535
```

扩大源端口范围

高并发环境将导致 Nginx Ingress 使用大量源端口与 upstream 建立连接，源端口范围从 `net.ipv4.ip_local_port_range` 内核参数中定义的区间随机选取。在高并发环境下，端口范围小容易导致源端口耗尽，使得部分连接异常。TKE 环境创建的 Pod 源端口范围默认为32768 – 60999，建议执行以下命令扩大源端口范围，调整为1024 – 65535：

```
sysctl -w net.ipv4.ip_local_port_range="1024 65535"
```

TIME_WAIT 复用

如果短连接并发量较高，所在 netns 中 TIME_WAIT 状态的连接将同样较多，而 TIME_WAIT 连接默认要等 2MSL 时长才释放，将长时间占用源端口，当这种状态连接数量累积到超过一定量之后可能会导致无法新建连接。
建议执行以下命令，为 Nginx Ingress 开启 TIME_WAIT 复用，即允许将 TIME_WAIT 连接重新用于新的 TCP 连接：

```
sysctl -w net.ipv4.tcp_tw_reuse=1
```

调大最大文件句柄数

Nginx 作为反向代理，每个请求将与 client 和 upstream server 分别建立一个连接，即占据两个文件句柄，因此理论上 Nginx 能同时处理的连接数最多是系统最大文件句柄数限制的一半。
系统最大文件句柄数由 `fs.file-max` 内核参数控制，TKE 默认值为838860。建议执行以下命令，将最大文件句柄数设置为1048576：

```
sysctl -w fs.file-max=1048576
```

配置示例

给 Nginx Ingress Controller 的 Pod 添加 initContainers 并设置内核参数。可参考以下代码示例：

```
initContainers:
- name: setsysctl
  image: busybox
  securityContext:
    privileged: true
  command:
  - sh
  - -c
  - |
      sysctl -w net.core.somaxconn=65535
      sysctl -w net.ipv4.ip_local_port_range="1024 65535"
      sysctl -w net.ipv4.tcp_tw_reuse=1
      sysctl -w fs.file-max=1048576
```

全局配置调优

除了内核参数需要调优，您可以通过以下方式对 Nginx 全局配置进行调优：

- 调高 keepalive 连接最大请求数
- 调高 keepalive 最大空闲连接数
- 调高单个 worker 最大连接数

调高 keepalive 连接最大请求数

Nginx 针对 client 和 upstream 的 keepalive 连接，具备 keepalive_requests 参数来控制单个 keepalive 连接的最大请求数，默认值均为100。当一个 keepalive 连接中请求次数超过默认值时，将断开并重新建立连接。
如果是内网 Ingress，单个 client 的 QPS 可能较大，例如达到10000QPS，Nginx 将可能频繁断开跟 client 建立的 keepalive 连接，并产生大量 TIME_WAIT 状态连接。为避免产生大量的 TIME_WAIT 连接，建议您在高并发环境中增大 Nginx 与 client 的 keepalive 连接的最大请求数量，在 Nginx Ingress 的配置对应 `keep-alive-requests`，可以设置为10000，详情请参见 keep-alive-requests 。
同样，Nginx 针对 upstream 的 keepalive 连接的请求数量的配置是 `upstream-keepalive-requests`，配置方法请参见 upstream-keepalive-requests 。

> **注意**
> 在非高并发环境，不必配此参数。如果将其调高，可能导致负载不均，因 Nginx 与 upstream 保持的 keepalive 连接过久，导致连接发生调度的次数减少，连接过于“固化”，将使流量负载不均衡。

调高 keepalive 最大空闲连接数

Nginx 针对 upstream 可配置参数 keepalive。该参数为最大空闲连接数，默认值为320。在高并发环境下将产生大量请求和连接，而实际生产环境中请求并不是完全均匀，有些建立的连接可能会短暂空闲，在空闲连接数多了之后关闭空闲连接，将可能导致 Nginx 与 upstream 频繁断连和建连，引发 TIME_WAIT 飙升。在高并发环境下，建议将 keepalive 值配置为1000，详情请参见 upstream-keepalive-connections 。

调高单个 worker 最大连接数

`max-worker-connections` 控制每个 worker 进程可以打开的最大连接数，TKE 环境默认为16384。在高并发环境下建议调高该参数值，例如配置为65536，调高该值可以让 Nginx 拥有处理更多连接的能力，详情请参见 max-worker-connections 。

配置示例

Nginx 全局配置通过 configmap 配置（Nginx Ingress Controller 会读取并自动加载该配置）。可参考以下代码示例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-ingress-controller
# nginx ingress 性能优化: https://www.nginx.com/blog/tuning-nginx/
data:
  # nginx 与 client 保持的一个长连接能处理的请求数量，默认100，高并发场景建议调高。
  # 参考: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#keep-alive-requests
  keep-alive-requests: "10000"
  # nginx 与 upstream 保持长连接的最大空闲连接数 (不是最大连接数)，默认 320，在高并发下场景下调大，避免频繁建联导致 TIME_WAIT 飙升。
  # 参考: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#upstream-keepalive-connections
  upstream-keepalive-connections: "2000"
  # 每个 worker 进程可以打开的最大连接数，默认 16384。
  # 参考: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/#max-worker-connections
  max-worker-connections: "65536"
```

相关文档

- 在 TKE 上部署 Nginx Ingress
- Nginx Ingress 配置参考
- Tuning NGINX for Performance
- ngx_http_upstream_module 官方文档

Nginx Ingress 最佳实践

最近更新时间：2024-08-28 09:50:11

操作场景

容器服务 TKE 支持安装 Nginx-ingress 扩展组件，可通过 Nginx-ingress 接入 Ingress 流量。关于 Nginx-ingress 组件的更多介绍，请参见 Nginx-ingress 说明 。本文将为您介绍 Nginx-ingress 组件常见最佳实践操作指引。

前提条件

已安装 Nginx-ingress 扩展组件。

操作步骤

为集群暴露多个 Nginx Ingress 流量入口

Nginx-ingress 扩展组件安装后，在 `kube-system` 下会有 Nginx-ingress 的 operator 组件，通过该组件可以创建多个 Nginx Ingress 实例，每个 Nginx Ingress 实例都使用不同的 IngressClass，且使用不同的 CLB 作为流量入口，从而实现不同的 Ingress 绑定到不同流量入口。可以根据实际需求，为集群创建多个 Nginx Ingress 实例。

1. 登录 容器服务控制台 ，选择左侧导航栏中的**集群**。
2. 在集群管理页面单击目标集群 ID，进入集群详情页面。
3. 选择左侧菜单栏中的**服务与路由** > **NginxIngress**。
4. 单击**新增 Nginx Ingress 实例**，根据需求配置 Nginx Ingress 实例，为每个实例指定不同的 IngressClass 名称。

> **说明：**
> 创建 Nginx Ingress 实例详细步骤，请参见 安装 Nginx-ingress 实例 。

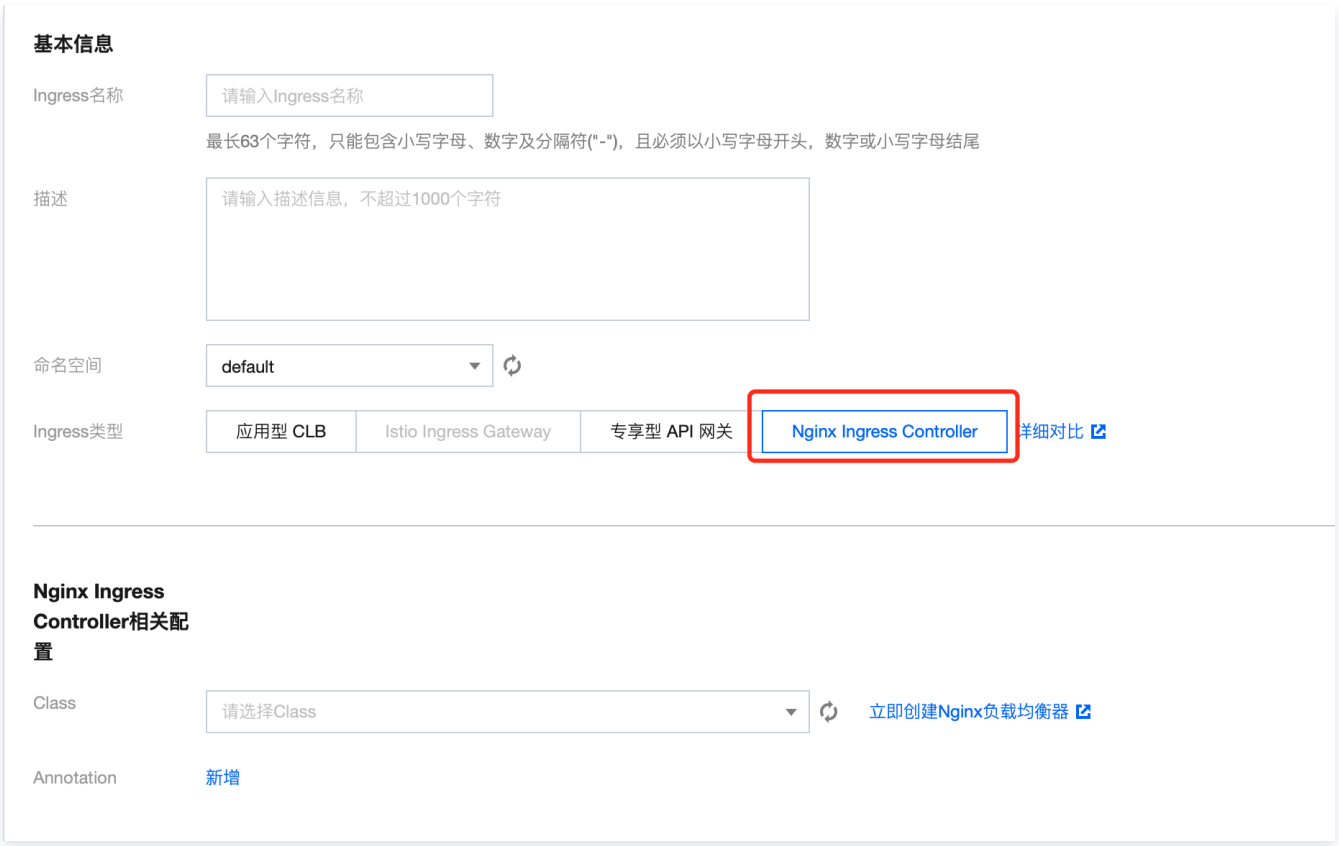
5. 创建 Ingress 时可指定具体的 IngressClass 将 Ingress 绑定到具体的 Nginx Ingress 实例上。您可通过控制台或 YAML 创建 Ingress：

通过控制台创建 Ingress

参考控制台 创建 Ingress 步骤创建 Ingress。其中：

- **Ingress类型**：选择 **Nginx Ingress Controller**。

- **Class**：选择上述步骤创建的 Nginx Ingress 实例。



通过 YAML 创建 Ingress

参考 YAML 创建 Ingress 步骤创建 Ingress，并指定 ingressClass 的 annotation（`kubernetes.io/ingress.class`）。如下图所示：

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    ingress.cloud.tencent.com/direct-access: "false"
    kubernetes.io/ingress.class: nginx-external
```

性能优化

LB 直通 Pod

集群网络模式为 Global Router 时，默认未开启 LB 直通 Pod，建议您按照以下步骤开启 LB 直通 Pod：

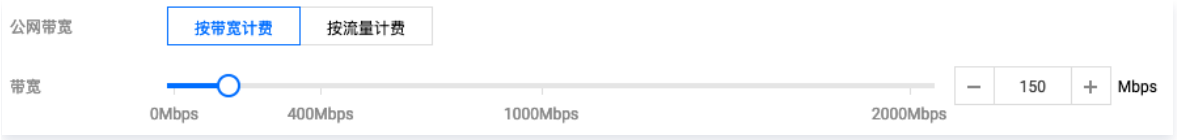
1. 为集群启用 VPC-CNI。

2. 创建 Nginx Ingress 实例时，勾选**使用 CLB 直连 Pod 模式**，可以使流量绕过 NodePort 直达 Pod，以此来提升性能。如下图所示：

> **说明：**
> 创建 Nginx Ingress 实例详细步骤，请参见 安装 Nginx-ingress 实例 。

提升 LB 带宽上限

LB 作为流量入口，如需较高的并发或吞吐，在创建 Nginx Ingress 实例时，可根据实际需求规划带宽上限，为 Nginx Ingress 分配更高的带宽。如下图所示：



若账号为非带宽上移类型（可参见 区分账户类型 文档进行区分），带宽上限取决于节点带宽，可根据以下情况调整节点的带宽上限：

- 若启用 LB 直通 Pod，LB 总带宽为 Nginx Ingress 实例 Pod 所在节点的带宽之和，建议专门规划一些高外网带宽节点部署 Nginx Ingress 实例（指定节点池 DaemonSet 部署）。
- 若未使用 LB 直通 Pod，LB 总带宽为所有节点的外网带宽之和。

优化 Nginx Ingress 参数

Nginx Ingress 实例已默认为内核参数与 Nginx Ingress 自身的配置进行优化，详情请参见 Nginx Ingress 高并发实践 。如需自定义，可参考下文介绍自行修改：

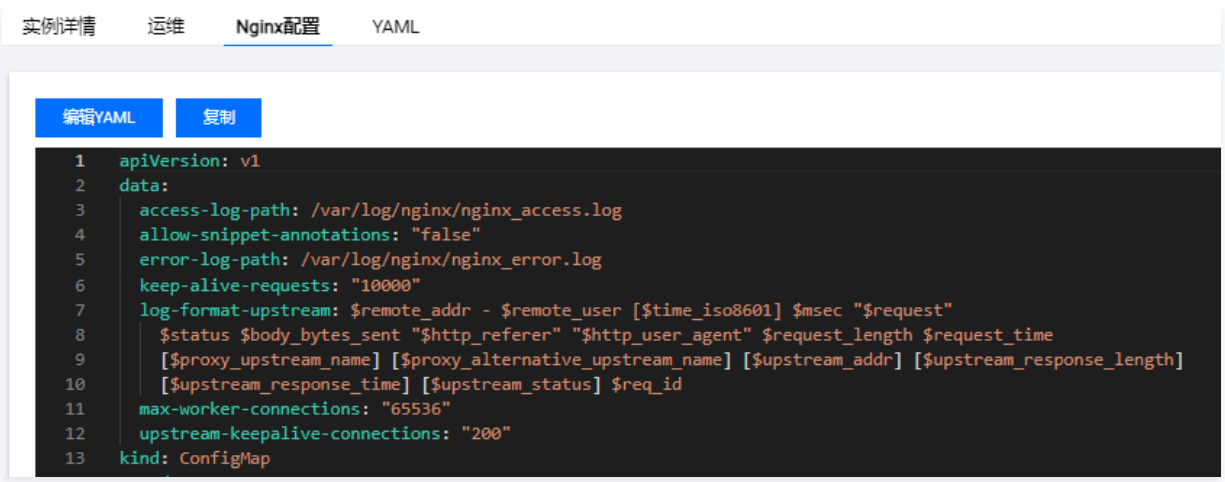
修改内核参数

编辑部署好的 nginx-ingress-controller 的 Daemonset 或 Deployment（取决于实例部署选项），修改 initContainers（使用 Kubectl 进行修改，控制台禁止修改 kube-system 下的资源）。如下图所示：

```
initContainers:
- command:
  - sh
  - -c
  - |-
    sysctl -w net.core.somaxconn=65535
    sysctl -w net.ipv4.ip_local_port_range="1024 65535"
    sysctl -w net.ipv4.tcp_tw_reuse=1
    sysctl -w fs.file-max=1048576
```

修改 Nginx Ingress 自身配置

在 **Nginx 配置**中选中对应的实例，单击**编辑 YAML** 可修改 Nginx Ingress 实例的 ConfigMap 配置。如下图所示：



说明：
ConfigMap 配置详细介绍请参见 官方文档 。

提升 Nginx Ingress 可观测性

开启日志

说明：
日志依赖 日志服务，如需开启请参见 Nginx-ingress 日志配置 。

创建 Nginx Ingress 实例后，在实例详情的**运维**功能入口里可以为实例开启**日志**，方便查看实例各项状态指标与问题排查。如下图所示：

注意：
v0.49.3 版本的实例，日志采集的索引配置文件存在名为 LogConfig 的 CRD 资源对象里，若您修改了该资源对象后，关闭/再打开该日志采集功能，该 LogConfig 的资源对象配置将被重置，请及时备份该资源对象里的数据。Nginx Ingress 实例本身的删除和 Nginx Ingress 组件的升级对该索引配置文件没有影响。
若有自定义日志的需求，请参见 文档 进行配置。

日志检索与日志仪表盘

开启日志配置后，在 Nginx Ingress 列表页可单击实例右侧**操作**项下的**更多**，在弹出的菜单中选择对应功能进行日志检索或查看日志仪表盘。

- 单击**查看 Nginx 访问日志**跳转到日志服务，在**检索分析**中选中实例对应的日志集与主题，即可查看 Nginx Ingress 的访问与错误日志。
- 单击**查看 Nginx 访问大盘**可以直接跳转到根据 Nginx Ingress 日志数据来展示统计信息的仪表盘。

在 TKE 上对 Pod 进行带宽限速

最近更新时间：2024-02-18 11:16:21

操作场景

腾讯云容器服务 TKE 暂不支持 Pod 限速，但可通过修改 CNI 插件来支持此功能。本文档介绍如何在 TKE 上实现对 Pod 的带宽限速，您可结合实际场景进行操作。

注意事项

- 腾讯云容器服务 TKE 支持使用社区的 bandwidth 插件对网络进行限速，目前适用于 Global Router 模式和 VPC-CNI 共享网卡模式。
- 暂不支持 VPC-CNI 独占网卡模式。

操作步骤

修改 CNI 插件

Global Router 模式

Global Router 网络模式是容器服务 TKE 基于底层私有网络 VPC 的全局路由能力，实现了容器网络和 VPC 互访的路由策略。GlobalRouter 网络模式适用于常规场景，可与标准 Kuberentes 功能无缝集成，更多信息请参见 Global Router模式介绍 。

1. 请参考 使用标准登录方式登录 Linux 实例（推荐），登录 Pod 所在节点。
2. 执行以下命令，查看 `tke-bridge-agent` 配置。

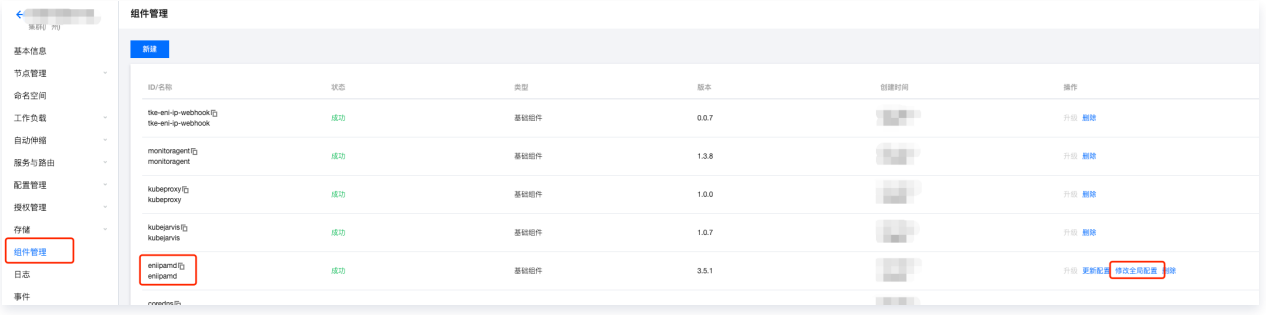
```
kubectl edit daemonset tke-bridge-agent -n kube-system
```

之后添加 args `--bandwidth`，开启 bandwidth 插件支持。

VPC-CNI 共享网卡模式

VPC-CNI 模式是容器服务 TKE 基于 CNI 和 VPC 弹性网卡实现的容器网络能力，适用于对时延有较高要求的场景。开源组件 Bandwidth 能够支持 Pod 出口和入口流量整形，以及支持带宽控制，使用详细介绍请参阅 VPC-CNI 模式介绍 。

1. 登录 容器服务控制台，单击左侧导航栏中**集群**。
2. 在**集群管理**页面，选择需开启安全组的集群 ID，进入集群详情页。
3. 在集群详情页面，选择左侧**组件管理**，在组件管理页面中，单击 **eniipamd** 组件右侧的**修改全局配置**。



4. 全局配置中找到 bandwidth 插件的配置项（路径：`agent.cniChaining.bandwidth`），修改为 `true`。

> **说明：**
> eniipamd 组件修改以上参数即可开启/关闭该特性。支持部署、变更开启和变更关闭，只影响增量的 Pod。

Pod 指定 annotation

可使用社区提供的方式设置：

- 通过 `kubernetes.io/ingress-bandwidth` 此 annotation 指定入带宽限速。
- 通过 `kubernetes.io/egress-bandwidth` 此 annotation 指定出带宽限速。

示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        kubernetes.io/ingress-bandwidth: 10M
        kubernetes.io/egress-bandwidth: 20M
    spec:
      containers:
      - name: nginx
        image: nginx
```

验证配置

您可通过以下两种方式验证配置是否成功：

- **方式1：** 登录 Pod 所在的节点，执行以下命令确认限制已经添加。

```
tc qdisc show
```

返回类似如下结果，则限制已添加成功。

```
 qdisc tbf 1: dev vethc09123a1 root refcnt 2 rate 10Mbit burst 256Mb lat 25.0ms
 qdisc ingress ffff: dev vethc09123a1 parent ffff:fff1 ----------------
 qdisc tbf 1: dev 6116 root refcnt 2 rate 20Mbit burst 256Mb lat 25.0ms
```

- **方式2：** 执行以下命令，使用 iperf 测试。

```
iperf -c <服务 IP> -p <服务端口> -i 1
```

返回类似如下结果，则说明限制已添加成功。

```
------------------------------------------------------------
Client connecting to 172.16.0.xxx, TCP port 80
TCP window size: 12.0 MByte (default)
------------------------------------------------------------
[  3] local 172.16.0.xxx port 41112 connected with 172.16.0.xx port 80
[ ID] Interval       Transfer     Bandwidth
```

```
[  3]  0.0- 1.0 sec   257 MBytes  2.16 Gbits/sec
[  3]  1.0- 2.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  2.0- 3.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  3.0- 4.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  4.0- 5.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  5.0- 6.0 sec  1.12 MBytes  9.38 Mbits/sec
[  3]  6.0- 7.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  7.0- 8.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  8.0- 9.0 sec  1.18 MBytes  9.90 Mbits/sec
[  3]  9.0-10.0 sec  1.12 MBytes  9.38 Mbits/sec
[  3]  0.0-10.3 sec   268 MBytes   218 Mbits/sec
```

TKE 基于弹性网卡直连 Pod 的网络负载均衡

最近更新时间：2023-11-20 17:21:22

概述

Kubernetes 在集群接入层设计并提供了 Service 及 Ingress 两种原生资源，分别负责四层和七层的网络接入层配置。传统方案是创建 Ingress 或 LoadBalancer 类型的 Service 来绑定腾讯云负载均衡，将服务对外暴露。此方式将用户流量负载至用户节点的 NodePort 上，再通过 KubeProxy 组件转发到容器网络中，此方案在业务性能和能力方面的支持会有所局限。
为解决此问题，腾讯云容器 TKE 团队为**使用独立或托管集群的用户提供了一种新的网络模式：TKE 基于弹性网卡直连 Pod 的网络负载均衡**。此模式增强了性能和业务能力的支持，您可通过本文了解两种模式的区别，及如何开始使用直连模式。

方案对比

对比项	直连方案	NodePort 转发	Local 转发
性能	无损失	NAT 转发+节点间转发	少量损失
Pod 更新	接入层后端主动同步更新，更新稳定	接入层后端 NodePort 保持不变	更新不同步可能导致服务中断
集群依赖	集群版本及 VPC-CNI 网络要求	-	-
业务能力限制	最佳	无法获取来源 IP，无法进行会话保持	有条件的会话保持

传统模式问题分析

性能与特性

KubeProxy 在集群中会将用户 NodePort 的流量通过 NAT 的方式转发到集群网络中。存在以下问题：

- NAT 转发导致请求在性能上有一定的损失。
- 进行 NAT 操作本身会带来性能上的损失。
- NAT 转发的目的地址可能会使得流量在容器网络内跨节点转发。
- NAT 转发导致请求的来源 IP 被修改，客户端无法获取来源 IP。
- 当负载均衡的流量集中到几个 NodePort 时，过于集中的流量会导致 NodePort 的 SNAT 转发过多，使得源端口耗尽流量异常。还可能导致 conntrack 插入冲突导致丢包，影响性能。
- KubeProxy 的转发具有随机性，无法支持会话保持。
- KubeProxy 的每个 NodePort 具有独立的负载均衡作用，由于负载均衡无法收敛至一处，难以达到全局的负载均衡。

针对以上问题，以前提供给用户的技术建议为：通过 Local 转发的方式，避免 KubeProxy NAT 转发带来的问题。但由于转发的随机性，一个节点上部署多个副本时会话保持依旧无法支持，此外，在进行滚动更新时，Local 转发容易导致服务的闪断，对业务的滚动更新策略以及停机提出了更高的要求。

业务可用性

通过 NodePort 接入服务时，NodePort 的设计存在极大的容错性。负载均衡会绑定集群所有节点的 NodePort 作为后端，集群任意一个节点的访问服务时，流量将随机分配到集群的工作负载中。则表明 NodePort 或 Pod 的不可用均不会影响服务的流量接入。
和 Local 访问相同，在直接将负载均衡后端连接至用户 Pod 的情况下，当业务在滚动更新时，如果负载均衡不能够及时绑定至新的 Pod，业务的快速滚动可能导致业务入口的负载均衡后端数量严重不足甚至被清空。因此在业务滚动更新时，接入层的负载均衡的状态良好，即保证滚动更新的安全平稳。

负载均衡的控制面性能

负载均衡的控制面接口，包括创建、删除、修改四层及七层监听器、创建及删除七层规则、绑定各个监听器或者规则的后端。这些接口大部分为异步接口，需要轮询请求结果，接口的调用时间相对较长。当用户集群规模较大时，大量的接入层资源同步会导致组件存在很大时延上的压力。

新旧模式对比

性能对比

TKE 已上线 Pod 直连模式，此模式是对负载均衡的控制面优化。针对整个同步流程，重点优化了批量调用和后端实例查询两个远程调用较频繁的地方。**优化完成后，Ingress 典型场景下的控制面性能较优化前版本有了95% - 97%左右的性能提升。**目前同步的耗时主要集中在异步接口的等待上。

后端节点突增数据

应对集群扩容的场景，数据如下：

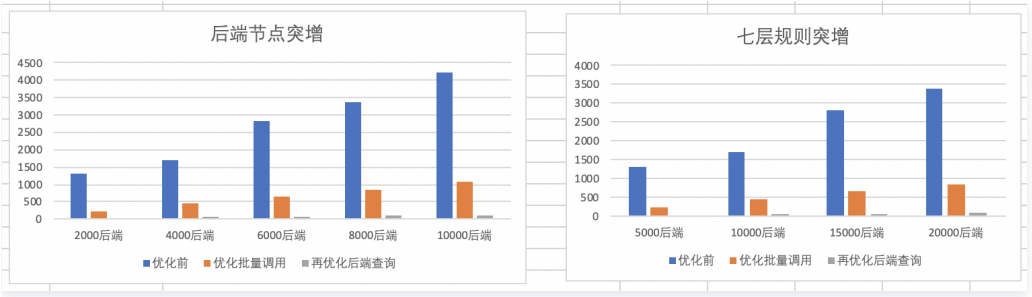
七层规则数量	集群节点数量	集群节点数量（更新）	优化前（秒）	优化批量调用（秒）	再优化后端实例查询（秒）	耗时减少（百分比）
200	1	10	1313.056	227.908	31.548	97.597%
200	1	20	1715.053	449.795	51.248	97.011%
200	1	30	2826.913	665.619	69.118	97.555%
200	1	40	3373.148	861.583	90.723	97.310%
200	1	50	4240.311	1085.03	106.353	97.491%

七层规则突增数据

应对业务第一次上线部署到集群的场景，数据如下：

七层规则数量	七层规则数量（更新）	集群节点数量	优化前（秒）	优化批量调用（秒）	再优化后端实例查询（秒）	耗时减少（百分比）
1	100	50	1631.787	451.644	68.63	95.79%
1	200	50	3399.833	693.207	141.004	95.85%
1	300	50	5630.398	847.796	236.91	95.79%
1	400	50	7562.615	1028.75	335.674	95.56%

对比图如下：

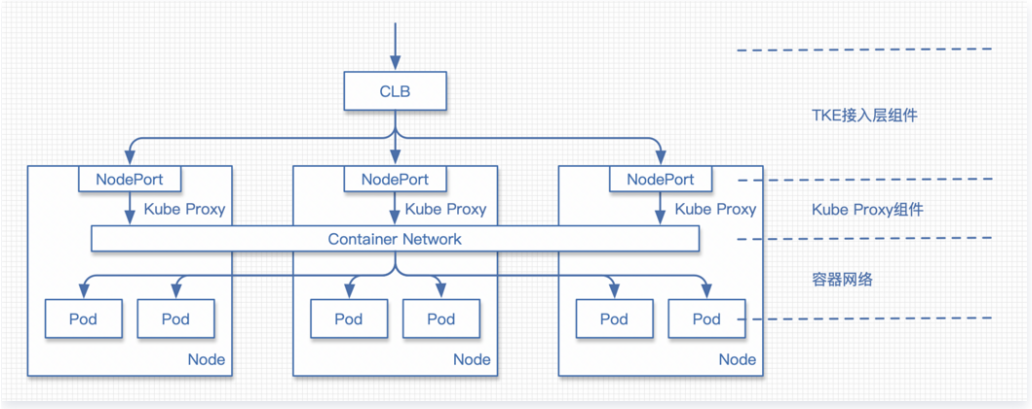


除控制面性能优化外，负载均衡能够直接访问容器网络的 Pod 即为组件业务能力最重要的组成部分，不仅避免了 NAT 转发性能上的损失，同时避免了 NAT 转发带来的各种对集群内业务功能影响，但在启动该项目时还不具备最优访问容器网络的支持。新模式结合集群 CN I网络模式下 Pod 有弹性网卡入口这一特性，实现直接接入到负载均衡以达到直接访问的目的。负载均衡直接后端访问到容器网络，目前已经有通过云联网解决的方案。

在能够直接访问后，还需保证滚动更新时的可用性。我们采用官方提供的特性 ReadinessGate，此特性于1.12版本正式发布，主要用于控制 Pod 的状态。默认情况下，Pod 有 PodScheduled、Initialized 及 ContainersReady 三种 Condition，当状态均为 Ready 时，Pod Ready 即通过了 Condition。但在云原生场景下，Pod 的状态需结合其他因素判断。而 ReadinessGate 提供允许为 Pod 状态判断增加栅栏，由第三方来进行判断与控制，Pod 的状态即可与第三方关联。

负载均衡流量对比

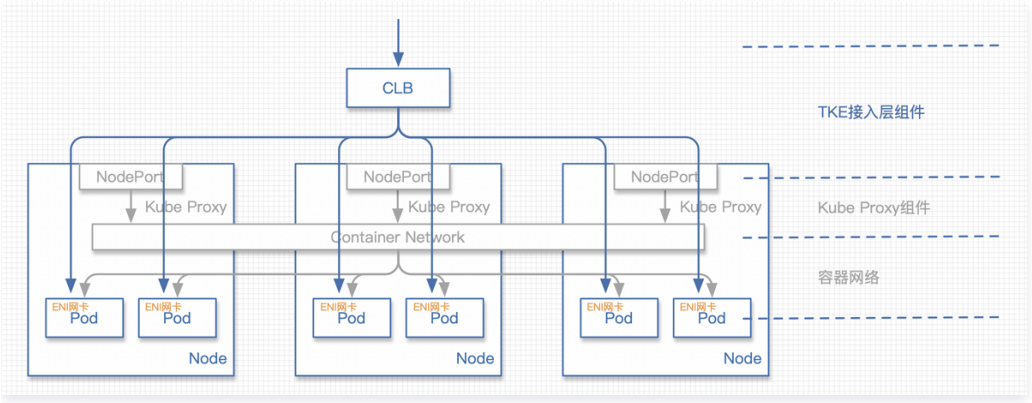
传统 NodePort 模式



请求过程：

1. 请求流量进入负载均衡。
2. 请求被负载均衡转发到某一个节点的 NodePort。
3. KubeProxy 将来自 NodePort 的流量进行 NAT 转发，目的地址是随机的一个 Pod。
4. 请求进入容器网络，并根据 Pod 地址转发到对应节点。
5. 请求来到 Pod 所属节点，转发到 Pod。

Pod 新直连模式



请求过程：

1. 请求流量进入负载均衡。
2. 请求被负载均衡转发到某一个 Pod 的 ENI 弹性网卡。

直连与 Local 访问的区别

- 从性能上区别不大，开启 Local 访问时，流量不会进行 NAT 操作也不会进行跨节点转发，仅多了一个到容器网络的路由。
- 没有进行 NAT 操作，即可正确获取来源 IP。会话保持功能可能会有问题：当一个节点上存在多个 Pod 时，流量随机到达 Pod，此机制可能会使会话保持出现问题。

引入 ReadinessGate

滚动更新相关问题

如需引入 ReadinessGate，集群版本需高于1.12。当用户开始为应用做滚动更新时，Kubernetes 会根据更新策略进行滚动更新。但其判断一批 Pod 启动的标识仅包括 Pod 自身的状态，并不会考虑该 Pod 在负载均衡上是否配置健康检查且通过。如在接入层组件高负载时，不能及时对此类 Pod 进行及时调度，则滚动更新成功的 Pod 可能并没有正在对外提供服务，从而导致服务的中断。为了关联滚动更新和负载均衡的后端状态，TKE 接入层组件引入了 Kubernetes 1.12中引入的新特性 ReadinessGate。TKE 接入层组件仅在确认后端绑定成功并且健康检查通过时，通过配置 ReadinessGate 的状态来使 Pod 达到 Ready 的状态，从而推动整个工作负载的滚动更新。

在集群中使用 ReadinessGate

Kubernetes 集群提供了服务注册的机制，只需要将您的服务以 `MutatingWebhookConfigurations` 资源的形式注册至集群即可。集群会在 Pod 创建的时候按照配置的回调路径进行通知，此时可对 Pod 进行创建前的操作，即给 Pod 加上 ReadinessGate。

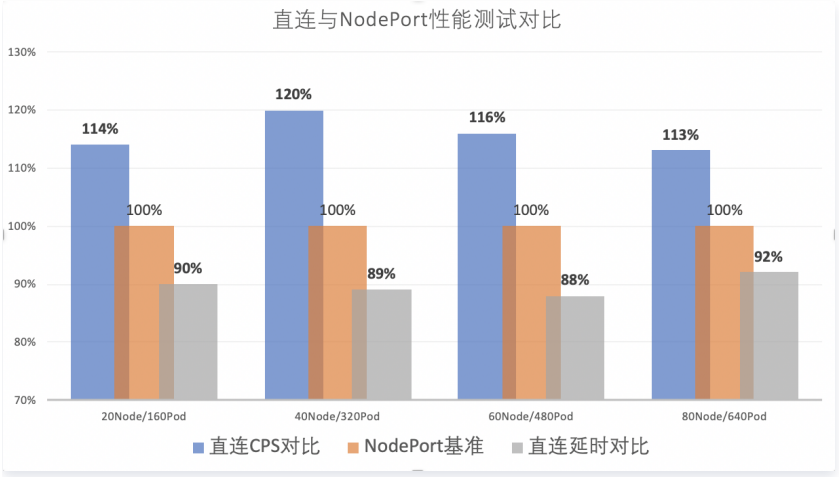
> **注意**
> 此回调过程必须是 HTTPS 的，即需要在 `MutatingWebhookConfigurations` 中配置签发请求的 CA，并在服务端配置该 CA 签发的证书。

ReadinessGate 机制的灾难恢复

用户集群中的服务注册或证书有可能被用户删除，虽然这些系统组件资源不应该被用户修改或破坏。但在用户对集群的探索或是误操作下，这类问题会不可避免的出现。接入层组件在启动时会检查以上资源的完整性，在完整性受到破坏时会重建以上资源，加强系统的鲁棒性。

QPS 和网络时延对比

直连与 NodePort 是服务应用的接入层方案，其实最终参与工作的仍为用户部署的工作负载，用户工作负载的能力直接决定了业务的 QPS 等指标。我们针对这两种接入层方案，在工作负载压力较低的情况下，重点对网络链路的时延进行了一些对比测试。直连在接入层的网络链路上能够优化10%左右的时间，且减少了大量 VPC 网络内的流量。测试场景从20节点到80节点，逐步增大集群规模，通过 wrk 工具对集群进行网络延时的测试。针对 QPS 和网络时延，直连场景与 NodePort 的对比测试如下图所示：



KubeProxy 设计思路

KubeProxy 具备一定的缺点，但基于云上负载均衡、VPC 网络的各种特性，我们具有更加本地化的接入层方案。KubeProxy 对集群接入层的设计极具普适性及容错性，基本适用于所有业务场景下的集群，作为一个官方提供的组件此设计是非常合适的。

新模式使用指引

前置要求

1. Kubernetes 集群版本需高于 1.12。
2. 集群网络模式需开启 VPC-CNI 弹性网卡模式。
3. 直连模式 Service 使用的工作负载需为 VPC-CNI 弹性网卡模式。

控制台操作指引

1. 登录 容器服务控制台 。
2. 参考控制台 创建 Service 步骤，在**新建 Service** 页面，根据实际需求设置 Service 参数。主要参数信息需进行如下设置：
   - **服务访问方式**：选择为**提供公网 LB 访问**或**内网 LB 访问访问**。
   - **网络模式**：勾选**采用负载均衡直连 Pod 模式**。
   - **Workload 绑定**：选择**引用 Worklocad**，并在弹出窗口中选择 VPC-CNI 模式的后端工作负载。
3. 单击**创建 Service** 即可完成创建。

Kubectl 操作指引

Workload 示例：nginx-deployment-eni.yaml

> **注意**

> 注意 `spec.template.metadata.annotations` 中声明了 `tke.cloud.tencent.com/networks: tke-route-eni`，即表示在工作负载使用 VPC-CNI 弹性网卡模式。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx-deployment-eni
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      annotations:
        tke.cloud.tencent.com/networks: tke-route-eni
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        name: nginx
        ports:
        - containerPort: 80
          protocol: TCP
```

Service 示例：nginx-service-eni.yaml

> ⚠ 注意
> `metadata.annotations` 中声明了 `service.cloud.tencent.com/direct-access: "true"`，Service 在同步负载均衡时将采用直连的方式配置访问后端。

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.cloud.tencent.com/direct-access: "true"
  labels:
    app: nginx
  name: nginx-service-eni
spec:
  externalTrafficPolicy: Cluster
  ports:
  - name: 80-80-no
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  sessionAffinity: None
  type: LoadBalancer
```

部署集群

> ⚠ 注意

> 在环境中您首先需要连接到集群（没有集群的需要先创建集群），可以参考 帮助文档 配置 kubectl 连接集群。

```
➜  ~ kubectl apply -f nginx-deployment-eni.yaml
deployment.apps/nginx-deployment-eni created

➜  ~ kubectl apply -f nginx-service-eni.yaml
service/nginx-service-eni configured

➜  ~ kubectl get pod -o wide
NAME                                    READY   STATUS    RESTARTS   AGE   IP               NODE
NOMINATED NODE   READINESS GATES
nginx-deployment-eni-bb7544db8-6ljkm    1/1     Running   0          24s   172.17.160.191   172.17.0.3
<none>           1/1
nginx-deployment-eni-bb7544db8-xqqtv    1/1     Running   0          24s   172.17.160.190   172.17.0.46
<none>           1/1
nginx-deployment-eni-bb7544db8-zk2cx    1/1     Running   0          24s   172.17.160.189   172.17.0.9
<none>           1/1
➜  ~ kubectl get service -o wide
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)         AGE    SELECTOR
kubernetes          ClusterIP      10.187.252.1     <none>           443/TCP         6d4h   <none>
nginx-service-eni   LoadBalancer   10.187.254.62    150.158.221.31   80:32693/TCP    6d1h   app=nginx
```

总结

目前 TKE 利用弹性网卡实现了 Pod 直连的网络模式，我们还将对这个特性进行更多优化，包括但不限于：

- 不依赖 VPC-ENI 的网络模式，实现普通容器网络下的 Pod 直连。
- 支持在 Pod 删除之前，摘除负载均衡后端。

与业界对比：

- AWS 有类似方案，通过弹性网卡的方式实现了 Pod 直连。
- Google Kubernetes Engine，GKE 也有类似方案，结合 Google Cloud Load Balancing，CLB 的 Network Endpoint Groups，NEG 特性实现接入层直连 Pod。

参考资料

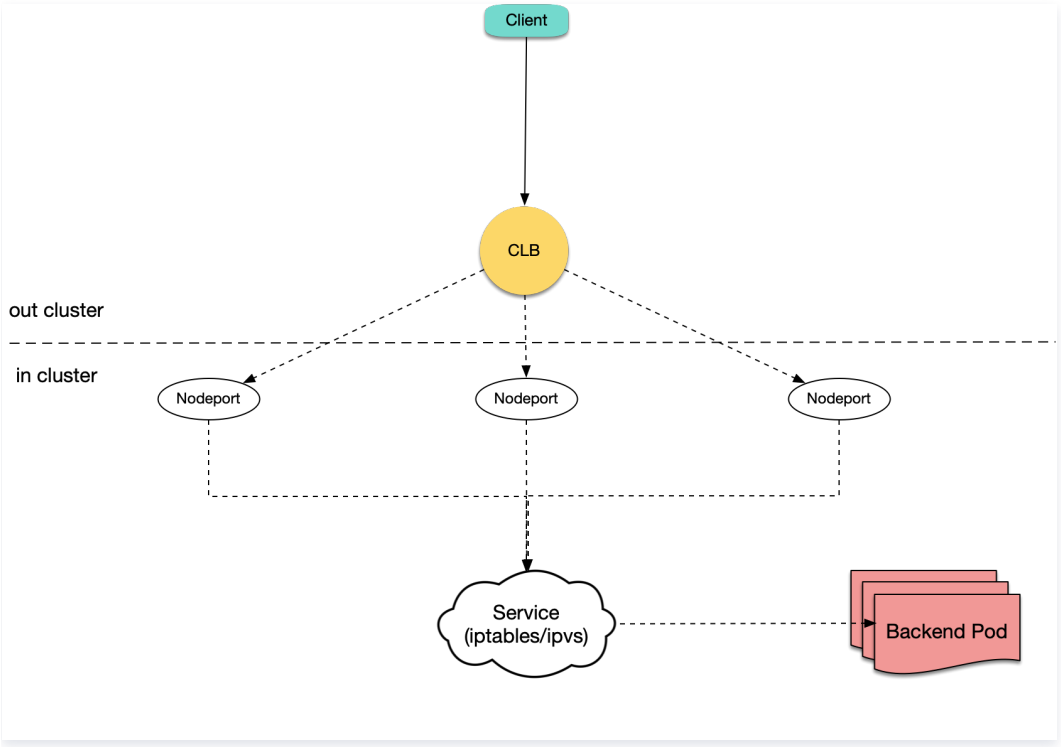
- Kubernetes Service 介绍
- Kubernetes Ingress 介绍
- Kubernetes Deployments 滚动更新策略
- Kubernetes Pods ReadinessGate 特性
- Kubernetes 通过 Local 转发获取来源 IP
- TKE 容器服务 网络模式选型
- TKE 容器服务 VPC-CNI 网络模式
- TKE 容器服务 配置 kubectl 并连接集群

在 TKE 上使用负载均衡直连 Pod

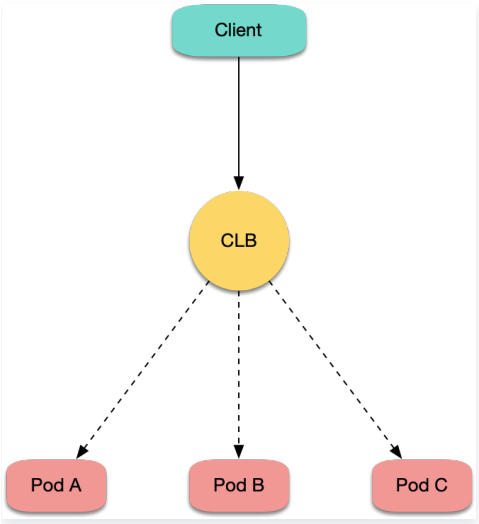
最近更新时间：2025-03-27 11:23:23

概述

Kubernetes 官方提供了 NodePort 类型的 Service，即给所有节点开通一个相同端口用于暴露该 Service。大多云上负载均衡（Cloud Load Balancer，CLB）类型 Service 的传统实现也都是基于 NodePort，即 CLB 后端绑定各节点的 NodePort，CLB 接收外界流量，转发到其中一个节点的 NodePort 上，再通过 Kubernetes 内部的负载均衡，使用 iptables 或 ipvs 转发到 Pod。示意图如下：



腾讯云容器服务 TKE 默认的 CLB 类型 Service 以及默认的 Ingress 实现方式与上述方法相同。然而，TKE 目前还支持 CLB 直连 Pod 的方式，即 CLB 后端直接绑定 Pod IP + Port，不绑定节点的 NodePort。示意图如下：



实现方式分析

传统 NodePort 方式问题分析

通常会使用 CLB 直接绑定 NodePort 此方式来创建云上 Ingress 或 LB 类型的 Service，但此传统 NodePort 实现方式会存在以下问题：

- 流量从 CLB 转发到 NodePort 后还需进行 SNAT 再转发到 Pod，造成额外的性能损耗。

- 如果流量过于集中到某几个 NodePort 时（例如，使用 nodeSelector 部署网关到固定几台节点上），可能导致源端口耗尽或 conntrack 插入冲突。
- NodePort 本身也充当负载均衡器，CLB 绑定过多节点 NodePort 时可能导致负载均衡状态过于分散，导致全局负载不均。

CLB 直连 Pod 方式优势

使用 CLB 直连 Pod 的方式不但不会存在传统 NodePort 方式的问题，还具备以下优势：

- 由于没有 SNAT，获取源 IP 不再需要 `externalTrafficPolicy: Local`。
- 实现会话保持更简单，仅需让 CLB 开启会话保持即可，不需要设置 Service 的 `sessionAffinity`。

操作场景

使用 CLB 直连 Pod 通常有以下场景：

- 需在四层获取客户端真实源 IP，但不期望使用 `externalTrafficPolicy: Local` 的方式。
- 需进一步提升网络性能。
- 需会话保持更容易。
- 解决全局连接调度的负载不均。

前提条件

Kubernetes 集群版本需高于1.12。CLB 直接绑定 Pod 时检查 Pod 是否 Ready，需查看 Pod 是否 Running、是否通过 readinessProbe，及是否通过 CLB 对 Pod 的健康监测，此项依赖于 `ReadinessGate` 特性，该特性在 Kubernetes 1.12 开始支持。

操作步骤

请参见 使用 LoadBalancer 直连 Pod 模式 Service。

参考资料

- TKE 基于弹性网卡直连 Pod 的网络负载均衡
- 集群开启 VPC-CNI 模式网络
- VPC-CNI 网络模式使用指引

在 TKE 中获取客户端真实源 IP

最近更新时间：2025-05-26 17:00:01

> **说明：**
> 本文适用于腾讯云容器服务（Tencent Kubernetes Engine，TKE），以下简称 TKE。

应用场景

当需明确服务请求来源以满足业务需求时，则需后端服务能够准确获取请求客户端的真实源 IP。例如以下场景：

- 具有对服务请求的来源进行审计的需求，例如异地登录告警。
- 具有针对安全攻击或安全事件溯源的需求，例如 APT 攻击及 DDoS 攻击等。
- 业务场景具有数据分析的需求，例如业务请求区域统计。
- 其他需获取客户端地址的需求。

实现方法

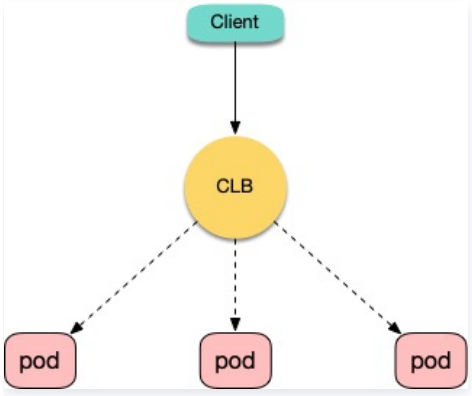
在 TKE 中默认的外部负载均衡器为 腾讯云负载均衡 作为服务流量的访问首入口，腾讯云负载均衡器会将请求流量负载转发到 Kubernetes 工作节点的 Kubernetes Service（默认）。此负载均衡过程会保留客户端真实源 IP（透传转发），但在 Kubernetes Service 转发场景下，无论使用 iptables 或 ipvs 的负载均衡转发模式，转发时都会对数据包做 SNAT，即不会保留客户端真实源 IP。在 TKE 使用场景下，本文提供以下四种方式获取客户端真实源 IP，请参考本文按需选择适用方式。

（推荐）通过 TKE 原生 CLB 直连 Pod 转发模式获取

该方式优缺点分析如下：

- **优点**：为 TKE 原生支持的功能特性，只需在控制台参考对应文档配置即可。
- **缺点**：会有一些使用限制，具体参考下面给出的文档链接。

使用 TKE 原生支持的 CLB 直连 Pod 的转发功能（CLB 透传转发，并绕过 Kubernetes Service 流量转发），后端 Pods 收到的请求的源 IP 即为客户端真实源 IP，此方式适用于四层及七层服务的转发场景。转发原理如下图：



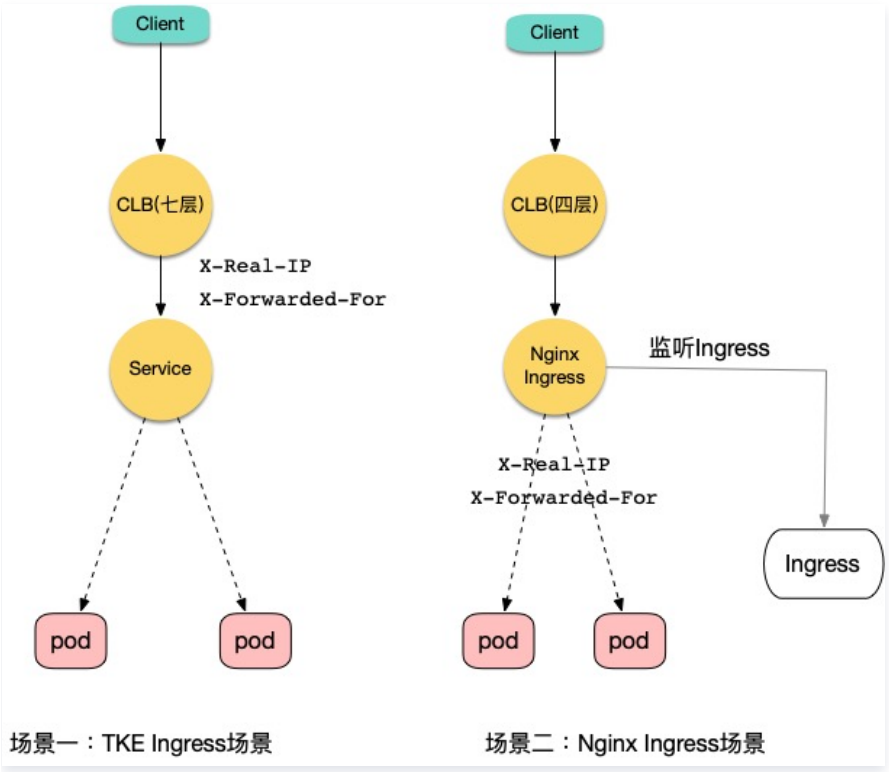
详细介绍和配置请参见 使用 LoadBalancer 直连 Pod 模式 Service 。

（推荐）通过 HTTP Header 获取

该方式优缺点分析如下：

- **优点**：在七层（HTTP/HTTPS）流量转发场景下推荐选择该方式，可通过 Web 服务代理的配置或后端应用代码直接获取 HTTP Header 中的字段，即可拿到客户端真实源 IP，非常简单高效。
- **缺点**：仅适用于七层（HTTP/HTTPS）流量转发场景，不适用于四层转发场景。

在七层（HTTP/HTTPS）服务转发场景下，可以通过获取 HTTP Header 中 `X-Forwarded-For` 和 `X-Real-IP` 字段的值来获取客户端真实源 IP。TKE 中有两种场景使用方式，原理介绍图如下所示：



场景一：使用 TKE Ingress 获取真实源 IP

腾讯云负载均衡器（CLB 七层）默认会将客户端真实源 IP 放至 HTTP Header 的 `X-Forwarded-For` 和 `X-Real-IP` 字段。当服务流量在经过 Service 四层转发后会保留上述字段，后端通过 Web 服务器代理配置或应用代码方式获取到客户端真实源 IP，详情请参见 负载均衡如何获取客户端真实IP 。通过容器服务控制台 获取源 IP 步骤如下：

1. 为工作负载创建一个主机端口访问方式的 Service 资源，本文以 nginx 为例。如下图所示：

2. 为该 Service 创建一个对应的 Ingress 访问入口，本文以 test 为例。如下图所示：

3. 待配置生效后，在后端通过获取 HTTP Header 中的 `X-Forwarded-For` 或 `X-Real-IP` 字段值得到客户端真实源 IP。后端抓包测试结果示例如下图所示：

```
▼ Hypertext Transfer Protocol
  ▼ GET / HTTP/1.1\r\n
    ▶ [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /
      Request Version: HTTP/1.1
    X-Stgw-Time: 1601284485.314\r\n
    Host: 175        r\n
    X-Client-Proto: http\r\n
    X-Forwarded-Proto: http\r\n
    X-Client-Proto-Ver: HTTP/1.1\r\n
    X-Real-IP: 61.      .r\n
    X-Forwarded-For: 61.
    Connection: keep-alive\r\n
    Proxy-Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.102 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7\r\n
    \r\n
    [Full request URI: http://175.97.145.108/]
    [HTTP request 1/1]
    [Response in frame: 458]
```

场景二：使用 Nginx Ingress 获取真实源 IP

Nginx Ingress 服务部署需要 Nginx Ingress 能直接感知客户端真实源 IP，可以采用保留客户端源 IP 的配置方式，详情请参见 Kubernetes 设置外部负载均衡器说明 。或通过 CLB 直通 Pod 的方式，详情请参见 在 TKE 上使用负载均衡直通 Pod 。当 Nginx Ingress 在转发请求时会通过 `X-Forwarded-For` 和 `X-Real-IP` 字段来记录客户端源 IP，后端可以通过此字段获得客户端真实源 IP。配置步骤如下：

1. Nginx Ingress 可以通过 TKE 应用商店、自定义 YAML 配置或使用官方（helm 安装）方式安装，原理和部署方法请参见 在 TKE 上部署 Nginx Ingress 中的部署方案1或方案3。若选择方案1部署，则需要修改 Nginx Ingress Controller Service 的 `externalTrafficPolicy` 字段值为 `Local` 。安装完成后，会在容器服务控制台 自动为 Nginx Ingress Controller 服务创建一个 CLB（四层）访问入口，如下图所示：

2. 为需转发的后端服务创建一个 Ingress 资源并配置转发规则。YAML 示例如下：

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx  # ingressClass类为"nginx"
  name: example
  namespace: default
spec:
  rules:  # 配置服务转发规则
  - http:
     paths:
       - backend:
           serviceName: nginx
           servicePort: 80
         path: /
```

3. 待配置生效后，在后端获取 HTTP Header 中的 `X-Forwarded-For` 或 `X-Real-IP` 字段值得到客户端真实源 IP。后端抓包测试结果示例如下图所示：

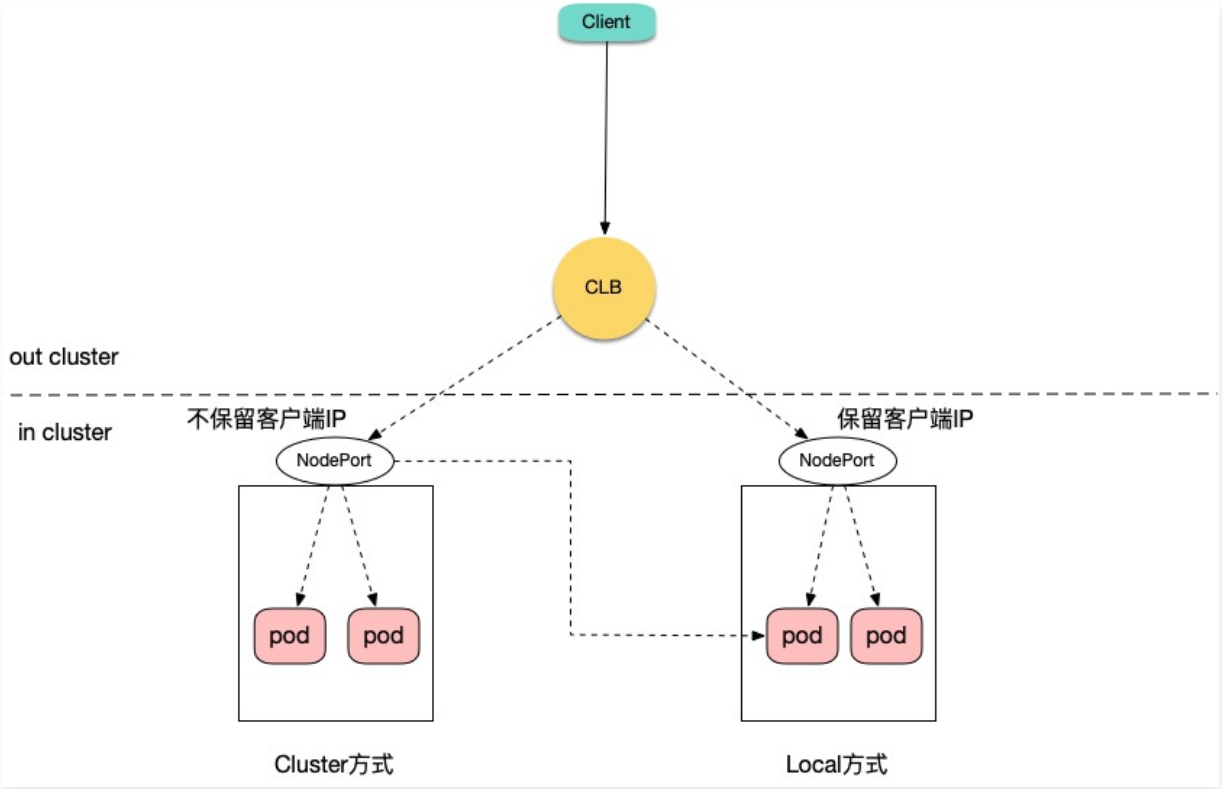
```
▼ Hypertext Transfer Protocol
  ▼ GET / HTTP/1.1\r\n
    ▶ [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /
      Request Version: HTTP/1.1
    Host: 140[redacted]\r\n
    X-Request-ID: 0980c3c5358db44caf90ec9e012d3091\r\n
    X-Real-IP: 61.[redacted]\r\n
    X-Forwarded-For: 61.[redacted]\r\n
    X-Forwarded-Host: 140.143.83.149\r\n
    X-Forwarded-Port: 80\r\n
    X-Forwarded-Proto: http\r\n
    X-Scheme: http\r\n
    Proxy-Connection: keep-alive\r\n
    Cache-Control: max-age=0\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.102 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7\r\n
    \r\n
```

通过 Service 资源的配置选项保留客户端源 IP

该方式优缺点分析如下：

- **优点**：只需修改 Kubernetes Service 资源配置即可。
- **缺点**：会存在潜在的 Pods（Endpoints）流量负载不均衡风险。在 K8s 版本为 1.26 及以上且 kube-proxy 转发模式为 iptables 的集群中，从集群内 Pod 访问此 Service 的 CLB VIP 时，若访问源端 Pod 所在节点无此 Service 所关联的 Pod，则会不通。

如需启用保留客户端 IP 功能，可在 Service 资源中配置字段 `Service.spec.externalTrafficPolicy`。该字段表示服务是否希望将外部流量路由到节点本地或集群范围的 Pods。有两个选项值：**Cluster**（默认）和 **Local** 方式。如下图所示：



- **Cluster**：表示隐藏客户端源 IP，LoadBalancer 和 NodePort 类型服务流量可能会被转发到其他节点的 Pods。
- **Local**：表示保留客户端源 IP 并避免 LoadBalancer 和 NodePort 类型的服务流量转发到其他节点的 Pods，详情请参考 Kubernetes 设置外部负载均衡器说明。相关 YAML 配置示例如下：

```
apiVersion: v1
kind: Service
metadata:
  name: example-Service
spec:
```

```
  selector:
    app: example-Service
  ports:
  - port: 8765
    targetPort: 9376
  externalTrafficPolicy: Local
  type: LoadBalancer
```

通过 TOA 内核模块加载获取真实源 IP

该方式优缺点分析如下：
优点：对于 TCP 传输方式，在内核层面且仅对 TCP 连接的首包进行改造，几乎没有性能损耗。
缺点：

- 需要在集群工作节点上加载 TOA 内核模块，且需在服务端通过函数调用获取携带的源 IP 及端口信息，配置使用较复杂。
- 对于 UDP 传输方式，会对每个数据包改造添加 option 数据（源 IP 和源端口），带来网络传输通道性能损耗。

TOA 内核模块原理和加载方式请参见 获取访问用户真实 IP 文档。

相关文档

- 腾讯云负载均衡器获取客户端真实 IP 介绍：如何获取客户端真实 IP
- 腾讯云负载均衡介绍：负载均衡 CLB
- 在 TKE 上部署 Nginx Ingress
- TKE 网络模式介绍：GlobalRouter 附加 VPC-CNI 模式说明
- 在 TKE 上使用负载均衡直通 Pod
- TOA 模块使用介绍：获取访问用户真实 IP
- Kubernetes 设置外部负载均衡器说明：创建外部负载均衡器 Kubernetes

在 TKE 上使用 Traefik Ingress

最近更新时间：2023-09-08 19:13:06

操作场景

Traefik 是一款优秀的反向代理工具，与 Nginx 相比，Traefik 具有以下优势：

- 原生支持动态配置。例如，Kubernetes 的 Ingress 资源或 IngressRoute 等 CRD 资源（Nginx 每次需重新加载完整配置，部分情况下可能会影响连接）。
- 原生支持服务发现。使用 Ingress 或 IngressRoute 等动态配置后，会自动 watch 后端 endpoint，同步更新到负载均衡的后端列表中。
- 提供美观的 Dashboard 管理页面。
- 原生支持 Metrics，与 Prometheus 和 Kubernetes 无缝集成。
- 拥有更丰富的高级功能。例如，多版本的灰度发布、流量复制、自动生成 HTTPS 免费证书、中间件等。

本文将介绍如何在 TKE 集群安装 Traefik 以及提供通过 Traefik 使用 Ingress 和 IngressRoute 示例。

前提条件

- 已创建 TKE 集群 并能够通过 Kubectl 连接集群。
- 已安装 Helm。

操作步骤

安装 Traefik

本文提供以下在 TKE 集群上安装 Traefik 为例，完整安装方法请参见 Traefik 官方文档。

1. 执行以下命令，添加 Traefik 的 Helm chart repo 源。示例如下：

```
helm repo add traefik https://helm.traefik.io/traefik
```

2. 准备安装配置文件 `values-traefik.yaml`。示例如下：

```
providers:
  kubernetesIngress:
    publishedService:
      enabled: true # 让 Ingress 的外部 IP 地址状态显示为 Traefik 的 LB IP 地址
additionalArguments:
- "--providers.kubernetesingress.ingressclass=traefik" # 指定 ingress class 名称
- "--log.level=DEBUG"
service:
  annotations:
    service.cloud.tencent.com/direct-access: "true" # 网关类的应用建议使用 LB 直通 Pod（绕过 NodePort）。若使用 VPC-CNI 与 Global Router 两种网络模式混用，用此注解来显示声明 LB 直绑 Pod（绕过 NodePort）；若创建集群时就选的 VPC-CNI 网络模式，则不需要显示声明（默认 LB 直通 Pod）。详情请参见官方文档 https://cloud.tencent.com/document/product/457/48793
    service.kubernetes.io/tke-existed-lbid: lb-lb57hvgl # 用此注解绑定提前创建好的 LB，使得即便 Traefik 日后重建，也能保证流量入口不变。若不指定此注解，默认自动创建新的 LB。详情请参见官方文档 https://cloud.tencent.com/document/product/457/45491
ports:
  web:
    expose: true
    exposedPort: 80 # 对外的 HTTP 端口号，使用标准端口号在国内需备案
  websecure:
    expose: true
    exposedPort: 443 # 对外的 HTTPS 端口号，使用标准端口号在国内需备案
deployment:
  enabled: true
  replicas: 1
  podAnnotations:
```

```
    tke.cloud.tencent.com/networks: "tke-route-eni" # 在 VPC-CNI 与 Global Router 两种网络模式混用的情况下，显示声明 Pod 要使用弹性网卡，与以下 eni-ip 的 request 与 limit 一起配合使用
  resources:
    requests:
      tke.cloud.tencent.com/eni-ip: "1"
    limits:
      tke.cloud.tencent.com/eni-ip: "1"
```

> **说明**
> 完整的默认配置可执行 `helm show values traefik/traefik` 命令查看。

3. 执行以下命令将 Traefik 安装到 TKE 集群。示例如下：

```
kubectl create ns ingress
helm upgrade --install traefik -f values-traefik.yaml traefik/traefik
```

4. 执行以下命令，获取流量入口的 IP 地址（如下为 EXTERNAL-IP 字段）。示例如下：

```
$ kubectl get service -n ingress
NAME      TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)                      AGE
traefik   LoadBalancer   172.22.252.242   49.233.239.84   80:31650/TCP,443:32288/TCP   42h
```

使用 Ingress

Traefik 支持使用 Kubernetes 的 Ingress 资源作为动态配置，可直接在集群中创建 Ingress 资源用于对外暴露集群，需要加上指定的 Ingress class（安装 Traefik 时定义）。示例如下：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: traefik # 这里指定 ingress class 名称
spec:
  rules:
  - host: traefik.demo.com
    http:
      paths:
      - path: /test
        backend:
          serviceName: nginx
          servicePort: 80
```

> **注意**
> TKE 暂未将 Traefik 产品化，无法直接在 TKE 控制台进行可视化创建 Ingress，需要使用 YAML 进行创建。

使用 IngressRoute

Traefik 不仅支持标准的 Kubernetes Ingress 资源，也支持 Traefik 特有的 CRD 资源，例如 IngressRoute，可以支持更多 Ingress 不具备的高级功能。IngressRoute 使用示例如下：

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
```

```
  name: test-ingressroute
spec:
  entryPoints:
    - web
  routes:
    - match: Host(`traefik.demo.com`) && PathPrefix(`/test`)
      kind: Rule
      services:
        - name: nginx
          port: 80
```

说明

Traefik 更多用法请参见 Traefik 官方文档。

发布
使用 CLB 实现简单的蓝绿发布和灰度发布

最近更新时间：2024-09-02 17:34:41

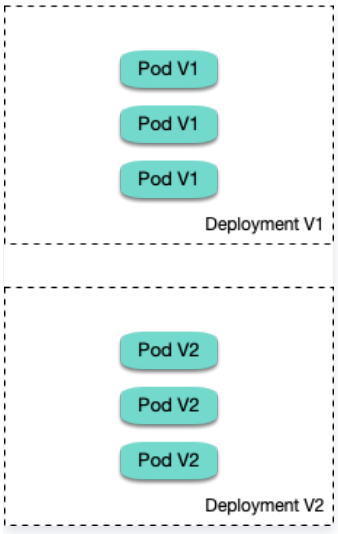
操作场景

腾讯云 Kubernetes 集群实现蓝绿发布或灰度发布通常需向集群额外部署其他开源工具，例如 Nginx Ingress、Traefik 或将业务部署至服务网格 Service Mesh，利用服务网格的能力实现。这些方案均具有一定难度，若您的蓝绿发布或灰度需求不复杂，且不希望集群引入过多的组件或复杂的用法，可以参考本文利用 Kubernetes 原生特性以及腾讯云容器服务 TKE 标准集群和 TKE Serverless 集群自带的 LB 插件，实现简单的蓝绿发布和灰度发布。

> **注意：**
> 本文仅适用于 TKE 标准集群及 TKE Serverless 集群。

原理介绍

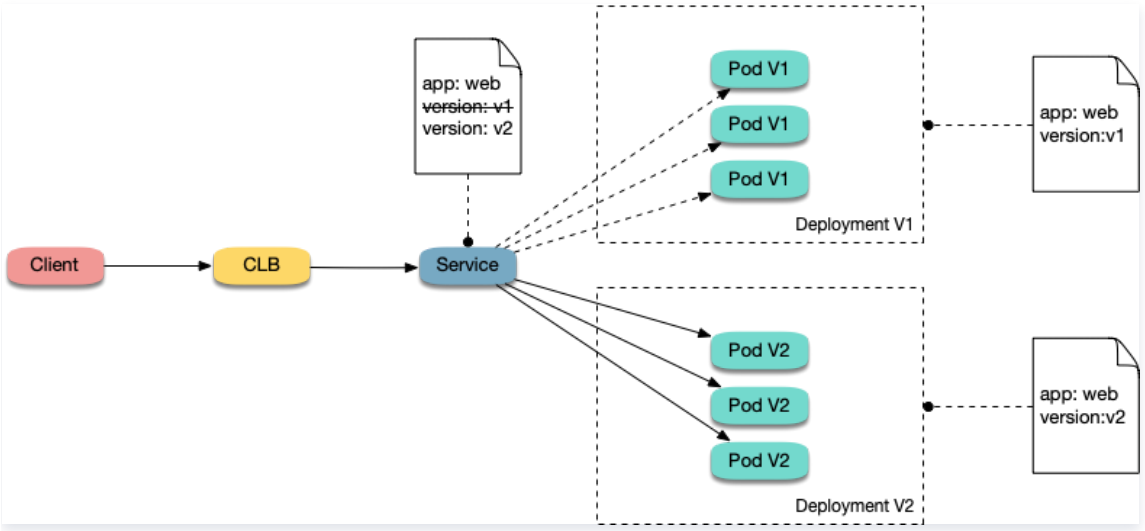
用户通常使用 Deployment、StatefulSet 等 Kubernetes 自带的工作负载来部署业务，每个工作负载管理一组 Pod。以 Deployment 为例，示意图如下：



通常还会为每个工作负载创建对应的 Service，Service 通过 selector 来匹配后端 Pod，其他服务或者外部通过访问 Service 即可访问到后端 Pod 提供的服务。如需对外暴露可直接将 Service 类型设置为 LoadBalancer，LB 插件会自动为其创建腾讯云负载均衡 CLB 作为流量入口。

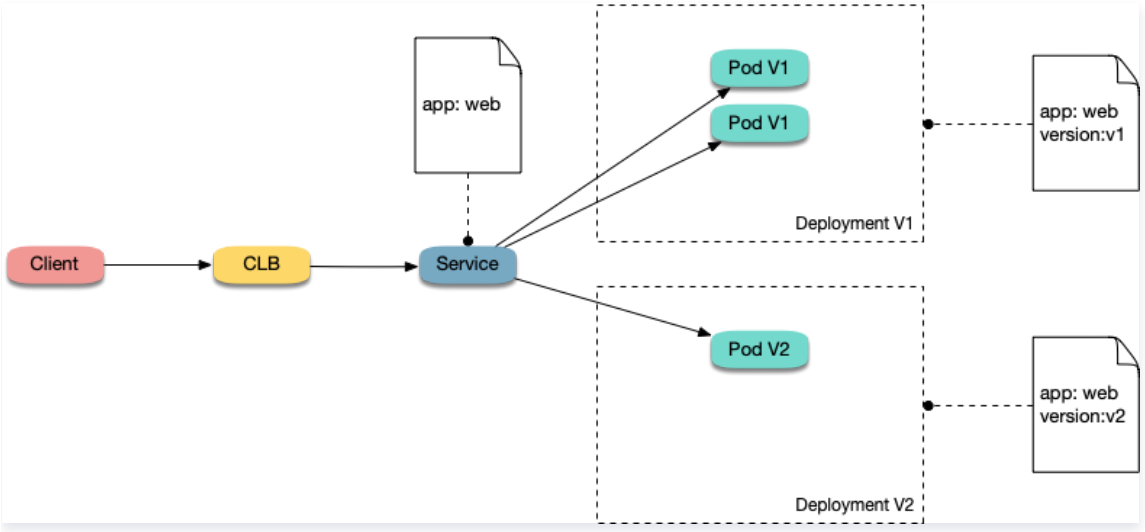
蓝绿发布原理

以 Deployment 为例，集群中已部署两个不同版本的 Deployment，其 Pod 拥有共同的 label。但有一个 label 值不同，用于区分不同的版本。Service 使用 selector 选中了其中一个版本的 Deployment 的 Pod，此时通过修改 Service 的 selector 中决定服务版本的 label 的值来改变 Service 后端对应的 Deployment，即可实现让服务从一个版本直接切换到另一个版本。示意图如下：



灰度发布原理

用户通常会为每个工作负载创建一个 Service，但 Kubernetes 未限制 Service 需与工作负载一一对应。Service 通过 selector 匹配后端 Pod，若不同工作负载的 Pod 被同一 selector 选中，即可实现一个 Service 对应多个版本工作负载。调整不同版本工作版本的副本数即调整不同版本服务的权重。示意图如下：



操作步骤

使用 YAML 创建资源

本文提供以下两种方式使用 YAML 部署工作负载及创建 Service：

- 方式1：登录 容器服务控制台，选择集群 ID 进入集群详情页，单击详情页右上角的 **YAML 创建资源**，并将本文示例的 YAML 文件内容输入编辑界面。
- 方式2：将示例 YAML 保存为文件，再使用 kubectl 指定 YAML 文件进行创建。例如 `kubectl apply -f xx.yaml` 。

部署多版本工作负载

1. 在集群中部署第一个版本的 Deployment，本文以 nginx 为例。YAML 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v1
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - name: nginx
        image: "openresty/openresty:centos"
        ports:
        - name: http
          protocol: TCP
          containerPort: 80
        volumeMounts:
        - mountPath: /usr/local/openresty/nginx/conf/nginx.conf
          name: config
          subPath: nginx.conf
      volumes:
      - name: config
        configMap:
          name: nginx-v1

---

apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app: nginx
    version: v1
  name: nginx-v1
data:
  nginx.conf: |-
    worker_processes 1;

    events {
        accept_mutex on;
        multi_accept on;
        use epoll;
        worker_connections 1024;
    }

    http {
        ignore_invalid_headers off;
        server {
            listen 80;
            location / {
                access_by_lua '
                    local header_str = ngx.say("nginx-v1")
                ';
            }
        }
```

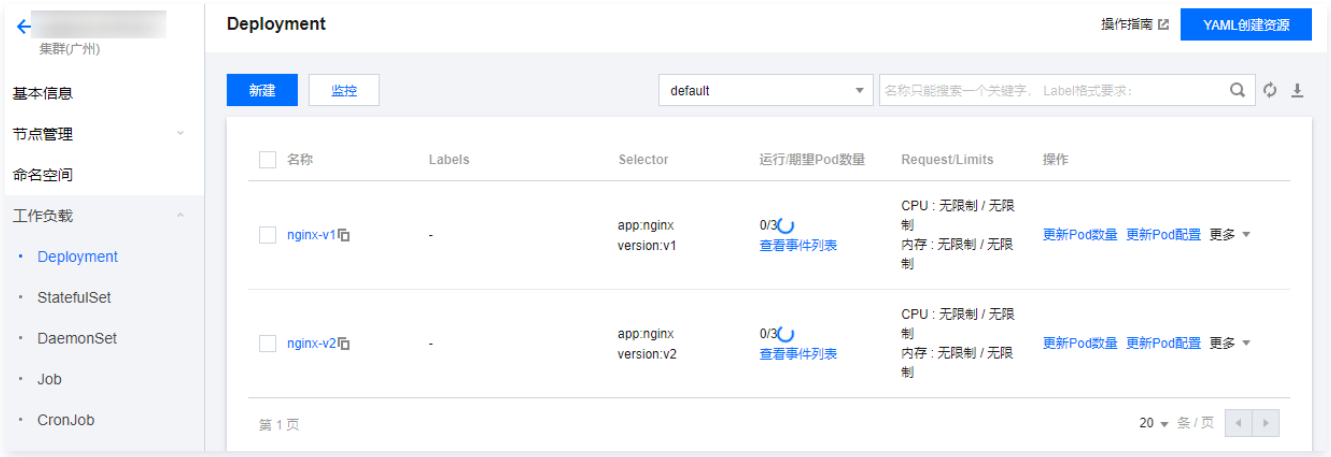
```
    }
```

2. 再部署第二个版本的 Deployment，本文以 nginx 为例。YAML 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v2
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      version: v2
  template:
    metadata:
      labels:
        app: nginx
        version: v2
    spec:
      containers:
      - name: nginx
        image: "openresty/openresty:centos"
        ports:
        - name: http
          protocol: TCP
          containerPort: 80
        volumeMounts:
        - mountPath: /usr/local/openresty/nginx/conf/nginx.conf
          name: config
          subPath: nginx.conf
      volumes:
      - name: config
        configMap:
          name: nginx-v2
---

apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app: nginx
    version: v2
  name: nginx-v2
data:
  nginx.conf: |-
    worker_processes  1;

    events {
        accept_mutex on;
        multi_accept on;
        use epoll;
        worker_connections  1024;
    }

    http {
        ignore_invalid_headers off;
        server {
            listen 80;
```

```
            location / {
                access_by_lua '
                    local header_str = ngx.say("nginx-v2")
                ';
            }
        }
    }
```

3. 登录 容器服务控制台，在集群的工作负载 > Deployment 页查看部署情况。如下图所示：



实现蓝绿发布

1. 为部署的 Deployment 创建 LoadBalancer 类型的 Service 对外暴露服务，指定使用 v1 版本的服务。YAML 示例如下：

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

2. 执行以下命令，测试访问。

```
for i in {1..10}; do curl EXTERNAL-IP; done; # 替换 EXTERNAL-IP 为 Service 的 CLB IP 地址
```

返回结果如下，均为 v1 版本的响应。

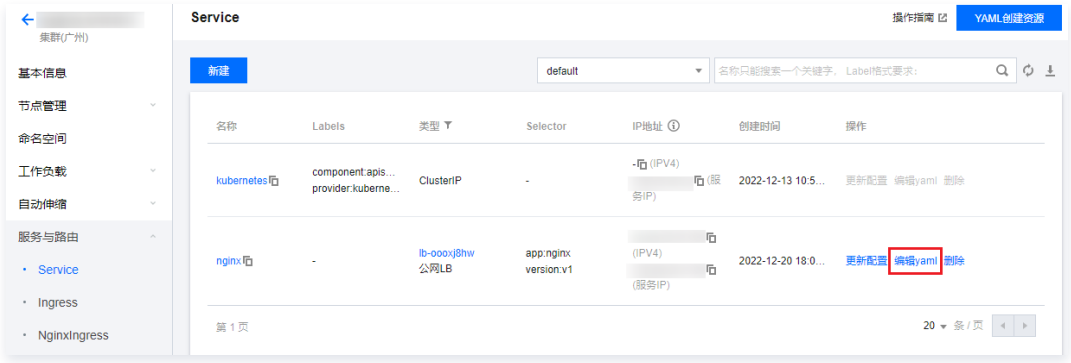
```
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
```

```
nginx-v1
```

3. 通过控制台或 kubectl 方式修改 Service 的 selector，使其选中 v2 版本的服务：

通过控制台

1. 进入集群详情页，选择左侧**服务与路由** > **Service**。
2. 在 **Service** 页面中选择需修改 Service 所在行右侧的**编辑 YAML**。如下图所示：



修改 selector 部分为如下内容：

```
selector:
    app: nginx
    version: v2
```

3. 单击**完成**。

通过 kubectl

执行以下命令：

```
kubectl patch service nginx -p '{"spec":{"selector":{"version":"v2"}}}'
```

4. 执行以下命令，再次测试访问。

```
$ for i in {1..10}; do curl EXTERNAL-IP; done; # 替换 EXTERNAL-IP 为 Service 的 CLB IP 地址
```

返回结果如下，均为 v2 版本的响应，成功实现了蓝绿发布。

```
nginx-v2
nginx-v2
nginx-v2
nginx-v2
nginx-v2
nginx-v2
nginx-v2
nginx-v2
nginx-v2
nginx-v2
```

实现灰度发布

1. 对比蓝绿发布，不指定 Service 使用 v1 版本服务。即从 selector 中删除 `version` 标签，让 Service 同时选中两个版本的 Deployment 的 Pod。YAML 示例如下：

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

2. 执行以下命令，测试访问。

```
for i in {1..10}; do curl EXTERNAL-IP; done; # 替换 EXTERNAL-IP 为 Service 的 CLB IP 地址
```

返回结果如下，一半是 v1 版本的响应，另一半是 v2 版本的响应。

```
nginx-v1
nginx-v1
nginx-v2
nginx-v2
nginx-v2
nginx-v1
nginx-v1
nginx-v1
nginx-v2
nginx-v2
```

3. 通过控制台或 kubectl 方式调节 v1 和 v2 版本的 Deployment 的副本，将 v1 版本调至 1 个副本，v2 版本调至 4 个副本：
   - **通过控制台修改：**
     3.1.1 进入集群的**工作负载 > Deployment** 页，选择 v1 版本 Deployment 所在行右侧的**更多 > 编辑YAML**。
     3.1.2 在 YAML 编辑页面，将 v1 版本的 `.spec.replicas` 修改为1并单击**完成**。
     3.1.3 重复上述步骤，将 v2 版本的 `.spec.replicas` 修改为4并单击**完成**。
   - **通过 kubectl 修改：**

```
kubectl scale deployment/nginx-v1 --replicas=1
kubectl scale deployment/nginx-v2 --replicas=4
```

4. 执行以下命令，再次进行访问测试。

```
for i in {1..10}; do curl EXTERNAL-IP; done; # 替换 EXTERNAL-IP 为 Service 的 CLB IP 地址
```

返回结果如下，10次访问中仅2次返回了 v1 版本，v1 与 v2 的响应比例与其副本数比例一致，为 1:4。通过控制不同版本服务的副本数就实现了灰度发布。

```
nginx-v2
nginx-v1
nginx-v2
nginx-v2
nginx-v2
nginx-v2
```

```
nginx-v1
nginx-v2
nginx-v2
nginx-v2
```

日志
TKE 日志采集最佳实践

最近更新时间：2024-08-28 09:50:11

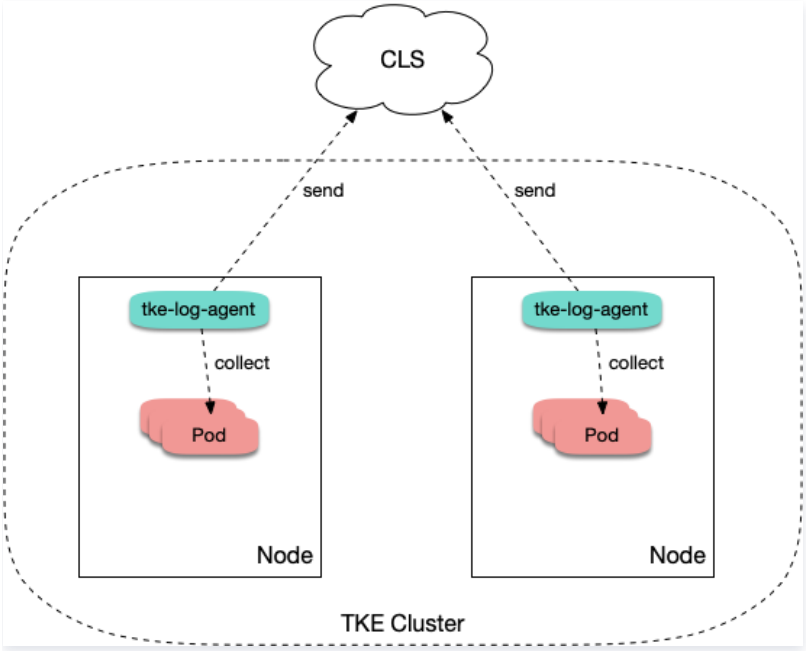
概述

本文介绍了容器服务 TKE 的日志功能，包括日志采集、存储、查询等功能的使用方法，并结合实际应用场景提供建议。您可以根据实际情况参考本文进行日志采集实践。

> **说明：**
> 本文仅适用于 TKE 集群。关于 TKE 集群如何启用日志采集及其基础用法，请参见 日志采集 。

技术架构

TKE 集群开启日志采集后，tke-log-agent 作为 DaemonSet 部署在每个节点上。它会根据采集规则采集节点上容器的日志，并上报至日志服务 CLS，由 CLS 进行统一存储、检索与分析。示意图如下：



采集类型使用场景

在使用 TKE 日志采集功能时，需要在新建日志采集规则时确定采集的目标数据源。TKE 支持“采集标准输出”、“采集容器内文件”及“采集宿主机文件”三种采集类型。请参考下文了解各类型使用场景及建议：

采集标准输出

“采集标准输出”是将 Pod 内容器日志输出到标准输出，日志内容由容器运行时（docker 或 containerd）管理。此方式最简单，推荐选择。具备以下优势：

- 不需要额外挂载 volume。
- 可直接通过 `kubectl logs` 查看日志内容。
- 业务无需关注日志轮转，容器运行时会对日志进行存储和自动轮转，避免因个别 Pod 日志量大将磁盘写满。
- 无需关注日志文件路径，可以使用较统一的采集规则，用更少的采集规则数量覆盖更多的工作负载，减少运维复杂度。

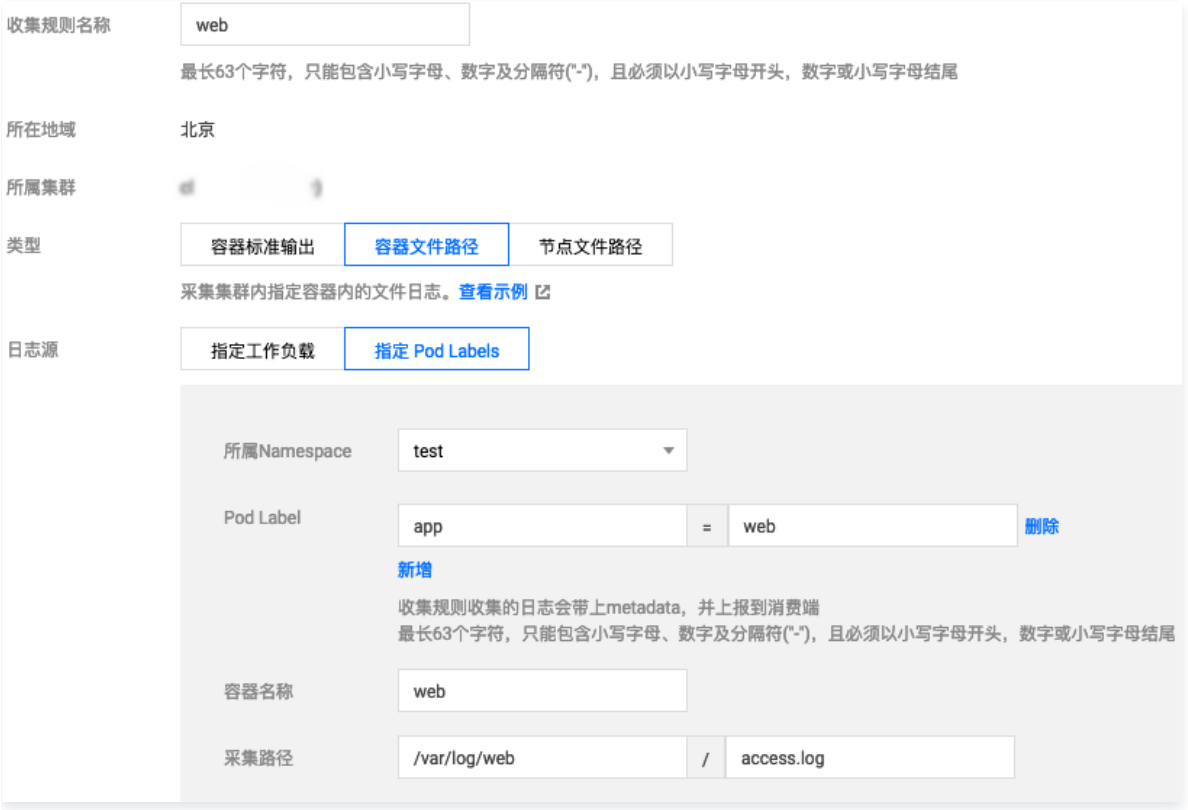
采集配置示例如下图所示，如何配置请参见 采集容器标准输出日志 。

采集容器内的文件

通常业务会使用写日志文件的方式来记录日志，当使用容器运行业务时，日志文件被写在容器内。请了解以下事项：

- 若日志文件所在路径未挂载 volume：
  日志文件会被写入容器可写层，落盘到容器数据盘里，通常路径是 `/var/lib/docker` 。建议挂载 volume 至该路径，避免与系统盘混用。容器停止后日志会被清理。
- 若日志文件所在路径已挂载 volume：
  日志文件会落盘到对应 volume 类型的后端存储，通常用 emptydir。容器停止后日志会被清理，运行期间日志文件会落盘到宿主机的 `/var/lib/kubelet` 路径下，此路径通常没有单独挂盘，即会使用系统盘。由于使用了日志采集功能，有统一存储的能力，不推荐再挂载其它持久化存储来存日志文件（例如云硬盘 CBS、对象存储 COS 或共享存储 CFS）。

大部分开源日志采集器需给 Pod 日志文件路径挂载 volume 后才可采集，而 TKE 的日志采集无需挂载。若将日志输出到容器内的文件，则无需关注是否挂载 volume。采集配置示例如下图所示，如何配置请参见 采集容器内文件日志 。



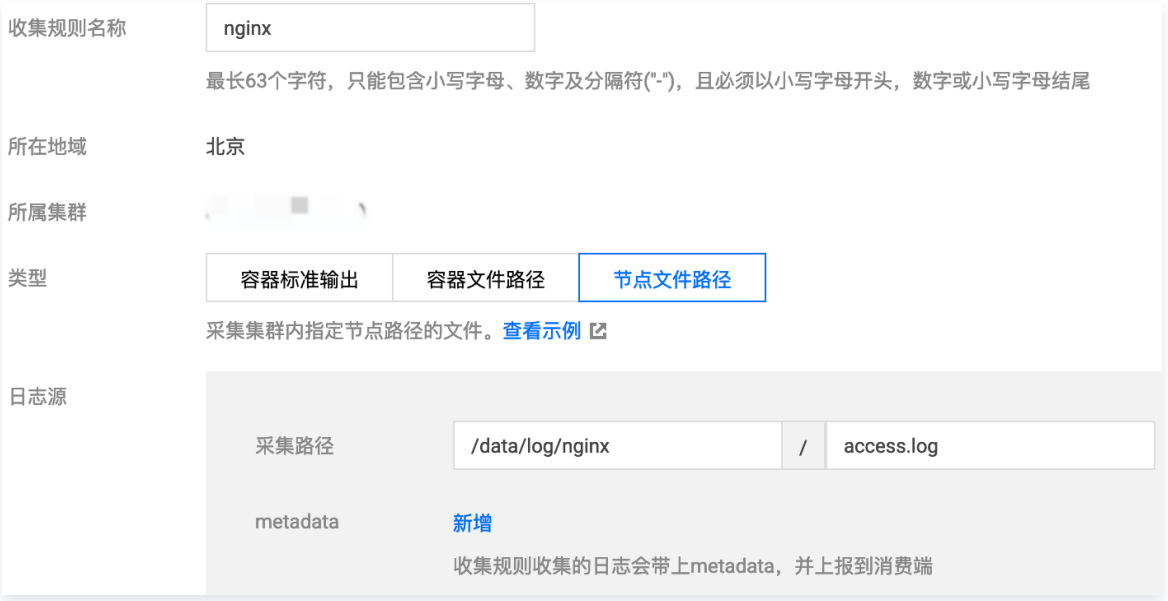
采集宿主机上的文件

若业务需将日志写入日志文件，但期望在容器停止后仍保留原始日志文件作为备份，避免采集异常时日志完全丢失。此时可以给日志文件路径挂载 hostPath，日志文件会落盘到宿主机指定目录，并且容器停止后不会清理日志文件。由于不会自动清理日志文件，可能会发生 Pod 调度走再调度回来，日志文件被写在相同路径，从而重复采集的问题。采集分以下两种情况：

- 文件名相同：
  例如，固定文件路径 `/data/log/nginx/access.log` 。此时不会重复采集，采集器会记住之前采集过的日志文件的位点，只采集增量部分。
- 文件名不同：
  通常业务用的日志框架会按照一定时间周期自动进行日志轮转，一般是按天轮转，并自动为旧日志文件进行重命名，加上时间戳后缀。如果采集规则里使用了 `*` 为通配符匹配日志文件名，则可能发生重复采集。日志框架对日志文件重命名后，采集器则会认为匹配到了新写入的日志文件，就又对其进行采集一次。

> **说明：**
> 通常情况下不会发生重复采集，若日志框架会对日志进行自动轮转，建议采集规则不要使用通配符 `*` 来匹配日志文件。

采集配置示例如下图所示，如何配置请参见 采集节点文件日志 。



日志输出

TKE 日志采集与云上的 CLS 日志服务集成，日志数据也将统一上报到日志服务。CLS 通过日志集和日志主题来对日志进行管理，日志集是 CLS 的项目管理单元，可以包含多个日志主题。一般将同一个业务的日志放在一个同一日志集，同一业务中的同一类的应用或服务使用相同日志主题。在 TKE 中，日志采集规则与日志主题一一对应。TKE 创建日志采集规则时选择消费端，则需要指定日志集与日志主题。其中，日志集通常提前创建好，日志主题通常选择自动创建。如下图所示：

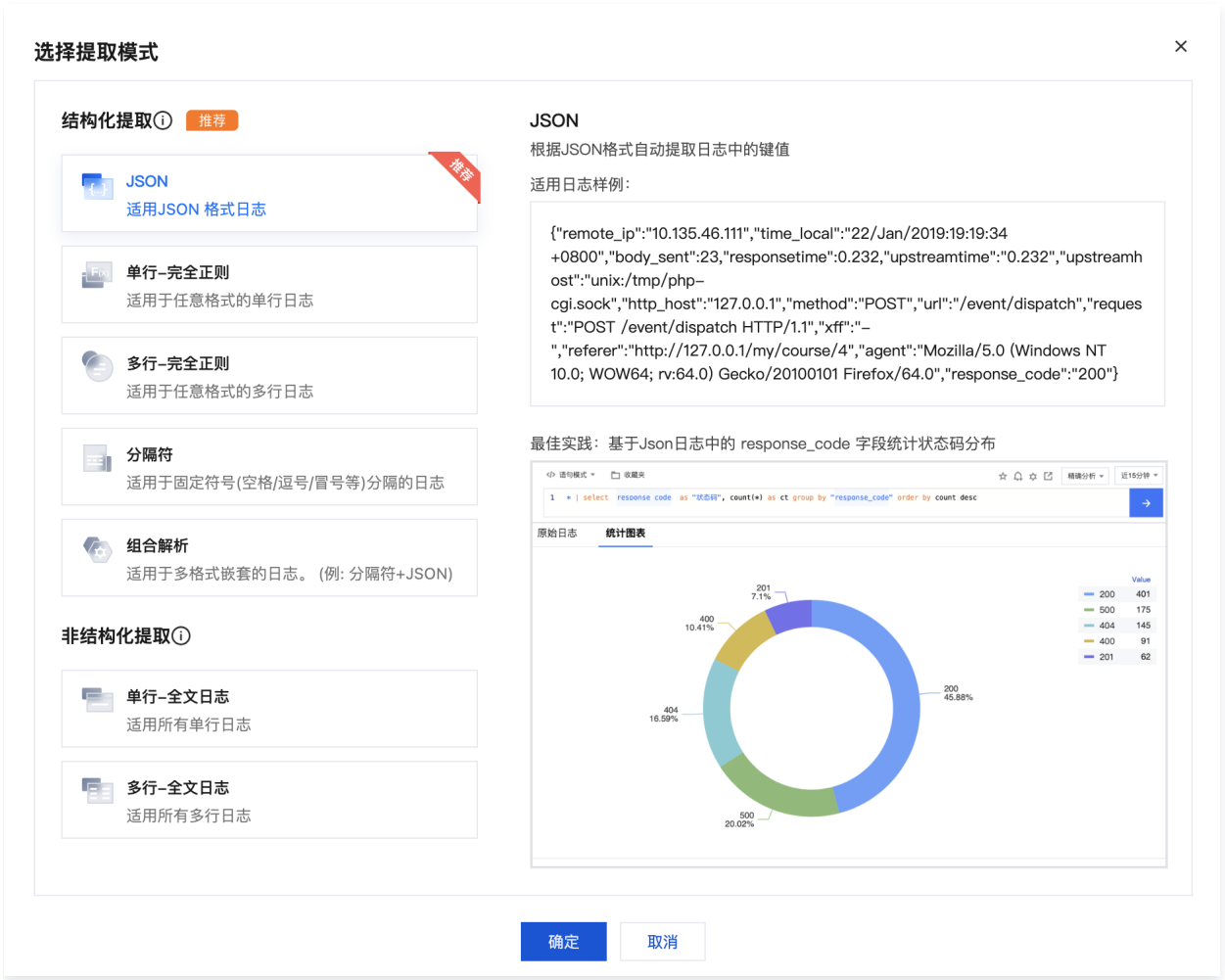
自动创建日志主题后，可前往 日志集管理 的对应日志集详情页面，进行重命名操作，以便后续检索时快速找到日志所在的日志主题。

配置日志格式解析

在创建日志采集规则时，需配置日志的解析格式，以便后续对其进行检索。请参考以下内容，对应实际情况进行配置。

选择提取模式

TKE 支持单行文本、多行文本、JSON、分隔符、完全正则、组合解析提取模式。如下图所示：



JSON 模式

选择 “JSON 模式”需日志本身是以 JSON 格式输出的，推荐选择该模式。JSON 格式本身已将日志结构化，CLS 可以提取 JSON 的 key 作为字段名，value 作为对应的字段值，不再需要根据业务日志输出格式配置复杂的匹配规则。日志示例如下：

