容器服务

实践教程

产品文档

腾讯云

文档目录

实践教程
集群
组建集群选型推荐

最近更新时间：2023-05-24 15:57:55

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： Docker 及 Containerd（beta）
- 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 支持以下两种网络模式架构，如需了解更多信息，请参见 如何选择容器服务网络模式。

- **GlobalRouter 模式架构**：
  - 基于 CNI 和网桥实现的容器网络能力，容器路由直接通过私有网络 VPC 底层实现。
  - 容器与节点在同一网络平面，但网段不与私有网络网段重叠，容器网段地址充裕。
- **VPC-CNI 模式架构**：

- 基于 CNI 和 VPC 弹性网卡实现的容器网络能力，容器路由通过弹性网卡，性能相比 Global Router 约提高10%。
- 容器与节点在同一网络平面，网段在 VPC 网段内。
- 支持 Pod 固定 IP。

使用方式

TKE 支持以下三种网络模式使用方式：

- 创建集群时指定 GlobalRouter 模式。
- 创建集群时指定 VPC-CNI 模式，则后续所有 Pod 都必须使用 VPC-CNI 模式创建。
- 创建集群时指定 GlobalRouter 模式，在需要使用 VPC-CNI 模式时为集群启用 VPC-CNI 的支持，即两种模式混用。

选型建议

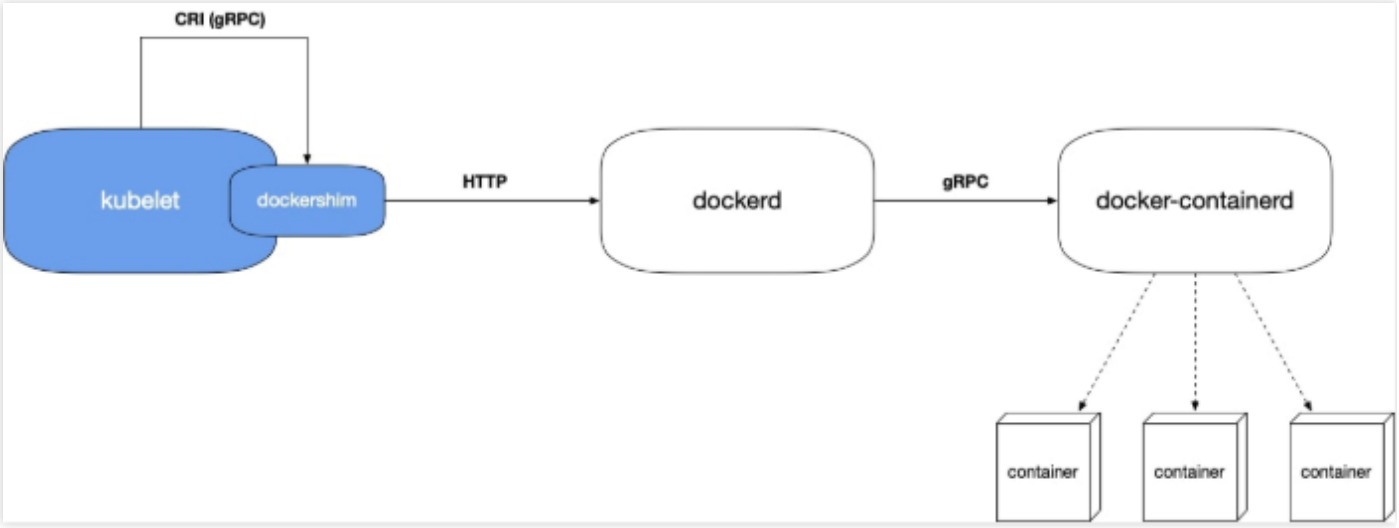
- 通常情况下应该选择 GlobalRouter，容器网段地址充裕、扩展性强且能适应规模较大的业务。
- 若后期部分业务需使用 VPC-CNI 模式，可在 GlobalRouter 集群中再开启 VPC-CNI 支持，即 GlobalRouter 与 VPC-CNI 混用，仅部分业务使用 VPC-CNI 模式。
- 若完全了解并接受 VPC-CNI 的使用限制，且集群内所有 Pod 都需用 VPC-CNI 模式，则可在创建集群时选择 VPC-CNI 模式。

运行时组件：Docker 及 Containerd（beta）

运行时架构

TKE 支持以下两种运行时架构，如需了解更多信息，请参见 如何选择 Containerd 和 Docker。

- **Docker 作为运行时架构**：



  - 调用链如下：

i. Kubelet 内置的 dockershim 模块帮助 docker 适配了 CRI 接口。

ii. Kubelet 通过 socket 文件自行调用 dockershim.