```
{"remote_ip":"10.135.46.111","time_local":"22/Jan/2019:19:19:34
+0800","body_sent":23,"responsetime":0.232,"upstreamtime":"0.232","upstreamhost":"unix:/tmp/php-
cgi.sock","http_host":"127.0.0.1","method":"POST","url":"/event/dispatch","request":"POST
/event/dispatch HTTP/1.1","xff":"-","referer":"http://127.0.0.1/my/course/4","agent":"Mozilla/5.0
(Windows NT 10.0; WOW64; rv:64.0) Gecko/20100101 Firefox/64.0","response_code":"200"}
```

单行文本及多行文本模式

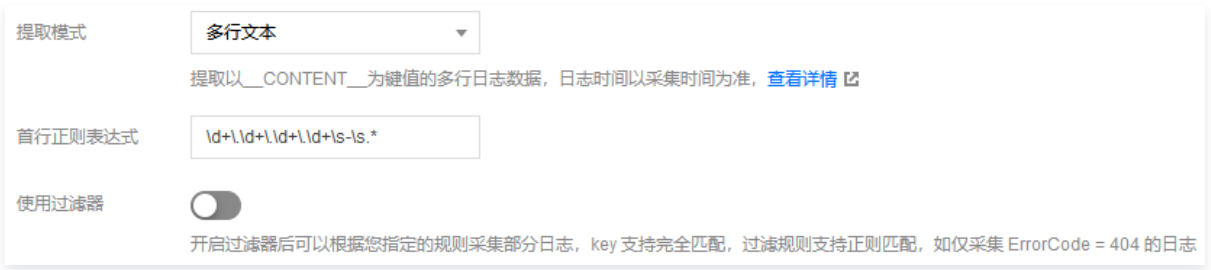
如果日志没有固定的输出格式，则考虑使用“单行文本”或“多行文本”的提取模式。使用这两种模式，不会对日志内容本身进行结构化处理及提取日志字段，每条日志的时间戳固定由日志采集的时间决定，检索时仅能进行简单的模糊查询。两种模式的区别在于日志内容为单行还是多行：

- **单行：无需设置任何匹配条件，每行为一条单独的日志。**
- **多行：需设置首行正则表达式，即匹配每条日志第一行的正则。当某行日志匹配上预先设置的首行正则表达式，即认为是一条日志的开头，而下一个行首出现作为该条日志的结束标识符。假设多行日志内容是：**

```
10.20.20.10 - - [Tue Jan 22 14:24:03 CST 2019 +0800] GET /online/sample HTTP/1.1 127.0.0.1 200 628
35 http://127.0.0.1/group/1
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:64.0) Gecko/20100101 Firefox/64.0 0.310 0.310
```

```
```

则首行正则表达式就可以设置为：`\d+\.\d+\.\d+\.\d+\s-\s.*`。如下图所示：



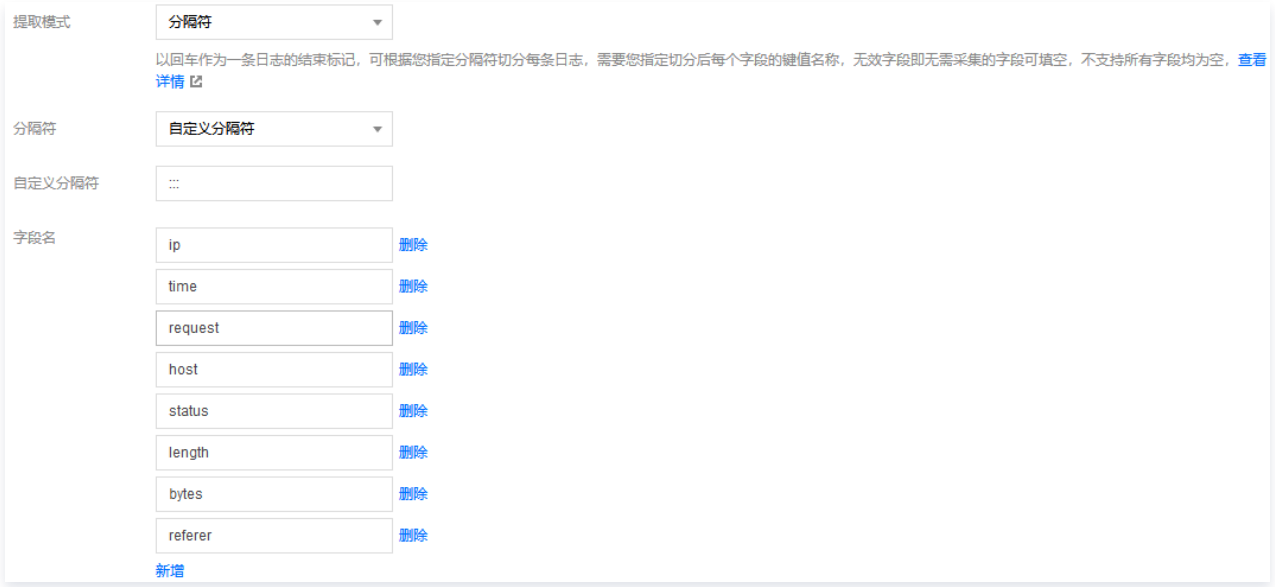
分隔符及完全正则模式

如果日志内容是以固定格式输出的单行文本，则考虑使用“分隔符”或“完全正则”提取模式：

- “分隔符”适用简单格式，日志中每个字段值都以固定的字符串分隔开。例如，用 `:::` 隔开，某一条日志内容为：

```
10.20.20.10 ::: [Tue Jan 22 14:49:45 CST 2019 +0800] ::: GET /online/sample HTTP/1.1 ::: 127.0.0.1
::: 200 ::: 647 ::: 35 ::: http://127.0.0.1/
```

则可以配置 `:::` 自定义分隔符，并且为每个字段按顺序配置字段名。如下图所示：



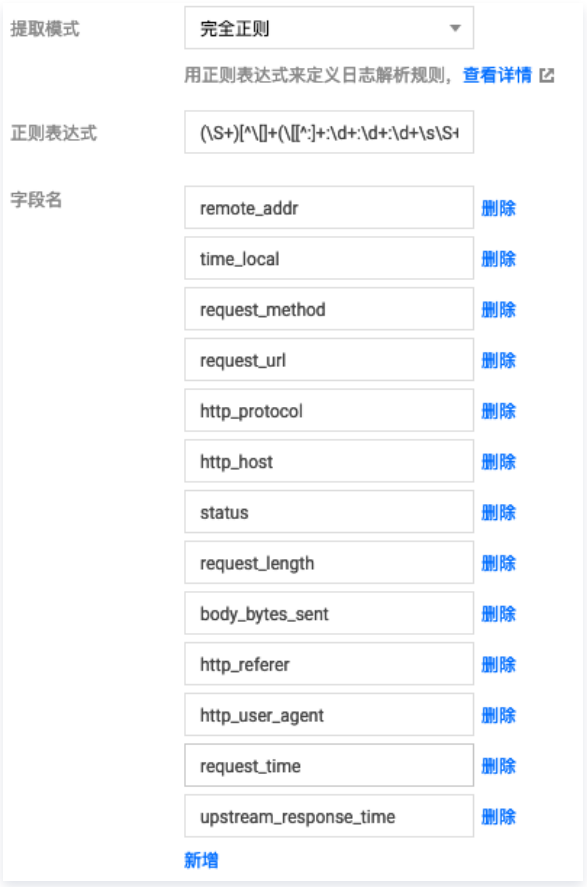
- “完全正则”适用复杂格式，使用正则表达式来匹配日志的格式。例如日志内容为：

```
10.135.46.111 - - [22/Jan/2019:19:19:30 +0800] "GET /my/course/1 HTTP/1.1" 127.0.0.1 200 782 9703
"http://127.0.0.1/course/explore?
filter%5Btype%5D=all&filter%5Bprice%5D=all&filter%5BcurrentLevelId%5D=all&orderBy=studentNum"
"Mozilla/5.0 (Windows NT 10.0; WOW64; rv:64.0) Gecko/20100101 Firefox/64.0"  0.354 0.354
```

则正则表达式就可以设置为：

```
(\S+)[^\[]+(\[[^:]+:\d+:\d+:\d+\s\S+)\s"(\w+)\s(\S+)\s([^"]+)"\s(\S+)\s(\d+)\s(\d+)\s(\d+)\s"
([^"]+)"\s"([^"]+)"\s+(\S+)\s(\S+).*
```

CLS 会使用 `()` 捕获组来区分每个字段，还需要为每个字段设置字段名。如下图所示：



组合解析

假设您的一条日志的原始数据为：

```
1571394459, http://127.0.0.1/my/course/4|10.135.46.111|200, status:DEAD,
```

自定义插件内容如下：

```
{
 "processors": [
   {
     "type": "processor_split_delimiter",
     "detail": {
       "Delimiter": ",",
       "ExtractKeys": [ "time", "msg1","msg2"]
     },
     "processors": [
       {
         "type": "processor_timeformat",
         "detail": {
           "KeepSource": true,
           "TimeFormat": "%s",
           "SourceKey": "time"
          }
       },
       {
         "type": "processor_split_delimiter",
```

```
          "detail": {
            "KeepSource": false,
            "Delimiter": "|",
            "SourceKey": "msg1",
            "ExtractKeys": [ "submsg1","submsg2","submsg3"]
          },
          "processors": []
        },
        {
          "type": "processor_split_key_value",
          "detail": {
            "KeepSource": false,
            "Delimiter": ": ",
            "SourceKey": "msg2"
          }
        }
      ]
    }
  ]
}
```

经过日志服务结构化处理后，该条日志将变为如下：

```
time: 1571394459
submsg1: http://127.0.0.1/my/course/4
submsg2: 10.135.46.111
submsg3: 200
status: DEAD
```

配置过滤内容

可选择过滤无需使用的日志信息，降低成本。

- 若使用 “JSON”、“分隔符”或“完全正则”的提取模式，日志内容会进行结构化处理，可以通过指定字段来对要保留的日志进行正则匹配。如下图所示：

- 若使用“单行文本”和“多行文本”的提取模式，由于日志内容没有进行结构化处理，无法指定字段来过滤，通常直接使用正则来对要保留的完整日志内容进行模糊匹配。如下图所示：

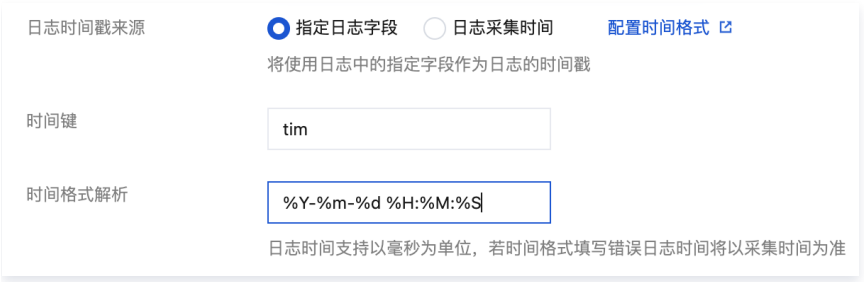
> **注意：**
> 匹配内容需使用正则而不是完整匹配。例如，需仅保留 `a.test.com` 域名的日志，匹配的表达式应为 `a\.test\.com` 而不是 `a.test.com`。

自定义日志时间戳

每条日志都需要具备主要用于检索的时间戳，可在检索时选择时间范围。默认情况下，日志的时间戳由采集的时间决定，您也可以进行自定义，选择某个字段作为时间戳，在有些场景下会更加精确。例如，在创建采集规则前，服务已运行一段时间，若不设置自定义时间格式，采集时会将之前的旧日志的时间戳设置为当前的时间，导致时间不准确。

“单行文本”和“多行文本”提取模式不会对日志内容进行结构化处理，无字段可指定为时间戳，即不支持此功能。其他提取模式均支持此功能，选取需作为时间戳的字段名称并配置时间格式。例如，使用日志的 `time` 字段作为时间戳，其中一条日志 `time` 的值为 `2020-09-22 18:18:18`，时间格式即可设置为 `%Y-%m-%d %H:%M:%S`。如下图所示：

> **注意：**
> 日志服务时间戳目前支持精确到秒，若业务日志的时间戳字段精确到毫秒，则将无法使用自定义时间戳，只能使用默认的采集时间作为时间戳。



更多时间格式配置信息请参见 配置时间格式 。

查询日志

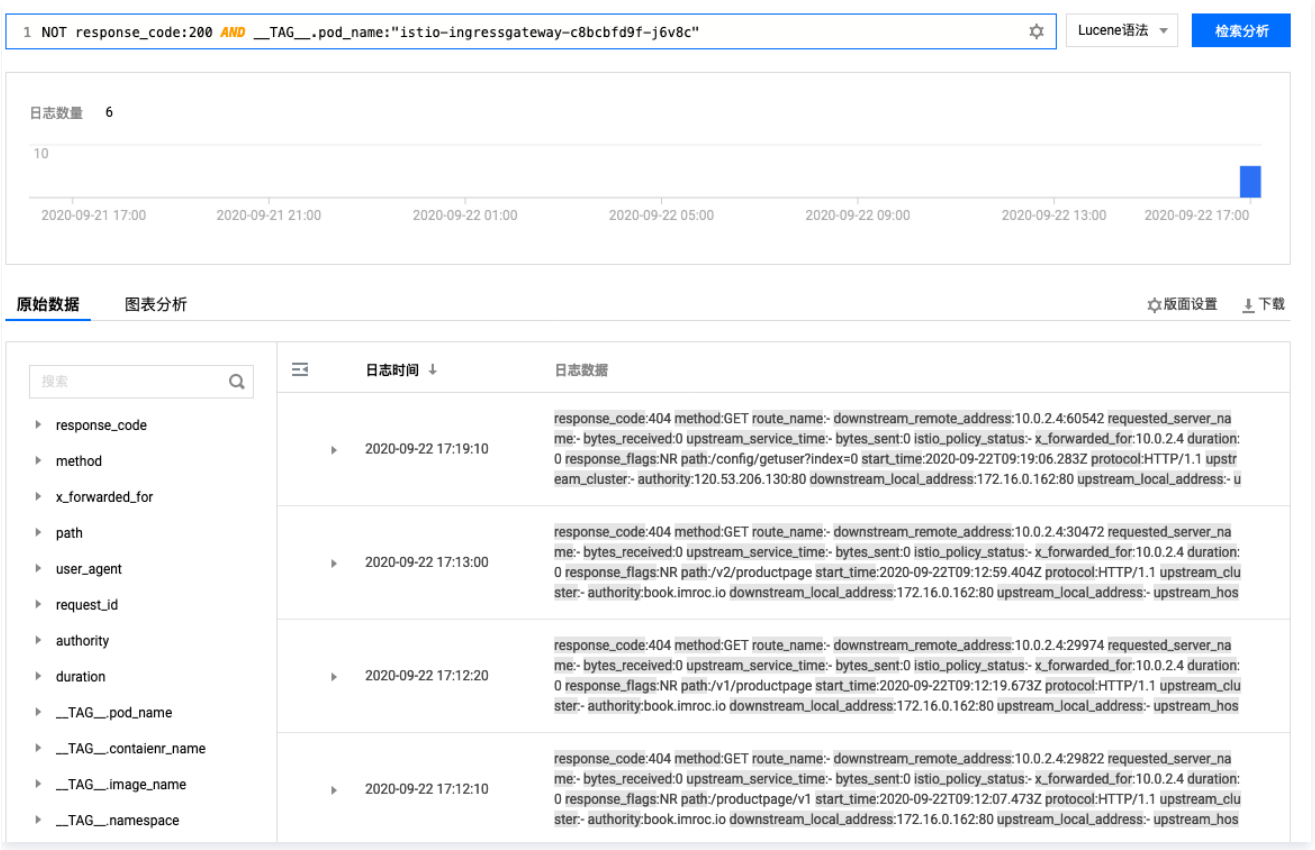
完成日志采集规则配置后，采集器会自动开始采集日志并上报到 CLS。您可在 日志服务控制台 的**检索分析**中查询日志，开启索引后支持 Lucene 语法。有以下几类索引：

- 全文索引。用于模糊搜索，不用指定字段。如下图所示：

- 键值索引。索引结构化处理过的日志内容，可以指定日志字段进行检索。如下图所示：

- 元字段索引。上报日志时额外自动附加的一些字段。例如 pod 名称、namespace 等，方便检索时指定这些字段进行检索。如下图所示：

查询示例如下图所示：



投递日志至 COS 及 Ckafka

CLS 支持将日志投递到对象存储 COS 和消息队列 CKafka，您可在日志主题里进行设置。如下图所示：

可用于以下场景：

- 需对日志数据进行长期归档存储。日志集默认存储7天的日志数据，可以调整时长。数据量越大，成本就越高，通常只保留几天的数据，如果需要将日志存更长时间，可以投递到 COS 进行低成本存储。
- 需要对日志进行进一步处理（例如离线计算），可以投递到 COS 或 Ckafka，由其它程序消费来处理。

参考资料

- 容器服务：日志采集用法指引
- 日志服务：配置时间格式
- 日志服务：投递至 COS
- 日志服务：投递至 Ckafka

NginxIngress 自定义日志

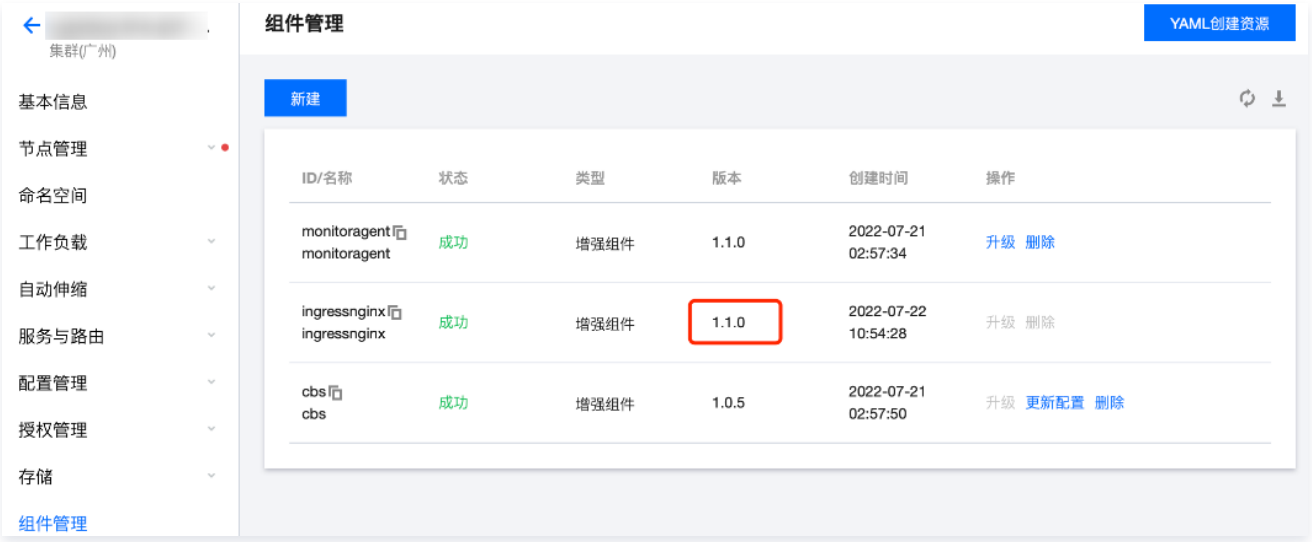
最近更新时间：2025-07-01 14:42:22

容器服务 TKE 通过集成日志服务 CLS，提供了全套完整的产品化能力，实现 NginxIngress 日志采集、消费能力。更多请查看 NginxIngress 日志配置。若默认的日志索引不符合您的日志需求，您可以自定义日志索引，本文向您介绍如何更新 NginxIngress 的日志索引。

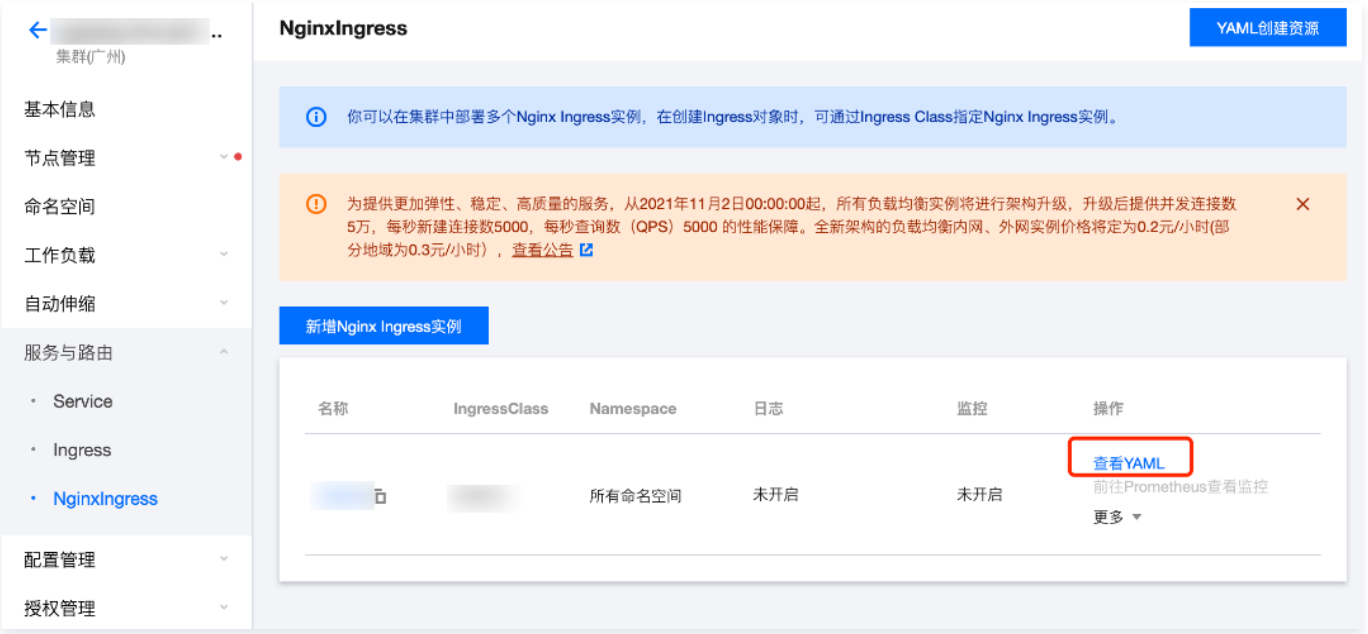
前提条件

1. NginxIngress 为 v1.1.0及以上版本。请登录 容器服务控制台，在**集群详情 > 组件管理**中查看 NginxIngress 的组件版本。

> **注意：**
> 仅 NginxIngress 为 v1.1.0及以上版本才支持该能力，若是 v1.1.0以下版本例如v1.0.0，用户修改日志索引会被组件回滚覆盖。



2. NginxIngress 实例为 v0.49.3及以上版本。请登录 容器服务控制台，在**集群详情 > 服务与路由**中选择 **NginxIngress**，单击实例右侧的**查看YAML**。在 YAML 中，镜像 `ccr.ccs.tencentyun.com/paas/nginx-ingress-controller` 的版本需要大于等于 v0.49.3。



3. 已开启 NginxIngress 日志服务。操作详情见 TKE NginxIngress 采集日志。

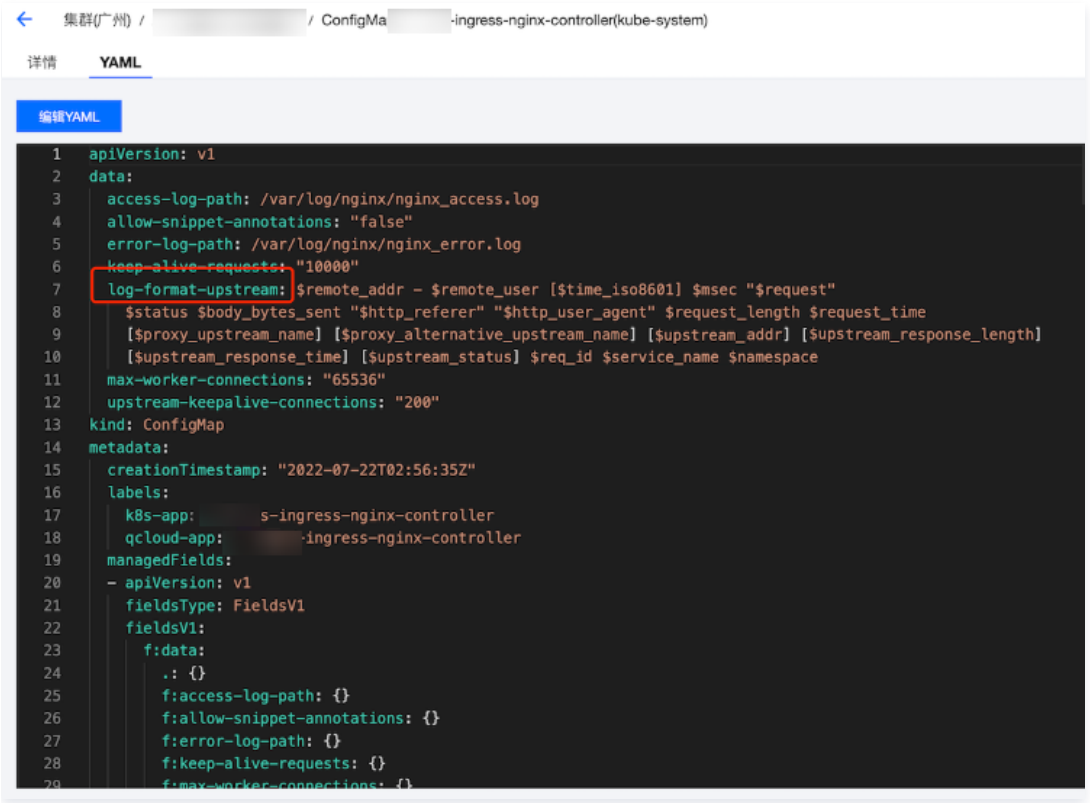
操作步骤

> **注意：**

> 修改日志结构需要了解 Nginx Ingress 的日志流，如日志的输出、日志的采集、日志的索引的配置，其中日志输出和采集缺失或配置出错，都会导致日志修改失败。

步骤1：修改 NginxIngress 实例的日志输出格式

NginxIngress 实例的日志配置在该实例的主配置 ConfigMap 中。ConfigMap 的名称为 `实例名-ingress-nginx-controller`，需要修改的 Key 是 `log-format-upstream`，如下图所示：



示例

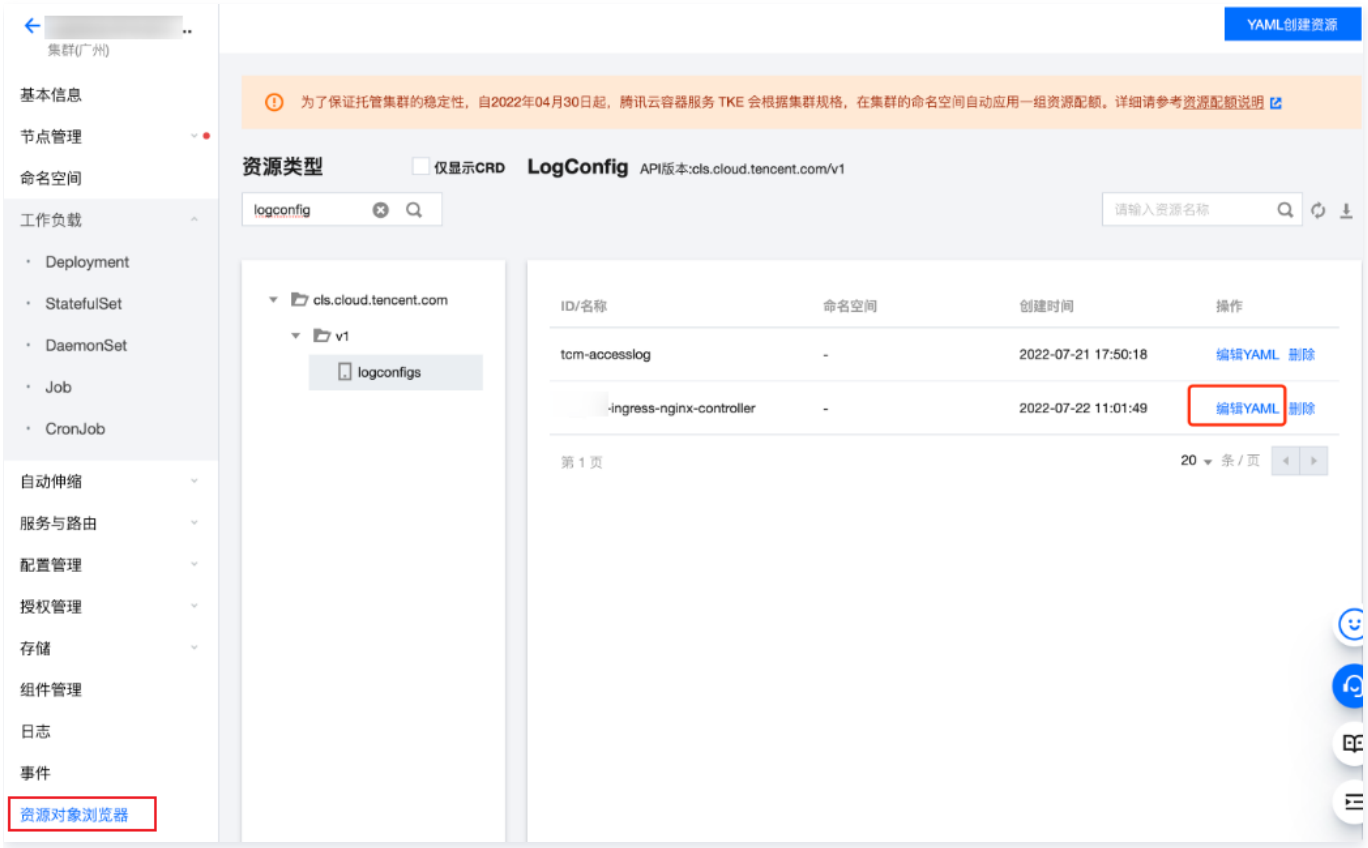
在日志中增加两个连续的字符串：`$namespace` 和 `$service_name`，并放在日志内容的最后，添加位置如下图所示：

```
apiVersion: v1
data:
  access-log-path: /var/log/nginx/nginx_access.log
  allow-snippet-annotations: "false"
  error-log-path: /var/log/nginx/nginx_error.log
  keep-alive-requests: "10000"
  log-format-upstream: $remote_addr - $remote_user [$time_iso8601] $msec "$request"
    $status $body_bytes_sent "$http_referer" "$http_user_agent" $request_length $request_time
    [$proxy_upstream_name] [$proxy_alternative_upstream_name] [$upstream_addr] [$upstream_response_length]
    [$upstream_response_time] [$upstream_status] $req_id $service_name $namespace
  max-worker-connections: "65536"
  upstream-keepalive-connections: "200"
kind: ConfigMap
```

如您需要了解更多 NginxIngress 的日志字段，请参考 文档 。

步骤2：修改集群内日志采集上报 Agent 的格式

集群内日志采集规则在 logconfigs.cls.cloud.tencent.com 型资源对象中。请登录 容器服务控制台，在**集群详情 > 资源对象浏览器**中，您可以找到该资源对象，名称为 `实例名-ingress-nginx-controller`。您可在**编辑YAML**中进行修改。



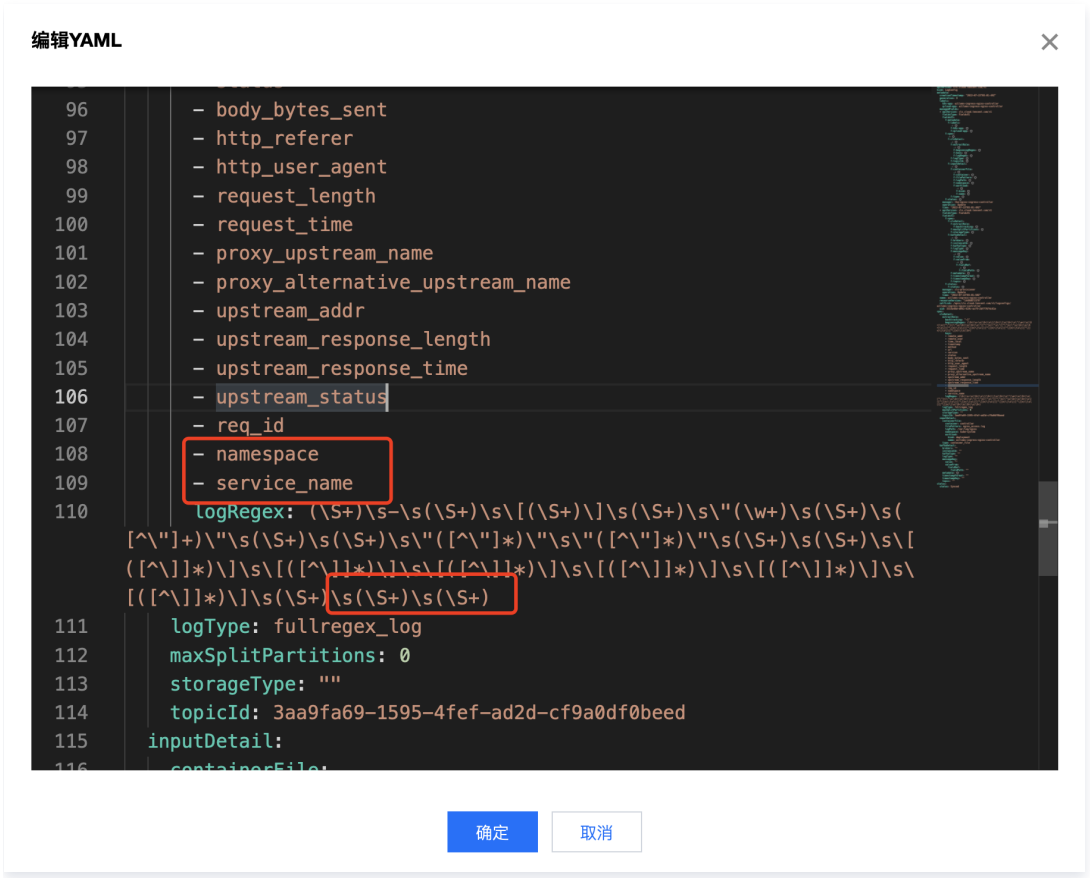
需要修改字段包括：

- beginningRegex：日志开始的正则表达式
- keys：日志的字段
- logRegex：日志结束的正则表达式

正则和 Nginx 的日志行格式匹配。建议在 Nginx 已有日志格式后面追加字段，同时声明在 keys 的末尾。并追加该字段的正则解析到 beginningRegex、logRegex 的末尾。

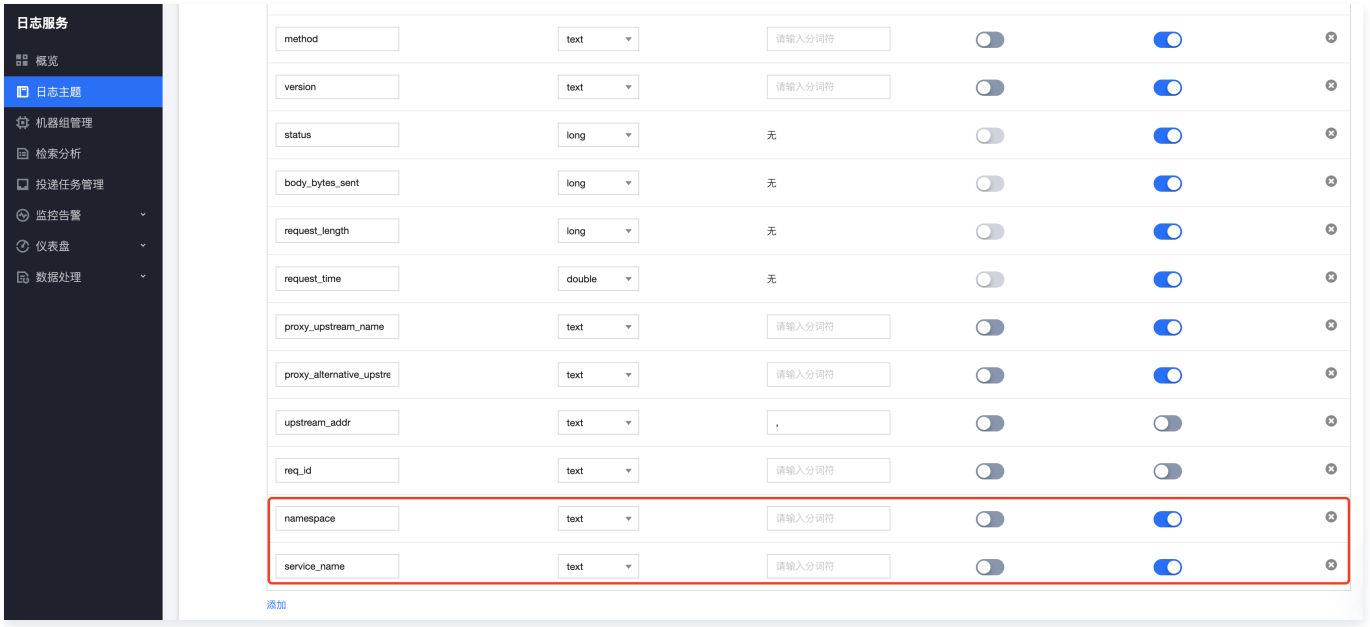
示例

在 keys 后面追加 步骤1 中的两个字段，然后分别在 beginningRegex、logRegex 的末尾增加正则表达式字符串。如下图所示：



（可选）步骤3：修改 CLS 的日志索引格式

如果需要检索该字段的能力，则需要在对应日志主题中，添加新字段的索引。您可以在日志服务控制台操作，操作完成之后所有采集到的日志都可以通过索引进行检索。操作详情见 配置索引 。



恢复初始设置

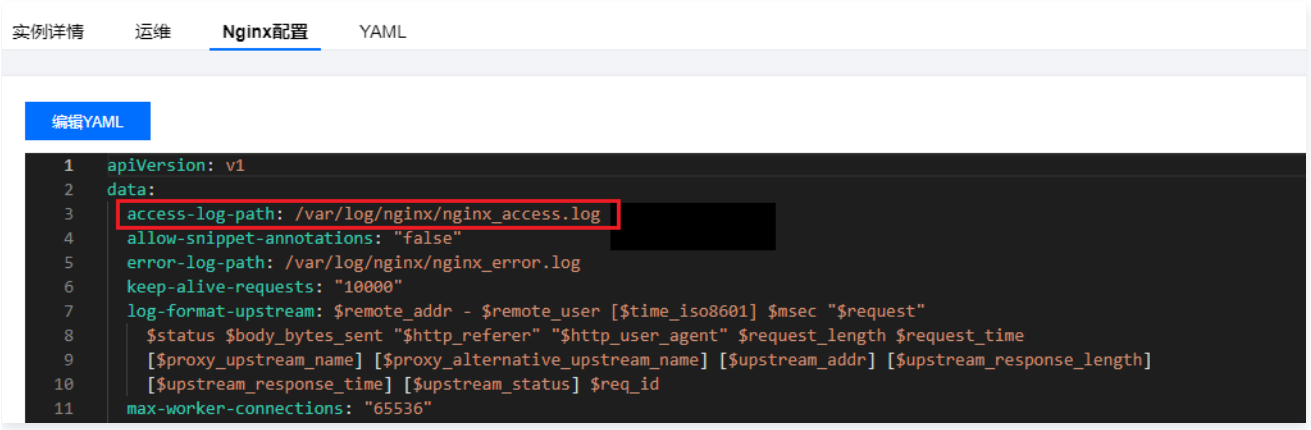
因为修改日志规则步骤较复杂，且涉及到正则表达式，操作过程中有任何一个步骤错误，都可能导致日志采集失败。若日志采集报错，此时建议您恢复原始的日志采集能力，您需要先关闭日志采集功能，然后再次 开启日志采集 。

TKE 使用 logrotate 切割 nginx-ingress 访问日志

最近更新时间：2023-11-01 14:34:52

使用场景

nginx-ingress 是使用 Nginx 作为反向代理和负载平衡器的 Kubernetes 的 Ingress 控制器，容器服务 TKE 提供了产品化的能力，可以直接在集群内安装和使用 Nginx-ingress。有关安装 Nginx-ingress 的详细信息，请参见 安装 Nginx-ingress 实例 。
使用 nginx-ingress，通常需要查看访问日志以定位问题。nginx-ingress 组件支持将日志直接采集到腾讯云的 CLS。nginx-ingress 实例默认配置的访问日志写入容器的 /var/log/nginx/nginx_access.log 文件中，然后配置日志采集规则，将该日志文件采集到 CLS。路径如下图所示：



nginx-ingress 默认不带有日志切割功能。如果运行时间长，nginx_access.log 文件会变得非常大，占用大量磁盘空间。由于日志直接写入容器内，最终会落在节点的容器存储目录中，占用 node 节点磁盘空间。
如果容器存储目录没有单独挂盘，通常会直接使用系统盘。随着日志文件的增大，系统盘的可用磁盘空间会不断减少，此时可能会导致节点的 kubelet 驱逐。
为了解决这个问题，需要对 nginx-ingress controller 容器的 /var/log/nginx/nginx_access.log 日志文件进行轮转切割。由于日志是容器的文件，因此可以部署一个 logrotate 的 sidecar 容器来轮转切割 nginx 的访问日志，以确保日志不会不断增加并变得非常大。

操作步骤

步骤1：拉取 logrotate 容器镜像

下面介绍如何在 nginx-ingress 中配置 logrotate 容器来轮转切割日志。本文中将使用 realz/logrotate 镜像来进行日志切割。

1. 执行以下命令，拉取 logrotate 镜像：

```
docker pull realz/logrotate:latest
```

   logrotate 的具体介绍请参见 官方 GitHub 文档 。

2. 您需要在容器中配置以下环境变量：

```
CRON_EXPR="* * * * *"
LOGROTATE_LOGFILES=/var/lib/docker/containers/*/*.log
LOGROTATE_FILESIZE=10M
LOGROTATE_FILENUM=5
```

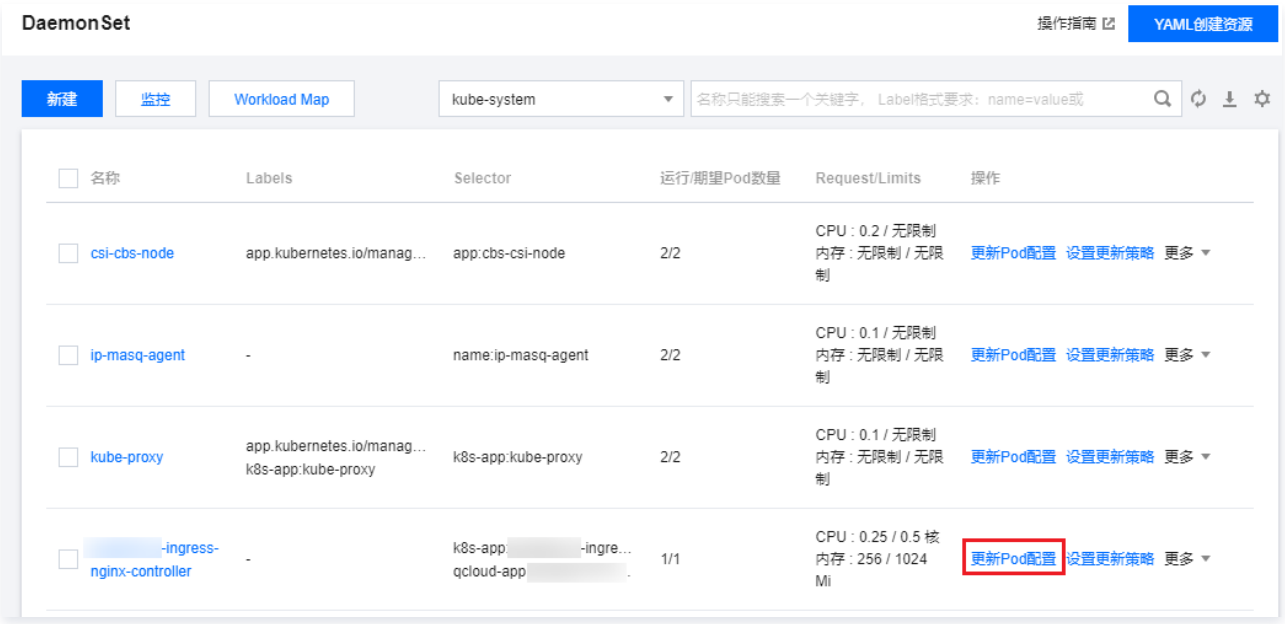
   环境变量说明：
   - CRON_EXPR：日志定时切割时间。
   - LOGROTATE_LOGFILES：需要切割的日志。
   - LOGROTATE_FILESIZE：日志文件达到某个大小时开始切割。
   - LOGROTATE_FILENUM：最多保留几个日志文件。

3. 启动容器。

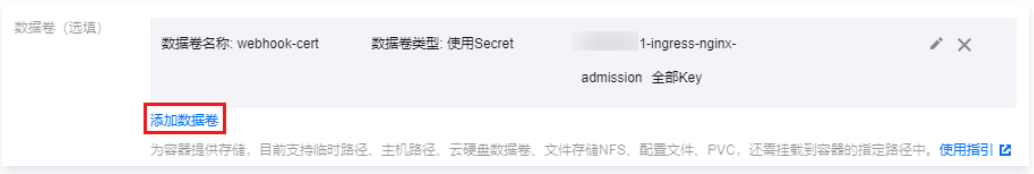
步骤2：nginx-ingress 工作负载配置 sidecar 容器

使用 sidecar 容器的方案是通过 emptydir 共享 /var/log/nginx/ 目录，以便 logrotate 容器可以直接访问 nginx_access.log。下面介绍如何在控制台中进行配置。

1. 登录 容器服务控制台，选择左侧导航栏中的**集群**。
2. 在**集群管理**页面，选择集群，进入集群详情页。
3. 在**工作负载 > DaemonSet** 中，命名空间选择 **kube-system**，找到 {xxx}-ingress-nginx-controller，单击右侧的**更新 Pod 配置**。如下图所示：

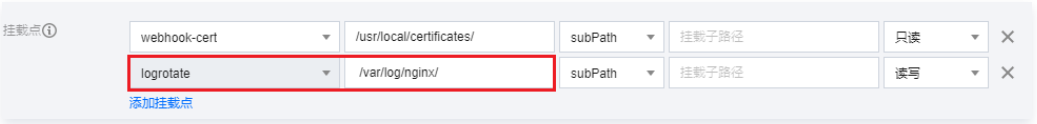


4. 在**更新 Pod 配置**页面中，添加一个数据卷，如下图所示：



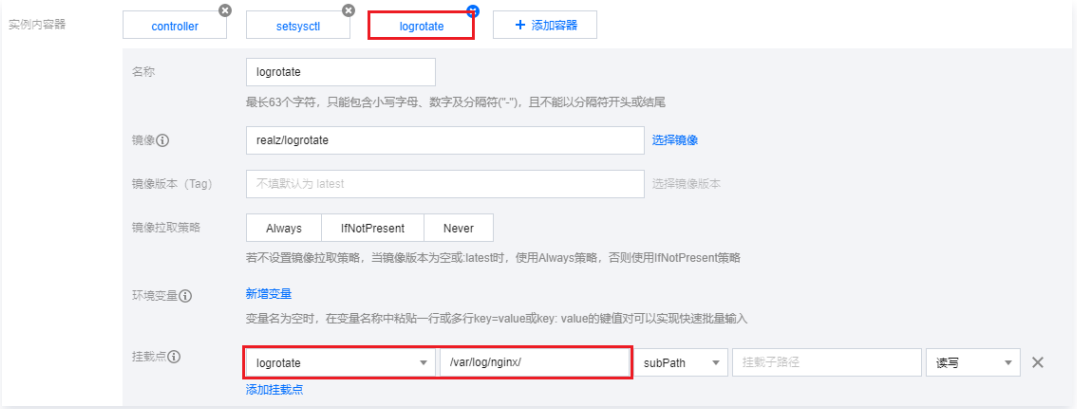
在**新增数据卷**中，数据卷类型选择**使用临时目录**，数据卷名称填写 `logrotate`。

5. 在实例内容器中，选择 controller，并添加挂载点。如下图所示：



在**添加挂载点**中，名称填写 `logrotate`，路径填写 `/var/log/nginx/`。

6. 添加一个 logrotate 容器。如下图所示：



镜像选择 `realz/logrotate`，挂载点选择 `logrotate` 数据卷，路径填写 `/var/log/nginx/`。在环境变量中，参考以下配置进行填写：

```
CRON_EXPR="* * * * *"
```

```
LOGROTATE_LOGFILES=/var/log/nginx/*.log
LOGROTATE_FILESIZE=1M
LOGROTATE_FILENUM=5
```

如下图所示：

7. **配置完成后，单击更新 Pod 配置。完整的 yaml 示例如下：**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: nginx-intranet-ingress-nginx-controller
    qcloud-app: nginx-intranet-ingress-nginx-controller
  name: nginx-intranet-ingress-nginx-controller
  namespace: kube-system
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      k8s-app: nginx-intranet-ingress-nginx-controller
      qcloud-app: nginx-intranet-ingress-nginx-controller
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        k8s-app: nginx-intranet-ingress-nginx-controller
        qcloud-app: nginx-intranet-ingress-nginx-controller
      name: nginx-intranet-ingress-nginx-controller
      namespace: kube-system
    spec:
      containers:
      - args:
        - /nginx-ingress-controller
        - --election-id=ingress-controller-leader
        - --ingress-class=nginx-intranet
        - --configmap=kube-system/nginx-intranet-ingress-nginx-controller
        - --publish-service=kube-system/nginx-intranet-ingress-nginx-controller
        - --validating-webhook=:8443
        - --validating-webhook-certificate=/usr/local/certificates/cert
        - --validating-webhook-key=/usr/local/certificates/key
        - --tcp-services-configmap=kube-system/nginx-intranet-ingress-nginx-tcp
        - --udp-services-configmap=kube-system/nginx-intranet-ingress-nginx-udp
        env:
        - name: POD_NAME
```

```
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.namespace
    image: ccr.ccs.tencentyun.com/paas/nginx-ingress-controller:v0.41.0
    imagePullPolicy: Always
    lifecycle:
      preStop:
        exec:
          command:
          - /wait-shutdown
    livenessProbe:
      failureThreshold: 3
      httpGet:
        path: /healthz
        port: 10254
        scheme: HTTP
      initialDelaySeconds: 10
      periodSeconds: 10
      successThreshold: 1
      timeoutSeconds: 1
    name: controller
    ports:
    - containerPort: 80
      name: http
      protocol: TCP
    - containerPort: 443
      name: https
      protocol: TCP
    - containerPort: 10254
      name: metrics
      protocol: TCP
    - containerPort: 8443
      name: webhook
      protocol: TCP
    readinessProbe:
      failureThreshold: 3
      httpGet:
        path: /healthz
        port: 10254
        scheme: HTTP
      initialDelaySeconds: 10
      periodSeconds: 10
      successThreshold: 1
      timeoutSeconds: 1
    resources: {}
    securityContext:
      allowPrivilegeEscalation: true
      capabilities:
        add:
        - NET_BIND_SERVICE
        drop:
        - ALL
      runAsUser: 101
    terminationMessagePath: /dev/termination-log
```

```
      terminationMessagePolicy: File
      volumeMounts:
      - mountPath: /usr/local/certificates/
        name: webhook-cert
        readOnly: true
      - mountPath: /var/log/nginx
        name: logrotate
    - env:
      - name: LOGROTATE_LOGFILES
        value: /var/log/nginx/*.log
      - name: CRON_EXPR
        value: '* * * * *'
      - name: LOGROTATE_FILESIZE
        value: 1M
      - name: LOGROTATE_FILENUM
        value: "5"
      image: realz/logrotate
      imagePullPolicy: Always
      name: logrotate
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
        requests:
          cpu: 250m
          memory: 256Mi
      securityContext:
        privileged: false
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
      volumeMounts:
      - mountPath: /var/log/nginx
        name: logrotate
    dnsPolicy: ClusterFirst
    initContainers:
    - command:
      - sh
      - -c
      - |-
        sysctl -w net.core.somaxconn=65535
        sysctl -w net.ipv4.ip_local_port_range="1024 65535"
        sysctl -w net.ipv4.tcp_tw_reuse=1
        sysctl -w fs.file-max=1048576
      image: ccr.ccs.tencentyun.com/tkeimages/busybox:latest
      imagePullPolicy: Always
      name: setsysctl
      resources: {}
      securityContext:
        privileged: true
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    serviceAccount: nginx-intranet-ingress-nginx
    serviceAccountName: nginx-intranet-ingress-nginx
    terminationGracePeriodSeconds: 300
    volumes:
    - name: webhook-cert
      secret:
```

```
          defaultMode: 420
          secretName: nginx-intranet-ingress-nginx-admission
      - emptyDir: {}
        name: logrotate
```

步骤3：验证配置是否生效

当 nginx-ingress controller pod 运行正常后，您可以通过 ingress 域名访问后端服务，从而生成访问日志。

```
[root@VM-0-4-centos sys]# for i in {1..10000}; do curl http://ng[redacted]cn; done
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
```

隔几分钟后，登录容器并查看日志，可以发现日志已被切割成多个文件。

```
bash-5.0$ ls -al
total 6164
drwxrwxrwx    2 root     root          4096 Dec 14 19:50 .
drwxr-xr-x    1 www-data www-data      4096 Oct 28  2020 ..
-rw-r--r--    1 root     root           645 Dec 14 15:52 .nginx_access.log.metadata
-rw-r--r--    1 www-data www-data        59 Dec 14 15:52 error.log
-rw-r--r--    1 www-data www-data   1001023 Dec 15 15:49 nginx_access.log
-rw-r--r--    1 www-data www-data   1059033 Dec 14 19:50 nginx_access.log.1
-rw-r--r--    1 www-data www-data   1054872 Dec 14 19:49 nginx_access.log.2
-rw-r--r--    1 www-data www-data   1057073 Dec 14 19:48 nginx_access.log.3
-rw-r--r--    1 www-data www-data   1054632 Dec 14 19:47 nginx_access.log.4
-rw-r--r--    1 www-data www-data   1054876 Dec 14 19:46 nginx_access.log.5
-rw-r--r--    1 www-data www-data         0 Dec 14 15:52 nginx_error.log
bash-5.0$
```

通过以上配置，您已成功地使用 logrotate 切割了 nginx-ingress controller 的访问日志，并且这样不会影响 CLS 采集日志。

使用 CLS 告警异常资源

最近更新时间：2025-06-20 16:17:11

使用场景

Kubernetes 使用事件（Event）反馈集群中资源对象的状态，它通常表示系统中的一些状态变化。例如在安装或修改工作负载时，您可以通过事件信息判断当前资源对象是否存在异常，以及查看导致异常的原因。事件的保留时间有限，在 TKE 集群中事件可保留1小时。
如果事件信息包含异常，则需要集群管理员及时关注。TKE 支持为您的所有集群配置事件持久化功能，开启该功能后，TKE 会将您的集群事件实时导出至配置的存储端。更多请参考 事件日志 。
Service/Ingress 作为 Kubernetes 中接入层的资源对象，其质量事关业务服务稳定性，因此，对 Service/Ingress 异常事件的监控告警成为了常见诉求。为此，TKE 也定义了常见的 Service/Ingress 异常事件错误码信息、异常原因和解决办法，更多请参考 Service&Ingress 常见报错和处理 。本文提供集群里 Service/Ingress 异常事件的告警实践。

操作步骤

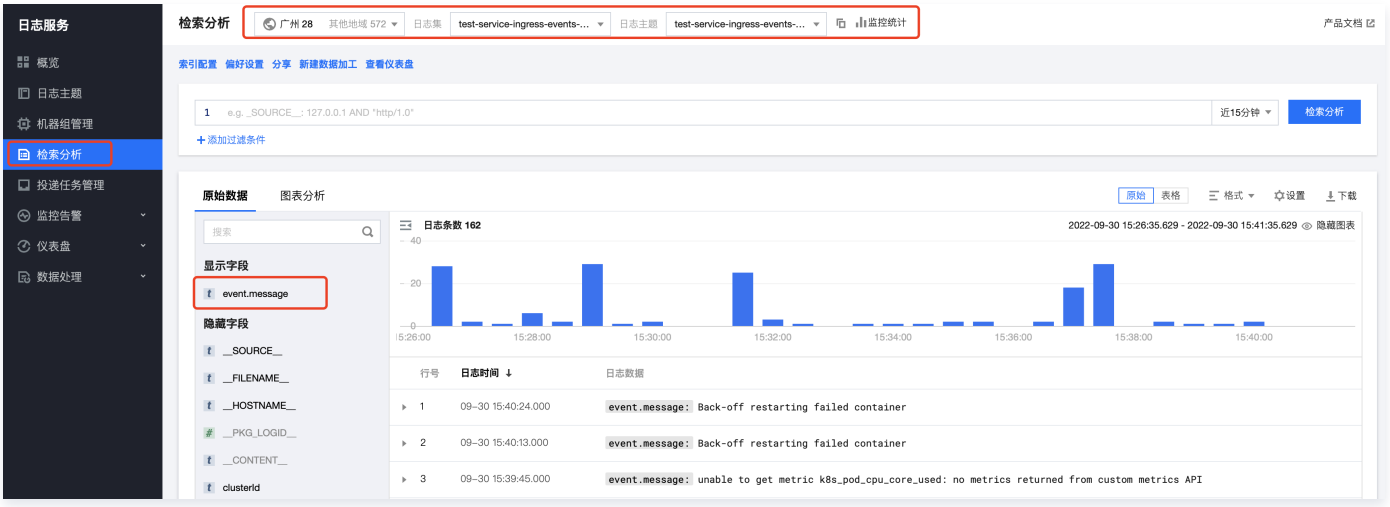
步骤1：打开集群的事件采集

1. 登录 容器服务控制台 。
2. 在左侧导航栏中，选择**运维功能管理**。
3. 在**功能管理**页面上方选择地域和集群类型，单击需要开启事件存储的集群右侧的**设置**。
4. 在**设置功能**页面，单击事件存储右侧的**编辑**。勾选**开启事件存储**，并配置日志集和日志主题。操作详情见 开启事件日志 。

> **注意：**
> 若您在同一个地域有多个 Kubernetes 集群，建议您可以打开多个集群的事件存储功能，并选择相同的日志主题和日志集。

步骤2：确定事件是否采集

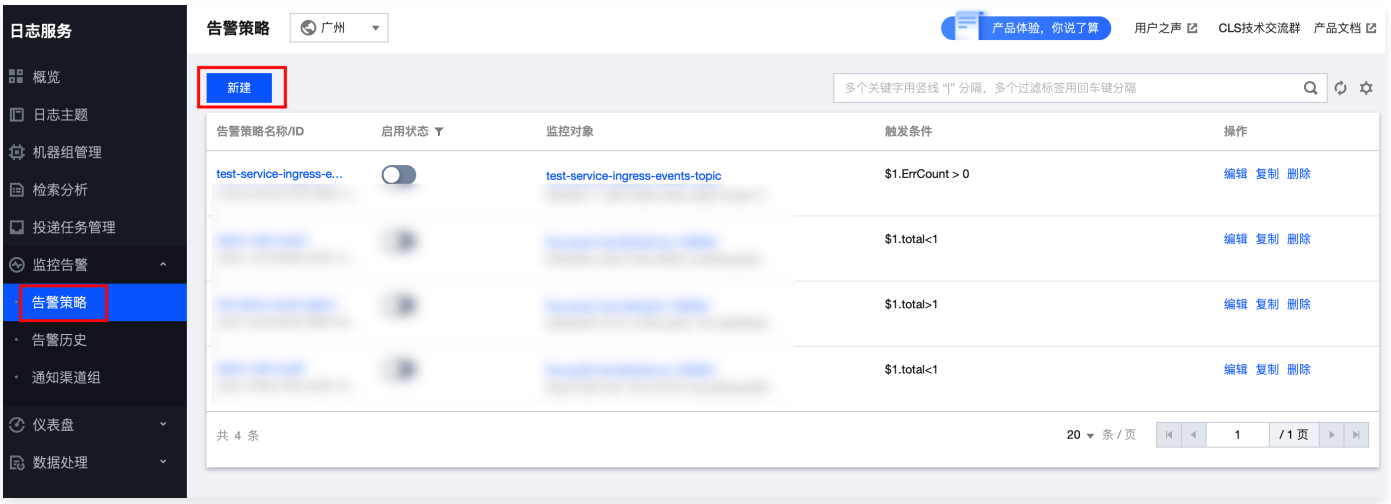
1. 登录 日志服务控制台 ，进入**检索分析**页。
2. 在**检索分析**页，选择地域、已开启事件采集的集群日志集、日志主题。
3. 在“原始数据”中，查找字段 `event.message` ，该字段为集群中资源对象产生的事件信息。如下图所示：



步骤3：新建告警策略

以告警 Ingress 的事件为例，Service 类似。
1. 登录 日志服务控制台 。选择**监控告警 > 告警策略**。

2. 在**告警策略**页，单击**新建**。如下图所示：

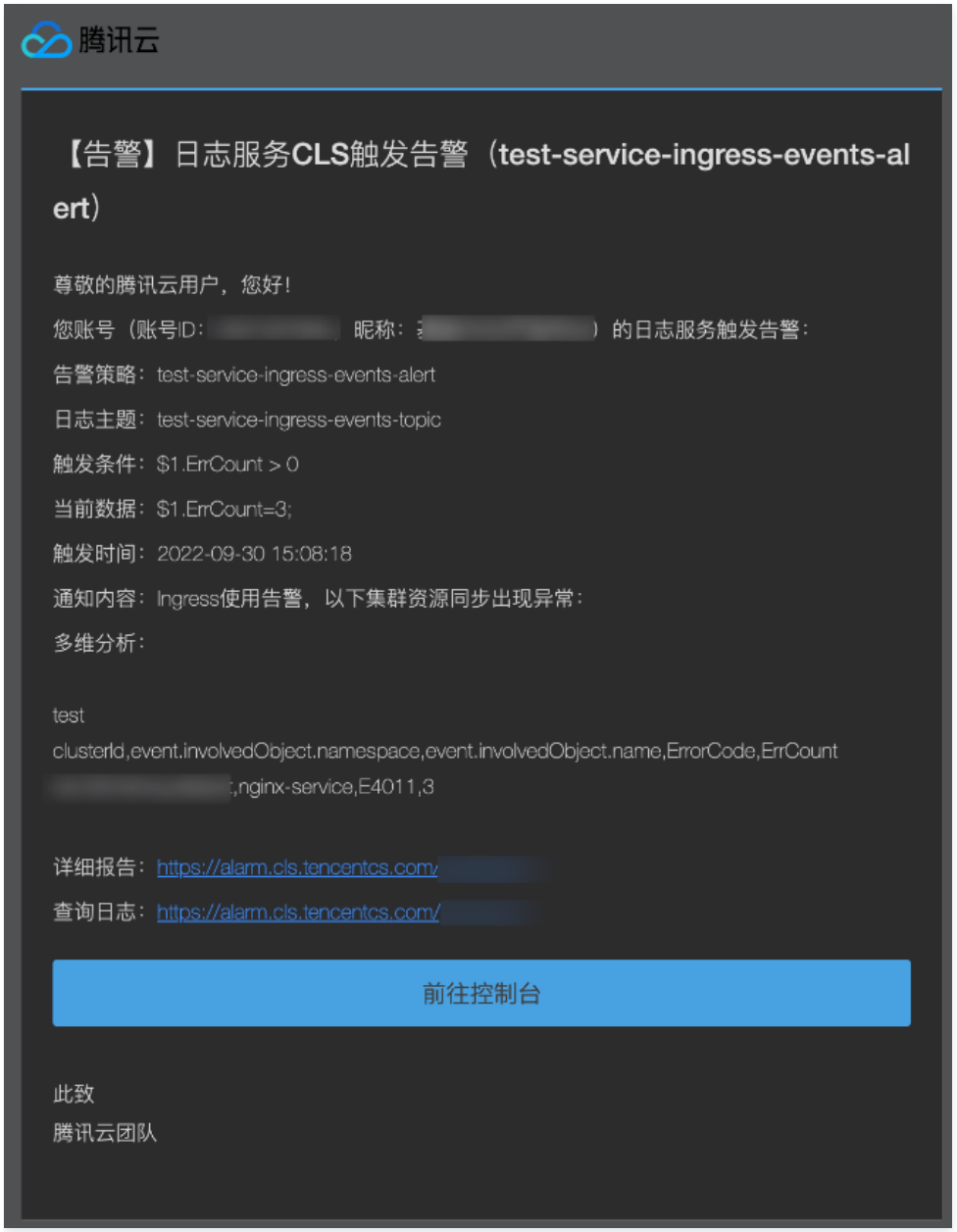


3. 在**新建告警策略**页，参考以下主要信息进行设置：
   - **日志主题**：选择您在 步骤1 中创建的主题。
   - **执行语句**：添加执行语句：
     ```
     (event.message:"Ingress Sync ClientError." OR event.message:"Ingress Sync DependencyError." OR event.message:"IngressError. ErrorCode:") | SELECT count(*) as ErrCount
     ```
     > **说明：**
     > 表示获取所有的 Ingress 的事件信息。
   - **触发条件**：添加触发条件 `$1.ErrCount > 0` 。
     > **说明：**
     > 表示一有事件信息就触发告警。
   - **多维分析**：选择**自定义检索分析**。
   - **名称**：您可以自定义名称。
   - **检索分析语句**：添加检索分析语句：
     ```
     (event.message:"Ingress Sync ClientError." OR event.message:"Ingress Sync DependencyError." OR event.message:"IngressError. ErrorCode:") | SELECT clusterId, event.involvedObject.namespace, event.involvedObject.name, split(split(event.message, 'ErrorCode: ')[2], ' ')[1] as ErrorCode, count(*) as ErrCount group by (clusterId, event.involvedObject.namespace, event.involvedObject.name, ErrorCode)
     ```
   - **通知内容**：添加通知内容 “Ingress 使用告警，以下集群资源同步出现异常：”

   完整参数配置方式请参考 配置告警策略 。

步骤4：查看告警

确保 步骤2 中有新的事件产生，且 步骤2 中告警策略的执行周期、告警通知频率合适（例如测试时可以设置为1分钟一次），就可以查看告警通知渠道中的告警内容了。本文示例设置为通过邮件进行告警，因此可参考邮件的告警内容，如下图所示：


腾讯云
【告警】日志服务CLS触发告警（test-service-ingress-events-alert）
尊敬的腾讯云用户，您好！
您账号（账号ID：[redacted] 昵称：[redacted]）的日志服务触发告警：
告警策略：test-service-ingress-events-alert
日志主题：test-service-ingress-events-topic
触发条件：$1.ErrCount > 0
当前数据：$1.ErrCount=3;
触发时间：2022-09-30 15:08:18
通知内容：Ingress使用告警，以下集群资源同步出现异常：
多维分析：
test
clusterId,event.involvedObject.namespace,event.involvedObject.name,ErrorCode,ErrCount
[redacted],nginx-service,E4011,3
详细报告：https://alarm.cls.tencentcs.com/[redacted]
查询日志：https://alarm.cls.tencentcs.com/[redacted]
前往控制台
此致
腾讯云团队

监控
JVM 接入

最近更新时间：2025-02-28 15:51:43

操作场景

在使用 Java 作为开发语言时，需要监控 JVM 的性能。Prometheus 监控服务通过采集应用暴露出来的 JVM 监控数据，并提供了开箱即用的 Grafana 监控大盘。
本文介绍了通过 client_java 或 jmx_exporter 两种方式输出 JVM 指标，用 Prometheus 监控服务监控其状态。

> **说明：**
> 若已使用 Spring Boot 作为开发框架，请参见 Spring Boot 接入 。

前提条件

- 创建腾讯云容器服务 托管版集群 。
- 使用 容器镜像服务 管理应用镜像。

指标埋点

client_java

client_java 是 Prometheus 官方提供的采集 SDK，提供简洁的 API 自定义指标埋点，还有开箱即用的 JVM 指标，是开发者接入 Prometheus 监控服务的首选方式。

修改应用的依赖及配置

1. 修改 pom 依赖。在 `pom.xml` 文件中添加相关的 Maven 依赖项，1.x 版本做了重构和老版本已经不兼容，优先选择最新版本，示例如下：

```
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>prometheus-metrics-core</artifactId>
    <version>1.3.3</version>
</dependency>
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>prometheus-metrics-instrumentation-jvm</artifactId>
    <version>1.3.3</version>
</dependency>
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>prometheus-metrics-exporter-httpserver</artifactId>
    <version>1.3.3</version>
</dependency>
```

2. 修改代码。初始化并注册 Metrics，示例如下：

```
package org.example;

import io.prometheus.metrics.core.metrics.Counter;
import io.prometheus.metrics.exporter.httpserver.HTTPServer;
import io.prometheus.metrics.instrumentation.jvm.JvmMetrics;

import java.io.IOException;

public class Main {
```

```
    public static void main(String[] args) throws InterruptedException, IOException {
        JvmMetrics.builder().register(); // 初始化并注册 JVM 指标
         // 初始化并注册业务自定义指标
        Counter counter = Counter.builder()
                .name("my_count_total")
                .help("example counter")
                .labelNames("status")
                .register();
        counter.labelValues("ok").inc();
        counter.labelValues("ok").inc();
        counter.labelValues("error").inc();
        // 启动 metrics http server
        HTTPServer server = HTTPServer.builder()
                .port(9400)
                .buildAndStart();
        System.out.println("HTTPServer listening on port http://localhost:" + server.getPort() +
"/metrics")

        Thread.currentThread().join();
    }

}
```

3. 本地验证。本地启动之后，可以通过 `http://localhost:9400/metrics` 访问到 Prometheus 协议的指标数据。

```
http://localhost:9400/metrics
```

将应用发布到腾讯云容器服务上

1. 本地配置 Docker 镜像环境。如果本地之前未配置过 Docker 镜像环境，可以参见容器镜像服务 Docker 镜像操作快速入门 进行配置。若已配置请执行下一步。
2. 打包及上传镜像。

   2.1 在项目根目录下添加 `Dockerfile`，请根据实际项目进行修改。示例如下：

```
FROM openjdk:8-jdk
WORKDIR /java-demo
ADD target/java-demo-*.jar /java-demo/java-demo.jar
CMD ["java","-jar","java-demo.jar"]
```

   2.2 打包镜像，在项目根目录下运行如下命令，需要替换对应的 [namespace]、[ImageName] 和 [镜像版本号]。

```
mvn clean package
docker build . -t ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
docker push ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

jmx_exporter

jmx_exporter 是 Prometheus 官方 exporter，把 JVM 原生 MBeans 数据转换为 Prometheus 格式的指标并通过 HTTP 服务暴露出来。jmx_exporter 以 Java Agent 无代码侵入方式运行，但是只能暴露已经注册到 MBeans 上的指标，无法做业务自定义埋点。对于绝大部分的开发者，client_java 是最常用的接入手段。

准备 jmx_exporter 资源

1. 下载 jar 包。在项目 发布页 下载最新版本的 Java Agent Jar 包，这里以1.0.1为例。

```
wget
https://repo.maven.apache.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/1.0.1/jmx_prometheus_j
```

```
avaagent-1.0.1.jar
```

2. 准备配置文件。此处使用最少可用配置，各个配置项的详细说明请参见 文档 。

```
rules:
- pattern: ".*"
```

3. 本地验证。jmx_exporter 启动格式是 `-javaagent:<jmx_exporter jar 路径>=< metric 暴露端口>:< jmx_exporter 配置文件路径>` 。

```
java -javaagent:./jmx_prometheus_javaagent-1.0.1.jar=9400:./jmx.yml -jar demo.jar
```

4. 正常启动后，访问 `http://localhost:9400/metrics` 会返回 jvm 开头的指标。

```
 http://localhost:9400/metrics
```

将应用发布到腾讯云容器服务上

1. 本地配置 Docker 镜像环境。如果本地之前未配置过 Docker 镜像环境，请参见容器镜像服务 Docker 镜像操作快速入门 进行配置。若已配置请执行下一步。
2. 打包及上传镜像。

   2.1 在项目根目录下添加 `Dockerfile` ，请根据实际项目进行修改。示例如下：

```
FROM openjdk:8-jdk
WORKDIR /java-demo
# 添加下载的 jmx_exporter jar 包
ADD ./jmx_prometheus_javaagent-1.0.1.jar /java-demo/jmx_prometheus_javaagent-1.0.1.jar
# 添加准备的 jmx_exporter 配置文件
ADD ./jmx.yml  /java-demo/jmx.yml
ADD ./target/application.jar /java-demo/java-demo.jar
# 通过 java agent 注入 jmx_exporter
CMD ["java", "-javaagent:/java-demo/jmx_prometheus_javaagent-1.0.1.jar=9400:/java-demo/jmx.yml", "-
jar", "java-demo.jar"]
```

   2.2 打包镜像，在项目根目录下运行如下命令，需要替换对应的 [namespace]、[ImageName] 和 [镜像版本号]。

```
docker build . -t ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
docker push ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

应用部署

1. 登录 容器服务控制台 ，在左侧菜单栏中集群页面，选择需要部署的容器集群。
2. 通过**工作负载 > Deployment** 进入 **Deployment** 管理页面，选择对应的**命名空间**进行部署服务，通过 YAML 来创建对应的 Deployment，YAML 配置如下。

> **说明：**
> 如需通过控制台创建，请参见 Spring Boot 接入 。

```
apiVersion: apps/v1
kind: Deployment
metadata:
    labels:
      k8s-app: java-demo
    name: java-demo
    namespace: java-demo
spec:
```

```
    replicas: 1
    selector:
      matchLabels:
        k8s-app: java-demo
    template:
      metadata:
        labels:
          k8s-app: java-demo
      spec:
       containers:
       - image: ccr.ccs.tencentyun.com/prometheus-demo/java-demo
         imagePullPolicy: Always
         name: java-demo
         ports:
          - containerPort: 9400
            name: metric-port
```

添加采集任务

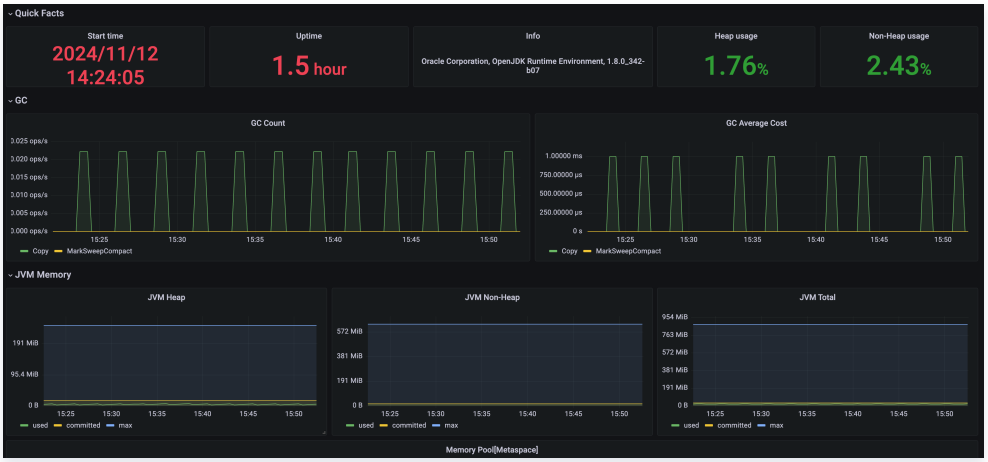
1. 登录 腾讯云可观测平台 。
2. 在左侧菜单栏中选择 **Prometheus 监控**，选择对应 Prometheus 实例进入管理页面。
3. 选择**数据采集 > 集成容器服务**，进入到容器服务集成管理页面。
4. 选择**数据采集配置 > 新增自定义监控 > yaml 编辑 > PodMonitors** 新增抓取任务，YAML 配置示例如下：

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: java-demo
  namespace: java-demo
spec:
  namespaceSelector:
    matchNames:
    - java-demo
  podMetricsEndpoints:
  - interval: 15s
    path: /metrics
    port: metric-port
    relabelings:
    - action: replace
      sourceLabels:
      - __meta_kubernetes_pod_label_k8s_app
      targetLabel: application
  selector:
    matchLabels:
      k8s-app: java-demo
```

查看监控

1. 进入对应 Prometheus 实例，在**数据采集 > 集成中心**中找到 **JVM 监控**，安装对应的 Grafana Dashboard 即可开启 JVM 监控大盘。
2. 打开 Prometheus 实例对应的 Grafana 地址，在 `Dashboards` 下查看应用相关的监控大屏。
   - **应用 JVM**：从应用角度出发，查看该应用下所有实例是否存在异常，当发现某个实例有异常时，可以下钻到对应的实例监控。

   - **实例 JVM**：单实例 JVM 详细的监控数据。


Quick Facts
Start time
2024/11/12 14:24:05
Uptime
1.5 hour
Info
Oracle Corporation, OpenJDK Runtime Environment, 1.8.0_342-b07
Heap usage
1.76%
Non-Heap usage
2.43%
GC
GC Count
GC Average Cost
JVM Memory
JVM Heap
JVM Non-Heap
JVM Total
Memory Pool[Metaspace]

MySQL Exporter 接入

最近更新时间：2025-02-21 10:23:23

操作场景

MySQL Exporter 是社区专门为采集 MySQL/MariaDB 数据库监控指标而设计开发，通过 Exporter 上报核心的数据库指标，用于异常报警和监控大盘展示，腾讯云可观测平台 Prometheus 提供了与 MySQL Exporter 集成及开箱即用的 Grafana 监控大盘。
目前，Exporter 支持高于5.6版本的 MySQL 和高于10.1版本的 MariaDB。在 MySQL/MariaDB 低于5.6版本时，部分监控指标可能无法被采集。

> **说明：**
> 如果需要监控的 MySQL 是腾讯云 云数据库 MySQL，推荐使用集成中心 云监控集成，支持一键采集云产品指标。

接入方式

方式一：一键安装（推荐）

操作步骤

1. 登录 Prometheus 监控服务控制台。
2. 在实例列表中，选择并进入对应的 Prometheus 实例。
3. 在实例详情页，选择**数据采集 > 集成中心**。
4. 在集成中心找到并单击 **MySQL**，即会弹出一个安装窗口，在安装页面填写指标采集名称和地址等信息，并单击**保存**即可。

配置说明

参数	说明
名称	集成名称，命名规范如下： • 名称具有唯一性。

	• 名称需要符合下面的正则：'^[a-z0-9]([-a-z0-9]*[a-z0-9])?(\.[a-z0-9]([-a-z0-9]*[a-z0-9])?)*$'。
用户名	MySQL 的用户名称。
密码	MySQL 的密码。
地址	MySQL 的连接地址。
标签	给指标添加自定义 Label。
Exporter 配置	参数描述可能存在偏差，具体可参考 官方文档 。 • auto_increment.columns：从 information_schema 采集 auto_increment 列和最大值。 • binlog_size：采集所有已注册的 binlog 文件的当前大小。 • engine_innodb_status：从 SHOW ENGINE INNODB STATUS 采集。 • engine_tokudb_status：从 SHOW ENGINE TOKUDB STATUS 采集。 • global_status：从 SHOW GLOBAL STATUS 采集，默认为 true。 • global_variables：从 SHOW GLOBAL VARIABLES 采集，默认为 true。 • info_schema.clientstats：如果使用 userstat=1 运行，则设置为 true 以采集客户端统计信息。 • info_schema.innodb_metrics：从 information_schema.innodb_metrics 采集指标。 • info_schema.innodb_tablespaces：从 information_schema.innodb_sys_tablespaces 采集指标。 • info_schema.innodb_cmp：从 information_schema.innodb_cmp 采集 InnoDB 压缩表指标，默认为 true。 • info_schema.innodb_cmpmem：从 information_schema.innodb_cmpmem 采集 InnoDB 缓冲池压缩指标(默认为 true)。 • info_schema.processlist：从 information_schema.processlist 采集线程状态计数。 • info_schema.query_response_time：如果 query_response_time_stats 为 ON，则采集查询响应时间分布(默认为 true)。 • info_schema.tables：从 information_schema.tables 采集指标。 • info_schema.tables.databases：用于收集表统计信息的数据库列表，逗号分隔，默认是 '*' 表示所有数据库。 • info_schema.tablestats：如果使用 userstat=1 运行，则设置为 true 以采集表统计信息。 • info_schema.schemastats：如果使用 userstat=1 运行，则设置为 true 以采集架构统计信息。 • info_schema.userstats：如果使用 userstat=1 运行，则设置为 true 以采集用户统计信息。 • perf_schema.eventsstatements：从 performance_schema.events_statements_summary_by_digest 采集指标。 • perf_schema.eventsstatements.digest_text_limit：标准化语句文本的最大长度，默认值为120。 • perf_schema.eventsstatements.limit：按响应时间限制事件语句摘要的数量，默认值为250。 • perf_schema.eventsstatements.timelimit：限制 'last_seen' 事件语句的最大时间，单位为秒，默认值为86400。 • perf_schema.eventsstatementssum：从 performance_schema.events_statements_summary_by_digest 汇总采集指标。 • perf_schema.eventswaits：从 performance_schema.events_waits_summary_global_by_event_name 采集指标。 • perf_schema.file_events：从 performance_schema.file_summary_by_event_name 采集指标。 • perf_schema.file_instances：从 performance_schema.file_summary_by_instance采集指标。 • perf_schema.indexiowaits：从 performance_schema.table_io_waits_summary_by_index_usage 采集指标。 • perf_schema.tableiowaits：从 performance_schema.table_io_waits_summary_by_table 采集指标。 • perf_schema.tablelocks：从 performance_schema.table_lock_waits_summary_by_table 采集指标。 • perf_schema.replication_group_member_stats：从 performance_schema.replication_group_member_stats 采集指标。 • perf_schema.replication_applier_status_by_worker：从 performance_schema.replication_applier_status_by_worker 采集指标。 • slave_status：从 SHOW SLAVE STATUS 采集，默认为 true。 • slave_hosts：从 SHOW SLAVE HOSTS 采集。 • heartbeat：从 heartbeat 采集。 • heartbeat.database：收集 heartbeat 数据的数据库，默认值为 heartbeat。 • heartbeat.table：收集 heartbeat 数据的表，默认值为 heartbeat。

方式二：自定义安装

> **说明**
> 为了方便安装管理 Exporter，推荐使用腾讯云 容器服务 来统一管理。

前提条件

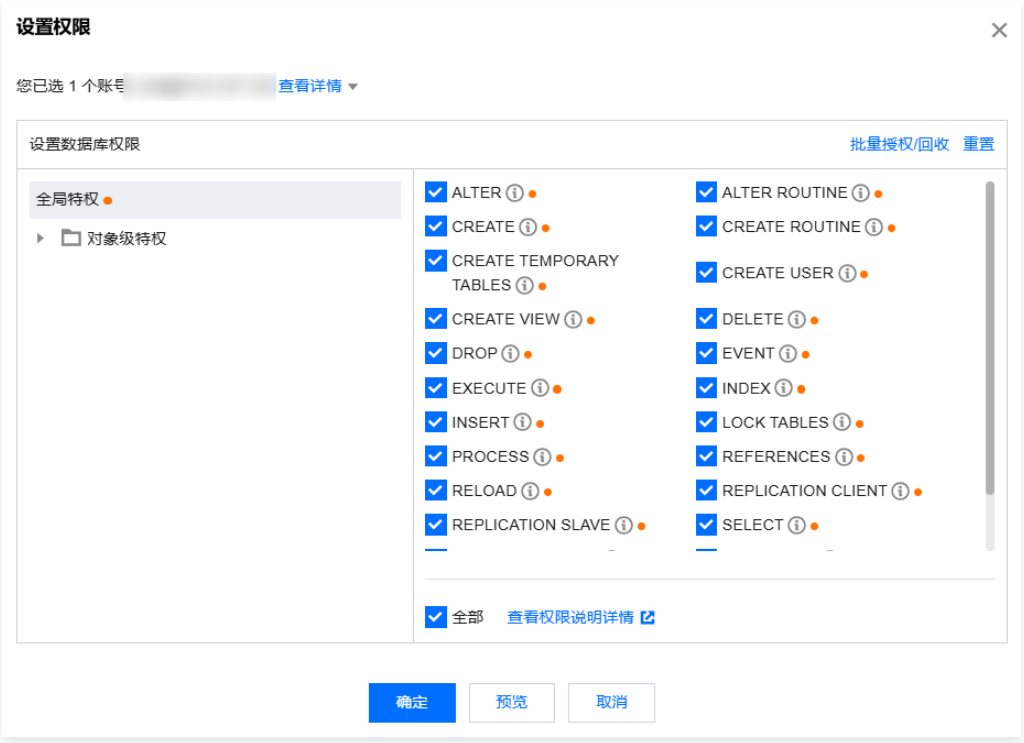
- 在 Prometheus 实例对应地域及私有网络（VPC）下，创建腾讯云容器服务 Kubernetes 集群，并为集群创建 命名空间。
- 在 Prometheus 监控服务控制台，选择并进入对应的 Prometheus 实例，在**数据采集 > 集成容器服务**中找到对应容器集群完成关联集群操作。可参见指引 关联集群。

操作步骤

步骤1：数据库授权

因为 MySQL Exporter 是通过查询数据库中状态数据来对其进行监控，所以需要为对应的数据库实例进行授权。账号和密码需根据实际情况而定，授权步骤如下：

1. 登录 云数据库 MySQL 控制台。
2. 在实例列表页面单击需要授权的数据库名称，进入数据库详情页。
3. 选择**数据库管理 > 账号管理**，进入账号管理页面，根据业务实际需要创建监控建立的账号。
4. 单击账号右侧操作项下的**修改权限**，修改对应权限。示例如下图所示：



您也可以在您的云服务器中通过执行以下命令进行授权。

```
CREATE USER 'exporter'@'ip' IDENTIFIED BY 'XXXXXXXX' WITH MAX_USER_CONNECTIONS 3;
GRANT PROCESS, REPLICATION CLIENT, SELECT ON *.* TO 'exporter'@'ip';
```

> **说明**
> 建议为该用户设置最大连接数限制，以避免因监控数据抓取对数据库带来影响。但并非所有的数据库版本中都可以生效，例如 MariaDB 10.1 版本不支持最大连接数设置，则无法生效。详情请参见 MariaDB 说明。

步骤2：Exporter 部署

1. 登录 容器服务控制台。
2. 在左侧菜单栏选择**集群**。
3. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的管理页面。

4. 使用 Secret 管理 MySQL 连接串。

4.1 在左侧菜单中选择**工作负载 > Deployment**，进入 Deployment 页面。

4.2 在页面右上角单击 **YAML 创建**，创建 YAML 配置，配置说明如下：

使用 Kubernetes 的 Secret 来管理连接串，并对连接串进行加密处理，在启动 MySQL Exporter 的时候直接使用 Secret Key，需要调整对应的**连接串**，YAML 配置示例如下：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret-test
  namespace: mysql-demo
type: Opaque
stringData:
  datasource: "user:password@tcp(ip:port)/"  #对应 MySQL 连接串信息
```

5. 部署 MySQL Exporter。

在 Deployment 管理页面，选择对应的命名空间来进行部署服务，可以通过 控制台的方式 创建。如下以 YAML 的方式部署 Exporter， 配置示例如下：

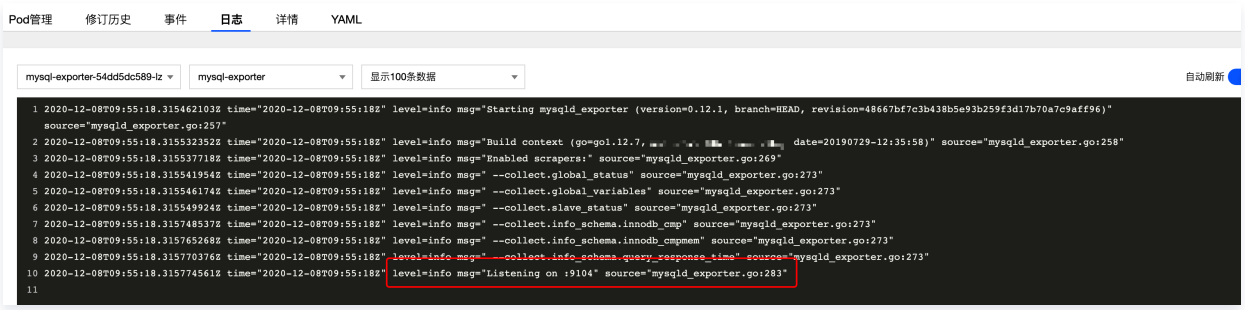
```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: mysql-exporter  # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
  name: mysql-exporter  # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
  namespace: mysql-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: mysql-exporter  # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
  template:
    metadata:
      labels:
        k8s-app: mysql-exporter  # 根据业务需要调整成对应的名称，建议加上 MySQL 实例的信息
    spec:
      containers:
      - env:
        - name: DATA_SOURCE_NAME
          valueFrom:
            secretKeyRef:
              name: mysql-secret-test # 对应上一步中的 Secret 的名称
              key: datasource # 对应上一步中的 Secret Key
        image: ccr.ccs.tencentyun.com/rig-agent/mysqld-exporter:v0.12.1
        imagePullPolicy: IfNotPresent
        name: mysql-exporter
        ports:
        - containerPort: 9104
          name: metric-port
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: qcloudregistrykey
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
```

```
      terminationGracePeriodSeconds: 30
```

6. 验证。

6.1 在 **Deployment** 页面单击上述步骤创建的 Deployment，进入 Deployment 管理页面。

6.2 单击**日志**页签，可以查看到 Exporter 成功启动并暴露对应的访问地址，如下图所示：

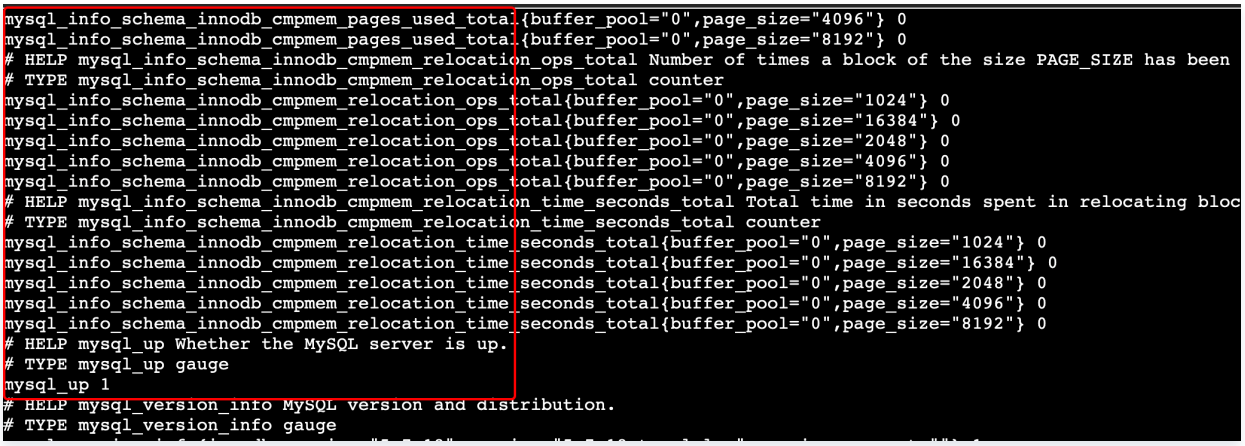


6.3 单击 **Pod 管理**页签进入 Pod 页面。

6.4 在右侧的操作项下单击**远程登录**，即可登录 Pod，在命令行窗口中执行以下 wget 命令，可以正常得到对应的 MySQL 指标。如发现未能得到对应的数据，请检查**连接串**是否正确，具体如下：

```
wget -O- localhost:9104/metrics
```

执行结果如下图所示：



步骤2：添加采集任务

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 选择**数据采集 > 集成容器服务**，选择已经关联的集群，通过**数据采集配置 > 新建自定义监控 > YAML 编辑**来添加采集配置。
3. 通过服务发现添加 **PodMonitors** 来定义 Prometheus 抓取任务，YAML 配置示例如下：

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: mysql-exporter   # 填写一个唯一名称
  namespace: cm-prometheus   # 按量实例：集群的 namespace；包年包月实例(已停止售卖)：namespace 固定，不要修改
spec:
  podMetricsEndpoints:
  - interval: 30s
    port: metric-port     # 填写pod yaml中Prometheus Exporter对应的Port的Name
    path: /metrics  # 填写Prometheus Exporter对应的Path的值，不填默认/metrics
    relabelings:
    - action: replace
      sourceLabels:
```

```
      - instance
        regex: (.*)
        targetLabel: instance
        replacement: 'crs-xxxxxx' # 调整成对应的 MySQL 实例 ID
      - action: replace
        sourceLabels:
        - instance
        regex: (.*)
        targetLabel: ip
        replacement: '1.x.x.x' # 调整成对应的 MySQL 实例 IP
    namespaceSelector:    # 选择要监控pod所在的namespace
      matchNames:
      - mysql-demo
    selector:  # 填写要监控pod的Label值，以定位目标pod
      matchLabels:
        k8s-app: mysql-exporter
```

查看监控

前提条件

Prometheus 实例已绑定 Grafana 实例。

操作步骤

1. 登录 Prometheus 监控服务控制台，选择对应 Prometheus 实例进入管理页面。
2. 选择**数据采集 > 集成中心**，进入集成中心页面，找到并单击 **MySQL**，选择 **Dashboard > Dashboard 操作**下的**安装/升级 Dashboard**，单击**安装/升级**安装对应的 Grafana Dashboard。
3. 选择**已集成**，在已集成列表中单击 Grafana 图标即可自动打开 MySQL 监控大盘，查看实例相关的监控数据，如下图所示：

配置告警

腾讯云 Prometheus 托管服务内置了部分 MySQL 数据库的报警策略模板，可根据业务实际的情况调整对应的阈值来添加告警策略。详情请参见 新建告警策略。

附录：MySQL Exporter 采集参数说明

MySQL Exporter 使用各种 `Collector` 来控制采集数据的启停，具体参数如下：

名称	MySQL 版本	描述
collect.auto_increment.columns	5.1	在 information_schema 中采集 auto_increment 和最大值。
collect.binlog_size	5.1	采集所有注册的 binlog 文件大小。
collect.engine_innodb_status	5.1	从 SHOW ENGINE INNODB STATUS 中采集状态数据。
collect.engine_tokudb_status	5.6	从 SHOW ENGINE TOKUDB STATUS 中采集状态数据。
collect.global_status	5.1	从 SHOW GLOBAL STATUS（默认开启）中采集状态数据。
collect.global_variables	5.1	从 SHOW GLOBAL VARIABLES（默认开启）中采集状态数据。
collect.info_schema.clientstats	5.5	如果设置了 userstat=1，设置成 true 来开启用户端数据采集。
collect.info_schema.innodb_metrics	5.6	从 information_schema.innodb_metrics 中采集监控数据。
collect.info_schema.innodb_tablespaces	5.7	从 information_schema.innodb_sys_tablespaces 中采集监控数据。
collect.info_schema.innodb_cmp	5.5	从 information_schema.innodb_cmp 中采集 InnoDB 压缩表的监控数据。
collect.info_schema.innodb_cmpmem	5.5	从 information_schema.innodb_cmpmem 中采集 InnoDB buffer pool compression 的监控数据。
collect.info_schema.processlist	5.1	从 information_schema.processlist 中采集线程状态计数的监控数据。
collect.info_schema.processlist.min_time	5.1	线程可以被统计所维持的状态的最小时间。（默认：0）
collect.info_schema.query_response_time	5.5	如果 query_response_time_stats 被设置成 ON，采集查询相应时间的分布。
collect.info_schema.replica_host	5.6	从 information_schema.replica_host_status 中采集状态数据。
collect.info_schema.tables	5.1	从 information_schema.tables 中采集状态数据。
collect.info_schema.tables.databases	5.1	设置需要采集表状态的数据库，或者设置成 '`*`' 来采集所有的。
collect.info_schema.tablestats	5.1	如果设置了 userstat=1，设置成 true 来采集表统计数据。
collect.info_schema.schemastats	5.1	如果设置了 userstat=1，设置成 true 来采集 schema 统计数据。
collect.info_schema.userstats	5.1	如果设置了 userstat=1，设置成 true 来采集用户统计数据。
collect.perf_schema.eventsstatements	5.6	从 performance_schema.events_statements_summary_by_digest 中采集监控数据。
collect.perf_schema.eventsstatements.digest_text_limit	5.6	设置正常文本语句的最大长度。（默认：120）
collect.perf_schema.eventsstatements.limit	5.6	事件语句的限制数量。（默认：250）

collect.perf_schema.eventsstatements.timelimit	5.6	限制事件语句 'last_seen' 可以保持多久，单位为秒。（默认：86400）
collect.perf_schema.eventsstatementssum	5.7	从 performance_schema.events_statements_summary_by_digest summed 中采集监控数据。
collect.perf_schema.eventswaits	5.5	从 performance_schema.events_waits_summary_global_by_event_name 中采集监控数据。
collect.perf_schema.file_events	5.6	从 performance_schema.file_summary_by_event_name 中采集监控数据。
collect.perf_schema.file_instances	5.5	从 performance_schema.file_summary_by_instance 中采集监控数据。
collect.perf_schema.indexiowaits	5.6	从 performance_schema.table_io_waits_summary_by_index_usage 中采集监控数据。
collect.perf_schema.tableiowaits	5.6	从 performance_schema.table_io_waits_summary_by_table 中采集监控数据。
collect.perf_schema.tablelocks	5.6	从 performance_schema.table_lock_waits_summary_by_table 中采集监控数据。
collect.perf_schema.replication_group_members	5.7	从 performance_schema.replication_group_members 中采集监控数据。
collect.perf_schema.replication_group_member_stats	5.7	从 from performance_schema.replication_group_member_stats 中采集监控数据。
collect.perf_schema.replication_applier_status_by_worker	5.7	从 performance_schema.replication_applier_status_by_worker 中采集监控数据。
collect.slave_status	5.1	从 SHOW SLAVE STATUS（默认开启）中采集监控数据。
collect.slave_hosts	5.1	从 SHOW SLAVE HOSTS 中采集监控数据。
collect.heartbeat	5.1	从 heartbeat 中采集监控数据。
collect.heartbeat.database	5.1	数据库心跳检测的数据源。（默认：heartbeat）
collect.heartbeat.table	5.1	表心跳检测的数据源。（默认：heartbeat）
collect.heartbeat.utc	5.1	对当前的数据库服务器使用 UTC 时间戳（`pt-heartbeat` is called with `--utc`）。（默认：false）

全局配置参数

名称	描述
config.my-cnf	用来读取数据库认证信息的配置文件 `.my.cnf` 位置。（默认：`~/.my.cnf`）
log.level	日志级别。（默认：info）
exporter.lock_wait_timeout	为链接设置 lock_wait_timeout（单位：秒）以避免对元数据的锁时间太长。（默认：2）
exporter.log_slow_filter	添加 log_slow_filter 以避免抓取的慢查询被记录。 **说明：** 不支持 Oracle MySQL。

web.listen-address	web 端口监听地址。
web.telemetry-path	metrics 接口路径。
version	打印版本信息。

heartbeat 心跳检测

如果开启 `collect.heartbeat`，mysqld_exporter 会通过心跳检测机制抓取复制延迟数据。

自建 Prometheus 监控 TKE 集群

最近更新时间：2024-10-21 14:57:11

本文主要描述使用自建 Prometheus 采集腾讯云容器服务 TKE 的监控数据时如何配置采集规则。TKE 集群内按照节点类型分为常规节点和超级节点，Prometheus 通过配置 `scrape_config` 来抓取节点和容器的监控数据，由于节点性质不同因此需要配置的采集规则略有差异。

常规节点采集规则

常规节点的采集配置文件如下所示：