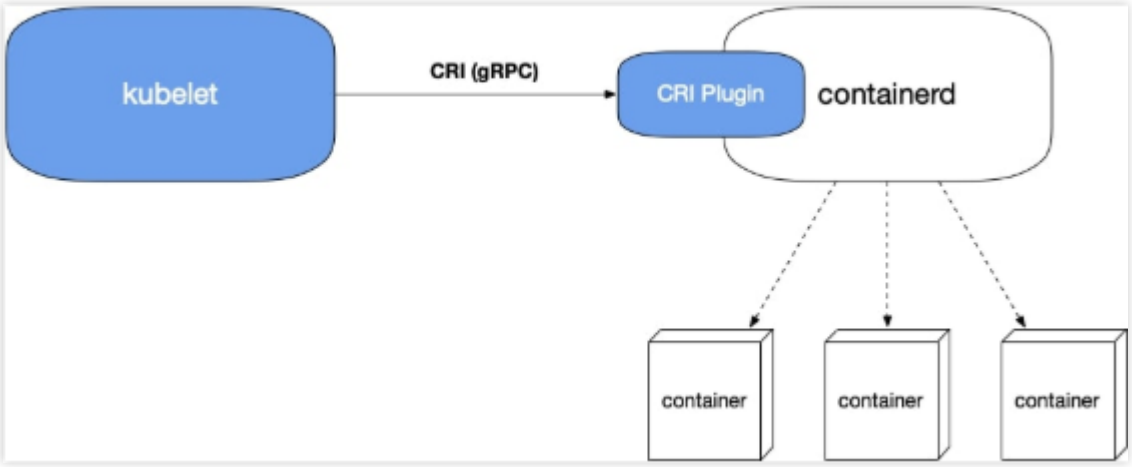
iii. Dockershim 调用 dockerd 接口（Docker HTTP API）。

iv. 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。

- **Containerd（beta） 作为运行时架构**：



  - 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。

运行时对比

- Containerd 方案由于绕过了 dockerd，具备调用链更短、组件更少、占用节点资源更少、绕过 dockerd 本身的一些 bug 等优点，但 containerd 自身也还存在一些 bug，目前 containerd 在 beta 阶段，已修复部分 bug。
- Docker 方案历史较悠久、相对更成熟、支持 Docker API 且功能丰富，符合大多数人的使用习惯。

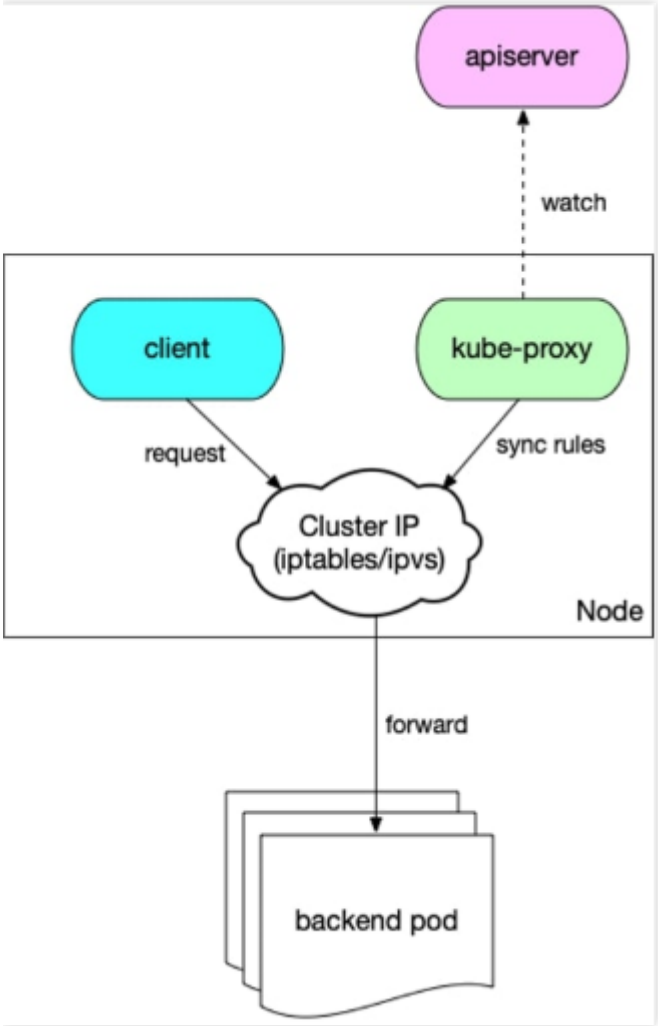
选型建议

- Docker 方案相比 containerd 更成熟，如果对稳定性要求较高，建议选择此方案。
- 以下场景仅支持使用 docker：
- Docker in docker（通常在 CI 场景）。
- 节点上使用 docker 命令。
- 调用 docker API。