```
- job_name: "tke-cadvisor"
  scheme: https
  metrics_path: /metrics/cadvisor # 采集容器 cadvisor 监控数据
  tls_config:
    insecure_skip_verify: true # tke 的 kubelet 使用自签证书，忽略证书校验
  authorization:
    credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: node
  relabel_configs:
  - source_labels: [__meta_kubernetes_node_label_node_kubernetes_io_instance_type]
    regex: eklet # 排除超级节点
    action: drop
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)
- job_name: "tke-kubelet"
  scheme: https
  metrics_path: /metrics # 采集 kubelet 自身的监控数据
  tls_config:
    insecure_skip_verify: true
  authorization:
    credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: node
  relabel_configs:
  - source_labels: [__meta_kubernetes_node_label_node_kubernetes_io_instance_type]
    regex: eklet
    action: drop
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)
- job_name: "tke-probes" # 采集容器健康检查健康数据
  scheme: https
  metrics_path: /metrics/probes
  tls_config:
    insecure_skip_verify: true
  authorization:
    credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: node
  relabel_configs:
  - source_labels: [__meta_kubernetes_node_label_node_kubernetes_io_instance_type]
    regex: eklet
    action: drop
  - action: labelmap
    regex: __meta_kubernetes_node_label_(.+)
```

使用说明：

- 使用节点服务发现（ `kubernetes_sd_configs` 的 role 为 `node` ），抓取所有节点 `kubelet:10250` 暴露的几种监控数据。

- 如果集群是普通节点与超级节点混用，排除超级节点（`relabel_configs` 中将带 `node.kubernetes.io/instance-type: eklet` 这种 label 的 node 排除）。
- TKE 节点上的 kubelet 证书是自签的，需要忽略证书校验，所以 `insecure_skip_verify` 要置为 true。
- kubelet 通过 `/metrics/cadvisor`，`/metrics` 与 `/metrics/probes` 路径分别暴露了容器 cadvisor 监控数据、kubelet 自身监控数据以及容器健康检查健康数据，为这三个不同路径分别配置采集 job 进行采集。

超级节点采集规则

超级节点的采集配置文件如下所示：

```
- job_name: eks # 采集超级节点监控数据
  honor_timestamps: true
  metrics_path: '/metrics' # 所有健康数据都在这个路径
  params: # 通常需要加参数过滤掉 ipvs 相关的指标，因为可能数据量较大，打高 Pod 负载。
    collect[]:
    - 'ipvs'
    # - 'cpu'
    # - 'meminfo'
    # - 'diskstats'
    # - 'filesystem'
    # - 'loadOvg'
    # - 'netdev'
    # - 'filefd'
    # - 'pressure'
    # - 'vmstat'
  scheme: http
  kubernetes_sd_configs:
  - role: pod # 超级节点 Pod 的监控数据暴露在 Pod 自身 IP 的 9100 端口，所以使用 Pod 服务发现
  relabel_configs:
  - source_labels: [__meta_kubernetes_pod_annotation_tke_cloud_tencent_com_pod_type]
    regex: eklet # 只采集超级节点的 Pod
    action: keep
  - source_labels: [__meta_kubernetes_pod_phase]
    regex: Running # 非 Running 状态的 Pod 机器资源已释放，不需要采集
    action: keep
  - source_labels: [__meta_kubernetes_pod_ip]
    separator: ;
    regex: (.*)
    target_label: __address__
    replacement: ${1}:9100 # 监控指标暴露在 Pod 的 9100 端口
    action: replace
  - source_labels: [__meta_kubernetes_pod_name]
    separator: ;
    regex: (.*)
    target_label: pod # 将 Pod 名字写到 "pod" label
    replacement: ${1}
    action: replace
  - source_labels: [__meta_kubernetes_namespace]
    separator: ;
    regex: (.*)
    target_label: namespace # 将 Pod 所在 namespace 写到 "namespace" label
    replacement: ${1}
    action: replace
  metric_relabel_configs:
  - source_labels: [__name__]
    separator: ;
    regex: (container_.*|pod_.*|kubelet_.*)
    replacement: $1
    action: keep
```

使用说明：

- 超级节点的监控数据暴露在每个 Pod 的9100端口的 `/metrics` 这个 HTTP API 路径（非 HTTPS），使用 Pod 服务发现（`kubernetes_sd_configs` 的 role 为 `pod`），用一个 job 就可以采集完。
- 超级节点的 Pod 支持通过 `collect[]` 这个查询参数来过滤掉不希望采集的指标，这样可以避免指标数据量过大，导致 Pod 负载升高，通常要过滤掉 `ipvs` 的指标。
- 如果集群是普通节点与超级节点混用，确保只采集超级节点的 Pod（`relabel_configs` 中只保留有 `tke.cloud.tencent.com/pod-type:eklet` 这个注解的 Pod）。
- 如果 Pod 的 phase 不是 Running 也无法采集，可以排除。
- `container_` 开头的指标是 cadvisor 监控数据，`pod_` 前缀指标是超级节点 Pod 所在子机的监控数据（相当于将 `node_exporter` 的 `node_` 前缀指标替换成了 `pod_`），`kubelet_` 前缀指标是超级节点 Pod 子机内兼容 kubelet 的指标（主要是 pvc 存储监控）。

kube-prometheus-stack 配置

通常使用 kube-prometheus-stack 这个 helm chart 来自建 Prometheus，在 `values.yaml` 中进行自定义配置然后安装到集群，其中可以配置 Prometheus 原生的 `scrape_config`（非 CRD），配置方法是将自定义的 `scrape_config` 写到 `prometheus.prometheusSpec.additionalScrapeConfigs` 字段下，示例如下：

```
prometheus:
  prometheusSpec:
    additionalScrapeConfigs:
    - job_name: "tke-cadvisor"
      scheme: https
      metrics_path: /metrics/cadvisor
      tls_config:
        insecure_skip_verify: true
      authorization:
        credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: node
      relabel_configs:
      - source_labels: [__meta_kubernetes_node_label_node_kubernetes_io_instance_type]
        regex: eklet
        action: drop
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
    - job_name: "tke-kubelet"
      scheme: https
      metrics_path: /metrics
      tls_config:
        insecure_skip_verify: true
      authorization:
        credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: node
      relabel_configs:
      - source_labels: [__meta_kubernetes_node_label_node_kubernetes_io_instance_type]
        regex: eklet
        action: drop
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
    - job_name: "tke-probes"
      scheme: https
      metrics_path: /metrics/probes
      tls_config:
        insecure_skip_verify: true
      authorization:
        credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
```

```
    - role: node
    relabel_configs:
    - source_labels: [__meta_kubernetes_node_label_node_kubernetes_io_instance_type]
      regex: eklet
      action: drop
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
  - job_name: eks
    honor_timestamps: true
    metrics_path: '/metrics'
    params:
      collect[]: ['ipvs']
      # - 'cpu'
      # - 'meminfo'
      # - 'diskstats'
      # - 'filesystem'
      # - 'loadavg'
      # - 'netdev'
      # - 'filefd'
      # - 'pressure'
      # - 'vmstat'
    scheme: http
    kubernetes_sd_configs:
    - role: pod
    relabel_configs:
    - source_labels: [__meta_kubernetes_pod_annotation_tke_cloud_tencent_com_pod_type]
      regex: eklet
      action: keep
    - source_labels: [__meta_kubernetes_pod_phase]
      regex: Running
      action: keep
    - source_labels: [__meta_kubernetes_pod_ip]
      separator: ;
      regex: (.*)
      target_label: __address__
      replacement: ${1}:9100
      action: replace
    - source_labels: [__meta_kubernetes_pod_name]
      separator: ;
      regex: (.*)
      target_label: pod
      replacement: ${1}
      action: replace
    - source_labels: [__meta_kubernetes_namespace]
      separator: ;
      regex: (.*)
      target_label: namespace
      replacement: ${1}
      action: replace
    metric_relabel_configs:
    - source_labels: [__name__]
      separator: ;
      regex: (container_.*|pod_.*|kubelet_.*)
      replacement: $1
      action: keep
  storageSpec:
   volumeClaimTemplate:
     spec:
       accessModes: ["ReadWriteOnce"]
       resources:
```

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

腾讯云 Prometheus 一键关联监控容器服务

最近更新时间：2024-10-24 16:42:21

实践背景

Prometheus 是容器场景下的最佳监控工具之一，但自建 Prometheus 对于运维人力有限的中小型企业而言，成本较高；对于业务发展快速的大企业又容易出现性能瓶颈。因此，使用云上托管的 Prometheus 已成为越来越多上云企业的首选。本文将介绍如何使用 托管 Prometheus 监控腾讯云容器服务 TKE。

前提条件

已创建腾讯云 Prometheus 监控实例，具体步骤请参考 Prometheus 监控服务-创建实例 。

实践步骤

步骤1：Prometheus 监控实例绑定集群

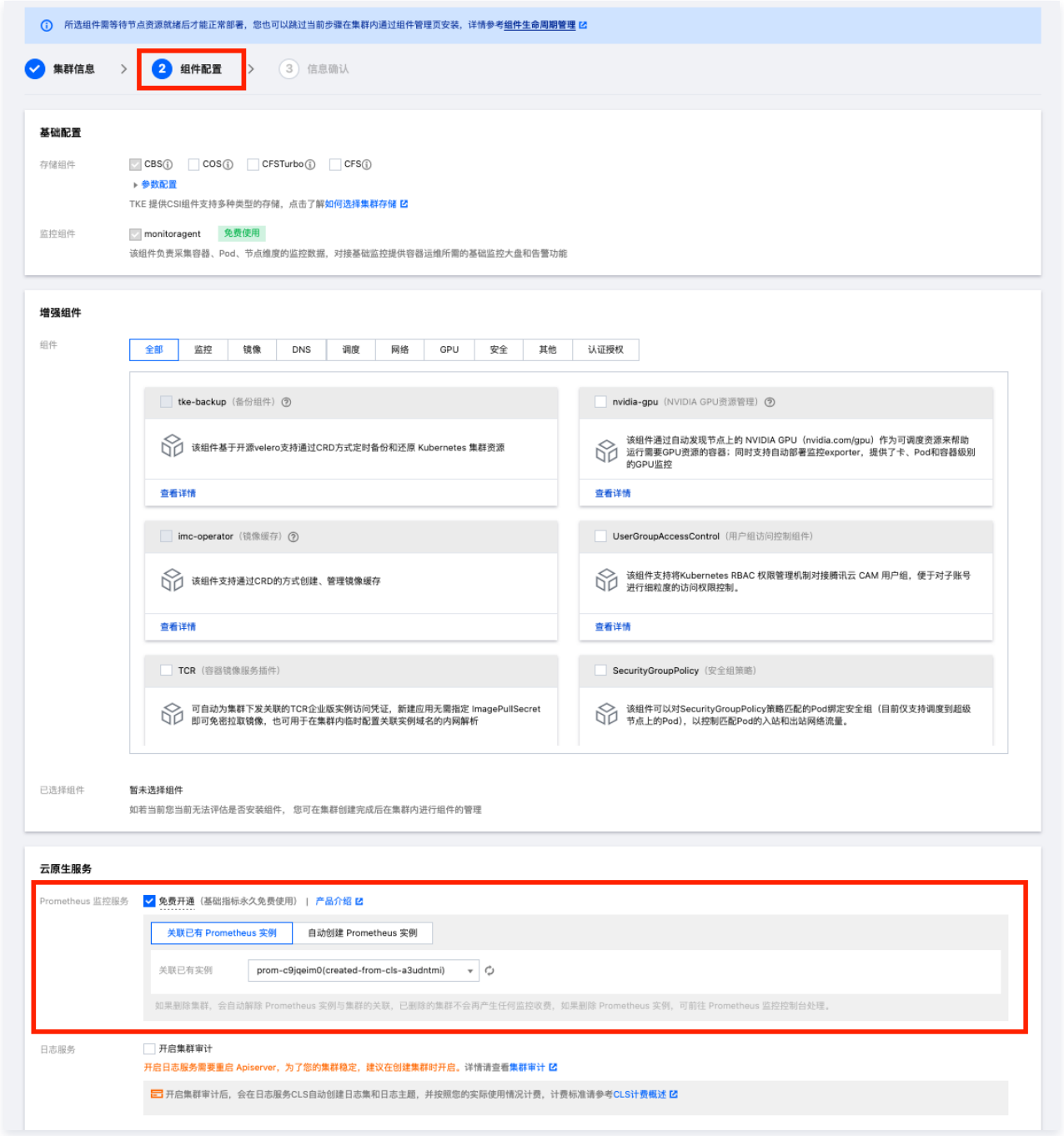
方式1：在 Prometheus 监控实例绑定新建集群

> **说明：**
> 仅支持新建标准集群和 Serverless 集群时绑定。

1. 登录 容器服务控制台 。
2. 在左侧菜单栏中单击**集群**，在集群管理页面单击**新建**。
3. 选择集群类型。

选择标准集群类型

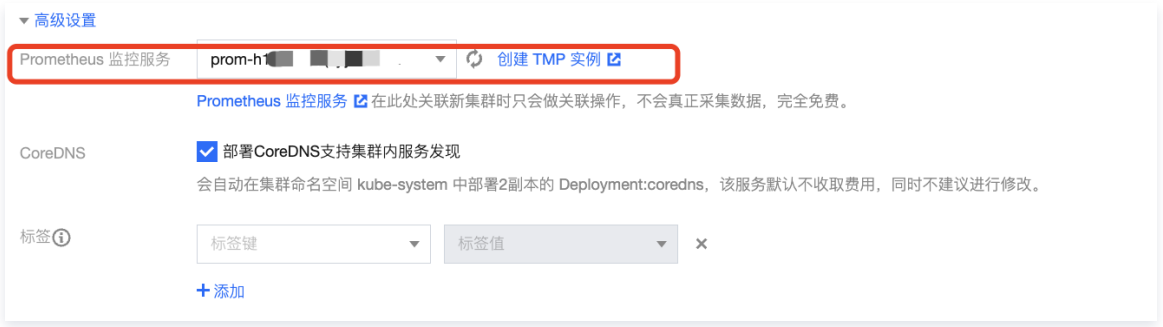
根据页面提示进行配置，更多配置说明可参考 标准集群-创建集群 。
在第2步骤-组件配置中绑定 Prometheus 实例。



选择 Serverless 集群

根据页面提示进行配置，更多配置说明可参考 Serverless 集群-创建实例 。

在高级配置中绑定 Prometheus 实例。



4. 所有配置项完成后，单击**完成**即绑定成功。

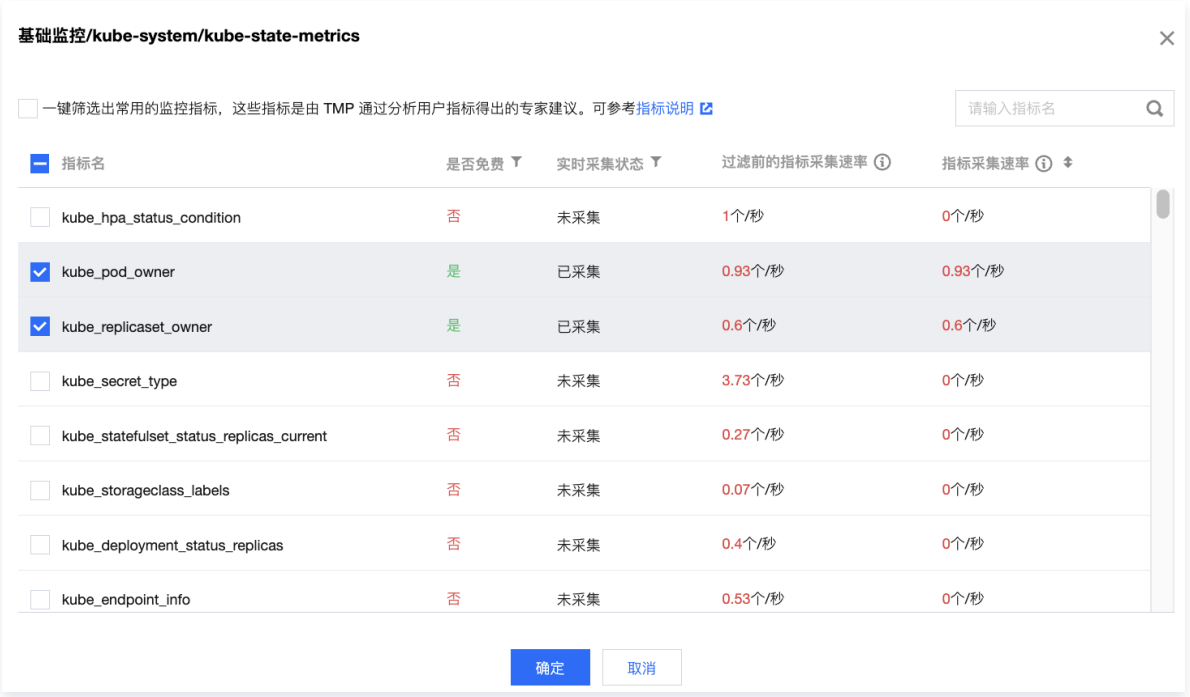
方式2：在 Prometheus 监控实例中绑定已有集群

> **说明：**
> 支持已有的标准集群、Serverless 集群、注册集群绑定 Prometheus 实例。

1. 登录 容器服务控制台 。
2. 在左侧菜单栏中单击 **Prometheus 监控**，进入 Prometheus 监控列表页。
3. 在 Prometheus 监控列表页，单击对应的 Prometheus 实例**操作**中的**关联集群**。
4. 在关联集群页面选择对应的集群类型，勾选对应的集群。
5. 完成后，单击**确定**即绑定成功。

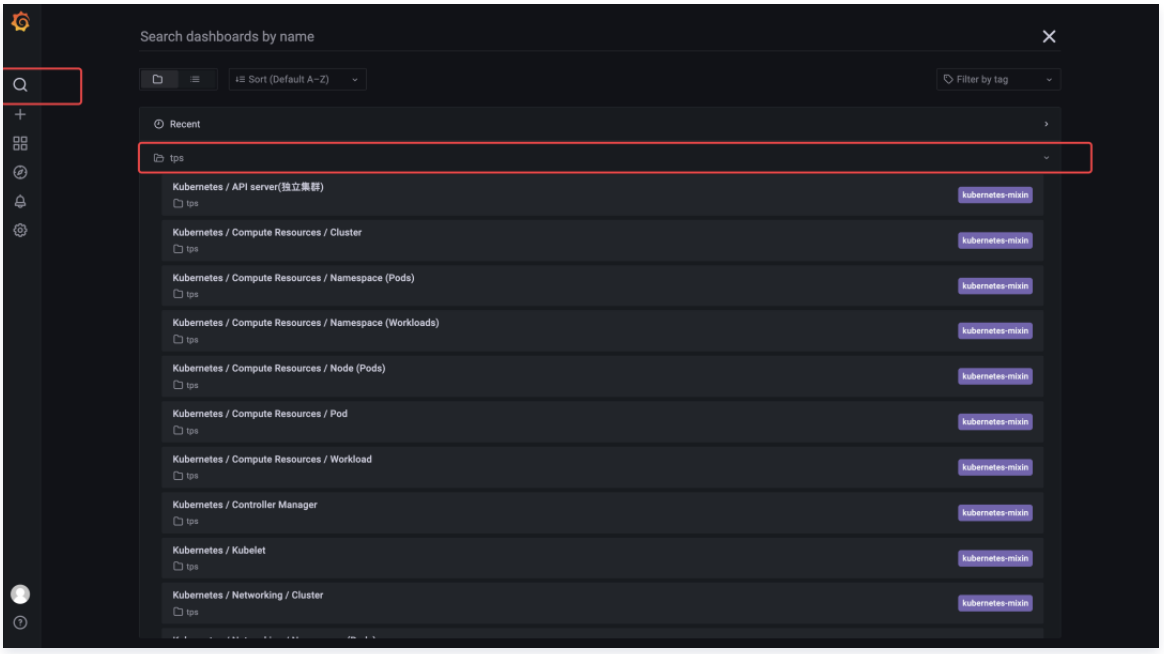
步骤2：数据采集配置

1. 登录 容器服务控制台 。
2. 在左侧菜单栏中单击 **Prometheus 监控**，进入 Prometheus 监控列表页。
3. 在 Prometheus 监控列表页，单击对应的 Prometheus 实例。
4. 单击**操作**中的**数据采集配置**。
5. 在数据采集配置-基础监控页面，单击指标详情，勾选需要监控的指标。详细配置说明请参见 数据采集配置 ，指标说明请参见 容器服务指标 。

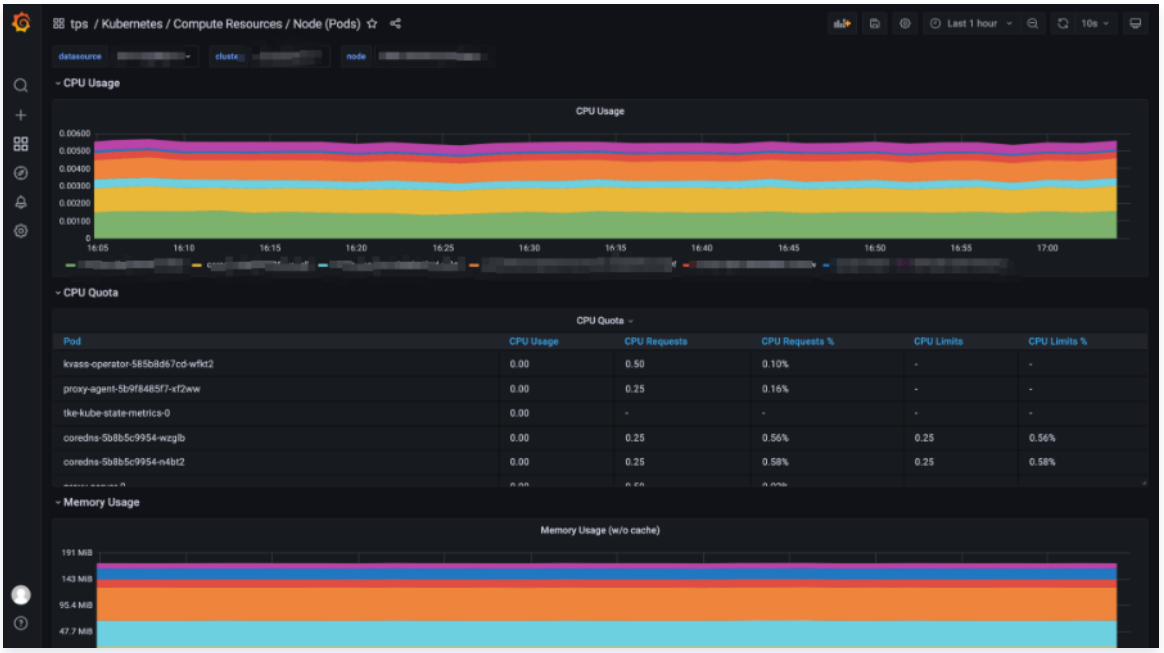


步骤3：登录 Grafana 查看监控数据

1. 登录 容器服务控制台 。
2. 在左侧菜单栏中单击 **Prometheus 监控**，进入 Prometheus 监控列表页。
3. 在 Prometheus 监控列表页中，单击实例名称右侧的 Grafana 图标，输入账号名称和密码，进入 Grafana 服务平台。
4. 在 Grafana 服务平台 > Dashboard 搜索列表，默认预设了容器相关的监控面板，单击某个面板名称。

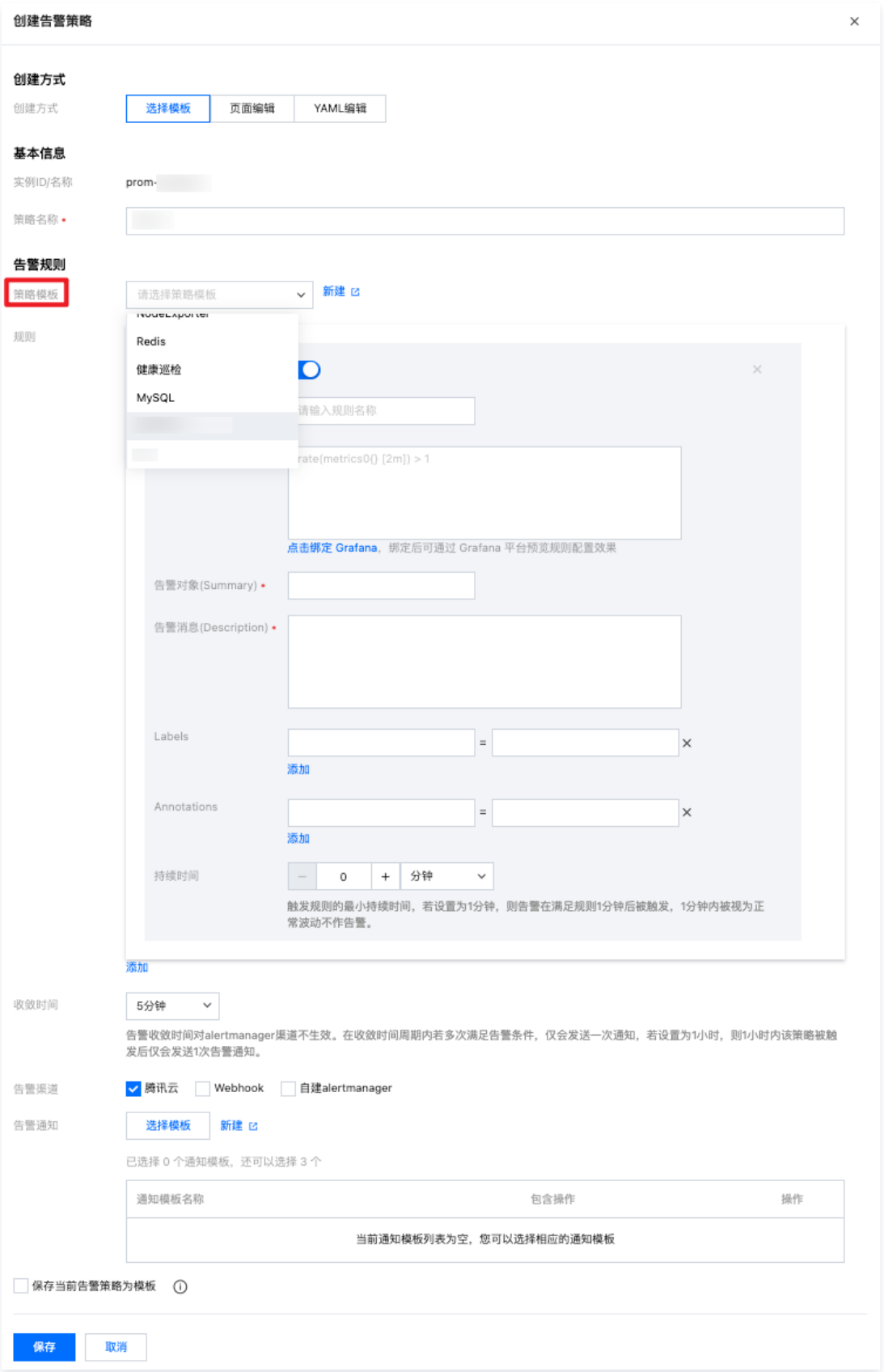


5. 进入面板页面，即可查看预设的监控数据图表。



步骤4：配置告警策略

1. 登录 容器服务控制台 。
2. 在左侧菜单栏中单击 **Prometheus 监控**，进入 Prometheus 监控列表页。
3. 在 Prometheus 监控列表页，单击对应的 Prometheus 实例。
4. 单机**告警管理**，在告警管理页面单击**新建告警策略**。
5. 您可以在新建告警策略页，选择预设的模板类型，无需手动配置，告警通知可选择腾讯云可观测平台已有的通知模板，从而实现快速配置告警。



更多操作说明请参考：

- 自定义上报指标
- Prometheus 监控服务–操作指南

一键接入腾讯云应用性能监控 APM

最近更新时间：2024-10-29 16:09:22

背景

为了尽可能的降低应用接入成本，减少 APM 工具对于整个软件研发流程的影响，腾讯云在2024年5月正式推出了 tencent-opentelemetry-operator 接入方案。该方案为部署在 Kubernetes 上的多语言应用提供了一种简单高效的 APM 接入方式。

应用接入方案对比

- **手动上报方案**

  这是最基本的接入方式，开发人员需要在代码中显式调用 APM 工具提供的 API 来采集和上报性能数据。优点是灵活性高，可以精确控制需要收集的数据和收集时机；缺点是需要修改应用程序代码，工作量非常大。
- **半手动上报方案**

  在这种方案中，开发人员仍需要在代码中调用 APM 工具提供的 API，但可以基于常用的框架或库简化一部分工作。例如，一些 APM 工具提供了与常见开发框架（如 Spring、Express 等）集成的库，开发人员只需要在配置文件中启用这些库，即可自动收集和上报性能数据。
- **探针方案**

  探针是一种自动收集性能数据的方法，它是一个运行在应用程序进程内的模块，可以在应用程序运行时动态地收集性能数据，无需修改应用程序代码。探针方案同样提供了对常见开发框架的集成，可以自动收集和上报性能数据。相比手动上报方案，部署更加简单，可以在应用编译后进行引入，甚至可以做到对整个开发过程无感知。
- **Sidecar 方案**

  Sidecar 是一种运行在应用程序进程外的模块，通常以单独的进程或容器的形式存在，可以监听应用程序的网络通信，收集性能数据。优点是无需修改应用程序代码，且对应用程序性能的影响非常小；缺点是 Sidecar 方案部署复杂，需要管理额外的进程或容器，只能基于 Service Mesh（服务网格）等特殊应用架构才能发挥价值。
- **eBPF 方案**

  eBPF（Extended Berkeley Packet Filter）是 Linux 内核中的一种技术，可以在内核中运行用户定义的程序，收集各种系统和网络性能数据。使用 eBPF 的 APM 工具可以提供非常详细和精确的性能数据，且对应用程序性能的影响极小。对于已经部署到 Kubernetes 的容器化应用，eBPF 方案的部署比较简单，但和 Sidecar 方案一样，缺少成熟的分布式链路追踪能力。

方案选型建议如下：

接入方案	接入难度	兼容性	功能覆盖度	成熟度
手动上报	极高	高	高	高
半手动上报	高	高	高	高
探针	低	比较高，适合大多数语言。	高	高
Sidecar	低	低，依赖特定应用架构。	低	高
eBPF	低	比较高，依赖高版本内核。	中	低

综合来看，对于 APM 方案的选型，接入难度是首要的考虑因素。如果接入 APM 工具还需要修改业务代码，会给开发团队带来沉重的负担，长远来看必将在团队内部造成推广困难的局面，这也是很多企业在引入 APM 工具的过程中放弃的主要原因。因此，我们需要尽量排除手动上报方案。虽然 eBPF 方案代表着未来 APM 的发展方向，但其成熟度较低，功能覆盖度也有较大不足。因此探针方案依然是目前最成熟、最值得考虑的选择，特别是对于 Java、Python 等基于虚拟机运行的编程语言，天然就和探针方案有极高的匹配度。

Operator 方案

tencent-opentelemetry-operator 由腾讯云在社区 opentelemetry-operator 基础上构建，是基于探针的接入方案。由于实现了 Kubernetes 环境下探针自动注入机制，免去了探针部署和配置的工作量，让应用接入变得更加简单。目前 tencent-opentelemetry-operator 支持 Java、Python、node.js、.Net 应用的快速接入，应用程序部署到腾讯云容器服务 TKE 以后，只需要在工作负载的 YAML 文件中添加2行 annotation，就能完成整个接入流程，无需在开发态涉及任何代码的修改。

安装 tencent-opentelemetry-operator

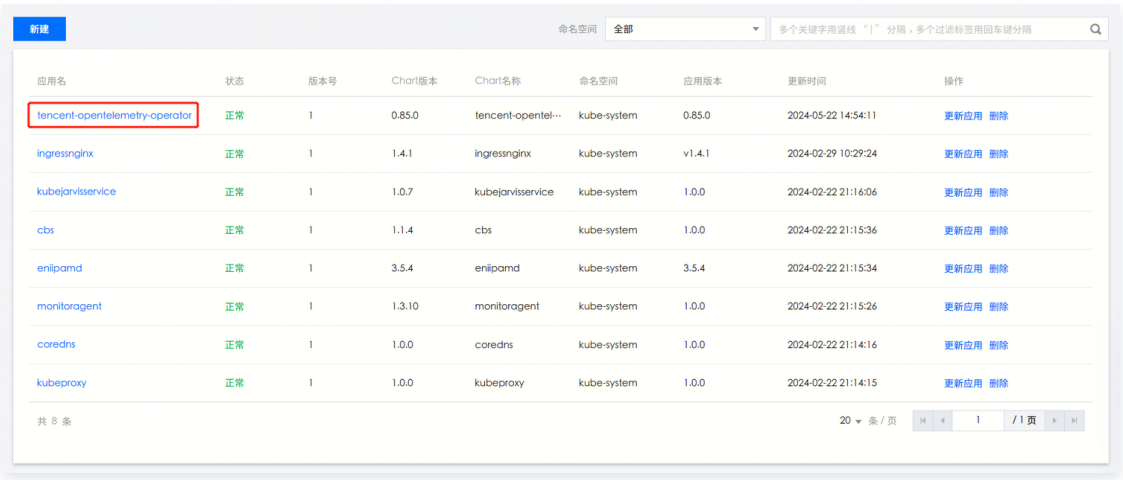
1. 登录 腾讯云可观测平台 。
2. 在左侧导航中选择**应用性能监控 > 应用列表**，在应用列表页面，单击**接入应用**。

3. 在弹出框选择需要接入的编程语言，例如 Java。
4. 进入接入 Java 应用页，选择 **TKE 环境自动接入 > 一键安装 Operator**。

5. 进入一键安装 Operator 的弹出窗口，通过下拉框选择上报地域、默认业务系统、TKE 集群等信息，确认后单击**安装**即可。

> **说明：**
> 上报地域为 APM 业务系统所在的地域，每个 TKE 集群只能指定唯一的上报地域，建议将上报地域与 TKE 集群所在的地域保持一致。

6. 安装完成之后，前往 容器服务控制台，在左侧导航中选择**应用市场 > 应用管理**，查看安装在 kube-system 命名空间的 tencent-opentelemetry-operator 应用，也可以通过 Helm 客户端执行 `helm list` 命令检查安装状态。



应用接入

在 TKE 集群完成 Operator 的安装后，得益于 Operator 方案动态探针注入能力，应用接入的过程极为简单，只需要编辑应用所在工作负载的 YAML 文件，在 Pod Template 中添加2个 annotation，即可完成应用接入。以 Java 应用为例，需要添加的内容如下：

```
cloud.tencent.com/inject-java: "true" #接入APM
cloud.tencent.com/otel-service-name: my-app #指定应用名
```

其中，otel-service-name 字段代表应用名，多个使用相同应用名接入的进程，在 APM 中会表现为相同应用下的多个实例。完整的工作负载 YAML 文件示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: my-app
  name: my-app
  namespace: default
spec:
  selector:
    matchLabels:
      k8s-app: my-app
  template:
    metadata:
      labels:
        k8s-app: my-app
      annotations:
        cloud.tencent.com/inject-java: "true"  #接入APM
        cloud.tencent.com/otel-service-name: my-app #指定应用名
    spec:
      containers:
        image: my-app:0.1
        name: my-app
```

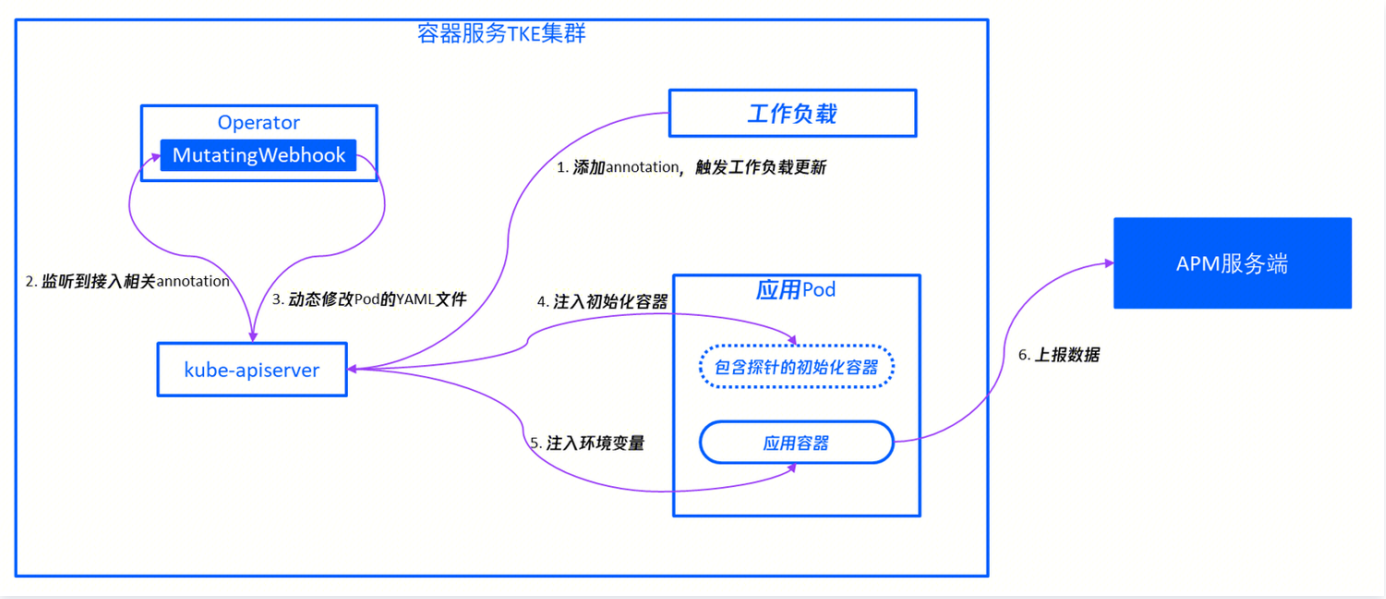
添加完 annotation 之后，将基于不同的工作负载发布策略对工作负载进行更新，触发应用 Pod 的重新创建。新启动的 Pod 会自动注入探针，并连接到 APM 服务端，将监控数据上报到 Operator 的默认业务系统。前往 腾讯云可观测平台控制台，在**应用性能监控 > 应用监控 > 应用列表**中，即可查询到新接入的应用。
更多应用接入指引，请参考下述帮助文档：

应用	参考文档
Java	TKE 环境自动接入 Java 应用（推荐）

Python	TKE 环境自动接入 Python 应用（推荐）
Node.js	TKE 环境自动接入 Node.js 应用（推荐）

接入原理

Operator 方案利用了 Kubernetes 提供的 Dynamic Admission Control 机制实现探针的注入，可以参考以下流程了解 Operator 方案的接入原理：



1. 用户对工作负载添加 annotation，API Server 将基于工作负载的发布策略启动工作负载的更新。
2. 在 Operator 的相关组件中，包含一个具备 Dynamic Admission Control 能力的工作负载，通过 Mutating Webhook 的形式在容器服务集群中运行，这个组件可以从 API Server 监听工作负载的变化。
3. Operator 监听到应用工作负载添加了接入相关的 annotation 之后，会通过 API Server 对即将创建的应用 Pod 进行动态修改，修改的内容通过 Pod 的 YAML 描述表现出来。
4. Operator 会在应用 Pod 中注入一个初始化容器，在容器本地路径中包含了探针文件。
5. Operator 还会在应用容器中注入一些环境变量，这些环境变量可以对应用进程进行配置，实现挂载探针的动作。以 Java 应用为例，Operator 会对应用容器添加 JAVA_TOOL_OPTIONS 环境变量，其中包括 -javaagent 参数，指向由初始化容器引入的探针文件本地路径，使 Java 进程启动的时候执行挂载探针操作。
6. 探针通过字节码增强机制自动采集监控数据，并向 APM 服务端上报。

接入优势

- **简化 Operator 安装流程**：安装部署流程，不但上架了 TKE 应用市场，还能在 APM 控制台实现 Operator 一键安装。
- **通过封装减少人力**：对接入过程中需要用到的一系列配置项进行了封装，不再需要手工填写，包括 APM 接入点的地址，业务系统 token 等。
- **优化证书管理机制**：解决了 Operator 安装过程中有可能遇到的各类兼容性问题。
- **降级处理**：当 APM 平台的任何一个组件出现故障的时候，整个接入机制都以不影响业务正常运行为前提，在必要的时候对探针注入和监控数据上报逻辑行降级处理，确保业务的稳定运行。

自定义埋点

在探针接入方案的基础上，Operator 模式通过探针注入进一步简化了应用接入 APM 的流程。tencent-opentelemetry-operator 注入的探针来自 OpenTelemetry 生态，对主流的开发框架与类库实现了运行态自动埋点。对大多数用户而言，应用在接入成功后即可完成监控数据上报，实现分布式链路追踪，不需要修改任何代码。但在某些特殊的场景中，如果探针的自动埋点机制不能满足用户需求，用户可以引入 OpenTelemetry API，在自动埋点的基础上，通过修改应用代码增加自定义埋点。例如下述场景中，可能需要引入自定义埋点：

- 应用引入的开发框架与类库没有在开发者中被广泛使用，或者版本很低。
- 应用引入的开发框架与类库由用户自己封装，或者来自于闭源商业化定制开发。
- 对于某些核心的自定义方法，需要在链路中追加方法级的埋点。

对于 tencent-opentelemetry-operator 注入的探针，可以参考 OpenTelemetry 方案支持的组件和框架 了解自动埋点的范围。自定义埋点代码的具体编写方式，可以参考如下 OpenTelemetry 的官方文档：

应用	参考链接
Java	https://opentelemetry.io/docs/languages/java/instrumentation/
Python	https://opentelemetry.io/docs/languages/python/instrumentation/
Node.js	https://opentelemetry.io/docs/languages/js/instrumentation/

以 Java 语言为例，如果我们需要对某个自定义方法在链路中追加方法级埋点，可以参考以下步骤实现。

1. 引入 OpenTelemetry API 依赖：

```
<dependencies>
    <!-- 其他依赖 -->
    <dependency>
        <groupId>io.opentelemetry</groupId>
        <artifactId>opentelemetry-api</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-bom</artifactId>
            <version>1.9.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. 通过如下代码，在链路中增加 doTask() 的方法级埋点。

```
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.StatusCode;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Scope;

// Trace 对象可以在业务方法中获取，或者通过参数传入业务方法
public void doTask(Tracer tracer) {
    // 创建一个 Span
    Span span = tracer.spanBuilder("doTask").startSpan();
    // 在 Span 中添加一些 Attributes
    span.setAttribute("RequestId", "5fc92ff1-8ca8-45f4-8013-24b4b5257666");
    // 将此 Span 设置为当前的Span
    try (Scope scope = span.makeCurrent()) {
        doSubTask1();
        doSubTask2();
    } catch (Throwable t) {
        // 处理异常，异常信息将记录到 Span 的对应事件中
        span.recordException(t);
        span.setStatus(StatusCode.ERROR);
        throw t;
    } finally {
        // 结束 Span
        span.end();
    }
}
```

3. 前往 腾讯云可观测平台控制台，在**应用性能监控 > 调用查询**中，找到相关的调用链，单击 Span ID 后，链路详情页面中，即可查到通过自定义埋点新增的 Span。

总结

腾讯云应用性能监控（APM）为企业提供了全面、高效、易用、低成本的应用性能管理解决方案，帮助企业优化应用程序的性能。新发布的 Operator 方案，降低了接入成本，将接入 APM 工具的决策时间点从开发态后移至部署态，用户不再需要在代码中引入 APM 相关 SDK 或者将探针文件打包到容器镜像中。这实现对应用开发的零侵入，对于在团队内部全部推广 APM 工具，有着非常重要的意义。

运维
TKE 集群中节点移出再移入操作指引

最近更新时间：2025-02-26 16:53:52

操作场景

在容器服务 TKE 的众多场景中，例如 K8S 版本升级、内核版本升级等，都需要进行节点移出再移入的操作。本文详细介绍了节点移出再移入的过程，主要分为以下几个步骤：

1. 驱逐节点上运行的 Pod。
2. 将节点移出集群再重新添加到集群，该节点将重装系统。
3. 解除封锁。

注意事项

- 如果单个集群中的多个节点都需进行移出再移入操作，建议逐个节点进行。即先完成单个节点移出再移入操作，并验证服务正常，再进行下一个节点的移出再移入操作，直到多个节点依次完成。
- 如果单个账号下的多个集群都需进行节点移出再移入操作，建议分批执行。且在每操作完一个集群之后，立即验证集群状态是否正常。

操作步骤

步骤1：驱逐 Pod

在集群节点进行移入再移出操作之前，需先将待移出节点上的 Pod 驱逐到其他节点上运行。驱逐的过程即逐个删除节点上的 Pod，再前往其他节点进行重建。

驱逐原理

为了简化节点维护操作，K8S 引入了 `drain` 命令，其使用原理如下：
K8S 1.4 之后的版本， `drain` 操作为先对节点进行封锁，再对节点上的所有 Pod 进行删除操作。如果该 Pod 被 Deployment 等控制器所管理，则控制器在检查到 Pod 副本数减少的情况下，会重新创建一个 Pod ，调度到其他满足条件的节点上。如果该 Pod 是裸 Pod ，不被控制器管理，则驱逐后不会重新创建。
此过程是先删除，再创建，并非滚动更新。因此更新过程中，可能会导致被驱逐的服务部分请求失败，如果被驱逐的服务所有相关 Pod 都在被驱逐的节点上，则可能导致该服务完全不可用。
为了避免这一情况的出现， K8S 1.4 之后的版本引入了 PDB。只要在 PDB 策略文件中选中某个业务（一组 Pod），声明该业务可容忍的最小副本数量，此时再执行 `drain` 操作，将不再直接删除 Pod，而是会通过 `evict api` 检查是否满足 PDB 策略，只有在满足 PDB 策略的情况下才会对 Pod 进行删除，保护了业务可用性。需要注意的是，只有正确配置 PDB 策略才能保证 `drain` 操作时业务影响在可控范围内。

驱逐前检查

驱逐的过程涉及了 Pod 的重建，可能会对集群中的服务造成影响，因此建议在驱逐前执行如下检查：

1. 检查集群中的剩余节点是否有足够资源去运行待驱逐节点上的 Pod。节点的资源分配情况可通过容器服务控制台查看。在 集群列表 页面，选择**目标集群 ID > 节点管理 > 节点**，检查“节点列表”页面中的“已分配/总资源”。如下图所示：

   如果节点的剩余资源不足，建议您向集群中新增节点，防止被驱逐的 Pod 出现无法运行的情况，从而对服务造成影响。
2. 检查集群中是否有配置主动驱逐保护 PodDisruptionBudget（PDB）。主动驱逐保护会中断驱逐操作的执行，建议先删除主动驱逐保护 PDB。
3. 检查集群中是否存在单个服务的所有 Pod 都落在待驱逐的节点上。如果单个服务的所有 Pod 都落在同一个节点上，驱逐 Pod 的动作会造成整个服务不可用。建议判断该服务是否强要求 Pod 都落在同一个节点上：
   - 否，建议给服务增加反亲和性调度。
   - 是，建议选择在业务低流量或者无流量的时间段内操作。
4. 检查服务是否使用了本地盘（hostpath）。如果服务使用了 `hostpath volume` 方式，则当 Pod 被调度到其他节点上时，数据会丢失，可能会对业务造成影响。如果是重要数据，建议先备份再进行驱逐。

> **说明：**

> 目前 kubelet 的镜像拉取策略是串行的，如果短时间内有大量的 Pod 都被调度到同一个节点上之后，Pod 的启动时间有可能会变长。

操作详情

目前，对于 TKE 集群可以有以下两种方式完成驱逐：

通过 TKE 控制台驱逐

1. 在 集群列表 页面，单击目标集群 ID。
2. 在集群详情页中，选择**节点管理 > 节点**。
3. 在**节点**页面，选择目标节点所在行右侧的**驱逐**。如下图所示：

4. 在弹窗中确认节点信息，并单击**确定**以驱逐节点上运行的 Pod。

通过 kubectl drain 命令驱逐

1. 请参考 使用标准登录方式登录 Linux 实例（推荐），登录节点。
2. 执行以下命令，进行该节点上 Pod 驱逐。

```
kubectl drain node <node-name>
```

步骤2：移出节点

当节点上运行的 Pod 被驱逐后，该节点处于封锁状态。如下图所示：

1. 在**节点列表**页面，单击目标节点所在行右侧的**移出**。
2. 在弹出窗口中，取消勾选“销毁按量计费的节点”，并单击**确定**即可将节点移除集群。如下图所示：

> **说明：**
> - 请记录该节点 ID，用于重新添加到集群。
> - 如果该节点是按量计费节点，注意不要勾选**销毁按量计费的节点**，销毁后不可恢复。包年包月计费模式的节点不支持销毁。

步骤3：重新加入该节点到集群

1. 在**节点列表**页面，单击页面上方的**添加已有节点**。
2. 在**添加已有节点**页面，输入记录的节点 ID，并单击 。
3. 在搜索结果列表中勾选节点，并配置云服务器其他参数。如下图所示：

> **注意：**
> **数据盘挂载**与**容器目录**默认不勾选。
> 如果您需要将容器和镜像存储在数据盘，则勾选**数据盘挂载**。选择数据盘挂载时，已格式化的 ext3、ext4、xfs 文件系统的数据盘将直接挂载，其他文件系统或未格式化的数据盘将自动格式化为 ext4 并挂载。
> 如果您需要保留数据盘数据并挂载，且需要避免数据盘被格式化，可参考以下步骤：
> 1. 在“云服务器配置”页面，不勾选**数据盘挂载**。

2. 打开“高级设置”，在“自定义数据”输入以下节点初始化脚本，并勾选**开启封锁**。如下图所示：


```
systemctl stop kubelet
docker stop $(docker ps -a | awk '{ print $1}' | tail -n +2)
systemctl stop dockerd
echo '/dev/vdb   /data    ext4   noatime,acl,user_xattr 1 1' >> /etc/fstab
mount -a
sed -i 's#"graph": "/var/lib/docker",#"data-root": "/data/docker",#g' /etc/docker/daemon.json
systemctl start dockerd
systemctl start kubelet
```

4. 请根据实际情况进行登录密码及安全组设置，并单击**完成**，等待节点添加成功。

步骤4：解除封锁

> **说明：**
> 节点添加成功后，处于封锁状态。

1. 在**节点列表**页面，选择该节点所在行右侧的**更多 > 取消封锁**。
2. 在弹出窗口中，单击**确定**即可解除封锁。

使用 Ansible 批量操作 TKE 节点

最近更新时间：2023-05-17 15:41:06

操作场景

容器服务 TKE 集群新增节点可通过在“自定义数据”中填入脚本来进行批量操作，例如统一修改内核参数。但如需对已新增的存量节点进行批量操作，您可参考本文使用开源工具 Ansible 进行操作。

原理介绍

Ansible 是一款流行的开源运维工具，可以直接通过 SSH 协议批量操作机器，无需事先进行手动安装依赖等操作，十分便捷。原理示意图如下：

操作步骤

准备 Ansible 控制节点

1. 选取实例作为 Ansible 的控制节点，通过此节点批量发起对存量 TKE 节点的操作。可选择与集群所在私有网络 VPC 中任意实例作为控制节点（包括 TKE 节点）。
2. 选定控制节点后，选择对应方式安装 Ansible：
   - Ubuntu 操作系统安装方式：

     ```
     sudo apt update && sudo apt install software-properties-common -y && sudo apt-add-repository --yes
     --update ppa:ansible/ansible && sudo apt install ansible -y
     ```

   - CentOS 操作系统安装方式：

     ```
     sudo yum install ansible -y
     ```

准备配置文件

将所有需要进行配置操作的节点内网 IP 配置到 `host.ini` 文件中，每行一个 IP。示例如下：

```
10.0.3.33
10.0.2.4
```

如需操作所有节点，可通过以下命令一键生成 `hosts.ini` 文件。

```
kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="InternalIP")].address}' | tr ' '
'\n' > hosts.ini
```

准备批量执行脚本

将需批量执行的操作写入脚本，并保存为脚本文件。示例如下：
自建镜像仓库后没有权威机构颁发证书，直接使用 HTTP 或 HTTPS 自签发的证书，默认情况下 dockerd 拉取镜像时会报错。此时可通过批量修改节点的 dockerd 配置，将自建仓库地址添加到 dockerd 配置的 `insecure-registries` 中使 dockerd 忽略证书校验。脚本文件 `modify-dockerd.sh` 内容如下：

```
# yum install -y jq # centos
apt install -y jq # ubuntu
```

```
cat /etc/docker/daemon.json | jq '."insecure-registries" += ["myharbor.com"]' > /tmp/daemon.json
cp /tmp/daemon.json /etc/docker/daemon.json
systemctl restart dockerd
```

使用 Ansible 批量执行脚本

通常 TKE 节点在新增时均指向一个 SSH 登录密钥或密码。请按照实际情况执行以下操作：

使用密钥

1. 准备密钥文件，例如 `tke.key` 。
2. 执行以下命令，授权密钥文件。

```
chmod 0600 tke.key
```

3. 批量执行脚本：
   - Ubuntu 操作系统节点批量执行示例如下：

```
ansible all -i hosts.ini --ssh-common-args="-o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null" --user ubuntu --become --become-user=root --private-key=tke.key -m
script -a "modify-dockerd.sh"
```

   - 其他操作系统节点批量执行示例如下：

```
ansible all -i hosts.ini --ssh-common-args="-o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null" --user root -m script -a "modify-dockerd.sh"
```

使用密码

1. 执行以下命令，将密码输入至 PASS 变量。

```
read -s PASS
```

2. 批量执行脚本：
   - Ubuntu 操作系统节点的 SSH 用户名默认为 ubuntu，批量执行示例如下：

```
ansible all -i hosts.ini --ssh-common-args="-o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null" --user ubuntu --become --become-user=root -e "ansible_password=$PASS"
-m script -a "modify-dockerd.sh"
```

   - 其他系统节点的 SSH 用户名默认为 root，批量执行示例如下：

```
ansible all -i hosts.ini --ssh-common-args="-o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null" --user root -e "ansible_password=$PASS" -m script -a "modify-
dockerd.sh"
```

使用集群审计排查问题

最近更新时间：2024-09-02 17:34:41

使用场景

当发生人为误操作、应用出现 bug、恶意程序调用 apiserver 接口，集群资源会被删除或修改。此时可通过集群审计功能记录 apiserver 的接口调用，即可根据条件检索和分析审计日志找到问题原因。本文介绍了集群审计功能的具体使用场景及使用示例，您可参考本文开始使用集群审计功能。

> **注意：**
> 本文仅适用于容器服务 TKE 集群。

前提条件

登录容器服务控制台，开启集群审计功能。详情请参见 开启集群审计 。

使用示例

获取分析结果

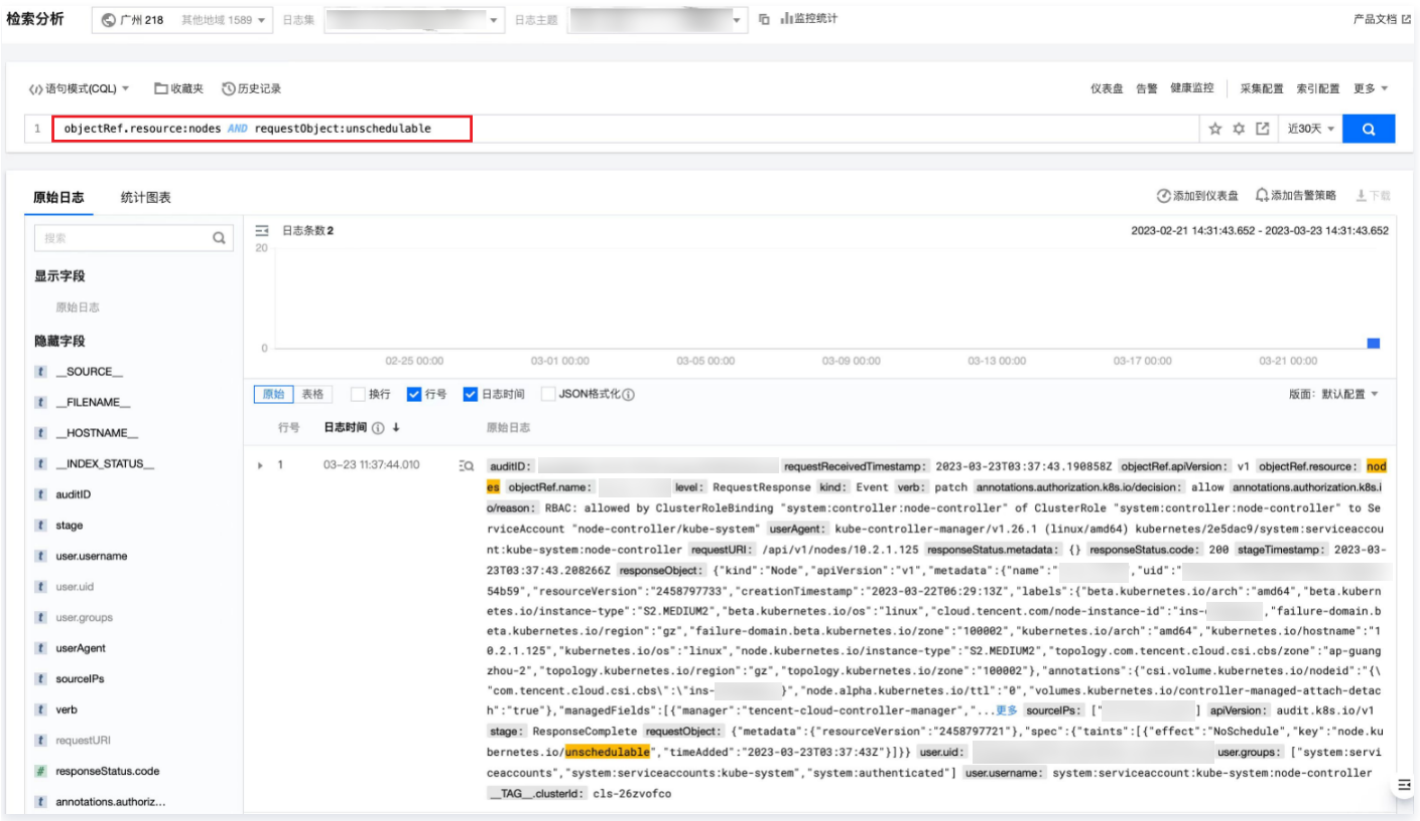
1. 登录 日志服务控制台，选择左侧导航栏中的**检索分析**。
2. 在**检索分析**页面，选择待检索的日志集，日志主题以及选择时间范围。
3. 输入分析语句后单击**检索分析**，即可获得分析结果。

示例1：查询封锁节点的操作者

例如，需查询封锁节点的操作者，则可执行以下命令进行检索：

```
objectRef.resource:nodes AND requestObject:unschedulable
```

在**检索分析**页面中，版面选择**默认配置**，查询结果如下图所示：

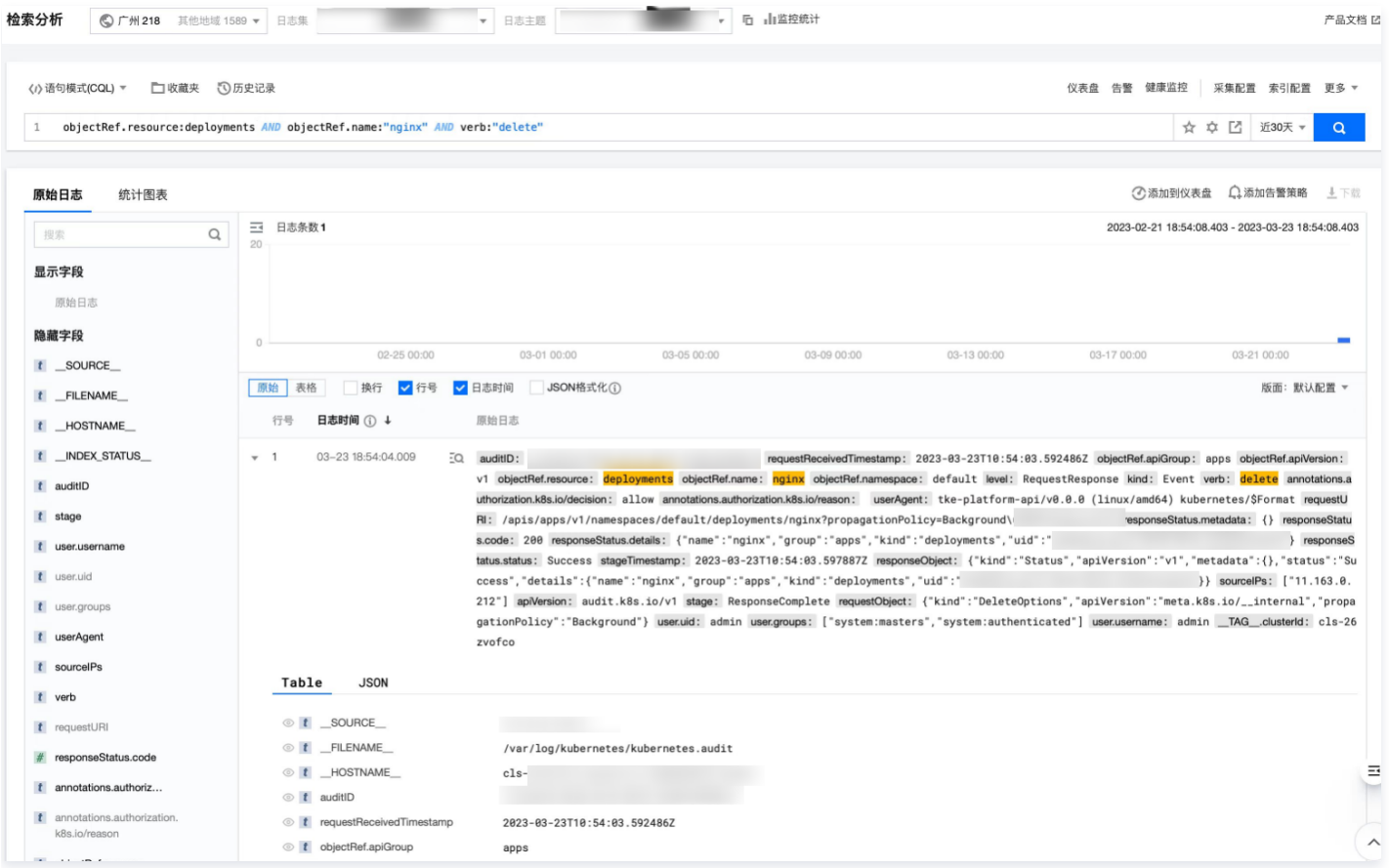


示例2：查询删除工作负载的操作者

例如，需查询删除工作负载的操作者，则可执行以下命令进行检索：

```
objectRef.resource:deployments AND objectRef.name:"nginx" AND verb:"delete"
```

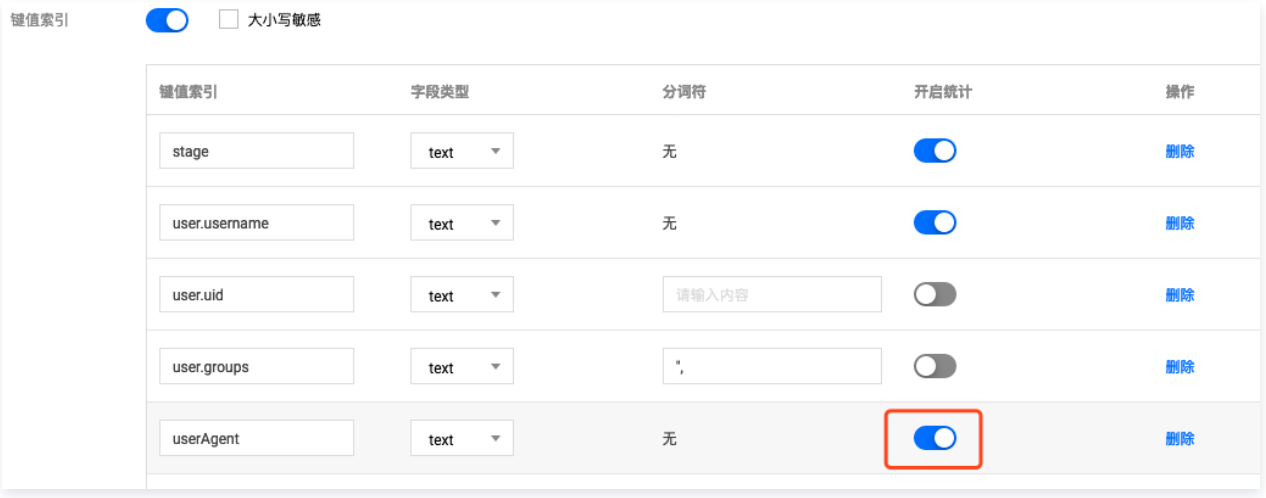
您可根据检索结果获取此子账号的详细信息。



示例3：定位 apiserver 限频原因

为避免恶意程序或 bug 导致对 apiserver 请求频率过高引发的 apiserver/etcd 负载过高，影响正常请求。apiserver 具备默认请求频率限制保护。如发生限频，可通过审计找到发出大量请求的客户端。

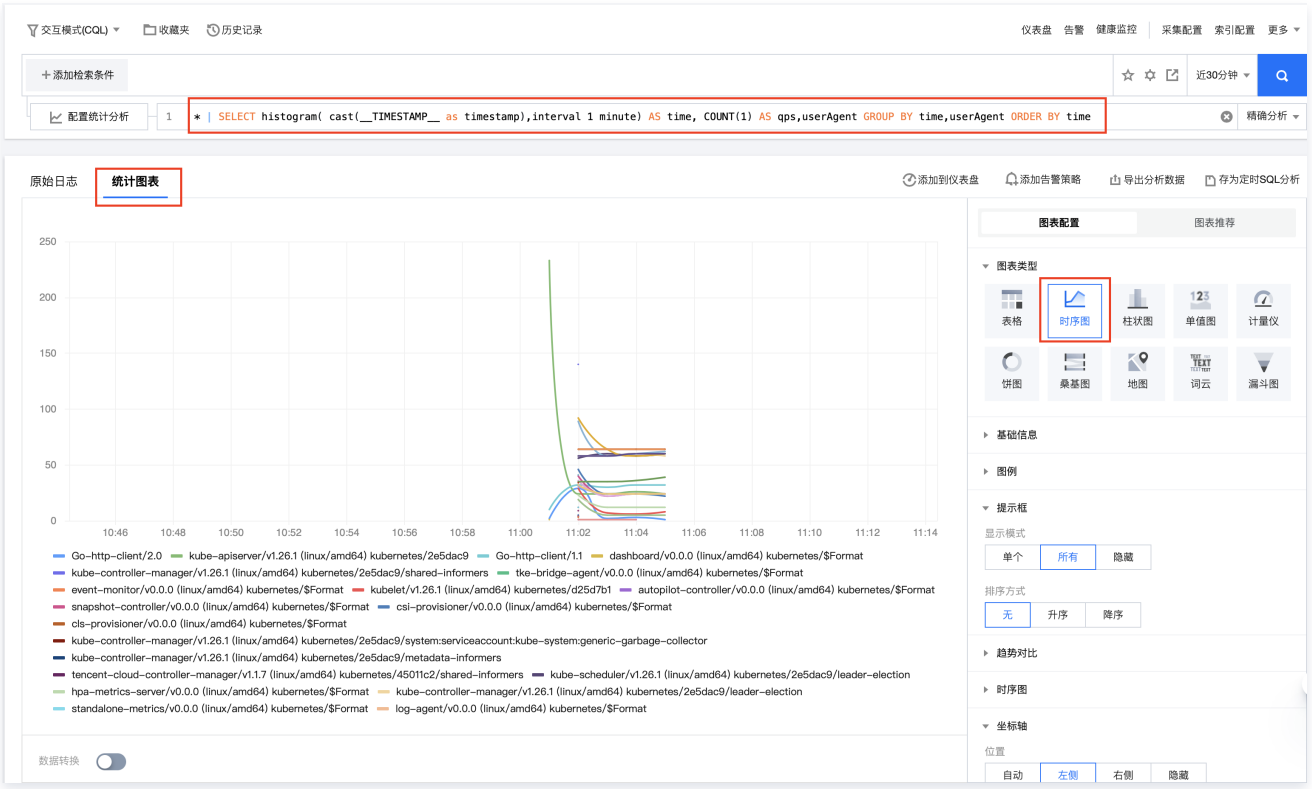
1. 如需通过 userAgent 分析统计请求的客户端，则需在“键值索引”窗口中修改日志主题，为 userAgent 字段开启统计。如下图所示：



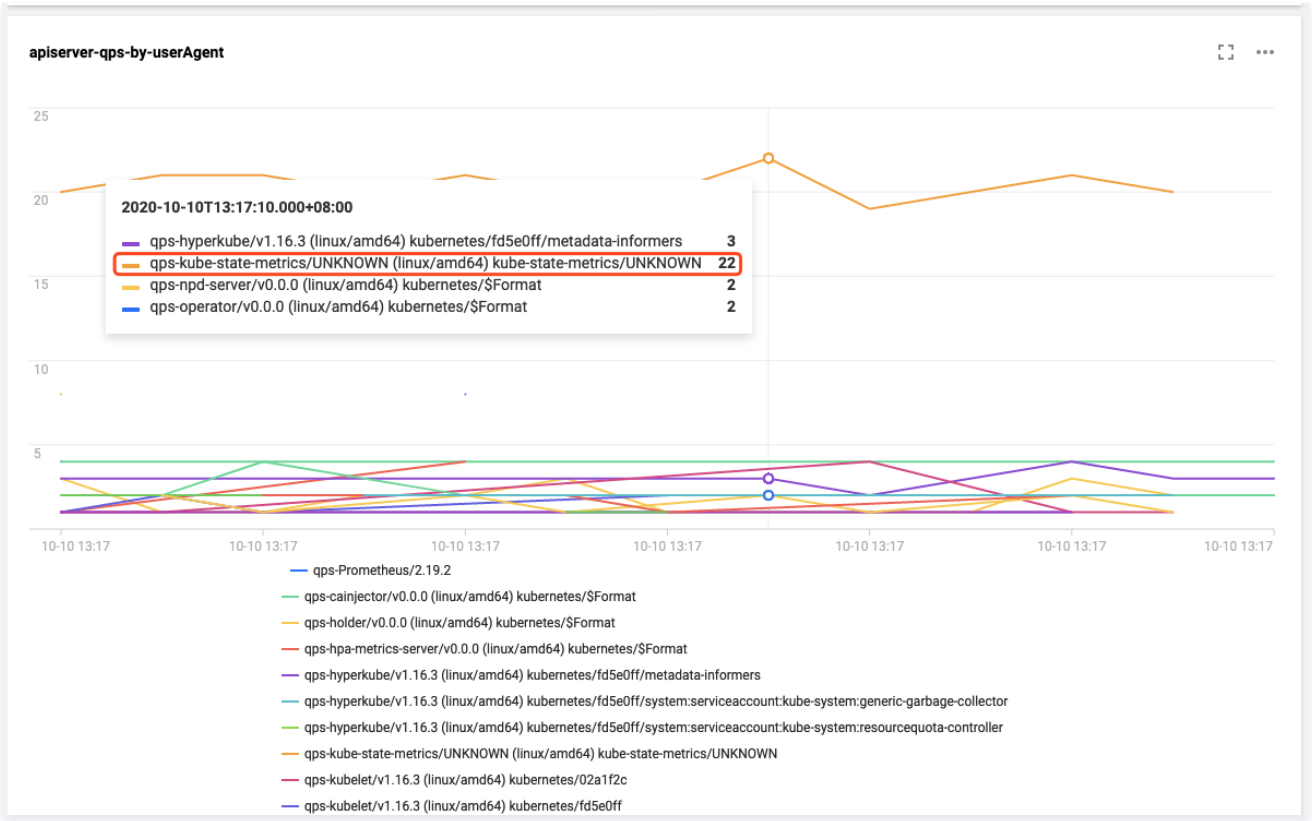
2. 执行以下命令，对每种客户端请求 apiserver 的 QPS 大小进行统计：

```
* | SELECT histogram( cast(__TIMESTAMP__ as timestamp),interval 1 minute) AS time, COUNT(1) AS qps,userAgent GROUP BY time,userAgent ORDER BY time
```

3. 切换到统计图表，选择时序图，可设置基本信息、坐标轴等，如下图所示：



获得数据后，可点击添加到仪表盘，放大显示。如下图所示：



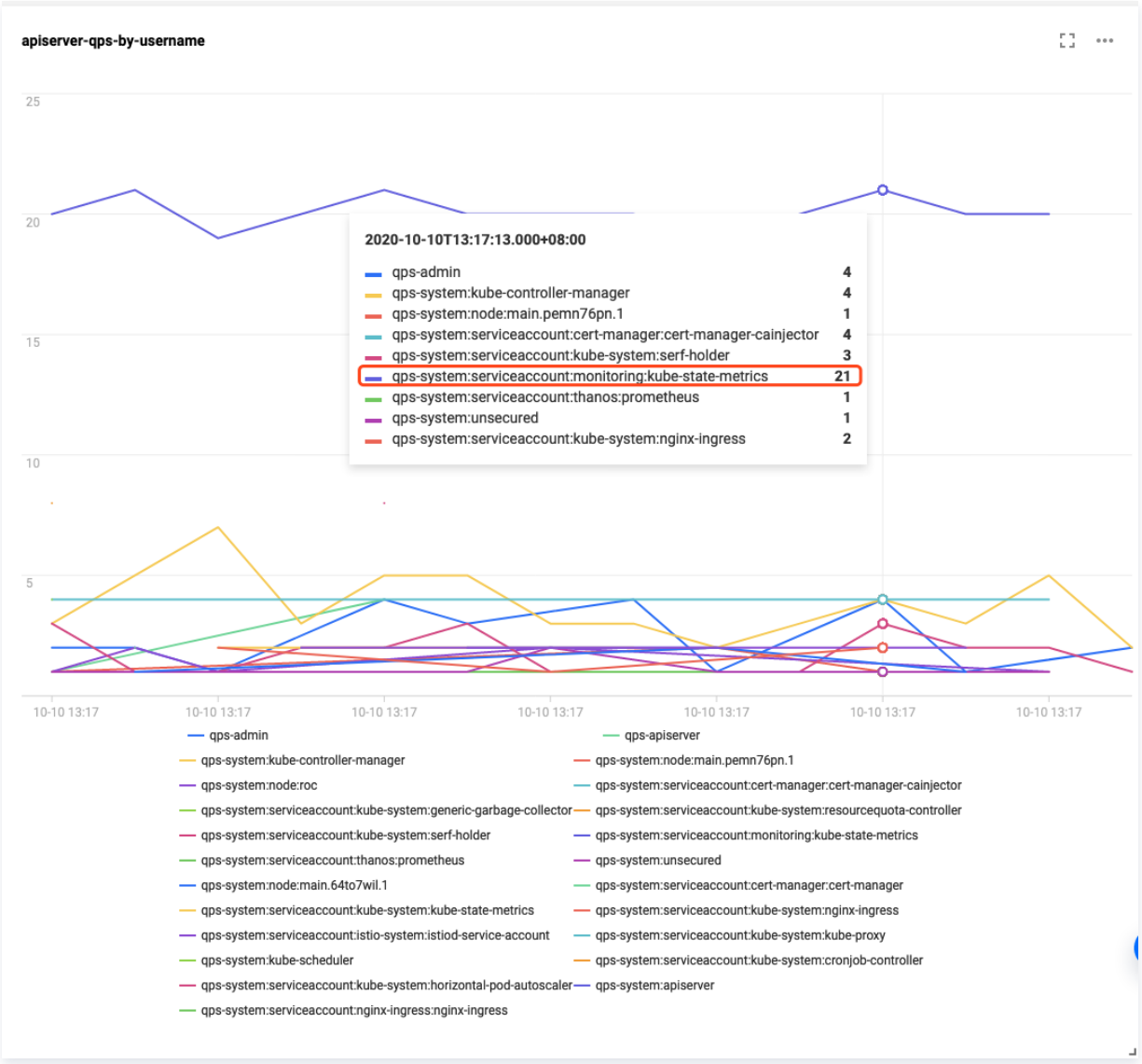
由图可见，kube-state-metrics 客户端对 apiserver 请求频率远远高于其它客户端。查看日志可得，由于 RBAC 授权问题导致 kube-state-metrics 不停的请求 apiserver 重试，触发了 apiserver 的限频。日志如下所示：

```
I1009 13:13:09.760767       1 request.go:538] Throttling request took 1.393921018s, request:
GET:https://172.16.252.1:443/api/v1/endpoints?limit=500&resourceVersion=1029843735
E1009 13:13:09.766106       1 reflector.go:156] pkg/mod/k8s.io/client-go@v0.0.0-20191109102209-
3c0d1af94be5/tools/cache/reflector.go:108: Failed to list *v1.Endpoints: endpoints is forbidden: User
"system:serviceaccount:monitoring:kube-state-metrics" cannot list resource "endpoints" in API group ""
at the cluster scope
```

同理，如果要使用其它字段来区分要统计的客户端，可以根据需求灵活修改 SQL，例如使用 user.username 来区分。SQL 语句可参考如下示例：

```
* | SELECT histogram( cast(__TIMESTAMP__ as timestamp),interval 1 minute)  AS time, COUNT(1) AS
qps,user.username GROUP BY time,user.username ORDER BY time
```

显示效果如下图所示：



相关文档

- 关于容器服务 TKE 的集群审计简介与基础操作，请参见 集群审计 。
- 集群审计的数据存储在日志服务，若需要在日志服务控制台中对审计结果进行检索和分析，检索语法请参见 日志检索语法与规则 。
- 进行分析需提供日志服务所支持的 SQL 语句，请参见 统计分析（SQL） 。

为 TKE Ingress 证书续期

最近更新时间：2023-05-17 15:41:06

操作场景

使用容器服务 TKE 控制台创建的 Ingress 配置的证书，会引用 SSL 证书 中托管的证书，若 Ingress 使用时间较长，证书存在过期的风险。证书过期会对线上业务造成巨大影响，因此需要在证书过期前进行续期，您可参考本文为 Ingress 证书续期。

操作步骤

查询证书到期时间

1. 登录 SSL 证书控制台，选择左侧导航栏中的**我的证书**。
2. 在证书列表项的“到期时间”中，查看即将过期的证书。

新增证书

在“证书管理”页面中，为旧证书续期生成新证书。您可根据自身情况选择**购买证书**、**申请免费证书**或**上传证书**中的任意一种方式来添加新证书。

查看引用旧证书的 Ingress

1. 登录 SSL 证书控制台，选择旧证书右侧的**关联资源**即可查看引用此证书的负载均衡器。
2. 点击负载均衡器的 ID 跳转到**负载均衡**详情页面。如果是 TKE Ingress 的负载均衡器，在标签栏会出现 `tke-clusterId` 和 `tke-lb-ingress-uuid` 的标签，分别表示集群 ID 和 Ingress 资源的 UID。如下图所示：

3. 在负载均衡器的“基本信息”页面，点击标签行右侧的编辑按钮，即可进入“编辑标签”页面。如下图所示：

4. 使用 Kubectl 可以查询集群 ID 对应集群的 Ingress，过滤 uid 为 `tke-lb-ingress.uuid` 对应值的 Ingress 资源。参考代码示例如下：

```
$ kubectl get ingress --all-namespaces -o=custom-
columns=NAMESPACE:.metadata.namespace,INGRESS:.metadata.name,UID:.metadata.uid | grep 1a******-****-
****-a329-eec697a28b35
api-prod    gateway      1a******-****-****-a329-eec697a28b35
```

由查询结果可知，该集群中 `api-prod/gateway` 引用了此证书，因此需要更新此 Ingress。

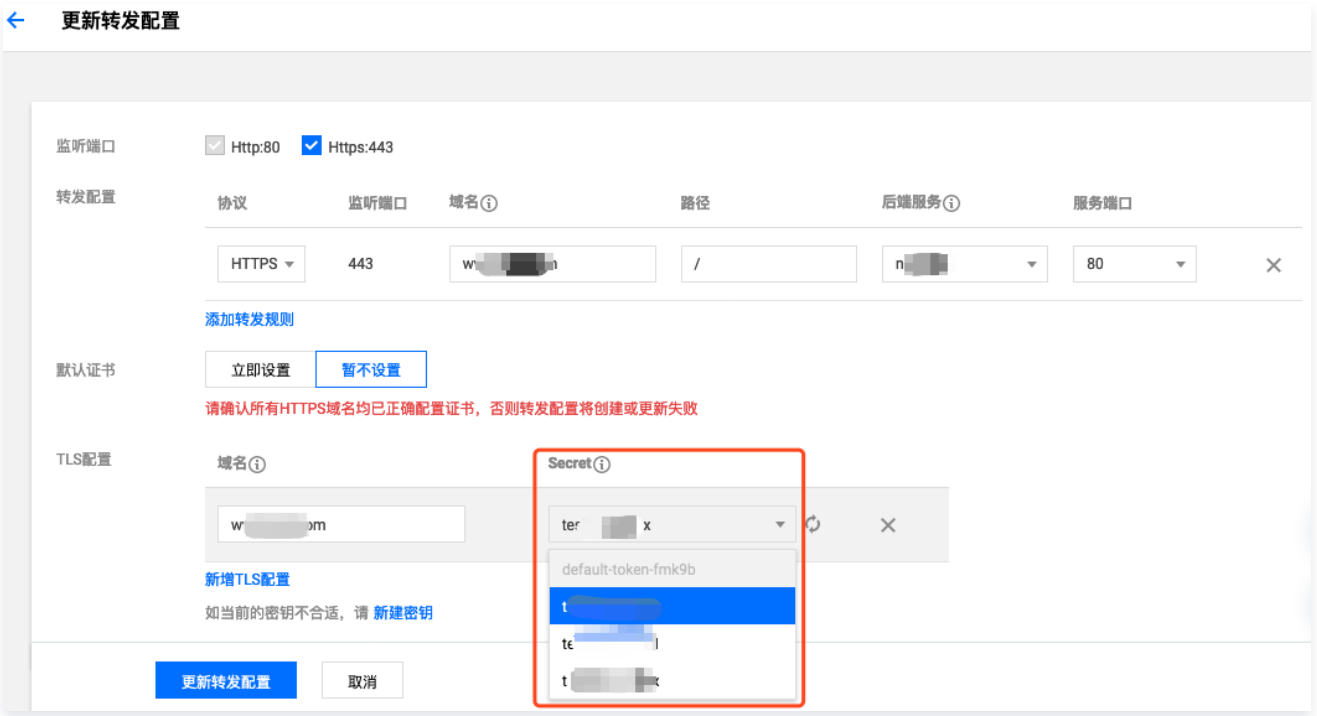
更新 Ingress

1. 在 容器服务控制台 找到 引用旧证书的 Ingress 中对应的 Ingress 资源，单击**更新转发配置**。如下图所示：

2. 在“更新转发配置”页面，为新证书**新建密钥**。如下图所示：

在“新建密钥”页面，选择新添加的证书，然后单击**创建Secret**。如下图所示：

返回至“更新转发配置”页面，修改 Ingress 的 TLS 配置，添加新创建的证书 Secret。如下图所示：



单击**更新转发配置**即可完成 Ingress 证书的续期。

使用 cert-manager 签发免费证书

最近更新时间：2023-09-04 14:49:12

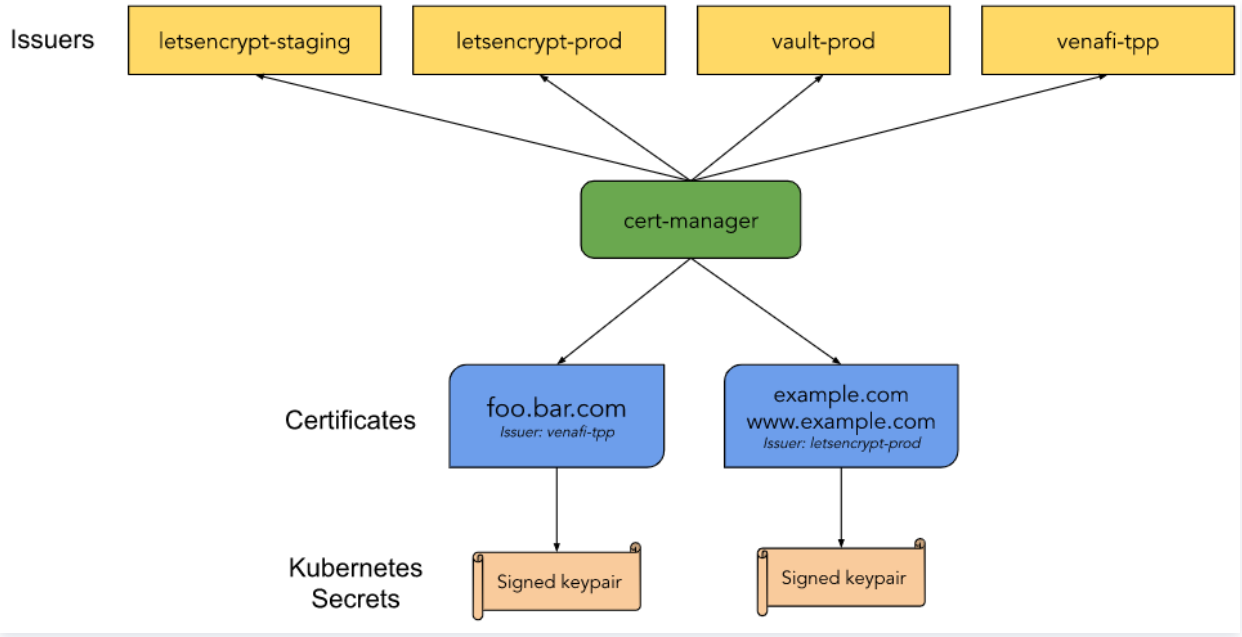
概述

随着 HTTPS 不断普及，大多数网站开始由 HTTP 升级到 HTTPS。使用 HTTPS 需要向权威机构申请证书，并且需要付出一定的成本，如果需求数量多，则开支也相对增加。cert-manager 是 Kubernetes 上的全能证书管理工具，支持利用 cert-manager 基于 ACME 协议与 Let's Encrypt 签发免费证书并为证书自动续期，实现永久免费使用证书。

操作原理

cert-manager 工作原理

cert-manager 部署到 Kubernetes 集群后会查阅其所支持的自定义资源 CRD，可通过创建 CRD 资源来指示 cert-manager 签发证书并为证书自动续期。如下图所示：



- **Issuer/ClusterIssuer**：用于指示 cert-manager 签发证书的方式，本文主要讲解签发免费证书的 ACME 方式。

> **说明**
> Issuer 与 ClusterIssuer 之间的区别是：Issuer 只能用来签发自身所在 namespace 下的证书，ClusterIssuer 可以签发任意 namespace 下的证书。

- **Certificate**：用于向 cert-manager 传递域名证书的信息、签发证书所需要的配置，以及对 Issuer/ClusterIssuer 的引用。

免费证书签发原理

Let’s Encrypt 利用 ACME 协议校验域名的归属，校验成功后可以自动颁发免费证书。免费证书有效期只有90天，需在到期前再校验一次实现续期。使用 cert-manager 可以自动续期，即实现永久使用免费证书。校验域名归属的两种方式分别是 **HTTP-01** 和 **DNS-01**，校验原理详情可参见 Let's Encrypt 的运作方式 。

HTTP-01 校验原理

HTTP-01 的校验原理是给域名指向的 HTTP 服务增加一个临时 location。此方法仅适用于给使用 Ingress 暴露流量的服务颁发证书，并且不支持泛域名证书。例如，Let’s Encrypt 会发送 HTTP 请求到 `http://<YOUR_DOMAIN>/.well-known/acme-challenge/<TOKEN>` 。`YOUR_DOMAIN` 是被校验的域名。`TOKEN` 是 ACME 协议客户端负责放置的文件，在此处 ACME 客户端即 cert-manager，通过修改或创建 Ingress 规则来增加临时校验路径并指向提供 `TOKEN` 的服务。Let’s Encrypt 会对比 `TOKEN` 是否符合预期，校验成功后就会颁发证书。

DNS-01 校验原理

DNS-01 的校验原理是利用 DNS 提供商的 API Key 拿到用户 DNS 控制权限。此方法不需要使用 Ingress，并且支持泛域名证书。在 Let’s Encrypt 为 ACME 客户端提供令牌后，ACME 客户端 `\(cert-manager\)` 将创建从该令牌和账户密钥派生的 TXT 记录，并将该记录放在 `_acme-challenge.<YOUR_DOMAIN>` 。Let’s Encrypt 将向 DNS 系统查询该记录，找到匹配项即可颁发证书。

校验方式对比

- HTTP-01 校验方式的优点是配置简单通用，不同 DNS 提供商均可使用相同的配置方法。缺点是需要依赖 Ingress，若仅适用于服务支持 Ingress 暴露流量，不支持泛域名证书。
- DNS-01 校验方式的优点是不依赖 Ingress，并支持泛域名。缺点是不同 DNS 提供商的配置方式不同，DNS 提供商过多而 cert-manager 的 Issuer 不能全部支持。部分可以通过部署实现 cert-manager 的 Webhook 服务来扩展 Issuer 进行支持。例如 DNSPod 和 阿里 DNS，详情请参见 Webhook 列表 。

本文向您推荐 `DNS-01` 方式，其限制较少，功能较全。

操作步骤

安装 cert-manager

通常使用 yaml 方式一键安装 cert-manager 到集群，可参考官网文档 Installing with regular manifests 。cert-manager 官方使用的镜像在 `quay.io` 进行拉取，在国内拉取镜像时您可以参考 境外镜像拉取加速 。

配置 DNS

登录 DNS 提供商后台，配置域名的 DNS A 记录，指向所需要证书的后端服务对外暴露的 IP 地址。以 cloudflare 为例，如下图所示：

HTTP-01 校验方式签发证书

若使用 HTTP-01 的校验方式，则需要用到 Ingress 来配合校验。cert-manager 会通过自动修改 Ingress 规则或自动新增 Ingress 来实现对外暴露校验所需的临时 HTTP 路径。为 Issuer 配置 HTTP-01 校验时，如果指定 Ingress 的 `name` ，表示会自动修改指定 Ingress 的规则来暴露校验所需的临时 HTTP 路径，如果指定 `class` ，则表示会自动新增 Ingress，可参考以下 示例 。
TKE 自带的 Ingress 中，每个 Ingress 资源都会对应一个负载均衡 CLB，如果使用 TKE 自带的 Ingress 暴露服务，并且使用 HTTP-01 方式校验，那么只能使用自动修改 Ingress 的方式，不能自动新增 Ingress。自动新增的 Ingress 会自动创建其他 CLB，使对外的 IP 地址与后端服务的 Ingress 不一致，Let's Encrypt 校验时将无法从服务的 Ingress 找到校验所需的临时路径，从而导致校验失败，无法签发证书。如果使用自建 Ingress，例如 在 TKE 上部署 Nginx Ingress，同一个 Ingress class 的 Ingress 共享同一个 CLB，则支持使用自动新增 Ingress 的方式。

示例

如果服务使用 TKE 自带的 Ingress 暴露服务，则不适合用 cert-manager 签发管理免费证书，证书从 证书管理 中被引用，不在 Kubernetes 中管理。假设是 在 TKE 上部署 Nginx Ingress，且后端服务的 Ingress 是 `prod/web` ，可参考以下代码示例创建 Issuer：

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-http01
  namespace: prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
```

```
    privateKeySecretRef:
      name: letsencrypt-http01-account-key
    solvers:
    - http01:
       ingress:
         name: web # 指定被自动修改的 Ingress 名称
```

使用 Issuer 签发证书，cert-manager 会自动创建 Ingress 资源，并自动修改 Ingress 的资源 `prod/web`，以暴露校验所需的临时路径。参考以下代码示例，自动新增 Ingress：

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-http01
  namespace: prod
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-http01-account-key
    solvers:
    - http01:
       ingress:
         class: nginx # 指定自动创建的 Ingress 的 ingress class
```

成功创建 Issuer 后，参考以下代码示例，创建 Certificate 并引用 Issuer 进行签发：

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: test-mydomain-com
  namespace: prod
spec:
  dnsNames:
  - test.mydomain.com # 要签发证书的域名
  issuerRef:
    kind: Issuer
    name: letsencrypt-http01 # 引用 Issuer，指示采用 http01 方式进行校验
  secretName: test-mydomain-com-tls # 最终签发出来的证书会保存在这个 Secret 里面
```

DNS-01 校验方式签发证书

若使用 DNS-01 的校验方式，则需要选择 DNS 提供商。cert-manager 内置 DNS 提供商的支持，详细列表和用法请参见 Supported DNS01 providers。若需要使用列表外的 DNS 提供商，可参考以下两种方案：

方案1：设置 Custom Nameserver

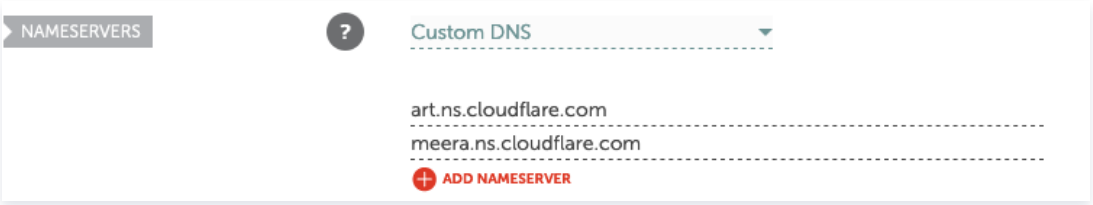
在 DNS 提供商后台设置 custom nameserver，指向例如 cloudflare 此类可管理其它 DNS 提供商域名的 nameserver 地址，具体地址可登录 cloudflare 后台查看。如下图所示：

Cloudflare nameservers

To use Cloudflare, ensure your authoritative DNS servers, or nameservers have been changed. These are your assigned Cloudflare nameservers.

Type	Value
NS	art.ns.cloudflare.com
NS	meera.ns.cloudflare.com

namecheap 可以设置 custom nameserver，如下图所示：



最后配置 Issuer 指定 DNS-01 验证时，添加 cloudflare 的信息即可。

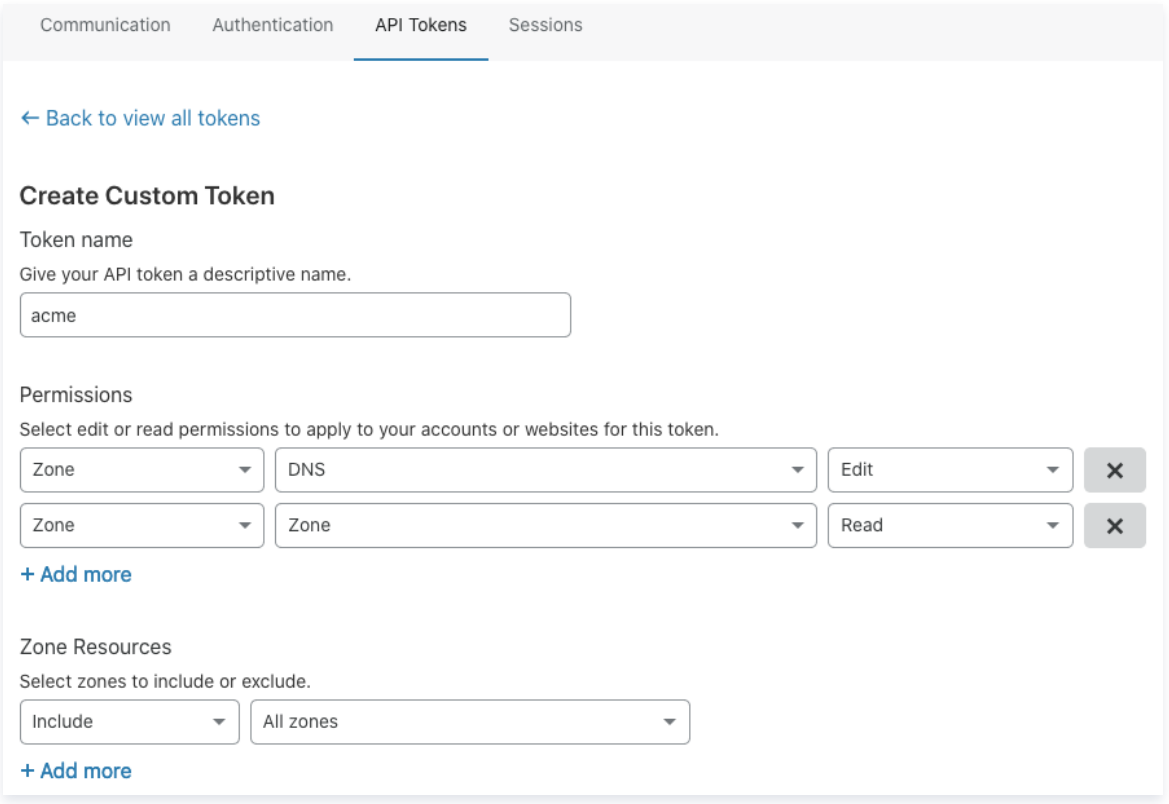
方案2：使用 Webhook

使用 cert-manager 的 Webhook 来扩展 cert-manager 的 DNS-01 验证所支持的 DNS 提供商，已经有许多第三方实现，包括国内常用的 DNSPod 与阿里 DNS，详细列表和用法请参见 Webhook 。

示例

参考以下步骤，以 cloudflare 为例来签发证书：

1. 登录 cloudflare，并创建 Token。如下图所示：



2. 复制 Token 并将 Token 保存到 Secret 中。yaml 示例如下：

> ⚠ 注意

- 如需创建 ClusterIssuer，Secret 需要创建在 cert-manager 所在命名空间中。
- 如需创建 Issuer，Secret 需要创建在 Issuer 所在命名空间中。

```
apiVersion: v1
kind: Secret
metadata:
  name: cloudflare-api-token-secret
  namespace: cert-manager
type: Opaque
stringData:
  api-token: <API Token> # 将 Token 粘贴到此处，不需要 base64 加密。
```

3. 创建 ClusterIssuer。yaml 示例如下：

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-dns01
spec:
  acme:
    privateKeySecretRef:
      name: letsencrypt-dns01
    server: https://acme-v02.api.letsencrypt.org/directory
    solvers:
    - dns01:
        cloudflare:
          email: my-cloudflare-acc@example.com # 替换成你的 cloudflare 邮箱账号，API Token 方式认证非必需，API Keys 认证是必需
          apiTokenSecretRef:
            key: api-token
            name: cloudflare-api-token-secret # 引用保存 cloudflare 认证信息的 Secret
```

4. 创建 Certificate。yaml 示例如下：

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: test-mydomain-com
  namespace: default
spec:
  dnsNames:
  - test.mydomain.com # 要签发证书的域名
  issuerRef:
    kind: ClusterIssuer
    name: letsencrypt-dns01 # 引用 ClusterIssuer，指示采用 dns01 方式进行校验
  secretName: test-mydomain-com-tls # 最终签发出来的证书会保存在这个 Secret 里面
```

获取和使用证书

创建 Certificate 后，即可通过 kubectl 查看证书是否签发成功。

```
$ kubectl get certificate -n prod
```

```
NAME                READY   SECRET                  AGE
test-mydomain-com   True    test-mydomain-com-tls   1m
```

- `READY` 为 `False`：则表示签发失败，可以通过 describe 命令查看 event 来排查失败原因。

  ```
  $ kubectl describe certificate test-mydomain-com -n prod
  ```

- `READY` 为 `True`：则表示签发成功，证书将保存在所指定的 Secret 中。例如，`default/test-mydomain-com-tls`。可以通过 kubectl 查看，其中 `tls.crt` 是证书，`tls.key` 是密钥。

  ```
  $ kubectl get secret test-mydomain-com-tls -n default
  ...
  data:
  tls.crt: <cert>
  tls.key: <private key>
  ```

您可以将其挂载到需要证书的应用中，或者直接在自建的 Ingress 中引用 secret。可参考以下示例：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/Ingress.class: nginx
spec:
  rules:
  - host: test.mydomain.com
    http:
      paths:
      - path: /web
        backend:
          serviceName: web
          servicePort: 80
  tls:
    hosts:
    - test.mydomain.com
    secretName: test-mydomain-com-tls
```

相关文档

- cert-manager 官网
- Let's Encrypt 的运作方式
- Issuer API 文档
- Certificate API 文档

使用 cert-manager 为 DNSPod 的域名签发免费证书

最近更新时间：2025-06-25 16:46:31

概述

如果您的域名使用 DNSPod 管理，并希望在 Kubernetes 上为域名自动签发免费证书，可以通过使用 cert-manager 来实现。
如果域名通过 DNSPod 管理，由于 cert-manager 自身并未实现 DNSPod 的 provider，但提供了 webhook 扩展机制，因此通过 cert-manager-webhook-dnspod 可以实现为 DNSPod 上的域名自动签发免费证书与续期。

操作步骤

创建 cert-manager 命名空间

请确保已创建 cert-manager 命名空间。如果没有该命名空间，请执行以下命令创建：

```
kubectl create ns cert-manager
```

安装 cert-manager

确保 cert-manager 已安装到集群，且版本 >= 1.13.0，可通过 TKE 应用市场 安装，命名空间选 `cert-manager` 。

安装 cert-manager-webhook-dnspod

在 TKE 应用市场 中搜索 `cert-manager-webhook-dnspod` 并安装到 `cert-manager` 命名空间。
如需自定义安装，也可以参考 cert-manager-webhook-dnspod 文档 使用 Helm 安装。

创建腾讯云 API 密钥

登录腾讯云控制台，在 API 密钥管理 中新建密钥，然后复制自动生成的 `SecretId` 和 `SecretKey` 并保存下来，以备后面的步骤使用。
要求账号至少具有以下权限：

```
{
    "version": "2.0",
    "statement": [
        {
            "effect": "allow",
            "action": [
                "dnspod:CreateRecord",
                "dnspod:DescribeRecordList",
                "dnspod:DeleteRecord",
            ],
            "resource": [
                "*"
            ]
        }
    ]
}
```

创建 Secret

在 cert-manager 所在命名空间中创建一个 `Secret` 对象，用于保存前面创建的腾讯云 API 密钥：

```
apiVersion: v1
kind: Secret
metadata:
  name: dnspod-secret
  namespace: cert-manager
type: Opaque
stringData:
```

```
  secretId: xxx
  secretKey: xxx
```

> **注意：**
> 请将 xxx 替换为您的 SecretId 和 SecretKey。

创建 ClusterIssuer

创建一个 DNSPod 的 `ClusterIssuer` 对象，用于为 DNSPod 管理的域名签发免费证书：

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: dnspod
spec:
  acme:
    email: your-email-address@example.com
    privateKeySecretRef:
      name: dnspod-letsencrypt
    server: https://acme-v02.api.letsencrypt.org/directory
    solvers:
      - dns01:
          webhook:
            config:
              secretIdRef:
                key: secretId
                name: dnspod-secret
              secretKeyRef:
                key: secretKey
                name: dnspod-secret
              ttl: 600
              recordLine: ""
            groupName: acme.dnspod.com
            solverName: dnspod
```

> **注意：**
> `email` 可替换成您的邮箱地址，用于接收来自 Let's Encrypt 的证书过期提醒（只有没正常自动续期的情况才会有通知）。

创建 Certificate

创建 `Certificate` 对象来签发您想要的免费证书：

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: example-crt
  namespace: istio-system
spec:
  secretName: example-crt-secret # 证书签发后会保存到这个 Secret 中
  issuerRef:
    name: dnspod # 这里引用前面创建的 ClusterIssuer 名称
    kind: ClusterIssuer
    group: cert-manager.io
  dnsNames: # 填入需要签发证书的域名列表，支持泛域名，确保域名是使用当前账号下的 dnspod 管理的
  - "example.com"
  - "*.example.com"
```

等待 Ready 变为 True 表示签发成功：

```
$ kubectl -n istio-system get certificates.cert-manager.io
NAME          READY   SECRET               AGE
example-crt   True    example-crt-secret   25d
```

若签发失败，可通过 describe 查看详细原因：

```
kubectl -n istio-system describe certificates.cert-manager.io example-crt
```

使用证书

证书签发成功后会保存到我们指定的 Secret 中，下面给出一些使用示例。

在 Ingress 中使用

如果集群中安装了 Ingress controller（如 Nginx Ingress），可在 `Ingress` 中引用证书 Secret：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: test.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web
          servicePort: 80
  tls:
    hosts:
    - test.example.com
    secretName: example-crt-secret # 引用证书 secret
```

在 Istio 的 IngressGateway 中使用

如果集群中安装了 Istio，可在 `Gateway` 中引用证书 Secret：

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: example-gw
  namespace: istio-system
spec:
  selector:
    app: istio-ingressgateway
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: HTTP-80
      protocol: HTTP
    hosts:
    - example.com
    - "*.example.com"
```

```
    tls:
      httpsRedirect: true
  - port:
      number: 443
      name: HTTPS-443
      protocol: HTTPS
    hosts:
    - example.com
    - "*.example.com"
    tls:
      mode: SIMPLE
      credentialName: example-crt-secret # 引用证书 secret
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: example-vs
  namespace: test
spec:
  gateways:
  - istio-system/example-gw # 将转发规则应用到指定 IngressGateway
  hosts:
  - 'test.example.com'
  http:
  - route:
    - destination:
        host: example
        port:
          number: 80
```

在 Gateway API 中使用

如果集群中安装了 Gateway API 的实现（如 Envoy Gateway），在定义 `Gateway` 时可引用证书 Secret：

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: eg
  namespace: envoy-gateway-system
spec:
  gatewayClassName: eg
  listeners:
  - allowedRoutes:
      namespaces:
        from: All
    name: https
    port: 443
    protocol: HTTPS
    tls:
      certificateRefs:
      - group: ""
        kind: Secret
        name: example-crt-secret # 引用证书 secret
      mode: Terminate
  - allowedRoutes:
      namespaces:
        from: All
    name: http
    port: 80
    protocol: HTTP
```

```
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example
  namespace: test
spec:
  hostnames:
  - example.com
  parentRefs: # 将转发规则应用到指定 Gateway
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: eg
    namespace: envoy-gateway-system
    sectionName: https
  rules:
  - backendRefs:
    - group: ""
      kind: Service
      name: website
      port: 80
      weight: 1
    matches:
    - path:
        type: PathPrefix
        value: /
```

注意事项

Private DNS 与 DNSPod 的 zone 需保持一致

如果满足以下条件：

1. 使用了 Private DNS 管理域名解析。
2. Private DNS 关联了当前 TKE 集群所在 VPC。
3. Private DNS 中也配置了待签发免费证书的域名的解析。
4. DNSPod 中有将子域名独立出来（单独成 zone）管理域名解析。

则必须确保 Private DNS 中也要有相同的 zone 配置，即也将相同子域名独立出来管理解析，在 Private DNS 控制台点击**新建私有域**，输入与 DNSPod 中相同的子域名。

这是因为在签发证书时，cert-manager 会通过 SOA 查询要签发的域名的 zone，在有 Private DNS 的环境中，如果 Private DNS 里也配置了相同的域名，会优先根据 Private DNS 中的配置返回，如果与 DNSPod 中配置的 zone 不一致，最终校验就会失败，导致一直无法签发成功。

使用 TKE NPDPlus 插件增强节点的故障自愈能力

最近更新时间：2023-11-01 14:34:53

在 Kubernetes 集群运行时，节点有时会因为组件问题、内核死锁、资源不足等原因不可用。Kubelet 默认对节点的 PIDPressure、MemoryPressure、DiskPressure 等资源状态进行监控，但是存在当 Kubelet 上报状态时节点已处于不可用状态的情况，甚至 Kubelet 可能已开始驱逐 Pod。在此类场景下，原生 Kubernetes 对节点健康的检测机制是不完善的，为了提前发现节点的问题，需要添加更加细致化的指标来描述节点的健康状态，实现智能运维，以节省开发和减轻运维人员的负担。

node-problem-detector 介绍

NPD（node-problem-detector）是 Kubernetes 社区开源的集群节点的健康检测组件。NPD 提供了通过正则匹配系统日志或文件来发现节点异常的功能。用户可以通过运维经验，配置可能产生异常问题日志的正则表达式，选择不同的上报方式。NPD 会解析用户的配置文件，当有日志能匹配到用户配置的正则表达式时，可以通过 NodeCondition、Event 或 Prometheus Metric 等方式将检测到的异常状态上报。除了日志匹配功能，NPD 还接受用户自行编写的自定义检测插件，用户可以开发自己的脚本或可执行文件集成到 NPD 的插件中，让 NPD 定期执行检测程序。

TKE NPDPlus 组件介绍

在 TKE 中通过扩展组件的形式集成了 NPD，并且对 NPD 的能力做了增强，称为 NodeProblemDetectorPlus（NPDPlus）扩展组件。用户可以对已有集群一键部署 NPDPlus 扩展组件，也可以在创建集群的时候同时部署 NPDPlus。TKE 提取了可以通过特定形式发现节点异常的指标，并将其集成在 NPDPlus 中。例如，可以在 NPDPlus 容器中检测 Kubelet 和 Docker 的 systemd 状态，以及检测主机的文件描述符和线程数压力等。
TKE 使用 NPDPlus 是为了能够提前发现节点的不可用状态，而不是当节点已经不健康后再上报状态。当用户在 TKE 集群中部署了 NPDPlus 后，使用命令 `kubectl describe node` 后会出现一些 Node Condition，例如，FDPressure 表示该节点上已经使用的文件描述符数量是否已经达到机器允许最大值的80%。ThreadPressure 表示节点上的线程数是否已经达到机器允许的90%等。用户可以监控这些 Condition，当异常状态出现时，提前采取规避策略。详情请参见 Node Conditions 。
同时，Kubernetes 目前认为节点 NotReady 的机制依赖于 kube-controller-manager 的参数设定，当节点网络完全不通的情况下，Kubernetes 很难在秒级别发现节点的异常。在一些场景下，例如直播、在线会议等，这种延迟是不能接受的。为了解决这个问题，NPDPlus 引入了分布式节点健康检测功能，该功能可以在秒级别快速地检测节点的网络状态，并判断节点是否能够在不依赖于 Kubernetes master 组件通信的情况下，与其他节点相互通信。TKE NPDPlus 组件使用详情请参见 NodeProblemDetectorPlus 使用方法 。

使用 kubecm 管理多集群 kubeconfig

最近更新时间：2023-05-17 15:41:07

操作场景

Kubernetes 提供 Kubectl 命令行工具用于操作集群，Kubectl 使用 Kubeconfig 作为配置文件（默认路径为 `~/.kube/config`），通过其配置多个集群的信息，并管理和操作多个集群。
通过 Kubectl 管理和操作容器服务 TKE 或 TKE Serverless 集群，需要在集群基本信息页面开启 APIServer 的外网访问或内网访问，获取 Kubeconfig（集群访问凭证）。如果需要使用 Kubectl 管理多个集群，通常做法是提取 Kubeconfig 中各个字段的内容，将其合并到 Kubectl 所在设备的 Kubeconfig 文件中，但该方式操作繁琐且容易出错。
借助 kubecm 工具，可以更简单高效的将多个集群访问凭证合并添加到 kubeconfig 中。本文将介绍如何利用 kubecm 实现多集群的 kubeconfig 高效管理。

前提条件

- 已创建 TKE 标准集群 或 TKE Serverless 集群。
- 已在需要管理多集群的设备上安装 kubectl 命令行工具。

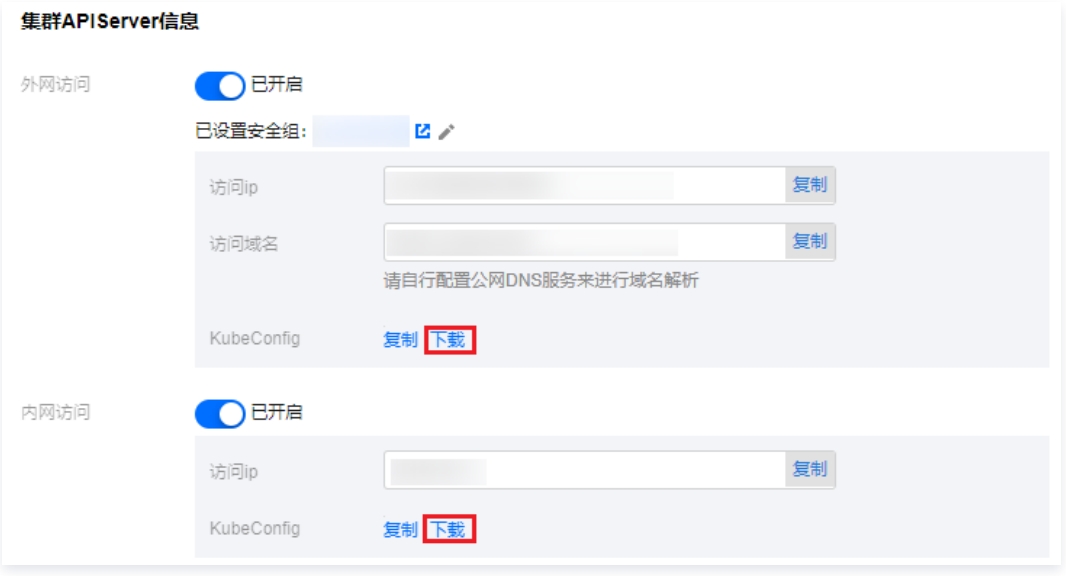
操作步骤

安装 kubecm

在管理多集群的设备上安装 Kubecm。

获取集群访问凭证

创建集群后，请按照以下步骤获取集群访问凭证：
1. 登录 容器服务控制台，单击左侧导航栏中的**集群**。
2. 单击需要获取集群访问凭证的集群 ID/名称，进入该集群的基本信息页面。
3. 在“基本信息”页面找到**集群APIServer信息**配置项，开启**外网访问**和**内网访问**。
4. 单击 Kubeconfig 右侧的**下载**即可。如下图所示：



使用 Kubecm 添加访问凭证到 Kubeconfig

本文以集群访问凭证文件名 `cls-l6whmzi3-config` 为例，执行以下命令，使用 Kubecm 将访问凭证添加到 Kubeconfig 中（`-n` 可指定 context 名称）。示例如下：

```
kubecm add -f cls-l6whmzi3-config -n cd -c
```

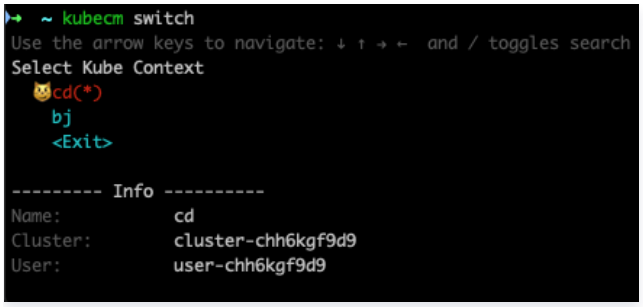
查看集群列表

执行以下 `kubecm ls` 命令查看 kubeconfig 中的集群列表（星号标识的是当前操作的集群）。示例如下：

```
$ kubecm ls
+------------+------------+----------------------+--------------------+--------------------------------
--+--------------------+
|   CURRENT  |    NAME    |        CLUSTER       |        USER        |               SERVER
|     Namespace      |
+============+============+======================+====================+================================
==+====================+
|      *     |     cd     |   cluster-chh6kgf9d9 |   user-chh6kgf9d9  |   https://cls-l6whmzi3.ccs.tence
|      default       |
|            |            |                      |                    |            nt-cloud.com
|                    |
+------------+------------+----------------------+--------------------+--------------------------------
--+--------------------+
|            |     bj     |   cluster-6qaua96n   |   user-6qaua96n    |   https://cls-6qaua96n.ccs.tence
|    kube-system     |
|            |            |                      |                    |            nt-cloud.com
|                    |
+------------+------------+----------------------+--------------------+--------------------------------
--+--------------------+
```

切换集群

执行以下 `kubecm switch` 命令可以交互式切换到其他集群。如下图所示：



移除集群

执行以下 `kubecm delete` 命令可以移除某个集群。示例如下：

```
$ kubecm delete bj
Context Delete:「bj」
「/Users/roc/.kube/config」 write successful!
+------------+---------+----------------------+--------------------+----------------------------------
+--------------+
|   CURRENT  |   NAME  |        CLUSTER       |        USER        |               SERVER               |
Namespace  |
+============+=========+======================+====================+==================================+
==============+
|            |    cd   |   cluster-chh6kgf9d9 |   user-chh6kgf9d9  |   https://cls-l6whmzi3.ccs.tence   |
default    |
|            |         |                      |                    |            nt-cloud.com            |
|
+------------+---------+----------------------+--------------------+----------------------------------
+--------------+
```

参考文档

- kubecm 开源地址

- kubecm 官方文档

使用 TKE 审计和事件服务快速排查问题

最近更新时间：2023-11-10 17:39:23

使用场景

容器服务 TKE 的集群审计和事件存储为用户配置了丰富的可视化图表，以多个维度对审计日志和集群事件进行呈现，操作简单且涵盖绝大多数常见集群运维场景，易于发现和定位问题，提升运维效率，将审计和事件数据的价值最大化。本文结合几个具体使用场景和示例，介绍如何利用审计和事件仪表盘快速定位集群问题。

前提条件

已登录 容器服务控制台，并已开启 集群审计 和 事件存储。

使用示例

示例1：排查工作负载消失问题

1. 登录 容器服务控制台。
2. 在左侧导航栏中选择**日志管理** > **审计日志**，进入**审计检索**页面。
3. 选择**K8S 对象操作概览**页签，在过滤项中指定需要排查的操作类型和资源对象，如下图所示：

4. 查询结果如下图所示：

由图可见，`10001****7138` 账号在 `2020-11-30T03:37:13` 时删除了 `nginx` 应用。可根据账号 ID 在**访问管理** > 用户列表 中查找关于此账号的详细信息。

示例2：排查节点被封锁问题

1. 登录 容器服务控制台。
2. 在左侧导航栏中选择**日志管理** > **审计日志**，进入**审计检索**页面。
3. 选择**节点操作概览**页签，在过滤项中指定被封锁的节点名称，如下图所示：

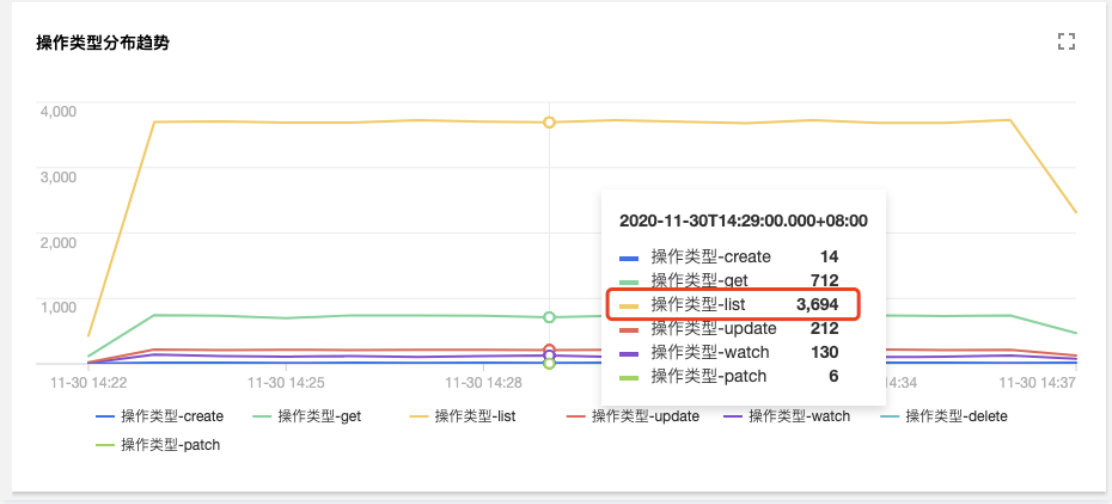
4. 单击**过滤**开始查询，查询结果如下图所示：

由图可见，`10001****7138` 账号在 `2020-11-30T06:22:18` 时对 `172.16.18.13` 节点进行了封锁操作。

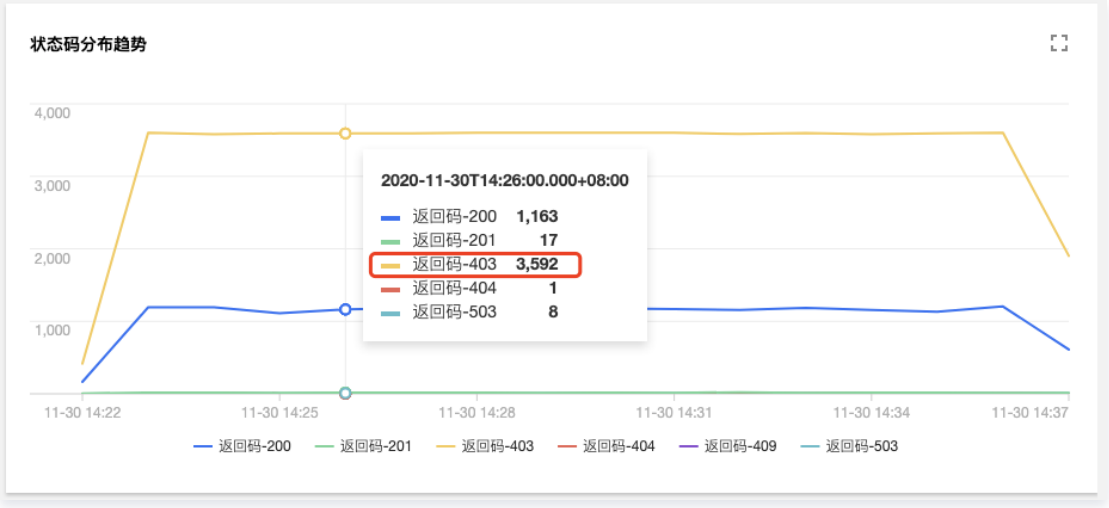
示例3：排查 apiserver 响应变慢问题

1. 登录 容器服务控制台 。
2. 在左侧导航栏中选择**日志管理** > **审计日志**，进入**审计检索**页面。
3. 选择**聚合检索**页签，进入**聚合检索**页面。该页面提供了**用户**、**操作类型**、**返回状态码**等多个维度对于 apiserver 访问的趋势图。如下图所示：
   - **操作用户分布趋势：**

   - **操作类型分布趋势：**



- **状态码分布趋势：**



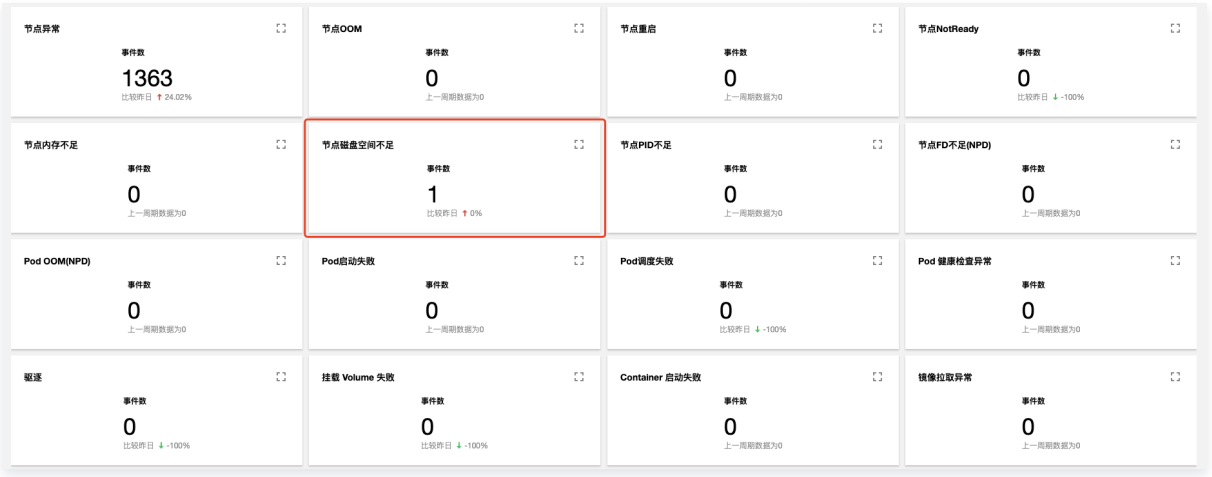
由图可见，用户 `tke-kube-state-metrics` 的访问量远高于其他用户，并且在 操作类型分布趋势 图中可以看出大多数为 list 操作，在**状态码分布趋势**图中可以看出，状态码大多数为403。结合业务日志可知，由于 RBAC 鉴权问题导致 `tke-kube-state-metrics` 组件不停的请求 apiserver 重试，导致 apiserver 访问剧增。日志示例如下：

```
E1130 06:19:37.368981       1 reflector.go:156] pkg/mod/k8s.io/client-go@v0.0.0-20191109102209-3c0d1af94be5/tools/cache/reflector.go:108: Failed to list *v1.VolumeAttachment: volumeattachments.storage.k8s.io is forbidden: User "system:serviceaccount:kube-system:tke-kube-state-metrics" cannot list resource "volumeattachments" in API group "storage.k8s.io" at the cluster scope
```

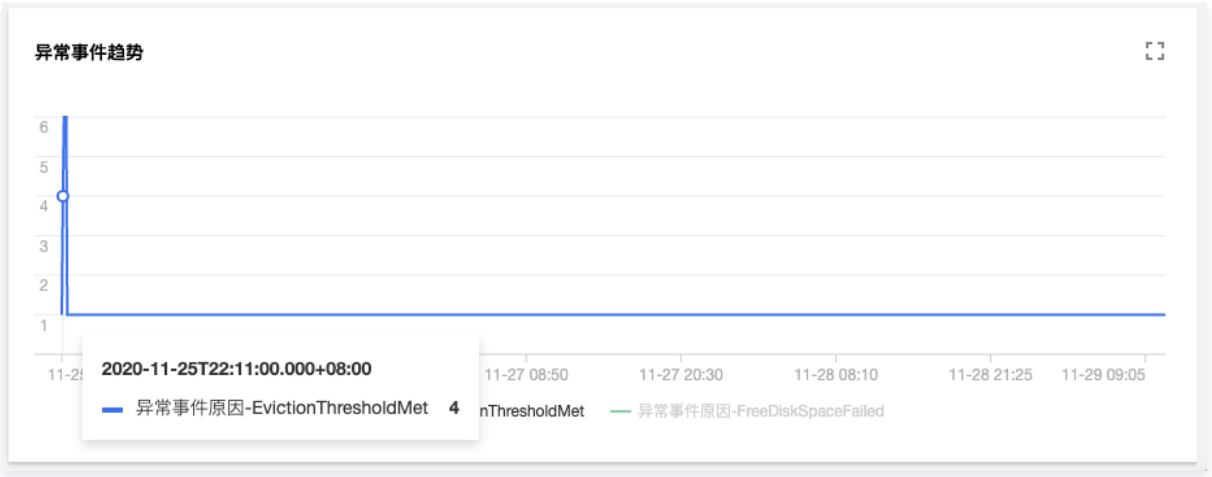
示例4：排查节点异常问题

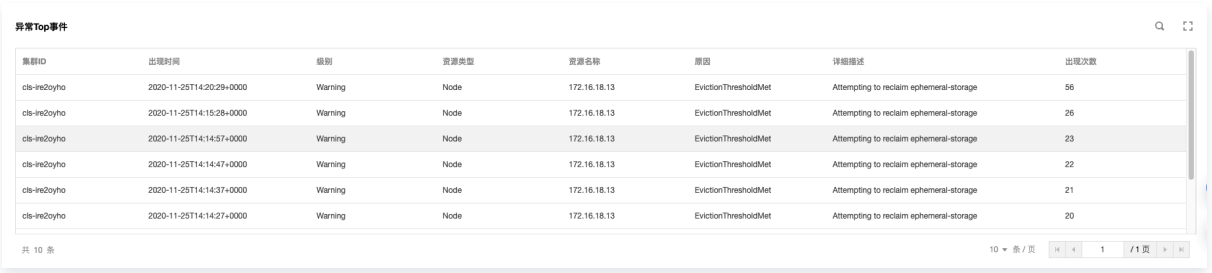
1. 登录 容器服务控制台 。
2. 在左侧导航栏中，选择**日志管理** > **事件日志**，进入**事件检索**页面。
3. 选择**事件总览**页签，在资源对象过滤项中输入异常节点 IP，如下图所示：

4. 单击**过滤**开始查询。查询结果显示，有一条“节点磁盘空间不足”的事件记录查询结果，如下图所示：



5. 单击该事件，进一步查看异常事件趋势。





由图可见，从 `2020-11-25` 开始，节点 `172.16.18.13` 由于磁盘空间不足导致节点异常，此后 kubelet 开始尝试驱逐节点上的 Pod 以回收节点磁盘空间。

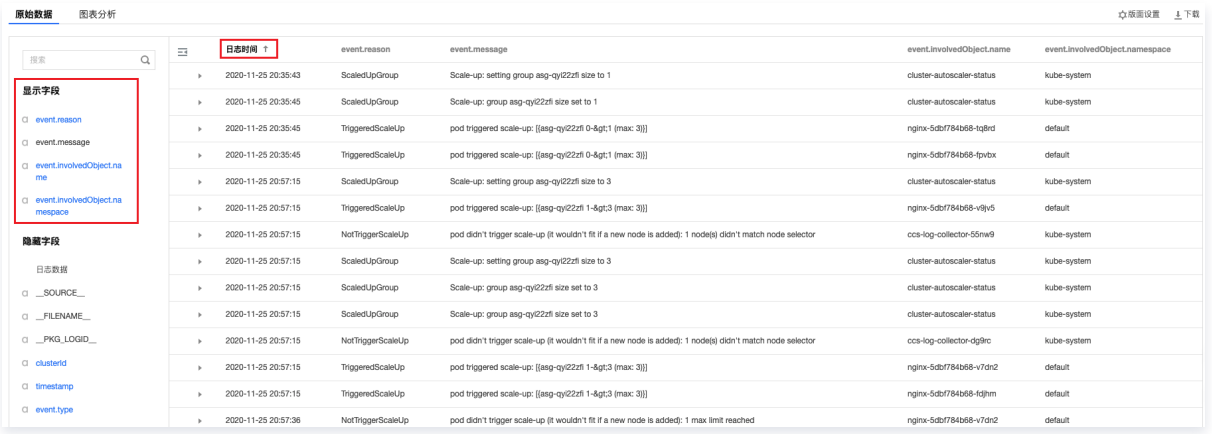
示例5：查找触发节点扩容的原因

开启了节点池**弹性伸缩**的集群，CA（cluster-autoscaler）组件会根据负载状况自动对集群中节点数量进行增减。如果集群中的节点发生了自动扩（缩）容，用户可通过事件检索对整个扩（缩）容过程进行回溯。

1. 登录 容器服务控制台 。
2. 在左侧导航栏中，选择**日志管理 > 事件日志**，进入**事件检索**页面。
3. 选择**全局检索**页签，在检索分析栏中输入以下检索命令：

```
event.source.component : "cluster-autoscaler"
```

4. 在左侧“隐藏字段”中选择“`event.reason`”、“`event.message`”、“`event.involvedObject.name`”、“`event.involvedObject.name`”进行显示，单击**检索分析**开始检索分析日志并将返回检索结果。
5. 将检索结果按照“日志时间”倒序排列，如下图所示：



由图可见，通过事件可以看到节点扩容操作在 `2020-11-25 20:35:45` 左右，分别由三个 `nginx pod(nginx-5dbf784b68-tq8rd、nginx-5dbf784b68-fpvbx、nginx-5dbf784b68-v9jv5)` 进行触发，最终扩增三个节点，后续的扩容

由于达到节点池的最大节点数未再次触发。

在 TKE 中自定义 RBAC 授权

最近更新时间：2024-02-18 11:16:21

容器服务 TKE 支持通过在控制台使用**授权管理**功能管理子账号的常用授权，也可以使用自定义 YAML 的方式（ RBAC 授权 ）来满足更加个性化的授权需求，Kubernetes RBAC 授权说明和原理如下：

- **权限对象（Role 或 ClusterRole）：** 权限对象使用 apiGroups、resources 和 verbs 来定义权限情况。其中：
  - Role 权限对象：作用于特定命名空间。
  - ClusterRole 权限对象：可复用于多个命名空间授权（ Rolebinding ）或为整个集群授权（ ClusterRoleBinding ）。
- **授权对象（Subjects）：** 权限授予的主体对象，分别为 User、Group 和 ServiceAccount 三种类型主体。
- **权限绑定（Rolebinding 或 ClusterRoleBinding）：** 将权限对象和授权对象进行组合绑定。其中：
  - Rolebinding：作用于某个命名空间。
  - ClusterRoleBinding：作用于整个集群。

Kubernetes RBAC 授权主要提供以下4种常用权限绑定方式，本文将为您分别介绍如何使用这4种权限绑定方式实现对用户的授权管理。

方式	说明
方式1：作用于单个命名空间的权限绑定	RoleBinding 引用 Role 对象，为 Subjects 只授予某单个命名空间下资源权限。
方式2：多个命名空间复用集群权限对象绑定	多个命名空间下不同的 Rolebinding 可引用同一个 ClusterRole 对象模板为 Subjects 授予相同模板权限。
方式3：整个集群权限的绑定	ClusterRoleBinding 引用 ClusterRole 模板，为 Subjects 授予整个集群的权限。
方式4：自定义权限	用户自定义权限，例如给一个用户预设的只读权限额外添加登录容器的权限。

> **说明**
> 除上述方式之外，从 Kubernetes RBAC 1.9版本开始，集群角色（ ClusterRole ）还可通过使用 aggregationRule 组合其他 ClusterRoles 的方式进行创建，本文不作详细介绍，您可参见官网文档 Aggregated ClusterRoles 说明。

方式1：作用于单个命名空间的权限绑定

此方式主要用于为某一个用户绑定某一个命名空间下的相关权限，适用于需要细化权限的场景。例如，开发、测试、运维人员只能在各自的命名空间下对资源操作。以下将为您介绍如何在 TKE 中实现作用于单个命名空间的权限绑定。

1. 使用以下 Shell 脚本，创建测试命名空间、ServiceAccount 类型的测试用户并设置集群访问凭证（ token ）认证 。示例如下：

```
USERNAME='sa-acc' # 设置测试账户名
NAMESPACE='sa-test' # 设置测试命名空间名
CLUSTER_NAME='cluster_name_xxx' # 设置测试集群名
# 创建测试命名空间
kubectl  create namespace ${NAMESPACE}
# 创建测试 ServiceAccount 账户
kubectl create sa ${USERNAME} -n ${NAMESPACE}
# 获取 ServiceAccount 账户自动创建的 Secret token 资源名
SECRET_TOKEN=$(kubectl get sa ${USERNAME} -n ${NAMESPACE} -o jsonpath='{.secrets[0].name}')
# 获取 secrets 的明文 Token
SA_TOKEN=$(kubectl get secret ${SECRET_TOKEN} -o jsonpath={.data.token} -n sa-test | base64 -d)
# 使用获取到的明文 token 信息设置一个 token 类型的访问凭证
kubectl config set-credentials ${USERNAME} --token=${SA_TOKEN}
# 设置访问集群所需要的 context 条目
kubectl config set-context ${USERNAME} --cluster=${CLUSTER_NAME} --namespace=${NAMESPACE} --
user=${USERNAME}
```

2. 执行 `kubectl config get-contexts` 命令，查看生成的 contexts 条目。如下图所示：

```
root@VM-0-13-ubuntu:/home/ubuntu# kubectl config get-contexts
CURRENT   NAME                                   CLUSTER         AUTHINFO        NAMESPACE
*         cls-i                -context-default   cls-i           1C
          sa-acc                                  cls-i           sa-acc          sa-test
```

3. 创建一个 Role 权限对象资源 sa-role.yaml 文件。示例如下：

```
kind: Role
```

```
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: sa-test # 指定 Namespace
  name: sa-role-test
rules: # 设置权限规则
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

4. 创建一个 RoleBinding 对象资源 sa-rb-test.yaml 文件。如下权限绑定表示，添加 ServiceAccount 类型的 sa-acc 用户在 sa-test 命名空间具有 sa-role-test（Role 类型）的权限。示例如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sa-rb-test
  namespace: sa-test
subjects:
- kind: ServiceAccount
  name: sa-acc
  namespace: sa-test # ServiceAccount 所在 Namespace
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: sa-role-test
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
```

5. 从下图验证结果可以得出，当 Context 为 sa-context 时，默认命名空间为 sa-test，且拥有 sa-test 命名空间下 sa-role-test（Role）对象中配置的权限，但在 default 命名空间下不具有任何权限。

```
root@VM-0-13-ubuntu:/home/ubuntu# kubectl get pod --context=sa-acc
No resources found in sa-test namespace.
root@VM-0-13-ubuntu:/home/ubuntu# kubectl run nginx --image=nginx -n sa-test  --context=sa-acc
pod/nginx created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl get pod --context=sa-acc
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          8s
root@VM-0-13-ubuntu:/home/ubuntu# kubectl run nginx --image=nginx -n default  --context=sa-acc
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:sa-test:sa-acc" cannot create resource "pods" in API group "" in the namespace "default"
```

方式2：多个命名空间复用集群权限对象绑定

此方式主要用于为用户授予多个命名空间下相同的权限，适用于使用一个权限模板为多个命名空间绑定授权的场景，例如需要为 DevOps 人员在多个命名空间绑定相同资源操作的权限。以下将为您介绍如何在 TKE 中使用多个命名空间复用集群权限绑定授权。

1. 使用以下 Shell 脚本，创建使用 X509 自签证书认证的用户、证书签名请求（CSR）和证书审批允许信任并设置集群资源访问凭证 Context。示例如下：

```
USERNAME='role_user' # 设置需要创建的用户名
NAMESPACE='default' # 设置测试命名空间名
CLUSTER_NAME='cluster_name_xxx' # 设置测试集群名
# 使用 Openssl 生成自签证书 key
openssl genrsa -out ${USERNAME}.key 2048
# 使用 Openssl 生成自签证书CSR 文件, CN 代表用户名，O 代表组名
openssl req -new -key ${USERNAME}.key -out ${USERNAME}.csr -subj "/CN=${USERNAME}/O=${USERNAME}"
# 创建 Kubernetes 证书签名请求（CSR）
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
   name: ${USERNAME}
spec:
   request: $(cat ${USERNAME}.csr | base64 | tr -d '\n')
   usages:
```

```
    - digital signature
    - key encipherment
    - client auth
EOF
# 证书审批允许信任
kubectl certificate approve ${USERNAME}
# 获取自签证书 CRT
kubectl get csr ${USERNAME} -o jsonpath={.status.certificate} | base64 --decode > ${USERNAME}.crt
# 设置集群资源访问凭证（X509 证书）
kubectl config set-credentials ${USERNAME} --client-certificate=${USERNAME}.crt --client-
key=${USERNAME}.key
# 设置 Context 集群、默认Namespace 等
kubectl config set-context  ${USERNAME} --cluster=${CLUSTER_NAME} --namespace=${NAMESPACE} --
user=${USERNAME}
```

2. 创建一个 ClusterRole 对象资源 test-clusterrole.yaml 文件。示例如下：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: test-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list", "create"]
```

3. 创建一个 RoleBinding 对象资源 clusterrole-rb-test.yaml 文件，如下权限绑定表示，添加自签证书认证类型的 role_user 用户在 default 命名空间具有 test-clusterrole（ClusterRole 类型）的权限。示例如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: clusterrole-rb-test
  namespace: default
subjects:
- kind: User
  name: role_user
  namespace: default # User 所在 Namespace
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: test-clusterrole
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
```

4. 从下图验证结果可以得出，当 Context 为 role_user 时，默认命名空间为 default，且拥有 test-clusterrole 权限对象配置的规则权限。

```
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod --context=role_user
No resources found in default namespace.
root@VM-0-13-ubuntu:/home/ubuntu# kubectl run nginx --image=nginx --context=role_user
pod/nginx created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod --context=role_user
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          4s
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  delete pod nginx --context=role_user
Error from server (Forbidden): pods "nginx" is forbidden: User "role_user" cannot delete resource "pods" in API group "" in the namespace "default"
```

5. 创建第二个 RoleBinding 对象资源 clusterrole-rb-test2.yaml 文件，如下权限绑定表示，添加自签证书认证类型的 role_user 用户在 default2 命名空间具有 test-clusterrole（ClusterRole 类型）的权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
```

```
metadata:
  name: clusterrole-rb-test
  namespace: default2
subjects:
- kind: User
  name: role_user
  namespace: default # User 所在 Namespace
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: test-clusterrole
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
```

6. 从下图验证结果可以得出，在 default2 命名空间下， role_user 同样拥有 test-clusterrole 配置的规则权限。至此通过上述步骤实现了多个命名空间复用集群权限的绑定。

```
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  create namespace default2
namespace/default2 created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod -n default2 --context=role_user
Error from server (Forbidden): pods is forbidden: User "role_user" cannot list resource "pods" in API group "" in the namespace "default2"
root@VM-0-13-ubuntu:/home/ubuntu# kubectl apply -f clusterrole-rb-test2.yaml
rolebinding.rbac.authorization.k8s.io/clusterrole-rb-test created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod -n default2 --context=role_user
No resources found in default2 namespace.
root@VM-0-13-ubuntu:/home/ubuntu# kubectl run nginx --image=nginx -n default2 --context=role_user
pod/nginx created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod -n default2 --context=role_user
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          7s
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  delete pod nginx -n default2 --context=role_user
Error from server (Forbidden): pods "nginx" is forbidden: User "role_user" cannot delete resource "pods" in API group "" in the namespace "default2"
```

方式3：整个集群权限的绑定

此方式主要用于为某个用户绑定所有命名空间下的权限（集群范围），适用于集群范围内授权的场景。例如，日志收集权限、管理人员权限等，以下将为您介绍在如何在 TKE 中使用多个命名空间复用集群权限绑定授权。

1. 创建一个 ClusterRoleBinding 对象资源 clusterrole-crb-test3.yaml 文件，如下权限绑定表示，添加证书认证类型的 role_user 用户在整个集群具有 test-clusterrole（ClusterRole 类型）的权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: clusterrole-crb-test
subjects:
- kind: User
  name: role_user
  namespace: default # User 所在 Namespace
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: test-clusterrole
  apiGroup: ""  # 默认 apiGroup 组为 rbac.authorization.k8s.io
```

2. 从下图验证结果可以得出，应用了权限绑定的 YAML 后，role_user 拥有集群范围的 test-clusterrole 权限。

```
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  apply -f clusterrole-crb-test.yaml
clusterrolebinding.rbac.authorization.k8s.io/clusterrole-crb-test created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  create namespace default3
namespace/default3 created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  create namespace default4
namespace/default4 created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  run nginx --image=nginx -n default3 --context=role_user
pod/nginx created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  run nginx --image=nginx -n default4 --context=role_user
pod/nginx created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod  -n default3 --context=role_user
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          33s
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod  -n default4 --context=role_user
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          32s
```

方式4：自定义权限

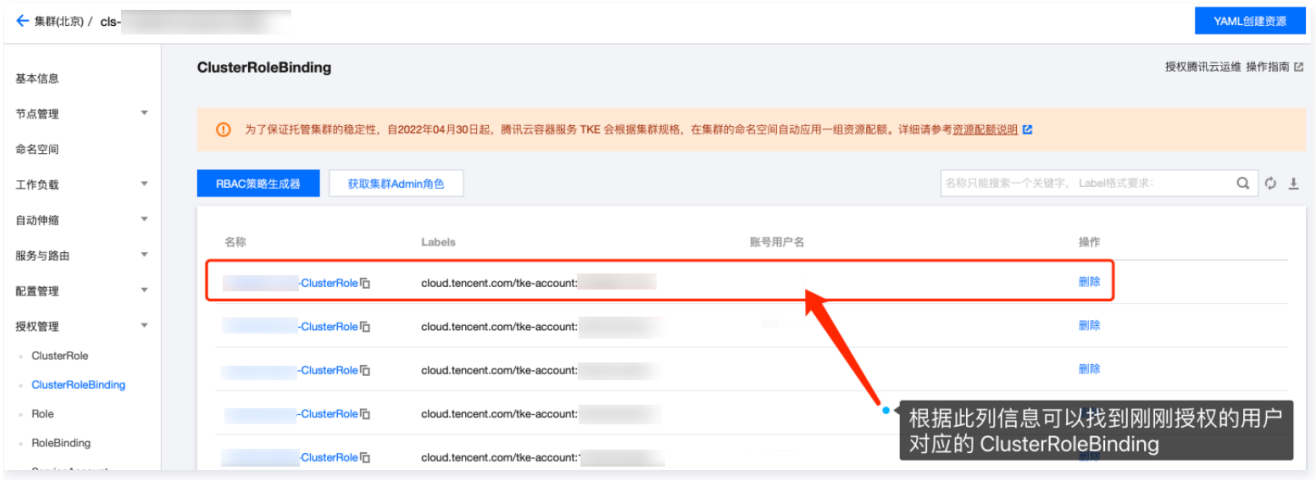
本文以集群管理员给一个用户自定义权限为例：权限包括预设的只读权限额外添加登录容器的权限。

1. 授权

首先集群管理员参考 使用预设身份授权 给指定用户赋予只读的权限。

2. 查看用户 RBAC 里的 User 信息

查看只读用户的 ClusterRoleBinding 的绑定的用户信息，作为新建 ClusterRoleBinding 的需要绑定的用户信息。如下图所示，需要在指定用户的 ClusterRoleBinding 对象中，查看详细信息。




```
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: 700000xxxxxx-1650879262  # RBAC 里指定用户的用户名，需要拿到您指定用户的该信息
```

3. 创建 ClusterRole

通过 YAML 创建有登录容器权限的只读用户的 ClusterRole，示例如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: "700000xxxxxx-ClusterRole-ro"  # ClusterRole 的名字
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - pods/attach
  - pods/exec  # Pod 的登录权限
  - pods/portforward
  - pods/proxy
  verbs:
  - create
  - get
  - list
  - watch
```

4. 创建 ClusterRoleBinding

创建指定用户 ClusterRoleBinding 的 YAML 文件，示例如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: "700000xxxxxx-ClusterRoleBinding-ro"
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: "700000xxxxxx-ClusterRole-ro"  # 使用步骤 3 中的 ClusterRole 的名字
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: "700000xxxxxx-1650879262"  # 使用步骤 2 中的用户信息
```

总结

容器服务 TKE 控制台授权管理功能结合了腾讯云访问权限管理和 Kubernetes RBAC 授权模式，界面配置简单方便，能满足大部分腾讯云子账号的权限控制场景，自定义 YAML 权限绑定方式适用于复杂和个性化的用户权限控制场景，更具灵活性，用户可根据实际授权需求选择合适的权限管理方式。

清理已注销的腾讯云账号资源

最近更新时间：2024-10-29 16:09:22

使用场景

假设您的组织内，因为人员的离职或变动，已经注销了腾讯云账号。腾讯云容器服务向您提供**一键清理**及**自动化清理**已注销腾讯云账号的能力。本文向您介绍如何在容器服务控制台中清除已注销腾讯云账号的 RBAC 资源对象。

操作原理

腾讯云用户对集群的访问由 RBAC 控制，您可以参考 TKE Kubernetes 对象级权限控制 获取更多信息。

操作步骤

查看已注销的腾讯云账号

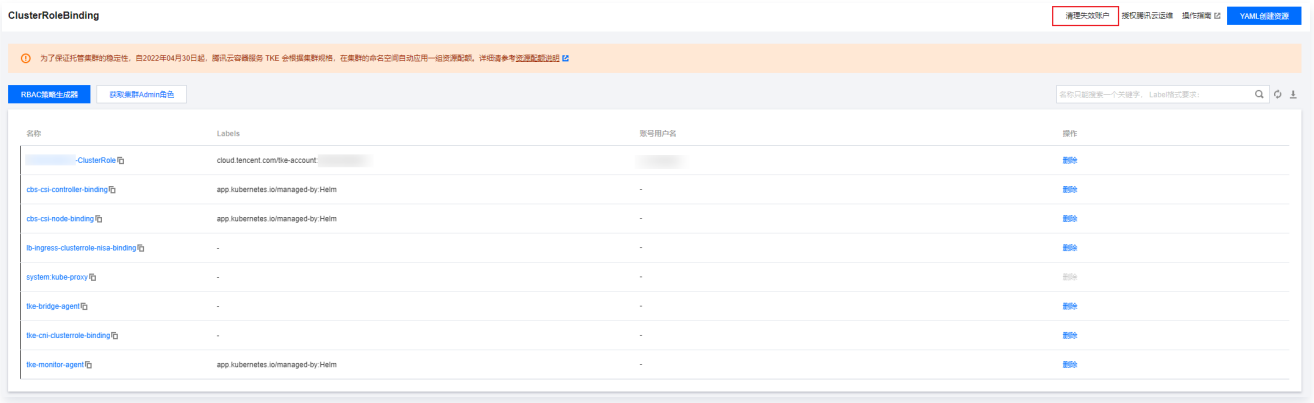
若您的集群中存在已注销的腾讯云账户，您可通过如下步骤查看：

1. 登录 容器服务控制台，在左侧导航栏中选择**集群**。
2. 在集群管理中，选择集群所在地域。
3. 在集群列表中，单击集群 ID，进入集群详情页。
4. 选择**授权管理 > ClusterRoleBinding** 或**授权管理 > RoleBinding**，在列表中的“账号用户名”下，已注销的腾讯云账户为红色，鼠标悬浮会提示清理相关资源对象。

清理失效账户

您可以通过如下步骤，快速实现**手动清理**或**自动清理**集群中已注销腾讯云账号的相关 RBAC 资源对象。

1. 登录 容器服务控制台，在左侧导航栏中选择**集群**。
2. 在集群管理中，选择集群所在地域。
3. 在集群列表中，单击集群 ID，进入集群详情页。
4. 选择**授权管理 > ClusterRoleBinding** 或**授权管理 > RoleBinding**，在 “ClusterRoleBinding” 或 “RoleBinding” 管理页面中，单击右上角**清理失效账户**。如下图所示：



5. 若存在未清理的已注销账号，在“清理已注销的腾讯云账号”弹窗中，单击**立即清理**。
   您也可以开启**自动化清理**，定时清理已注销账号。

Terraform
使用 Terraform 管理 TKE 集群和节点池

最近更新时间：2024-10-18 14:21:42

安装 Terraform

前往 Terraform 官网，使用命令行直接安装 Terraform 或下载二进制安装文件。

认证和鉴权

获取凭证

在首次使用 Terraform 之前，请前往 云 API 密钥页面 申请安全凭证 SecretId 和 SecretKey。若已有可使用的安全凭证，则跳过该步骤。

1. 登录 访问管理控制台，在左侧导航栏，选择**访问密钥 > API 密钥管理**。
2. 在 API 密钥管理页面，单击**新建密钥**，即可以创建一对 SecretId/SecretKey。

鉴权

方式1：（推荐）使用环境变量注入账号的访问密钥

请将如下信息添加至环境变量配置：

```
export TENCENTCLOUD_SECRET_ID="xxx"          # 替换为账号访问密钥的SecretId
export TENCENTCLOUD_SECRET_KEY="xxx"         # 替换为账号访问密钥的SecretKey
```

方式2：在 Terraform 配置文件的 provider 代码块中填写账号的访问密钥

在用户目录下创建 `provider.tf` 文件，输入如下内容：

> **注意：**
> 使用此方式请务必注意配置文件中密钥的安全性。

```
provider "tencentcloud" {
  secret_id = "xxx"                                  # 替换为账号访问密钥的SecretId
  secret_key = "xxx"                                 # 替换为账号访问密钥的SecretKey
}
```

使用 Terraform 创建 TKE 集群

1. 创建一个工作目录，并在工作目录中创建名为 `main.tf` 的 Terraform 配置文件。

> **说明：**
> `main.tf` 文件描述的是以下 Terraform 配置：
> - 创建一个新的 VPC，并创建一个该 VPC 下的 Subnet 子网。
> - 创建一个 TKE 托管集群。
> - 在该 TKE 集群下创建一个节点池。

`main.tf` 文件内容如下：

```
# 标识使用腾讯云的Terraform Provider
terraform {
  required_providers {
    tencentcloud = {
      source = "tencentcloudstack/tencentcloud"
    }
```

```
  }
}

# 定义本地变量，实际使用时按需修改下列变量实际值。后面各代码块中会引用下列变量的值。
locals {
    region = "xxx"                              # 使用的地域，如ap-beijing，即北京
    zone1 = "xxx"                                 # 地域下的一个可用区，如ap-beijing-1，即北京一区
    vpc_name = "xxx"                           # 设置VPC的名字，如tke-tf-demo
    vpc_cidr_block = "xxx"                  # VPC的CIDR设置，如10.0.0.0/16
    subnet1_name = "xxx"                    # 子网1的名字，如tke-tf-demo-sub1
    subnet1_cidr_block = "xxx"          # 子网1的CIDR设置，如10.0.1.0/24
    cluster_name = "xxx"                    # TKE集群的name，如tke-tf-demo-cluster
    network_type = "xxx"                    # TKE托管集群的网络模式，如GR，表示使用Global Route
    cluster_cidr = "xxx"                    # 集群的容器网络，不能与网络冲突，如172.26.0.0/20
    cluster_version = "xxx"                 # TKE集群的Kubernetes版本，如1.22.5
}


# 腾讯云provider的基本配置
provider "tencentcloud" {
    # 如果使用配置文件中写入密钥的方式，在此处写入SecretId和SecretKey。但更推荐使用环境变量注入的方式。
    # secret_id  = "xxx"
    # secret_key = "xxx"
  region = local.region
}

# 声明VPC资源
resource "tencentcloud_vpc" "vpc_example" {
  name = local.vpc_name
  cidr_block    = local.vpc_cidr_block
}

# 声明子网资源
resource "tencentcloud_subnet" "subnet_example" {
  availability_zone = local.zone1
  cidr_block        = local.subnet1_cidr_block
  name              = local.subnet1_name
  vpc_id            = tencentcloud_vpc.vpc_example.id            # 指定子网资源所属VPC为前面创建的
}

# 声明TKE集群资源，将创建网络为Global Route的集群
resource "tencentcloud_kubernetes_cluster" "managed_cluster_example" {
  vpc_id = tencentcloud_vpc.vpc_example.id                                  # 引用前面创建获得的VPC Id
  cluster_name = local.cluster_name
  network_type = local.network_type
  cluster_cidr = local.cluster_cidr
  cluster_version = local.cluster_version
}


# 如果需要创建VPC-CNI模式的集群，可以用下面的声明
# resource "tencentcloud_kubernetes_cluster" "managed_cluster_example" {
#   vpc_id = tencentcloud_vpc.vpc_example.id                                  # 引用前面创建获得的VPC
Id
#   cluster_name = local.cluster_name
#   network_type = "VPC-CNI"
#   eni_subnet_ids = [tencentcloud_subnet.subnet_example.id]
#   service_cidr = "172.16.0.0/24"
#   cluster_version = local.cluster_version
```

```
# }
```

2. （可选）**若您首次使用腾讯云容器服务，您需要为当前服务角色授权，赋予容器服务操作权限后才能正常地访问您的其他云服务资源。**如果您已完成过授权，请直接跳过此步骤。
   - 您可以在首次登录 容器服务控制台 时，对当前账号授予腾讯云容器服务操作云服务器 CVM、负载均衡 CLB、云硬盘 CBS 等云资源的权限。详情请参见 服务授权 。
   - 您也可以在 Terraform 配置文件中完成授权。您需要在工作目录下新建 `cam.tf` 文件，文件内容如下：

```
########################## 请按需在Terraform配置文件中添加声明配置，已在控制台完成授权的角色，则不需要再添加
#############

# 创建服务预设角色TKE_QCSRole
resource "tencentcloud_cam_role" "TKE_QCSRole" {
 name        = "TKE_QCSRole"
 document    = <<EOF
{
 "statement": [
{
  "action":"name/sts:AssumeRole",
  "effect":"allow",
  "principal":{
    "service":"ccs.qcloud.com"
  }
}
 ],
 "version":"2.0"
}
EOF
 description = "当前角色为 腾讯云容器服务 服务角色，该角色将在已关联策略的权限范围内访问您的其他云服务资源。"
}

# 预设策略 QcloudAccessForTKERole
data "tencentcloud_cam_policies" "qca" {
 name = "QcloudAccessForTKERole"
}

# 预设策略 QcloudAccessForTKERoleInOpsManagement
data "tencentcloud_cam_policies" "ops_mgr" {
 name = "QcloudAccessForTKERoleInOpsManagement"
}

# 角色TKE_QCSRole关联QcloudAccessForTKERole策略
resource "tencentcloud_cam_role_policy_attachment" "QCS_QCA" {
 role_id   = lookup(tencentcloud_cam_role.TKE_QCSRole, "id")
 policy_id = data.tencentcloud_cam_policies.qca.policy_list.0.policy_id
}

# 角色TKE_QCSRole关联策略QcloudAccessForTKERoleInOpsManagement
resource "tencentcloud_cam_role_policy_attachment" "QCS_OpsMgr" {
 role_id   = lookup(tencentcloud_cam_role.TKE_QCSRole, "id")
 policy_id = data.tencentcloud_cam_policies.ops_mgr.policy_list.0.policy_id
}

########################## 上述声明完成TKE_QCSRole角色的创建和授权 ##########################
########################## 下列声明完成IPAMDofTKE_QCSRole角色的创建和授权 ###################


# 创建服务预设角色IPAMDofTKE_QCSRole
resource "tencentcloud_cam_role" "IPAMDofTKE_QCSRole" {
```

```
 name = "IPAMDofTKE_QCSRole"
 document     = <<EOF
{
 "statement": [
{
  "action":"name/sts:AssumeRole",
  "effect":"allow",
  "principal":{
    "service":"ccs.qcloud.com"
  }
}
 ],
 "version":"2.0"
}
EOF
 description = "当前角色为 容器服务IPAMD支持 服务角色，该角色将在已关联策略的权限范围内访问您的其他云服务资源。"
}

# 预设策略 QcloudAccessForIPAMDofTKERole
data "tencentcloud_cam_policies" "qcs_ipamd" {
 name = "QcloudAccessForIPAMDofTKERole"
}

# 角色IPAMDofTKE_QCSRole关联策略QcloudAccessForIPAMDofTKERole
resource "tencentcloud_cam_role_policy_attachment" "QCS_Ipamd" {
 role_id   = lookup(tencentcloud_cam_role.IPAMDofTKE_QCSRole, "id")
 policy_id = data.tencentcloud_cam_policies.qcs_ipamd.policy_list.0.policy_id
}
########################## 上述声明完成IPAMDofTKE_QCSRole角色的创建和授权 ##########################
########################## 下列声明完成TKE_QCSLinkedRoleInEKSLog角色的创建和授权 ###################
# 创建服务预设角色TKE_QCSLinkedRoleInEKSLog，如需开启日志采集使用。
resource "tencentcloud_cam_service_linked_role" "service_linked_role" {
  qcs_service_name = ["cvm.qcloud.com", "ekslog.tke.cloud.tencent.com"]
  description      = "tke log role created by terraform"
  tags = {
    "createdBy" = "terraform"
  }
}
```

3. 执行以下命令，初始化 Terraform 的运行环境。

```
terraform init
```

返回信息如下所示：

```
Initializing the backend...

Initializing provider plugins...
- Finding tencentcloudstack/tencentcloud versions matching "~> 1.78.13"...
- Installing tencentcloudstack/tencentcloud v1.78.13...
...

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

...
```

4. 执行以下命令，查看 Terraform 根据配置文件生成的资源规划。

```
terraform plan
```

返回信息如下所示：

```
Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:
...

Plan: 3 to add, 0 to change, 0 to destroy.
...
```

5. 执行以下命令，创建资源。

```
terraform apply
```

返回信息如下所示：

```
...
Plan: 3 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

根据提示输入 `yes` 创建资源，返回信息如下所示：

```
...
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

至此，上述步骤完成了 VPC、子网、TKE 托管集群的创建。您可以在腾讯云控制台查看创建的资源。

使用 Terraform 创建 TKE 节点池

1. 创建一个工作目录，并在工作目录中创建名为 `nodepool.tf` 的 Terraform 配置文件。
`nodepool.tf` 文件内容如下：

```
# 定义本地变量，实际使用时按需修改下列变量实际值。后面各代码块中会引用下列变量的值。
# 实际使用时您也可以通过引用Terraform相关resource实例（如集群tencentcloud_kubernetes_cluster）的方式获取需要的
值。
locals {
    node_pool_name = "xxx"              # 节点池名称，如tke-tf-demo-node-pool
    max_node_size = xxx                     # 节点池最大节点数量
    min_node_size = xxx                     # 节点池最小节点数量
    cvm_instance_type = "xxx"           # 节点池CVM机型，可选值参考
https://cloud.tencent.com/document/api/213/15749获取
    cvm_pass_word = "xxx"                   # 节点池CVM机器登录密码，请注意密码安全性，8-16位
    security_group_ids = ["sg-xxx", "sg-xxx"]                       # 节点池关联的安全组Id数组
}

# 声明TKE节点池资源
```

```
resource "tencentcloud_kubernetes_node_pool" "example_node_pool" {
  cluster_id = tencentcloud_kubernetes_cluster.managed_cluster_example.id # 节点池关联前面创建的集群
  delete_keep_instance = false                  # 设置为false，表明删除节点池时删除关联的CVM实例
  max_size   = local.max_node_size
  min_size   = local.min_node_size
  name       = local.node_pool_name
  vpc_id     = tencentcloud_vpc.vpc_example.id
  subnet_ids = [tencentcloud_subnet.subnet_example.id] # 节点池关联的子网Id数组
  auto_scaling_config {
    instance_type = local.cvm_instance_type
    # key_ids = ["xxx"]                           # 设置节点池CVM机器登录密钥
    password = local.cvm_pass_word                # 设置节点池CVM机器登录密码，请注意密码安全性
    security_group_ids = local.security_group_ids
  }
}
```

2. 执行以下命令，查看 Terraform 根据配置文件生成的资源规划。

```
terraform plan
```

返回信息如下所示：

```
Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:
...

Plan: 1 to add, 0 to change, 0 to destroy.
...
```

3. 执行以下命令，创建资源。

```
terraform apply
```

返回信息如下所示：

```
...
Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

根据提示输入 `yes` 创建资源，返回信息如下所示：

```
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

至此，上述步骤完成了节点池的创建。您可以在腾讯云控制台查看创建的资源。

使用 Terraform 清理资源

如果您需要删除已创建的 VPC、子网、TKE 托管集群资源，可以执行以下命令。

```
terraform destroy
```

返回信息如下所示：

```
...
Plan: 0 to add, 0 to change, 3 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value:
```

根据提示输入 `yes` 确认执行计划，返回信息如下所示：

```
...
Destroy complete! Resources: 3 destroyed.
```

相关文档

- Terraform 官方文档
- 腾讯云Terraform Provider
- 腾讯云 TKE 标准集群
- 腾讯云 TKE 节点池

DevOps
在 containerd 集群中使用 Docker 做镜像构建服务

最近更新时间：2023-05-17 15:41:08

操作场景

在 Kubernetes 集群中，部分 CI/CD 流水线业务可能需要使用 Docker 来提供镜像打包服务。可通过宿主机的 Docker 实现，将 Docker 的 UNIX Socket（`/var/run/docker.sock`）作为 hostPath 挂载到 CI/CD 的业务 Pod 中，之后在容器里通过 UNIX Socket 来调用宿主机上的 Docker 进行构建。该方式操作简单，比真正意义上的 Docker in Docker 更节省资源，但该方式可能会遇到以下问题：

- 无法运行在 Runtime 是 containerd 的集群中。
- 如果不加以控制，可能会覆盖掉节点上已有的镜像。
- 在需要修改 Docker Daemon 配置文件的情况下，可能会影响到其他业务。
- 在多租户的场景下并不安全，当拥有特权的 Pod 获取到 Docker 的 UNIX Socket 之后，Pod 中的容器不仅可以调用宿主机的 Docker 构建镜像、删除已有镜像或容器，甚至可以通过 `docker exec` 接口操作其他容器。

针对上述第1个问题，Kubernetes 在官方博客宣布将在1.22版本之后弃用 Docker，这部分用户可能会将业务转投到 containerd。对于部分需要 containerd 集群，而不改变 CI/CD 业务流程仍使用 Docker 构建镜像一部分的场景，可以通过在原有 Pod 上添加 DinD 容器作为 Sidecar 或者使用 DaemonSet 在节点上部署专门用于构建镜像的 Docker 服务。本文将为您介绍以下两种方式实现在 CI/CD 流水线业务上使用 Docker 构建镜像：

- 方式1：使用 DinD 作为 Pod 的 Sidecar
- 方式2：使用 DaemonSet 在每个 containerd 节点上部署 Docker

操作步骤

方式1：使用 DinD 作为 Pod 的 Sidecar

DinD（Docker in Docker）实现原理可参见 DinD 官方文档，本文示例将为 clean-ci 容器添加一个 Sidecar，配合 emptyDir 使 clean-ci 容器可以通过 UNIX Socket 访问 DinD 容器。示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: clean-ci
spec:
  containers:
  - name: dind
    image: 'docker:stable-dind'
    command:
    - dockerd
    - --host=unix:///var/run/docker.sock
    - --host=tcp://0.0.0.0:8000
    securityContext:
      privileged: true
    volumeMounts:
    - mountPath: /var/run
      name: cache-dir
  - name: clean-ci
    image: 'docker:stable'
    command: ["/bin/sh"]
    args: ["-c", "docker info >/dev/null 2>&1; while [ $? -ne 0 ] ; do sleep 3; docker info >/dev/null
2>&1; done; docker pull library/busybox:latest; docker save -o busybox-latest.tar library/busybox:latest;
docker rmi library/busybox:latest; while true; do sleep 86400; done"]
    volumeMounts:
    - mountPath: /var/run
      name: cache-dir
  volumes:
  - name: cache-dir
```

```
    emptyDir: {}
```

方式2：使用 DaemonSet 在每个 containerd 节点上部署 Docker

该方式较为简单，直接在 containerd 集群中下发 DaemonSet 即可（挂载 hostPath），为不影响节点上 `/var/run` 路径，可以指定其他路径。

1. 使用以下 YAML 部署 DaemonSet。示例如下：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: docker-ci
spec:
  selector:
    matchLabels:
      app: docker-ci
  template:
    metadata:
      labels:
        app: docker-ci
    spec:
      containers:
      - name: docker-ci
        image: 'docker:stable-dind'
        command:
        - dockerd
        - --host=unix:///var/run/docker.sock
        - --host=tcp://0.0.0.0:8000
        securityContext:
          privileged: true
        volumeMounts:
        - mountPath: /var/run
          name: host
      volumes:
      - name: host
        hostPath:
          path: /var/run
```

2. 将业务 Pod 与 DaemonSet 共享同一个 hostPath。示例如下：

```
apiVersion: v1
kind: Pod
metadata:
   name: clean-ci
spec:
   containers:
   - name: clean-ci
     image: 'docker:stable'
     command: ["/bin/sh"]
     args: ["-c", "docker info >/dev/null 2>&1; while [ $? -ne 0 ] ; do sleep 3; docker info
>/dev/null 2>&1; done; docker pull library/busybox:latest; docker save -o busybox-latest.tar
library/busybox:latest; docker rmi library/busybox:latest; while true; do sleep 86400; done"]
     volumeMounts:
     - mountPath: /var/run
       name: host
   volumes:
   - name: host
     hostPath:
       path: /var/run
```

在 TKE 上部署 Jenkins

最近更新时间：2023-09-04 14:47:11

操作场景

许多 DevOps 的需求需要借助 Jenkins 来实现，本文将介绍如何在容器服务 TKE 上部署 Jenkins。

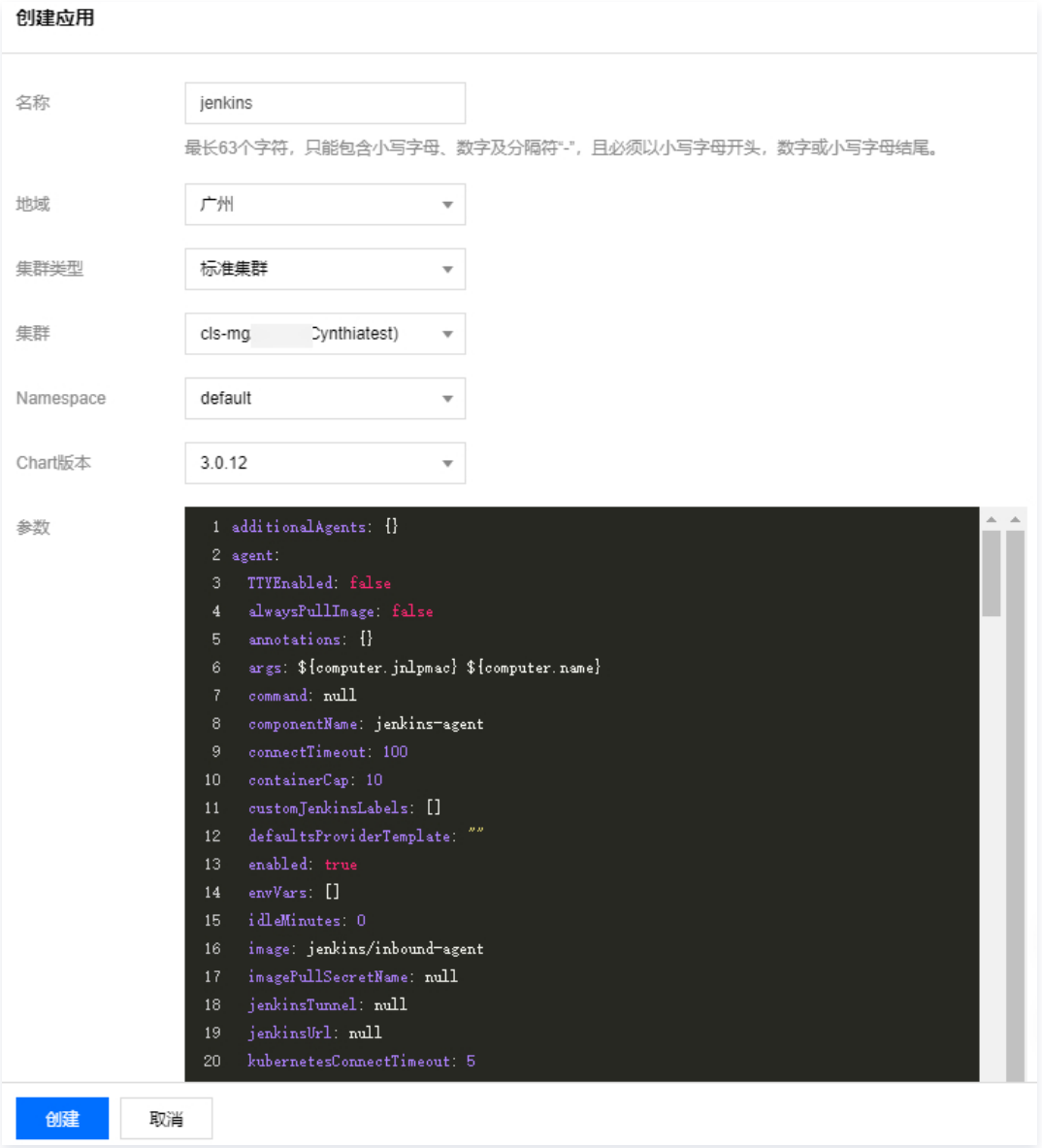
前提条件

已创建 TKE 集群。

操作步骤

安装 Jenkins

1. 登录容器服务控制台，选择左侧导航栏中的 应用市场。
2. 在**应用市场**页面搜索 Jenkins，并进入 Jenkins 应用页面。
3. 单击**创建应用**，创建应用窗口中的“参数” values.yaml 部分，可以根据自身需求进行微调。



4. 单击**创建**既可安装 Jenkins。

暴露 Jenkins UI

默认情况下，在集群外无法访问 Jenkins UI。如需访问 Jenkins UI，通常使用 Ingress 来暴露访问。TKE 提供 CLB 类型 Ingress 与 Nginx 类型 Ingress 两种 Ingress，您可参考文档自行选择。

> **说明**

以下示例使用 Jenkins 2.263版本， 不同 Jenkins 版本使用 UI 上存在差异。您可以根据业务需要进行选择。

登录 Jenkins

进入 Jenkins UI 界面，输入初始用户名和密码登录 Jenkins 后台，用户名为 admin，初始密码需通过以下命令获取。

```
kubectl -n devops get secret jenkins -o jsonpath='{.data.jenkins-admin-password}' | base64 -d
```

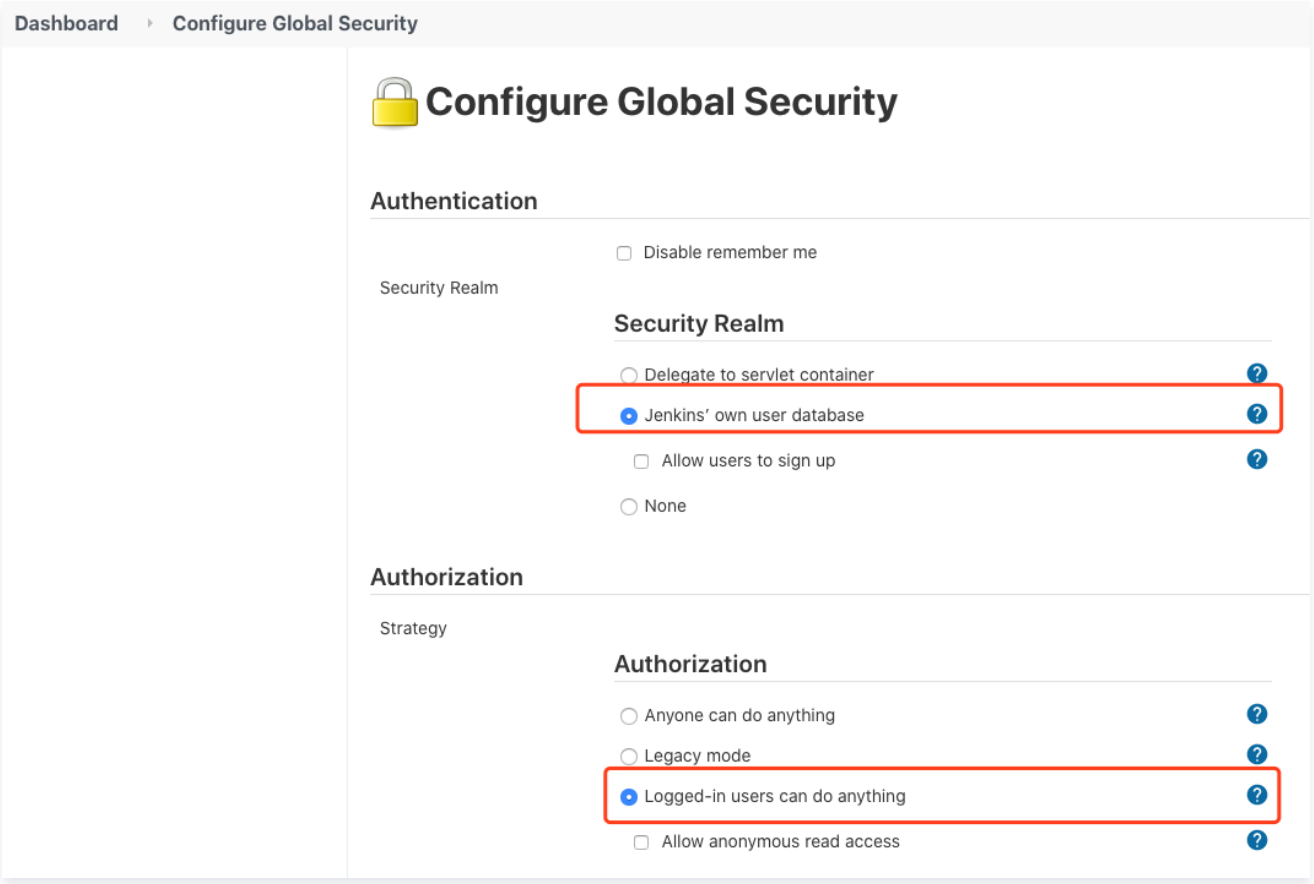
注意

执行上述命令时，需替换为实际环境所安装的命名空间。

创建用户

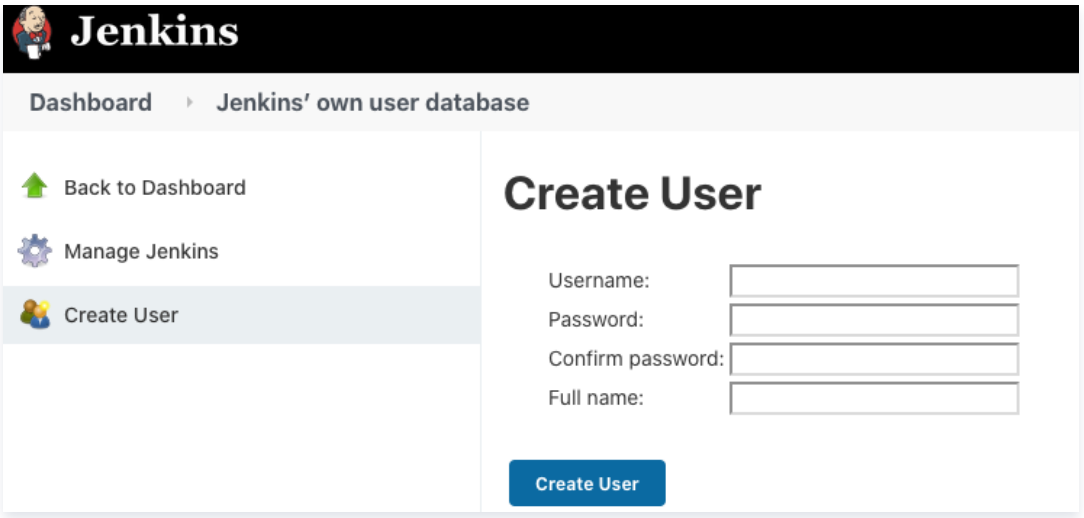
建议通过普通用户管理 Jenkins，创建普通用户之前，需配置认证与授权策略。

1. 登录 Jenkins 后台，选择 **Dashboard** > **Manage Jenkins** > **Security** > **Configure Global Security**，进入认证与授权策略页面。如下图所示：



- **Security Realm**：选择 **Jenkins' own user database**。
- **Authorization**：选择 **Logged-in users can do anything**。

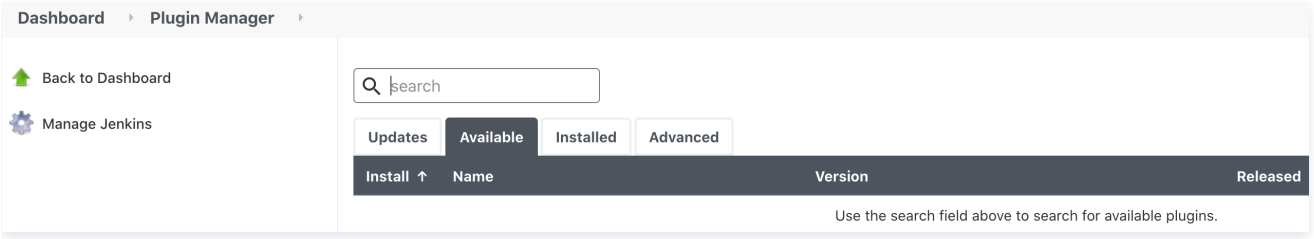
2. 选择 **Dashboard** > **Manage Jenkins** > **Security** > **Manage Users** > **Create User**，进入创建用户界面，根据以下提示创建用户。如下图所示：



   - **Username**：输入用户名。
   - **Password**：输入用户密码。
   - **Confirm password**：确认用户密码。
   - **Full name**：输入用户名全称。
3. 单击 **Create User** 即可创建用户。

安装插件

登录 Jenkins 后台，选择 **Dashboard** > **Manage Jenkins** > **System Configuration** > **Manage Plugins**，进入插件管理页面。



您可以安装以下常用插件：

- kubernetes
- pipeline
- git
- gitlab
- github

弹性伸缩
事件驱动弹性伸缩最佳实践（KEDA）
认识 KEDA

最近更新时间：2024-10-29 16:09:22

什么是 KEDA?

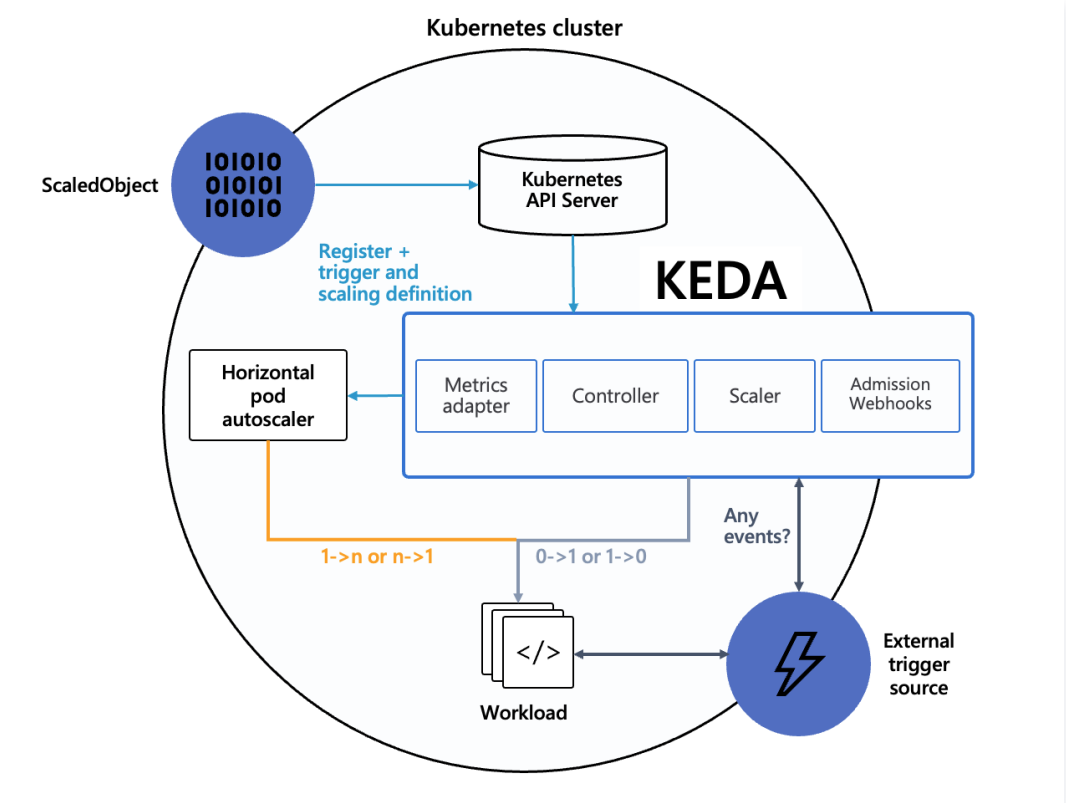
KEDA（Kubernetes-based Event-Driven Autoscaler）是在 Kubernetes 中事件驱动的弹性伸缩器，功能非常强大。不仅支持根据基础的 CPU 和内存指标进行伸缩，还支持根据各种消息队列中的长度、数据库中的数据统计、QPS、Cron 定时计划以及您可以想象的任何其他指标进行伸缩，甚至还可以将副本缩到0。
该项目于2020年3月被 CNCF 接收，并于2021年8月开始孵化，最终在2023年8月宣布毕业，目前已经非常成熟，可放心在生产环境中使用。

为什么需要 KEDA?

HPA 是 Kubernetes 自带的 Pod 水平自动伸缩器，只能根据监控指标对工作负载自动扩缩容，指标主要是工作负载的 CPU 和内存的利用率（Resource Metrics），如果需要支持其它自定义指标，一般是安装 prometheus-adapter 来作为 HPA 的 Custom Metrics 和 External Metrics 的实现来将 Prometheus 中的监控数据作为自定义指标提供给 HPA。理论上，用 HPA + prometheus-adapter 也能实现 KEDA 的功能，但实现上会非常麻烦。例如，如果想根据数据库中任务表里记录的待执行的任务数量统计进行伸缩，就需要编写并部署 Exporter 应用，将统计结果转换为 Metrics 暴露给 Prometheus 进行采集，然后 prometheus-adapter 再从 Prometheus 查询待执行的任务数量指标来决定是否伸缩。
KEDA 的出现主要是为了解决 HPA 无法基于灵活的事件源进行伸缩的问题，内置了几十种常见的 Scaler，可直接跟各种第三方应用对接，例如各种开源和云托管的关系型数据库、时序数据库、文档数据库、键值存储、消息队列、事件总线等，也可以使用 Cron 表达式进行定时自动伸缩，它涵盖了常见的伸缩场景，并且如果发现不支持的场景，还可以自己实现一个外部 Scaler 来配合 KEDA 使用。

KEDA 的原理

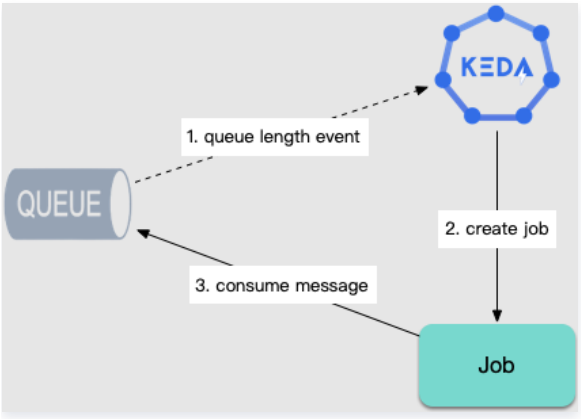
KEDA 并不是要替代 HPA，而是作为 HPA 的补充或者增强。实际上，KEDA 经常与 HPA 一起使用。以下是 KEDA 官方的架构图：



- 当要将工作负载的副本数缩到闲时副本数，或从闲时副本数扩容时，由 KEDA 通过修改工作负载的副本数实现（闲时副本数小于 `minReplicaCount`，包括0，即可以缩到0）。
- 其他情况下的扩缩容由 HPA 实现，HPA 由 KEDA 自动管理，HPA 使用 External Metrics 作为数据源，而 External Metrics 后端的数据由 KEDA 提供。

- KEDA 各种 Scalers 的核心其实就是为 HPA 暴露 External Metrics 格式的数据，KEDA 会将各种外部事件转换为所需的 External Metrics 数据，最终实现 HPA 通过 External Metrics 数据进行自动伸缩，直接复用了 HPA 已有的能力，如果需要控制扩缩容的行为细节（例如快速扩容、缓慢缩容），可以直接通过配置 HPA 的 `behavior` 字段来实现（要求 Kubernetes 版本 ≥1.18）。

除了工作负载的扩缩容，对于任务计算类场景，KEDA 还可以根据排队的任务数量自动创建 Job 来实现对任务的及时处理：

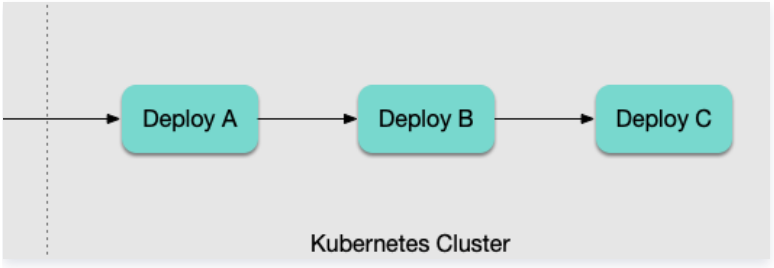


哪些场景适合使用 KEDA?