若非以上场景，建议选择 containerd。

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：

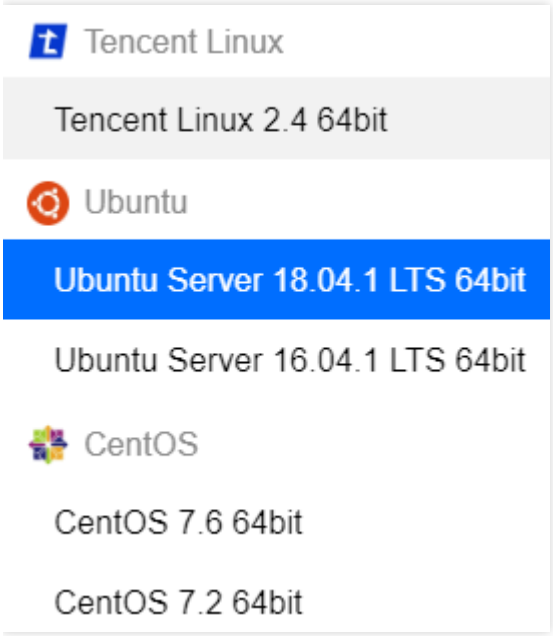
- **托管集群**：
- Master 组件用户不可见，由腾讯云托管。
- 会率先支持大部分新功能的托管。
- Master 的计算资源会根据集群规模自动扩容
- 用户不需要为 Master 付费。
- **独立集群**：
  - 用户可完全掌控 Master 组件。
  - 用户需要为 Master 付费购买机器。

    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 Tencent Linux、Ubuntu 和 CentOS 三类发行版操作系统，其中 Tencent Linux 版本的操作系统使用了腾讯云团队维护定制内核 TencentOS-kernel，其余的操作系统使用了 Linux 社区官方开源内核。如下图所示：



说明：
在 Tencent Linux 公共镜像上线之前，为了提升镜像稳定性，并提供更多特性，容器服务 TKE 团队制作并维护 TKE-Optimized 系列镜像。目前控制台已不支持新建集群选择 TKE-Optimized 镜像，更多相关详情请参见 TKE-Optimized 系列镜像说明。

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 可以避免其它业务 Pod 调度上来）。

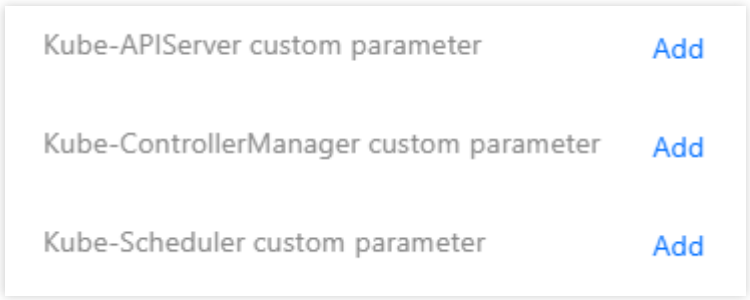
当业务量快速上升时，该 IO 密集型业务也需要更多的计算资源。在业务高峰时段，HPA 功能自动为该业务扩容了 Pod，而节点计算资源不够用，此时节点池的自动伸缩功能自动扩容了节点，守住了流量高峰。

使用启动脚本

组件自定义参数

> 说明：
> 如需使用该功能，请通过 提交工单 进行申请。

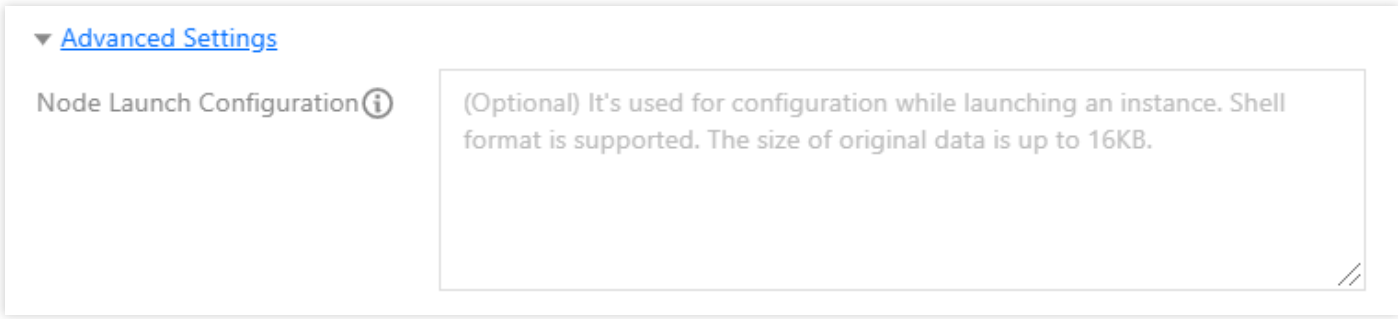
- 在创建集群时，可在配置“集群信息”的“高级设置”中，自定义 Master 组件部分启动参数。如下图所示：