以下是适合使用 KEDA 的场景。

微服务多级调用

在微服务中，存在多级调用的业务场景，压力是逐级传递的，下面展示了一个常见的情况：



如果使用传统的 HPA 根据负载扩缩容，用户流量进入集群后：

1. `Deploy A` 负载升高，指标变化迫使 `Deploy A` 扩容。
2. A 扩容之后，吞吐量变大，B 受到压力，再次采集到指标变化，扩容 `Deploy B` 。
3. B 吞吐变大，C 受到压力，扩容 `Deploy C` 。

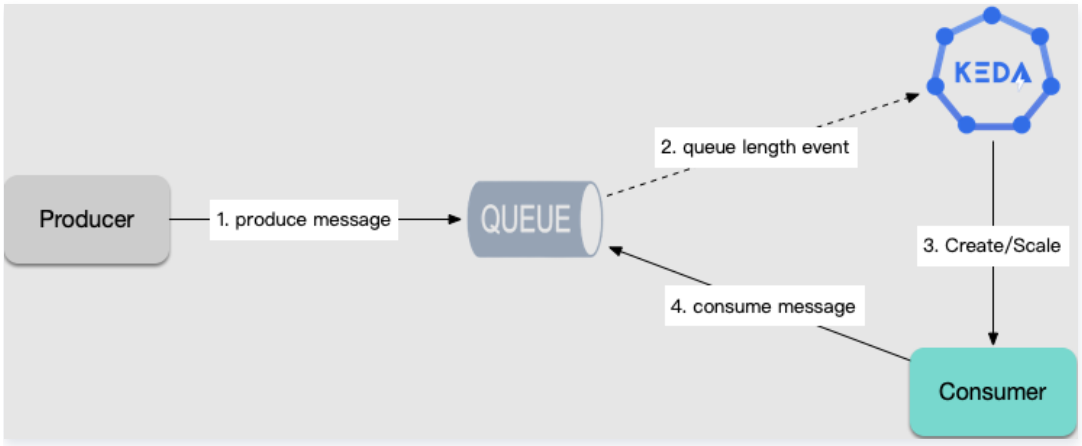
这个逐级传递的过程缓慢且危险：每一级的扩容都是直接被 CPU 或内存的飙高触发，被 “冲垮” 的可能性是普遍存在。这种被动、滞后的方式，很明显是有问题的。

此时，我们可以利用 KEDA 来实现多级快速扩容：

- `Deploy A` 可根据自身负载或网关记录的 QPS 等指标扩缩容。
- `Deploy B` 和 `Deploy C` 可根据 `Deploy A` 副本数扩缩容（各级服务副本数保持一定比例）。

任务执行（生产者与消费者）

对于需要长时间执行的计算任务，如数据分析、ETL、机器学习等场景，从消息队列或数据库中获取任务进行执行，需要根据任务数量进行伸缩。使用 KEDA 可以根据排队中的任务数量对工作负载进行自动伸缩，并可以自动创建 Job 来消费任务。



周期性规律

如果业务具有周期性的波峰波谷特征，可以使用 KEDA 配置定时伸缩。在波峰到来之前提前扩容，波峰结束后缓慢缩容，以适应业务的周期性变化。

在 TKE 上部署 KEDA

最近更新时间：2024-09-12 11:05:31

添加 helm repo

在 TKE 上部署 KEDA，首先需要添加 KEDA 的 Helm 仓库，可以使用以下命令添加：

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update
```

准备 values.yaml

接下来，可以查看默认的 values.yaml 文件，以了解可以自定义的配置项。可以使用以下命令查看：

```
helm show values kedacore/keda
```

由于默认的依赖镜像在国内环境无法拉取，可以替换为使用 Docker Hub 上的 mirror 镜像，在 values.yaml 文件中进行配置，示例如下：

```
image:
  keda:
    registry: docker.io
    repository: imroc/keda
  metricsApiServer:
    registry: docker.io
    repository: imroc/keda-metrics-apiserver
  webhooks:
    registry: docker.io
    repository: imroc/keda-admission-webhooks
```

> **说明：**
> 以上镜像会长期自动同步，您可以放心使用和更新版本。

安装

使用以下命令安装 KEDA：

```
helm upgrade --install keda kedacore/keda \
  --namespace keda --create-namespace \
  -f values.yaml
```

版本与升级

每个 KEDA 的版本都有对应适配的 Kubernetes 版本区间，在安装 KEDA 之前，您需要确认 TKE 集群的版本与要安装的 KEDA 版本是否兼容。您可以查看 KEDA Kubernetes Compatibility 来确认当前集群版本能兼容的 KEDA 版本。
例如 TKE 集群版本是1.26，对应能兼容的 KEDA 最新版本是 v2.12，再查询到 KEDA v2.12 (APP VERSION) 对应的 Chart 版本 (CHART VERSION) 最高版本是 2.12.1：

```
$ helm search repo keda --versions
NAME                                    CHART VERSION   APP VERSION     DESCRIPTION
kedacore/keda                           2.13.2          2.13.1          Event-based autoscaler
for workloads on Kubernetes
kedacore/keda                           2.13.1          2.13.0          Event-based autoscaler
for workloads on Kubernetes
kedacore/keda                           2.13.0          2.13.0          Event-based autoscaler
for workloads on Kubernetes
```

```
kedacore/keda                                   2.12.1          2.12.1          Event-based autoscaler
for workloads on Kubernetes
kedacore/keda                                   2.12.0          2.12.0          Event-based autoscaler
for workloads on Kubernetes
kedacore/keda                                   2.11.2          2.11.2          Event-based autoscaler
for workloads on Kubernetes
kedacore/keda                                   2.11.1          2.11.1          Event-based autoscaler
for workloads on Kubernetes
```

安装 KEDA 时指定版本：

```
helm upgrade --install keda kedacore/keda \
  --namespace keda --create-namespace \
  --version 2.12.1 \
  -f values.yaml
```

后续升级版本时可复用上面的安装命令，只需修改下版本号即可。

> ⚠ **注意：**
> 在升级 TKE 集群前也用这里的方法先确认下升级后的集群版本能否兼容当前版本的 KEDA，如果不能，请提前升级 KEDA 到当前集群版本所能兼容的最新 KEDA 版本。

卸载

具体操作请参见 官方卸载说明 。

参考资料

KEDA 官方文档：Deploying KEDA

定时水平伸缩 (Cron 触发器)

最近更新时间：2024-05-14 14:47:01

Cron 触发器

KEDA 支持 Cron 触发器，可以使用 Cron 表达式来配置周期性的定时扩缩容。用法请参见 KEDA Scalers: Cron 。
Cron 触发器适用于有周期性特征的业务，例如业务流量具有固定的周期性波峰和波谷特征。

使用示例

每天固定时间点的秒杀活动

秒杀活动的特征是时间比较固定，可以在活动开始前提前扩容，以下展示了 `ScaledObject` 配置示例。

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: seckill
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: seckill
  pollingInterval: 15
  minReplicaCount: 2 # 至少保留 2 个副本
  maxReplicaCount: 1000
  advanced:
    horizontalPodAutoscalerConfig:
      behavior: # 控制扩缩容行为，使用比较保守的策略，快速扩容，缓慢缩容
        scaleDown: # 缓慢缩容：至少冷却 10 分钟才能缩容
          stabilizationWindowSeconds: 600
          policies:
            - type: Percent
              value: 100
              periodSeconds: 15
        scaleUp: # 快速扩容：每 15s 最多允许扩容 5 倍
          policies:
            - type: Percent
              value: 500
              periodSeconds: 15
  triggers:
    - type: cron # 每天早上 10 点秒杀活动，确保前后半小时内至少有 200 个副本
      metadata:
        timezone: Asia/Shanghai
        start: 30 9 * * *
        end: 30 10 * * *
        desiredReplicas: "200"
    - type: cron # 每天晚上 6 点秒杀活动，确保前后半小时内至少有 200 个副本
      metadata:
        timezone: Asia/Shanghai
        start: 30 17 * * *
        end: 30 18 * * *
        desiredReplicas: "200"
    - type: memory # CPU 利用率超过 60% 扩容
      metricType: Utilization
      metadata:
        value: "60"
    - type: cpu # 内存利用率超过 60% 扩容
      metricType: Utilization
```

```
      metadata:
        value: "60"
```

注意事项

通常情况下，触发器不能仅配置 Cron，还需要与其他触发器配合使用。因为如果在 Cron 的 start 和 end 区间之外的时间段，如果没有其它触发器活跃，副本数将会降到 `minReplicaCount` 。

多级服务同步水平伸缩（Workload 触发器）

最近更新时间：2024-05-14 14:47:01

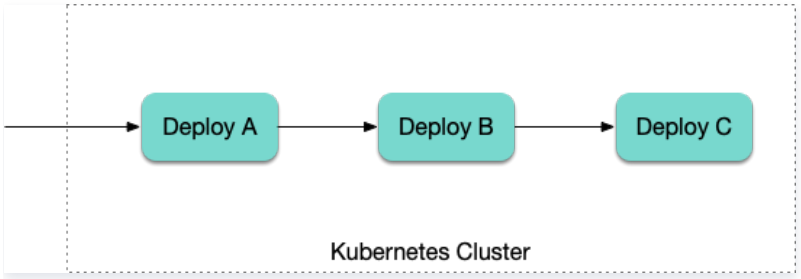
Workload 触发器

KEDA 支持 Kubernetes Workload 触发器，可以根据一个或多个工作负载的 Pod 数量来进行扩缩容，在多级服务调用的场景下非常有用。具体用法请参见 KEDA Scalers: Kubernetes Workload 。

使用示例

多级服务同时扩容

图片示例为多级微服务调用：



- A、B、C 这组服务通常具有固定的数量比例。
- A 的压力突增，迫使扩容，B 和 C 也可以用 KEDA 的 Kubernetes Workload 触发器实现与 A 几乎同时扩容，而无需等待压力逐级传导才缓慢迫使扩容。

首先配置 A 的扩容，可以根据 CPU 和内存压力扩容。例如：

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: a
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: a
  pollingInterval: 15
  minReplicaCount: 10
  maxReplicaCount: 1000
  triggers:
    - type: memory
      metricType: Utilization
      metadata:
        value: "60"
    - type: cpu
      metricType: Utilization
      metadata:
        value: "60"
```

然后配置 B 和 C 的扩容，假设固定比例 A:B:C = 3:3:2。例如：

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: b
spec:
  scaleTargetRef:
```

```
    apiVersion: apps/v1
    kind: Deployment
    name: b
  pollingInterval: 15
  minReplicaCount: 10
  maxReplicaCount: 1000
  triggers:
    - type: kubernetes-workload
      metadata:
        podSelector: 'app=a' # 选中 A 服务
        value: '1' # A/B=3/3=1
```

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: c
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: c
  pollingInterval: 15
  minReplicaCount: 3
  maxReplicaCount: 340
  triggers:
    - type: kubernetes-workload
      metadata:
        podSelector: 'app=a' # 选中 A 服务
        value: '3' # A/C=3/2=1.5
```

通过以上配置，当 A 的压力增加时，A、B 和 C 将几乎同时进行扩容，而无需等待压力逐级传导。这样可以更快地适应压力变化，提高系统的弹性和性能。

基于 Prometheus 自定义指标的弹性伸缩

最近更新时间：2024-07-03 17:16:01

Prometheus 触发器

KEDA 支持 `prometheus` 类型的触发器，即根据自定义的 PromQL 查询到的 Prometheus 指标数据进行伸缩，完整配置参数请参见 KEDA Scalers: Prometheus，本文将给出使用案例。

案例：基于 istio 的 QPS 指标伸缩

如果您使用 istio，业务 Pod 注入了 sidecar，会自动暴露一些七层的监控指标，最常见的是 `istio_requests_total`，可以通过这个指标计算 QPS。假设场景是：A 服务需要根据 B 服务处理的 QPS 进行伸缩。配置示例如下：

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: b-scaledobject
  namespace: prod
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: a # 对 A 服务进行伸缩
  pollingInterval: 15
  minReplicaCount: 1
  maxReplicaCount: 100
  triggers:
    - type: prometheus
      metadata:
        serverAddress: http://monitoring-kube-prometheus-prometheus.monitoring.svc.cluster.local:9090 #
替换 Prometheus 的地址
        query: | # 计算 B 服务 QPS 的 PromQL

sum(irate(istio_requests_total{reporter=~"destination",destination_workload_namespace=~"prod",destination
_workload=~"b"}[1m]))
        threshold: "100" # A服务副本数=ceil(B服务QPS/100)
```

相比 prometheus-adapter 的优势

prometheus-adapter 也支持相同的能力，即根据 Prometheus 中的监控指标数据进行伸缩，但相比 KEDA 的方案有以下不足：

- 每次新增自定义指标，都要改动 `prometheus-adapter` 的配置，且改配置是集中式管理的，不支持通过 CRD 管理，配置维护较为繁琐。而 KEDA 方案只需要配置 `ScaledObject` 或 `ScaledJob` 这种 CRD，不同业务可以使用不同的 YAML 文件维护，利于配置维护。
- `prometheus-adapter` 的配置语法晦涩难懂，不能直接写 `PromQL`，需要学习 `prometheus-adapter` 的配置语法，增加了学习成本，而 KEDA 的 prometheus 配置则非常简单，指标可以直接使用 `PromQL` 查询语句，简单明了。
- `prometheus-adapter` 仅支持根据 Prometheus 监控数据进行伸缩，而对于 KEDA 来说，Prometheus 只是众多触发器中的一种。

基于 CLB 监控指标的水平伸缩

最近更新时间：2025-06-05 18:37:41

业务场景

TKE 上的业务流量通常通过 CLB（腾讯云负载均衡器）进行接入。有时候，您希望工作负载能够根据 CLB 的监控指标进行伸缩，例如：

1. 长连接场景（如游戏房间、在线会议）：每个用户对应一条连接，工作负载里的每个 Pod 处理的连接数上限相对固定。这时可以根据 CLB 连接数指标进行伸缩。
2. HTTP 协议的在线业务：工作负载里的单个 Pod 所能支撑的 QPS 相对固定。这时可以根据 CLB 的 QPS（每秒请求数）指标进行伸缩。

keda-tencentcloud-clb-scaler 介绍

KEDA 有很多内置的触发器，但没有腾讯云 CLB 的，不过 KEDA 支持 external 类型的触发器来对触发器进行扩展，keda-tencentcloud-clb-scaler 是基于腾讯云 CLB 监控指标的 KEDA External Scaler，可实现基于 CLB 连接数、QPS 和带宽等指标的弹性伸缩。

操作步骤

准备访问密钥

需要准备一个腾讯云账号的访问密钥（SecretID、SecretKey），具体操作可参考 子账号访问密钥管理，要求账号至少具有以下权限：

```
{
    "version": "2.0",
    "statement": [
        {
            "effect": "allow",
            "action": [
                "clb.DescribeLoadBalancers",
                "monitor.DescribeProductList",
                "monitor.GetMonitorData",
                "monitor.DescribeBaseMetrics"
            ],
            "resource": [
                "*"
            ]
        }
    ]
}
```

安装 keda-tencentcloud-clb-scaler

```
helm repo add clb-scaler https://imroc.github.io/keda-tencentcloud-clb-scaler
helm upgrade --install clb-scaler clb-scaler/clb-scaler -n keda \
  --set region="ap-chengdu" \
  --set credentials.secretId="xxx" \
  --set credentials.secretKey="xxx"
```

- 请将 region 修改为您的 CLB 所在地域（一般与集群所在地域相同），地域列表详情请参见 地域与可用区 。
- credentials.secretId 和 credentials.secretKey 是您腾讯云账户的密钥对，用于访问和获取 CLB 的监控数据。请将其替换为您自己的密钥对。

部署工作负载

您可以使用以下用于测试的工作负载 YAML 样例：

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

```
    app: httpbin
  name: httpbin
spec:
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 80
  selector:
    app: httpbin
  type: LoadBalancer

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
  template:
    metadata:
      labels:
        app: httpbin
    spec:
      containers:
        - image: kennethreitz/httpbin:latest
          name: httpbin
```

部署完成后，将自动创建响应的公网 CLB 接入流量，您可以使用以下命令获取对应的 CLB ID：

```
$ kubectl get svc httpbin -o jsonpath='{.metadata.annotations.service\.kubernetes\.io/loadbalance-id}'
lb-********
```

记录下获取到的 CLB ID，这将在后续的 KEDA 配置中使用。

使用 ScaledObject 配置基于 CLB 监控指标的弹性伸缩

配置方法

基于 CLB 的监控指标通常用于在线业务，使用 KEDA 的 ScaledObject 配置弹性伸缩，配置 external 类型的 trigger，并传入所需的 metadata，主要包含以下字段：

- scalerAddress 是 keda-operator 调用 keda-tencentcloud-clb-scaler 时使用的地址。
- loadBalancerId 是 CLB 的实例 ID。
- metricName 是 CLB 的监控指标名称，公网和内网的大部分指标相同。
- threshold 是扩缩容的指标阈值，即会通过比较 metricValue / Pod 数量 与 threshold 的值来决定是否扩缩容。
- listener 是唯一可选的配置，指定监控指标的 CLB 监听器，格式：协议/端口。

配置示例一：基于 CLB 连接数指标的弹性伸缩

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: httpbin
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
```

```
    name: httpbin
  pollingInterval: 15
  minReplicaCount: 1
  maxReplicaCount: 100
  triggers:
    - type: external
      metadata:
        # highlight-start
        scalerAddress: clb-scaler.keda.svc.cluster.local:9000
        loadBalancerId: lb-xxxxxxxx
        metricName: ClientConnum # 连接数指标
        threshold: "100" # 每个 Pod 处理 100 条连接
        listener: "TCP/8080" # 可选，指定监听器，格式：协议/端口
        # highlight-end
```

配置示例二：基于 CLB QPS指标的弹性伸缩

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: httpbin
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: httpbin
  pollingInterval: 15
  minReplicaCount: 1
  maxReplicaCount: 100
  triggers:
    - type: external
      metadata:
        # highlight-start
        scalerAddress: clb-scaler.keda.svc.cluster.local:9000
        loadBalancerId: lb-xxxxxxxx
        metricName: TotalReq # 每秒连接数指标
        threshold: "500" # 平均每个 Pod 支撑 500 QPS
        listener: "TCP/8080" # 可选，指定监听器，格式：协议/端口
        # highlight-end
```

基于 Apache Pulsar 消息队列的弹性伸缩

最近更新时间：2024-05-14 14:47:01

概述

KEDA 的触发器支持 Apache Pulsar，即根据 Pulsar 消息队列中的未消费的消息数量进行水平伸缩，用法请参见 KEDA Scalers: Apache Pulsar 。腾讯云也有商业版的 Pulsar 产品，即 TDMQ for Pulsar 。本文举例介绍配置基于 TDMQ for Pulsar 消息队列中未消费的消息数量进行水平伸缩。当然，如果您使用的是自建的开源 Apache Pulsar，配置方法也是类似的。

操作步骤

下面使用 pulsar-demo 来模拟 Pulsar 生产者和消费者，再结合 KEDA 配置实现 Pulsar 消费者基于 Pulsar 消息数量的水平伸缩，在实际使用中，可根据自己的情况进行相应替换。

1. 获取 Pulsar API 调用地址

1. 在 Pulsar 集群管理页面 找到需要使用的 Pulsar 集群，单击**接入地址**可获取 Pulsar 的 URL，通常使用 VPC 内网接入地址（解析出来是169保留网段的 IP，在任意 VPC 都可用）。如下图所示：

2. 复制并记录 API 调用地址。

获取 Pulsar Topic

在 Pulsar Topic 管理页面，复制需要使用的 Topic 名称。如下图所示：

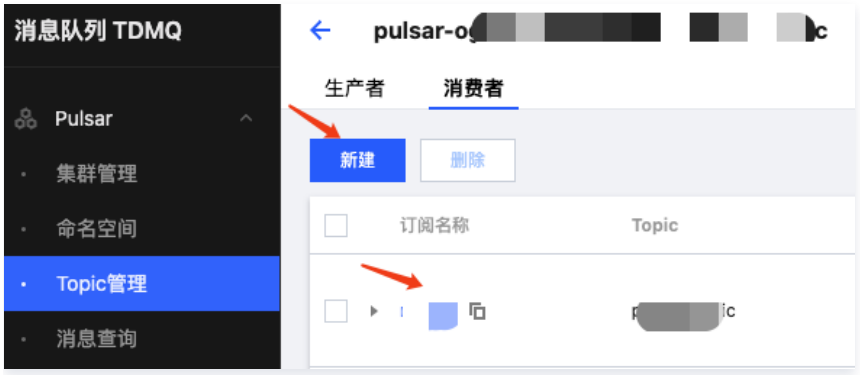
> **注意：**
> 只支持持久化类型的 Topic，配置所需的 Topic 是在已复制的 Topic 名称前面加 `persistent://` 。

获取 Pulsar JWT Token

1. 确保在 Pulsar 角色管理 创建了所需的角色，并在 Pulsar 命名空间 中配置了相应的权限，确保所需角色有相应的生产消息或消费消息的权限。
2. 复制密钥，即 Pulsar 客户端所需的 JWT Token。如下图所示：

获取订阅名称

在 Topic 管理的消费者页面，根据需要，查看已有的订阅，或者新建订阅，并记录下需要使用的订阅名称。如下图所示：



部署生产者

1. 准备生产者配置，根据之前获取的 Pulsar 相关信息替换配置。示例如下：

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: producer-secret
stringData:
  URL: http://pulsar-xxxxxxxxxxxx.tdmq.ap-cd.qcloud.tencenttdmq.com:5005 # 替换 API 调用地址
  TOPIC: persistent://pulsar-xxxxxxxxxxxx/test-ns/test-topic # 替换 Topic
  TOKEN: xxx # 替换角色密钥 (JWT Token)
```

2. 部署生产者持续发送新消息：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: producer
spec:
  replicas: 1
  selector:
  matchLabels:
    app: producer
  template:
  metadata:
    labels:
      app: producer
  spec:
    containers:
      - name: producer
        image: imroc/pulsar-demo:main
        imagePullPolicy: Always
        args:
          - producer
          - --produce-duration
          - 2s # 每 2s 生产一条消息
        envFrom:
          - secretRef:
              name: producer-secret
    terminationGracePeriodSeconds: 1
```

部署消费者

1. 准备消费者配置，根据前面获取的 Pulsar 相关信息替换配置。示例如下：

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: consumer-secret
stringData:
  URL: http://pulsar-xxxxxxxxxxxx.tdmq.ap-cd.qcloud.tencenttdmq.com:5005 # 替换 API 调用地址
  TOPIC: persistent://pulsar-xxxxxxxxxxxx/test-ns/test-topic # 替换 Topic
  TOKEN: xxx # 替换角色密钥 (JWT Token)
  SUBSCRIPTION: xxx # 替换订阅名称
```

2. 通过 Deployment 部署消费者，持续消费消息：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumer
spec:
  replicas: 1
  selector:
  matchLabels:
    app: consumer
  template:
  metadata:
    labels:
      app: consumer
  spec:
    containers:
    - args:
      - consumer
      - --consume-duration
      - 10s # 单个消费者每 10s 处理完一条消息
      envFrom:
      - secretRef:
          name: consumer-secret
      image: imroc/pulsar-demo:main
      imagePullPolicy: Always
      name: consumer
    terminationGracePeriodSeconds: 1
```

配置 ScaledObject

1. 先创建 `TriggerAuthentication` 并引用 `consumer-secret` 中的 TOKEN：

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: consumer-auth
spec:
  secretTargetRef:
  - parameter: bearerToken
    name: consumer-secret
    key: TOKEN
```

2. 创建 ScaledObject（替换高亮行配置）：

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: consumer-scaledobject
spec:
  scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: consumer
  pollingInterval: 15
  idleReplicaCount: 0 # 没有消息时缩到 0
  minReplicaCount: 1
  maxReplicaCount: 100
  triggers:
  - type: pulsar
    metadata:
      adminURL: http://pulsar-xxxxxxxxxxx.tdmq.ap-cd.qcloud.tencenttdmq.com:5005 # 替换 API 调用地址
      topic: persistent://pulsar-xxxxxxxxxxx/test/persist-topic # 替换 Topic
      subscription: my-sub # 替换订阅名称
      isPartitionedTopic: "true" # 如果分区数大于 1，这里就置为 true
      msgBacklogThreshold: "5" # 伸缩阈值，副本数=CEIL(消息堆积数/msgBacklogThreshold)
      activationMsgBacklogThreshold: "1" # 如果当前副本数为 0，只要队列里来新消息了，就将副本置为 1 并启用伸缩
      authModes: bearer # 角色密钥（JWT Token）本质上是 bearer 的认证模式
    authenticationRef:
      name: consumer-auth # 引用前面创建的 TriggerAuthentication
```

查看 HPA

如果配置正确，会自动创建出对应的 HPA 资源。执行如下命令，查看 HPA。

```
$ kubectl get hpa
NAME                              REFERENCE              TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
keda-hpa-consumer-scaledobject    Deployment/consumer    4600m/5 (avg)    1         10        5          31m
```

> **说明：**
> 在上述输出中，可以通过 "TARGETS" 字段反推当前消息堆积数量。以上述输出为例，堆积消息数为4.6 * 5 = 23。

ScaledJob + 超级节点

如果单条消息处理耗时较大，但又需要尽量及时获取处理结果，可以配置 ScaledJob，每当队列中有新消息时，将自动创建一个 Job 来消费，让 Job 的 Pod 调度到超级节点，以实现按需使用计算资源和按量计费。
触发器的配置对于 ScaledObject 与 ScaledJob 完全一致，如需配置 ScaledJob，可参考 ScaledObject 的配置。

基于云原生 API 网关监控指标的水平伸缩

最近更新时间：2025-06-20 16:17:11

概述

云原生 API 网关 是腾讯云上基于 Kong 托管的网关产品，提供丰富的七层流量管理功能，也支持将请求转发到 TKE 集群上的服务。
云原生网关提供了丰富的 Prometheus 监控指标，本文将以指定服务的 QPS 指标为例，介绍如何在 TKE 上利用 KEDA 实现基于云原生 API 网关监控指标的水平伸缩。

操作步骤

配置 Prometheus 采集

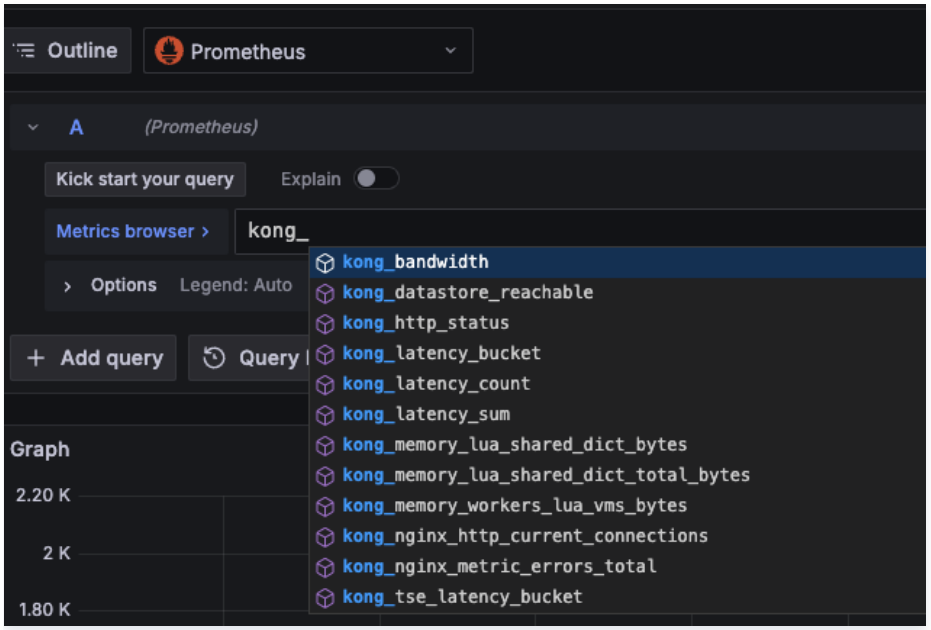
您需要使用腾讯云 Prometheus 来采集云原生 API 网关的监控数据，在云原生 API 网关实例的数据观测页面切换到 Prometheus 标签页：

选择关联腾讯云 Prometheus。复制 Prometheus 实例内网地址 IP，通过 `static_configs` 配置到自建 Prometheus 的采集配置中。代码示例如下：

```
- job_name: apigw
  honor_timestamps: true
  metrics_path: "/metrics"
  scheme: http
  static_configs:
    - targets: ["10.10.12.23:2100", "10.10.12.144:2100"]
```

> **说明：**
> 云原生 API 网关的 metrics 接口地址是：`节点IP:2100/metrics` 。

如果正常采集，您可以在 Prometheus 中查到以 `kong_` 开头的监控指标。如下图所示：



部署测试应用到 TKE 集群

您可以参考以下代码部署一个简单的 nginx 应用到 TKE 集群：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest

---

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

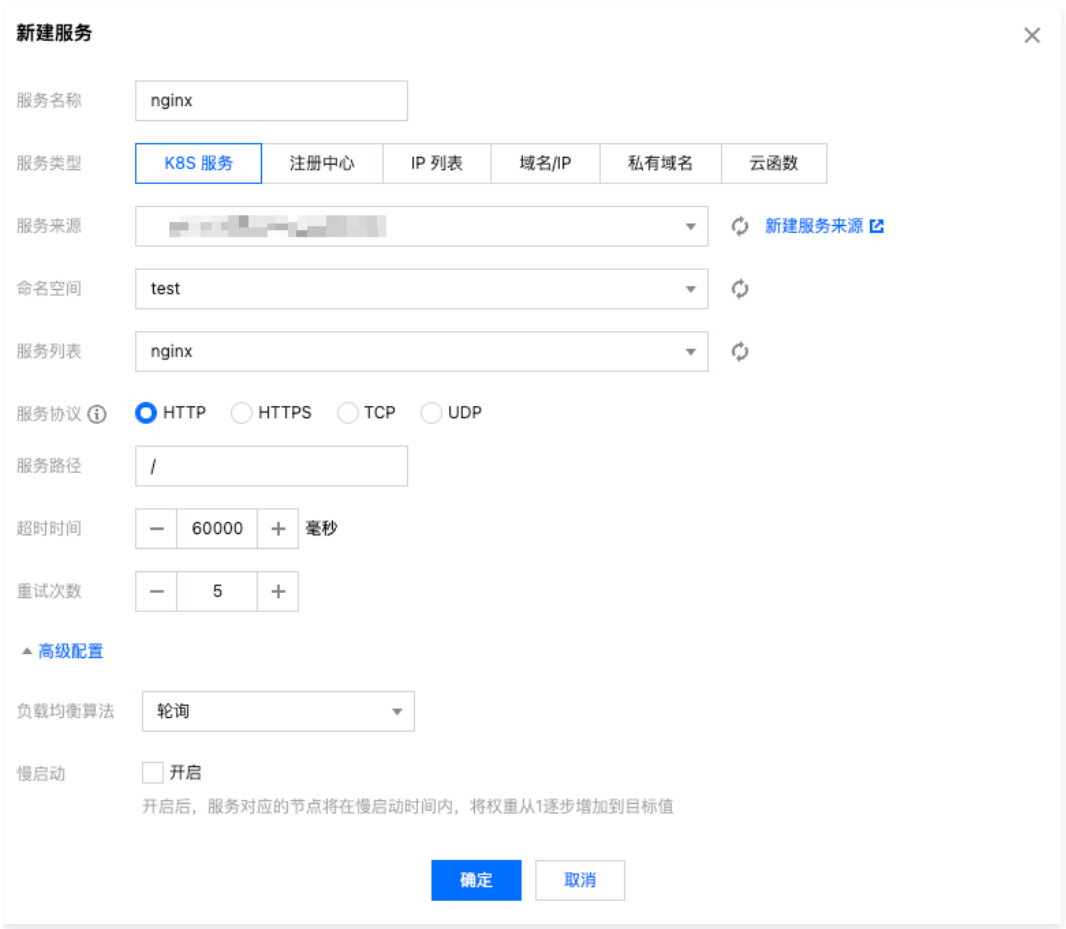
配置云原生 API 网关

在**服务路由**中新建服务来源。如下图所示：

根据您的需求添加容器服务的集群。如下图所示：

在服务路由中新建服务。如下图所示：

新建服务时，从 **K8S 服务**中选择部署了测试应用的集群、命名空间以及服务。如下图所示：



创建完成后，通过导入路由方式创建路由：

根据需求配置规则，详情参考云原生 API 网关的文档。

配置完成后，您可以通过云原生网关访问 TKE 集群中的服务，压测一段时间后，观察 `Prometheus` 中的监控数据是否正常。

配置 KEDA ScaledObject

在安装了 KEDA 的前提下，您可以创建类似以下的 `ScaledObject`：

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: nginx-scaledobject
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  pollingInterval: 15
  minReplicaCount: 1
  maxReplicaCount: 100
```

```
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          policies:
            - periodSeconds: 15
              type: Percent
              value: 900
  triggers:
    - type: prometheus
      metadata:
        serverAddress: http://monitoring-kube-prometheus-prometheus.monitoring.svc.cluster.local:9090
        query: |
          sum(irate(kong_http_status{service="nginx"}[1m]))
        threshold: "300"
```

- `scaleTargetRef` 中填写要被自动扩缩容的工作负载，这里是对 `nginx` 这个工作负载进行自动扩缩容。
- `serverAddress` 中填写 `Prometheus` 的地址，根据实际情况进行修改。
- `query` 中填写查询指标数据的 PromQL，示例中是查 `nginx` 这个服务在云原生 API 网关中的 QPS 值。
- `threshold` 表示扩缩容阈值，300 表示每个 `nginx` pod 平均承受 300 QPS 的阈值，实际的平均 QPS 与这个阈值比较来进行相应的扩缩容操作。

配置 Ingress

云原生 API 网关除了能在控制台配置路由，还可以通过 Ingress 的方式来配置，在云原生 API 网关实例的 Ingress 页面关联 `Kong Ingress Controller` 到容器集群后，就可以在集群里直接创建 Ingress 来配置规则了，示例如下：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  ingressClassName: kong
  rules:
    - host: "example.com"
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx
                port:
                  number: 80
```

> **说明：**
> `ingressClassName` 要指定 `Kong Ingress Controller` 所使用的 IngressClass。

然后在 KEDA 的 `ScaledObject` 里，请根据实际情况修改 PromQL 查询语句写法：

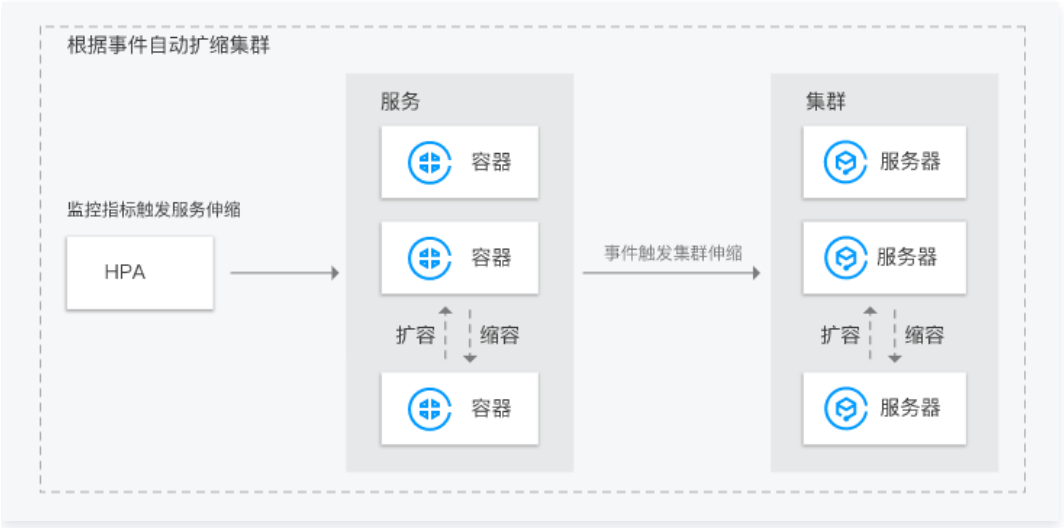
```
sum(irate(kong_http_status{service="test.nginx.pnum-80"}[1m]))
```

> **说明：**
> `service` 的格式为 `<Ingress 所在命名空间>.<引用的 Service 名称>.pnum-<Service 端口>` 。

集群弹性伸缩实践

最近更新时间：2023-05-17 15:41:08

腾讯云容器服务（Tencent Kubernetes Engine，TKE）提供集群和服务两个层级的弹性伸缩能力， 能够根据业务运行情况监控容器的 CPU、内存、带宽等指标，进行自动扩缩服务。同时您可以根据容器的部署情况，在容器不够资源分配或者有过多剩余资源的情况下自动伸缩集群。如下图所示：



集群弹性伸缩特点

TKE 支持用户为集群开启自动伸缩，帮助用户高效管理计算资源，用户可根据业务设置伸缩策略，而集群弹性伸缩策略具备以下特点：

- 根据业务负载情况，动态实时自动创建和释放云服务器（Cloud Virtual Machine，CVM），帮助用户以最合适的实例数量应对业务情况。全程无需人工干预，为用户免去人工部署负担。
- 帮助用户以最适合的节点资源面对业务情况。当业务需求增加时，为用户无缝地自动增加适量 CVM 到容器集群。当业务需求下降时，为用户自动削减不需要的 CVM，提高设备利用率，为用户节省部署和实例成本。

集群弹性伸缩功能说明

Kubernetes cluster autoscaling 基本功能

- 支持设置多伸缩组。
- 支持设置扩容和缩容策略，详情请参见 Cluster Autoscaler 。

TKE 伸缩组扩展功能

- 支持创建新的机型的伸缩组（推荐）。
- 支持从集群内的节点作为模板创建伸缩组。
- 支持竞价实例伸缩组（推荐）。
- 支持机型售罄时自动适配合适的伸缩组。
- 支持跨可用区配置伸缩组。

集群弹性伸缩限制

- 集群弹性伸缩可扩容节点数量受私有网络 、容器网络、TKE 集群节点配额及可购买云服务器配额限制。
- 扩容节点受机型当前售卖情况限制。若机型出现售罄，将无法扩容节点，建议配置多个伸缩组。
- 需要配置工作负载下容器的 request 值。自动扩容的触发条件是集群中存在由于资源不足而无法调度的 Pod，而判断资源是否充足正是基于 Pod 的 request 来进行的。
- 建议不要启用基于监控指标的节点弹性伸缩。
- 删除伸缩组会同时销毁伸缩组内的 CVM，请谨慎操作。

集群伸缩组配置

- 多伸缩组配置建议（推荐）
  集群存在多个伸缩组时， 自动扩缩容组件将按照您选择的扩容算法选择伸缩组进行扩容，一次选择一个伸缩组。 当出现目标伸缩组由于售罄等原因扩容失败时，将会置为休眠一段时间，同时触发重新选择第二匹配的伸缩组进行扩容。
  - 随机：随机选择一个伸缩组进行扩容。
  - Most-pods：根据当前 pending 的 Pod 和伸缩组的机型，判断并选择能调度更多 Pod 的伸缩组进行扩容。
  - Least-waste：根据当前 pending 的 Pod 和伸缩组的机型，判断并选择 Pod 调度后资源剩余更少的伸缩组进行扩容。

  建议您在集群内配置多个不同机型的伸缩组，以防止某种机型出现售罄的情况。同时可以使用竞价机型和常规机型混用以减少成本。
- 集群单伸缩组配置
  若您仅接受单种机型作为集群的扩容机型， 推荐您将伸缩组配置到多个子网多个不同的可用区下。

使用 tke-autoscaling-placeholder 实现秒级弹性伸缩

最近更新时间：2023-11-10 17:39:23

操作场景

如 TKE 集群配置了节点池并启用弹性伸缩，则在节点资源不够时可以触发节点的自动扩容（自动购买机器并加入集群），该扩容流程需要一定的时间才能完成，在一些流量突高的场景，该扩容速度可能会显得太慢，影响业务正常运行。而 `tke-autoscaling-placeholder` 可以用于在 TKE 上实现秒级伸缩，应对流量突高场景。本文将介绍如何使用 `tke-autoscaling-placeholder` 实现秒级弹性伸缩。

实现原理

`tke-autoscaling-placeholder` 利用低优先级的 Pod（带有 request 的 pause 容器，实际资源消耗较低）对资源进行提前占位，为可能出现流量突增的高优先级业务预留一部分资源作为缓冲。当需要扩容 Pod 时，高优先级的 Pod 可以快速抢占低优先级 Pod 的资源进行调度，这将导致低优先级的 `tke-autoscaling-placeholder` 的 Pod 进入 Pending 状态，如果配置了节点池并启用弹性伸缩，将会触发节点的扩容。通过这种资源缓冲机制，即使节点扩容速度较慢，也能确保部分 Pod 能够迅速扩容并调度，实现秒级伸缩。根据实际需求，可以调整 `tke-autoscaling-placeholder` 的 request 或副本数，以便调整预留的缓冲资源量。

使用限制

使用 `tke-autoscaling-placeholder` 应用，集群版本需要在1.18以上。

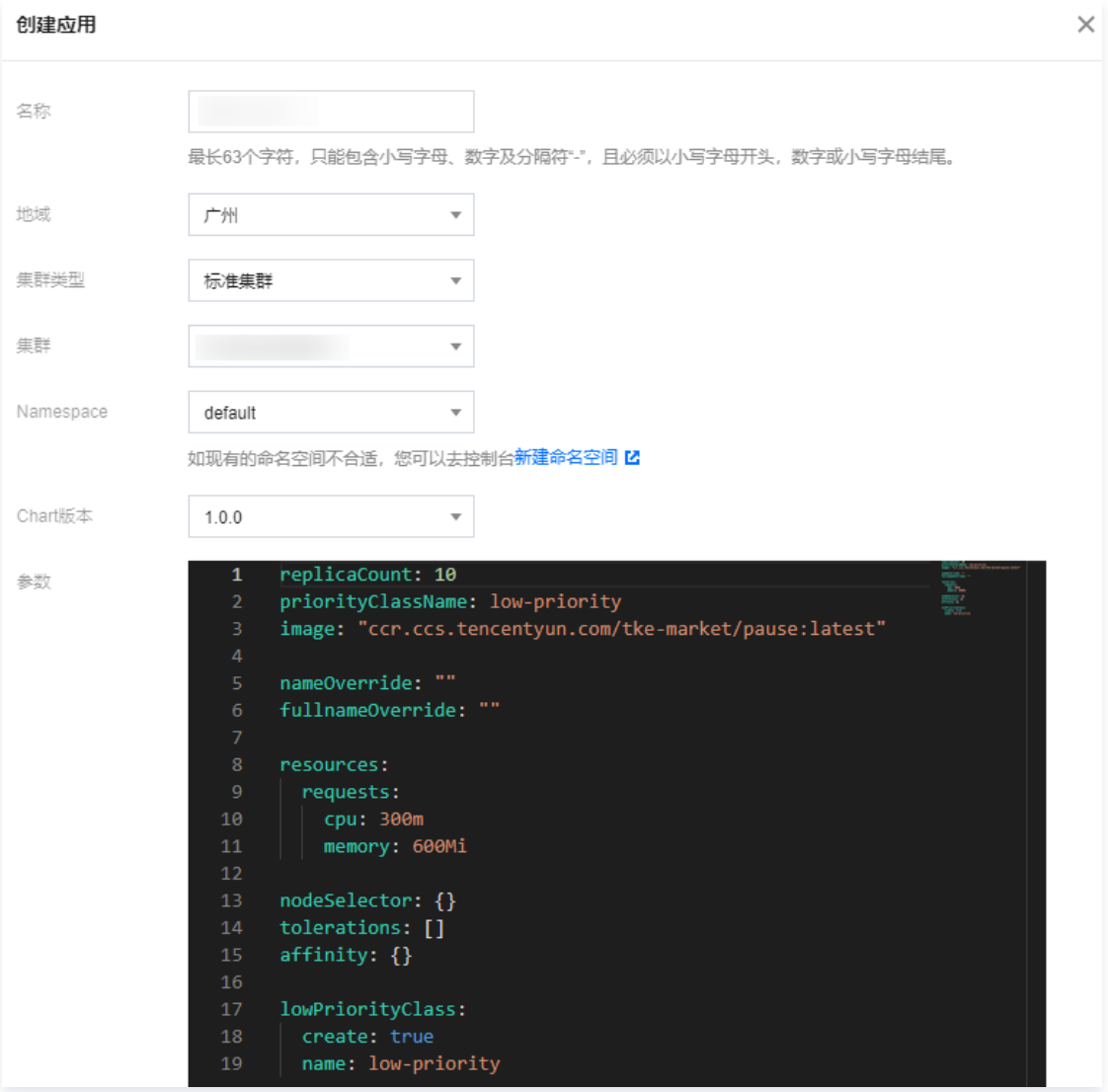
操作步骤

安装 tke-autoscaling-placeholder

1. 登录容器服务控制台，选择左侧导航栏中的 应用市场 。
2. 在**应用市场**页面，输入关键词 **tke-autoscaling-placeholder** 进行搜索，找到该应用。如下图所示：

3. 单击应用，在应用详情中，单击基本信息模块中的**创建应用**。

4. 在**创建应用**页面，按需配置并创建应用。如下图所示：



配置说明如下：

- **名称**：输入应用名称。最长63个字符，只能包含小写字母、数字及分隔符“-”，且必须以小写字母开头，数字或小写字母结尾。
- **地域**：选择需要部署的所在地域。
- **集群类型**：选择**标准集群**。
- **集群**：选择需要部署的集群 ID。
- **Namespace**：选择需要部署的 namespace。
- **Chart 版本**：选择需要部署的 Chart 版本。
- **参数**：配置参数中最重要的是 `replicaCount` 与 `resources.request`，分别表示 `tke-autoscaling-placeholder` 的副本数与每个副本占位的资源大小，它们共同决定缓冲资源的大小，可以根据流量突高需要的额外资源量来估算进行设置。`tke-autoscaling-placeholder` 完整参数配置说明请参考如下表格：

参数名称	描述	默认值
replicaCount	placeholder 的副本数	10
image	placeholder 的镜像地址	ccr.ccs.tencentyun.com/tke-market/pause:latest
resources.requests.cpu	单个 placeholder 副本占位的 CPU 资源大小	300m
resources.requests.memory	单个 placeholder 副本占位的内存大小	600Mi

lowPriorityClass.create	是否创建低优先级的 PriorityClass (用于被 placeholder 引用)	true
lowPriorityClass.name	低优先级的 PriorityClass 的名称	low-priority
nodeSelector	指定 placeholder 被调度到带有特定 label 的节点	{}
tolerations	指定 placeholder 要容忍的污点	[]
affinity	指定 placeholder 的亲和性配置	{}

5. 单击**创建**，部署 tke-autoscaling-placeholder 应用。
6. 执行如下命令，查看进行资源占位的 Pod 是否启动成功。

```
kubectl get pod -n default
```

示例如下：

```
$ kubectl get pod -n default
tke-autoscaling-placeholder-b58fd9d5d-2p6ww   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-55jw7   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-6rq9r   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-7c95t   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-bfg8r   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-cfqt6   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-gmfmr   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-grwlh   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-ph7vl   1/1     Running   0          8s
tke-autoscaling-placeholder-b58fd9d5d-xmrmv   1/1     Running   0          8s
```

部署高优先级 Pod

`tke-autoscaling-placeholder` 默认优先级较低，其中业务 Pod 可以指定一个高优先的 PriorityClass，方便抢占资源实现快速扩容。如果还未创建 PriorityClass，您可以参考如下示例进行创建：

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "high priority class"
```

在业务 Pod 中指定 `priorityClassName` 为高优先的 PriorityClass。示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 8
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
```

```
    spec:
      priorityClassName: high-priority # 这里指定高优先的 PriorityClass
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 400m
            memory: 800Mi
```

当集群节点资源不够时，扩容出来的高优先级业务 Pod 就可以将低优先级的 `tke-autoscaling-placeholder` 的 Pod 资源抢占过来并调度上，此时 `tke-autoscaling-placeholder` 的 Pod 状态将变成 Pending。示例如下：

```
$ kubectl get pod -n default
NAME                                          READY   STATUS    RESTARTS   AGE
nginx-bf79bbc8b-5kxcw                         1/1     Running   0          23s
nginx-bf79bbc8b-5xhbx                         1/1     Running   0          23s
nginx-bf79bbc8b-bmzff                         1/1     Running   0          23s
nginx-bf79bbc8b-l2vht                         1/1     Running   0          23s
nginx-bf79bbc8b-q84jq                         1/1     Running   0          23s
nginx-bf79bbc8b-tq2sx                         1/1     Running   0          23s
nginx-bf79bbc8b-tqgxg                         1/1     Running   0          23s
nginx-bf79bbc8b-wz5w5                         1/1     Running   0          23s
tke-autoscaling-placeholder-b58fd9d5d-255r8   0/1     Pending   0          23s
tke-autoscaling-placeholder-b58fd9d5d-4vt8r   0/1     Pending   0          23s
tke-autoscaling-placeholder-b58fd9d5d-55jw7   1/1     Running   0          94m
tke-autoscaling-placeholder-b58fd9d5d-7c95t   1/1     Running   0          94m
tke-autoscaling-placeholder-b58fd9d5d-ph7vl   1/1     Running   0          94m
tke-autoscaling-placeholder-b58fd9d5d-qjrsx   0/1     Pending   0          23s
tke-autoscaling-placeholder-b58fd9d5d-t5qdm   0/1     Pending   0          23s
tke-autoscaling-placeholder-b58fd9d5d-tgvmw   0/1     Pending   0          23s
tke-autoscaling-placeholder-b58fd9d5d-xmrmv   1/1     Running   0          94m
tke-autoscaling-placeholder-b58fd9d5d-zxtwp   0/1     Pending   0          23s
```

如果配置了节点池弹性伸缩，则将触发节点的扩容，虽然节点速度慢，但由于缓冲资源已分配到业务 Pod，业务能够快速得到扩容，因此不会影响业务的正常运行。

总结

本文介绍了用于实现秒级伸缩的工具 `tke-autoscaling-placeholder`，巧妙的利用了 Pod 优先级与抢占的特点，提前部署一些用于占位资源的低优先级“空 Pod” 作为缓冲资源填充，在流量突高并且集群资源不够的情况下抢占这些低优先级的“空 Pod” 的资源，同时触发节点扩容，实现在资源紧张的情况下也能做到秒级伸缩，不影响业务正常运行。

相关文档

- Pod 优先级与抢占
- 创建节点池

在 TKE 上安装 metrics-server

最近更新时间：2023-11-10 17:39:23

操作场景

metrics-server 可实现 Kubernetes 的 Resource Metrics API（metrics.k8s.io），通过此 API 可以查询 Pod 与 Node 的部分监控指标，Pod 的监控指标用于 HPA、VPA 与 `kubectl top pods` 命令，而 Node 指标目前只用于 `kubectl top nodes` 命令。容器服务 TKE 自带 Resource Metrics API 的实现，指向 TKE 侧维护的 metrics-server，该 metrics-server 的实现基于社区版，且目前提供 Pod 的监控指标。
将 metrics-server 安装到集群后，可以通过 `kubectl top nodes` 获取节点的监控概览，以替换 Resource Metrics API 的实现。容器服务控制台创建的 HPA 不会用到 Resource Metrics，仅使用 Custom Metrics，因此安装 metrics-server 不会影响在 TKE 控制台创建的 HPA。本文将介绍如何在 TKE 上安装 metrics-server。

操作步骤

下载 yaml 部署文件

执行以下命令，下载 metrics-server 官方的部署 yaml:

```
wget https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.5.0/components.yaml
```

修改 metrics-server 启动参数

metrics-server 会请求每台节点的 kubelet 接口来获取监控数据，接口通过 HTTPS 暴露，但 TKE 节点的 kubelet 使用的是自签证书，若 metrics-server 直接请求 kubelet 接口，将产生证书校验失败的错误，因此需要在 components.yaml 文件中加上 `--kubelet-insecure-tls` 启动参数。且由于 metrics-server 官方镜像仓库存储在 `k8s.gcr.io`，国内可能无法直接拉取，您可以自行同步到 CCR 或使用已同步的镜像 `ccr.ccs.tencentyun.com/mirrors/metrics-server:v0.5.0`。
components.yaml 文件修改示例如下：

```
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=15s
        - --kubelet-insecure-tls # 加上该启动参数
        image: ccr.ccs.tencentyun.com/mirrors/metrics-server:v0.5.0 # 国内集群，请替换成这个镜像
```

部署 metrics-server

修改 components.yaml 之后，执行以下命令，通过 kubectl 一键部署到集群：

```
kubectl apply -f components.yaml
```

检查运行状态

1. 执行以下命令，检查 metrics-server 是否正常启动。示例如下：

```
$ kubectl get pod -n kube-system | grep metrics-server
metrics-server-f976cb7d-8hssz          1/1     Running   0          1m
```

2. 执行以下命令，检查配置文件。示例如下：

```
$ kubectl get --raw /apis/metrics.k8s.io/v1beta1  | jq
{
  "kind": "APIResourceList",
```

```
  "apiVersion": "v1",
  "groupVersion": "metrics.k8s.io/v1beta1",
  "resources": [
 {
   "name": "nodes",
   "singularName": "",
   "namespaced": false,
   "kind": "NodeMetrics",
   "verbs": [
     "get",
     "list"
   ]
 },
 {
   "name": "pods",
   "singularName": "",
   "namespaced": true,
   "kind": "PodMetrics",
   "verbs": [
     "get",
     "list"
   ]
 }
  ]
}
```

3. 执行以下命令，检查节点占用性能情况。示例如下：

```
$ kubectl top nodes
NAME     CPU(cores)    CPU%    MEMORY(bytes)    MEMORY%
test1    1382m         35%     2943Mi           44%
test2    397m          10%     3316Mi           49%
test3    81m           8%      464Mi            77%
```

在 TKE 上使用自定义指标进行弹性伸缩

最近更新时间：2025-06-05 18:30:12

操作场景

容器服务 TKE 基于 Custom Metrics API 支持许多用于弹性伸缩的指标，涵盖 CPU、内存、硬盘、网络以及 GPU 相关的指标，覆盖绝大多数的 HPA 弹性伸缩场景，详细列表请参见 自动伸缩指标说明 。针对例如基于业务单副本 QPS 大小来进行自动扩缩容等复杂场景，可通过安装 prometheus-adapter 来实现自动扩缩容。而 Kubernetes 提供 Custom Metrics API 与 External Metrics API 来对 HPA 指标进行扩展，让用户能够根据实际需求进行自定义。prometheus-adapter 支持以上两种 API，在实际环境中，使用 Custom Metrics API 即可满足大部分场景。本文将介绍如何通过 Custom Metrics API 实现使用自定义指标进行弹性伸缩。

前提条件

- 已创建1.14或以上版本的 TKE 集群，详情请参见 创建集群 。
- 已购买 Prometheus 实例，并完成实例和 TKE 集群的关联，详情请参见 创建 Prometheus 实例 、关联集群 。
- 已安装 Helm 。

操作步骤

暴露监控指标

本文以 Golang 业务程序为例，该示例程序暴露了 `httpserver_requests_total` 指标，并记录 HTTP 的请求，通过该指标可以计算出业务程序的 QPS 值。示例如下：

```
package main

import (
        "github.com/prometheus/client_golang/prometheus"
        "github.com/prometheus/client_golang/prometheus/promhttp"
        "net/http"
        "strconv"
)

var (
HTTPRequests = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "httpserver_requests_total",
            Help: "Number of the http requests received since the server started",
        },
        []string{"status"},
    )
)

func init() {
    prometheus.MustRegister(HTTPRequests)
}

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        path := r.URL.Path
        code := 200
        switch path {
        case "/test":
            w.WriteHeader(200)
            w.Write([]byte("OK"))
        case "/metrics":
            promhttp.Handler().ServeHTTP(w, r)
        default:
            w.WriteHeader(404)
```

```
            w.Write([]byte("Not Found"))
        }
        HTTPRequests.WithLabelValues(strconv.Itoa(code)).Inc()
    })
    http.ListenAndServe(":80", nil)
}
```

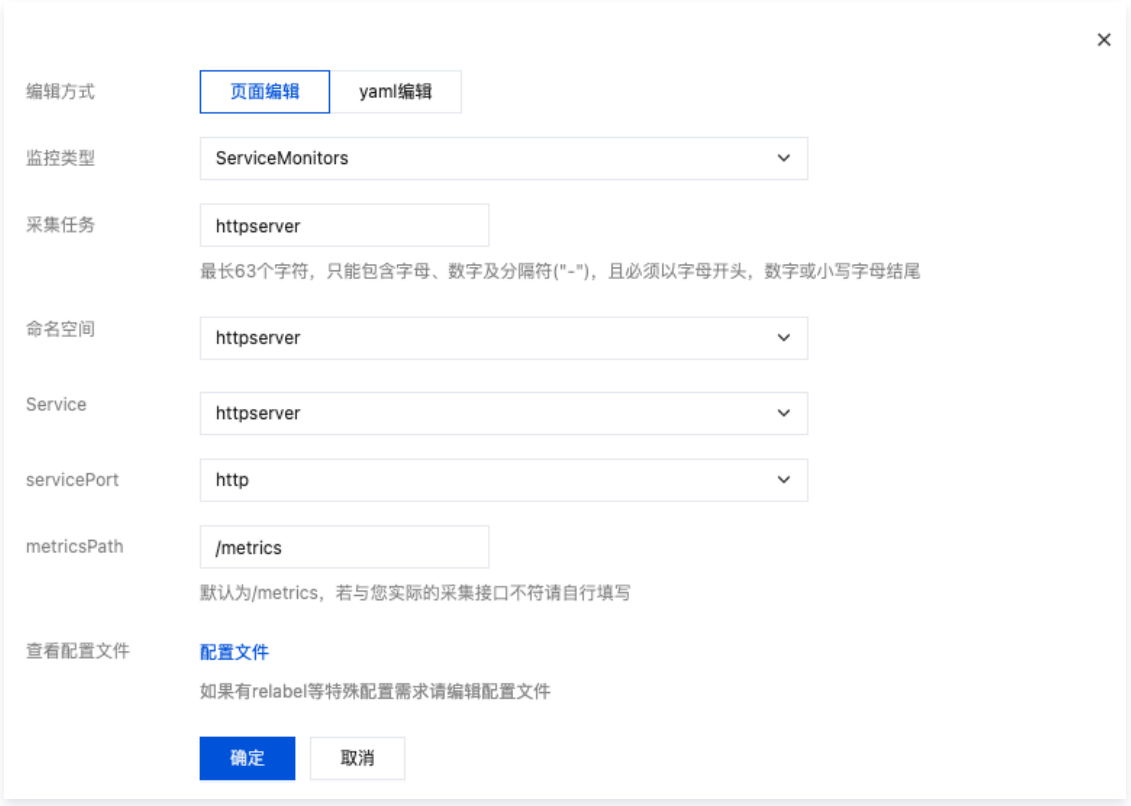
部署业务程序

将前面的程序打包成容器镜像，然后部署到集群，例如使用 Deployment 部署：

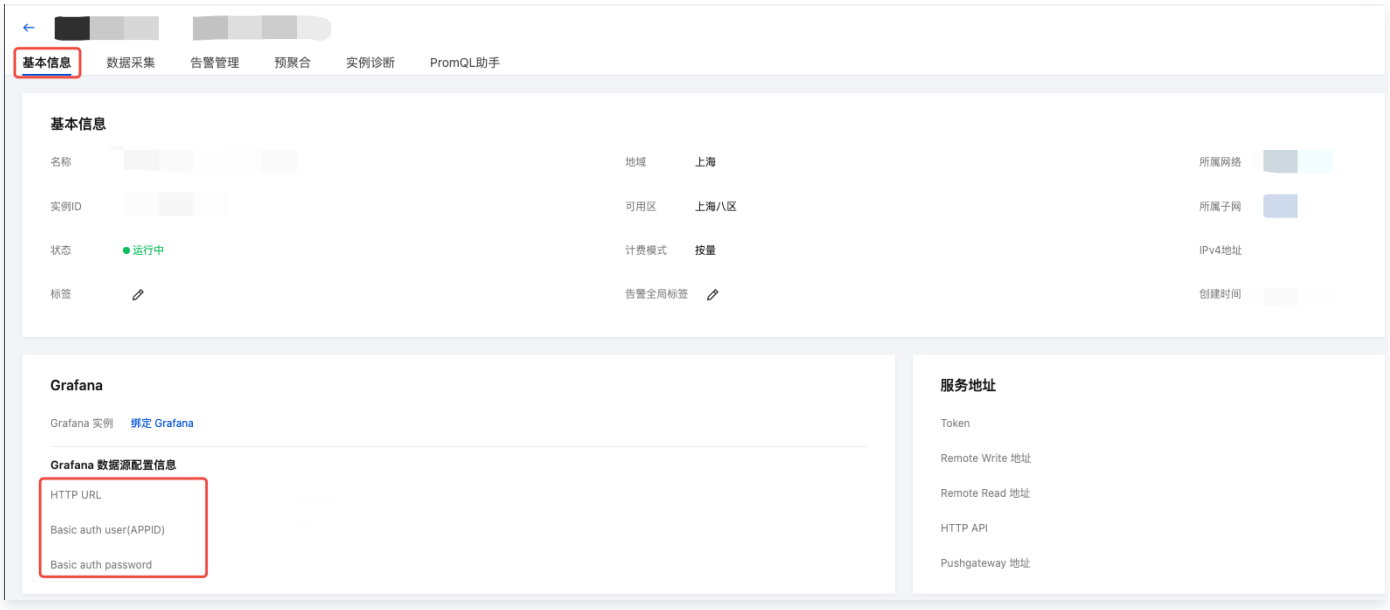
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpserver
  namespace: httpserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpserver
  template:
    metadata:
      labels:
        app: httpserver
    spec:
      containers:
      - name: httpserver
        image: ccr.ccs.tencentyun.com/tkedemo/custom-metrics-demo:v1.0.0
        imagePullPolicy: Always

---

apiVersion: v1
kind: Service
metadata:
  name: httpserver
  namespace: httpserver
  labels:
    app: httpserver
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "http"
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: httpserver
```

Prometheus 采集业务监控

1. 如果您已完成 Prometheus 监控实例与 TKE 集群的关联，可直接登录 容器服务控制台，选择左侧导航栏的 **Prometheus 监控**。
2. 选择监控实例，在**数据采集> 集成容器服务**页面，找到目标实例，单击右侧的**数据采集配置**，进入数据采集配置列表页。
3. 单击**新建自定义监控**，为已部署的业务程序配置新的采集规则。单击**确定**，如下图所示：



4. 单击左上角**基本信息**页面，获取访问 Prometheus API 的地址（HTTP URL）和鉴权账号信息（Basic auth user 和 Basic auth password），后续安装 prometheus-adapter 时使用，如下图所示：



安装 prometheus-adapter

1. 使用 Helm 安装 prometheus-adapter，安装前请确定并配置自定义指标。按照上文 暴露监控指标 中的示例，在业务中使用 `httpserver_requests_total` 指标来记录 HTTP 请求，因此可以通过如下的 PromQL 计算出每个业务 Pod 的 QPS 监控。示例如下：

```
sum(rate(http_requests_total[2m])) by (pod)
```

2. 将其转换为 prometheus-adapter 的配置，创建 `values.yaml`，内容如下：

```
rules:
   default: false
   custom:
   - seriesQuery: 'httpserver_requests_total'
     resources:
       template: <<.Resource>>
     name:
       matches: "httpserver_requests_total"
       as: "httpserver_requests_qps" # PromQL 计算出来的 QPS 指标
     metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)
prometheus:
   url: http://127.0.0.1 # 替换上一步获取到的 Prometheus API 的地址（HTTP URL）（不写端口）
   port: 9090
extraArguments:
- --prometheus-header=Authorization=Basic {token}# 其中{token} 为您从控制台获取的 Basic auth user:Basic auth password 字符串的 base64 编码
```

3. 执行以下 Helm 命令安装 prometheus-adapter，示例如下：

> ⚠ **注意：**
> 安装前需要删除 TKE 已经注册的 Custom Metrics API，删除命令如下：
>
> ```
> kubectl delete apiservice v1beta1.custom.metrics.k8s.io
> ```

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
# Helm 3
helm install prometheus-adapter prometheus-community/prometheus-adapter -f values.yaml
# Helm 2
# helm install --name prometheus-adapter prometheus-community/prometheus-adapter -f values.yaml
```

4. 添加 prometheus 认证鉴权参数。

   当前社区提供的 chart 中没有暴露认证鉴权相关的入参，会导致认证鉴权失败无法正常连接 TMP 服务，为了解决这个问题，您可以查看 社区文档。解决方案要求您手动修改 Prometheus Adapter deployment，在 adapter 启动参数中添加 `--prometheus-header=Authorization=Basic {token}`，其中 `{token}` 为您从控制台获取的 Basic auth user:Basic auth password 的 base64 编码。

测试验证

若安装正确，执行以下命令，可以查看到 Custom Metrics API 返回配置的 QPS 相关指标。示例如下：

```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
```

```
      "name": "jobs.batch/httpserver_requests_qps",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "pods/httpserver_requests_qps",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "namespaces/httpserver_requests_qps",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    }
  ]
}
```

执行以下命令，可以查看 Pod 的 QPS 值。示例如下：

> **说明：**
> 下述示例 QPS 为500m，表示 QPS 值为0.5。

```
$ kubectl get --raw
/apis/custom.metrics.k8s.io/v1beta1/namespaces/httpserver/pods/*/httpserver_requests_qps
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink":
"/apis/custom.metrics.k8s.io/v1beta1/namespaces/httpserver/pods/%2A/httpserver_requests_qps"
  },
  "items": [
    {
      "describedObject": {
        "kind": "Pod",
        "namespace": "httpserver",
        "name": "httpserver-6f94475d45-7rln9",
        "apiVersion": "/v1"
      },
      "metricName": "httpserver_requests_qps",
      "timestamp": "2020-11-17T09:14:36Z",
      "value": "500m",
      "selector": null
    }
  ]
}
```

测试 HPA

假如设置每个业务 Pod 的平均 QPS 达到50时将触发扩容，最小副本为1个，最大副本为1000个，则配置示例如下：

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: httpserver
  namespace: httpserver
spec:
  minReplicas: 1
  maxReplicas: 1000
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: httpserver
  metrics:
  - type: Pods
    pods:
      metric:
        name: httpserver_requests_qps
      target:
        averageValue: 50
        type: AverageValue
```

执行以下命令对业务进行压测，观察是否自动扩容。示例如下：

```
# 使用 wrk 或者其他 http 压测工具对 httpserver 服务进行压测。
$ kubectl proxy --port=8080 > /dev/null 2>&1 &
$ wrk -t12 -c3000 -d60s
http://localhost:8080/api/v1/namespaces/httpserver/services/httpserver:http/proxy/test
$ kubectl get hpa -n httpserver
NAME         REFERENCE               TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
httpserver   Deployment/httpserver   35266m/50   1         1000      2          50m

# 观察 hpa 的扩容情况
$ kubectl get pods -n httpserver
NAME                          READY   STATUS              RESTARTS   AGE
httpserver-7f8dffd449-pgsb7   0/1     ContainerCreating   0          4s
httpserver-7f8dffd449-wsl95   1/1     Running             0          93s
httpserver-7f8dffd449-pgsb7   1/1     Running             0          4s
```

若扩容正常，则说明已实现 HPA 基于业务自定义指标进行弹性伸缩。

在 TKE 上利用 HPA 实现业务的弹性伸缩

最近更新时间：2024-09-03 17:12:51

概述

Kubernetes Pod 水平自动扩缩（Horizontal Pod Autoscaler，以下简称 HPA）可以基于 CPU 利用率、内存利用率和其他自定义的度量指标自动扩缩 Pod 的副本数量，以使得工作负载服务的整体度量水平与用户所设定的目标值匹配。本文将介绍和使用腾讯云容器服务 TKE 的 HPA 功能实现 Pod 自动水平扩缩容。

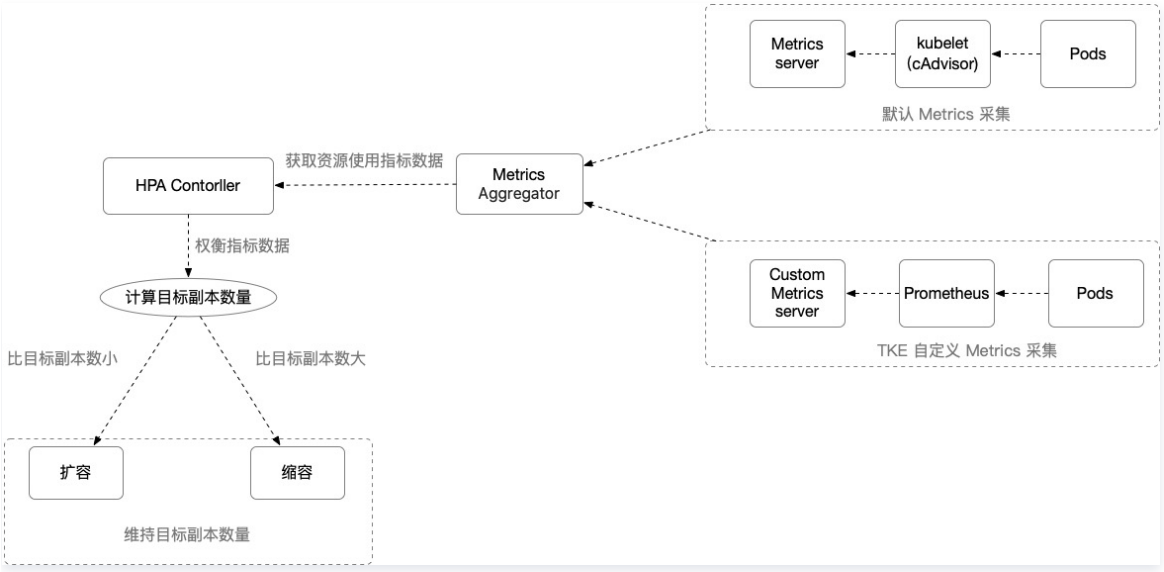
使用场景

HPA 自动伸缩特性使容器服务具有非常灵活的自适应能力，能够在用户设定内快速扩容多个 Pod 副本来应对业务负载的急剧飙升，也可以在业务负载变小的情况下根据实际情况适当缩容来节省计算资源给其他的服务，整个过程自动化无需人为干预，适合服务波动较大、服务数量多且需要频繁扩缩容的业务场景，例如：电商服务、线上教育、金融服务等。

原理概述

Pod 水平自动扩缩特性由 Kubernetes API 资源和控制器实现。资源利用指标决定控制器的行为，控制器会周期性的根据 Pod 资源利用情况调整服务 Pod 的副本数量，以使得工作负载的度量水平与用户所设定的目标值匹配。其扩缩容流程如下图所示：

> **注意：**
> Pod 自动水平扩缩不适用于无法扩缩的对象，例如 DaemonSet 资源。



重点内容说明：

- **HPA Controller**：控制 HPA 扩缩逻辑的控制组件。
- **Metrics Aggregator**：度量指标聚合器。通常情况下，控制器将从一系列的聚合 API（`metrics.k8s.io`、`custom.metrics.k8s.io` 和 `external.metrics.k8s.io`）中获取度量值。`metrics.k8s.io` API 通常由 Metrics 服务器提供，社区版可提供基本的 CPU、内存度量类型。相比于社区版，TKE 使用自定义 Metrics Server 采集可支持更广泛的 HPA 的度量指标触发类型，提供包括 CPU、内存、硬盘、网络和 GPU 相关指标，更多详细内容请参见 TKE 自动伸缩指标说明。

  > **说明：**
  > 控制器也可从 Heapster 获取指标。但自 Kubernetes 1.11 版本起，从 Heapster 获取指标特性的方式已废弃。

- **HPA 计算目标副本数算法**：TKE HPA 扩缩容算法请参见 工作原理，更多详细算法请参见 算法细节。

前提条件

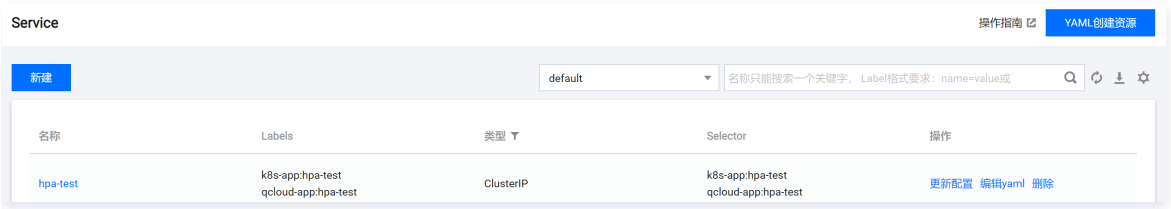
- 已 注册腾讯云账户。
- 已登录 腾讯云容器服务控制台。

- 已创建 TKE 集群。关于创建集群，详情请参见 创建集群 。

操作步骤

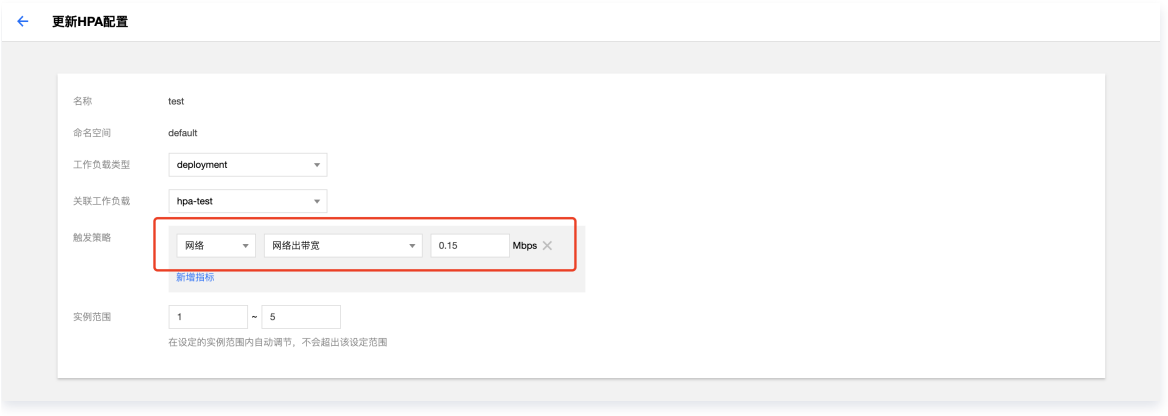
部署测试工作负载和服务

以 Deployment 资源类型的工作负载为例，创建一个单副本数，服务类型为 Web 服务的 “hpa-test” 工作负载和 ClusterIp 类型的 Service。在容器服务控制台创建 Deployment 类型工作负载方法请参见 Deployment 管理，在容器服务控制台创建 ClusterIP 类型的 Service 方法请参见 创建Service 。本示例创建结果如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：



配置 HPA

在容器服务控制台为测试工作负载绑定一个 HPA 配置，关于如何绑定配置 HPA 请参见 HPA 操作步骤，本文以配置当网络出带宽达到0.15Mbps（150Kbps）时触发扩容的策略为例。如下图所示：



功能验证

模拟扩容过程

执行以下命令，在集群中启动一个临时 Pod 对配置的 HPA 功能进行测试（模拟客户端）：

```
kubectl run -it --image alpine hpa-test --restart=Never --rm /bin/sh

# 登录进入这个临时的容器
kubectl exec --stdin --tty hpa-test -- /bin/sh
```

在临时 Pod 中执行以下命令，模拟在短时间内用大量请求访问 "hpa-test" 服务使出口流量带宽增大：

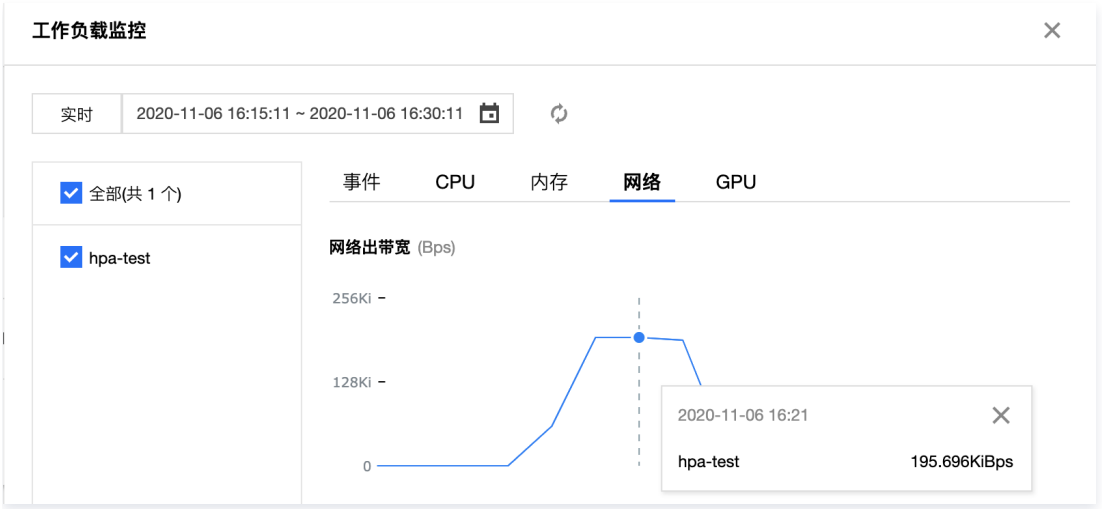
```
# hpa-test.default.svc.cluster.local 为服务在集群中的域名，当需要停止脚本时按 Ctrl+C 即可
```

```
while true; do wget -q -O - hpa-test.default.svc.cluster.local; done
```

在测试 Pod 中执行模拟请求命令后，通过观察工作负载的 Pod 数量监控，发现在16:21分时工作负载扩容副本数量至2个，由此可推断出已经触发了 HPA 的扩容事件。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：

再通过工作负载的网络出口带宽监控可以观察到在16:21时网络出口带宽增至大概196Kbps，已经超过 HPA 设定的网络出口带宽目标值，进一步证明此时触发 HPA 扩缩容算法，扩容了一个副本数来满足设定的目标值，故工作负载的副本数量变成了2个。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：

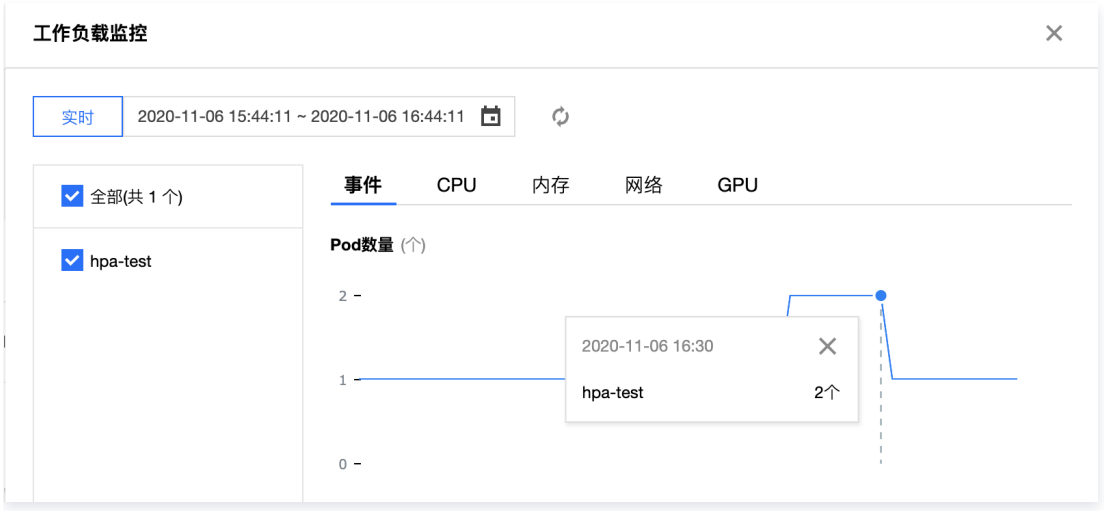
> **注意：**
> HPA 扩缩容算法 不只以公式计算维度去控制扩缩容逻辑，而会多维度去衡量是否需要扩容或缩容，所以在实际情况中可能和预期会稍有偏差，详情可参见 算法细节 。



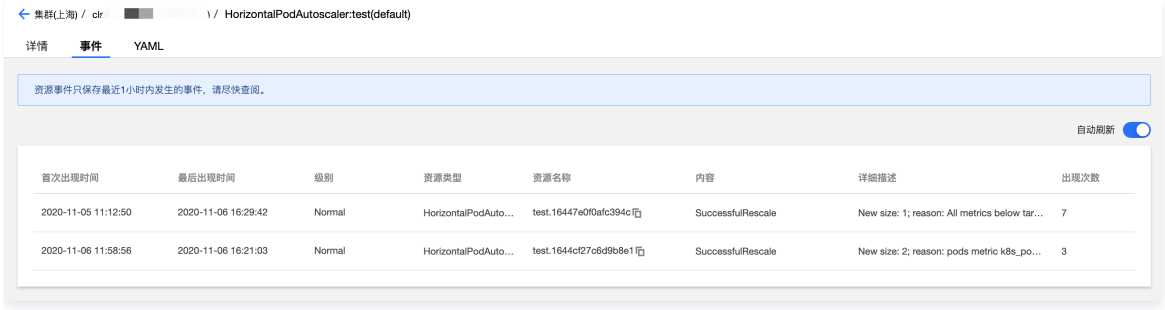
模拟缩容过程

模拟缩容过程时，在16:24左右手动停止执行模拟请求的命令，从监控可以观察到此时网络出口带宽值下降到扩容前位置，按照 HPA 的逻辑，此时已经满足工作负载缩容的条件。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：

但从下图工作负载的 Pod 数量监控可以看出，工作负载在16:30分时才触发了 HPA 的缩容，原因是触发 HPA 缩容后有默认5分钟容忍的时间算法，以防止度量指标短时间波动导致的频繁的扩缩容，详情请参见 稳定窗口 。从下图可以看出工作负载副本数在停止命令5分钟后按照 HPA 扩缩容算法 缩容到了最初设定的1个副本数。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：

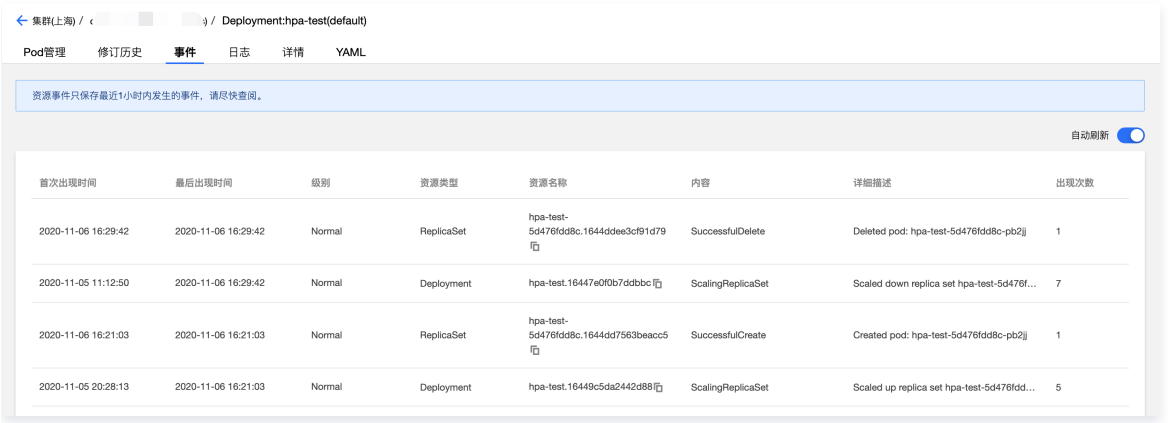


当 TKE 发生 HPA 扩缩容事件时，会在对应的 HPA 实例的事件列表展示。需要注意的是事件通知列表的时间分为 “首次出现时间” 和 “最后出现时间”，“首次出现时间” 表示相同事件第一次出现的时间，“最后出现时间” 为相同事件出现的最新时间，所以从下图事件列表 “最后出现时间” 字段可以看到本示例扩容事件时间点是16:21:03，缩容事件时间是16.29:42，时间点与工作负载监控看到的时间点相吻合。如下图所示：



此外，工作负载事件列表也会记录 HPA 发生时工作负载的增删副本数事件，从下图可以看出工作负载扩缩容时间点与 HPA 事件列表的时间点也是吻合的，增

加副本数时间点是16:21:03，减少副本数时间点是16:29:42。



总结

在本示例中主要演示了 TKE 的 HPA 功能，即使用 TKE 自定义的网络出口带宽度量类型作为工作负载 HPA 的扩缩容度量指标：

- 当工作负载实际度量值超过 HPA 配置的度量目标值时，HPA 根据扩容算法计算出合适的副本数实现水平扩容，保证工作负载的度量指标满足预期及工作负载健康稳定运行。
- 当实际度量值远低于 HPA 配置的度量目标值时，HPA 会在容忍时间后计算合适的副本数实现水平缩容，适当释放闲置资源，达到提升资源利用率的目的，并且整个过程在 HPA 和工作负载事件列表都会有相应的事件记录，使整个工作负载水平扩缩容全程可追溯。

根据不同业务场景调节 HPA 扩缩容灵敏度

最近更新时间：2023-05-17 15:41:09

HPA v2beta2 版本开始支持调节扩缩容速率

在 K8S 1.18之前，HPA 扩容是无法调整灵敏度的：

- 对于缩容，由 `kube-controller-manager` 的 `--horizontal-pod-autoscaler-downscale-stabilization-window` 参数控制缩容时间窗口，默认 5 分钟，即负载减小后至少需要等 5 分钟才会缩容。
- 对于扩容，由 hpa controller 固定的算法、硬编码的常量因子来控制扩容速度，无法自定义。

这样的设计逻辑导致用户无法自定义 HPA 的扩缩容速率，而不同的业务场景对于扩容灵敏度要求可能是不一样的，如：

1. 对于有流量突发的关键业务，在需要的时候应该快速扩容（即便可能不需要，以防万一），但缩容要慢（防止另一个流量高峰）。
2. 处理关键数据的应用，数据量飙升时它们应该尽快扩容以减少数据处理时间，数据量降低时应尽快缩小规模以降低成本，数据量的短暂抖动导致不必要的频繁扩缩是可以接受的。
3. 处理常规数据/网络流量的业务，不是很重要，它们可能会以一般的方式扩大和缩小规模，以减少抖动。

HPA 在 K8S 1.18迎来了一次更新，在之前 v2beta2版本上新增了扩缩容灵敏度的控制，不过版本号依然保持 v2beta2不变。

原理与误区

HPA 在进行扩缩容时，先是由固定的算法计算出期望副本数：
期望副本数 = ceil[当前副本数 * (当前指标 / 期望指标)]
其中“当前指标 / 期望指标”的比例如果接近1 (在容忍度范围内，默认为0.1，即比例在0.9~1.1之间)，则不进行伸缩，避免抖动导致频繁扩缩容。

> **说明**
> 容忍度是由 `kube-controller-manager` 参数 `--horizontal-pod-autoscaler-tolerance` 决定，默认是0.1，即10%。

本文要介绍的扩缩容速率调节，不是指要调整期望副本数的算法，它并不会加大或缩小扩缩容比例或数量，仅是控制扩缩容的速率，实现的效果是：控制 HPA 在自定义时间内最大允许扩容/缩容自定义比例/数量的 Pod。

如何使用

本次更新在 HPA Spec 下新增了一个 `behavior` 字段，下面有 `scaleUp` 和 `scaleDown` 两个字段分别控制扩容和缩容的行为，详情见 官方 API 文档。

使用示例

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: web
spec:
  minReplicas: 1
  maxReplicas: 1000
  metrics:
  - pods:
      metric:
        name: k8s_pod_rate_cpu_core_used_limit
      target:
        averageValue: "80"
        type: AverageValue
    type: Pods
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web
  behavior: # 这里是重点
    scaleDown:
      stabilizationWindowSeconds: 300 # 需要缩容时，先观察 5 分钟，如果一直持续需要缩容才执行缩容
```

```
      policies:
      - type: Percent
        value: 100 # 允许全部缩掉
        periodSeconds: 15
    scaleUp:
      stabilizationWindowSeconds: 0 # 需要扩容时，立即扩容
      policies:
      - type: Percent
        value: 100
        periodSeconds: 15 # 每 15s 最大允许扩容当前 1 倍数量的 Pod
      - type: Pods
        value: 4
        periodSeconds: 15 # 每 15s 最大允许扩容 4 个 Pod
      selectPolicy: Max # 使用以上两种扩容策略中算出来扩容 Pod 数量最大的
```

使用说明

- 以上 `behavior` 配置是默认的，即如果不配置，会默认加上。
- `scaleUp` 和 `scaleDown` 都可以配置1个或多个策略，最终扩缩时用哪个策略，取决于 `selectPolicy` 。
- `selectPolicy` 默认是 `Max` ，即扩缩时，评估多个策略算出来的结果，最终选取扩缩 Pod 数量最多的那个策略的结果。
- `stabilizationWindowSeconds` 是稳定窗口时长，即需要指标高于或低于阈值，并持续这个窗口的时长才会真正执行扩缩，以防止抖动导致频繁扩缩容。扩容时，稳定窗口默认为0，即立即扩容；缩容时，稳定窗口默认为5分钟。
- `policies` 中定义扩容或缩容策略， `type` 的值可以是 `Pods` 或 `Percent` ，表示每 `periodSeconds` 时间范围内，允许扩缩容的最大副本数或比例。

场景与示例

快速扩容

当您的应用需要快速扩容时，可以使用类似如下的 HPA 配置：

```
behavior:
  scaleUp:
    policies:
    - type: Percent
      value: 900
      periodSeconds: 15 # 每 15s 最多允许扩容 9 倍于当前副本数
```

上面的配置表示扩容时立即新增当前9倍数量的副本数，当然也不能超过 `maxReplicas` 的限制。
假如一开始只有1个Pod，如果遭遇流量突发，且指标持续超阈值9倍以上，它将以飞快的速度进行扩容，扩容时 Pod 数量变化趋势如下：

```
1 -> 10 -> 100 -> 1000
```

没有配置缩容策略，将等待全局默认的缩容时间窗口（默认5分钟）后开始缩容。

快速扩容，缓慢缩容

如果流量高峰过了，并发量骤降，如果用默认的缩容策略，等几分钟后 Pod 数量也会随之骤降，如果 Pod 缩容后突然又来一个流量高峰，虽然可以快速扩容，但扩容的过程毕竟还是需要一定时间的，如果流量高峰足够高，在这段时间内还是可能造成后端处理能力跟不上，导致部分请求失败。这时候我们可以为 HPA 加上缩容策略，HPA `behavior` 配置示例如下：

```
behavior:
  scaleUp:
    policies:
    - type: Percent
      value: 900
      periodSeconds: 15 # 每 15s 最多允许扩容 9 倍于当前副本数
  scaleDown:
```

```
    policies:
    - type: Pods
      value: 1
      periodSeconds: 600 # 每 10 分钟最多只允许缩掉 1 个 Pod
```

上面示例中增加了 `scaleDown` 的配置，指定缩容时每10分钟才缩掉1个 Pod，大大降低了缩容速度，缩容时的 Pod 数量变化趋势如下：

```
1000 -> … (10 min later) -> 999
```

这个可以让关键业务在可能有流量突发的情况下保持处理能力，避免流量高峰导致部分请求失败。

缓慢扩容

如果想要您的应用不太关键，希望扩容时不要太敏感，可以让它扩容平稳缓慢一点，为 HPA 加入下面的 `behavior`：

```
behavior:
  scaleUp:
    policies:
    - type: Pods
      value: 1
      periodSeconds: 300 # 每 5 分钟最多只允许扩容 1 个 Pod
```

假如一开始只有1个 Pod，指标一直持续超阈值，扩容时它的 Pod 数量变化趋势如下：

```
1 -> 2 -> 3 -> 4
```

禁止自动缩容

如果应用非常关键，希望扩容后不自动缩容，需要人工干预或其它自己开发的 controller 来判断缩容条件，可以使用类型如下的 `behavior` 配置来禁止自动缩容：

```
behavior:
  scaleDown:
    selectPolicy: Disabled
```

延长缩容时间窗口

缩容默认时间窗口是5分钟，如果我们需要延长时间窗口以避免一些流量毛刺造成的异常，可以指定下缩容的时间窗口，`behavior` 配置示例如下：

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 600 # 等待 10 分钟再开始缩容
    policies:
    - type: Pods
      value: 5
      periodSeconds: 600 # 每 10 分钟最多只允许缩掉 5 个 Pod
```

上面的示例表示当负载降下来时，会等待600s (10分钟) 再缩容，每10分钟最多只允许缩掉5个 Pod。

延长扩容时间窗口

有些应用经常会有数据毛刺导致频繁扩容，而扩容出来的 Pod 可能会浪费资源。例如数据处理管道的场景，需要的副本数取决于队列中的事件数量，当队列中堆积了大量事件时，我们希望可以快速扩容，但又不希望太灵敏，因为可能只是短时间内的事件堆积，即使不扩容也可以很快处理掉。
默认的扩容算法会在较短的时间内扩容，针对这种场景我们可以给扩容增加一个时间窗口以避免毛刺导致扩容带来的资源浪费，`behavior` 配置示例如下：

```
behavior:
  scaleUp:
```

```
    stabilizationWindowSeconds: 300 # 扩容前等待 5 分钟的时间窗口
    policies:
    - type: Pods
      value: 20
      periodSeconds: 60 # 每分钟最多只允许扩容 20 个 Pod
```

上面的示例表示扩容时，需要先等待5分钟的时间窗口，如果在这段时间内指标又降下来了就不再扩容，如果一直持续超过阈值才扩容，并且每分钟最多只允许扩容20个 Pod。

常见问题

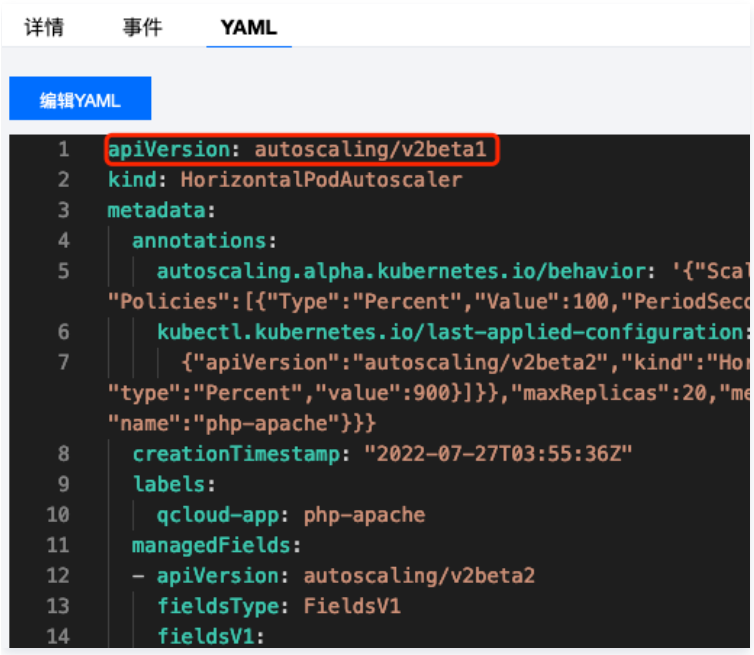
为什么用 v2beta2创建的 HPA，创建后获取到的 yaml 版本是 v1或 v2beta1?

```
❯ kubectl get hpa php-apache -o yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  annotations:
    autoscaling.alpha.kubernetes.io/behavior: '{"
    autoscaling.alpha.kubernetes.io/conditions: '
      HPA controller was able to get the target''
      HPA was unable to compute the replica count
      unable to get metrics for resource cpu: no
      API"},{"type":"ScalingLimited","status":"Tr
      desired replica count is less than the mini
    autoscaling.alpha.kubernetes.io/current-metri
    kubectl.kubernetes.io/last-applied-configurat
      {"apiVersion":"autoscaling/v2beta2","kind":
"name":"cpu","target":{"averageUtilization":40,"t
  creationTimestamp: "2022-07-27T03:55:36Z"
  labels:
    qcloud-app: php-apache
  name: php-apache
  namespace: test
  resourceVersion: "2437754900"
  selfLink: /apis/autoscaling/v1/namespaces/test/
  uid:
spec:
  maxReplicas: 20
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 40
status:
  currentCPUUtilizationPercentage: 0
  currentReplicas: 1
  desiredReplicas: 1
  lastScaleTime: "2022-07-27T08:44:10Z"
```

这是因为 HPA 有多个 apiVersion 版本:

```
kubectl api-versions | grep autoscaling
autoscaling/v1
autoscaling/v2beta1
autoscaling/v2beta2
```

以任意一种版本创建，都可以以任意版本获取（自动转换）。
如果是用 kubectl 获取，kubectl 在进行 API discovery 时，会缓存 apiserver 返回的各种资源与版本信息，有些资源存在多个版本，在 get 时如果不指定版本，会使用默认版本获取，对于 HPA，默认是 v1。如果是通过一些平台的界面获取，取决于平台的实现方式，若用腾讯云容器服务控制台，默认用v2beta1版本展示:



如何使用 v2beta2版本获取或编辑?

指定包含版本信息的完整资源名即可:

```
kubectl get horizontalpodautoscaler.v2beta2.autoscaling php-apache -o yaml
# kubectl edit horizontalpodautoscaler.v2beta2.autoscaling php-apache
```

配置快速扩容，为什么快不起来?

如以下配置:

```
behavior:
  scaleUp:
    policies:
    - type: Percent
      value: 900
      periodSeconds: 10
```

含义是允许每10秒最大允许扩出9倍于当前数量的 Pod，实测中可能发现压力已经很大了，但扩容却并不快。
通常原因是计算周期与指标延时:

- 期望副本数的计算有个计算周期，默认是15秒（由 `kube-controller-manager` 的 `--horizontal-pod-autoscaler-sync-period` 参数决定）。
- 每次计算时，都会通过相应的 metrics API 去获取当前监控指标的值，这个返回的值通常不是实时的，对于腾讯云容器服务而言，监控数据是每分钟上报一次；对于自建的 prometheus + prometheus-adapter 而言，监控数据的更新取决于监控数据抓取间隔，prometheus-adapter 的 `--metrics-relist-interval` 参数决定监控指标刷新周期（从 prometheus 中查询），这两部分时长之和为监控数据更新的最长时间。

通常都不需要 HPA 极度的灵敏，有一定的延时一般都是可以接受的。如果实在有对灵敏度特别敏感的场景，可以考虑使用 prometheus，缩小监控指标抓取间隔和 prometheus-adapter 的 `--metrics-relist-interval` 。

小结

本文介绍了如何利用 HPA 的新特性来控制扩缩容的速率，以更好的满足各种不同场景对扩容速度的需求，也提供了常见的几种场景与配置示例，可自行根据自己需求对号入座。

参考资料

- HPA 官方介绍文档
- 控制 HPA 扩容速度的提案

容器化
境外镜像拉取加速

最近更新时间：2023-07-24 14:12:11

操作场景

目前大多数开源应用的容器镜像（例如 Kubernetes、TensorFlow 等），都托管在境外镜像托管平台（例如 DockerHub、`quay.io` 等），在国内拉取镜像时可能存在网络问题导致拉取速度慢、甚至无法成功拉取等问题。常见解决方法为手动将镜像 Pull 到本地，再 Push 到自主搭建的镜像仓库进行手动同步，过程极其繁琐且无法覆盖全部仓库及最新镜像版本。腾讯云 容器镜像服务 TCR 企业版提供主流境外镜像托管平台加速服务，可以有效解决境外镜像拉取难导致开源应用无法顺利部署的问题。本文将介绍 TKE 集群如何通过 TCR 加速服务实现境外镜像拉取加速。

限制条件

- 加速服务目前仅面向容器服务 TKE、容器镜像服务 TCR 用户。
- 加速服务目前只支持腾讯云 私有网络 VPC 访问，公网访问能力暂未开放，相关域名可以访问但无法提供实际的加速功能。

操作步骤

对于 TKE 集群，DockerHub 平台（docker.io）内公开镜像已默认配置加速，如需加速其他平台内镜像仓库，例如 `quay.io`，则需要进行相关配置。集群运行时为 Docker 或 Containerd，配置方法有所不同：

集群运行时为 Docker 的配置

对于运行时为 Docker 的节点，由于 Docker 本身不支持 `docker.io` 以外的加速配置，使用 `docker.io` 之外的境外容器镜像时，需要执行以下命令更改镜像地址的域名，将 `quay.io` 替换为 `quay.tencentcloudcr.com`。示例如下：

```
docker pull quay.tencentcloudcr.com/k8scsi/csi-resizer:v0.5.0
```

集群运行时为 Containerd 的配置

对于运行时为 Containerd 的节点，由于 Containerd 本身支持任意镜像仓库的加速地址配置，可以通过修改 Containerd 配置，实现不更改镜像地址就可自动加速拉取镜像，适用于大量境外镜像的场景，可减少大量地址修改的繁琐步骤。

1. TKE 添加节点或者使用节点池，可以将节点写入自定义脚本，通过脚本可以统一修改增量节点的 Containerd 配置、添加境外镜像的加速地址。脚本示例如下：

```
sed -i '/\[plugins\.cri\.registry\.mirrors\]/ a\\ \ \ \ \ \ \
[plugins.cri.registry.mirrors."quay.io"]\n\ \ \ \ \ \ \ \ \ endpoint =
["https://quay.tencentcloudcr.com"]' /etc/containerd/config.toml
```

或者手动修改存量节点的 Containerd 配置（`/etc/containerd/config.toml`），添加类似如下配置：

```
[plugins.cri.registry]
[plugins.cri.registry.mirrors]
[plugins.cri.registry.mirrors."quay.io"]
endpoint = ["https://quay.tencentcloudcr.com"]
[plugins.cri.registry.mirrors."docker.io"]
endpoint = ["https://mirror.ccs.tencentyun.com"]
```

> **说明**
> 您还可以使用 Ansible 批量修改存量节点的 Containerd 配置，详情请参见 使用 Ansible 批量操作 TKE 节点。

2. 执行以下命令重启 Containerd。示例如下：

```
systemctl restart containerd
```

3. 执行以下命令，使用原始镜像地址拉取镜像。示例如下：

```
crictl pull quay.io/k8scsi/csi-resizer:v0.5.0
```

镜像分层最佳实践

最近更新时间：2024-08-14 16:08:11

操作场景

本文介绍如何把业务镜像分层构建与管理，使用 TCR 高效的管理各类容器镜像的最佳实践。

容器镜像分层的优势

- 共享资源，提升资源利用率。
- 镜像管理规范化与标准化，便于 Devops 落地实施。
- TCR 的免运维、镜像加速，可轻松提升大规模镜像分发速度 5-10 倍。
- TCR 企业版内实例、命名空间、镜像仓库等资源的读写操作已接入操作审计，通过控制台进入**审计日志**即可查看相关操作记录。

前提条件

在使用 TCR 内托管的私有镜像进行应用部署前，您需要完成以下准备工作：

- 已在 容器镜像服务 创建企业版实例。如尚未创建，请参考 创建企业版实例 完成创建。
- 如果使用子账号进行操作，请参考 企业版授权方案示例 提前为子账号授予对应实例的操作权限。

> **说明：**
> 已有容器镜像服务也适用，修改镜像仓库地址即可。

1. F3S Docker Files 介绍

项目由以下部分组成：

```
$ tree -L 3 ./f3s-docker-files
./f3s-docker-files
├── README.md                                    ------  说明文件
├── DockerBuildImages.sh                         ------  镜像构建脚本
├── 0.base                                       ------  0. 构建 Base 层各类系统镜像
│   ├── alpine                                   ------    构建 Base 层 alpine 系统镜像
│   │   └── Dockerfile
│   └── centos-7.8                               ------    构建 Base 层 Centos7.8 系统镜像
│       ├── Dockerfile
│       └── centos-7.8.2003-x86_64-docker.tar.xz
├── 1.ops                                        ------  1. 构建运维层各类镜像
│   └── Dockerfile-alpine                        ------    构建运维层 alpine 镜像
├── 2.lang                                       ------  2. 构建语言层各类镜像
│   └── Dockerfile-alpine-kona                   ------    语言层 alpine-kona 镜像
└── 3.app                                        ------  3. 构建应用层各类镜像
    ├── jmeter
    │   ├── Dockerfile-jmeter-base               ------    构建 jmeter-base 镜像
    │   ├── Dockerfile-jmeter-grafana-reporter   ------    构建 jmeter-grafana-reporter 镜像
    │   ├── Dockerfile-jmeter-master             ------    构建 jmeter-master 镜像
    │   └── Dockerfile-jmeter-slave              ------    构建 jmeter-slave 镜像
    ├── nginx
    │   ├── Dockerfile-alpine-nginx              ------    构建 alpine-nginx 镜像
    │   ├── default.conf
    │   └── nginx.conf
    └── skywalking
        └── Dockerfile-alpine-kona-skywalking    ------    构建 alpine-kona-skywalking 镜像
```

- **alpine/Dockerfile**：使用 alpine 官方提供的 3.13 docker 镜像 构建，添加常用运维工具与中文支持等配置。
- **centos-7.8/Dockerfile**：使用 Centos 官方提供的 7.8 docker 镜像 构建，添加常用运维工具与中文支持等配置。

- **Dockerfile-alpine-kona**：使用 Dockerfile-alpine 和 TencentKona 8.0.5 二进制包 构建，为控制镜像大小对 Kona 做了部分裁剪。
- **Dockerfile-jmeter-base**：基于 Jmeter 官方提供的 5.4.1 二进制包 构建。
- **Dockerfile-jmeter-grafana-reporter**：基于 Grafana-Reporter 构建，提供 Grafana 仪表板生成 Jmeter PDF 报告的功能。
- **Dockerfile-jmeter-master**：基于 Jmeter-base 镜像构建，实现 Jmeter 分布式压测 Master 的功能。
- **Dockerfile-jmeter-slave**：基于 Jmeter-base 镜像构建，实现 Jmeter 分布式压测 Slave 的功能。
- **Dockerfile-alpine-nginx**：基于 Dockerfile-alpine 构建，添加 nginx 配置初始化与日志规范等配置。
- **Dockerfile-alpine-kona-skywalking**：使用 Dockerfile-alpine-kona 和 skywalking 官方提供的 8.5 二进制包 构建。

2. 项目资源说明

2.0 Dockerfile-alpine

==========ALPINE DOCKER FILE==========

```
# build
FROM alpine:3.13

ENV FROM alpine:3.13

# Alpine 镜像中并没有包含 tzdata，所以无法直接通过环境变量 TZ 设置时区，因此需要安装 tzdata：
ENV TZ=Asia/Shanghai

RUN echo 'http://mirrors.tencent.com/alpine/v3.13/main/' > /etc/apk/repositories \
    && echo 'http://mirrors.tencent.com/alpine/v3.13/community/' >> /etc/apk/repositories \
    && apk --no-cache add apache2-utils \
                          bind-tools \
                          bridge-utils \
                          busybox-extras \
                          curl \
                          ebtables \
                          ethtool \
                          fio \
                          fping \
                          iperf3 \
                          iproute2 \
                          iptables \
                          iputils \
                          ipvsadm \
                          jq \
                          lftp \
                          lsof \
                          mtr \
                          netcat-openbsd \
                          net-tools \
                          nmap \
                          procps \
                          psmisc \
                          rsync \
                          smartmontools \
                          strace \
                          sysstat \
                          tcpdump \
                          tree \
                          tzdata \
                          unzip \
                          util-linux \
                          wget \
                          zip  \
    && echo "${TZ}" > /etc/timezone \
```

```
    && ln -sf /usr/share/zoneinfo/${TZ} /etc/localtime \
    && rm -rf /var/cache/apk/*

ENV BUILD f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:v3.13

# docker build -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:v3.13 .
```

==========Build, tag and push the base image==========

```
cd $pwd/0.base/alpine
docker build -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:v3.13   -f Dockerfile .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:v3.13
```

2.1 Dockerfile-CentOS-7.8

==========Centos-7.8 DOCKER FILE==========

```
# build
# centos 7.8 官方 Dockerfile: https://github.com/CentOS/sig-cloud-instance-images/blob/CentOS-7.8.2003-x86_64/docker/Dockerfile
# centos 7.8 官方包:  wget https://raw.githubusercontent.com/CentOS/sig-cloud-instance-images/CentOS-7.8.2003-x86_64/docker/centos-7.8.2003-x86_64-docker.tar.xz

FROM scratch

ADD centos-7.8.2003-x86_64-docker.tar.xz /

LABEL name="CentOS Base Image" \
      vendor="CentOS" \
      license="GPLv2" \
      build-date="20200504"

# 增加一些小工具，并修改时区
RUN set -ex \
    && yum install -y wget \
    && rm -rf /etc/yum.repos.d/CentOS-* \
# 添加 Tencent yum 源
    && wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.cloud.tencent.com/repo/centos7_base.repo \
    && yum fs filter documentation \
    && yum install -y atop \
                      bind-utils \
                      curl \
                      dstat \
                      ebtables \
                      ethtool \
                      fping \
                      htop \
                      iftop \
                      iproute \
                      jq \
                      less \
                      lsof \
                      mtr \
                      nc \
                      net-tools \
                      nmap-ncat \
                      perf \
```

```
                    psmisc \
                    strace \
                    sysstat \
                    tcpdump \
                    telnet \
                    tree \
                    unzip \
                    wget \
                    which \
                    zip \
                    ca-certificates \
    && rm -rf /etc/localtime \
    && ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime \
# Install dumb-init
    && wget -O /usr/local/bin/dumb-init https://github.com/Yelp/dumb-init/releases/download/v1.2.5/dumb-
init_1.2.5_x86_64 \
    && chmod +x /usr/local/bin/dumb-init \
# Install gosu grab gosu for easy step-down from root
# https://github.com/tianon/gosu/releases
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/1.13/gosu-amd64" \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
# 安装中文语言包，解决中文乱码问题，vi 乱码需要这里解决
    && yum -y install kde-l10n-Chinese glibc-common \
    && localedef -c -f UTF-8 -i zh_CN zh_CN.utf8 \
    && export LC_ALL=zh_CN.utf8 \
    && yum clean all  \
    && rm -rf /tmp/* \
    && rm -rf /var/lib/yum/* \
    && rm -rf /var/cache/yum

# 解决 less 乱码问题
ENV LESSCHARSET utf-8

# 设置语言环境变量
ENV LANG=en_US.UTF-8

# 不加这句则 kubernetes 中的 stdin: true 和 tty: true 不生效
CMD ["/bin/bash"]

ENV BUILD f3s-docker-file.tencentcloudcr.com/f3s-tcr/centos:v7.8

# docker build -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/centos:v7.8 .
```

==========Build, tag and push the base image==========

```
cd $pwd/0.base/centos-7.8
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/centos:v7.8  -f Dockerfile .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/centos:v7.8
# To test run:  docker run --name test -it --rm f3s-docker-file.tencentcloudcr.com/f3s-tcr/centos:v7.8
uname -a
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.com/f3s-tcr/centos:v7.8
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencentcloudcr.com/f3s-
tcr/centos:v7.8 sh
```

2.2 Dockerfile-Ops

==========Ops DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:v3.13
MAINTAINER westzhao

ENV LANG=C.UTF-8

# 下载运维工具
RUN apk --no-progress --purge --no-cache add --upgrade wget \
    curl \
    mysql-client \
    busybox \
    busybox-extras \
    bash \
    bash-doc \
    bash-completion \
    tzdata \
    vim \
    unzip && \
# 下载 glibc 支持 jdk、解决中文支持问题 && \
wget -q -O /etc/apk/keys/sgerrand.rsa.pub https://alpine-pkgs.sgerrand.com/sgerrand.rsa.pub && \
    wget https://github.com/sgerrand/alpine-pkg-glibc/releases/download/2.33-r0/glibc-2.33-r0.apk && \
    wget https://github.com/sgerrand/alpine-pkg-glibc/releases/download/2.33-r0/glibc-bin-2.33-r0.apk &&
\
    wget https://github.com/sgerrand/alpine-pkg-glibc/releases/download/2.33-r0/glibc-i18n-2.33-r0.apk &&
\
    apk add glibc-2.33-r0.apk glibc-bin-2.33-r0.apk glibc-i18n-2.33-r0.apk && \
    rm glibc-2.33-r0.apk glibc-bin-2.33-r0.apk glibc-i18n-2.33-r0.apk && \
    /usr/glibc-compat/bin/localedef -i en_US -f UTF-8 C.UTF-8  && \
    echo "export LANG=$LANG" > /etc/profile.d/locale.sh && \
# 修改时区
    mkdir -p /share/zoneinfo/Asia/ && \
    mkdir -p /etc/zoneinfo/Asia/ && \
    cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime  && \
    cp /usr/share/zoneinfo/Asia/Shanghai /share/zoneinfo/Asia/Shanghai && \
    cp /usr/share/zoneinfo/Asia/Shanghai /etc/zoneinfo/Asia/Shanghai && \
    echo "Asia/Shanghai" > /etc/timezone  && \
    apk del tzdata && \
# 删除 apk 缓存 && \
    rm -rf /var/cache/apk/*
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:latest  -f
./1.ops/Dockerfile-alpine .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:latest
# To test run:  docker run --name test -it --rm f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:latest
sh $(java -version)
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencentcloudcr.com/f3s-
tcr/alpine:latest sh
```

2.3 Dockerfile-alpine-kona

==========Alpine Kona DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:latest
```

```
MAINTAINER westzhao

ENV LANG=C.UTF-8

# 通过 wget 下载 Kona 安装包 && \
RUN cd  /opt && \
wget https://github.com/Tencent/TencentKona-8/releases/download/8.0.5-GA/TencentKona8.0.5.b12_jdk_linux-
x86_64_8u282.tar.gz && \
    tar -xvf TencentKona8.0.5.b12_jdk_linux-x86_64_8u282.tar.gz && \
    rm TencentKona8.0.5.b12_jdk_linux-x86_64_8u282.tar.gz && \
    ln -nfs /opt/TencentKona-8.0.5-282 /opt/jdk && \
# 裁剪 jdk 未使用资源 && \
rm /opt/jdk/release && \
    rm /opt/jdk/THIRD_PARTY_README && \
    rm /opt/jdk/LICENSE && \
    rm /opt/jdk/ASSEMBLY_EXCEPTION && \
    rm -rf /opt/jdk/sample/ && \
    rm -rf /opt/jdk/demo/ && \
    rm -rf /opt/jdk/src.zip && \
    rm -rf /opt/jdk/man/ && \
    rm -rf /opt/jdk/lib/missioncontrol  && \
    rm -rf /opt/jdk/lib/visualvm  && \
    rm -rf /opt/jdk/lib/ant-javafx.jar  && \
    rm -rf /opt/jdk/lib/javafx-mx.jar  && \
    rm -rf /opt/jdk/lib/jconsole.jar  && \
    rm -rf /opt/jdk/jre/lib/amd64/libawt_xawt.so  && \
    rm -rf /opt/jdk/jre/lib/amd64/libjavafx_font_freetype.so  && \
    rm -rf /opt/jdk/jre/lib/amd64/libjavafx_font_pango.so  && \
    rm -rf /opt/jdk/jre/lib/amd64/libjavafx_font.so  && \
    rm -rf /opt/jdk/jre/lib/amd64/libjavafx_font_t2k.so  && \
    rm -rf /opt/jdk/jre/lib/amd64/libjavafx_iio.so  && \
    rm -rf /opt/jdk/jre/lib/amd64/libjfxwebkit.so  && \
    rm -rf /opt/jdk/jre/lib/desktop  && \
    rm -rf /opt/jdk/jre/lib/ext/jfxrt.jar  && \
    rm -rf /opt/jdk/jre/lib/fonts  && \
    rm -rf /opt/jdk/jre/lib/locale/de  && \
    rm -rf /opt/jdk/jre/lib/locale/fr  && \
    rm -rf /opt/jdk/jre/lib/locale/it  && \
    rm -rf /opt/jdk/jre/lib/locale/ja  && \
    rm -rf /opt/jdk/jre/lib/locale/ko  && \
    rm -rf /opt/jdk/jre/lib/locale/ko.UTF-8  && \
    rm -rf /opt/jdk/jre/lib/locale/pt_BR  && \
    rm -rf /opt/jdk/jre/lib/locale/sv  && \
    rm -rf /opt/jdk/jre/lib/locale/zh_HK.BIG5HK  && \
    rm -rf /opt/jdk/jre/lib/locale/zh_TW  && \
    rm -rf /opt/jdk/jre/lib/locale/zh_TW.BIG5  && \
    rm -rf /opt/jdk/jre/lib/oblique-fonts  && \
    rm -rf /opt/jdk/jre/lib/deploy.jar  && \
    rm -rf /opt/jdk/jre/lib/locale/

# JAVA_HOME
ENV JAVA_HOME=/opt/jdk
ENV CLASSPATH=.:$JAVA_HOME/lib/
ENV PATH=$JAVA_HOME/bin:$PATH
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest  -f
./2.lang/Dockerfile-alpine-kona .
```

```
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest
# To test run:  docker run --name test -it --rm f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest sh $(java -version)
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest sh
```

2.4 Dockerfile-alpine-kona-skywalking

==========Alpine Kona SkyWalking DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest
MAINTAINER westzhao

ENV LANG=C.UTF-8

# 下载运维工具
RUN mkdir /3.app && \
    wget -q -O /3.app/apache-skywalking-apm-8.5.0.tar.gz https://archive.apache.org/dist/skywalking/8.5.0/apache-skywalking-apm-8.5.0.tar.gz && \
    tar zxf /3.app/apache-skywalking-apm-8.5.0.tar.gz -C /3.app && \
    mv /3.app/apache-skywalking-apm-bin/agent /3.app/skywalking && \
    rm -rf /3.app/apache-skywalking-apm-8.5.0.tar.gz && \
    rm -rf /3.app/apache-skywalking-apm-bin/

# JAVA_HOME
ENV JAVA_HOME=/opt/jdk
ENV CLASSPATH=.:$JAVA_HOME/lib/
ENV PATH=$JAVA_HOME/bin:$PATH
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona-skywalking:latest  -f ./3.app/skywalking/Dockerfile-alpine-kona-skywalking .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona-skywalking:latest
# To test run:  docker run --name test -it --rm f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona-skywalking:latest sh $(java -version)
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona-skywalking:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona-skywalking:latest sh
```

2.5 Dockerfile-jmeter-base

==========JMETER BASE DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest
MAINTAINER westzhao

ARG JMETER_VERSION=5.4.1

# 下载 jmeter
RUN mkdir /jmeter && \
cd /jmeter && \
wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-$JMETER_VERSION.tgz && \
    tar -xzf apache-jmeter-$JMETER_VERSION.tgz && \
```

```
    rm apache-jmeter-$JMETER_VERSION.tgz && \
# 下载 JMeterPlugins-Standard && \
cd /jmeter/apache-jmeter-$JMETER_VERSION/ && \
    wget -q -O /tmp/JMeterPlugins-Standard-1.4.0.zip https://jmeter-
plugins.org/downloads/file/JMeterPlugins-Standard-1.4.0.zip && \
    unzip -n /tmp/JMeterPlugins-Standard-1.4.0.zip && \
    rm /tmp/JMeterPlugins-Standard-1.4.0.zip && \
# 下载 pepper-box && \
    wget -q -O /jmeter/apache-jmeter-$JMETER_VERSION/lib/ext/pepper-box-1.0.jar
https://github.com/raladev/load/blob/master/JARs/pepper-box-1.0.jar?raw=true && \
# 下载 bzm-parallel && \
cd /jmeter/apache-jmeter-$JMETER_VERSION/ && \
    wget -q -O /tmp/bzm-parallel-0.7.zip https://jmeter-plugins.org/files/packages/bzm-parallel-0.7.zip
&& \
    unzip -n /tmp/bzm-parallel-0.7.zip && \
    rm /tmp/bzm-parallel-0.7.zip

ENV JMETER_HOME /jmeter/apache-jmeter-$JMETER_VERSION/

ENV PATH $JMETER_HOME/bin:$PATH
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-base:latest -f
./3.app/jmeter/Dockerfile-jmeter-base .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-base:latest
```

2.6 Dockerfile-jmeter-master

==========JMETER-MASTER DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-base:latest
MAINTAINER westzhao

EXPOSE 60000
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-master:latest -f
./3.app/jmeter/Dockerfile-jmeter-master .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-master:latest
```

2.7 Dockerfile-jmeter-slave

==========JMETER-SLAVES DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-base:latest
MAINTAINER westzhao

EXPOSE 1099 50000

ENTRYPOINT $JMETER_HOME/bin/jmeter-server \
-Dserver.rmi.localport=50000 \
-Dserver_port=1099 \
-Jserver.rmi.ssl.disable=true
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-slave:latest -f
./3.app/jmeter/Dockerfile-jmeter-slave .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-slave:latest
```

2.8 Dockerfile-jmeter-grafana-reporter

==========JMETER-GRAFANA-REPORTER DOCKER FILE==========

```
# build
# 多阶构建
FROM golang:1.14.7-alpine3.12 AS build
MAINTAINER westzhao
# 下载运维/编译工具
WORKDIR /go/src/${owner:-github.com/8710925}/reporter
# ADD . .
# RUN go install -v github.com/8710925/reporter/cmd/grafana-reporter

RUN apk --no-progress --purge --no-cache add --upgrade git && \
# 编译 grafana-reporter
    git clone https://${owner:-github.com/8710925}/reporter . \
    && go install -v github.com/8710925/reporter/cmd/grafana-reporter

# create grafana reporter image
FROM alpine:3.12
COPY --from=build /go/src/${owner:-github.com/8710925}/reporter/util/texlive.profile /
COPY --from=build /go/src/${owner:-github.com/8710925}/reporter/util/SIMKAI.ttf /usr/share/fonts/west/

RUN apk --no-progress --purge --no-cache add --upgrade  wget \
    curl \
    fontconfig \
    unzip \
    tzdata \
    perl-switch && \
    wget -qO- \
    "https://raw.githubusercontent.com/rstudio/tinytex/main/tools/install-unx.sh" | \
    sh -s - --admin --no-path \
    && mv ~/.TinyTeX /opt/TinyTeX \
    && /opt/TinyTeX/bin/*/tlmgr path add \
    && tlmgr path add \
    && chown -R root:adm /opt/TinyTeX \
    && chmod -R g+w /opt/TinyTeX \
    && chmod -R g+wx /opt/TinyTeX/bin \
    && tlmgr update --self --repository http://mirrors.tuna.tsinghua.edu.cn/CTAN/systems/texlive/tlnet \
    && tlmgr install epstopdf-pkg ctex  everyshi everysel euenc \
# 修改时区
    && cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime  \
    && echo "Asia/Shanghai" > /etc/timezone  \
    && apk del tzdata \
    # Cleanup
    && fmtutil-sys  --all \
    && texhash \
    && mktexlsr \
    && apk del --purge -qq \
    && rm -rf /var/lib/apt/lists/*

COPY --from=build /go/bin/grafana-reporter /usr/local/bin
```

```
ENTRYPOINT [ "/usr/local/bin/grafana-reporter","-ip","jmeter-grafana:3000" ]
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-grafana-reporter:latest -f
./3.app/jmeter/Dockerfile-jmeter-grafana-reporter .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-grafana-reporter:latest
```

2.9 Dockerfile-alpine-nginx

==========Alpine Nginx DOCKER FILE==========

```
# build
FROM f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-kona:latest
MAINTAINER westzhao

ENV LANG=C.UTF-8

# 下载运维工具
RUN apk --no-progress --purge --no-cache add --upgrade nginx && \
# 删除 apk 缓存 && \
    rm -rf /var/cache/apk/*

COPY ./3.app/nginx/default.conf /etc/nginx/http.d/default.conf
COPY ./3.app/nginx/nginx.conf /etc/nginx/nginx.conf

EXPOSE 80 443

CMD ["/usr/sbin/nginx", "-g", "daemon off;", "-c", "/etc/nginx/nginx.conf"]
```

==========Build, tag and push the base image==========

```
docker build --no-cache -t f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-nginx:latest -f
./3.app/nginx/Dockerfile-alpine-nginx .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-nginx:latest
# To test run:  docker run --name test -it --rm -p 8888:80 f3s-docker-file.tencentcloudcr.com/f3s-
tcr/alpine-nginx:latest nginx -v
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine-
nginx:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencentcloudcr.com/f3s-
tcr/alpine-nginx:latest sh
```

成本管理
资源利用率提升工具大全

最近更新时间：2023-11-13 16:10:02

背景

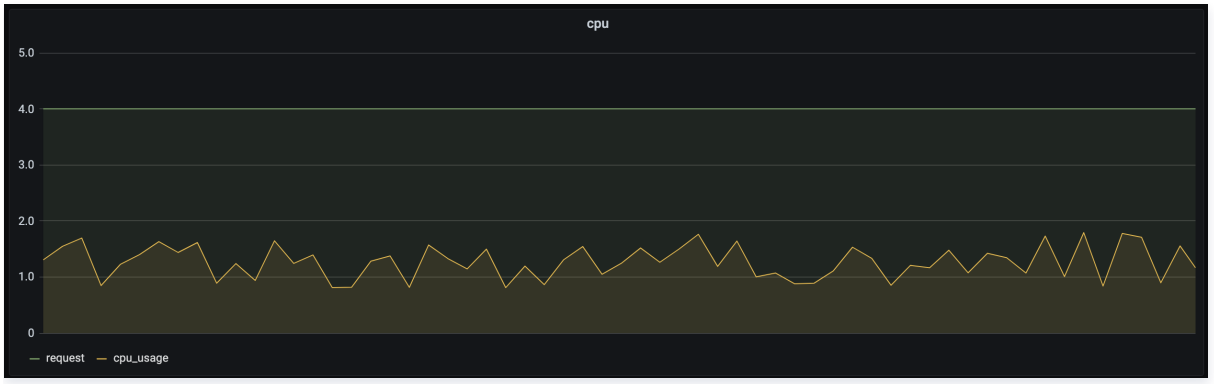
公有云的发展为业务的稳定性、可拓展性、便利性带来了极大帮助。这种用租代替买、并且提供完善的技术支持和保障的服务，理应为业务带来降本增效的效果。但实际上业务上云并不意味着成本一定较少，还需适配云上业务的应用开发、架构设计、管理运维、合理使用等多方面解决方案，才能真正助力业务的降本增效。在《Kubernetes 降本增效标准指南》系列的文章《容器化计算资源利用率现象剖析》中可看到，IDC 上云后资源利用率提高有限，即使已经容器化，节点的平均利用率依旧仅在13%左右，资源利用率的提升任重道远。
本篇文章将带您了解：
1. 为什么 Kubernetes 集群中的 **CPU 和内存资源利用率**通常都如此之低？
2. 现阶段在 TKE 上面有哪些产品化的方法可以轻松提升资源利用率？

资源浪费场景

为何资源利用率通常都如此之低？首先可以了解几个业务的实际使用资源场景：

场景1：资源预留普遍存在50%以上的浪费

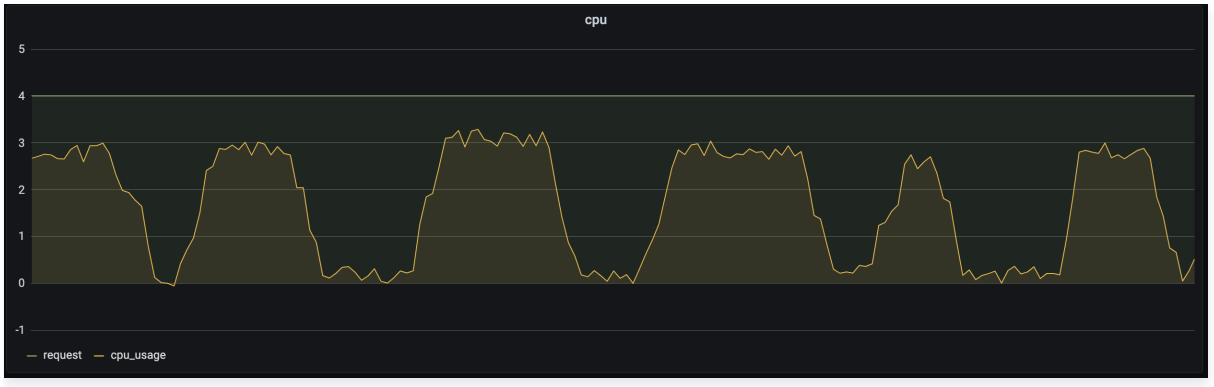
Kubernetes 中的 Request（请求）字段用于管理容器对 CPU 和内存资源预留的机制，保证容器至少可以达到的资源量，该部分资源不能被其他容器抢占，详情见 Kubernetes 官方文档 。当 Request 设置过小，无法保证业务的资源量，当业务的负载变高时无力承载，因此用户通常习惯将 Request 设置得很高，以保证服务的可靠性。但实际上，业务在大多数时段负载不会很高。以 CPU 为例，下图是某个实际业务场景下容器的资源预留（Request）和实际使用量（CPU_Usage）关系图：



从图中可以看出，资源预留远大于实际使用量，两者之间差值所对应的资源不能被其他负载使用，因此 Request 设置过大势必会造成较大的资源浪费。如何解决这样的问题？现阶段需要用户自己根据实际的负载情况设置更合理的 Request、以及限制业务对资源的无限请求，防止资源被某些业务过度占用。这里可以参考后文中的 Request Quota 和 Limit Ranges 的设置。此外，TKE 将推出 Request 推荐产品，帮助用户智能缩小 Request 和 Usage 之间的差值，在保障业务的稳定性的情况下有效提升资源利用率。

场景2：业务资源波峰波谷现象普遍，通常波谷时间大于波峰时间，资源浪费明显

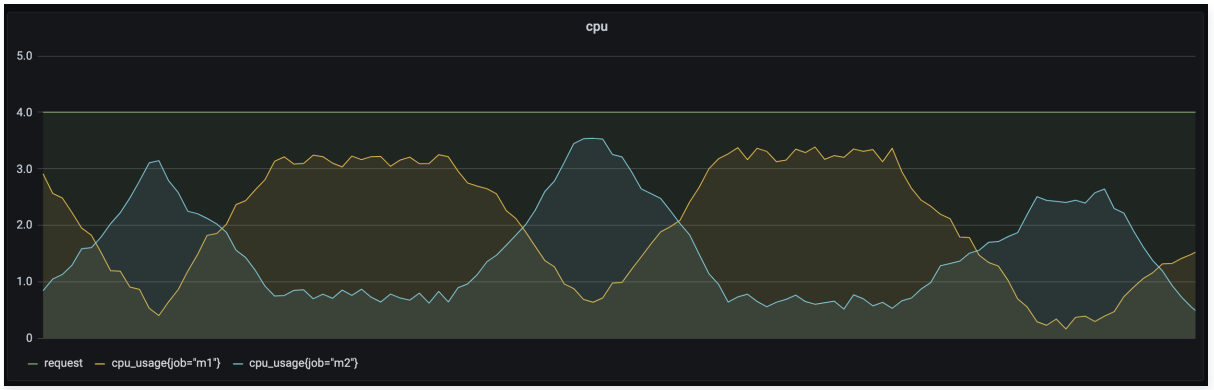
大多数业务存在波峰波谷，例如公交系统通常在白天负载增加，夜晚负载减少；游戏业务通常在周五晚上开始出现波峰，在周日晚开始出现波谷。如下图所示：



从图中可以看出，同一业务在不同的时间段对资源的请求量不同，如果用户设置的是固定的 Request，在负载较低时利用率很低。这时可以通过动态调整副本数以高资源利用率承载业务的波峰波谷，可以参考后文中的 HPA 、HPC、CA。

场景3：不同类型的业务，导致资源利用率有较大差异

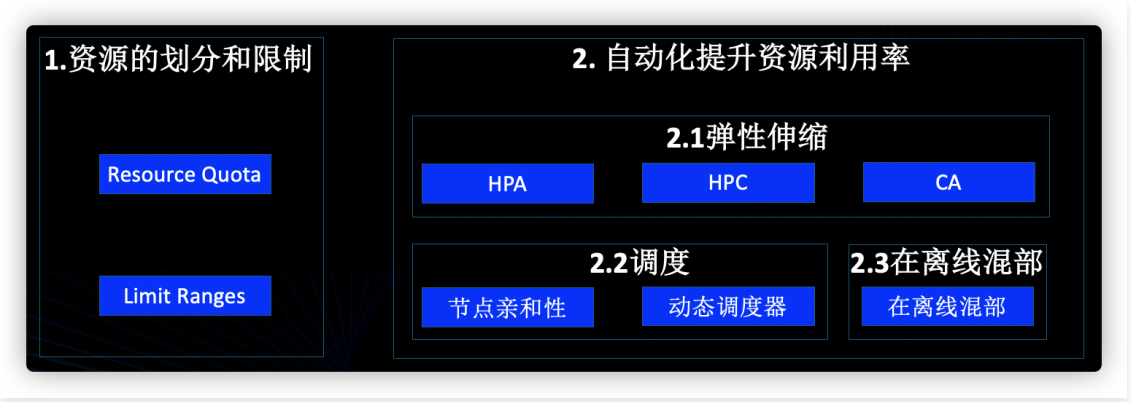
在线业务通常白天负载较高，对时延要求较高，必须优先调度和运行；而离线的计算型业务通常对运行时段和时延要求相对较低，理论上可以在在线业务波谷时运行。此外，有些业务属于计算密集型，对 CPU 资源消耗较多，而有些业务属于内存密集型，对内存消耗较多。



如上图所示，通过在离线混部可以动态调度离线业务和在线业务，让不同类型业务在不同的时间段运行以提升资源利用率。对于计算密集型业务和内存密集型业务，可以使用亲和性调度，为业务分配更合适的节点，有效提升资源利用率。具体方式可参考后文中的离在线混部和亲和性调度。

在 Kubernetes 上提升资源利用率

腾讯云容器服务 TKE 基于大量的用户实际业务，已经产品化了一系列工具，帮助用户轻松有效的提升资源利用率。主要从两方面着手：一是利用原生的 Kubernetes 能力手动进行资源的划分和限制；二是结合业务特性的自动化方案。



1. 资源划分和限制

设想，您是集群管理员，现在有4个业务部门使用同一个集群，您的责任是保证业务稳定性的前提下，让业务真正做到资源的按需使用。为了有效提升集群整体的资源利用率，这时就需要限制各业务使用资源的上限，以及通过一些默认值防止业务过量使用。
理想情况下，业务应该根据实际情况，设置合理的 Request 和 Limit（Request 用于对资源的占位，表示容器至少可以获得的资源；Limit 用于对资源的限制，表示容器至多可以获得的资源）。这样更利于容器的健康运行和资源的充分使用。但实际上用户经常忘记设置容器对资源的 Request 和 Limit。此外，对于共享使用一个集群的团队/项目来说，他们通常都将自己容器的 Request 和 Limit 设置得很高以保证自己服务的稳定性。当您使用容器服务控制台，创建负载时会给所有的容器设置如下默认值。该默认值是 TKE 根据真实业务分析预估得出，和具体的业务需求之间可能存在偏差。

资源	Request	Limit
CPU（核）	0.25	0.5
Memory(MiB)	256	1024

为了更细粒度的划分和管理资源，您可以在 TKE 上设置命名空间级别的 Resource Quota 以及 Limit Ranges。

使用 Resource Quota 划分资源

如果您管理的某个集群有4个业务，为了实现业务间的隔离和资源的限制，您可以使用命名空间和 Resource Quota。
Resource Quota 用于设置命名空间资源的使用配额，命名空间是 Kubernetes 集群里面的一个隔离分区，一个集群里面通常包含多个命名空间，例如 Kubernetes 用户通常会将不同的业务放在不同的命名空间里，您可以为不同的命名空间设置不同的 Resource Quota，以限制一个命名空间对集群整体资源的使用量，达到预分配和限制的效果。Resource Quota 主要作用于如下方面，详情见 Kubernetes 官方文档 。

1. 计算资源：所有容器对 CPU 和 内存的 Request 以及 Limit 的总和。
2. 存储资源：所有 PVC 的存储资源请求总和。
3. 对象数量：PVC/Service/Configmap/Deployment 等资源对象数量的总和。

Resource Quota 使用场景

- 给不同的项目/团队/业务分配不同的命名空间，通过设置每个命名空间资源的 Resource Quota 以达到资源分配的目的。
- 设置一个命名空间的资源使用数量的上限以提高集群的稳定性，防止一个命名空间对资源的多度侵占和消耗。

TKE 上的 Resource Quota

TKE 上已经实现对 Resource Quota 的产品化，您可以直接在控制台利用 Resource Quota 限制一个命名空间的资源使用量，操作详情可参见 Namespaces 文档 。

使用 Limit Ranges 限制资源

用户经常忘记设置资源的 Request 和 Limit，或者将值设置得很大怎么办？作为管理员，如果可以为不同的业务设置不同资源使用默认值以及范围，可以有效减少业务创建时的工作量同时，限制业务对资源的过度侵占。
与 Resource Quota 对命名空间整体的资源限制不同，Limit Ranges 适用于一个命名空间下的单个容器。可以防止用户在命名空间内创建对资源申请过小或过大容器，防止用户忘记设置容器的 Request 和 Limit。Limit Ranges 主要作用于如下方面，详情见 Kubernetes 官方文档 。

1. 计算资源：对所有容器设置 CPU 和内存使用量的范围。
2. 存储资源：对所有 PVC 能申请的存储空间的范围。
3. 比例设置：控制一种资源 Request 和 Limit 之间比例。
4. 默认值：对所有容器设置默认的 Request/Limit，如果容器未指定自己的内存请求和限制，将为它指定默认的内存请求和限制。

Limit Ranges 使用场景

1. 设置资源使用默认值，以防用户遗忘，也可以避免 QoS 驱逐重要的 Pod。
2. 不同的业务通常运行在不同的命名空间里，不同的业务通常资源使用情况不同，为不同的命名空间设置不同的 Request/Limit 可以提升资源利用率。
3. 限制容器对资源使用的上下限，保证容器正常运行的情况下，限制其请求过多资源。

TKE 上的 Limit Ranges

TKE 上已经实现对 Limit Ranges 的产品化，您可以直接在控制台管理命名空间的 Limit Ranges，操作详情可参见 Namespaces 文档 。Limit Ranges 具体可参考 Kubernetes 官方文档 。

2. 自动化提升资源利用率

上面提到的利用 Resource Quota 和 Limit Ranges 来分配和限制资源的方法依赖经验和手工，主要解决的是资源请求和分配不合理。如何更自动化的动态调整以提升资源利用率是用户更关心的问题，接下来从弹性伸缩、调度、在离线混部三大产品化的方向，详述如何提升资源利用率。

2.1 弹性伸缩

通过 HPA 按指标弹性扩缩容

在资源浪费场景2中，如果您的业务存在波峰波谷，固定的资源 Request 注定在波谷时会造成资源浪费，针对这样的场景，如果波峰的时候可以自动增加业务负载的副本数量，波谷的时候可以自动减少业务负载的副本数量，将有效提升资源整体利用率。
HPA（Horizontal Pod Autoscaler）可以基于一些指标（例如 CPU、内存的利用率）自动扩缩 Deployment 和 StatefulSet 中的 Pod 副本的数量，达到工作负载稳定的目的，真正做到按需使用。

HPA 使用场景

1. 流量突发：突然流量增加，负载过载时会自动增加 Pod 数量以及时响应。

2. 自动缩容：流量较少时，负载对资源的利用率过低时会自动减少 Pod 的数量以避免浪费。

TKE 上的 HPA

TKE 基于 Custom Metrics API 支持许多用于弹性伸缩的指标，涵盖 CPU、内存、硬盘、网络以及 GPU 相关的指标，覆盖绝大多数的 HPA 弹性伸缩场景，详细列表请参见 自动伸缩指标说明 。此外，针对例如基于业务单副本 QPS 大小来进行自动扩缩容等复杂场景，可通过安装 prometheus-adapter 来实现自动扩缩容，详情见 在 TKE 上使用自定义指标进行弹性伸缩 。

通过 HPC 定时扩缩容

假设您的业务是电商平台，双十一要进行促销活动，这时可以考虑使用 HPA 自动扩缩容。但是 HPA 需要先监控各项指标后，再进行反应，可能扩容速度不够快，无法及时承载高流量。针对这种有预期的流量暴增，如果能提前发生副本扩容，将有效承载流量井喷。
HPC（HorizontalPodCronscaler）是 TKE 自研组件，旨在定时控制副本数量，以达到提前扩缩容、和提前触发动态扩容时资源不足的影响，相较社区的 CronHPA，额外支持：

1. 与 HPA 结合：可以实现定时开启和关闭 HPA，让您的业务在高峰时更弹性。
2. 例外日期设置：业务的流量不太可能永远都是规律的，设置例外日期可以减少手工调整 HPC。
3. 单次执行：以往的 CronHPA 都是永久执行，类似 Cronjob，单次执行可以更灵活的应对大促场景。

HPC 使用场景

以游戏服务为例，从周五晚上到周日晚上，游戏玩家数量暴增。如果可以将游戏服务器在星期五晚上前扩大规模，并在星期日晚上后缩放为原始规模，则可以为玩家提供更好的体验。如果使用 HPA，可能因为扩容速度不够快导致服务受影响。

TKE 上的 HPC

TKE 上已经实现对 HPC 的产品化，但您需要提前在”组件管理“里面安装 HPC，HPC 使用 CronTab 语法格式。操作详情见 自动伸缩基本操作 。

通过 CA 自动调整节点数量

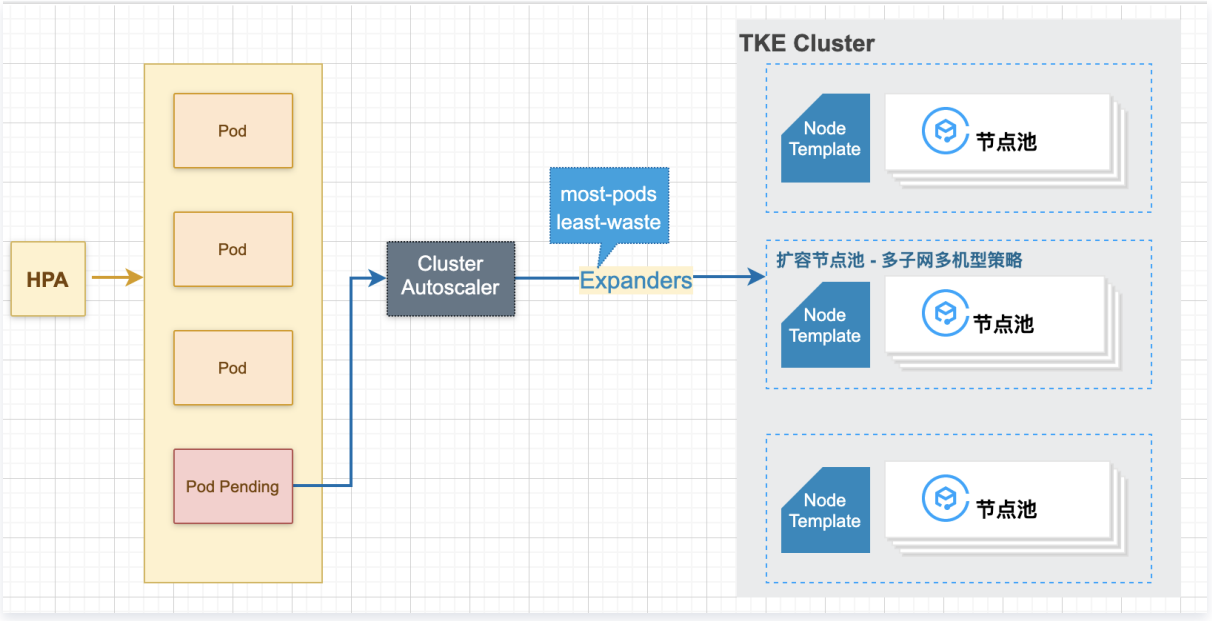
上述 HPA 和 HPC，都是在业务负载层面的自动扩缩副本数量，以灵活应对流量的波峰波谷，提升资源利用率。但是对于集群整体而言，资源总数是固定的，HPA 和 HPC 只是让集群有更多空余的资源，是否有一种方法，能在集群整体较“空”时回收部分资源，能在集群整体较“满”时扩充集群整体资源？因为集群整体资源的使用量直接决定了账单费用，这种集群级别的弹性扩缩将真正帮助您节省使用成本。
CA（Cluster Autoscaler）用于自动扩缩集群节点数量，以真正实现资源利用率的提升，并直接作用于用户的费用，是降本增效的关键。

CA 使用场景

1. 在业务波峰时，根据业务突增的负载扩容合适的节点。
2. 在业务波谷时，根据资源的空闲情况释放多余的节点。

TKE 上的 CA

TKE 上的 CA 是以节点池的形态来让用户使用的，CA 推荐和 HPA 一起使用：HPA 负责应用层的扩缩容，CA 负责资源层（节点层）的扩缩容，当 HPA 扩容造成集群整体资源不足时，会引发 Pod 的 Pending，Pod Pending 会触发 CA 扩充节点池以增加集群整体资源量，整体扩容逻辑可参考下图：



具体的参数配置方式以及应用场景可参考 《像管理 Pod 一样管理 Node》，或者可参考腾讯云容器服务 节点池概述 。

2.2 调度

Kubernetes 调度机制是 Kubernetes 原生提供的一种高效优雅的资源分配机制，它的核心功能是为每个 Pod 找到最适合它的节点，在 TKE 场景下，调度机制帮助实现了应用层弹性伸缩到资源层弹性伸缩的过渡。通过合理利用 Kubernetes 提供的调度能力，根据业务特性配置合理的调度策略，也能有效提高集群中的资源利用率。

节点亲和性

倘若您的某个业务是 CPU 密集型，不小心被 Kubernetes 的调度器调度到内存密集型的节点上，导致内存密集型的 CPU 被占满，但内存几乎没怎么用，会造成较大的资源浪费。如果您能为节点设置一个标记，表明这是一个 CPU 密集型的节点，然后在创建业务负载时也设置一个标记，表明这个负载是一个 CPU 密集型的负载，Kubernetes 的调度器会将这个负载调度到 CPU 密集型的节点上，这种寻找最合适的节点的方式，将有效提升资源利用率。创建 Pod 时，可以设置节点亲和性，即指定 Pod 想要调度到哪些节点上（这些节点是通过 K8s Label）来指定的。

节点亲和性使用场景

节点亲和性非常适合在一个集群中有不同资源需求的工作负载同时运行的场景。例如，腾讯云的 CVM（节点）有 CPU 密集型的机器，也有内存密集型的机器。如果某些业务对 CPU 的需求远大于内存，此时使用普通的 CVM 机器，势必会对内存造成较大浪费。此时可以在集群里添加一批 CPU 密集型的 CVM，并且把这些对 CPU 有较高需求的 Pod 调度到这些 CVM 上，这样可以提升 CVM 资源的整体利用率。同理，还可以在集群中管理异构节点（例如 GPU 机器），在需要 GPU 资源的工作负载中指定需要 GPU 资源的量，调度机制则会帮助您寻找合适的节点去运行这些工作负载。

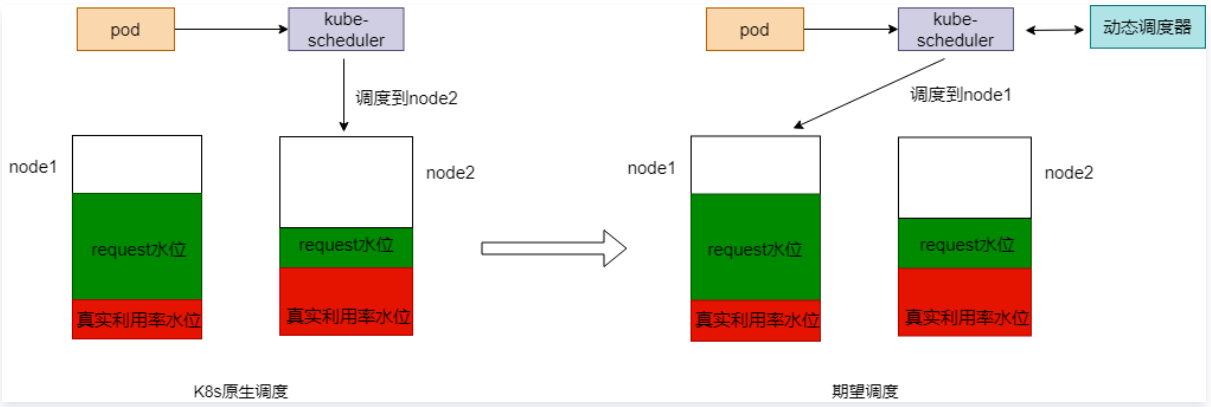
TKE 上的节点亲和性

TKE 提供与原生 Kubernetes 完全一致的亲和性使用方式，您可通过控制台或配置 YAML 的方式使用此项功能，详情见 资源合理分配 。

动态调度器

原生的 Kubernetes 调度策略倾向于调度 Pod 到节点剩余资源较多的节点上，例如默认的 LeastRequestedPriority 策略。但是原生调度策略存在一个问题：这样的资源分配是静态的，Request 不能代表资源真实使用情况，因此一定会存在一定程度的浪费。因此，如果调度器可以基于节点的实际资源利用率进行调度，将一定程度上解决资源浪费的问题。

TKE 自研的动态调度器解决了以上问题。动态调度器的核心原理如下图所示：



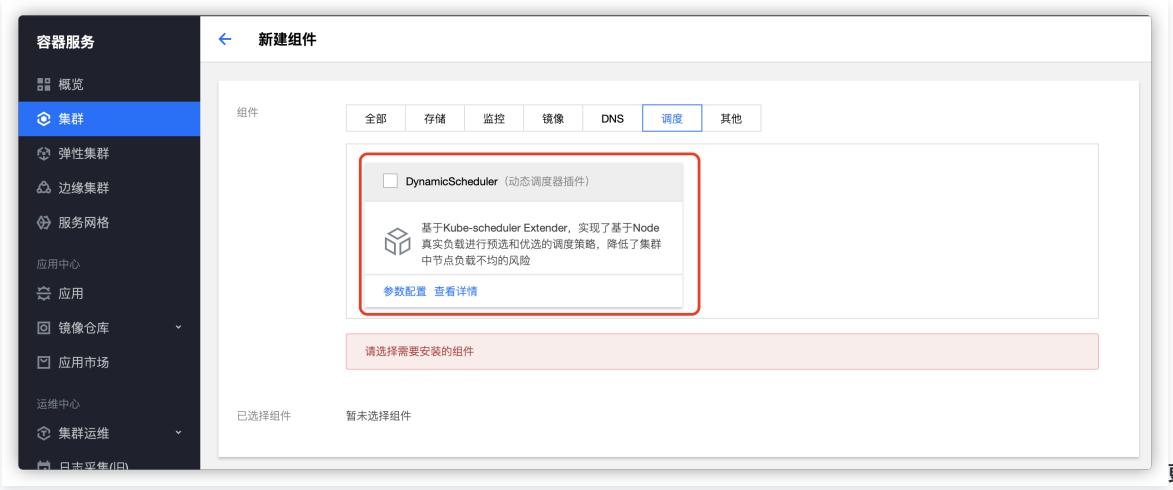
动态调度器的使用场景

除了降低资源浪费，动态调度器还可以很好的缓解集群调度热点的问题。

1. 动态调度器会统计过去一段时间调度到节点的 Pod 数目，避免往同一节点上调度过多的 Pod。
2. 动态调度器支持设置节点负载阈值，在调度阶段过滤掉超过阈值的节点。

TKE 上的动态调度器

您可以在扩展组件里面安装和使用动态调度器：



更多关于动态调度器的使用指南，可以参考《TKE 重磅推出全链路调度解决方案》和官方 DynamicScheduler 说明。

2.3 离在线业务混部

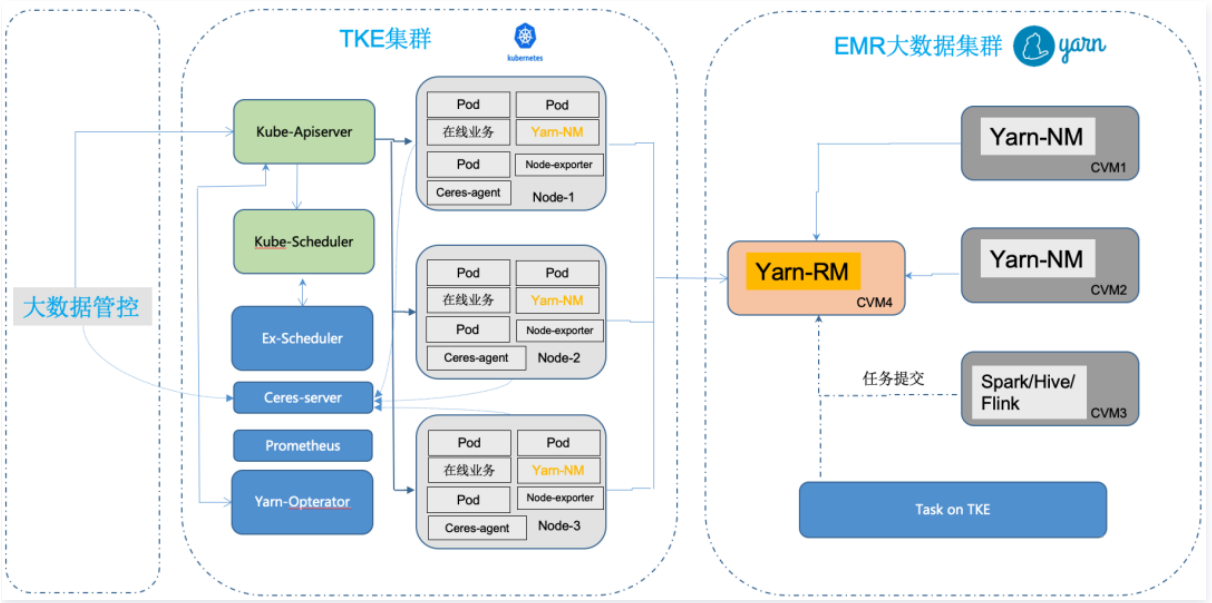
如果您既有在线 Web 服务业务，又有离线的计算服务业务，借助 TKE 的离在线业务混部技术可以动态调度和运行不同的业务，提升资源利用率。
在传统架构中，大数据业务和在线业务往往部署在不同的资源集群中，这两部分业务相互独立。但大数据业务一般更多的是离线计算类业务，在夜间处于业务高峰，而在线业务与之相反，夜间常常处于空载状态。云原生技术借助容器完整（CPU，内存，磁盘 IO，网络 IO 等）的隔离能力，及 Kubernetes 强大的编排调度能力，实现在线和离线业务混合部署，从而使离在线业务充分利用在线业务空闲时段的资源，以提高资源利用率。

离在线业务混部使用场景

在 Hadoop 架构下，离线作业和在线作业往往分属不同的集群，然而在线业务、流式作业具有明显的波峰波谷特性，在波谷时段，会有大量的资源处于闲置状态，造成资源的浪费和成本的提升。在离线混部集群，通过动态调度削峰填谷，当在线集群的使用率处于波谷时段，将离线任务调度到在线集群，可以显著的提高资源的利用率。然而，Hadoop Yarn 目前只能通过 NodeManager 上报的静态资源情况进行分配，无法基于动态资源调度，无法很好的支持在线、离线业务混部的场景。

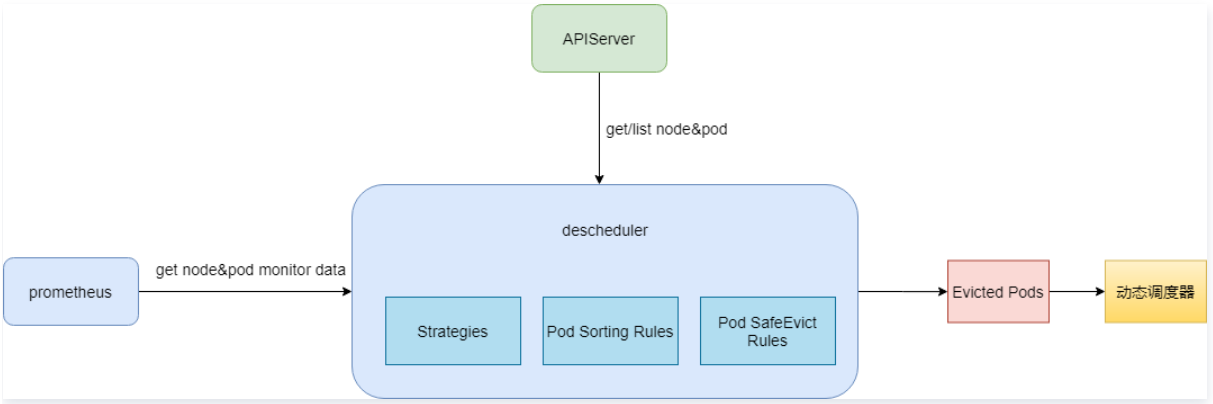
TKE 上的离在线混部

在线业务具有明显的波峰浪谷特征，而且规律比较明显，尤其是在夜间，资源利用率比较低，这时候大数据管控平台向 Kubernetes 集群下发创建资源的请求，可以提高大数据应用的算力，详情见 大数据系统云原生渐进式演进最佳实践 。



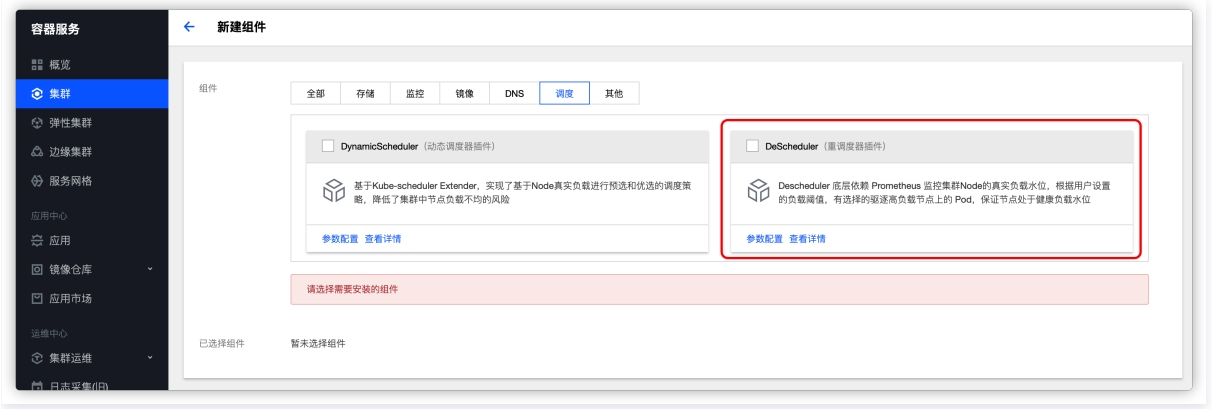
如何权衡资源利用率与稳定性

在企业的运维工作中，除了成本，系统的稳定性也是十分重要的指标。如何在两者间达到平衡，可能是很多运维人员心中的“痛点”。一方面，为了降低成本，资源利用率当然是越高越好，但是资源利用率达到一定水位后，负载过高极有可能导致业务 OOM 或 CPU 抖动等问题。
为了减小企业成本控制之路上的顾虑，TKE 还提供了**重调度器**来保障集群负载水位在可控范围内。重调度器与动态调度器的关系可以参考下图，重调度器主要负责“保护”节点中已经负载比较“危险”的节点，优雅驱逐这些节点上的业务。



TKE 上的重调度器

您可以在扩展组件中安装和使用重调度器，安装组件详情见 DeScheduler 说明 。



更多关于重调度器的使用指南，可参考《TKE 重磅推出全链路调度解决方案》。

混合云
IDC 集群添加超级节点用于弹性扩容

最近更新时间：2024-04-10 11:02:51

使用场景

IDC 的资源有限，当需要应对业务突发流量，IDC 内的算力资源不足以应对时，可以选择使用公有云资源应对临时流量。TKE Resilience Chart 利用 TKE Serverless 容器服务，基于自定义的调度策略，通过添加超级节点的方式，将用户集群中的工作负载弹性上云，使用户 IDC 集群获得极大的弹性拓展能力，优势如下：

1. 用户 IDC / 私有云的硬件和维护成本保持不变。
2. 实现了用户 IDC / 私有云和公有云级别的应用高可用。
3. 用户按需使用公有云的资源，按需付费。

使用须知

1. 已开通 TKE Serverless 集群。
2. 用户 IDC 与腾讯云 VPC 通过专线内网互联。
3. IDC 集群的 API Server 地址，腾讯云 VPC 网络可达。
4. 用户自有 IDC 集群可以访问公网，需要通过公网调用云 API。

TKE Resilience Chart 特性说明

组件说明

TKE Resilience Chart 主要是由超级节点管理器，调度器，容忍控制器3部分组成，如表格所示：

简称	组件名称	描述
eklet	超级节点管理器	负责 Podsandbox 生命周期的管理，并对外提供原生 kubelet 与节点相关的接口。
tke-scheduler	调度器	负责根据调度策略将 workload 弹性上云，仅会安装在非 TKE 发行版的 K8S 集群上，TKE 发行版集群不会安装此组件。其中 TKE 发行版（TKE Kubernetes Distro）是由腾讯云 TKE 发布的 K8S 发行版本，用于帮助用户创建与云上 TKE 完全一致的 K8S 集群，目前 TKE 发行版集群已经在 GitHub 开源，详情见 TKE Kubernetes Distro。
admission-controller	容忍控制器	负责为处于 `pending` 状态的 Pod 添加容忍，使其可以调度到超级节点上。

主要特性

1. 如需要 TKE Serverless Pod 和本地集群的 Pod 互通，则要求本地集群是 Underlay 的网络模型（使用 Calico 之类的基于 BGP 路由，而不是 SDN 封装的 CNI 插件），并且需要在腾讯云 VPC 中添加本地 Pod CIDR 的路由信息，详情见 路由配置。
2. Workload resilience 特性控制开关 `AUTO_SCALE_EKS=true|false` 分为全局开关和局部开关，用来控制 workload 在 `pending` 的情况下是否弹性调度到腾讯云 TKE Serverless，如表格所示：
   - 全局开关：`kubectl get cm -n kube-system eks-config` 中 `AUTO_SCALE_EKS`，默认开启。
   - 局部开关：`spec.template.metadata.annotations ['AUTO_SCALE_EKS']`

全局开关	局部开关	行为
AUTO_SCALE_EKS=true	AUTO_SCALE_EKS=false	调度成功
AUTO_SCALE_EKS=true	未定义	调度成功
AUTO_SCALE_EKS=true	AUTO_SCALE_EKS=true	调度成功
AUTO_SCALE_EKS=false	AUTO_SCALE_EKS=false	调度失败

AUTO_SCALE_EKS=false	未定义	调度失败
AUTO_SCALE_EKS=false	AUTO_SCALE_EKS=true	调度成功
未定义	AUTO_SCALE_EKS=false	调度成功
未定义	未定义	调度成功
未定义	AUTO_SCALE_EKS=true	调度成功

3. 当使用社区版 K8S 的时候，需要在 workload 中指定调度器为 `tke-scheduler`，TKE 发行版 K8S 则不需要指定调度器。
4. Workload 设定本地集群保留副本数量 `LOCAL_REPLICAS: N` 。
5. Workload 扩容：
   - 当本地集群资源不足，并满足全局和局部开关中**调度成功**的行为设定， `pending` 的 workload 将扩容到腾讯云 EKS。
   - 当实际创建 workload 副本数量达到 N 后，并满足全局和局部开关中**调度成功**的行为设定， `pending` 的 workload 将扩容到 TKE Serverless 集群。
6. Workload 缩容：
   - TKE 发行版 K8S 会优先缩容 TKE Serverless 集群上的实例。
   - 社区版 K8S 会随机缩容。
7. 调度规则的限制条件：
   - 无法调度 DaemonSet Pod 到超级节点，此特性只在 TKE 发行版 K8S 有效，社区版 K8S DaemonSet Pod 会调度到超级节点，但会显示 `DaemonsetForbidden` 。
   - 无法调度 kube-system，tke-eni-ip-webhook 命名空间下的 Pod 到超级节点。
   - 无法调度 securityContext.sysctls ["net.ipv4.ip_local_port_range"] 的值包含61000 - 65534的端口。
   - 无法调度 Pod.Annotations [tke.cloud.tencent.com/vpc-ip-claim-delete-policy] 的 Pod。
   - 无法调度 container (initContainer).ports [].containerPort (hostPort) 包含61000 - 65534的端口。
   - 无法调度 container (initContainer) 中探针指定61000 - 65534的端口。
   - 无法调度除了 nfs，Cephfs，hostPath，qcloudcbs 以外的 PV。
   - 无法调度启用固定 IP 特性的 Pod 到超级节点。
8. 超级节点支持自定义默认 DNS 配置：用户在超级节点上新增 `eks.tke.cloud.tencent.com/resolv-conf` 的 annotation 后，生成的 cxm 子机里的 /etc/resolv.conf 就会被更新成用户定义的内容。

> ⚠ **注意：**
> 会覆盖原来超级节点的 DNS 配置，最终以用户的配置为准。

```
eks.tke.cloud.tencent.com/resolv-conf: |
nameserver 4.4.4.4
nameserver 8.8.8.8
```

操作步骤

获取 tke-resilience helm chart

```
git clone https://github.com/tkestack/charts.git
```

配置相关信息

编辑 charts/incubator/tke-resilience/values.yaml，配置以下信息：

```
cloud:
  uniqueID:"{IDC集群的唯一标识符，用于区分账号下不同集群，不超过16个字节，提交后不可修改，当账号下有多个集群时必填}"
  appID: "{腾讯云账号 APPID}"
  ownerUIN: "{腾讯云用户账号 ID}"
  secretID: "{腾讯云账号 secretID}"
```

```
  secretKey: "{腾讯云账号 secretKey}"
  vpcID: "{EKS Pod 放置的 VPC ID}"
  regionShort: "{EKS Pod 放置的 region 简称}"
  regionLong: "{EKS Pod 放置的 region 全称}"
  subnets:
    - id: "{EKS Pod 放置的子网 ID}"
      zone: "{EKS Pod 放置的可用区}"
eklet:
  PodUsedApiserver: "{当前集群的 API Server 地址}"
```

说明：
TKE Serverless 容器服务支持售卖的地域和可用区请参见 地域和可用区 。

安装 TKE Resilience Chart

您可通过 本地 Helm 客户端连接集群 。
执行以下命令，在第三方集群中通过 Helm Chart 安装 TKE Resilience Chart。

```
helm install tke-resilience --namespace kube-system ./tke-resilience --debug
```

执行以下命令，确认 Helm 应用中组件是否安装完成。本文以 TKE 发行版的集群为例，未安装 tke-scheduler。

```
# kubectl get Pod -n kube-system | grep resilience
eklet-tke-resilience-5f9dcd99df-rgsmc            1/1     Running   0          43h
eks-admission-tke-resilience-5bb588dc44-9hvhs    1/1     Running   0          44h
```

查看集群中已经部署了1个超级节点。

```
# kubectl get node
NAME                     STATUS   ROLES    AGE    VERSION
10.0.1.xx                Ready    <none>   2d4h   v1.20.4-tke.1
10.0.1.xx                Ready    master   2d4h   v1.20.4-tke.1
eklet-subnet-xxxxxxxx    Ready    <none>   43h    v2.4.6
```

创建测试用例

创建 demo 应用 `nginx-deployment` ，该应用有4个副本，其中3个在 TKE Serverless 集群，1个在本地集群，Yaml 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      annotations:
        AUTO_SCALE_EKS: "true"
        LOCAL_REPLICAS: "1" #设置本地集群运行的副本数为 1
      labels:
```

```
        app: nginx
    spec:
      #schedulerName: tke-scheduler 如果是第三方集群则需要执行调度器为 tke-scheduler
      containers:
      - name: nginx
        image: nginx
        imagePullPolicy: IfNotPresent
```

验证副本的状态以及分布，符合预期。

```
# kubectl get Pod -owide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
nginx-deployment-77b9b9bc97-cq9ds   1/1     Running   0          27s   10.232.1.88   10.0.1.xxx
<none>           <none>
nginx-deployment-77b9b9bc97-s9vzc   1/1     Running   0          27s   10.0.1.118    eklet-subnet-
xxxxxxxx   <none>           <none>
nginx-deployment-77b9b9bc97-sd4z5   1/1     Running   0          27s   10.0.1.7      eklet-subnet-
xxxxxxxx   <none>           <none>
nginx-deployment-77b9b9bc97-z86tx   1/1     Running   0          27s   10.0.1.133    eklet-subnet-
xxxxxxxx   <none>           <none>
```

并验证缩容的特性，由于使用的是 TKE 发行版的集群，会优先缩容 TKE Serverless 集群的实例。这里应用的副本数从4调整为3。

```
# kubectl scale deployment nginx-deployment --replicas=3
```

由以下结果可以看出，优先缩容了云上的副本，符合预期。

```
# kubectl get Pod -owide
NAME                                READY   STATUS    RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES
nginx-deployment-77b9b9bc97-cq9ds   1/1     Running   0          7m38s   10.232.1.88   10.0.1.xxx
<none>           <none>
nginx-deployment-77b9b9bc97-s9vzc   1/1     Running   0          7m38s   10.0.1.118    eklet-subnet-
xxxxxxxx   <none>           <none>
nginx-deployment-77b9b9bc97-sd4z5   1/1     Running   0          7m38s   10.0.1.7      eklet-subnet-
xxxxxxxx   <none>           <none>
```

AI
在 TKE 上部署 AI 大模型

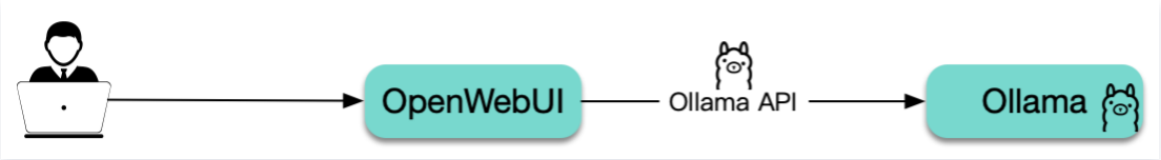
最近更新时间：2025-02-24 10:22:42

概述

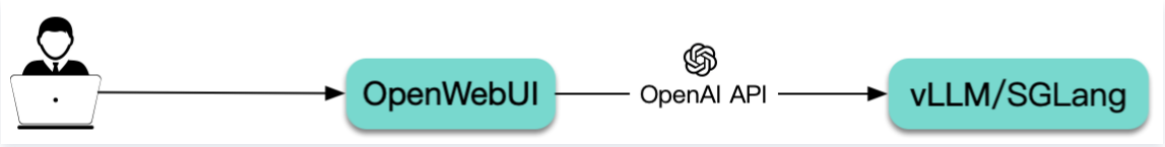
本文介绍如何在 TKE 上部署 AI 大模型，以 DeepSeek-R1 为例。使用 Ollama、vLLM 或 SGLang 等工具运行大模型并对外暴露 API，同时结合 OpenWebUI 提供交互界面。

部署架构

- Ollama：提供的是 Ollama API。



- vLLM 和 SGLang：两者均提供兼容 OpenAI 的 API。



工具介绍

- Ollama 是一个运行大模型的工具，可以看作是大模型领域的 Docker，能够下载所需的大模型并暴露 API，简化大模型的部署。
- vLLM 与 Ollama 类似，也是一个运行大模型的工具，针对推理做了很多优化，提高了模型的运行效率和性能，使得在资源有限的情况下也能高效运行大语言模型，并且提供兼容 OpenAI 的 API。
- SGLang 与 vLLM 类似，性能更强，且针对 DeepSeek 做了深度优化，也是 DeepSeek 官方推荐的工具。
- OpenWebUI 是一个大模型的 Web UI 交互工具，支持通过 Ollama 和 OpenAI 两种 API 与大模型进行互动。

技术选型

Ollama、vLLM 还是 SGLang?

- **Ollama**：适合个人开发者或在本地开发环境中快速上手，具备良好的 GPU 硬件和大模型兼容性，易于配置，但在性能方面略逊于 vLLM。
- **vLLM**：推理性能更好，也更节约资源，适合部署到服务器供多人协作，支持多机多卡分布式部署，上限更高，但能适配的 GPU 硬件比 Ollama 少，且需要根据不同 GPU 和大模型来调整 vLLM 的启动参数才能跑起来或者获得更好的性能表现。
- **SGLang**：新兴的高性能解决方案，针对特定模型（如 DeepSeek）优化，具有更高的吞吐量。
- **选型建议**：对于有一定技术基础并能投入精力调试的用户，优先考虑使用 vLLM 或 SGLang 在 Kubernetes 集群中进行部署；若追求简便快捷，则可以选择 Ollama。文中将对这两种方案分别给出详细的部署步骤。

AI 大模型数据如何存储?

AI 大模型通常占用大量存储空间，直接将其打包到容器镜像中并不现实。如果在启动时通过 `initContainers` 自动下载模型文件，会导致启动时间过长，因此建议使用共享存储来挂载 AI 大模型（即先通过 Job 任务将模型下载至共享存储，随后将该存储卷挂载到大模型运行的 Pod 内）。如此一来，后续 Pod 启动即可跳过模型下载环节，尽管仍需从共享存储通过网络加载模型，但如果选用高性能的共享存储（例如 Turbo 类型），这一过程依然迅速有效。
在腾讯云上，可以使用文件存储 CFS 作为共享存储，CFS 具有高性能和高可用性，适合存储 AI 大模型。本文示例将使用 CFS 来存储 AI 大模型。

GPU 机型如何选?

不同机型搭载的 GPU 型号各异，请参照 GPU 计算型实例 和 GPU 渲染型实例 获取对应关系。相较于 vLLM，Ollama 对各类 GPU 的支持范围更广，兼容性更佳。建议您首先明确所选工具及目标大模型的需求，据此挑选合适的 GPU 型号，进而依据上述对照表确定要使用的 GPU 机型。此外，务必关注所选机型在各地区的销售状态及库存情况，可通过 购买云服务器 页面进行查询（**实例族选择 GPU 机型**）。

镜像说明

本文中示例使用的镜像都是官方提供的镜像 tag 为 latest，建议您根据自己的需求将这些镜像替换为特定版本的 tag。您可以访问以下这些链接来查看镜像的 tag 列表：

- SGLang：lmsysorg/sglang
- Ollama：ollama/ollama
- vLLM：vllm/vllm-openai

这些官方镜像均托管在 DockerHub 上，且体积较大。在 TKE 环境中，默认提供免费的 DockerHub 镜像加速服务。中国大陆用户也可以直接从 DockerHub 拉取镜像，但速度可能较慢，尤其是对于较大的镜像，等待时间会更长。为提高镜像拉取速度，建议将镜像同步至 容器镜像服务 TCR，并在 YAML 文件中替换相应的镜像地址，这样可以显著加快镜像的拉取速度。

操作步骤

步骤1：准备集群

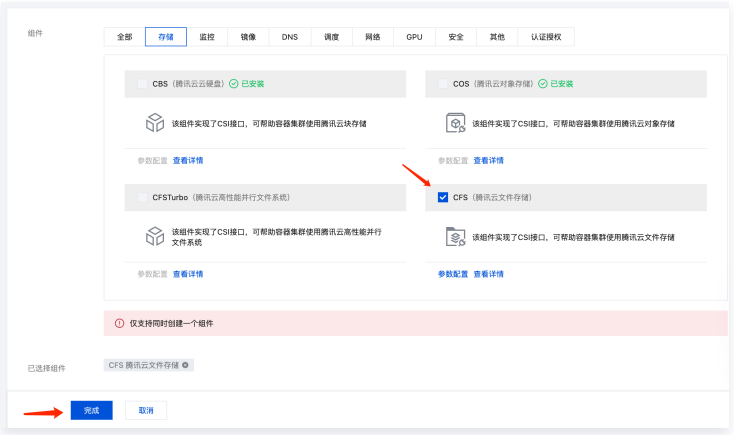
登录 容器服务控制台，创建一个集群，集群类型选择 **TKE 标准集群**。详情请参见 创建集群 。

步骤2：准备 CFS 存储

安装 CFS 组件

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧菜单栏中的**组件管理**，在组件页面单击**新建**。
3. 在新建组件管理页面中勾选 **CFS（腾讯云文件存储）**。

> **说明：**
> - 支持选择 **CFS（腾讯云文件存储）**或 **CFSTurbo（腾讯云高性能并行文件系统）**，本文以 CFS（腾讯云文件存储）为例。
> - CFS-Turbo 的性能更强，读写速度更快，但成本也更高。如果希望大模型运行速度更快，可以考虑使用 CFS-Turbo。



4. 单击**完成**即可创建组件。

新建 StorageClass

> **说明：**
> 该步骤选择项较多，因此本文示例通过容器服务控制台来创建 PVC。若您希望通过 YAML 来创建，可以先用控制台创建一个测试 PVC，然后复制生成的 YAML 文件。

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧菜单栏中的**存储**，在 StorageClass 页面单击**新建**。
3. 在新建存储页面，根据实际需求，创建 CFS 类型的 StorageClass。如下图所示：

> **说明：**
> 如果是新建 CFS-Turbo StorageClass，则需要在 文件存储控制台 创建一个 CFS-Turbo 文件系统，然后，在创建 StorageClass 时，引用对应的 CFS-Turbo 实例。

- 名称：请输入 StorageClass 名称，本文以 “cfs-ai” 为例。
- Provisioner：选择 “文件存储 CFS”。
- 存储类型：建议选择“性能存储”，其读写速度比“标准存储”更快。

4. 单击**创建 StorageClass**，完成创建。

创建 PVC

1. 登录 容器服务控制台，在集群管理页面，选择集群 ID，进入集群的基本信息页面。
2. 单击页面右上角的 **YAML 创建**，进入 Yaml 创建资源页面。
3. 复制以下代码，创建一个 CFS 类型的 PVC，用于存储 AI 大模型：

> **注意：**
> - 请根据实际情况替换 storageClassName。
> - 对于 CFS 而言，storage 大小可以随意指定，因为费用是根据实际占用的空间计算的。

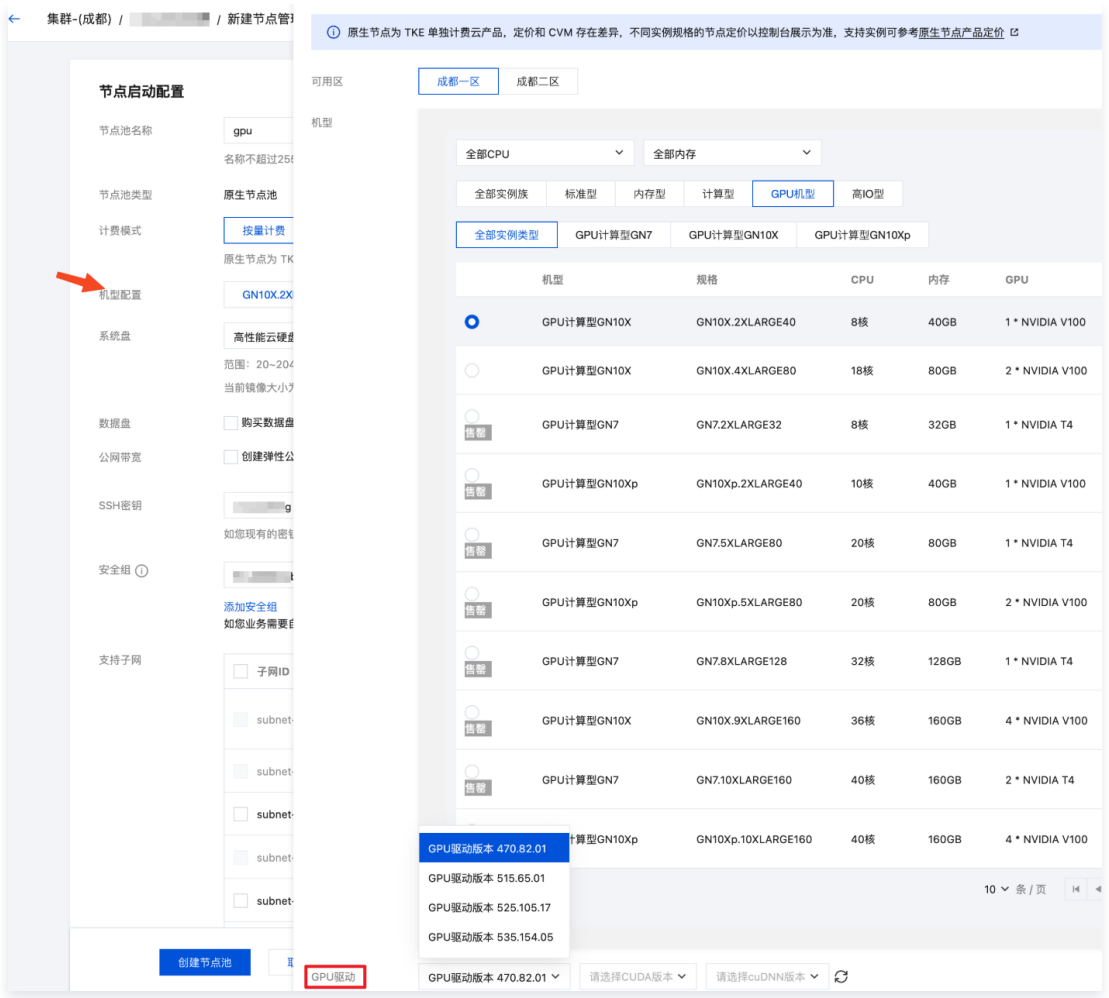
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ai-model
  labels:
    app: ai-model
spec:
  storageClassName: cfs-ai
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 100Gi
```

4. 创建另一个 PVC 给 OpenWebUI 使用，可使用同一个 `storageClassName`：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: webui
  labels:
    app: webui
spec:
  accessModes:
  - ReadWriteMany
  storageClassName: cfs-ai
  resources:
    requests:
      storage: 100Gi
```

步骤3：新建 GPU 节点池

1. 在集群管理页面，选择集群 ID，进入集群的基本信息页面。
2. 选择左侧菜单栏中的**节点管理**，在**节点池**页面单击**新建**。
3. 选择节点类型。配置详情请参见 创建节点池 。
   - 如果选择**原生节点**或**普通节点**：
     - 建议选择较新的操作系统。
     - 系统盘和数据盘默认大小为50GB，建议增加，如200GB，以避免因 AI 相关镜像较大而导致的节点磁盘空间压力大。
     - 在机型配置中的 GPU 机型中选择一个符合需求且未售罄的机型，如果有 GPU 驱动选项，选择最新版本。



- 如果选择**超级节点**：超级节点是虚拟节点，每个 Pod 都是独占的轻量虚拟机，因此无需选择机型，在部署时通过 Pod 注解指定 GPU 卡的型号（后续示例中会有说明）。

4. 单击**创建节点池**。

> **说明：**
> GPU 插件无需显式安装。对于普通节点或原生节点，配置了 GPU 机型后，系统会自动安装 GPU 插件。对于超级节点，无需安装 GPU 插件。

步骤4：使用 Job 下载 AI 大模型

下发一个 Job，将所需的 AI 大模型下载到 CFS 共享存储中。以下分别是 vLLM、SGLang 和 Ollama 的 Job 示例：

> **注意：**
> - 请使用之前的 Ollama 或 vLLM 镜像执行脚本，下载所需的 AI 大模型。本示例中下载的是 DeepSeek-R1 模型。可以通过修改 `LLM_MODEL` 环境变量来替换所需的大语言模型。
> - 如果使用 Ollama，可以在 Ollama 模型库 中查询和搜索所需的模型。
> - 如果使用 vLLM，可以在 Hugging Face 模型库 和 ModelScope 模型库 查询和搜索所需的模型。在中国大陆环境中，建议使用 ModelScope 模型库，以避免因网络问题导致的下载失败，通过 `USE_MODELSCOPE` 环境变量控制是否从 ModelScope 下载。

vLLM Job

vllm-download-model-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
```

```
  name: vllm-download-model
  labels:
    app: vllm-download-model
spec:
  template:
    metadata:
      name: vllm-download-model
      labels:
        app: vllm-download-model
      annotations:
        eks.tke.cloud.tencent.com/root-cbs-size: '100' # 如果用超级节点，默认系统盘只有 20Gi，vllm 镜像解压后会撑爆磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
    spec:
      containers:
      - name: vllm
        image: vllm/vllm-openai:latest
        env:
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-7B
        - name: USE_MODELSCOPE
          value: "1"
        command:
        - bash
        - -c
        - |
          set -ex
          if [[ "$USE_MODELSCOPE" == "1" ]]; then
            exec modelscope download --local_dir=/data/$LLM_MODEL --model="$LLM_MODEL"
          else
            exec huggingface-cli download --local-dir=/data/$LLM_MODEL $LLM_MODEL
          fi
        volumeMounts:
        - name: data
          mountPath: /data
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      restartPolicy: OnFailure
```

SGLang Job

sglang-download-model-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sglang-download-model
  labels:
    app: sglang-download-model
spec:
  template:
    metadata:
      name: sglang-download-model
      labels:
        app: sglang-download-model
      annotations:
```

```
        eks.tke.cloud.tencent.com/root-cbs-size: '100' # 如果用超级节点，默认系统盘只有 20Gi，sglang 镜像解压后会撑爆磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
    spec:
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        env:
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
        - name: USE_MODELSCOPE
          value: "1"
        command:
        - bash
        - -c
        - |
          set -ex
          if [[ "$USE_MODELSCOPE" == "1" ]]; then
            exec modelscope download --local_dir=/data/$LLM_MODEL --model="$LLM_MODEL"
          else
            exec huggingface-cli download --local-dir=/data/$LLM_MODEL $LLM_MODEL
          fi
        volumeMounts:
        - name: data
          mountPath: /data
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      restartPolicy: OnFailure
```

Ollama Job

ollama-download-model-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: ollama-download-model
  labels:
    app: ollama-download-model
spec:
  template:
    metadata:
      name: ollama-download-model
      labels:
        app: ollama-download-model
    spec:
      containers:
      - name: ollama
        image: ollama/ollama:latest
        env:
        - name: LLM_MODEL
          value: deepseek-r1:7b
        command:
        - bash
        - -c
        - |
```

```
          set -ex
          ollama serve &
          sleep 5 # sleep 5 seconds to wait for ollama to start
          exec ollama pull $LLM_MODEL
        volumeMounts:
        - name: data
          mountPath: /root/.ollama # ollama 的模型数据存储在 `/root/.ollama` 目录下，挂载 CFS 类型的 PVC 到
该路径。
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      restartPolicy: OnFailure
```

步骤5：部署 Ollama、vLLM 或 SGLang

部署 vLLM

通过 Deployment 部署 vLLM：

原生节点或普通节点

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vllm
  labels:
    app: vllm
spec:
  selector:
    matchLabels:
      app: vllm
  replicas: 1
  template:
    metadata:
      labels:
        app: vllm
    spec:
      containers:
      - name: vllm
        image: vllm/vllm-openai:latest
        imagePullPolicy: Always
        env:
        - name: PYTORCH_CUDA_ALLOC_CONF
          value: expandable_segments:True
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-7B
        command:
        - bash
        - -c
        - |
          vllm serve /data/$LLM_MODEL \
            --served-model-name $LLM_MODEL \
            --host 0.0.0.0 \
            --port 8000 \
            --trust-remote-code \
            --enable-chunked-prefill \
```

```
            --max_num_batched_tokens 1024 \
            --max_model_len 1024 \
            --enforce-eager \
            --tensor-parallel-size 1
        securityContext:
          runAsNonRoot: false
        ports:
        - containerPort: 8000
        readinessProbe:
          failureThreshold: 3
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 5
          periodSeconds: 5
        resources:
          requests:
            cpu: 2000m
            memory: 2Gi
            nvidia.com/gpu: "1"
          limits:
            nvidia.com/gpu: "1"
        volumeMounts:
        - name: data
          mountPath: /data
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      # vLLM needs to access the host's shared memory for tensor parallel inference.
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: "2Gi"
      restartPolicy: Always


---

apiVersion: v1
kind: Service
metadata:
  name: vllm-api
spec:
  selector:
    app: vllm
  type: ClusterIP
  ports:
  - name: api
    protocol: TCP
    port: 8000
    targetPort: 8000
```

超级节点

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: vllm
  labels:
    app: vllm
spec:
  selector:
    matchLabels:
      app: vllm
  replicas: 1
  template:
    metadata:
      labels:
        app: vllm
      annotations:
        eks.tke.cloud.tencent.com/gpu-type: V100 # 指定 GPU 卡型号
        eks.tke.cloud.tencent.com/root-cbs-size: '100' # 超级节点系统盘默认只有 20Gi，vllm 镜像解压后会撑爆
磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
    spec:
      containers:
      - name: vllm
        image: vllm/vllm-openai:latest
        imagePullPolicy: Always
        env:
        - name: PYTORCH_CUDA_ALLOC_CONF
          value: expandable_segments:True
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-7B
        command:
        - bash
        - -c
        - |
          vllm serve /data/$LLM_MODEL \
            --served-model-name $LLM_MODEL \
            --host 0.0.0.0 \
            --port 8000 \
            --trust-remote-code \
            --enable-chunked-prefill \
            --max_num_batched_tokens 1024 \
            --max_model_len 1024 \
            --enforce-eager \
            --tensor-parallel-size 1
        securityContext:
          runAsNonRoot: false
        ports:
        - containerPort: 8000
        readinessProbe:
          failureThreshold: 3
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 5
          periodSeconds: 5
        resources:
          requests:
            cpu: 2000m
            memory: 2Gi
            nvidia.com/gpu: "1"
          limits:
            nvidia.com/gpu: "1"
        volumeMounts:
```

```
        - name: data
          mountPath: /data
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      # vLLM needs to access the host's shared memory for tensor parallel inference.
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: "2Gi"
      restartPolicy: Always

---

apiVersion: v1
kind: Service
metadata:
  name: vllm-api
spec:
  selector:
    app: vllm
  type: ClusterIP
  ports:
  - name: api
    protocol: TCP
    port: 8000
    targetPort: 8000
```

1. `--served-model-name` 参数指定大模型名称，应与前面下载 Job 中指定的名称一致。
2. 模型数据引用前面下载 Job 使用的 PVC，并挂载到 `/data` 目录下。
3. vLLM 监听 8000 端口暴露 API，定义 Service 以便后续被 OpenWebUI 调用。

部署 SGLang

通过 Deployment 部署 SGLang：

原生节点或普通节点

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sglang
  labels:
    app: sglang
spec:
  selector:
    matchLabels:
      app: sglang
  replicas: 1
  template:
    metadata:
      labels:
        app: sglang
    spec:
```

```
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        env:
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
        command:
        - bash
        - -c
        - |
          set -x
          exec python3 -m sglang.launch_server \
            --host 0.0.0.0 \
            --port 30000 \
            --model-path /data/$LLM_MODEL
        resources:
          limits:
            nvidia.com/gpu: "1"
        ports:
        - containerPort: 30000
        volumeMounts:
        - name: data
          mountPath: /data
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: 40Gi
      restartPolicy: Always

---
apiVersion: v1
kind: Service
metadata:
  name: sglang
spec:
  selector:
    app: sglang
  type: ClusterIP
  ports:
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

超级节点

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sglang
  labels:
    app: sglang
spec:
```

```
  selector:
    matchLabels:
      app: sglang
  replicas: 1
  template:
    metadata:
      labels:
        app: sglang
      annotations:
        eks.tke.cloud.tencent.com/gpu-type: V100 # 指定 GPU 卡型号
        eks.tke.cloud.tencent.com/root-cbs-size: '100' # 超级节点系统盘默认只有 20Gi，sglang 镜像解压后会撑爆磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
    spec:
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        env:
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
        command:
        - bash
        - -c
        - |
          set -x
          exec python3 -m sglang.launch_server \
            --host 0.0.0.0 \
            --port 30000 \
            --model-path /data/$LLM_MODEL
        resources:
          limits:
            nvidia.com/gpu: "1"
        ports:
        - containerPort: 30000
        volumeMounts:
        - name: data
          mountPath: /data
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: 40Gi
      restartPolicy: Always

---
apiVersion: v1
kind: Service
metadata:
  name: sglang
spec:
  selector:
    app: sglang
  type: ClusterIP
  ports:
  - name: api
    protocol: TCP
```

```
    port: 30000
    targetPort: 30000
```

1. `LLM_MODEL` 环境变量指定大模型名称，应与前面下载 Job 中指定的名称一致。
2. 模型数据引用前面下载 Job 使用的 PVC，并挂载到 `/data` 目录下。
3. SGLang 监听 30000 端口暴露 API，并定义 Service 以便后续被 OpenWebUI 调用。

部署 Ollama

通过 Deployment 部署 Ollama：

原生节点或普通节点

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ollama
  labels:
    app: ollama
spec:
  selector:
    matchLabels:
      app: ollama
  replicas: 1
  template:
    metadata:
      labels:
        app: ollama
    spec:
      containers:
      - name: ollama
        image: ollama/ollama:latest
        imagePullPolicy: IfNotPresent
        command: ["ollama", "serve"]
        env:
        - name: OLLAMA_HOST
          value: ":11434"
        resources:
          requests:
            cpu: 2000m
            memory: 2Gi
            nvidia.com/gpu: "1"
          limits:
            cpu: 4000m
            memory: 4Gi
            nvidia.com/gpu: "1"
        ports:
        - containerPort: 11434
          name: ollama
        volumeMounts:
        - name: data
          mountPath: /root/.ollama
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      restartPolicy: Always
```

```
---

apiVersion: v1
kind: Service
metadata:
  name: ollama
spec:
  selector:
    app: ollama
  type: ClusterIP
  ports:
  - name: server
    protocol: TCP
    port: 11434
    targetPort: 11434
```

超级节点

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ollama
  labels:
    app: ollama
spec:
  selector:
    matchLabels:
      app: ollama
  replicas: 1
  template:
    metadata:
      labels:
        app: ollama
      annotations:
        eks.tke.cloud.tencent.com/gpu-type: V100
    spec:
      containers:
      - name: ollama
        image: ollama/ollama:latest
        imagePullPolicy: IfNotPresent
        command: ["ollama", "serve"]
        env:
        - name: OLLAMA_HOST
          value: ":11434"
        resources:
          requests:
            cpu: 2000m
            memory: 2Gi
            nvidia.com/gpu: "1"
          limits:
            cpu: 4000m
            memory: 4Gi
            nvidia.com/gpu: "1"
        ports:
        - containerPort: 11434
          name: ollama
        volumeMounts:
        - name: data
```

```
          mountPath: /root/.ollama
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      restartPolicy: Always

---

apiVersion: v1
kind: Service
metadata:
  name: ollama
spec:
  selector:
    app: ollama
  type: ClusterIP
  ports:
  - name: server
    protocol: TCP
    port: 11434
    targetPort: 11434
```

1. Ollama 的模型数据存储在 `/root/.ollama` 目录下，因此需要将已经下载好 AI 大模型的 CFS 类型 PVC 挂载到该路径。
2. Ollama 监听 11434 端口暴露 API，并定义 Service 以便后续被 OpenWebUI 调用。
3. Ollama 默认监听的是回环地址(`127.0.0.1`)，通过指定 `OLLAMA_HOST` 环境变量，强制对外暴露 11434 端口。

⚠ 注意：

- 运行大模型需要使用 GPU，因此在 requests/limits 中指定了 `nvidia.com/gpu` 资源，以便让 Pod 调度到 GPU 机型并分配 GPU 卡使用。
- 如果希望大模型运行在超级节点，可以通过 Pod 注解 `eks.tke.cloud.tencent.com/gpu-type` 指定 GPU 类型，可选 V100、T4、A10*PNV4、A10*GNV4，具体可参考 GPU 规格。

步骤6：配置 GPU 弹性伸缩

如果需要对 GPU 资源实施弹性伸缩，可参照以下步骤进行配置。GPU 的 Pod 提供多种监控指标，详情请参见 GPU 监控指标，您可以根据这些指标配置 HPA 以实现 GPU Pod 的弹性伸缩。例如，基于 GPU 利用率的配置示例如下：

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: vllm
spec:
  minReplicas: 1
  maxReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: vllm
  metrics: # 更多 GPU 指标参考 https://cloud.tencent.com/document/product/457/38929#gpu
  - pods:
      metric:
        name: k8s_pod_rate_gpu_used_request # GPU利用率 （占 Request）
      target:
        averageValue: "80"
        type: AverageValue
    type: Pods
  behavior:
```

```
    scaleDown:
      policies:
      - periodSeconds: 15
        type: Percent
        value: 100
      selectPolicy: Max
      stabilizationWindowSeconds: 300
    scaleUp:
      policies:
      - periodSeconds: 15
        type: Percent
        value: 100
      - periodSeconds: 15
        type: Pods
        value: 4
      selectPolicy: Max
      stabilizationWindowSeconds: 0
```

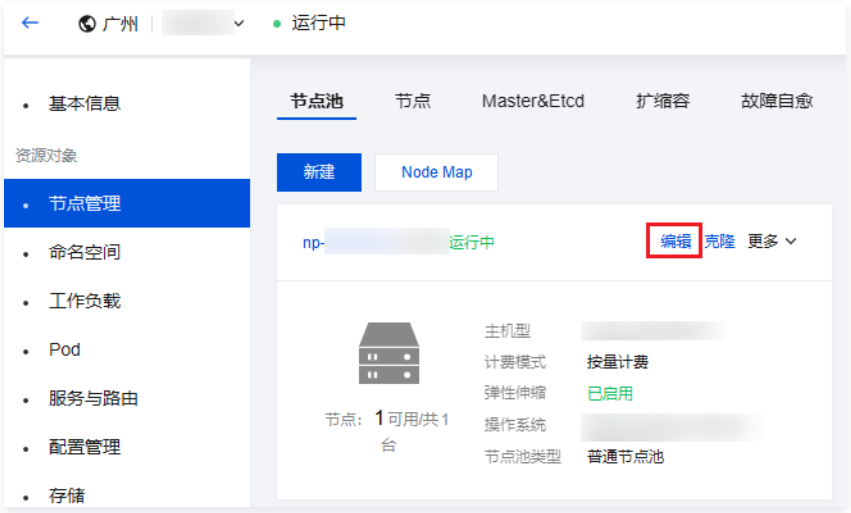
注意：

由于 GPU 资源通常比较紧张，缩容后可能无法重新获取。如果不希望触发缩容操作，可以通过以下代码给 HPA 配置禁止缩容：

```
behavior:
  scaleDown:
    selectPolicy: Disabled
```

如果使用**原生节点**或**普通节点**，还需要对节点池启动弹性伸缩，否则 GPU Pod 扩容后将因缺乏可用 GPU 节点导致 Pod 一直处于 Pending 状态。节点池启用弹性伸缩的方法如下：

1. 登录 容器服务控制台，在集群管理页面，选择集群 ID，进入集群的基本信息页面。
2. 选择左侧菜单栏中的**节点管理**，在**节点池**页面选择节点池右侧的**编辑**。本文以普通节点池为例。



3. 在调整节点池配置中，勾选开启弹性伸缩，并设置节点数量范围。

4. 单击**确定**。

步骤7：部署 OpenWebUI

使用 Deployment 部署 OpenWebUI，并定义 Service 方便后续对外暴露访问。后端 API 可以由 vLLM、SGLang 和 Ollama 提供，以下是各种场景下的 OpenWebUI 部署示例：

vLLM 后端

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webui
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      labels:
        app: webui
    spec:
      containers:
      - name: webui
        image: imroc/open-webui:main # docker hub 中的 mirror 镜像，长期自动同步，可放心使用
        env:
        - name: OPENAI_API_BASE_URL
          value: http://vllm-api:8000/v1 # vllm 的地址
        - name: ENABLE_OLLAMA_API # 禁用 Ollama API，只保留 OpenAI API
          value: "False"
        tty: true
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: "500m"
            memory: "500Mi"
          limits:
            cpu: "1000m"
            memory: "1Gi"
        volumeMounts:
        - name: webui-volume
          mountPath: /app/backend/data
```

```
      volumes:
      - name: webui-volume
        persistentVolumeClaim:
          claimName: webui

---
apiVersion: v1
kind: Service
metadata:
  name: webui
  labels:
    app: webui
spec:
  type: ClusterIP
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: webui
```

SGLang 后端

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webui
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      labels:
        app: webui
    spec:
      containers:
      - name: webui
        image: imroc/open-webui:main # docker hub 中的 mirror 镜像，长期自动同步，可放心使用
        env:
        - name: OPENAI_API_BASE_URL
          value: http://sglang:30000/v1 # sglang 的地址
        - name: ENABLE_OLLAMA_API # 禁用 Ollama API，只保留 OpenAI API
          value: "False"
        tty: true
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: "500m"
            memory: "500Mi"
          limits:
            cpu: "1000m"
            memory: "1Gi"
```

```
        volumeMounts:
        - name: webui-volume
          mountPath: /app/backend/data
      volumes:
      - name: webui-volume
        persistentVolumeClaim:
          claimName: webui

---
apiVersion: v1
kind: Service
metadata:
  name: webui
  labels:
    app: webui
spec:
  type: ClusterIP
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: webui
```

Ollama 后端

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webui
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      labels:
        app: webui
    spec:
      containers:
      - name: webui
        image: imroc/open-webui:main # docker hub 中的 mirror 镜像，长期自动同步，可放心使用
        env:
        - name: OLLAMA_BASE_URL
          value: http://ollama:11434 # ollama 的地址
        - name: ENABLE_OPENAI_API # 禁用 OpenAI API，只保留 Ollama API
          value: "False"
        tty: true
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: "500m"
            memory: "500Mi"
          limits:
```

```
            cpu: "1000m"
            memory: "1Gi"
        volumeMounts:
        - name: webui-volume
          mountPath: /app/backend/data
      volumes:
      - name: webui-volume
        persistentVolumeClaim:
          claimName: webui

---
apiVersion: v1
kind: Service
metadata:
  name: webui
  labels:
    app: webui
spec:
  type: ClusterIP
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: webui
```

说明：
OpenWebUI 的数据存储在 `/app/backend/data` 目录（如账号密码、聊天历史等数据），本文将 PVC 挂载到该路径。

步骤8：暴露 OpenWebUI 并与模型对话

本地测试

如果只是进行本地测试，可以使用 `kubectl port-forward` 命令来暴露服务：

说明：
前提是本地能够使用 kubectl 连接集群，参考连接集群。

```
kubectl port-forward service/webui 8080:8080
```

然后在浏览器中访问 `http://127.0.0.1:8080` 即可。

通过 Ingress 或 Gateway API 暴露服务

您还可以通过 Ingress 或 Gateway API 来暴露服务。以下是相关示例：

Gateway API

注意：
使用 Gateway API 需要确保您的集群中装有 Gateway API 的实现，如 TKE 应用市场中的 EnvoyGateway。具体 Gateway API 用法请参见 Gateway API 官方文档。

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
```

```
metadata:
  name: ai
spec:
  parentRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    namespace: envoy-gateway-system
    name: ai-gateway
  hostnames:
  - "ai.your.domain"
  rules:
  - backendRefs:
    - group: ""
      kind: Service
      name: webui
      port: 8080
```

说明：

1. `parentRefs` 引用定义好的 `Gateway`（通常一个 Gateway 对应一个 CLB）。
2. `hostnames` 替换为您自己的域名，确保域名能正常解析到 Gateway 对应的 CLB 地址。
3. `backendRefs` 指定 OpenWebUI 的 Service。

Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webui
spec:
  rules:
  - host: "ai.your.domain"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: webui
            port:
              number: 8080
```

说明：

1. `host` 字段需填写您的自定义域名，确保域名能正常解析到 Ingress 对应的 CLB 地址。
2. `backend.service` 则需指定为 OpenWebUI 的 Service。

完成配置后，在浏览器中访问相应的地址即可进入 OpenWebUI 页面。

首次登录

首次进入 OpenWebUI 会提示创建管理员账号密码，创建完毕后即可登录，然后默认会加载之前部署的大语言模型进行对话。

常见问题

当您在 TKE 上部署 AI 大模型遇到问题时，请参见 部署大模型常见问题 进行排查。

部署大模型常见问题

最近更新时间：2025-03-26 11:18:42

CUDA、GPU 驱动、PyTorch 与大模型兼容性问题

通常 Ollama 和 vLLM 官方的 latest 容器镜像中的 CUDA 版本能兼容大部分 GPU 卡和驱动，但能否成功运行大模型，和 CUDA、GPU卡及其驱动、PyTorch（vLLM）以及大模型本身都可能有关系，很难枚举所有情况。尤其是 vLLM，并非所有大模型都能支持，并且它依赖于 PyTorch，而不同 PyTorch 版本能兼容的 CUDA 版本也不一样，不同 CUDA 版本能兼容的 GPU 驱动版本也不一样。
如果在 vLLM 启动或运行过程中出现以下错误，建议首先检查各种版本信息，确认是否兼容。若发现问题，可尝试更换 GPU 卡或调整 CUDA 版本（GPU 驱动通常是自动安装的，一般无法改变）。下文将介绍如何指定最佳的 CUDA 版本以解决此类问题。

报错1

```
Traceback (most recent call last):
  File "/usr/local/bin/vllm", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 204, in main
    args.dispatch_function(args)
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 44, in serve
    uvloop.run(run_server(args))
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 109, in run
    return __asyncio.run(
           ^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 195, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 118, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "uvloop/loop.pyx", line 1518, in uvloop.loop.Loop.run_until_complete
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 61, in wrapper
    return await main
           ^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 875, in
run_server
    async with build_async_engine_client(args) as engine_client:
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 136, in
build_async_engine_client
    async with build_async_engine_client_from_engine_args(
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 230, in
build_async_engine_client_from_engine_args
    raise RuntimeError(
RuntimeError: Engine process failed to start. See stack trace for the root cause.
```

报错2

```
ERROR 02-07 02:51:31 client.py:300] RuntimeError('Engine process (pid 20) died.')
ERROR 02-07 02:51:31 client.py:300] NoneType: None
ERROR 02-07 02:51:34 serving_chat.py:661] Error in chat completion stream generator.
ERROR 02-07 02:51:34 serving_chat.py:661] Traceback (most recent call last):
ERROR 02-07 02:51:34 serving_chat.py:661]   File "/usr/local/lib/python3.12/dist-
packages/vllm/entrypoints/openai/serving_chat.py", line 359, in chat_completion_stream_generator
ERROR 02-07 02:51:34 serving_chat.py:661]     async for res in result_generator:
ERROR 02-07 02:51:34 serving_chat.py:661]   File "/usr/local/lib/python3.12/dist-
packages/vllm/engine/multiprocessing/client.py", line 658, in _process_request
ERROR 02-07 02:51:34 serving_chat.py:661]     raise request_output
ERROR 02-07 02:51:34 serving_chat.py:661] vllm.engine.multiprocessing.MQEngineDeadError: Engine loop
is not running. Inspect the stacktrace to find the original error: RuntimeError('Engine process (pid
20) died.').
CRITICAL 02-07 02:51:34 launcher.py:101] MQLLMEngine is already dead, terminating server process
INFO:     Shutting down
INFO:     Waiting for application shutdown.
INFO:     Application shutdown complete.
INFO:     Finished server process [1]
```

报错3

```
RuntimeError: The NVIDIA driver on your system is too old (found version 11080). Please update your
GPU driver by downloading and installing a new version from the URL:
http://www.nvidia.com/Download/index.aspx Alternatively, go to: https://pytorch.org to install a
PyTorch version that has been compiled with your version of the CUDA driver.
Traceback (most recent call last):
  File "/usr/local/bin/vllm", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 204, in main
    args.dispatch_function(args)
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 44, in serve
    uvloop.run(run_server(args))
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 109, in run
    return __asyncio.run(
           ^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 195, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 118, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "uvloop/loop.pyx", line 1518, in uvloop.loop.Loop.run_until_complete
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 61, in wrapper
    return await main
           ^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 875, in
run_server
    async with build_async_engine_client(args) as engine_client:
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
```

```
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 136, in
build_async_engine_client
    async with build_async_engine_client_from_engine_args(
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 230, in
build_async_engine_client_from_engine_args
    raise RuntimeError(
RuntimeError: Engine process failed to start. See stack trace for the root cause.
```

报错4

```
ERROR 02-06 23:41:11 engine.py:389] RuntimeError: CUDA error: no kernel image is available for
execution on the device
ERROR 02-06 23:41:11 engine.py:389] CUDA kernel errors might be asynchronously reported at some other
API call, so the stacktrace below might be incorrect.
ERROR 02-06 23:41:11 engine.py:389] For debugging consider passing CUDA_LAUNCH_BLOCKING=1
ERROR 02-06 23:41:11 engine.py:389] Compile with `TORCH_USE_CUDA_DSA` to enable device-side
assertions.
ERROR 02-06 23:41:11 engine.py:389]
Traceback (most recent call last):
  File "/usr/lib/python3.12/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "/usr/lib/python3.12/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 391, in
run_mp_engine
    raise e
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 380, in
run_mp_engine
    engine = MQLLMEngine.from_engine_args(engine_args=engine_args,
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 123, in
from_engine_args
    return cls(ipc_path=ipc_path,
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 75, in
__init__
    self.engine = LLMEngine(*args, **kwargs)
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/llm_engine.py", line 276, in __init__
    self._initialize_kv_caches()
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/llm_engine.py", line 416, in
_initialize_kv_caches
    self.model_executor.determine_num_available_blocks())
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/executor/executor_base.py", line 101, in
determine_num_available_blocks
    results = self.collective_rpc("determine_num_available_blocks")
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/executor/uniproc_executor.py", line 51, in
collective_rpc
    answer = run_method(self.driver_worker, method, args, kwargs)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
  File "/usr/local/lib/python3.12/dist-packages/vllm/utils.py", line 2220, in run_method
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/utils/_contextlib.py", line 116, in
decorate_context
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/worker/worker.py", line 229, in
determine_num_available_blocks
    self.model_runner.profile_run()
  File "/usr/local/lib/python3.12/dist-packages/torch/utils/_contextlib.py", line 116, in
decorate_context
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/worker/model_runner.py", line 1235, in
profile_run
    self._dummy_run(max_num_batched_tokens, max_num_seqs)
  File "/usr/local/lib/python3.12/dist-packages/vllm/worker/model_runner.py", line 1346, in
_dummy_run
    self.execute_model(model_input, kv_caches, intermediate_tensors)
  File "/usr/local/lib/python3.12/dist-packages/torch/utils/_contextlib.py", line 116, in
decorate_context
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/worker/model_runner.py", line 1719, in
execute_model
    hidden_or_intermediate_states = model_executable(
                                    ^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1736, in
_wrapped_call_impl
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1747, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/model_executor/models/qwen2.py", line 486, in
forward
    hidden_states = self.model(input_ids, positions, kv_caches,
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/compilation/decorators.py", line 172, in
__call__
    return self.forward(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/model_executor/models/qwen2.py", line 348, in
forward
    hidden_states, residual = layer(
                              ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1736, in
_wrapped_call_impl
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1747, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/model_executor/models/qwen2.py", line 247, in
forward
    hidden_states = self.self_attn(
                    ^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1736, in
_wrapped_call_impl
```

```
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1747, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/model_executor/models/qwen2.py", line 176, in
forward
    qkv, _ = self.qkv_proj(hidden_states)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1736, in
_wrapped_call_impl
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1747, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/model_executor/layers/linear.py", line 382, in
forward
    output_parallel = self.quant_method.apply(self, input_, bias)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/model_executor/layers/linear.py", line 142, in
apply
    return F.linear(x, layer.weight, bias)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
RuntimeError: CUDA error: no kernel image is available for execution on the device
CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below
might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.

[rank0]:[W206 23:41:12.978693132 ProcessGroupNCCL.cpp:1250] Warning: WARNING: process group has NOT
been destroyed before we destruct ProcessGroupNCCL. On normal program exit, the application should
call destroy_process_group to ensure that any pending NCCL operations have finished in this process.
In rare cases this process can exit before this point and block the progress of another member of the
process group. This constraint has always been present,  but this warning has only been added since
PyTorch 2.4 (function operator())
Traceback (most recent call last):
  File "/usr/local/bin/vllm", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 204, in main
    args.dispatch_function(args)
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 44, in serve
    uvloop.run(run_server(args))
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 109, in run
    return __asyncio.run(
           ^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 195, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 118, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "uvloop/loop.pyx", line 1518, in uvloop.loop.Loop.run_until_complete
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 61, in wrapper
    return await main
           ^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 875, in
run_server
    async with build_async_engine_client(args) as engine_client:
```

```
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 136, in
build_async_engine_client
    async with build_async_engine_client_from_engine_args(
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 230, in
build_async_engine_client_from_engine_args
    raise RuntimeError(
RuntimeError: Engine process failed to start. See stack trace for the root cause.
```

如何指定最佳的 CUDA 版本?