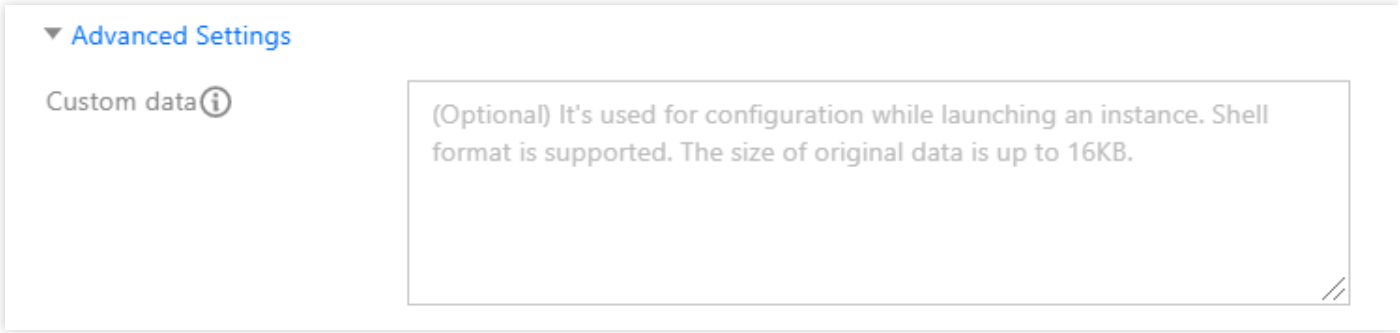
- 在“选择机型”时，可在 “Worker 配置”的“高级设置”中，自定义 kubelet 部分启动参数。如下图所示：

节点启动配置

- 在创建集群时，可在“云服务器配置”的“高级设置”中，自定义数据配置节点启动脚本（可用于修改组件启动参数、内核参数等）。如下图所示：



- 在添加节点时，可“云服务器配置”的“高级设置”中，自定义数据配置节点启动脚本（可用于修改组件启动参数、内核参数等）。如下图所示：



实现独立集群的 Master 容灾

最近更新时间：2021-04-07 17:24:13

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。

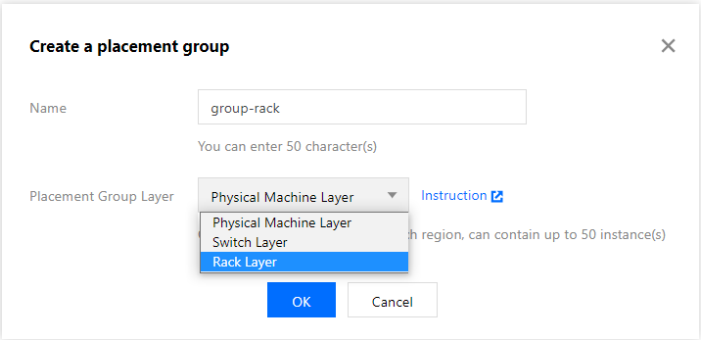
容灾实现思路

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：

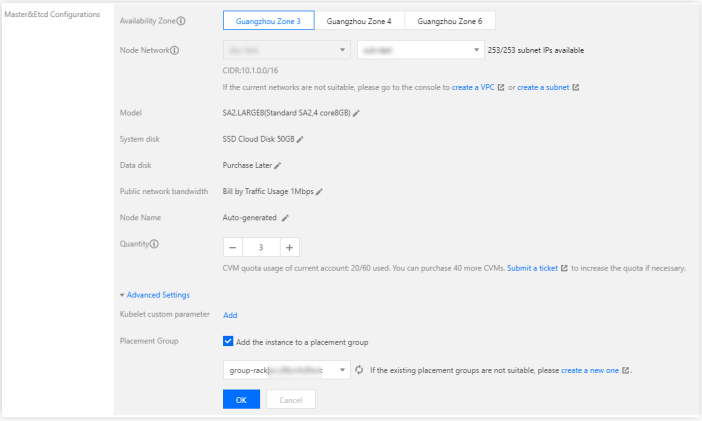
> **⚠ 注意：**
> 置放群组需与 TKE 独立集群在同一地域。



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。使用这个层级以将 Master 打散部署到连到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 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 配置”时，在多个可用区添加机型即可。如下图所示：