如果需要精确控制 CUDA 版本以达到最佳效果或避免兼容性问题，可按照以下步骤指定 CUDA 版本。

步骤1：确认 GPU 驱动和所需的 CUDA 版本

1. 确认 GPU 驱动版本：
   - 如果是普通节点或原生节点，在创建节点池时，选择机型并勾选“后台自动安装 GPU 驱动”，此时会提示 GPU 驱动版本。若无此信息，也可登录节点后执行 `nvidia-smi` 命令查看。
   - 如果调度到超级节点，可进入 Pod 后执行 `nvidia-smi` 命令查看 GPU 驱动版本。
2. 确认 CUDA 版本：在 NVIDIA 官网的 CUDA Toolkit and Corresponding Driver Versions 中，查询与当前 GPU 驱动版本兼容的 CUDA 版本，用于后续打包镜像时选择对应版本的基础镜像。

步骤2：编译 Ollama、vLLM 或 SGLang 镜像

Ollama 镜像

如果使用 Ollama 运行大模型，按照以下方法编译指定 CUDA 版本的 Ollama 镜像。

1. 准备 Dockerfile：

> **说明：**
> 基础镜像使用 `nvidia/cuda`，具体使用的 tag 可根据 步骤1 确定的 CUDA 版本选择。完整 tag 列表请参见 nvidia/cuda 。

```
FROM nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu22.04

RUN apt update -y && apt install -y curl

RUN curl -fsSL https://ollama.com/install.sh | sh
```

2. 编译并上传镜像，请注意替换为您实际使用的镜像名称。

```
docker build -t ccr.ccs.tencentyun.com/imroc/ollama:cuda11.8-ubuntu22.04 .
docker push ccr.ccs.tencentyun.com/imroc/ollama:cuda11.8-ubuntu22.04
```

vLLM 镜像

如果使用 vLLM 运行大模型，按照以下方法编译指定 CUDA 版本的 vLLM 镜像。

1. 克隆 vLLM 仓库：

```
git clone --depth=1 https://github.com/vllm-project/vllm.git
```

2. 指定 CUDA 版本并编译上传，请通过 `CUDA_VERSION` 参数指定 `CUDA` 版本，并注意替换为您实际使用的镜像名称。

```
cd vllm
docker build --build-arg CUDA_VERSION=12.4.1 -t ccr.ccs.tencentyun.com/imroc/vllm-openai:cuda-12.4.1 .
docker push ccr.ccs.tencentyun.com/imroc/vllm-openai:cuda-12.4.1
```

> **注意：**
> 该方法仅适用于 CUDA 版本的微调，避免跨越大版本。例如官方 Dockerfile 中使用的 `CUDA_VERSION` 是 12.x，那么指定的 `CUDA_VERSION` 就不要低于12，以免引发 vLLM、PyTorch 和 CUDA 之间的兼容性问题。
> 如需编译更低版本的 CUDA，建议参考官方文档，通过 `pip` 安装预编译的低版本 vLLM 二进制文件，并编写相应的 Dockerfile 编译镜像。

SGLang 镜像

SGLang 官方镜像提供了各个 CUDA 版本，可直接根据需要修改镜像 tag 即可，可选 tag 列表请参见 lmsysorg/sglang 。若未找到合适版本，可参考以下方法自行编译：

1. 克隆 SGLang 仓库：

```
git clone --depth=1 https://github.com/sgl-project/sglang.git
```

2. 指定 CUDA 版本并编译上传，请通过 `CUDA_VERSION` 参数指定 `CUDA` 版本，并注意替换为您实际使用的镜像名称。

```
cd sglang/docker
docker build --build-arg CUDA_VERSION=12.4.1 -t ccr.ccs.tencentyun.com/imroc/sglang:cuda-12.4.1 .
docker push ccr.ccs.tencentyun.com/imroc/sglang:cuda-12.4.1
```

步骤3: 替换镜像

在部署 Ollama、vLLM 或 SGLang 的 Deployment 中，将镜像替换为您指定的 CUDA 版本编译上传的镜像名称，即可完成指定最佳的 CUDA 版本的操作。

模型为何下载失败?

模型下载失败的常见原因是未开启公网访问权限。以下是针对不同节点类型的公网开通方法：

- 如果使用普通节点或原生节点，可以在创建节点池时，指定公网带宽：

- 如果使用超级节点，Pod 默认没有公网，可以使用 NAT 网关来访问外网，详情请参见 通过 NAT 网关访问外网 。此方法同样适用于普通节点和原生节点。

如何使用超过 2T 的系统盘?

如果出于成本和性能的权衡考虑，或者在测试阶段为了降低复杂度而暂时不引入 CFS，希望直接使用本地系统盘存储大模型，而大模型占用的空间较大，希望使用超过 2T 的系统盘，则需要操作系统支持才可以，只有名称中包含 `UEFI` 字样的系统镜像才能支持超过 2T 的系统盘，默认不可用，如有需要可 提交工单 开通使用。

如何实现多卡并行?

Ollama、vLLM 和 SGLang 默认将模型部署到单张 GPU 卡上，如果是多人使用、并发请求，或者模型太大，可以配置 Ollama 和 vLLM，将模型部署到多张 GPU 卡上并行计算来提升推理速度和吞吐量。
首先在定义 Ollama 或 vLLM 的 Deployment 时，需声明 GPU 的数量大于 1，示例：

```
resources:
  requests:
    nvidia.com/gpu: "2"
  limits:
    nvidia.com/gpu: "2"
```

对于 Ollama，指定环境变量 `OLLAMA_SCHED_SPREAD` 为 1 表示将模型部署到所有 GPU 卡上，示例：

```
env:
- name: OLLAMA_SCHED_SPREAD # 多卡部署
  value: "1"
```

对于 vLLM，则需显式指定 `--tensor-parallel-size` 参数，表示将模型部署到多少张 GPU 卡上，示例：

```
command:
- bash
- -c
- |
  set -ex
  exec vllm serve /data/DeepSeek-R1-Distill-Qwen-7B \
    --served-model-name DeepSeek-R1-Distill-Qwen-7B \
    --host 0.0.0.0 --port 8000 \
    --trust-remote-code \
    --enable-chunked-prefill \
    --max_num_batched_tokens 1024 \
    --max_model_len 1024 \
    --tensor-parallel-size 2 # 指定 N 张卡并行，与 requests 中指定的 GPU 数量一致
```

对于 SGLang，与 vLLM 类似，显式指定 `--tp` 参数，表示将模型部署到多少张卡上 GPU 卡上，示例：

```
command:
- bash
- -c
- |
  set -x
  exec python3 -m sglang.launch_server \
    --host 0.0.0.0 \
    --port 30000 \
    --model-path /data/$LLM_MODEL \
    --tp 4
```

如何实现多机分布式部署?

上文所述多卡部署仅限单台机器内的多卡，如果单个模型过大，而单台机器的 GPU 推理太慢，可以考虑用多机多卡分布式部署。
如何做到多机部署？如果只是简单增加副本数，各个节点的 GPU 并不能协同处理同一个任务，只能提升并发量，不能提升单个任务的推理速度。以下是实现多机分布式部署的思路和具体方案，请结合本文给出的示例 YAML 并进行相关修改。

- vLLM 官方支持通过 Ray 实现多机分布式部署，请参见 Running vLLM on multiple nodes 和 Deploy Distributed Inference Service with vLLM and LWS on GPUs。
- SGLang 官方支持多机分布式部署，请参见 Run Multi-Node Inference。
- Ollama 官方不支持多机分布式部署，但 llama.cpp 提供了一些支持，请参见 issue Llama.cpp now supports distributed inference across multiple machines.（实现门槛较高）。

vLLM 多机部署

对于 vLLM 来说，在 Kubernetes 环境中推荐使用 lws 来实现多机分布式部署，部署示例如下：

1. 按照 lws 官方文档 安装 lws 到集群。请注意，默认镜像 `registry.k8s.io/lws/lws` 可能无法下载，需修改 Deployment 的镜像地址为 `docker.io/k8smirror/lws`，该镜像为 lws 在 DockerHub 上的 mirror 镜像，长期自动同步，可放心使用（TKE 环境可直接拉取 DockerHub 的镜像）。
2. 编写 LeaderWorkerSet 的 YAML 文件并将其部署到集群中：

> **说明：**
> 这里假设每台 GPU 节点至少有 2 张 GPU 算卡，每个 Pod 使用 2 张卡，leader + worker 一共 2 个 Pod。

```
apiVersion: leaderworkerset.x-k8s.io/v1
kind: LeaderWorkerSet
metadata:
  name: vllm
spec:
  replicas: 1
  leaderWorkerTemplate:
    size: 2
    restartPolicy: RecreateGroupOnPodRestart
    leaderTemplate:
      metadata:
        labels:
          role: leader
      spec:
        containers:
        - name: vllm-leader
          image: ccr.ccs.tencentyun.com/imroc/vllm-lws:latest
          env:
          - name: RAY_CLUSTER_SIZE
            valueFrom:
              fieldRef:
                fieldPath: metadata.annotations['leaderworkerset.sigs.k8s.io/size']
          command:
          - sh
          - -c
          - |
            bash /vllm-workspace/examples/online_serving/multi-node-serving.sh leader --
ray_cluster_size=$RAY_CLUSTER_SIZE
            python3 -m vllm.entrypoints.openai.api_server \
              --port 8000 \
              --model /data/DeepSeek-R1-Distill-Qwen-32B \
              --served-model-name DeepSeek-R1-Distill-Qwen-32B \
              --tensor-parallel-size 2 \
              --pipeline-parallel-size 2 \
              --enforce-eager
          resources:
            limits:
              nvidia.com/gpu: "2"
          ports:
          - containerPort: 8000
          readinessProbe:
```

```
            tcpSocket:
              port: 8000
            initialDelaySeconds: 15
            periodSeconds: 10
          volumeMounts:
          - mountPath: /dev/shm
            name: dshm
          - mountPath: /data
            name: data
        volumes:
        - name: dshm
          emptyDir:
            medium: Memory
            sizeLimit: 15Gi
        - name: data
          persistentVolumeClaim:
            claimName: ai-model
    workerTemplate:
      spec:
        containers:
        - name: vllm-worker
          image: ccr.ccs.tencentyun.com/imroc/vllm-lws:latest
          command:
          - sh
          - -c
          - "bash /vllm-workspace/examples/online_serving/multi-node-serving.sh worker --
ray_address=$(LWS_LEADER_ADDRESS)"
          resources:
            limits:
              nvidia.com/gpu: "2"
          volumeMounts:
          - mountPath: /dev/shm
            name: dshm
          - mountPath: /data
            name: data
        volumes:
        - name: dshm
          emptyDir:
            medium: Memory
            sizeLimit: 15Gi
        - name: data
          persistentVolumeClaim:
            claimName: ai-model
---

apiVersion: v1
kind: Service
metadata:
  name: vllm-api
spec:
  ports:
  - name: http
    port: 8000
    protocol: TCP
    targetPort: 8000
  selector:
    leaderworkerset.sigs.k8s.io/name: vllm
    role: leader
  type: ClusterIP
```

说明：

- `nvidia.com/gpu` 和 `--tensor-parallel-size` 指定每台节点有多少张 GPU 卡。
- `--pipeline-parallel-size` 指定有多少台节点。
- `--model` 指定模型文件在容器内的路径。
- `--served-model-name` 指定模型名称。

3. Pod 成功跑起来后进入 leader Pod：

```
kubectl exec -it vllm-0 -- bash
```

4. 测试 API：

```
curl -v http://127.0.0.1:8000/v1/completions -H "Content-Type: application/json" -d '{
      "model": "DeepSeek-R1-Distill-Qwen-32B",
      "prompt": "你是谁?",
      "max_tokens": 100,
      "temperature": 0
    }'
```

如果部署了 OpenWebUI，确保 `OPENAI_API_BASE_URL` 指向上面示例 YAML 中 Service 的地址，如 `http://vllm-api:8000/v1`。

SGLang 多机部署

对于 SGLang 来说，官方并未提供在 Kubernetes 上多机部署的方案和实例，但可以参考其官方示例 Example: Serving with two H200*8 nodes and docker，将其转化为 StatefulSet 和 LeaderWorkerSet 方式进行部署，推荐使用 LeaderWorkerSet 部署（需安装 lws 组件）。
以下是 2 个 4 卡 GPU 节点组成的 GPU 集群的示例：

StatefulSet 方式部署

用 Statefulset 部署无需引入 lws 依赖，但扩容 GPU 集群时较麻烦，需要手动创建新的 StatefulSet。
请根据实际情况修改：

- `nvidia.com/gpu` 为每个节点的 GPU 卡数。
- `replicas` 为 GPU 集群的总节点数，需 `REPLICAS` 环境变量的值保持一致。
- `LLM_MODEL` 环境变量为模型名称，与前面下载 Job 中指定的名称一致。
- `TOTAL_GPU` 环境变量为总 GPU 卡数，等于每个节点的 GPU 数量乘以节点数。
- `STATEFULSET_NAME` 环境变量的值 StatefulSet 实际名称保持一致。
- `SERVICE_NAME` 环境变量的值与 StatefulSet 中指定的 `serviceName`，以及实际的 Service 的名称保持一致。
- 如果部署了 OpenWebUI，确保 `OPENAI_API_BASE_URL` 指向第一个副本的地址（leader），如 `http://sglang-0.sglang:3000/v1`。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sglang
spec:
  selector:
    matchLabels:
      app: sglang
  serviceName: sglang
  replicas: 2
  template:
    metadata:
      labels:
        app: sglang
    spec:
      containers:
      - name: sglang
```

```
        image: lmsysorg/sglang:latest
        env:
        - name: LLM_MODEL
          value: deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
        - name: TOTAL_GPU
          value: "8"
        - name: REPLICAS
          value: "2"
        - name: STATEFULSET_NAME
          value: "sglang"
        - name: SERVICE_NAME
          value: "sglang"
        - name: ORDINAL_NUMBER
          valueFrom:
            fieldRef:
              fieldPath: metadata.labels['apps.kubernetes.io/pod-index']
        ports:
        - containerPort: 30000
        resources:
          limits:
            nvidia.com/gpu: "4"
        command:
        - bash
        - -c
        - |
          if [[ "$ORDINAL_NUMBER" == "0" ]]; then
            python3 -m sglang.launch_server --model-path /data/$LLM_MODEL --tp $TOTAL_GPU --dist-
init-addr $STATEFULSET_NAME-0.$SERVICE_NAME:5000 --nnodes $REPLICAS --node-rank $ORDINAL_NUMBER --
trust-remote-code --host 0.0.0.0 --port 30000
          else
            python3 -m sglang.launch_server --model-path /data/$LLM_MODEL --tp $TOTAL_GPU --dist-
init-addr $STATEFULSET_NAME-0.$SERVICE_NAME:5000 --nnodes $REPLICAS --node-rank $ORDINAL_NUMBER --
trust-remote-code
          fi
        volumeMounts:
        - name: data
          mountPath: /data
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: 40Gi
---

apiVersion: v1
kind: Service
metadata:
  name: sglang
spec:
  selector:
    app: sglang
  type: ClusterIP
  clusterIP: None
  ports:
  - name: api
```

```
    protocol: TCP
    port: 30000
    targetPort: 30000
```

LeaderWorkerSet 方式部署

使用 LeaderWorkerSet 部署的前提需在集群中安装 lws 组件，可按照 lws 官方文档 安装 lws 到集群，请注意，默认镜像 `registry.k8s.io/lws/lws` 可能无法下载，需修改 Deployment 的镜像地址为 `docker.io/k8smirror/lws`，该镜像为 lws 在 DockerHub 上的 mirror 镜像，长期自动同步，可放心使用（TKE 环境可直接拉取 DockerHub 的镜像）。
根据实际情况修改 YAML 中的一些配置：

- `nvidia.com/gpu` 为每个节点的 GPU 卡数。
- `--tp` 为总 GPU 卡数，等于每个节点的 GPU 数量乘以节点数。
- `--model-path` 模型加载的目录。

```
apiVersion: leaderworkerset.x-k8s.io/v1
kind: LeaderWorkerSet
metadata:
  name: sglang
spec:
  replicas: 1
  leaderWorkerTemplate:
    size: 2
    restartPolicy: RecreateGroupOnPodRestart
    leaderTemplate:
      metadata:
        labels:
          role: leader
      spec:
        containers:
        - name: sglang-leader
          image: lmsysorg/sglang:latest
          command:
          - bash
          - -c
          - |
            set -x
            exec python3 -m sglang.launch_server \
              --model-path /data/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B \
              --nnodes $LWS_GROUP_SIZE \
              --tp 8 \
              --node-rank 0 \
              --dist-init-addr $(LWS_LEADER_ADDRESS):5000 \
              --trust-remote-code \
              --host 0.0.0.0 \
              --port 30000
          resources:
            limits:
              nvidia.com/gpu: "4"
          ports:
          - containerPort: 30000
          volumeMounts:
          - mountPath: /dev/shm
            name: dshm
          - mountPath: /data
            name: data
        volumes:
```

```
      - name: dshm
        emptyDir:
          medium: Memory
          sizeLimit: 40Gi
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
  workerTemplate:
    spec:
      containers:
      - name: sglang-worker
        image: lmsysorg/sglang:latest
        env:
        - name: ORDINAL_NUMBER
          valueFrom:
            fieldRef:
              fieldPath: metadata.labels['apps.kubernetes.io/pod-index']
        command:
        - bash
        - -c
        - |
          set -x
          exec python3 -m sglang.launch_server \
            --model-path /data/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B \
            --nnodes $LWS_GROUP_SIZE \
            --tp 8 \
            --node-rank $ORDINAL_NUMBER \
            --dist-init-addr $(LWS_LEADER_ADDRESS):5000 \
            --trust-remote-code
        resources:
          limits:
            nvidia.com/gpu: "4"
        volumeMounts:
        - mountPath: /dev/shm
          name: dshm
        - mountPath: /data
          name: data
      volumes:
      - name: dshm
        emptyDir:
          medium: Memory
          sizeLimit: 40Gi
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
```

多机部署如何扩容 GPU?

分布式多机部署一般要求每台节点 GPU 数量一致，且要事先规划好总节点数量，然后根据这些信息配置启动参数（GPU 并行数量，总节点数量），如果要扩容，只能增加新的 GPU 集群，同时让请求负载均衡到多个 GPU 集群。以下是具体的扩容思路和实现方法。

- 新增 GPU 集群：可以利用 lws 的能力，调整 replicas 的值，replicas + 1 表示新增一个 leader + worker 的集群，即扩容新的 GPU 集群。
- 多 GPU 集群负载均衡：可以新建一个 Service 选中多个不同 GPU 集群的 leader Pod 来实现，示例如下：

> ⚠ **注意：**
>
> - `leaderworkerset.sigs.k8s.io/name` 指定 lws 的名称。
> - 所有 GPU 集群的 leader Pod 的 index 固定为 0，可以通过 `apps.kubernetes.io/pod-index: "0"` 这个 label 来选中。

- 涉及 API 地址配置的地方（如 OpenWebUI），指向这个新 Service 的地址（如 `http://vllm-leader:8000/v1`）。

```
apiVersion: v1
kind: Service
metadata:
  name: vllm-leader
spec:
  type: ClusterIP
  selector:
    leaderworkerset.sigs.k8s.io/name: vllm
    apps.kubernetes.io/pod-index: "0"
  ports:
  - name: api
    port: 8000
    targetPort: 8000
```

vLLM 报错 ValueError: invalid literal for int() with base 10: 'tcp://xxx.xx.xx.xx:8000'

vLLM 启动时报错：

```
ERROR 02-06 18:29:55 engine.py:389] ValueError: invalid literal for int() with base 10:
'tcp://172.16.168.90:8000'
Traceback (most recent call last):
  File "/usr/lib/python3.12/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "/usr/lib/python3.12/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 391, in
run_mp_engine
    raise e
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 380, in
run_mp_engine
    engine = MQLLMEngine.from_engine_args(engine_args=engine_args,
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 123, in
from_engine_args
    return cls(ipc_path=ipc_path,
           ^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/multiprocessing/engine.py", line 75, in
__init__
    self.engine = LLMEngine(*args, **kwargs)
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/engine/llm_engine.py", line 273, in __init__
    self.model_executor = executor_class(vllm_config=vllm_config, )
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/executor/executor_base.py", line 51, in __init__
    self._init_executor()
  File "/usr/local/lib/python3.12/dist-packages/vllm/executor/uniproc_executor.py", line 29, in
_init_executor
    get_ip(), get_open_port())
              ^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/utils.py", line 506, in get_open_port
    port = envs.VLLM_PORT
           ^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/envs.py", line 583, in __getattr__
    return environment_variables[name]()
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/envs.py", line 188, in <lambda>
```

```
    lambda: int(os.getenv('VLLM_PORT', '0'))
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- 原因：vLLM 会解析 `VLLM_PORT` 这个环境变量，并期望其值为一个数字，但实际获取到的值并非数字，从而导致报错。此处并未定义 `VLLM_PORT` 这个环境变量，这个环境变量是 Kubernetes 根据 Service 自动生成注入到 Pod 中的。
- 解决办法：不要将 vLLM 的 Service 名称定义为 `vllm`，而是改为其他名称。

vLLM 报错 ValueError: Bfloat16 is only supported on GPUs with compute capability of at least 8.0.

vLLM 启动时报错：

```
ValueError: Bfloat16 is only supported on GPUs with compute capability of at least 8.0. Your Tesla V100-
SXM2-32GB GPU has compute capability 7.0. You can use float16 instead by explicitly setting the`dtype`
flag in CLI, for example: --dtype=half.
Traceback (most recent call last):
  File "/usr/local/bin/vllm", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 204, in main
    args.dispatch_function(args)
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 44, in serve
    uvloop.run(run_server(args))
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 109, in run
    return __asyncio.run(
           ^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 195, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 118, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "uvloop/loop.pyx", line 1518, in uvloop.loop.Loop.run_until_complete
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 61, in wrapper
    return await main
           ^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 875, in
run_server
    async with build_async_engine_client(args) as engine_client:
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 136, in
build_async_engine_client
    async with build_async_engine_client_from_engine_args(
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 230, in
build_async_engine_client_from_engine_args
    raise RuntimeError(
RuntimeError: Engine process failed to start. See stack trace for the root cause.
```

- 原因：如报错所提示，GPU 卡不支持指定的 `--dtype` 类型（`bfloat16`），并指定 `--dtype=half` 的建议。
- 解决办法：修改 vLLM 的 Deployment 中的启动参数，将 `--dtype` 的值指定为 `half`。

vLLM 启动报 KeyboardInterrupt: terminated 然后退出

退出前日志：

```
Loading safetensors checkpoint shards:   0% Completed | 0/2 [00:00<?, ?it/s]
Traceback (most recent call last):
  File "/usr/local/bin/vllm", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 204, in main
    args.dispatch_function(args)
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 44, in serve
    uvloop.run(run_server(args))
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 109, in run
    return __asyncio.run(
           ^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 195, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 118, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "uvloop/loop.pyx", line 1512, in uvloop.loop.Loop.run_until_complete
  File "uvloop/loop.pyx", line 1505, in uvloop.loop.Loop.run_until_complete
  File "uvloop/loop.pyx", line 1379, in uvloop.loop.Loop.run_forever
  File "uvloop/loop.pyx", line 557, in uvloop.loop.Loop._run
  File "uvloop/handles/poll.pyx", line 216, in uvloop.loop.__on_uvpoll_event
  File "uvloop/cbhandles.pyx", line 83, in uvloop.loop.Handle._run
  File "uvloop/cbhandles.pyx", line 66, in uvloop.loop.Handle._run
  File "uvloop/loop.pyx", line 399, in uvloop.loop.Loop._read_from_self
  File "uvloop/loop.pyx", line 404, in uvloop.loop.Loop._invoke_signals
  File "uvloop/loop.pyx", line 379, in uvloop.loop.Loop._ceval_process_signals
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 871, in
signal_handler
    raise KeyboardInterrupt("terminated")
KeyboardInterrupt: terminated
```

- 原因：vLLM 启动慢，存活检查失败到阈值，主进程收到 SIGTERM 信号后退出。
- 解决办法：延长 `livenessProbe` 的 `initialDelaySeconds`，避免因 vLLM 启动慢被终止，或者去掉 `livenessProbe`。

vLLM 报错: max seq len is larger than the maximum number of tokens

报错日志：

```
ValueError: The model's max seq len (131072) is larger than the maximum number of tokens that can be
stored in KV cache (93760). Try increasing `gpu_memory_utilization` or decreasing `max_model_len` when
initializing the engine.
[rank0]:[W207 01:57:35.912382100 ProcessGroupNCCL.cpp:1250] Warning: WARNING: process group has NOT been
destroyed before we destruct ProcessGroupNCCL. On normal program exit, the application should call
destroy_process_group to ensure that any pending NCCL operations have finished in this process. In rare
cases this process can exit before this point and block the progress of another member of the process
group. This constraint has always been present,  but this warning has only been added since PyTorch 2.4
(function operator())
Traceback (most recent call last):
  File "/usr/local/bin/vllm", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 204, in main
    args.dispatch_function(args)
  File "/usr/local/lib/python3.12/dist-packages/vllm/scripts.py", line 44, in serve
    uvloop.run(run_server(args))
```

```
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 109, in run
    return __asyncio.run(
           ^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 195, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/asyncio/runners.py", line 118, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "uvloop/loop.pyx", line 1518, in uvloop.loop.Loop.run_until_complete
  File "/usr/local/lib/python3.12/dist-packages/uvloop/__init__.py", line 61, in wrapper
    return await main
           ^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 875, in
run_server
    async with build_async_engine_client(args) as engine_client:
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 136, in
build_async_engine_client
    async with build_async_engine_client_from_engine_args(
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/contextlib.py", line 210, in __aenter__
    return await anext(self.gen)
           ^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/vllm/entrypoints/openai/api_server.py", line 230, in
build_async_engine_client_from_engine_args
    raise RuntimeError(
RuntimeError: Engine process failed to start. See stack trace for the root cause.
```

解决办法：vllm 启动参数指定 `--max-model-len`，**如** `--max-model-len 1024` **。**

vLLM 或 SGLang 报错: CUDA out of memory

vLLM 报错

```
ERROR 02-07 03:25:19 engine.py:389] CUDA out of memory. Tried to allocate 150.00 MiB. GPU 0 has a
total capacity of 14.58 GiB of which 95.56 MiB is free. Process 81610 has 14.48 GiB memory in use. Of
the allocated memory 14.30 GiB is allocated by PyTorch, and 34.90 MiB is reserved by PyTorch but
unallocated. If reserved but unallocated memory is large try setting
PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True to avoid fragmentation.  See documentation for
Memory Management  (https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
ERROR 02-07 03:25:19 engine.py:389] Traceback (most recent call last):
Process SpawnProcess-1:
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/engine/multiprocessing/engine.py", line 380, in run_mp_engine
ERROR 02-07 03:25:19 engine.py:389]     engine =
MQLLMEngine.from_engine_args(engine_args=engine_args,
ERROR 02-07 03:25:19 engine.py:389]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/engine/multiprocessing/engine.py", line 123, in from_engine_args
ERROR 02-07 03:25:19 engine.py:389]     return cls(ipc_path=ipc_path,
ERROR 02-07 03:25:19 engine.py:389]            ^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/engine/multiprocessing/engine.py", line 75, in __init__
```

```
ERROR 02-07 03:25:19 engine.py:389]     self.engine = LLMEngine(*args, **kwargs)
ERROR 02-07 03:25:19 engine.py:389]                   ^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/engine/llm_engine.py", line 276, in __init__
ERROR 02-07 03:25:19 engine.py:389]     self._initialize_kv_caches()
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/engine/llm_engine.py", line 416, in _initialize_kv_caches
ERROR 02-07 03:25:19 engine.py:389]     self.model_executor.determine_num_available_blocks())
ERROR 02-07 03:25:19 engine.py:389]     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/executor/executor_base.py", line 101, in determine_num_available_blocks
ERROR 02-07 03:25:19 engine.py:389]     results =
self.collective_rpc("determine_num_available_blocks")
ERROR 02-07 03:25:19 engine.py:389]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/executor/uniproc_executor.py", line 51, in collective_rpc
ERROR 02-07 03:25:19 engine.py:389]     answer = run_method(self.driver_worker, method, args, kwargs)
ERROR 02-07 03:25:19 engine.py:389]              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-packages/vllm/utils.py",
line 2220, in run_method
ERROR 02-07 03:25:19 engine.py:389]     return func(*args, **kwargs)
ERROR 02-07 03:25:19 engine.py:389]            ^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/torch/utils/_contextlib.py", line 116, in decorate_context
ERROR 02-07 03:25:19 engine.py:389]     return func(*args, **kwargs)
ERROR 02-07 03:25:19 engine.py:389]            ^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/worker/worker.py", line 229, in determine_num_available_blocks
ERROR 02-07 03:25:19 engine.py:389]     self.model_runner.profile_run()
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/torch/utils/_contextlib.py", line 116, in decorate_context
ERROR 02-07 03:25:19 engine.py:389]     return func(*args, **kwargs)
ERROR 02-07 03:25:19 engine.py:389]            ^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/worker/model_runner.py", line 1235, in profile_run
ERROR 02-07 03:25:19 engine.py:389]     self._dummy_run(max_num_batched_tokens, max_num_seqs)
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/worker/model_runner.py", line 1346, in _dummy_run
ERROR 02-07 03:25:19 engine.py:389]     self.execute_model(model_input, kv_caches,
intermediate_tensors)
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/torch/utils/_contextlib.py", line 116, in decorate_context
ERROR 02-07 03:25:19 engine.py:389]     return func(*args, **kwargs)
ERROR 02-07 03:25:19 engine.py:389]            ^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/worker/model_runner.py", line 1775, in execute_model
ERROR 02-07 03:25:19 engine.py:389]     output: SamplerOutput = self.model.sample(
ERROR 02-07 03:25:19 engine.py:389]                             ^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/model_executor/models/qwen2.py", line 505, in sample
ERROR 02-07 03:25:19 engine.py:389]     next_tokens = self.sampler(logits, sampling_metadata)
ERROR 02-07 03:25:19 engine.py:389]                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/torch/nn/modules/module.py", line 1736, in _wrapped_call_impl
ERROR 02-07 03:25:19 engine.py:389]     return self._call_impl(*args, **kwargs)
ERROR 02-07 03:25:19 engine.py:389]            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/torch/nn/modules/module.py", line 1747, in _call_impl
ERROR 02-07 03:25:19 engine.py:389]     return forward_call(*args, **kwargs)
```

```
ERROR 02-07 03:25:19 engine.py:389]             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ERROR 02-07 03:25:19 engine.py:389]   File "/usr/local/lib/python3.12/dist-
packages/vllm/model_executor/layers/sampler.py", line 271, in forward
```

SGLang 报错

```
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 540.00 MiB. GPU 0 has a total capacity
of 14.76 GiB of which 298.75 MiB is free. Process 63729 has 14.46 GiB memory in use. Of the allocated
memory 14.35 GiB is allocated by PyTorch, and 1.52 MiB is reserved by PyTorch but unallocated. If
reserved but unallocated memory is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True
to avoid fragmentation.  See documentation for Memory Management
(https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
```

- 原因：GPU 卡显存不够。
- 解决方案：换显存更大的 GPU 卡，或使用多机部署组成 GPU 集群。

SGLang 报错：SGLang only supports sm75 and above.

报错日志：

```
[2025-02-12 02:56:48 TP0] Compute capability below sm80. Use float16 due to lack of bfloat16 support.
[2025-02-12 02:56:48 TP0] Scheduler hit an exception: Traceback (most recent call last):
  File "/sgl-workspace/sglang/python/sglang/srt/managers/scheduler.py", line 1787, in
run_scheduler_process
    scheduler = Scheduler(server_args, port_args, gpu_id, tp_rank, dp_rank)
  File "/sgl-workspace/sglang/python/sglang/srt/managers/scheduler.py", line 240, in __init__
    self.tp_worker = TpWorkerClass(
  File "/sgl-workspace/sglang/python/sglang/srt/managers/tp_worker_overlap_thread.py", line 63, in
__init__
    self.worker = TpModelWorker(server_args, gpu_id, tp_rank, dp_rank, nccl_port)
  File "/sgl-workspace/sglang/python/sglang/srt/managers/tp_worker.py", line 68, in __init__
    self.model_runner = ModelRunner(
  File "/sgl-workspace/sglang/python/sglang/srt/model_executor/model_runner.py", line 186, in __init__
    self.load_model()
  File "/sgl-workspace/sglang/python/sglang/srt/model_executor/model_runner.py", line 293, in load_model
    raise RuntimeError("SGLang only supports sm75 and above.")
RuntimeError: SGLang only supports sm75 and above.
```

- 原因：GPU 显卡计算能力不够，提示至少计算能力要 SM7.5。
- 解决方案：换成计算能力大于等于 SM7.5 的 GPU，如 T4、A100。

GPU 数量需被注意力头数整除

在进行多机部署时，需确保模型的注意力头数能整除总 GPU 数量，否则加载模型时会报错：

vLLM 报错

```
ValueError: Total number of attention heads (32) must be divisible by tensor parallel size
```

SGLang 报错

```
[2025-02-14 02:47:45 TP0] Scheduler hit an exception: Traceback (most recent call last):
  File "/sgl-workspace/sglang/python/sglang/srt/managers/scheduler.py", line 1816, in
run_scheduler_process
    scheduler = Scheduler(server_args, port_args, gpu_id, tp_rank, dp_rank)
  File "/sgl-workspace/sglang/python/sglang/srt/managers/scheduler.py", line 240, in __init__
    self.tp_worker = TpWorkerClass(
  File "/sgl-workspace/sglang/python/sglang/srt/managers/tp_worker_overlap_thread.py", line 63, in
__init__
    self.worker = TpModelWorker(server_args, gpu_id, tp_rank, dp_rank, nccl_port)
  File "/sgl-workspace/sglang/python/sglang/srt/managers/tp_worker.py", line 68, in __init__
    self.model_runner = ModelRunner(
  File "/sgl-workspace/sglang/python/sglang/srt/model_executor/model_runner.py", line 194, in
__init__
    self.load_model()
  File "/sgl-workspace/sglang/python/sglang/srt/model_executor/model_runner.py", line 317, in
load_model
    self.model = get_model(
  File "/sgl-workspace/sglang/python/sglang/srt/model_loader/__init__.py", line 22, in get_model
    return loader.load_model(
  File "/sgl-workspace/sglang/python/sglang/srt/model_loader/loader.py", line 357, in load_model
    model = _initialize_model(
  File "/sgl-workspace/sglang/python/sglang/srt/model_loader/loader.py", line 138, in
_initialize_model
    return model_class(
  File "/sgl-workspace/sglang/python/sglang/srt/models/qwen2.py", line 332, in __init__
    self.model = Qwen2Model(config, quant_config=quant_config)
  File "/sgl-workspace/sglang/python/sglang/srt/models/qwen2.py", line 241, in __init__
    self.layers = make_layers(
  File "/sgl-workspace/sglang/python/sglang/srt/utils.py", line 313, in make_layers
    [
  File "/sgl-workspace/sglang/python/sglang/srt/utils.py", line 314, in <listcomp>
    maybe_offload_to_cpu(layer_fn(idx=idx, prefix=f"{prefix}.{idx}"))
  File "/sgl-workspace/sglang/python/sglang/srt/models/qwen2.py", line 243, in <lambda>
    lambda idx, prefix: Qwen2DecoderLayer(
  File "/sgl-workspace/sglang/python/sglang/srt/models/qwen2.py", line 180, in __init__
    self.self_attn = Qwen2Attention(
  File "/sgl-workspace/sglang/python/sglang/srt/models/qwen2.py", line 105, in __init__
    assert self.total_num_heads % tp_size == 0
AssertionError
```

模型的注意力头数等于模型文件 `config.json` 中 `num_attention_heads` 的值，例如 `DeepSeek-R1-Distill-Qwen-32B` 的注意力头数为 40：

```
root@test-7778876c87-mlc2x:/data/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B# cat config.json
{
  "architectures": [
    "Qwen2ForCausalLM"
  ],
  "attention_dropout": 0.0,
  "bos_token_id": 151643,
  "eos_token_id": 151643,
  "hidden_act": "silu",
  "hidden_size": 5120,
  "initializer_range": 0.02,
  "intermediate_size": 27648,
  "max_position_embeddings": 131072,
  "max_window_layers": 64,
  "model_type": "qwen2",
  "num_attention_heads": 40,
  "num_hidden_layers": 64,
  "num_key_value_heads": 8,
  "rms_norm_eps": 1e-05,
  "rope_theta": 1000000.0,
  "sliding_window": 131072,
  "tie_word_embeddings": false,
  "torch_dtype": "bfloat16",
  "transformers_version": "4.43.1",
  "use_cache": true,
  "use_sliding_window": false,
  "vocab_size": 152064
}
```

如果每台节点一张 GPU 卡，总共 3 个节点，由于 40/3 不能整除，因此启动时会报错。如果将节点数量调整为 2，此时 40/2 可以整除，模型能够正常启动。

在 TKE 上部署满血版 DeepSeek-R1 (SGLang)

最近更新时间：2025-03-26 11:18:42

概述

DeepSeek-R1 满血版模型参数是 671B（6710 亿参数），为了充分发挥其性能，推荐使用 SGLang 进行部署。SGLang 是性能强悍的 AI 大模型部署工具，与 DeepSeek 官方合作并专门针对 DeepSeek 进行了深度优化，也是 DeepSeek 官方推荐的部署工具。
本文将基于 SGLang 在 TKE 集群上部署满血版 DeepSeek-R1 模型，提供最佳实践的部署示例。

> **镜像说明：**
> 本文中的示例使用的镜像是 SGLang 官方提供的镜像（ lmsysorg/sglang ），tag 为 latest，建议指定 tag 到固定版本。
> 官方镜像托管在 DockerHub，且体积较大，在 TKE 环境中，默认提供免费的 DockerHub 镜像加速服务。中国大陆用户也可以直接从 DockerHub 拉取镜像，但速度可能较慢，尤其是对于较大的镜像，等待时间会更长。为提高镜像拉取速度，建议将镜像同步至 容器镜像服务 TCR，并在 YAML 文件中替换相应的镜像地址，这样可以显著加快镜像的拉取速度。

机型与部署方案

由于满血版的 DeepSeek-R1 参数量较大，需要使用较大显存且支持 FP8 的大规格 GPU 实例，目前合适的机型规格包括 HCCPNV6.96XLARGE2304（ 高性能计算集群 ）和 PNV6.32XLARGE1280 / PNV6.96XLARGE2304（ GPU 云服务器 ），推荐的部署方案是用两台该机型的节点组建 GPU 集群来运行满血 DeepSeek-R1，如果对并发和性能要求不高，也可以单台部署。
以下是这几种规格的核心参数：

规格	RDMA	CPU 核心数	内存(GB)	GPU 卡数
PNV6.32XLARGE1280	不支持	128	1280	8
PNV6.96XLARGE2304	不支持	384	2304	8
HCCPNV6.96XLARGE2304	支持 (3.2Tbps)	384	2304	8

> **注意：**
> 这些规格的实例目前处于邀测阶段，且资源紧张，需联系您的销售经理开通使用并协调资源。

选型建议

- 两台组建 GPU 集群来运行满血 DeepSeek-R1，建议选择 `HCCPNV6.96XLARGE2304` ，因为支持 RDMA，可显著提升 DeepSeek 运行性能，生成速度约为 **600~700 token/s**。
- 单台部署无需 RDMA，优先考虑 `PNV6.32XLARGE1280` 和 `PNV6.96XLARGE2304` 以节约成本，生成速度约为 **20~40 token/s**。

操作步骤

购买 GPU 服务器

测试 POC 阶段，可先在 云服务器购买页面 进行购买，按量计费。

> **说明：**
> 对于 **PNV6.32XLARGE1280** 和 **PNV6.96XLARGE2304** 的这两种规格，**架构**选择**异构计算**才能看到；对于 **HCCPNV6.96XLARGE2304** 的规格，**架构**选择**高性能计算集群**才能看到，且要先提前创建高性能计算集群，详情请参见 创建高性能计算集群 。

正式购买阶段，需通过 高性能计算平台-工作空间 购买，包年包月计费，请联系腾讯云架构师进行开通使用。

创建 TKE 集群

登录 容器服务控制台，参考步骤 创建集群，创建一个集群：

- **地域**：选择购买的 GPU 服务器所在地域。
- **集群类型**：选择 **TKE 标准集群**。
- **Kubernetes 版本**：需要大于等于1.28（多机部署依赖的 LWS 组件的要求），建议选最新版。

- VPC：选择购买的 GPU 资源所在的 VPC。

添加 GPU 节点

通过 添加已有节点 的方式将购买到的 GPU 服务器加入 TKE 集群。其中，系统镜像选择 `TencentOS Server 3.1 (TK4) UEFI | img-39ywauzd`，驱动和 CUDA 选择最新版本。

准备存储与模型文件

满血版 DeepSeek-R1 体积较大，为加快模型下载和加载速度，建议使用性能最强的存储，本文给出本地存储和 CFS-Trubo 共享存储两种方案的示例。

CFS-Turbo 共享存储

共享存储使用 CFS-Turbo 性能更好，Turbo 系列性能与规格详情参考 腾讯云文件存储官方文档，使用下面的步骤准备 CFS-Turbo 存储和下载大模型文件。

安装 CFS 插件

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧菜单栏中的**组件管理**，在组件页面单击**新建**。
3. 在新建组件管理页面中勾选 **CFSTurbo（腾讯云高性能并行文件系统）**。

4. 单击**完成**即可创建组件。

创建 CFS-Turbo 实例

1. 登录 CFS 控制台，单击**创建**来新建一个 CFS-Turbo 实例。

2. **文件系统类型**选择 Turbo 系列的：

3. **地域**选择 TKE 集群所在地域。
4. **可用区**选择 GPU 节点池所在可用区（降低时延）。
5. **网络类型**如果选**云联网网络**，需确保 TKE 集群所在 VPC 已加入该云联网中；如果选**VPC 网络**，则需选择 TKE 集群所在 VPC，子网选与 GPU 节点池在同一个可用区的子网。
6. 单击**立即创建**。

新建 StorageClass

新建一个后续使用 CFS 存储大模型的 PVC，可通过控制台或 YAML 创建。

通过控制台创建

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧菜单栏中的**存储**，在 StorageClass 页面单击**新建**。
3. 在新建存储页面，根据实际需求，创建 CFS-Turbo 类型的 StorageClass。如下图所示：

- 名称：请输入 StorageClass 名称，本文以 “cfs-turbo” 为例。
- Provisioner：选择 “文件存储CFS turbo”。
- CFS turbo：选择前面**创建 CFS-Turbo 实例**步骤中创建出来的 CFS-Turbo 的实例。

通过 YAML 创建

> ⚠ **注意：**
> - `fsid` 替换为前面步骤新建的 CFS-Turbo 实例的挂载点 ID（在实例的挂载点信息页面可查看，注意不是 cfs-xxx 的 ID）。
> - `host` 替换为前面步骤新建的 CFS-Turbo 实例的IPv4地址（同样也在挂载点信息页面可查看）。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: cfs-turbo
provisioner: com.tencent.cloud.csi.cfsturbo
reclaimPolicy: Delete
volumeBindingMode: Immediate
parameters: # 注意替换 fsid 和 host
  fsid: 564b8ef1
  host: 11.0.0.7
```

创建 PVC

创建一个使用 CFS-Turbo 的 PVC，用于存储 AI 大模型，可通过控制台或 YAML 创建。

通过控制台创建

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧菜单栏中的**存储**，在 PersistentVolumeClaim 页面单击**新建**。
3. 在新建存储页面，创建存储大模型的 PVC。如下图所示：

- 名称：请输入 PVC 名称，本文以 “ai-model” 为例。
- 命名空间：SGLang 将要被部署的命名空间。
- Provisioner：选择 “文件存储 CFS turbo”。
- 是否指定 StorageClass：选择 “指定”。
- StorageClass：选择前面新建的 StorageClass 的名称。
- 是否指定 PersistentVolume：选择 “不指定”。

通过 YAML 创建

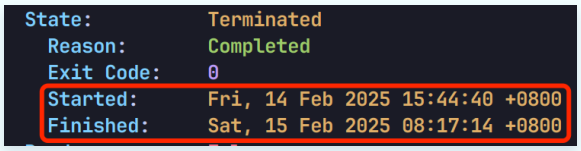
> **注意：**
> 注意替换 `storageClassName` 为新建 StorageClass 步骤中配置的名称。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ai-model
  labels:
    app: ai-model
spec:
  storageClassName: cfs-turbo
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 700Gi
```

使用 Job 下载大模型文件

创建一个 Job 用于下载大模型文件到 CFS：

> ⚠ **注意：**
> 满血版的 DeepSeek-R1 是 671B 的大模型，一共 642G，下载耗时可能较长，实测在上海下载 ModelScope 上的模型文件，100Mbps 的云服务器带宽，耗时 16 个多小时：




```
apiVersion: batch/v1
kind: Job
metadata:
  name: download-model
  labels:
    app: download-model
spec:
  template:
    metadata:
      name: sglang
      labels:
        app: download-model
    spec:
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        command:
        - modelscope
        - download
        - --local_dir=/data/model/DeepSeek-R1
        - --model=deepseek-ai/DeepSeek-R1
        volumeMounts:
        - name: data
          mountPath: /data/model
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      restartPolicy: OnFailure
```

- CFS-Turbo 的 PVC 挂载到 `/data` 目录，存储下载的模型文件。
- `--local_dir` 指定模型文件下载目录。
- `--model` 指定 ModelScope 模型库 中的模型名称，满血版的 DeepSeek-R1 模型名称为 `deepseek-ai/DeepSeek-R1` 。

本地存储

如果使用本地存储大模型，可以创建一个下载模型文件的 DaemonSet，相当于给每个节点都下发一个下载 Job：

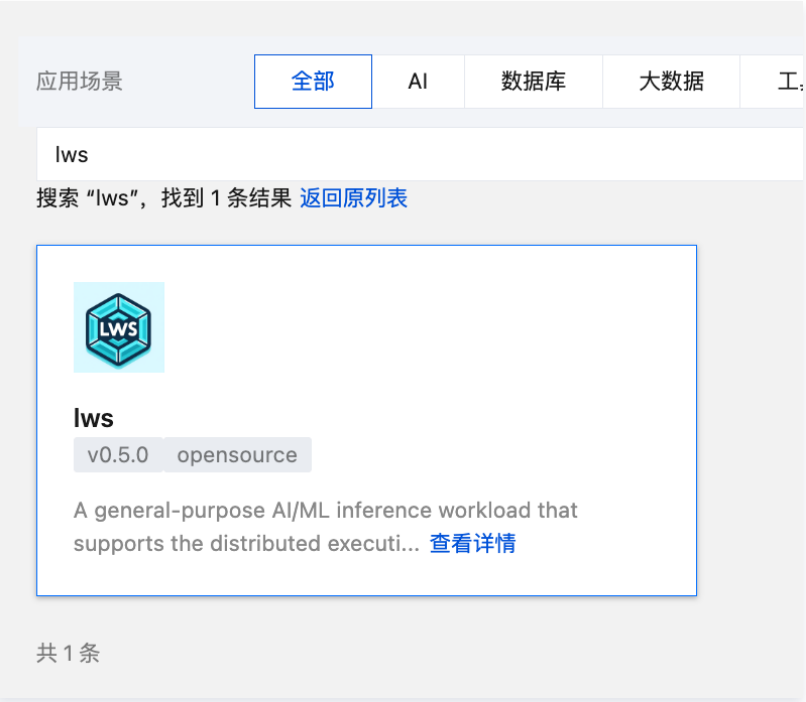
```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: download-model
  labels:
    app: download-model
spec:
  selector:
    matchLabels:
      app: download-model
```

```
  template:
    metadata:
      labels:
        app: download-model
    spec:
      restartPolicy: OnFailure # 默认 Always，改成 OnFailure 避免重复下载
      nodeSelector:
        nvidia-device-enable: "true" # 只让 GPU 节点下载
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        command:
        - modelscope
        - download
        - --local_dir=/data/model/DeepSeek-R1
        - --model=deepseek-ai/DeepSeek-R1
        volumeMounts:
        - name: model
          mountPath: /data/model
      volumes:
      - name: model
        hostPath:
          path: /data/model
          type: DirectoryOrCreate
```

安装 LWS 组件

> **注意：**
> 如果只使用单机部署的方案，无需安装 LWS 组件，可跳过此步骤。

SGLang 多机部署（GPU 集群）需借助 LWS 组件，在 TKE 应用市场 中找到 lws：



安装到集群中：

- 应用名：建议填 `lws`。
- 命名空间：建议使用 `lws-system`（新建命名空间）。

> **说明：**
> 如需希望使用 kubectl 或 helm 等方式部署 LWS，可参考 LWS 官方文档（kubectl 方式安装 和 helm 方式安装）。
> 需要注意的是，官方默认使用镜像是 `registry.k8s.io/lws/lws`，该镜像在中国大陆环境无法直接下载，可替换镜像地址为 `docker.io/k8smirror/lws`，该镜像为 lws 在 DockerHub 上的 mirror 镜像，长期自动同步，可放心使用（TKE 环境可直接拉取 DockerHub 的镜像），也可以同步到自己的 TCR 或 CCR 镜像仓库，提高镜像下载速度。

部署 DeepSeek-R1

使用本文指定的机型，每台有 8 张 GPU 算卡，单机部署也能成功运行，如果并发和吞吐要求较高，建议使用双机集群部署。
双机（多机）集群部署使用 LWS 中的 `LeaderWorkerSet` 来部署，单机部署则直接使用 `Deployment` 部署。
下面提供单机和双机两种部署方式的示例。

双机集群部署

使用 `LeaderWorkerSet` 部署满血版的 DeepSeek-R1 双机集群(2 台 8 卡的 GPU 节点，1 个 leader 和 1 个 worker)：

挂载 CFS 共享存储

```
apiVersion: leaderworkerset.x-k8s.io/v1
kind: LeaderWorkerSet
metadata:
  name: deepseek-r1
spec:
  replicas: 1
  leaderWorkerTemplate:
    size: 2
    restartPolicy: RecreateGroupOnPodRestart
    leaderTemplate:
      metadata:
        labels:
          role: leader
      spec:
        hostNetwork: true # 如果使用 HCCPNV6 机型，支持 RDMA，需要使用 HostNetwork 才能让 RDMA 生效。
        hostPID: true
        dnsPolicy: ClusterFirstWithHostNet # 如果使用 HostNetwork，默认使用节点上 /etc/resolv.conf 中的
dns server，会导致 LWS_LEADER_ADDRESS 指定的域名解析失败，所以 dnsPolicy 指定为 ClusterFirstWithHostNet 以便
使用 coredns 解析。
        containers:
        - name: leader
          image: lmsysorg/sglang:latest
          env:
          - name: LWS_WORKER_INDEX
            valueFrom:
              fieldRef:
                fieldPath: metadata.labels['leaderworkerset.sigs.k8s.io/worker-index']
          - name: TOTAL_GPU
            value: "16"
          - name: MODEL_DIRECTORY
            value: "DeepSeek-R1"
          - name: MODEL_NAME
            value: "DeepSeek-R1"
          - name: API_KEY
            value: ""
          command:
          - bash
          - -c
          - |
            set -x
            MODEL_DIRECTORY="${MODEL_DIRECTORY:-MODEL_NAME}"
            EXTRA_ARGS=""
            if [[ "$API_KEY" != "" ]]; then
                EXTRA_ARGS="--api-key $API_KEY"
            fi
            exec python3 -m sglang.launch_server \
              --model-path /data/model/$MODEL_DIRECTORY \
              --served-model-name $MODEL_NAME \
              --nnodes $LWS_GROUP_SIZE \
              --tp $TOTAL_GPU \
              --node-rank $LWS_WORKER_INDEX \
              --dist-init-addr $LWS_LEADER_ADDRESS:5000 \
              --log-requests \
              --enable-metric \
              --allow-auto-truncate \
              --watchdog-timeout 3600 \
```

```
            --disable-custom-all-reduce \
            --trust-remote-code \
            --host 0.0.0.0 \
            --port 30000 $EXTRA_ARGS
        resources:
          limits:
            nvidia.com/gpu: "8" # 每台节点 8 张 GPU 卡，每个 Pod 独占 1 台节点。
        ports:
        - containerPort: 30000
        volumeMounts:
        - mountPath: /dev/shm
          name: dshm
        - mountPath: /data/model
          name: data
      volumes:
      - name: dshm
        emptyDir:
          medium: Memory
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
  workerTemplate:
    spec:
      hostNetwork: true # worker 与 master 保持一致
      hostPID: true
      dnsPolicy: ClusterFirstWithHostNet # worker 与 master 保持一致
      containers:
      - name: worker
        image: lmsysorg/sglang:latest
        env:
        - name: LWS_WORKER_INDEX
          valueFrom:
            fieldRef:
              fieldPath: metadata.labels['leaderworkerset.sigs.k8s.io/worker-index']
        - name: TOTAL_GPU
          value: "16"
        - name: MODEL_DIRECTORY
          value: "DeepSeek-R1"
        - name: MODEL_NAME
          value: "DeepSeek-R1"
        command:
        - bash
        - -c
        - |
          set -x
          MODEL_DIRECTORY="${MODEL_DIRECTORY:-MODEL_NAME}"
          exec python3 -m sglang.launch_server \
            --model-path /data/model/$MODEL_DIRECTORY \
            --served-model-name $MODEL_NAME \
            --nnodes $LWS_GROUP_SIZE \
            --tp $TOTAL_GPU \
            --node-rank $LWS_WORKER_INDEX \
            --dist-init-addr $LWS_LEADER_ADDRESS:5000 \
            --log-requests \
            --enable-metric \
            --allow-auto-truncate \
            --watchdog-timeout 3600 \
            --disable-custom-all-reduce \
            --trust-remote-code
        resources:
```

```
            limits:
              nvidia.com/gpu: "8" # 每台节点 8 张 GPU 卡，每个 Pod 独占 1 台节点。
          volumeMounts:
          - mountPath: /dev/shm
            name: dshm
          - mountPath: /data/model
            name: data
        volumes:
        - name: dshm
          emptyDir:
            medium: Memory
        - name: data
          persistentVolumeClaim:
            claimName: ai-model

---
apiVersion: v1
kind: Service
metadata:
  name: deepseek-r1-api
spec:
  type: ClusterIP
  selector:
    leaderworkerset.sigs.k8s.io/name: deepseek-r1
    role: leader
  ports:
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

挂载本地存储

```
apiVersion: leaderworkerset.x-k8s.io/v1
kind: LeaderWorkerSet
metadata:
  name: deepseek-r1
spec:
  replicas: 1
  leaderWorkerTemplate:
    size: 2
    restartPolicy: RecreateGroupOnPodRestart
    leaderTemplate:
      metadata:
        labels:
          role: leader
      spec:
        hostNetwork: true # 如果使用 HCCPNV6 机型，支持 RDMA，需要使用 HostNetwork 才能让 RDMA 生效。
        hostPID: true
        dnsPolicy: ClusterFirstWithHostNet # 如果使用 HostNetwork，默认使用节点上 /etc/resolv.conf 中的
dns server，会导致 LWS_LEADER_ADDRESS 指定的域名解析失败，所以 dnsPolicy 指定为 ClusterFirstWithHostNet 以便
使用 coredns 解析。
        containers:
        - name: leader
          image: lmsysorg/sglang:latest
          env:
```

```
          - name: LWS_WORKER_INDEX
            valueFrom:
              fieldRef:
                fieldPath: metadata.labels['leaderworkerset.sigs.k8s.io/worker-index']
          - name: TOTAL_GPU
            value: "16"
          - name: MODEL_DIRECTORY
            value: "DeepSeek-R1"
          - name: MODEL_NAME
            value: "DeepSeek-R1"
          - name: API_KEY
            value: ""
          command:
          - bash
          - -c
          - |
            set -x
            MODEL_DIRECTORY="${MODEL_DIRECTORY:-MODEL_NAME}"
            EXTRA_ARGS=""
            if [[ "$API_KEY" != "" ]]; then
                EXTRA_ARGS="--api-key $API_KEY"
            fi
            exec python3 -m sglang.launch_server \
              --model-path /data/model/$MODEL_DIRECTORY \
              --served-model-name $MODEL_NAME \
              --nnodes $LWS_GROUP_SIZE \
              --tp $TOTAL_GPU \
              --node-rank $LWS_WORKER_INDEX \
              --dist-init-addr $LWS_LEADER_ADDRESS:5000 \
              --log-requests \
              --enable-metric \
              --allow-auto-truncate \
              --watchdog-timeout 3600 \
              --disable-custom-all-reduce \
              --trust-remote-code \
              --host 0.0.0.0 \
              --port 30000 $EXTRA_ARGS
          resources:
            limits:
              nvidia.com/gpu: "8" # 每台节点 8 张 GPU 卡，每个 Pod 独占 1 台节点。
          ports:
          - containerPort: 30000
          volumeMounts:
          - mountPath: /dev/shm
            name: dshm
          - mountPath: /data/model
            name: data
      volumes:
      - name: dshm
        emptyDir:
          medium: Memory
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
  workerTemplate:
    spec:
      hostNetwork: true # worker 与 master 保持一致
      hostPID: true
      dnsPolicy: ClusterFirstWithHostNet # worker 与 master 保持一致
      containers:
```

```
        - name: worker
          image: lmsysorg/sglang:latest
          env:
          - name: LWS_WORKER_INDEX
            valueFrom:
              fieldRef:
                fieldPath: metadata.labels['leaderworkerset.sigs.k8s.io/worker-index']
          - name: TOTAL_GPU
            value: "16"
          - name: MODEL_DIRECTORY
            value: "DeepSeek-R1"
          - name: MODEL_NAME
            value: "DeepSeek-R1"
          command:
          - bash
          - -c
          - |
            set -x
            MODEL_DIRECTORY="${MODEL_DIRECTORY:-MODEL_NAME}"
            exec python3 -m sglang.launch_server \
              --model-path /data/model/$MODEL_DIRECTORY \
              --served-model-name $MODEL_NAME \
              --nnodes $LWS_GROUP_SIZE \
              --tp $TOTAL_GPU \
              --node-rank $LWS_WORKER_INDEX \
              --dist-init-addr $LWS_LEADER_ADDRESS:5000 \
              --log-requests \
              --enable-metric \
              --allow-auto-truncate \
              --watchdog-timeout 3600 \
              --disable-custom-all-reduce \
              --trust-remote-code
          resources:
            limits:
              nvidia.com/gpu: "8" # 每台节点 8 张 GPU 卡，每个 Pod 独占 1 台节点。
          volumeMounts:
          - mountPath: /dev/shm
            name: dshm
          - mountPath: /data/model
            name: data
        volumes:
        - name: dshm
          emptyDir:
            medium: Memory
        - name: data
          hostPath:
            path: /data/model
            type: DirectoryOrCreate

---
apiVersion: v1
kind: Service
metadata:
  name: deepseek-r1-api
spec:
  type: ClusterIP
  selector:
    leaderworkerset.sigs.k8s.io/name: deepseek-r1
    role: leader
  ports:
```

```
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

说明：

- `nvidia.com/gpu` 为单机 GPU 卡数，本文示例中为 8 卡（leader 和 worker 保持一致）。
- `leaderWorkerTemplate.size` 为单个 GPU 集群的节点数，2 表示两个节点组成的 GPU 集群（1 个 leader 和 1 个 worker）。
- `replicas` 为 GPU 集群数量，这里是 1 个 GPU 集群，如需扩容，准备好节点资源后，调整此数量即可。
- `TOTAL_GPU` 为单个 GPU 集群的 GPU 总卡数（节点数量 * 单机 GPU 卡数），本文示例中为 16 卡。
- `MODEL_DIRECTORY` 为模型文件的子目录路径。
- `MODEL_NAME` 为模型名称，API 调用将使用此模型名称进行交互。
- leader 和 worker 的环境变量需一致，如需调整记得将 leader 和 worker 的 template 都做相同的修改。
- 如果使用支持 RDMA 的机型，需使用 HostNetwork 才能让 RDMA 生效。
- Service 中 `leaderworkerset.sigs.k8s.io/name` 指定的是 lws 的名称。
- 涉及 OpenAI API 地址配置的地方（如 OpenWebUI），指向这个 Service 的地址（如 `http://deepseek-r1-api:30000/v1`）。

部署完成后，如果需要扩容，可以通过调高 `replicas` 来增加 GPU 集群数量（前提是已准备好新的 GPU 节点资源）。

单机部署

使用 `Deployment` 部署单机满血版的 DeepSeek-R1：

挂载 CFS 共享存储

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deepseek-r1
  labels:
    app: deepseek-r1
spec:
  selector:
    matchLabels:
      app: deepseek-r1
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: deepseek-r1
    spec:
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        env:
        - name: TOTAL_GPU
          value: "8"
        - name: MODEL_DIRECTORY
          value: "DeepSeek-R1"
        - name: MODEL_NAME
          value: "DeepSeek-R1"
```

```
        - name: API_KEY
          value: ""
        command:
        - bash
        - -c
        - |
          set -x
          MODEL_DIRECTORY="${MODEL_DIRECTORY:-MODEL_NAME}"
          EXTRA_ARGS=""
          if [[ "$API_KEY" != "" ]]; then
              EXTRA_ARGS="--api-key $API_KEY"
          fi
          exec python3 -m sglang.launch_server \
            --model-path /data/model/$MODEL_DIRECTORY \
            --served-model-name $MODEL_NAME \
            --tp $TOTAL_GPU \
            --log-requests \
            --enable-metric \
            --allow-auto-truncate \
            --watchdog-timeout 3600 \
            --disable-custom-all-reduce \
            --trust-remote-code \
            --mem-fraction-static 0.9 \
            --max-running-request 8 \
            --host 0.0.0.0 \
            --port 30000 $EXTRA_ARGS
        resources:
          limits:
            nvidia.com/gpu: "8"
        ports:
        - containerPort: 30000
        readinessProbe:
          httpGet:
            path: /health
            port: 30000
          periodSeconds: 5
        volumeMounts:
        - name: data
          mountPath: /data/model
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: ai-model
      - name: shm
        emptyDir:
          medium: Memory
      restartPolicy: Always

---
apiVersion: v1
kind: Service
metadata:
  name: deepseek-r1-api
spec:
  type: ClusterIP
  selector:
    app: deepseek-r1
  ports:
```

```
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

挂载本地存储

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deepseek-r1
  labels:
    app: deepseek-r1
spec:
  selector:
    matchLabels:
      app: deepseek-r1
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: deepseek-r1
    spec:
      containers:
      - name: sglang
        image: lmsysorg/sglang:latest
        env:
        - name: TOTAL_GPU
          value: "8"
        - name: MODEL_DIRECTORY
          value: "DeepSeek-R1"
        - name: MODEL_NAME
          value: "DeepSeek-R1"
        - name: API_KEY
          value: ""
        command:
        - bash
        - -c
        - |
          set -x
          MODEL_DIRECTORY="${MODEL_DIRECTORY:-MODEL_NAME}"
          EXTRA_ARGS=""
          if [[ "$API_KEY" != "" ]]; then
              EXTRA_ARGS="--api-key $API_KEY"
          fi
          exec python3 -m sglang.launch_server \
            --model-path /data/model/$MODEL_DIRECTORY \
            --served-model-name $MODEL_NAME \
            --tp $TOTAL_GPU \
            --log-requests \
            --enable-metric \
            --allow-auto-truncate \
            --watchdog-timeout 3600 \
            --disable-custom-all-reduce \
```

```
            --trust-remote-code \
            --mem-fraction-static 0.9 \
            --max-running-request 8 \
            --host 0.0.0.0 \
            --port 30000 $EXTRA_ARGS
        resources:
          limits:
            nvidia.com/gpu: "8"
        ports:
        - containerPort: 30000
        readinessProbe:
          httpGet:
            path: /health
            port: 30000
          periodSeconds: 5
        volumeMounts:
        - name: data
          mountPath: /data/model
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: data
        hostPath:
          path: /data/model
          type: DirectoryOrCreate
      - name: shm
        emptyDir:
          medium: Memory
      restartPolicy: Always

---
apiVersion: v1
kind: Service
metadata:
  name: deepseek-r1-api
spec:
  type: ClusterIP
  selector:
    app: deepseek-r1
  ports:
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

说明：

- `nvidia.com/gpu` 和 `TOTAL_GPU` 都是单机 GPU 卡数，本文示例中为 8 卡。
- `replicas` 为 DeepSeek-R1 副本数，1 个副本占用 1 台 GPU 节点。
- `MODEL_DIRECTORY` 为模型文件的子目录路径。
- `MODEL_NAME` 为模型名称，API 调用将使用此模型名称进行交互。
- 由于是单机部署，无需 RDMA，也无需使用 HostNetwork。
- 单机部署配置了 `--mem-fraction-static` 和 `--max-running-request` 参数，用于避免显存不足导致 SGLang 启动失败。
- 涉及 OpenAI API 地址配置的地方（如 OpenWebUI），指向这里创建的 Service 的地址（如 `http://deepseek-r1-api:30000/v1` ）。

部署完成后，如果需要扩容，可以通过调高 `replicas` 来增加 DeepSeek-R1 副本数（前提是准备好新的 GPU 节点资源）。

验证 API

Pod 成功跑起来后用 kubectl exec 进入 leader Pod，使用 curl 测试 API：

```
curl -v http://127.0.0.1:30000/v1/completions -H 'X-API-Key: ******'  -H "Content-Type: application/json"
-d '{
      "model": "DeepSeek-R1",
      "prompt": "你是谁?",
      "max_tokens": 100,
      "temperature": 0
    }'
```

常见问题

如何对外暴露 API？

通常对外暴露 API 一般会配置 API 密钥，配置方法是修改本文示例中的 YAML，将密钥配置到 `API_KEY` 环境变量中。
如果希望将 API 对外暴露，最简单的是直接修改 DeepSeek 的 Service 类型为 LoadBalancer，TKE 会自动为其创建公网 CLB 将 API 暴露到公网：

双机集群部署版

```
apiVersion: v1
kind: Service
metadata:
  name: deepseek-r1-api
spec:
  type: LoadBalancer
  selector:
    leaderworkerset.sigs.k8s.io/name: deepseek-r1
    role: leader
  ports:
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

单机部署版

```
apiVersion: v1
kind: Service
metadata:
  name: deepseek-r1-api
spec:
  type: LoadBalancer
  selector:
    app: deepseek-r1
  ports:
  - name: api
    protocol: TCP
    port: 30000
    targetPort: 30000
```

如果需要更灵活的方式暴露，例如配置证书通过 HTTPS 协议暴露，或者与其他服务共用网关入口，可以通过 Ingress 或 Gateway API 来暴露，示例：

Gateway API

注意：
使用 Gateway API 需要集群中装有 Gateway API 的实现，如 TKE 应用市场中的 EnvoyGateway，具体 Gateway API 用法参考 官方文档 。

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: deepseek-api
spec:
  parentRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    namespace: envoy-gateway-system
    name: deepseek
  hostnames:
  - "deepseek.your.domain"
  rules:
  - backendRefs:
    - group: ""
      kind: Service
      name: deepseek-r1-api
      port: 30000
```

说明：

1. `parentRefs` 引用定义好的 `Gateway`（通常一个 Gateway 对应一个 CLB）。
2. `hostnames` 替换为您自己的域名，确保域名能正常解析到 Gateway 对应的 CLB 地址。
3. `backendRefs` 指定 DeepSeek 的 Service。

Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: deepseek-api
spec:
  rules:
  - host: "deepseek.your.domain"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: deepseek-r1-api
            port:
              number: 30000
```

说明：

1. `host` 替换为您自己的域名，确保域名能正常解析到 Ingress 对应的 CLB 地址。
2. `backend.service` 指定 DeepSeek 的 Service。

最后在需要使用 API 的应用中配置 API 地址、API 密钥、模型名称等，例如 Chatbox 的配置：

说明：

- **模型提供方**：由于 SGLang 兼容 OpenAI 的 API，所以选择 **OPENAI API**。
- **API 密钥**：填写 DeepSeek-R1 部署时指定的 API KEY（`API_KEY` 环境变量）。
- **API 域名**：用 DeepSeek-R1 最终被暴露出来的外部地址。
- **模型**：填写 DeepSeek-R1 部署时指定的模型名称（`MODEL_NAME` 环境变量）。

如何使用 OpenWebUI 与模型对话?

SGLang 提供了兼容 OpenAI 的 API，部署 OpenWebUI 时，如不需要 Ollama API 可禁用掉，再配置下 OpenAI 的 API 地址，指向 DeepSeek-R1 的地址即可。

如果 使用 helm 部署 OpenWebUI，`values.yaml` 配置示例：

```
ollama:
  enabled: false
openaiBaseApiUrl: "http://deepseek-r1:30000/api/v1"
```

如果通过 YAML 部署 OpenWebUI，需配置下 Pod 环境变量，示例：

```
env:
```

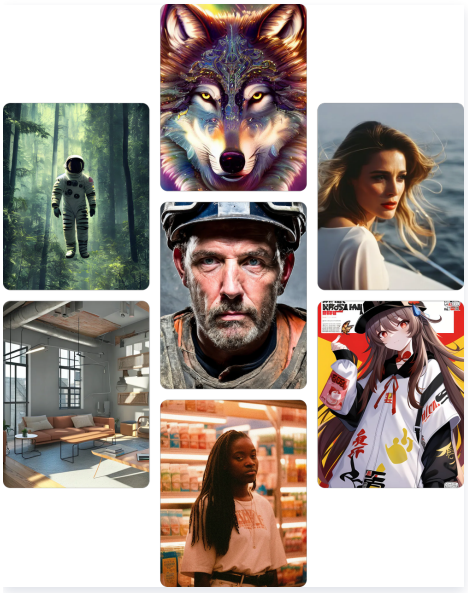
```
- name: OPENAI_API_BASE_URL
  value: http://deepseek-r1:30000/api/v1 # vllm 的地址
- name: ENABLE_OLLAMA_API # 禁用 Ollama API，只保留 OpenAI API
  value: "False"
```

使用 TKE 完整部署生产级 Stable Diffusion

最近更新时间：2024-11-20 17:56:42

技术背景

Stable Diffusion 是一种深度学习的文本到图像模型，由 Runway 和慕尼黑大学合作构建，第一个版本于2021年发布。目前主流版本包含 v1.5、v2和 v2.1。它主要用于生成基于文本描述的详细图像，也应用于其他任务，如修复图像、生成受文本提示引导的图像到图像的转换等。Stable Diffusion 示例图如下：



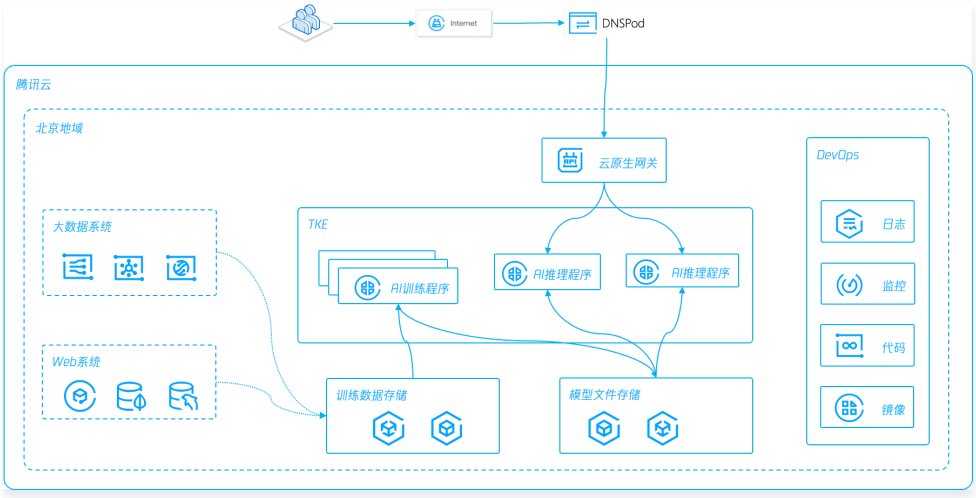
应用场景

AI 绘图在互联网行业领域会起到革命性的作用，目前多数是加快现有工作的效率和质量。以下是几个典型的应用场景：

- **插画**：无论是游戏 CG 还是概念插画，使用 AI 绘图都能起到很好的效果。一般来说，在商用环境下不会直接使用结果上线，而是在这个基础上使用 PS 等软件进行修改，或以此作为灵感重绘。
- **游戏 UI**：过去游戏界面的绘制，一般需要对单个物品（如宝箱、装备、武器）进行多个版本的绘制。使用 AI 绘图大量生产图片或图标，然后进行修改或者临摹，在缩短出图时间的同时，减少出图成本。
- **平面包装**：平面包装是一个依赖抽象图形的领域方向，整体上对图片精确率要求不高，只需要模糊的方向和概念，并且接受随机生成的样式结果。这一特性刚好符合 AI 绘图的特性，通过外部轮廓设定后，使用 AI 对内容图案进行绘制。
- **服装设计及模特拍摄**：在服装设计领域，很多时候都依赖一些抽象的灵感和思路，采用 AI 出图后，服装设计师可以根据图片进行灵感设计，根据服装材料、流行程度和季节，针对图片进行重新调整。另外，拍摄模特图片非常耗费时间和成本，采用 AI 方式进行试穿，也是 AI 绘图的发展方向。
- **建筑效果图**：利用 ControlNet 的能力，建筑领域也开始尝试使用 AI 绘图进行效果图生成，为建筑设计师提供灵感，未来结合 3D 模型生成，所写即所得，提高建筑设计师和客户的沟通效率。

行业客户普遍基于上述场景做 AI 绘图业务尝试：Stable Diffusion 预训练模型，加上各种微调插件，如 LoRA 进行风格定制，ControlNet 控制图像，能有效输出符合业务场景定义的图片素材。

部署Stable Diffusion 架构图



搭建步骤

1. 准备需使用 Stable Diffusion 容器镜像

1. 从 GitHub 下载 Stable Diffusion web UI 代码，制作 Docker 镜像。也可使用以下命令获取：

```
docker pull gpulab.tencentcloudcr.com/ai/stable-diffusion:1.0.7
```

2. 将准备好的 Stable Diffusion 容器镜像上传到容器镜像仓库 TCR，具体操作可参见 TCR 企业版指南 。
3. 如果您需要对仓库权限进行细粒度控制，例如在镜像内打包模型文件的使用场景，推荐使用 CAM 对命名空间或仓库进行访问控制，具体操作可参见 容器镜像服务权限管理 。另外，由于模型文件加上推理服务，镜像体积可能达到几十GB，而企业版容器仓库支持按需加载，提升应用分发效率，具体操作可参见 按需加载容器镜像 。

2. 准备待部署 Stable Diffusion 的 TKE 集群

1. 开通并创建 TKE 集群，操作步骤详情可参见 创建容器服务集群 。在创建集群时，Kubernetes 版本选择最新的1.26.1，容器网络插件选择 Global Router，其他选项默认即可。
2. 集群选择**托管**类型，Worker 节点选择 GPU 计算型PNV4 – A10，安装 GPU470驱动，CUDA 版本11.4.3，cuDNN 版本 8.2.4。如下图所示：

3. 根据部署对 GPU 共享的需求，您可以选择开启 qGPU，详情见 Stable Diffusion 使用 qGPU 。

3. 通过 TKE+CFS 快速部署 Stable Diffusion Web UI

1. 创建存放模型的文件存储 CFS

1. 开通 CFS 服务，创建文件系统及挂载点时选择与集群相同的 VPC 和子网。在 CFS 远程挂载点，新建 /models/Stable-diffusion 目录。挂载点和文件操作，详情可参见 创建文件系统及挂载点 。
2. 下载 v1-5-pruned-emaonly.safetensors 模型文件至 /models/Stable-diffusion，地址见：runwayml/stable-diffusion-v1-5 。
3. 如需更大的带宽和读写 IOPS，建议选择 CFS Turbo，容器环境使用可参见 在 TKE 上使用 CFS Turbo 。

2. 创建静态 PV/PVC

1. 在 容器服务控制台 中创建 CFS 类型 StorageClass，并选择**共享实例**。操作步骤可参见 通过控制台创建 StorageClass 。

2. 使用 CFS 中新建的 /models/Stable-diffusion 目录以及已创建的 StorageClass，静态创建 PV/PVC。
   创建 PV 如下图所示：

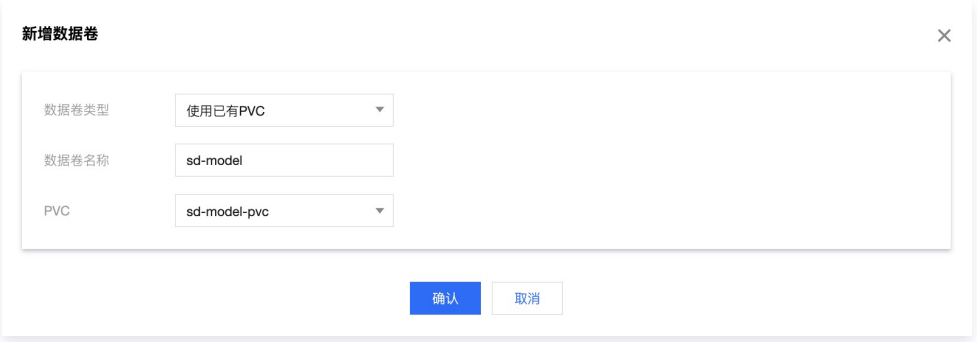
创建 PVC 如下图所示：

3. **如果您有其他模型目录挂载的需求，同样需要在 CFS 挂载点中新建子目录，并进行 PV/PVC 的静态创建。Stable Diffusion Web UI 服务的 models 子目录结构如下：**

```
[root@VM-0-5-centos models]# tree
.
|-- Codeformer
|-- ESRGAN
|-- GFPGAN
|-- LDSR
|-- Lora
|-- Stable-diffusion
|   |-- Put\ Stable\ Diffusion\ checkpoints\ here.txt
|   `-- v1-5-pruned-emaonly.safetensors
|-- SwinIR
|-- VAE
|   `-- Put\ VAE\ here.txt
|-- VAE-approx
|   `-- model.pt
|-- deepbooru
|   `-- Put\ your\ deepbooru\ release\ project\ folder\ here.txt
`-- hypernetworks

11 directories, 5 files
[root@VM-0-5-centos models]# pwd
/root/stable-diffusion-webui/models
```

3. 创建 Stable Diffusion Web UI 工作负载

1. 在 容器服务控制台 中，选择左侧导航中的**集群**。
2. 在集群详情页，选择**工作负载 > Deployment**，单击**新建**，开始部署 stable-diffusion-webui 镜像。
3. 在**新建 Deployment** 页，填写 Deployment 基本信息，其中数据卷选择**添加数据卷**。
4. 在**新增数据卷**页，数据卷类型选择**使用已有 PVC**，添加 已创建的 PVC，完成后单击**确认**。



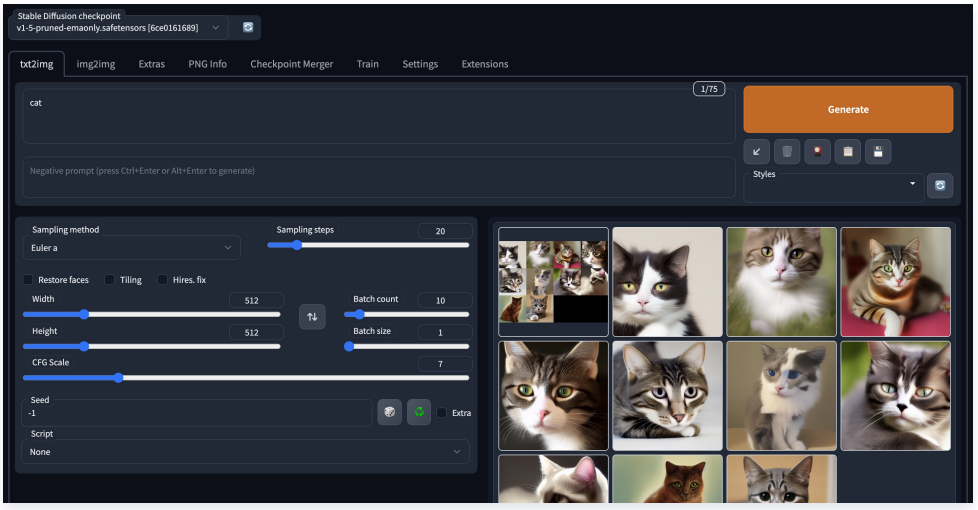
5. 在实例内容器中，单击**选择镜像**，选择已保存在 TCR 中的 stable-diffusion-webui 镜像。
6. 将新建的数据卷进行挂载点配置。挂载点与 CFS 远程目录对应关系如表格所示：

PVC	容器挂载点	对应的CFS目录
sd-model-pvc	/dockerx/stable-diffusion-webui/models/Stable-diffusion	/models/Stable-diffusion

7. 展开**显示高级设置**，添加运行参数 `--listen`，将 stable-diffusion-webui 进程监听在0.0.0.0上。更多参数选项参见 Command Line Arguments and Settings。

8. 将 GPU 资源的卡数设置为1，如果开启了 qGPU，您还可以填写0.1-1之间的数值，对 GPU 卡进行虚拟化切分。
9. 创建 Deployment 对应的 Service，并**选择公网 LB 访问**，对外暴露7860端口访问。

10. 通过 CLB 公网 IP 地址，您就可以成功访问 Stable Diffusion Web UI 服务了。如果您有限流或访问控制的需求，我们推荐您选择云原生网关充当 Ingress Controller。在本文的 通过云原生 API 网关对外提供 Stable Diffusion 服务 部分中，我们会介绍云原生网关的使用。



（进阶）Stable Diffusion 使用 qGPU

Stable Diffusion Web UI 服务以串行方式处理请求，如果您希望增加推理服务的并发性能，可以考虑扩展 Deployment 的 Pod 数量，以轮询的方式响应请求。在这里，我们采用 TKE qGPU 能力，将多个实例 Pod 运行在同一张 A10 卡上。在保障业务稳定性的前提下，切分显卡资源，降低部署成本。
采用 qGPU 方式，您需要先将 Pod 的资源申请方式进行修改。例如，如果您计划在单卡上部署2个 Pod，您需要在 YAML 文件中将 tke.cloud.tencent.com/qgpu-core 从100更改为50，也就是将50%的算力分配给每个 Pod。同时，您还需要将 tke.cloud.tencent.com/qgpu-memory 的数值设置为 A10 显存的一半。