使用 Private DNS 实现内网访问集群时的自动域名解析

最近更新时间：2022-06-10 16:48:45

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：

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 提交工单。

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。

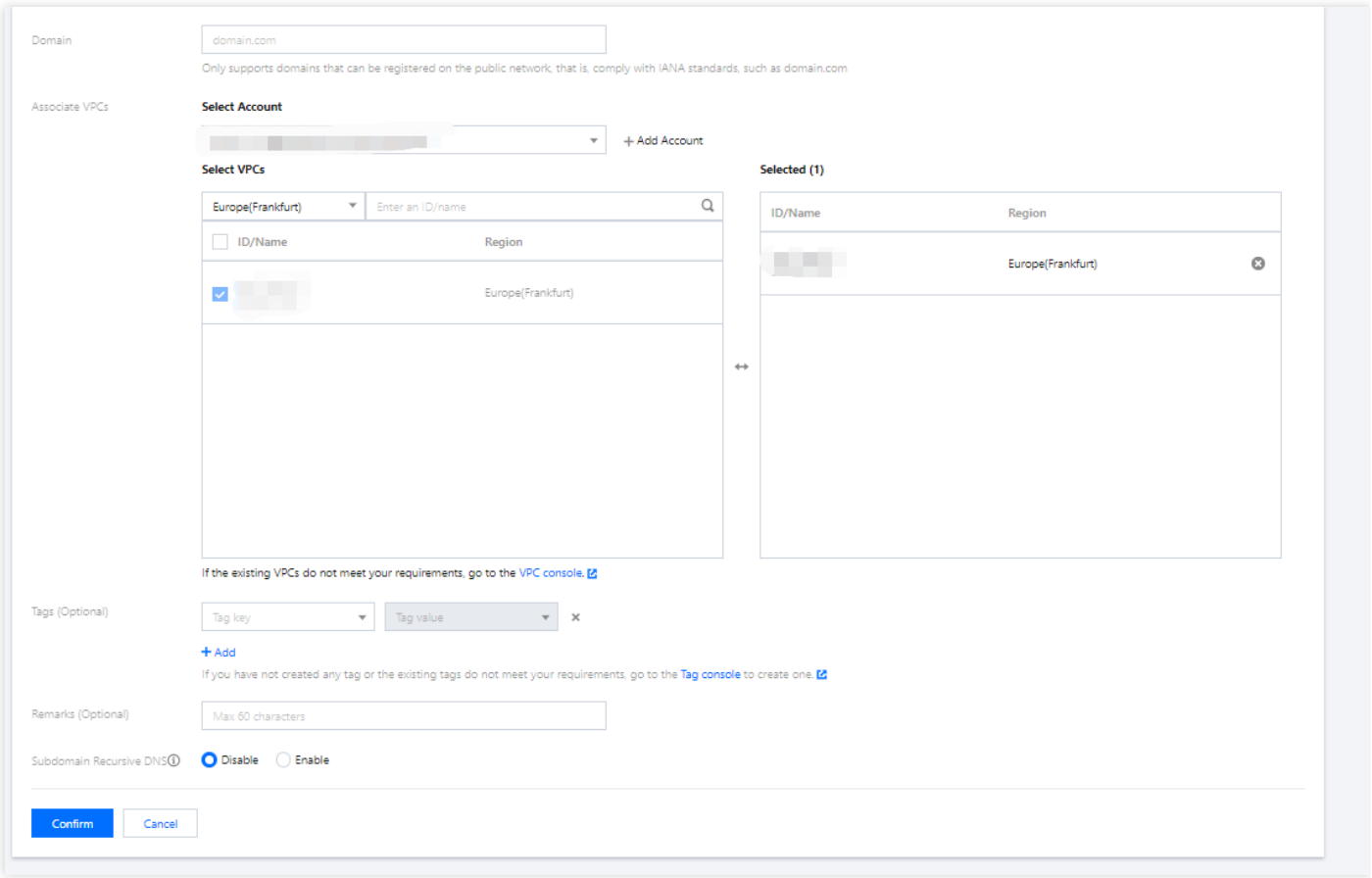
操作步骤

开通 Private DNS

请参见官方文档开通 Privae 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。

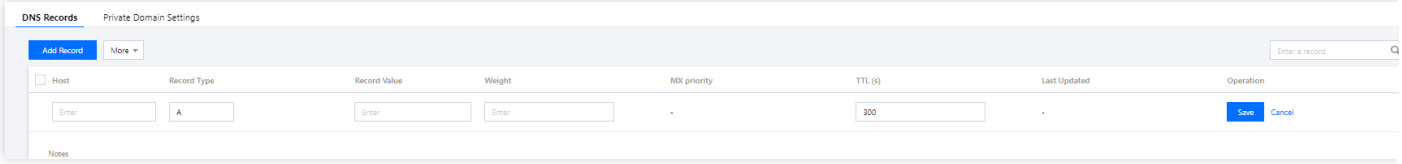


- **域名**：输入“tencent-cloud.com”（TKE 为集群访问分配的域名）。
- **关联 VPC**：选择需要访问集群的节点网络 VPC。

3. 单击**确定**即可创建私有域。

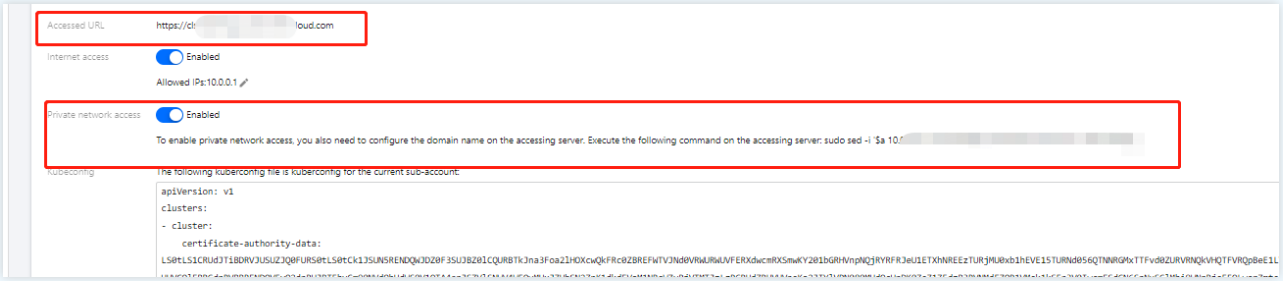
配置解析记录

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。

> 说明：
> **主机记录**和**记录值**可前往 **集群管理** >**集群** > **基本信息**获取。其中**主机记录**对应**访问地址**中的域名，**记录值**对应**内网访问**中的 IP 地址，如下图所示：
>
> 
>

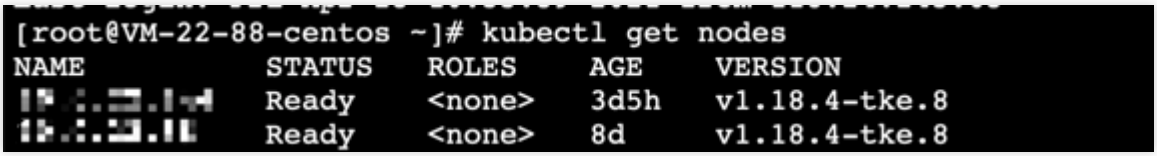
3. 单击右侧操作栏下的**保存**以保存配置。

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
使用对象存储 COS 作为 Velero 存储实现集群资源备份和还原

最近更新时间：2023-05-23 16:26:03

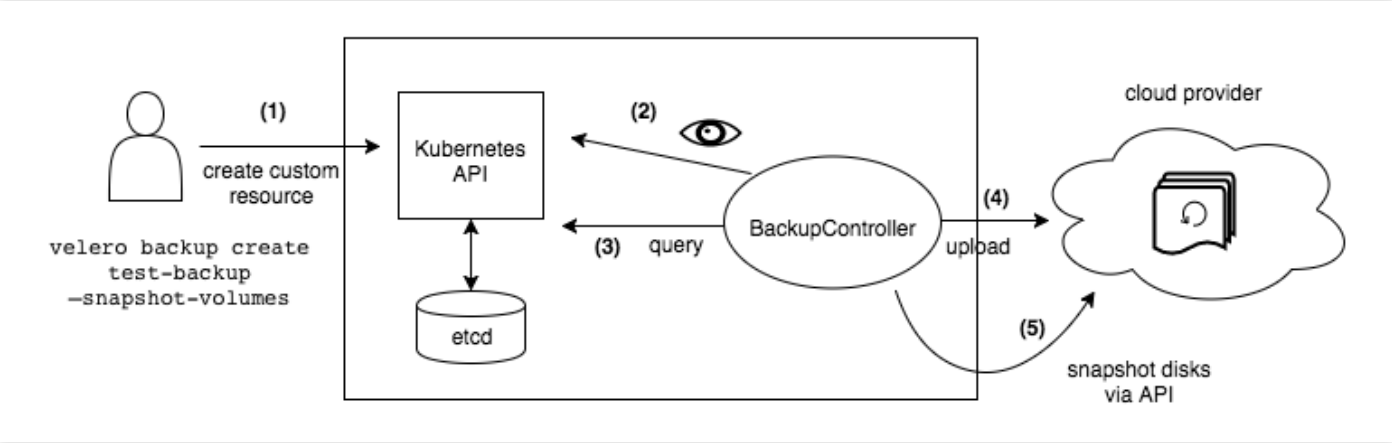
操作场景

开源工具 Velero（旧版本名称为 Heptio Ark）可以安全地备份和还原、执行灾难恢复以及迁移 Kubernetes 集群资源和持久卷。在容器服务 TKE 集群或自建 Kubenetes 集群中部署 Velero 可以实现以下功能：

- 备份集群资源并在丢失的情况下进行还原。
- 将集群资源迁移到其他集群。
- 将生产集群资源复制到开发和测试集群。

Velero 工作原理图如下图所示（来源于 Velero 官网），当用户执行备份命令时，备份过程说明如下：

1. 调用自定义资源 API 创建备份对象（1）。
2. BackupController 控制器检测到生成的备份对象时（2）执行备份操作（3）。
3. 将备份的集群资源和存储卷快照上传到 Velero 的后端存储（4）和（5）。