```
resources:
  limits:
```

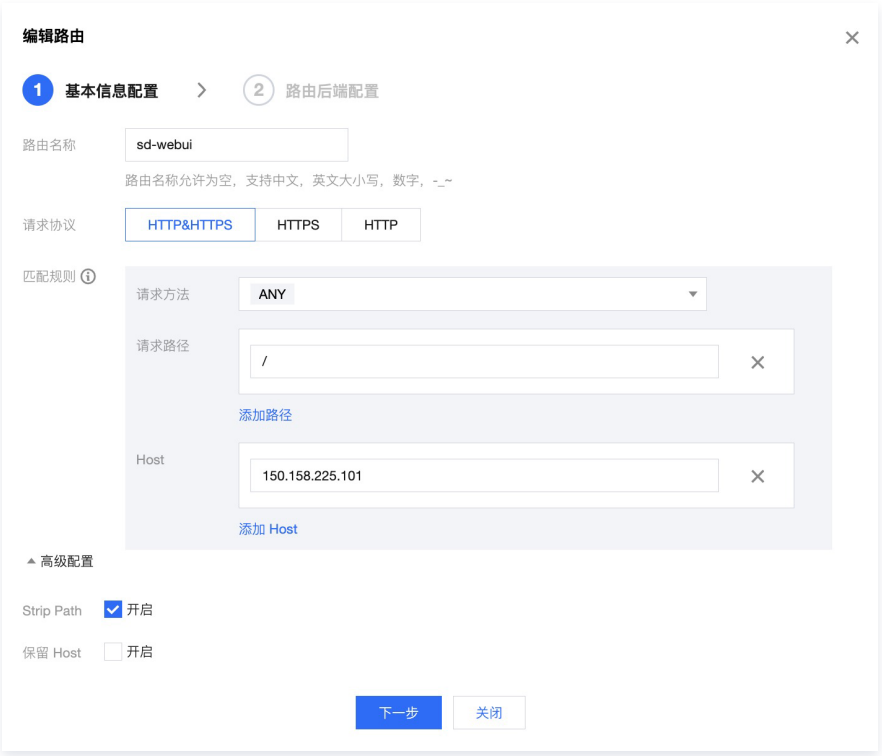
```
    cpu: "20"
    memory: 50Gi
    tke.cloud.tencent.com/qgpu-core: "50"
    tke.cloud.tencent.com/qgpu-memory: "10"
```

Deployment YAML 文件更新完成后，调整 Pod 数量为2个，即可实现负载均衡的 Stable Diffusion 轮询模式。

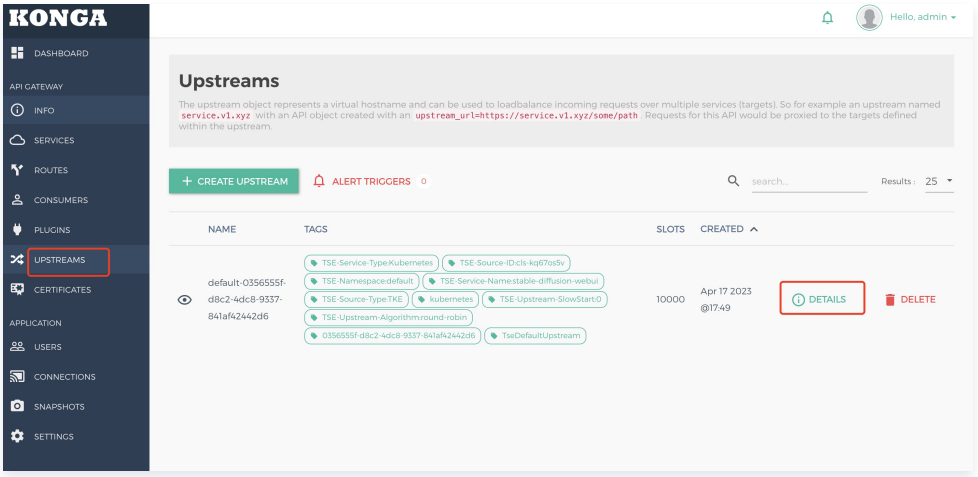
> **注意：**
> 在 tke.cloud.tencent.com/qgpu-core 为100的整倍数时，您可以不写 qgpu-memory，系统会默认分配整卡显存。但是，当 tke.cloud.tencent.com/qgpu-core 小于100时，也就是按比例切分 GPU 资源时，您必须显式指定 tke.cloud.tencent.com/qgpu-memory。如需了解更多关于 qGPU 的详情，请参见 容器服务使用 qGPU 。

（进阶）通过云原生 API 网关对外提供 Stable Diffusion 服务

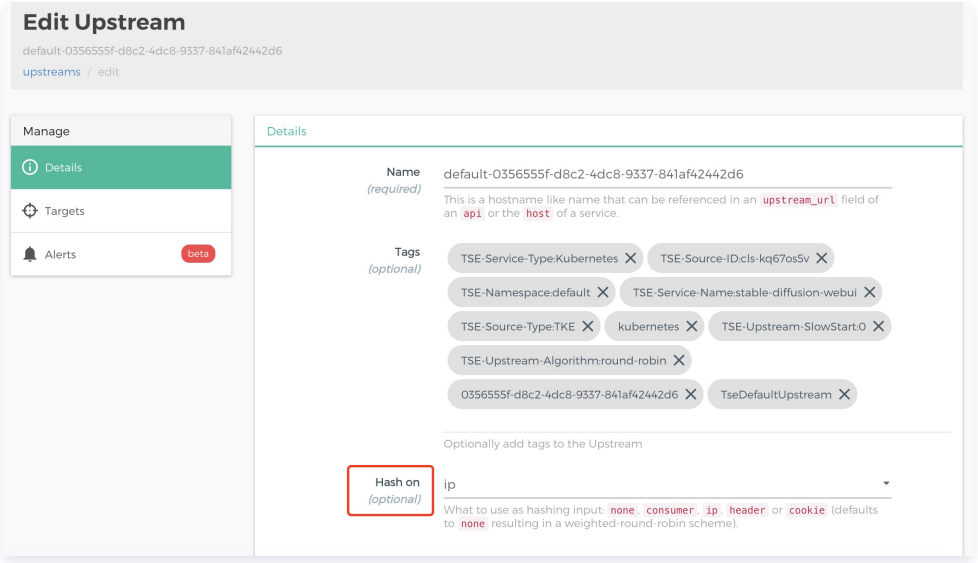
1. 开通云原生网关，选择和 TKE 集群、CFS 同 VPC 的实例。更多关于如何新建网关的信息，可参见 微服务引擎 TSE 新建网关 。
2. 在 腾讯云微服务引擎控制台 上，选择实例名称，进入实例详情页。
3. 选择**路由管理 > 服务来源**，单击**新建**，在**新建服务来源**中选择**容器服务**，绑定 TKE 集群。
4. 选择**路由管理 > 服务**，单击**新建**，新建网关服务。选择**服务列表**时，选择部署 Deployment 时启用的 Service 进行映射。云原生网关会自动拉取 TKE Service 关联的 Pod IP。当 Pod IP 变化时，动态更新网关服务里的 Upstream 配置项。如需了解更多关于如何创建 TKE 服务的路由的信息，可参见 微服务引擎 TSE 创建 TKE 服务的路由 。
5. 单击服务名，新建访问路由。在基本信息配置中，将请求方法设置为 ANY，Host 填写云原生网关的公网 IP。如果后期绑定域名使用，Host 还需要加上域名地址。如下图所示：



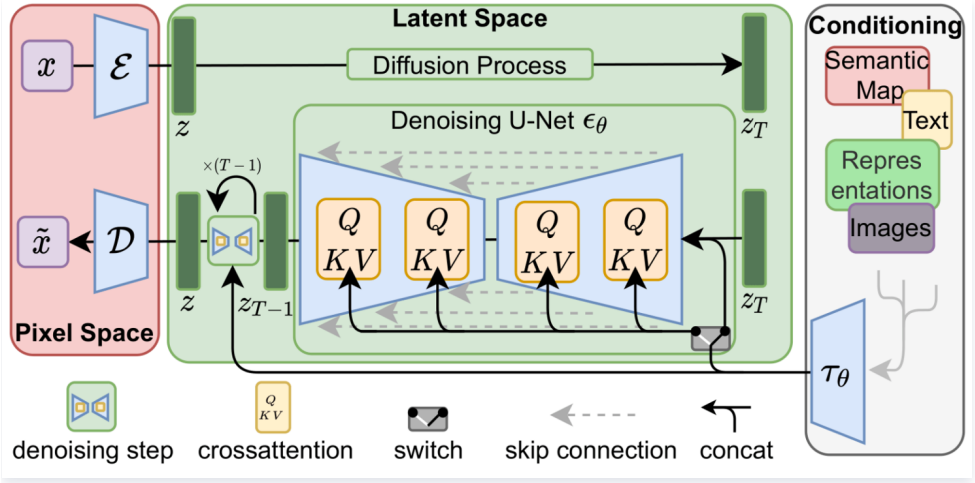
6. 根据资源用量和计划访问请求数，您可以选择配置网关限流策略，并自定义限流响应内容，更多关于如何配置限流策略的信息，可参见 微服务引擎 TSE 配置限流策略 。
7. Stable Diffusion Web UI 出图时会进行多轮请求，将 Deployment 的 Pod 副本数量修改为大于1时，您还需要配置 Session 会话保持，以保证同一 IP 的客户请求落在相同的 Pod 里。选择**路由管理 > Konga 控制台**，找到 Konga 公网访问地址，在 Konga 控制台里找到 UPSTREAM，单击 **DETAILS**，如下图所示：



在 HASH ON 下拉框里，选择 IP，完成基于客户端 IP 的会话保持配置。

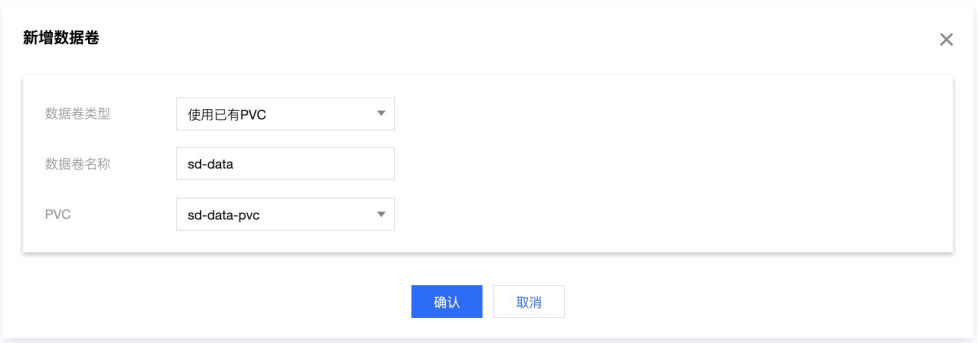


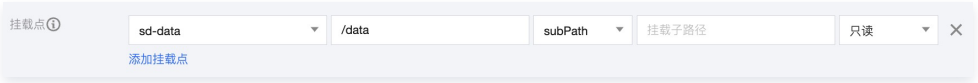
（进阶）优化 Stable Diffusion 推理性能



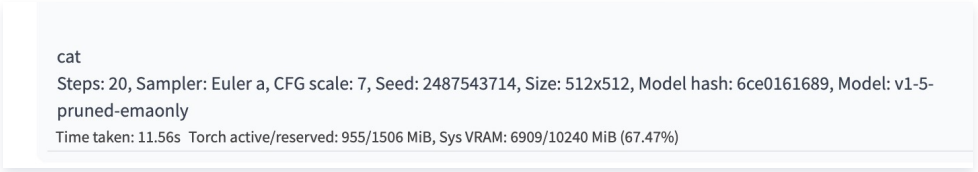
Stable Diffusion 是一个多模型组成的扩散 Pipeline，主要由三个部分组成：变分自编码器 VAE、U-Net 和文本编码器 CLIP。推理耗时主要集中在 UNet 部分，我们选择对这部分进行模型优化，以加速推理速度。

1. 下载 A10 GPU 优化的 stable-diffusion-v1.5 UNet 模型文件，以及 sd_v1.5_demo 镜像，该镜像里的 Web UI 修改了模型加载代码，UNet 部分会加载独立优化模型。镜像及 UNet 优化模型获取，可参见 TACO Infer 部署 Stable Diffusion web UI 。
2. 将 sd_v1.5_demo 镜像服务部署在 TKE 上：按前述步骤进行操作，其中替换镜像为 sd_v1.5_demo，并额外为 UNet 优化模型创建 CFS /data 目录和 PV/PVC。





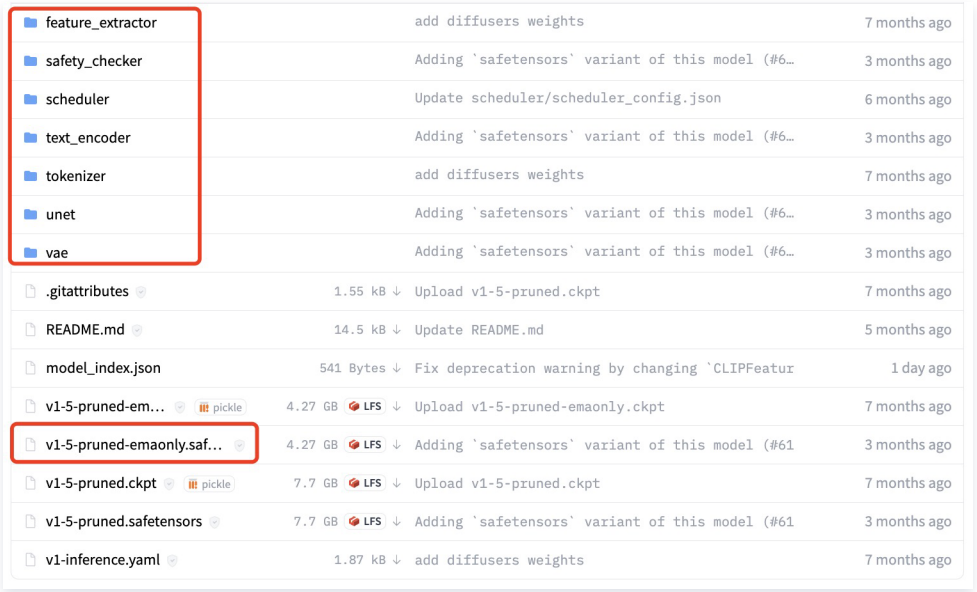
3. 在相同的参数配置下，生成10张猫的图片。在优化前，推理耗时为16.14s。在加载 TACO 优化的 UNet 模型后，10张图片仅耗时11.56s，端到端性能提高30%。



4. TACO 可以对 Stable Diffusion 系列模型进行优化。如果您希望对其他 Stable Diffusion 微调模型进行推理优化，并部署在上述环境中，可以按照以下步骤操作：

   4.1 参见 TACO Infer 优化 Stable Diffusion 模型，拉取预置库环境的 sd_taco:v3 镜像。

   4.2 Stable Diffusion 模型主要有两种存储方式：单文件和 diffusers 目录结构。其中 diffusers 结构按照 Stable Diffusion 的模型结构组织，包含 unet、vae、text-encoder 等。在 TACO 优化过程中，会使用 diffusers 结构读取模型。您可以在 HuggingFace 上找到这种格式的模型文件进行下载。



   4.3 如果 HuggingFace 速度较慢，也可以使用官方的转换脚本，将单文件格式（ckpt 或 safetensors）转化成 diffusers 格式使用。脚本见 diffusers/scripts at main · huggingface/diffusers · GitHub 。

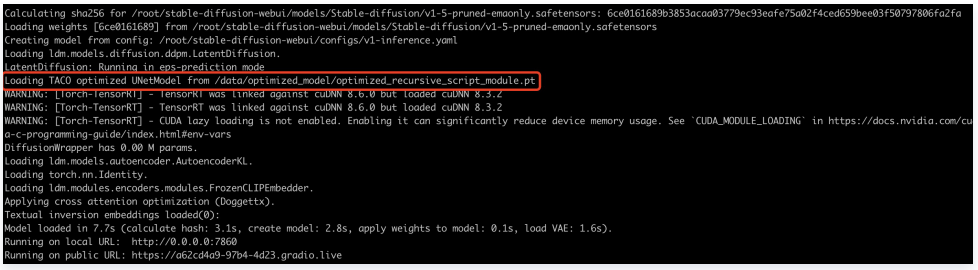
```
python convert_original_stable_diffusion_to_diffusers.py --checkpoint_path [single_file_model_name]
--dump_path [diffusers_model_directory] --from_safetensors
```

4.4 选择一台 A10 GPU CVM，使用 -v 命令挂载上面的 diffusers 模型目录，交互式启动容器，在容器内部对挂载好的模型进行优化。

```
docker run -it --gpus=all --network=host -v
/[diffusers_model_directory]:/[custom_container_directory] sd_taco:v3 bash
```

4.5 使用 diffusers 加载模型权重，从中导出 UNet 模型进行优化。运行脚本请参见 TACO Infer 优化 Stable Diffusion 模型 。

4.6 完成后将优化后的模型放入 CFS 挂载的 /data 目录。UNet 从优化文件中加载，而单文件格式模型（ckpt 或 safetensors）仍然放入 CFS 挂载的 /models/Stable-diffusion 目录，Stable Diffusion 其他部分从原始文件里加载。



4.7 重启 stable-diffusion-webui 界面，选择新模型使用。

结论

本文介绍了 Stable Diffusion 模型在互联网行业的应用场景，并展示了如何利用腾讯云云原生产品能力，进行高可用部署的工程化实践。
在生产环境中，推理服务需要考虑并发请求下服务的可用性和扩展性，同时也需要考虑多模型文件管理的便利性，以及配合当前业务架构的灵活性；Stable Diffusion 的前向推理过程是一个比较耗时的过程，GPU 应用部署对比 CPU 应用部署成本也较高，在控制成本的前提下，如何有效地提高推理速度，也是企业需要重点考虑的因素；另外，作为在线业务，推理服务需要合理设计限流熔断，避免流量激增造成整体业务不可用。
利用腾讯云云原生能力，能够轻松满足上述需求。在此基础上，可以考虑将前端 Web 应用和后端推理服务进行解耦，提高架构吞吐能力；当业务访问具有明显潮汐现象时，还可以通过 TKE GPU HPA 的弹性能力，进一步降低资源部署成本。

相关文档

- 容器服务 TKE
- 容器镜像服务
- 文件存储 CFS
- 云原生 API 网关
- 计算加速套件 TACO Kit

使用 TKE 快速部署 ChatGLM

最近更新时间：2023-11-23 16:53:22

背景介绍

ChatGLM-6B 是一个开源的、支持中英双语的对话语言模型，基于 General Language Model (GLM) 架构，具有62亿参数。结合模型量化技术，用户可以在消费级的显卡上进行本地部署（INT4 量化级别下最低只需6GB显存）。ChatGLM-6B 使用了和 ChatGPT 相似的技术，针对中文问答和对话进行了优化。经过约1TB标识符的中英双语训练，辅以监督微调、反馈自助、人类反馈强化学习等技术的加持。
在开源的中文大模型中，ChatGLM-6B 是非常不错的选择。最近，来自 LMSYS Org（UC 伯克利主导）的研究人员发起的大语言模型版排位赛中，ChatGLM 虽然只有60亿参数，但依然冲进了前五，只比130亿参数的 Alpaca 落后了23分。基于或使用了 ChatGLM-6B 的开源项目如下：

- langchain-ChatGLM：基于 langchain 的 ChatGLM 应用，实现基于可扩展知识库的问答。
- 闻达：大型语言模型调用平台，基于 ChatGLM-6B 实现了类 ChatPDF 功能。
- chatgpt_academic：支持 ChatGLM-6B 的学术写作与编程工具箱，具有模块化和多线程调用 LLM 的特点，可并行调用多种 LLM。
- glm-bot：将 ChatGLM 接入 Koishi 可在各大聊天平台上调用 ChatGLM 话题。

部署步骤

通过控制台创建。

准备集群

1. 开通并创建 TKE 集群，操作步骤详情可参见 创建容器服务集群 。如下图所示：

2. 在创建集群时，集群选择**托管**类型。由于 ChatGLM-6B 的 GPU 版本最少需要14G显存，因此 Worker 节点选择 V100 GPU 节点，型号为 GN10Xp.2XLARGE40，系统盘空间为100GB。如下图所示：

3. 根据业务部署时对 GPU 的共享需求，可以选择并打开 qGPU 选项。关于 qGPU 的使用方法，可参见 容器服务使用 qGPU 。
4. 配置完毕后，确认选项及扩展组件无误，单击**确认**。

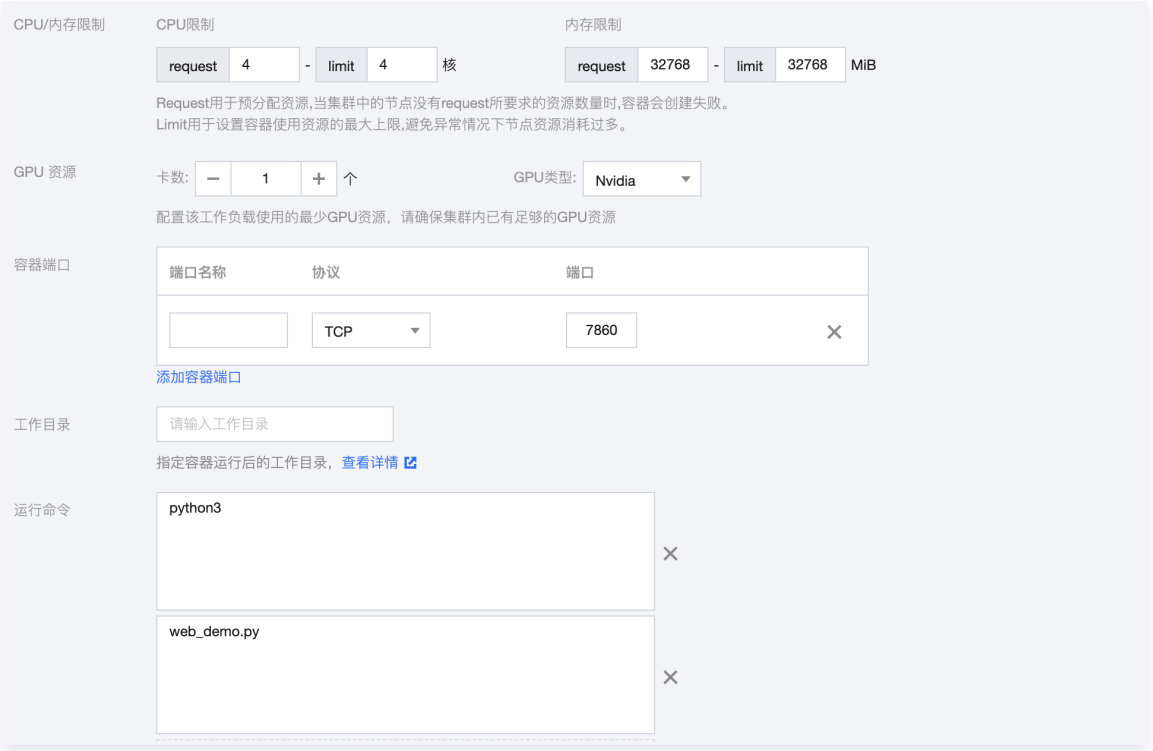
创建应用

使用控制台创建 Deployment

1. 在 容器服务控制台 中，选择集群 ID，进入集群详情页。
2. 选择**工作负载 > Deployment**，在 Deployment 页面单击**新建**。
3. 在**新建 Deployment** 中，参考以下信息开始创建 ChatGLM 应用。

在实例内容器中，

- **名称：**填写 chatglm-6b。
- **镜像：**填写 ccr.ccs.tencentyun.com/chatglm/chatglm-6b:v1.2。
- **CPU/内存限制：**由于 ChatGLM-6B 的资源需要主要是 GPU，因此 CPU 和内存可以按需设置。
- **GPU 资源：**设置为1个，GPU 类型选择 Nvidia。
- **容器端口：**端口填写7860。
- **运行命令：**分别添加 `python3` 和 `web_demo.py` 。如下图所示：



4. **在基本信息中，实例数量选择手动调节，并设置为1个。如下图所示：**

5. **为了让外部可访问，在访问设置中，您可参考以下信息进行设置。服务访问方式：选择公网 LB 访问。**
   - **端口映射：将容器端口和服务端口都填写为7860。**

- **服务访问方式**：选择公网 LB 访问。
- **端口映射**：将容器端口和服务端口都填写为7860。

6. 设置完成后，单击**创建 Deployment**，等待 Pod Ready。需注意，由于镜像大小为25GB，因此需要等待一段时间才能完成部署，大致需要等待半小时左右。
7. 选择**服务与路由 > Service**，在页面上获取项目的公网 IP。如下图所示：

在浏览器中输入 http:// `IP` :7860，即可访问。

使用 YAML 创建 Deployment

您也可以使用以下 YAML 配置直接创建 Deployment：

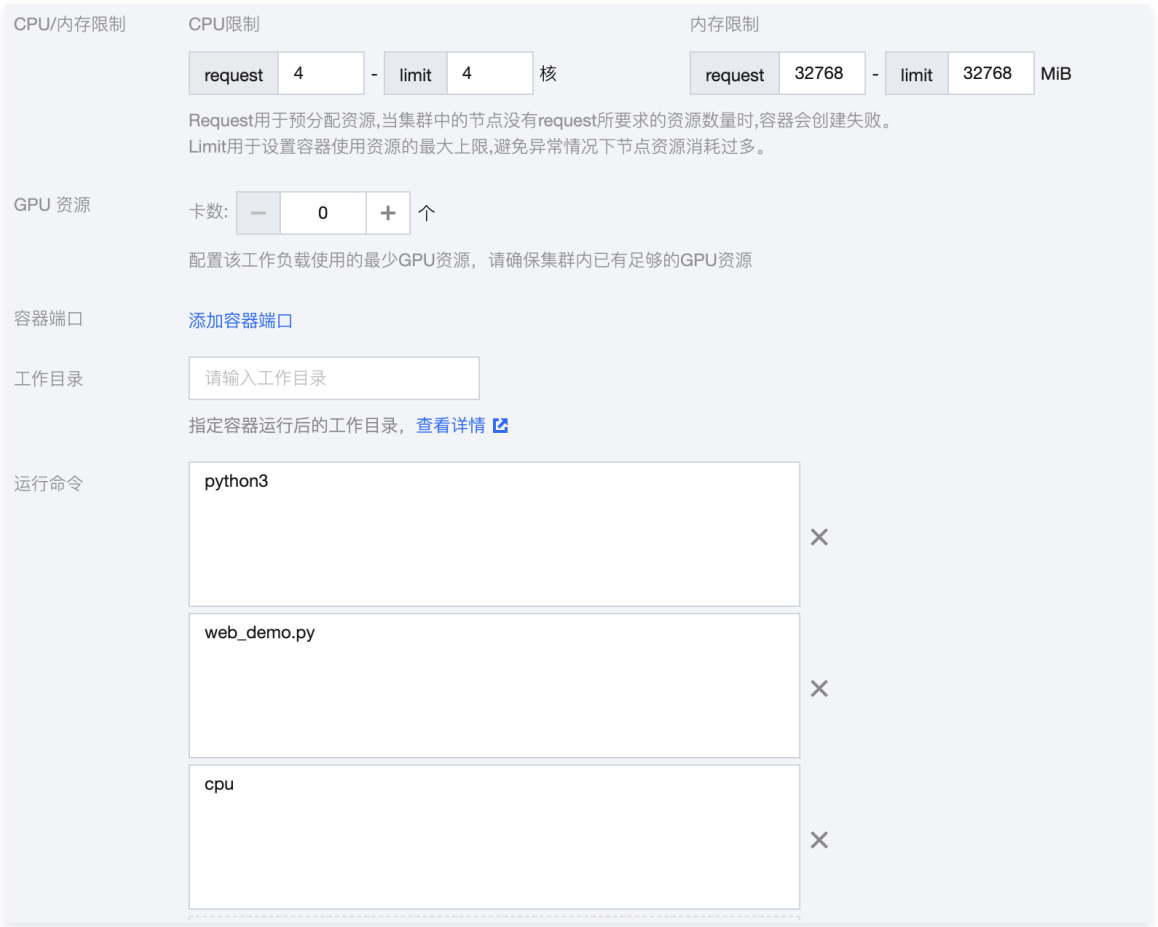
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: chatglm-6b
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: chatglm-6b
  template:
    metadata:
      labels:
        app: chatglm-6b
    spec:
      containers:
      - name: chatglm-6b
        image: ccr.ccs.tencentyun.com/chatglm/chatglm-6b:v1.2
        command:
        - python3
        - web_demo.py
        ports:
        - containerPort: 7860
        resources:
          limits:
            nvidia.com/gpu: "1"

---

apiVersion: v1
kind: Service
metadata:
  name: chatglm-6b-svc
spec:
  type: ClusterIP
  ports:
  - port: 7860
    targetPort: 7860
  selector:
    app: chatglm-6b
```

创建 CPU 版本应用

如果没有 GPU 硬件，ChatGLM-6B 也可以在 CPU 上进行推理，但推理速度会更慢。资源需要大概32GB内存，相比创建 GPU 版本应用，您需要注意以下两点：

1. 内存资源需要32GB。
2. 将 运行命令 改为 python3 web_demo.py cpu。如下图所示：


CPU/内存限制
CPU限制
request 4 - limit 4 核
内存限制
request 32768 - limit 32768 MiB
Request用于预分配资源,当集群中的节点没有request所要求的资源数量时,容器会创建失败。
Limit用于设置容器使用资源的最大上限,避免异常情况下节点资源消耗过多。
GPU 资源
卡数: 0 个
配置该工作负载使用的最少GPU资源，请确保集群内已有足够的GPU资源
容器端口
添加容器端口
工作目录
请输入工作目录
指定容器运行后的工作目录，查看详情
运行命令
python3
web_demo.py
cpu

使用 TKE + 超级节点快速体验 Stable Diffusion

最近更新时间：2024-01-09 10:18:41

技术背景

Stable Diffusion 是2022年发布的深度学习文本到图像生成模型，随着开源社区的活跃增长和 Stable Diffusion WebUI 开源项目的推出，已经成为了 AIGC 场景下首选开源方案，深受广大个人用户欢迎。而随着 GPT-4 的推出和 Stable DIffusion 最新版发布，越来越多企业开始认识到基于大模型的 AGI 和 AIGC 技术已经从“玩具”走向生产力工具，并快速拥抱 AI 技术。然而 AI 模型训练及推理需要消耗大量计算资源，尤其是 GPU 资源。随着 AI 技术的火热落地，GPU 资源也出现一卡难求的场景。对于企业客户来说，低成本获取 GPU 资源，并快速部署 AI 模型，已经成为最紧迫的事情。容器服务 TKE 为客户提供了 GPU 容器托管服务，客户可按需申请 GPU 资源，按秒付费，并快速进行服务扩缩，轻松应对训练推理业务需求波动，并降低整体 GPU 成本。

部署步骤

新建 TKE Serverless 容器集群和超级节点

1. 登录 容器服务控制台 。
2. 新建一个 TKE Serverless 集群，操作步骤详情可参见 创建 TKE Serverless 集群 。在创建 Serverless 集群时，请参考以下提示配置集群：
   - **集群名称**：例如 “Stable Diffusion 体验集群” 。
   - **Kubernetes 版本**：选择默认的 Kubernetes 版本。
   - **所在地域**：建议选择使用上海地域体验，当前 GPU 资源较为充足。
   - **集群网络**：选择已有私有网络 VPC，或新建私有网络。
   - **配置超级节点配置**：
     - **可用区**：建议选择上海五区 ，当前可用区资源较为充足。
     - **计费模式**：选择按量计费，按秒粒度计费，按小时生成账单。
     - **容器网络**：选择可用区内已有的子网，或新建子网。
   - **其他参数**：保持默认值即可。

新建 Stable Diffusion 应用，并配置访问入口

集群创建完成后，在 容器服务控制台 单击集群名称进入集群详情页，并选择左侧导航中的**工作负载**。可以使用 YAML 文件快速部署，也可以使用控制台手动部署，建议使用 YAML 文件方式。

使用 YAML 快速部署

1. 登录 容器服务控制台 ，在集群列表中，单击集群名称。
2. 进入集群详情页，选择左侧导航中的**工作负载**。
3. 单击右上角 **YAML 创建资源**，复制以下内容并粘贴到 YAML 编辑器中：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: stable-diffusion
    qcloud-app: stable-diffusion
  name: stable-diffusion
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: stable-diffusion
      qcloud-app: stable-diffusion
  template:
    metadata:
      annotations:
        eks.tke.cloud.tencent.com/gpu-type: A10*GNV4v
        eks.tke.cloud.tencent.com/root-cbs-size: "40"
```

```
      labels:
        k8s-app: stable-diffusion
        qcloud-app: stable-diffusion
    spec:
      containers:
      - args:
        - --listen
        image: ccr.ccs.tencentyun.com/ai-aigc/stable-diffusion:taco.gpu.v1
        imagePullPolicy: IfNotPresent
        name: stable-diffusion
        resources:
          limits:
            cpu: "28"
            memory: 116Gi
            nvidia.com/gpu: "1"
          requests:
            cpu: "28"
            memory: 116Gi
            nvidia.com/gpu: "1"
        securityContext:
          privileged: false
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: qcloudregistrykey
      restartPolicy: Always
      terminationGracePeriodSeconds: 30

---
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: stable-diffusion
    qcloud-app: stable-diffusion
  name: stable-diffusion
  namespace: default
spec:
  externalTrafficPolicy: Cluster
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: 7860-7860-tcp
    port: 7860
    protocol: TCP
    targetPort: 7860
  selector:
    k8s-app: stable-diffusion
    qcloud-app: stable-diffusion
  sessionAffinity: None
  type: LoadBalancer
```

4. 单击**完成**，即可创建工作负载，并配置访问入口。

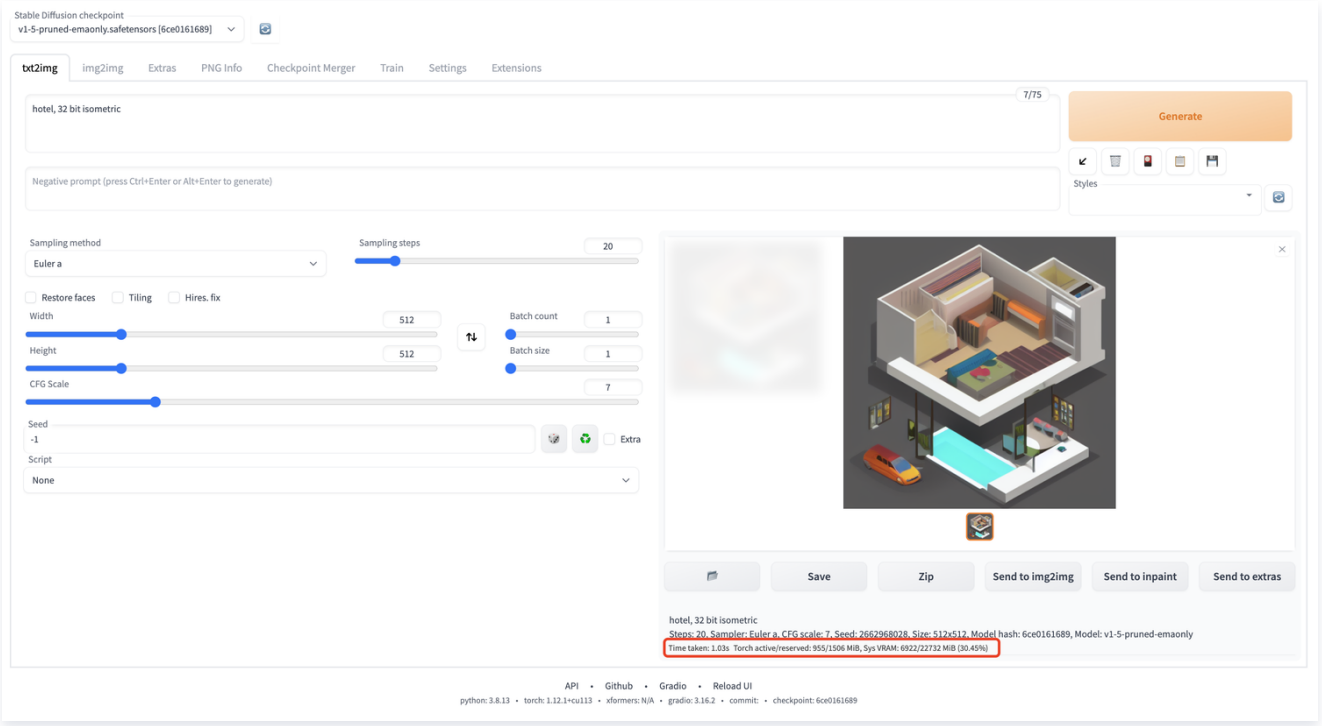
使用控制台手动部署

1. 登录 容器服务控制台，在集群列表中，单击集群名称。

2. 进入集群详情页，选择左侧导航中的**工作负载 > Deployment**，在 Deployment 页面单击**新建**。
3. 在**新建 Deployment 页面**，请参考以下提示进行配置：
   - **名称**：例如 “stable-diffusion-webui”。
   - **Labels**：一个键 – 值对（Key-Value），用于对资源进行分类管理。
   - **命名空间**：选择默认命名空间 Default。
   - **实例类型**：选择 GPU，并选择具体显卡型号，建议选择 A10 显卡，如 A10*GNV4。
   - **系统盘大小**：配置40GiB，因包含模型文件及加速框架，镜像体积较大。
   - **实例内容器**：
     - **名称**：输入容器名称，如 “stable-diffusion-webui ”。
     - **镜像**：直接输入或选择 `ccr.ccs.tencentyun.com/ai-aigc/stable-diffusion` 镜像，镜像版本为 `taco.gpu.v1`。
     - **CPU/内存**：可按照所选的 GPU 自带的 CPU、内存进行配置，如使用 A10*GNV4 显卡可配置最高 CPU 12 核，内存 44GiB = 45056 MiB。
     - **GPU 限制**：调整为1卡，即该容器会使用上面指定的 GPU 型号 * 1。
     - **运行参数**：新增运行参数，输入 `--listen`。
   - **配置访问入口**：
     - **Service**：勾选“启用”。
     - **服务访问方式**：选择**公网 LB 访问**。
     - **配置端口映射**：容器端口设置为7860，服务端口设置为7860。
   - **其他参数**：保持默认值即可。
4. 确认 Pod 规格，单击**创建 Deployment**。如按照上述配置，则为：12核44GiB 1×A10*GNV4、实例数量 ~ 1个、配置费用10.14元/小时（具体费用与登录账号有关，此处为官网刊例价）。
5. 启动部署后，返回**工作负载 > Deployment** 页面，可查看到正在部署的工作负载。因镜像体积较大，预计需要一定时间才能完成部署。当“运行/期望 Pod 数量”变为“1/1”时，即说明完成部署。也可以进入工作负载内，通过事件查看部署日志。

访问 Stable Diffusion WebUI 体验极速出图

1. 在集群左侧导航中选择**服务与路由 > Service**，查看与工作负载名称一致的 Service，复制访问入口的公网 IP，如 175.xx.xxx.174。
2. 打开浏览器，输入 `http:\\175.xx.xxx.174:7860`（替换为实际 IP），即可进入 Stable Diffusion WebUI。
3. 输入 Prompt，单击 **Generate**，即可体验一秒出图的功能。



随时调整服务状态，按需使用，按秒付费

相对于直接使用 GPU 服务器，超级节点底层基于 Serverless 容器技术，支持缩容至0，停止计费。您可以根据实际使用需求，随时调整服务状态，降低使用费用。具体操作步骤如下：

1. 在集群左侧导航中选择**工作负载 > Deployment**，在 stable-diffusion 工作负载对应操作中选择**更新 Pod 数量**。
2. 在**更新 Pod 数量**页面，
   - 暂停服务：将 Pod 实例数量调整为0，对应 GPU 资源自动销毁，停止计费。
   - 恢复服务：将 Pod 实例数量调整为1~N，无需重新配置，实例自动重新部署，开始计费。

相关文档

- TKE Serverless - 新建集群
- TKE Serverless - 工作负载管理
- TACO Infer 部署 Stable Diffusion web UI

TKE Serverless 运行 ChatGLM-6B 微调

最近更新时间：2024-01-09 10:18:41

背景说明

ChatGLM-6B 是一款拥有 62 亿参数的中英双语语言模型，专注于提供强大的对话和问答能力。通过对1:1中英语料进行 1T token 的预训练，使得模型具备双语应用的能力。借助模型量化技术，ChatGLM-6B 可以高效地在消费级显卡上部署。其最大序列长度达到2048，以应对更复杂的对话和应用需求。总体而言，ChatGLM-6B 是一款功能强大、适用广泛的语言模型。
ChatGLM-6B 也支持微调，针对特定任务或领域进行进一步训练，使得在特定任务上可以达到更好的性能。在本文中，我们将介绍下如何基于 TKE Serverless 集群，利用官方示例中提供的 P-Tuning v2 方法进行微调。

部署步骤

本文所展示的部署方案基于单张 V100显卡进行。

部署前提

- 创建 TKE Serverless 集群，操作详情请参见 创建 Serverless 集群 。
- 创建 TCR 镜像仓库（个人版或者企业版），操作详情请参见 个人版快速入门 。
- 开通 CFS 文件存储，操作详情请参见 创建文件系统及挂载点 。

镜像准备

以 TCR 个人版镜像为例，并且使用的示例镜像命名为 `ccr.ccs.tencentyun.com/chatglm/chatglm-6b-ptv2:v1.0` 。

> **注意：**
> 在实际操作中，请将其替换为您自己的镜像名称。

接下来，我们将详细说明如何配置镜像的操作示例。

1. 代码下载

通过 Git 下载示例代码。执行以下命令：

```
git clone $ git clone https://github.com/coderwangke/tke-run-
chatglm.githttps://github.com/coderwangke/tke-run-chatglm.git
```

2. 制作镜像

进入 `tke-run-chatglm/finetune` 目录，并使用 `docker` 命令制作镜像。执行以下命令：

```
cd tke-run-chatglm/finetune

docker build -f Dockerfile.V100 -t ccr.ccs.tencentyun.com/chatglm/chatglm-6b-ptv2:v1.0 .
```

该镜像将基于示例中的 Dockerfile.V100 文件创建。

3. 上传镜像

登录 容器镜像服务控制台 个人版镜像仓库，并将制作的镜像推送到仓库中。执行以下命令：

```
# 登录 TCR 个人版镜像仓库，请根据提示输入用户名和密码
docker login ccr.ccs.tencentyun.com

docker push ccr.ccs.tencentyun.com/chatglm/chatglm-6b-ptv2:v1.0
```

> **注意：**

> 在运行 `docker login` 时，请根据提示输入您的用户名和密码。

通过以上步骤，您将能够下载示例代码、制作镜像，并将镜像上传至个人版 TCR 镜像仓库。

存储准备

为了存储模型、数据、checkpoint 等文件，本文将使用腾讯云文件存储 CFS 作为示例。

目录划分

我们将使用以下目录结构进行存储：

- models：用于存放模型数据，例如 ChatGLM-6B。
- datasets：用于存储训练数据。
- output：用于存放 checkpoint 等输出。

下载模型

在微调过程中，可以从 Hugging Face 仓库自动下载模型，但下载过程可能较慢，造成 GPU 资源的浪费。因此，建议预先将模型下载至 CFS 文件存储中的 `/models` 目录下。以下步骤基于 CVM 服务器执行：

1. 安装依赖。

   以 TencentOS 为例，执行以下命令来安装依赖：

   ```
   # 以 TencentOS 为例
   yum update && yum install git git-lfs -y
   ```

2. 挂载CFS文件存储（假设挂载点为 `/data`）。

   执行以下命令来挂载 CFS 文件存储到 `/data` 目录：

   ```
   sudo mount -t nfs -o vers=3,nolock,proto=tcp,noresvport xxx.xxx.xxx.xxx:/xxxx/ /data
   ```

3. 创建 models 子目录并下载模型。

   执行以下命令来创建 /data/models 子目录并下载模型：

   ```
   cd /data && mkdir -p models
   cd models

   git lfs install
   git clone https://huggingface.co/THUDM/chatglm-6b
   ```

上传训练数据

本文使用 ADGEN（广告生成）数据集进行微调，数据集包括训练数据集 `train.json` 和验证数据集 `dev.json`，需要提前存放到 CFS 文件存储中的 `/datasets` 目录下。

> **说明：**
> 您可以从 Tsinghua Cloud 下载处理好的 ADGEN 数据集。

创建存储

CFS 文件存储通过 PV/PVC 的方式挂载使用。以下示例以 CFS 标准型为例。

安装组件

1. 登录 容器服务控制台，在左侧导航栏中选择**集群**。
2. 在集群列表中，单击目标集群 ID，进入集群详情页。
3. 选择左侧菜单栏中的**组件管理**，在组件管理页面单击**新建**。
4. 在**新建组件管理**页面中勾选 **CFS（腾讯云文件存储）**。

5. 单击**完成**。

创建 PV

1. 在集群列表中，单击集群 ID，进入集群详情页。
2. 选择左侧菜单栏中的**存储** > **PersistentVolume**，在 PersistentVolume 页面单击**新建**。
3. 在**新建 PersistentVolume** 页面中，配置 PV 关键参数。如下图所示：

配置项	描述
来源设置	选择**静态创建**。
名称	填写 cfs-pv。
Provisioner	选择**文件存储 CFS**。
读写权限	文件存储仅支持多机读写。
是否指定 StorageClass	选择**不指定 StorageClass**。
选择 CFS	请选择要挂载的 CFS ID。
CFS 子目录	请根据实际情况填写，默认是 `/` 。

4. 单击**创建 PersistentVolume**。

创建 PVC

1. 在集群列表中，单击集群 ID，进入集群详情页。
2. 选择左侧菜单栏中的**存储** > **PersistentVolumeClaim**，在 PersistentVolumeClaim 页面单击**新建**。
3. 在**新建 PersistentVolumeClaim** 页面中，配置 PVC 关键参数。如下图所示：

配置项	描述
名称	填写 cfs-pvc。
命名空间	请选择微调任务运行的命名空间。
Provisioner	选择文件存储 CFS。
读写权限	文件存储仅支持多机读写。
是否指定 StorageClass	选择**不指定 StorageClass**。
是否指定 PersistVolume	选择**指定**。
PersistentVolume	选择 创建 PV 步骤中创建的 PV。

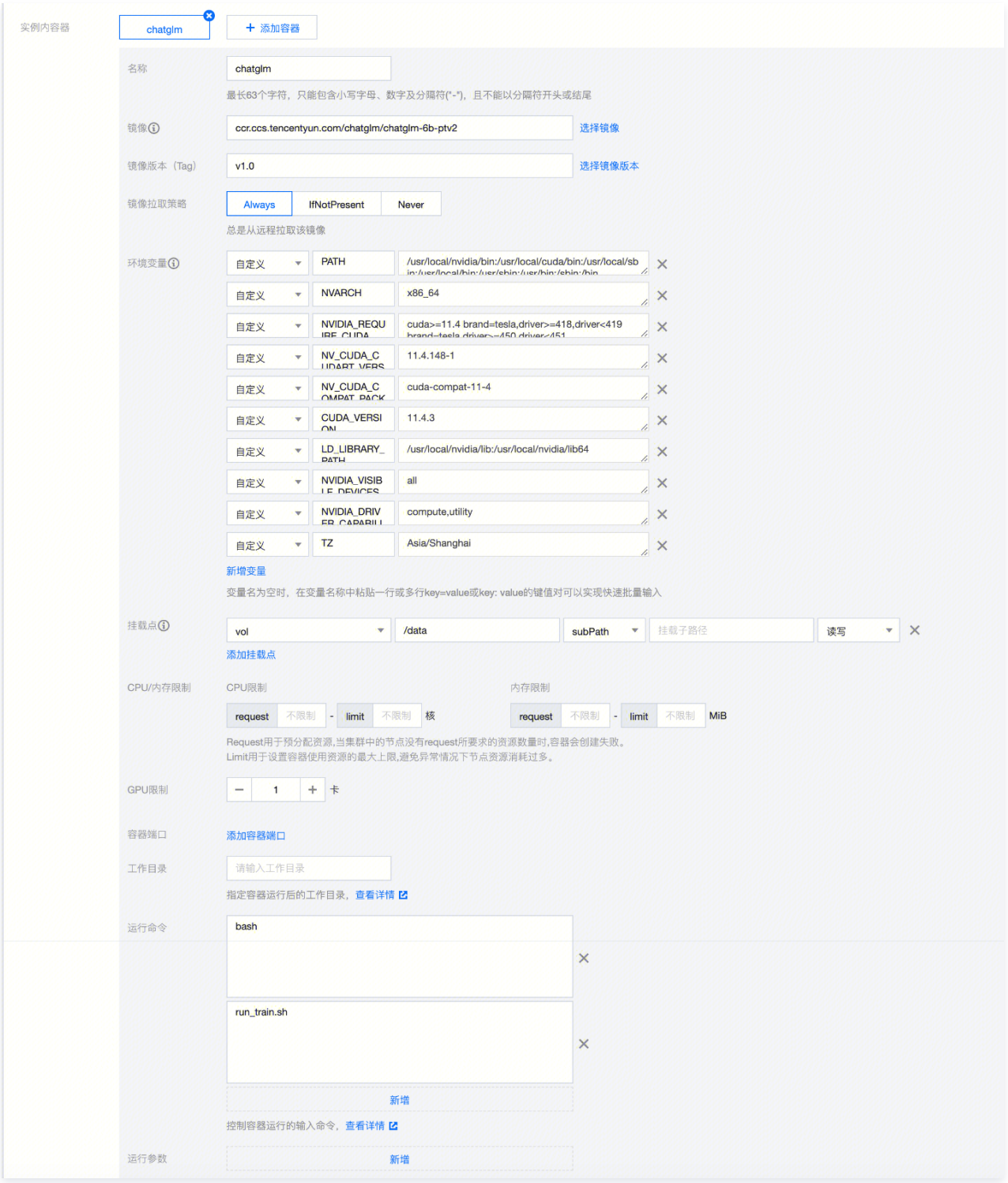
4. 单击**创建 PersistentVolumeClaim**。

创建微调应用

1. 登录 容器服务控制台，在左侧导航栏中选择**集群**。
2. 在集群列表中，单击目标集群 ID，进入集群详情页。
3. 选择左侧菜单栏中的**工作负载 > Job**，在 Job 页面单击**新建**。
4. 在**新建 Job** 页面，设置 Job 参数。关键参数信息如下：

配置项	描述
实例类型	选择 GPU，和 V100。
系统盘大小	推荐调大到 50GiB。
数据卷	选择 创建 PVC 步骤中创建的 PVC。

在实例内容器中，设置参数信息，如下图所示：



配置项	描述
名称	自定义名称。
镜像	填写 **镜像准备** 阶段制作的 docker 镜像名称。
挂载点	设置挂载的目标路径为 `/data`。
CPU/内存限制	由于微调的资源需要主要是 GPU，因此 CPU 和内存可以按需设置，示例设置为空。
GPU 限制	设置为1个。

运行命令	分别填写 `bash`、`run_train.sh`。

在 TKE 上运行基于 NCCL 的 RDMA 分布式训练任务

最近更新时间：2025-03-10 17:11:32

简介

RDMA（Remote Direct Memory Access，远程直接内存访问）是一种高速网络互联技术，该技术旨在减少在数据传输过程中收发端的处理延迟以及资源消耗。RDMA 技术使计算机能够直接访问远程计算机的内存，在内存层面进行数据传输而无需 CPU 频繁介入，从而显著增强网络通信性能。使用 RDMA，能够提高网络吞吐，降低网络时延，从而有效提升大规模分布式训练任务的训练效率。
本文档介绍在 TKE 上如何使用基于 NCCL 的 RDMA 运行分布式训练任务。

环境准备

1. 已创建并部署好 TKE 集群。如果您还没有集群，请参见 创建集群 。
2. 已购买 RDMA 实例，RDMA netns 为 `shared` 模式，且加入的 高性能计算集群（THCC）已支持 /28 及以上 RDMA 网段。
3. 部署安装 rdma-agent 组件。可以在 容器服务控制台 的**集群 > 组件管理**进行安装，组件安装详情请参见 rdma-agent 说明 。

4. 将欲部署的 RDMA 实例节点加入节点池，修改节点池配置添加 label 从而部署 rdma-agent。操作详情请参见 部署节点设置 label 。

环境验证

1. 组件部署后，等待安装成功，确保 tke-rdma-shared-agent 皆为 Running。

```
kubectl -nkube-system get po -o wide -lk8s-app=tke-rdma-shared-agent
```

2. 检查 RDMA 节点 Capacity 和 Allocatable 资源，确保其中的 `tke.cloud.tencent.com/tke-shared-rdma` 不为 0。

```
kubectl describe node <nodeName> | grep tke-shared-rdma
```

```
tke.cloud.tencent.com/tke-shared-rdma:  1k
tke.cloud.tencent.com/tke-shared-rdma:  1k
tke.cloud.tencent.com/tke-shared-rdma   2             2
```

提交训练任务

YAML 关键配置

1. 网络配置。

   指定 CNI，RDMA 网络平面和 VPC 网络平面是两个网络平面，为了使 Pod 能够同时连接 VPC 网络和 RDMA 网络，需要在工作负载的 Pod template 中指定以下 annotation 配置：

   - GlobalRouter（全局路由）网络模式集群：

     ```
     tke.cloud.tencent.com/networks: "tke-bridge,tke-rdma-ipvlan"
     ```

   - VPC-CNI 共享网卡模式集群：

     ```
     tke.cloud.tencent.com/networks: "tke-route-eni,tke-rdma-ipvlan"
     ```

   集群的网络模式可以通过集群里的 cni-agent 配置来确定：

   ```
   kubectl -nkube-system get cm tke-cni-agent-conf -oyaml | grep defaultDelegates
   ```

   如全局路由模式集群的配置：

   ```
   apiVersion: v1
   data:
     00-multus.conf: |2

           {
             "cniVersion": "0.3.1",
             "name": "multus-cni",
             "type": "multus",
             "kubeconfig": "/etc/kubernetes/tke-cni-kubeconfig",
             "logLevel": "info",
             "defaultDelegates": "tke-bridge",
             "capabilities": {
               "bandwidth": true,
               "portMappings": true
             }
           }
   kind: ConfigMap
   metadata:
   ```

2. 指定 securityContext 使能进程间通信，设置 RDMA 和 GPU resource 资源请求。

   ```
           securityContext:
             capabilities:
               add: [ "IPC_LOCK" ]
           resources:
             limits:
               nvidia.com/gpu: "8"
               tke.cloud.tencent.com/tke-shared-rdma: 1
   ```

3. 为了支持 NCCL 共享数据，需要为 IPC 和固定（页面锁定）系统内存资源共享系统内存。

   ```
           volumeMounts:
           - mountPath: /dev/shm
             name: dshm
         volumes:
         - emptyDir:
             medium: Memory
             sizeLimit: 64Gi
           name: dshm
   ```

YAML 样例

以 statefulset 为例，具体的样例如下：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    k8s-app: mofed-test
  name: mofed-test
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      k8s-app: mofed-test
      qcloud-app: mofed-test
  serviceName: ""
  template:
    metadata:
      annotations:
        tke.cloud.tencent.com/networks: "tke-bridge,tke-rdma-ipvlan"
      creationTimestamp: null
      labels:
        k8s-app: mofed-test
        qcloud-app: mofed-test
    spec:
      containers:
      - command:
        - sh
        - -c
        - |
          ls -l /dev/infiniband /sys/class/infiniband /sys/class/net
          sleep 1000000
        image: ccr.ccs.tencentyun.com/tke-eni-test/mofed-5.4-3.1.0.0:ubuntu20.04-amd64-test
        imagePullPolicy: Always
        name: mofed-test-ctr
        securityContext:
          capabilities:
            add: [ "IPC_LOCK" ]
        resources:
          limits:
            nvidia.com/gpu: "8"
            tke.cloud.tencent.com/tke-shared-rdma: 1
        volumeMounts:
        - mountPath: /dev/shm
          name: dshm
      volumes:
      - emptyDir:
          medium: Memory
          sizeLimit: 64Gi
        name: dshm
```

注意事项：获取 gid

在基于 NCCL 的训练任务中，`NCCL_IB_GID_INDEX` 环境变量是关键配置，需要配置为 RDMA 设备分配到的 GID。由于主机侧的网卡和容器侧网卡会同时运行，**容器内的 GID 不是固定值3，且同一个节点上的每个 Pod 各不相同，需要为每个 Pod 单独获取和设置。**RDMA 设备的 GID 可以通过 `show_gids` 命令获取。

请注意，只有在 NCCL 2.21 版本以前需要设置这个变量，而在 2.21 版本及以后，该值会被动态设置，不需要设置，详情请参见 NCCL 库官网文档 。

测试验证

部署测试用 Statefulset，可以部署上述样例中的 mofed-test。Pod 创建以后，可以登录 Pod 内进行测试：

```
kubectl exec -it mofed-test-0 -- bash
```

基础功能测试

使用 `ibv_devinfo` 命令确认 RDMA 设备功能正常：

```
# ibv_devinfo
hca_id: mlx5_bond_0
    transport:          InfiniBand (0)
    fw_ver:             22.33.1048
    node_guid:          946d:ae03:00a9:3ed0
    sys_image_guid:         946d:ae03:00a9:3ed0
    vendor_id:          0x02c9
    vendor_part_id:         4125
    hw_ver:             0x0
    board_id:           MT_0000000359
    phys_port_cnt:          1
        port:   1
            state:          PORT_ACTIVE (4)
            max_mtu:        4096 (5)
            active_mtu:     4096 (5)
            sm_lid:         0
            port_lid:       0
            port_lmc:       0x00
            link_layer:     Ethernet

hca_id: mlx5_bond_1
    transport:          InfiniBand (0)
    fw_ver:             22.33.1048
    node_guid:          946d:ae03:00ab:d6dc
    sys_image_guid:         946d:ae03:00ab:d6dc
    vendor_id:          0x02c9
    vendor_part_id:         4125
    hw_ver:             0x0
    board_id:           MT_0000000359
    phys_port_cnt:          1
        port:   1
            state:          PORT_ACTIVE (4)
            max_mtu:        4096 (5)
            active_mtu:     4096 (5)
            sm_lid:         0
            port_lid:       0
            port_lmc:       0x00
            link_layer:     Ethernet
...
```

使用 `show_gids` 命令获取 gid，这里可以看到对应的 gid 为 7：

```
# show_gids
DEV PORT    INDEX   GID                 IPv4        VER DEV
--- ----    -----   ---                 ------------    --- ---
mlx5_bond_0 1   4   fe80:0000:0000:0000:401d:55ff:fe14:6ce8         v1  bond0
mlx5_bond_0 1   5   fe80:0000:0000:0000:401d:55ff:fe14:6ce8         v2  bond0
mlx5_bond_0 1   6   0000:0000:0000:0000:0000:ffff:1e3d:b4ac 30.61.180.172   v1  bond0
mlx5_bond_0 1   7   0000:0000:0000:0000:0000:ffff:1e3d:b4ac 30.61.180.172   v2  bond0
```

```
mlx5_bond_1 1    4    fe80:0000:0000:0000:0c0d:00ff:fe48:33aa          v1   bond1
mlx5_bond_1 1    5    fe80:0000:0000:0000:0c0d:00ff:fe48:33aa          v2   bond1
mlx5_bond_1 1    6    0000:0000:0000:0000:0000:ffff:1e3d:b4bc 30.61.180.188    v1   bond1
mlx5_bond_1 1    7    0000:0000:0000:0000:0000:ffff:1e3d:b4bc 30.61.180.188    v2   bond1
...
```

连通性测试

使用 `ibv_rc_pingpong` 命令测试 RDMA 连通性：

- Server 端配置：

```
# ibv_rc_pingpong -d mlx5_bond_0 -g 7
  local address:  LID 0x0000, QPN 0x003a22, PSN 0xc882ae, GID ::ffff:30.61.130.20
  remote address: LID 0x0000, QPN 0x001234, PSN 0x58320b, GID ::ffff:30.61.180.172
8192000 bytes in 0.01 seconds = 7687.51 Mbit/sec
1000 iters in 0.01 seconds = 8.52 usec/iter
```

- Client 端配置，注意 Client 端和 Server 的 gid 不一定相同。

```
# ibv_rc_pingpong -d mlx5_bond_0 -i 1 30.61.130.20 -g 7
  local address:  LID 0x0000, QPN 0x001234, PSN 0x58320b, GID ::ffff:30.61.180.172
  remote address: LID 0x0000, QPN 0x003a22, PSN 0xc882ae, GID ::ffff:30.61.130.20
8192000 bytes in 0.01 seconds = 7924.55 Mbit/sec
1000 iters in 0.01 seconds = 8.27 usec/iter
```

带宽测试

使用 `ib_write_bw` 命令测试 RDMA 带宽：

- Server 端配置：

```
# ib_write_bw -d mlx5_bond_0 -x 7

************************************
* Waiting for client to connect... *
************************************


---------------------------------------------------------------------------------------
                    RDMA_Write BW Test
 Dual-port       : OFF          Device         : mlx5_bond_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : Ethernet
 GID index       : 7
 Max inline data : 0[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------------------
 local address: LID 0000 QPN 0x3a25 PSN 0x8877cd RKey 0x00ca88 VAddr 0x007fb84630d000
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:30:61:130:20
 remote address: LID 0000 QPN 0x1235 PSN 0xe2acbc RKey 0x007c3e VAddr 0x007f8f164b7000
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:30:61:180:172
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[MB/sec]    BW average[MB/sec]   MsgRate[Mpps]
 65536      5000             97.50              97.49          0.185950
---------------------------------------------------------------------------------------
```

- Client 端配置：

```
# ib_write_bw -d mlx5_bond_0 -x 7 30.61.130.20 --report_gbits
---------------------------------------------------------------------------------------
                    RDMA_Write BW Test
 Dual-port       : OFF      Device         : mlx5_bond_0
 Number of qps   : 1        Transport type : IB
 Connection type : RC       Using SRQ      : OFF
 TX depth        : 128
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : Ethernet
 GID index       : 7
 Max inline data : 0[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------------------
 local address: LID 0000 QPN 0x1235 PSN 0xe2acbc RKey 0x007c3e VAddr 0x007f8f164b7000
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:30:61:180:172
 remote address: LID 0000 QPN 0x3a25 PSN 0x8877cd RKey 0x00ca88 VAddr 0x007fb84630d000
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:30:61:130:20
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]   MsgRate[Mpps]
 65536      5000             97.50              97.49          0.185950
---------------------------------------------------------------------------------------
```

NCCL 测试

准备工作

NCCL 测试需使用带有 NCCL 相关库的测试镜像，可使用如下样例 YAML：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    k8s-app: nccl-test
  name: nccl-test
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      k8s-app: nccl-test
      qcloud-app: nccl-test
  serviceName: ""
  template:
    metadata:
      annotations:
        tke.cloud.tencent.com/networks: "tke-bridge,tke-rdma-ipvlan"
      creationTimestamp: null
      labels:
        k8s-app: nccl-test
        qcloud-app: nccl-test
    spec:
      containers:
      - command:
        - sh
        - -c
        - |
```

```
          ls -l /dev/infiniband /sys/class/infiniband /sys/class/net
          sleep 1000000
        image: ccr.ccs.tencentyun.com/qcloud-ti-platform/nccl_ofed:cu12.0
        imagePullPolicy: IfNotPresent
        name: nccl-test
        securityContext:
          capabilities:
            add: [ "IPC_LOCK" ]
        resources:
          limits:
            nvidia.com/gpu: "8"
            tke.cloud.tencent.com/tke-shared-rdma: 1
        volumeMounts:
        - mountPath: /dev/shm
          name: dshm
      volumes:
      - emptyDir:
          medium: Memory
          sizeLimit: 64Gi
        name: dshm
```

测试步骤

1. 创建部署测试 Pod。按照上述 yaml 部署生成了两个 Pod：nccl-test-0，nccl-test-1。
2. 配置 ssh 免密登录。

   mpirun 需要通过 ssh 登录到其他节点执行相关命令，所以需要配置免密登录，首先在 nccl-test-0 上执行：

   ```
   # 生成密钥
   ssh-keygen
   # 查看公钥
   cat ~/.ssh/id_rsa.pub
   ```

   然后在 nccl-test-1 上执行下述命令，配置免密登录：

   ```
   mkdir -p /run/sshd
   # 启动 sshd
   /usr/sbin/sshd
   mkdir -p ~/.ssh
   # <id_rsa.pub> 为 nccl-test-0 Pod 里 /root/.ssh/id_rsa.pub 的文件内容
   echo "<id_rsa.pub>" >> ~/.ssh/authorized_keys
   ```

   在 nccl-test-0 上测试是否可以免密登录 nccl-test-1，假设 nccl-test-0 的 ip 为 172.21.4.8，nccl-test-1 的 ip 为 172.21.5.6：

   ```
   ssh root@172.21.5.6
   ```

3. 启动 nccl 测试任务。

   在 nccl-test-0 上执行 `mpirun` 可启动 nccl all_reduce_perf 测试任务：

   ```
   mpirun --allow-run-as-root -np 16 \
   -npernode 8 -H 172.21.4.8:8,172.21.5.6:8 \
   -mca btl_tcp_if_include bond0 \
   -x NCCL_IB_DISABLE=0 \
   -x NCCL_IB_GID_INDEX=5 \
   -x NCCL_DEBUG=INFO \
   /root/nccltest/all_reduce_perf -b 1G -e 1G -f 2 -g 1 -n 20
   ```

   部分参数说明：

- **btl_tcp_if_include**：指定 openmpi 运行时通信使用的网卡，由于网络环境同时有多个网段（每个 RDMA 独属一个网段，eth0 一个网段），所以需要指定只使用 bond0 或者 eth0 进行通信。若该变量不设置会出现报错：`Open MPI failed to TCP connect to a peer MPI process.`
- **NCCL_IB_DISABLE**：0 代表 NCCL 使用的 IB/RoCE 传输，1 代表禁止 NCCL 使用的 IB/RoCE 传输。此时 NCCL 将退回到使用IP套接字。
- **NCCL_IB_GID_INDEX**：指定 RoCE 模式中使用的全局 ID 索引。获取方法请参考本文中的 注意事项：获取 gid 。只有在 NCCL 2.21 版本以前需要设置这个变量，而在 2.21 版本及以后，该值会被动态设置，不需要设置。
- **NCCL_DEBUG**：NCCL_DEBUG 变量控制从 NCCL 显示的调试信息。

以上测试任务成功之后的回显结果类似为：

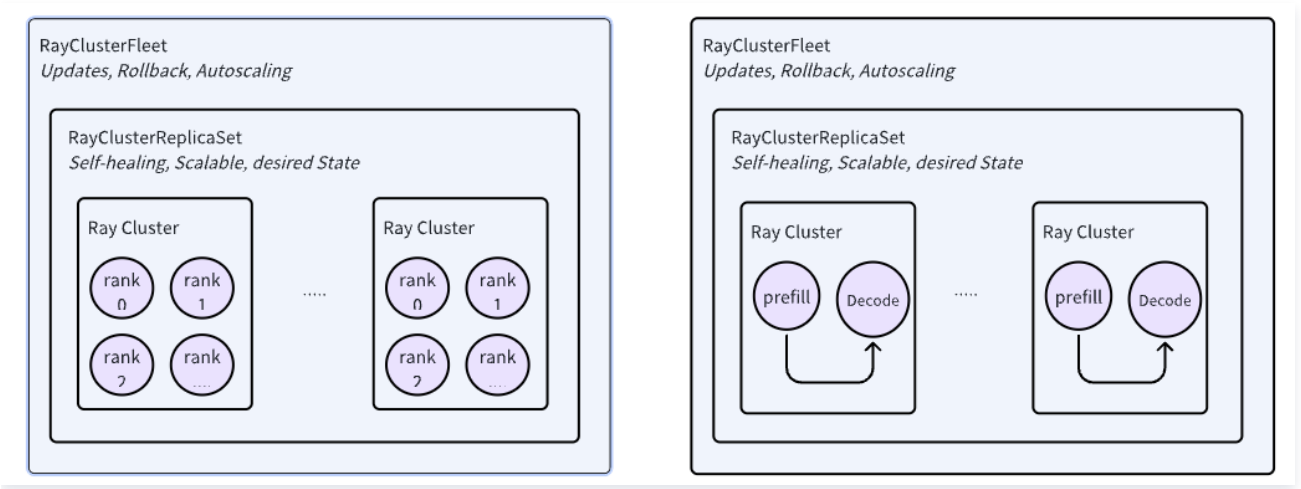
```
#
#                                                              out-of-place                       in-
place
#       size         count      type   redop    root     time   algbw   busbw #wrong     time   algbw
busbw #wrong
#        (B)    (elements)                               (us)  (GB/s)  (GB/s)            (us)  (GB/s)
(GB/s)
  1073741824     268435456     float     sum      -1   6455.8  166.32  311.85      0   6426.3  167.09
313.29      0
# Out of bounds values : 0 OK
# Avg bus bandwidth    : 312.571
#
```

在 TKE 上使用 AIBrix 进行多节点分布式推理

最近更新时间：2025-04-11 09:18:22

概述

AIBrix 是在2025年2月开源的云原生大模型推理控制平面项目，专为优化大规模语言模型（LLM）的生产化部署设计。作为首个深度集成 vLLM 的 Kubernetes 全栈方案，它提供了 LoRA 动态加载、多节点推理、异构 GPU 调度、分布式 KV 缓存等多种核心特性。
分布式推理是指在多个节点或设备上拆分和处理 LLM 模型的技术，这种方法对于单台机器内存无法容纳的大型模型尤其有用。AIBrix 采用 Ray 作为其分布式计算框架，结合 KubeRay 来协调 Ray 集群实现了分布式推理技术。
AIBrix 为管理 RayCluster 引入了两个关键 API，即 RayClusterReplicaSet 和 RayClusterFleet。RayClusterFleet 管理 RayClusterReplicaSet，RayClusterReplicaSet 管理 RayCluster，三者之间的关系与 Kubernetes 的核心概念 Deployment、ReplicaSet、Pod 之间类似，大部分情况下，用户只需要使用 RayClusterFleet。



aibrix-multi-host

在本文中，我们将介绍如何在 TKE 集群上使用 AIBrix 进行分布式推理。

> **镜像说明：**
> 本文示例中所使用的镜像为 vllm/vllm-openai，托管在 DockerHub 上，且体积较大（8GB左右）。
> 在 TKE 环境中，默认会提供免费的 DockerHub 镜像加速服务，因此中国大陆用户也可以直接拉取到镜像，但速度可能较慢。建议将镜像同步至 容器镜像服务 TCR 中提高镜像拉取速度，并在 YAML 文件中替换相应的镜像地址。

操作步骤

1. 创建 TKE 集群

登录 容器服务控制台，参考步骤 创建集群，创建一个 TKE 集群。

- **集群类型：**TKE 标准集群。
- **Kubernetes 版本：**需要大于等于 1.28，建议选择最新版本。本文使用的是 1.30。
- **基础配置：**存储组件需勾选 CFS，如下图所示：

2. 创建超级节点

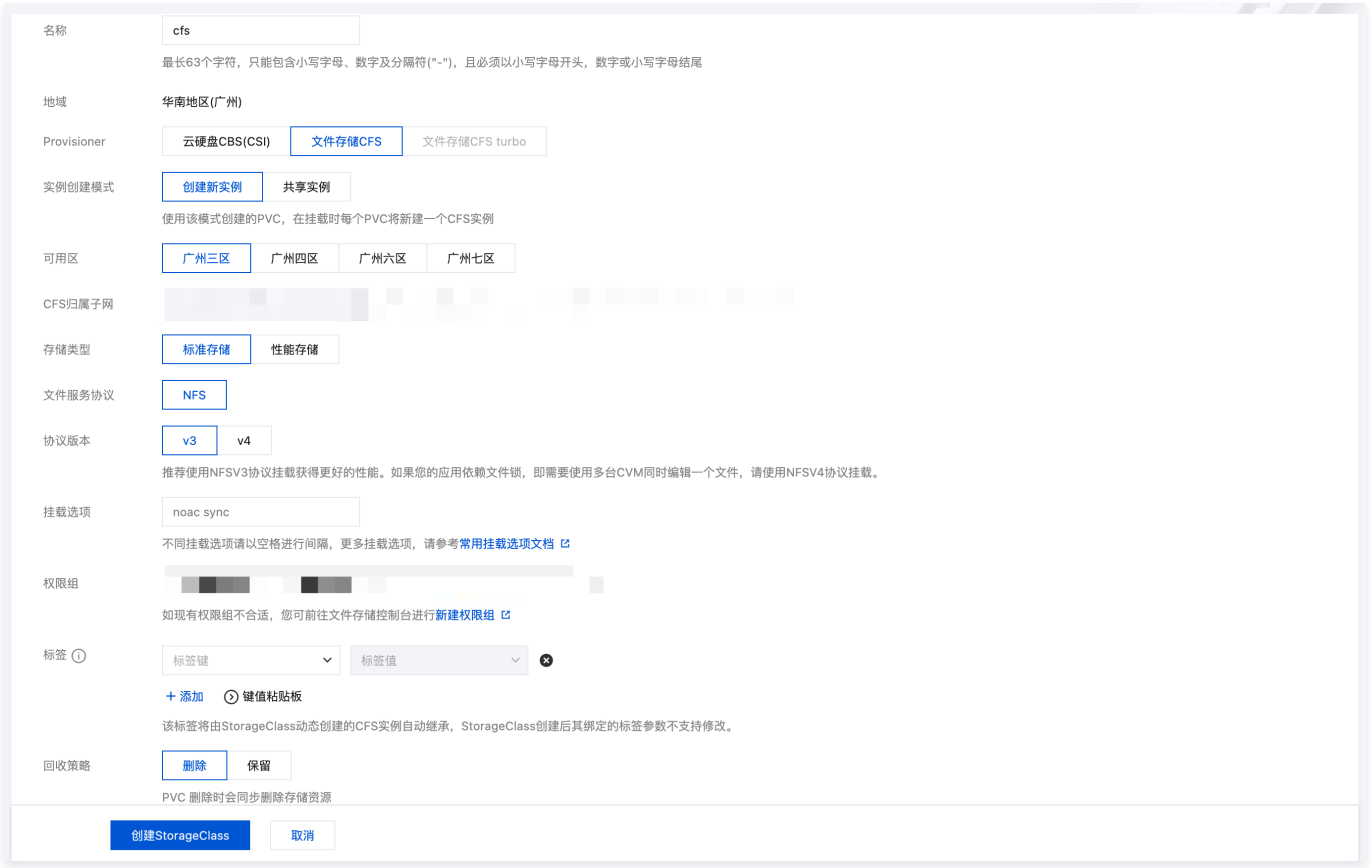
在集群列表中，单击**集群 ID**，进入集群详情页，参考步骤 **创建超级节点**，创建一个超级节点池。

3. 下载模型

3.1 创建 StorageClass

通过控制台创建 StorageClass：

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧导航中的**存储**，在 StorageClass 页面单击**新建**。
3. 在新建存储页面，根据实际需求，创建 CFS 类型的 StorageClass。如下图所示：



3.2 创建 PVC

通过控制台创建 PVC：

1. 在集群列表中，单击**集群 ID**，进入集群详情页。
2. 选择左侧导航中的**存储**，在 PersistentVolumeClaim 页面单击**新建**。
3. 在新建存储页面，根据实际需求，创建存储模型文件的 PVC。如下图所示：

3.3 使用 Job 下载模型文件

创建一个 Job 用于下载大模型文件到 CFS。

> **说明：**
> 本文示例中所用模型为 Qwen2.5-Coder 的 7B 版本。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: download-model
  labels:
    app: download-model
spec:
  template:
    metadata:
      name: download-model
      labels:
        app: download-model
      annotations:
        eks.tke.cloud.tencent.com/root-cbs-size: "100" # 超级节点系统盘默认只有 20Gi，vllm 镜像解压后会撑爆磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
    spec:
      containers:
        - name: vllm
          image: vllm/vllm-openai:v0.7.1
          command:
            - modelscope
            - download
            - --local_dir=/data/model/Qwen2.5-Coder-7B-Instruct
            - --model=Qwen/Qwen2.5-Coder-7B-Instruct
          volumeMounts:
            - name: data
              mountPath: /data/model
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: ai-model # 此处为创建的 PVC 名称
      restartPolicy: OnFailure
```

4. 安装 AIBrix

参考 AIBrix 的官方文档 Installation | AIBrix，安装 AIBrix。

```
# Install component dependencies
kubectl create -f https://github.com/vllm-project/aibrix/releases/download/v0.2.1/aibrix-dependency-
v0.2.1.yaml

# Install aibrix components
kubectl create -f https://github.com/vllm-project/aibrix/releases/download/v0.2.1/aibrix-core-v0.2.1.yaml
```

检查 AIBrix 安装情况，确认所有 Pod 都处于 Running 状态。

```
kubectl -n aibrix-system get pods
```

5. 部署模型

创建 RayClusterFleet 部署 Qwen2.5-Coder-7B-Instruct 模型。

```
apiVersion: orchestration.aibrix.ai/v1alpha1
kind: RayClusterFleet
metadata:
  labels:
    app.kubernetes.io/name: aibrix
    app.kubernetes.io/managed-by: kustomize
  name: qwen-coder-7b-instruct
spec:
  replicas: 1
  selector:
    matchLabels:
      model.aibrix.ai/name: qwen-coder-7b-instruct
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        model.aibrix.ai/name: qwen-coder-7b-instruct
      annotations:
        ray.io/overwrite-container-cmd: "true"
    spec:
      rayVersion: "2.10.0" # 必须匹配容器内的 Ray 版本
      headGroupSpec:
        rayStartParams:
          dashboard-host: "0.0.0.0"
        template:
          metadata:
            annotations:
              eks.tke.cloud.tencent.com/gpu-type: V100 # 指定 GPU 卡型号
              eks.tke.cloud.tencent.com/root-cbs-size: '100' # 超级节点系统盘默认只有 20Gi，vllm 镜像解压后会撑
爆磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
          spec:
            containers:
              - name: ray-head
                image: vllm/vllm-openai:v0.7.1
                ports:
                  - containerPort: 6379
                    name: gcs-server
                  - containerPort: 8265
```

```
                name: dashboard
              - containerPort: 10001
                name: client
              - containerPort: 8000
                name: service
            command: ["/bin/bash", "-lc", "--"]
            args:
              - |
                ulimit -n 65536;
                echo head;
                $KUBERAY_GEN_RAY_START_CMD & KUBERAY_GEN_WAIT_FOR_RAY_NODES_CMDS;
                vllm serve /data/model/Qwen2.5-Coder-7B-Instruct \
                  --served-model-name Qwen/Qwen2.5-Coder-7B-Instruct \
                  --tensor-parallel-size 2 \
                  --distributed-executor-backend ray \
                  --dtype=half
            resources:
              limits:
                cpu: "4"
                nvidia.com/gpu: 1
              requests:
                cpu: "4"
                nvidia.com/gpu: 1
            volumeMounts:
              - name: data
                mountPath: /data/model
        volumes:
          - name: data
            persistentVolumeClaim:
              claimName: ai-model # 此处为创建的 PVC 名称
    workerGroupSpecs:
      - replicas: 1
        minReplicas: 1
        maxReplicas: 5
        groupName: small-group
        rayStartParams: {}
        template:
          metadata:
            annotations:
              eks.tke.cloud.tencent.com/gpu-type: V100 # 指定 GPU 卡型号
              eks.tke.cloud.tencent.com/root-cbs-size: '100' # 超级节点系统盘默认只有 20Gi，vllm 镜像解压后会
撑爆磁盘，用这个注解自定义一下系统盘容量（超过20Gi的部分会收费）。
          spec:
            containers:
              - name: ray-worker
                image: vllm/vllm-openai:v0.7.1
                command: ["/bin/bash", "-lc", "--"]
                args:
                  ["ulimit -n 65536; echo worker; $KUBERAY_GEN_RAY_START_CMD"]
                lifecycle:
                  preStop:
                    exec:
                      command: ["/bin/sh", "-c", "ray stop"]
                resources:
                  limits:
                    cpu: "4"
                    nvidia.com/gpu: 1
                  requests:
                    cpu: "4"
                    nvidia.com/gpu: 1
```

```
                volumeMounts:
                  - name: data
                    mountPath: /data/model
            volumes:
              - name: data
                persistentVolumeClaim:
                  claimName: ai-model # 此处为创建的 PVC 名称
```

6. 验证 API

当 RayClusterFleet 部署的 Pod 成功运行以后，可以通过 kubectl port-forward 快速验证 API。

```
# 获取 service 名称
svc=$(kubectl get svc -o name | grep qwen-coder-7b-instruct)

# 使用 port-forward 功能暴露 API 到本地的 18000 端口
kubectl port-forward $svc 18000:8000

# 启动另外一个终端，运行如下命令测试 API
curl -X POST "http://localhost:18000/v1/chat/completions" \
-H "Content-Type: application/json" \
-d '{
  "model": "Qwen/Qwen2.5-Coder-7B-Instruct",
  "messages": [
    {"role": "system", "content": "你是一个AI编程助手"},
    {"role": "user", "content": "用 Python 实现快速排序算法"}
  ],
  "temperature": 0.3,
  "max_tokens": 512,
  "top_p": 0.9
}'
```

常见问题

aibrix-kuberay-operator 无法启动，报错

`runtime/cgo: pthread_create failed: Operation not permitted`

检查 aibrix-kuberay-operator 是否部署在超级节点上，如果 aibrix-kuberay-operator 部署在超级节点上，参见如下两种解决方法：

1. 修改 aibrix-kuberay-operator 的 Deployment，在 Pod Template 中增加以下注解：

```
eks.tke.cloud.tencent.com/cpu-type: intel # 指定 CPU 类型为 intel
```

2. 参考 设置工作负载的调度规则，将 aibrix-kuberay-operator 调度到普通节点上。

如何设置 API 密钥限制访问?

vLLM 提供了以下两种方式设置 API 密钥：

1. 设置 `--api-key` 参数。
2. 设置环境变量 `VLLM_API_KEY` 。

修改 RayClusterFleet 的定义，在 headGroupSpec 中按上面任意一种方式设置 API 密钥之后，就需要在请求中带上以下 Header 进行访问：

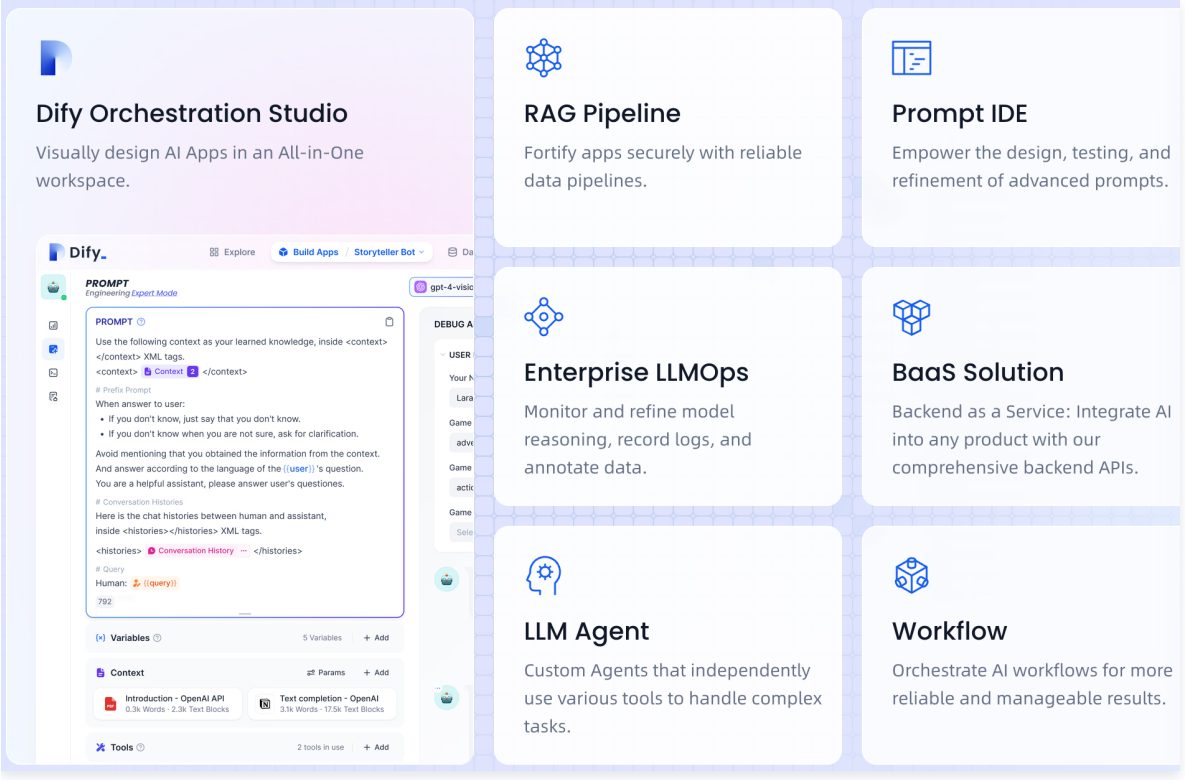
```
Authorization: Bearer <VLLM_API_KEY>
```

基于 TKE 部署 Dify 最佳实践

最近更新时间：2025-05-26 17:00:01

Dify 简介

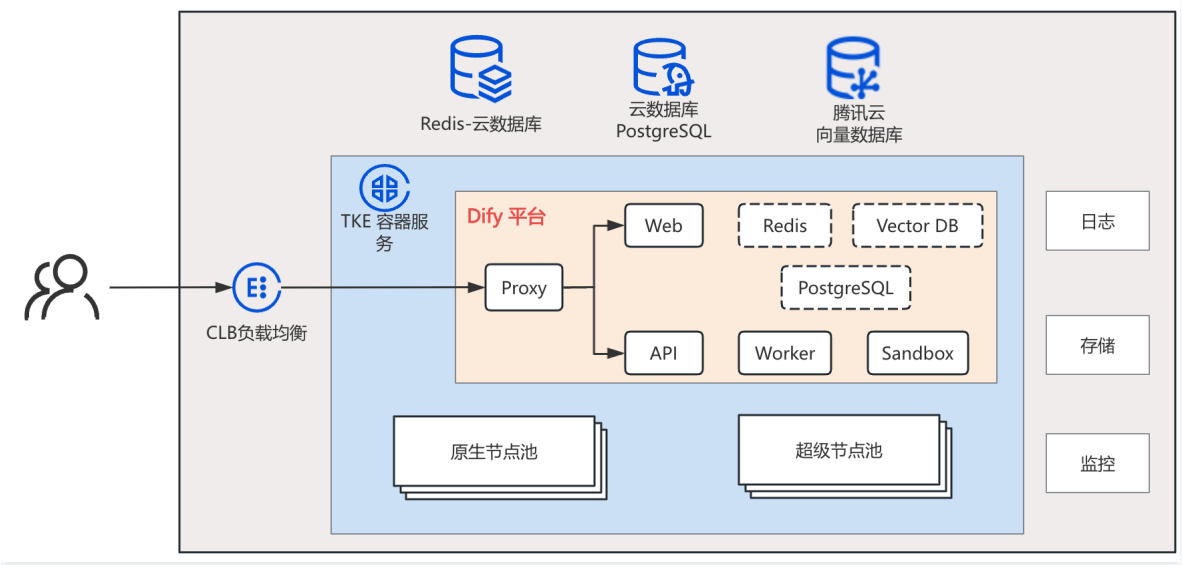
Dify 是一款开源的大语言模型（LLM）应用开发平台。平台内置了构建 LLM 应用所需的关键技术栈，包括对数百个模型的支持、直观的 Prompt 编排界面、高质量的 RAG 引擎、稳健的 Agent 框架、灵活的流程编排。同时，Dify 提供易用的界面和 API，使得开发者可以聚焦于创造应用的核心价值。



本文将详细介绍如何在腾讯云容器服务（TKE）上部署 Dify 平台，帮助企业和用户快速搭建自己的大模型应用开发平台。

整体架构

与单机部署相比，腾讯云 TKE 的部署提供了高可用、灵活弹性等特点，可以满足企业的生产部署需求，整体架构如下图。
部署方案全面涵盖了 Dify 平台的核心组件以及基础组件。其中，核心组件包括 api、web、worker 和 sandbox 等，是 Dify 平台正常运行的关键部分；基础组件包含向量数据库、db、redis、proxy、ssrf_proxy（专门用于防范 SSRF 攻击的安全代理）等服务，为 Dify 平台的稳定运行提供基础支撑。
在数据库选型上，可根据实际业务需求，选用腾讯云上对应的云服务替换默认的社区版数据库，如云数据库 Redis 、云数据库 PostgreSQL 和腾讯云向量数据库等，保障业务数据的安全稳定。



一键部署

腾讯云 TKE 应用市场已上架 **tke-dify** 应用，用户可通过应用市场实现一键部署，极大简化了部署流程。

实践步骤

存储准备

Dify 的关键组件 API 和 Work 需要共享存储，推荐在 TKE 集群中使用腾讯云 CFS 文件存储：

1. 在组件管理页面启用 CFS 文件存储。操作详情请参见 安装并设置 CFS 扩展组件 。

2. 组件创建完成后，在 StorageClass 中创建命名为 `cfs` 的 Storageclass 对象，操作详情请参见 创建 CFS 类型 StorageClass 。创建后的效果如下：

Dify 部署

TKE 的应用市场已经支持 tke-dify 应用的快速部署。

1. 登录 容器服务控制台 ，选择左侧导航栏中的**应用市场**。
2. 在应用管理中单击**新建**。
3. 在**新建应用**页面，填写应用信息。
   - 应用名：可自定义命名，本文示例为 `dify` 。
   - 应用场景：选择 AI，并选择 tke-dify 应用。

4. 单击**完成**，进行应用创建。

高可用保障

为确保 Dify 服务的高可用性，建议从多副本部署和健康检查两方面进行配置。

1. **多副本和反亲和性**

   以 api 组件为例（其他组件：worker、web、sandbox 均支持）：

```
api:
  replicas: 2
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: component
            operator: In
            values:
            - api
        topologyKey: topology.kubernetes.io/zone
```

2. **健康检测机制**

```
api:
  ## @param api.livenessProbe.enabled Enable livenessProbe on API nodes
  livenessProbe:
    enabled: true
    initialDelaySeconds: 30
    periodSeconds: 30
    timeoutSeconds: 5
    failureThreshold: 5
    successThreshold: 1
  ## @param api.readinessProbe.enabled Enable readinessProbe on API nodes
  readinessProbe:
    enabled: true
```

```
    initialDelaySeconds: 10
    periodSeconds: 10
    timeoutSeconds: 5
    failureThreshold: 5
    successThreshold: 1
```

弹性伸缩

Dify 作为 AI 服务平台，可能会面临突发流量增长。建议基于 HPA 配置弹性伸缩能力。
以 api 组件为例（其他组件：worker、web、sandbox 均支持），配置基于 CPU 使用率进行自动弹性：

```
api:
  autoscaling:
    enabled: true
    minReplicas: 1
    maxReplicas: 100
    targetCPUUtilizationPercentage: 80
```

请求入口

在 TKE 集群中，可以通过 Ingress 和 CLB 负载均衡将 Dify 服务暴露出来，并提供安全的访问方式。

```
service:
  type: NodePort
  port: 80

ingress:
  enabled: true
  className: ""
  annotations: {}
    # kubernetes.io/ingress.class: qcloud
  hosts:
    - host: dify-example.local
      paths:
        - path: /
          pathType: ImplementationSpecific
    # - host: dify-example2.local
    #   paths:
    #     - path: /
    #       pathType: Prefix
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - dify-example.local
```

更多 Ingress 能力请参见 CLB 类型 Ingress 。
部署完成后，在浏览器输入配置的域名（如 dify-example.local）即可访问 Dify 应用。首次访问需先配置管理员账号。登入界面如下图所示：

云产品集成

可以通过如下配置方式切换到相应云服务。

1. 腾讯云 Redis

```
redis:
  enabled: false
externalRedis:
  enabled: true
  host: "redis.example"
  port: 6379
  username: ""
  password: "difyai123456"
  useSSL: false
```

2. 腾讯云 PostgreSQL

```
postgresql:
  enabled: false
externalPostgres:
  enabled: true
  username: "postgres"
  password: "difyai123456"
  address: localhost
  port: 5432
  database:
    api: "dify"
    pluginDaemon: "dify_plugin"
  maxOpenConns: 20
  maxIdleConns: 5
```

3. 腾讯云向量数据库

```
weaviate:
  enabled: false
externalTencentVectorDB:
  enabled: true
  url: "your-tencent-vector-db-url"
  apiKey: "your-tencent-vector-db-api-key"
  timeout: 30
```

```
  username: "root"
  database: "dify"
  shard: 1
  replicas: 2
```

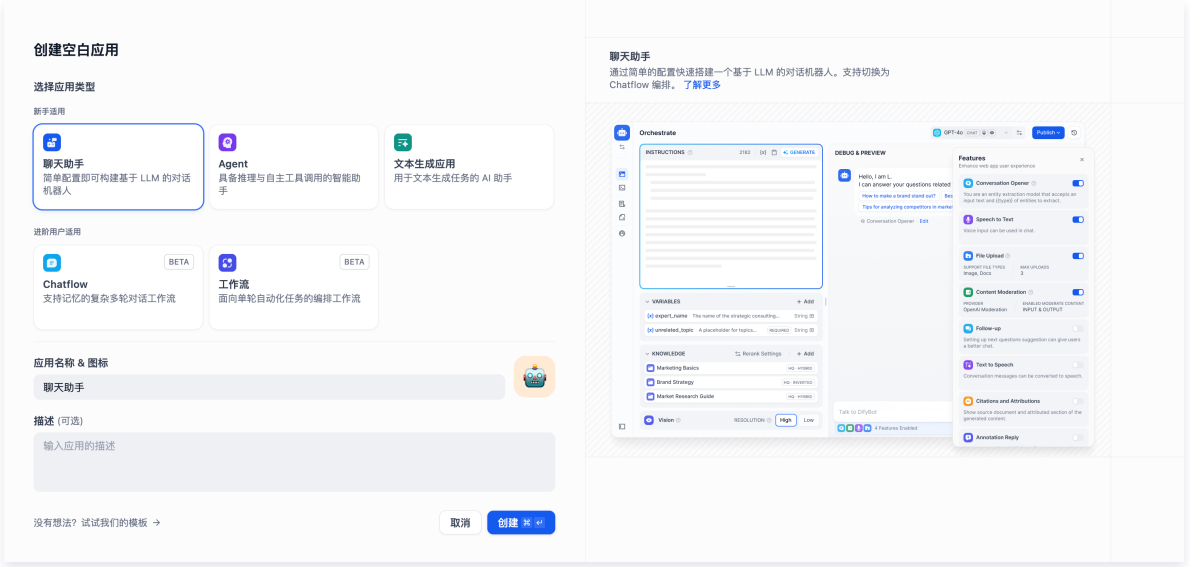
聊天助手

接下来，演示在 Dify 平台中创建聊天助手，并验证 AI 模型接入的使用效果。

1. 在 Dify 平台首页，进入用户设置，添加所需的 AI 模型，本文以**腾讯混元**为例。

2. 在腾讯混元插件的 API-KEY 配置页面，单击**从腾讯混元获取 API Key**，获取 SecretID 和 SecretKey，并填写到对应输入框。

3. 确认信息无误后单击**保存**，完成配置。
4. 完成上述配置后，返回 Dify 平台首页，通过首页空白应用的窗口，依照系统提示逐步创建聊天助手。



5. 进入聊天助手页面，可通过与聊天助手互动交流，输入各类问题并查看回复，以此验证聊天助手效果，确保 AI 模型接入正常且运行稳定。

总结

腾讯云 TKE 为 Dify 平台提供了生产级的容器化部署方案，从资源调度到维护管理全面赋能，助力企业快速落地 AI 应用，充分释放云原生技术的价值。

游戏
使用 CLB 为 Pod 分配公网地址映射

最近更新时间：2025-06-05 18:30:12

概述

在房间类游戏中，每个房间通常需要独立的公网地址。TKE 集群默认只支持 EIP 方案，但 EIP 资源有限，存在申请的数量限制和每日申请的次数限制（详情请参见 EIP 配额限制），在规模较大或频繁扩缩容的情况下，可能会触达限制导致 EIP 分配失败。而如果保留 EIP，在 EIP 没被绑定前，又会产生额外的闲置费。
除了 EIP 方案，您还可以使用 `tke-extend-network-controller` 插件的方案，本文将介绍如何使用 `tke-extend-network-controller` 插件来实现为每个 Pod 的指定端口都分配一个独立的公网地址映射（公网 `IP:Port` 到内网 Pod `IP:Port` 的映射）。

> **说明：**
> `tke-extend-network-controller` 的代码是开源的，源码托管在 GitHub：tke-extend-network-controller 。

前提条件

安装 `tke-extend-network-controller` 前请确保满足以下前提条件：

1. 确保腾讯云账号是带宽上移账号，参考 账户类型说明 进行判断或升级账号类型（如果账号创建的时间很早，有可能是传统账号）。
2. 创建了 TKE 集群，且集群版本大于等于 1.26。
3. 集群中安装了 cert-manager (webhook 依赖证书)，可通过 TKE 应用市场 安装。
4. 需要一个腾讯云子账号的访问密钥(SecretID、SecretKey)，参考子账号访问密钥管理，要求账号至少具有以下权限：

```
{
    "version": "2.0",
    "statement": [
        {
            "effect": "allow",
            "action": [
                "clb:CreateLoadBalancer",
                "clb:DeleteLoadBalancer",
                "clb:DescribeLoadBalancers",
                "clb:CreateListener",
                "clb:DeleteListener",
                "clb:DeleteLoadBalancerListeners",
                "clb:DescribeListeners",
                "clb:RegisterTargets",
                "clb:BatchRegisterTargets",
                "clb:DeregisterTargets",
                "clb:BatchDeregisterTargets",
                "clb:DescribeTargets",
                "clb:DescribeQuota",
                "clb:DescribeTaskStatus",
                "vpc:DescribeAddresses"
            ],
            "resource": [
                "*"
            ]
        }
    ]
}
```

操作步骤

安装 tke-extend-network-controller

在 TKE 应用市场 的网络分类中找到 `tke-extend-network-controller`，编辑 `values.yaml`，根据需求进行配置，以下几个参数是必填的：

```
vpcID: "" # TKE 集群所在 VPC ID (vpc-xxx)
region: "" # TKE 集群所在地域，如 ap-guangzhou
clusterID: "" # TKE 集群 ID (cls-xxx)
secretID: "" # 腾讯云子账号的 SecretID
secretKey: "" # 腾讯云子账号的 SecretKey
```

配置完成后单击**完成**即可安装到集群。
另外您也可以通过 helm 安装：

```
helm repo add tke-extend-network-controller https://tkestack.github.io/tke-extend-network-controller
helm upgrade --install -f values.yaml \
  --namespace tke-extend-network-controller --create-namespace \
  tke-extend-network-controller tke-extend-network-controller/tke-extend-network-controller
```

确保 Pod 调度到原生节点或超级节点

要使用 CLB 为 Pod 分配公网地址映射，需保证承载游戏房间的 Pod 调度到原生节点或超级节点上。如果 Pod 在普通节点（CVM），将不会为该 Pod 分配 CLB 公网地址映射。

使用 CLB 端口池为 Pod 映射公网地址

参考 使用 CLB 端口池为 Pod 映射公网地址 。