另外当执行还原操作时，Velero 会将指定备份对象的数据从后端存储同步到 Kubernetes 集群完成还原工作。
更多关于 Velero 介绍，请参见 Velero 官网文档。本文将介绍如何使用腾讯云 对象存储 COS 作为 Velero 后端存储实现集群备份和还原。

前提条件

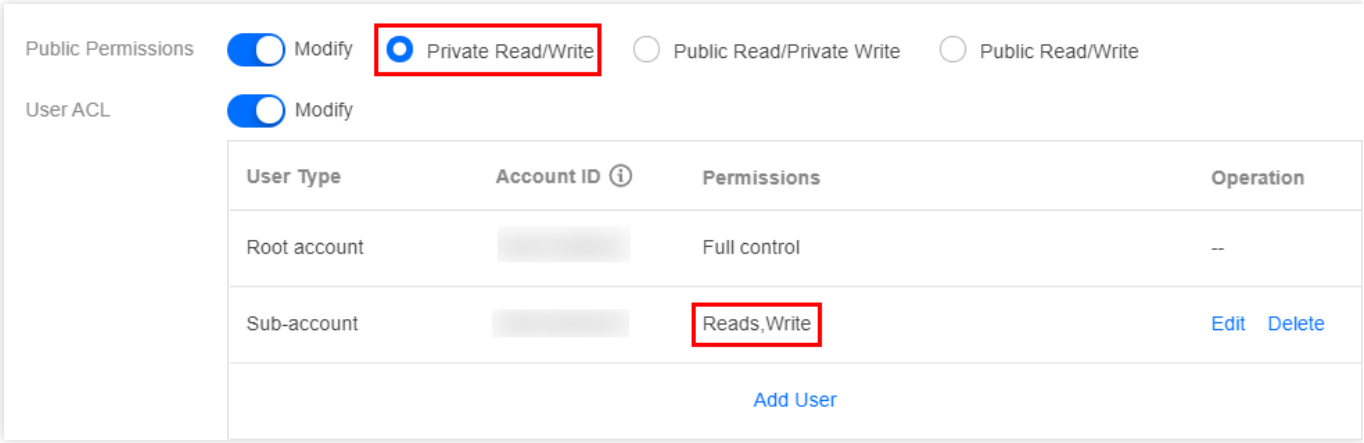
- 已 注册腾讯云账号。
- 已开通腾讯云 对象存储 COS 服务。
- 已创建 v1.10 或以上版本的 Kubernetes 集群，集群可正常使用 DNS 和 互联网服务，详情请参见 创建集群。

操作步骤

配置对象存储

创建存储桶

1. 在 对象存储控制台 为 Velero 创建一个对象存储桶用于存储备份，详情请参见 创建存储桶。
2. 为存储桶 设置访问权限。对象存储 COS 支持设置两种权限类型：
   - **公共权限**：为了安全起见，推荐存储桶权限类别为私有读写，关于公共权限的说明，请参见存储桶概述中的 权限类别。
   - **用户权限**：主账号默认拥有存储桶所有权限（即完全控制）。另外 COS 支持添加子账号有数据读取、数据写入、权限读取、权限写入，甚至**完全控制**的最高权限。
     由于需要对存储桶进行读写操作，为示例子账号授予**数据读取、数据写入**权限，如下图所示：



获取存储桶访问凭证

Velero 使用与 AWS S3 兼容的 API 访问 COS，需要使用一对访问密钥 ID 和密钥创建的签名进行身份验证，在 S3 API 参数中：

- `access_key_id`：访问密钥 ID
- `secret_access_key`：密钥

1. 在腾讯云 访问管理控制台 新建和获取 COS 授权子账号的腾讯云密钥 `SecretId` 与 `SecretKey`。其中：

- `SecretId` 值对应 `access_key_id` 字段
- `SecretKey` 值对应 `secret_access_key` 字段

2. 根据上述对应关系，在本地目录创建 Velero 所需的凭证配置文件 `credentials-velero`，内容如下：

```
[default]
aws_access_key_id=<SecretId>
aws_secret_access_key=<SecretKey>
```

安装 Velero

1. 下载 Velero 最新版本安装包到集群环境中，本文以 v1.5.2 版本为例。示例如下：

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.5.2/velero-v1.5.2-linux-amd64.tar.gz
```

2. 执行以下命令解压安装包，安装包提供 Velero 命令行执行文件和一些示例文件。示例如下：

```
tar -xvf velero-v1.5.2-linux-amd64.tar.gz
```

3. 执行以下命令，将 Velero 可执行文件从解压后的目录迁移到系统环境变量目录下直接使用，本文以迁移至 `/usr/bin` 目录为例。示例如下：

```
mv velero-v1.5.2-linux-amd64/velero /usr/bin/
```

4. 执行以下命令安装 Velero，创建 Velero 和 Restic 工作负载以及其他必要的资源对象（安装参数说明请见 下表）。示例如下：

```
velero install --provider aws --plugins velero/velero-plugin-for-aws:v1.1.0 --bucket <BucketName> \
```

`--secret-file ./credentials-velero`

`--use-restic`

`--default-volumes-to-restic`

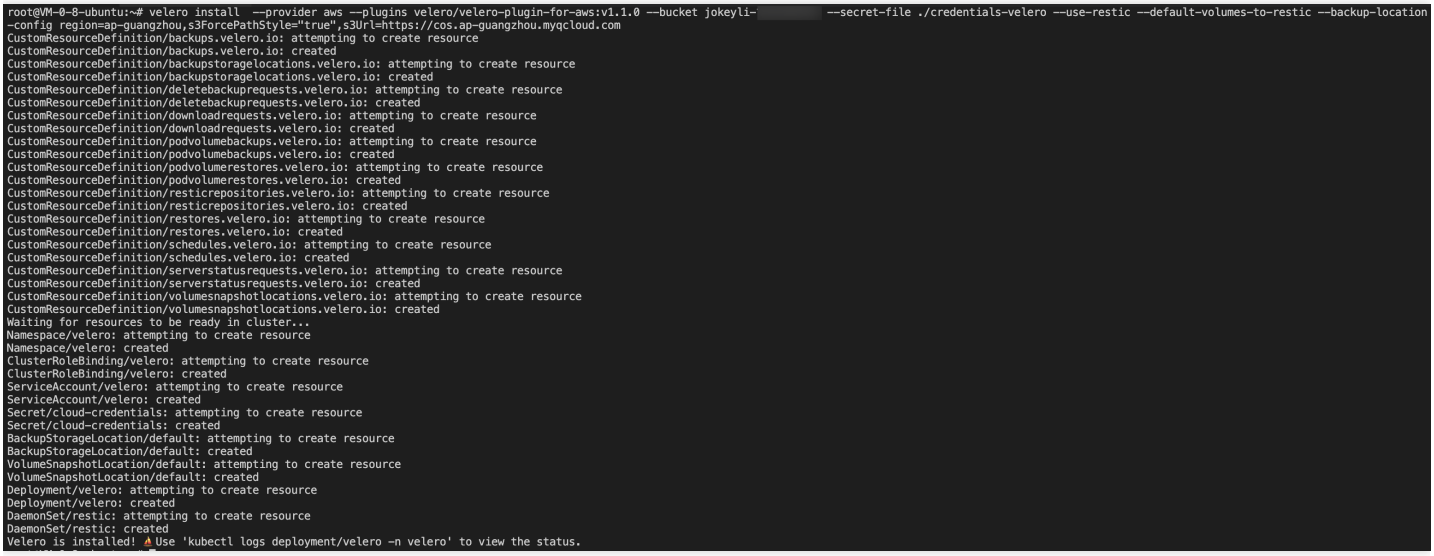
```
--backup-location-config
region=ap-guangzhou,s3ForcePathStyle="true",s3Url=https://cos.ap-
guangzhou.myqcloud.com
```

安装参数说明：

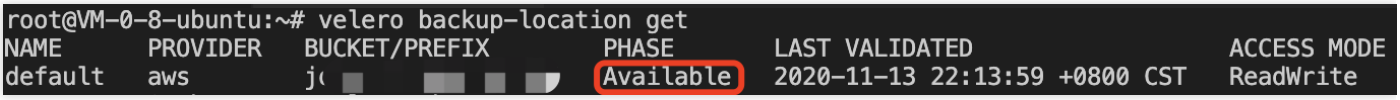
安装参数	参数说明
--provider	声明使用 `aws` 提供的插件类型。
--plugins	使用 AWS S3 兼容 API 插件 “velero-plugin-for-aws”。
--bucket	在对象存储 COS 创建的存储桶名。
--secret-file	访问对象存储 COS 的访问凭证文件，详情参见上述创建的 “credentials-velero” 凭证文件。
--use-restic	Velero 支持使用免费开源备份工具 Restic 备份和还原 Kubernetes 存储卷数据（不支持 `hostPath` 卷，详情请参见 Restic 限制），该集成是 Velero 备份功能的补充，建议开启。
--default-volumes-to-restic	启用使用 Restic 来备份所有 Pod 卷，前提是需要开启 `--use-restic` 参数。
--backup-location-config	备份存储桶访问相关配置，包括 region、s3ForcePathStyle、s3Url 等。
region	兼容 S3 API 的对象存储 COS 存储桶地域，例如创建地域为广州，region 参数值为 “ap-guangzhou”
s3ForcePathStyle	使用 S3 文件路径格式。
s3Url	对象存储 COS 兼容的 S3 API 访问地址。请注意该访问地址中的域名不是上述创建 COS 存储桶的公网访问域名，须使用格式为 https://cos.<region>.myqcloud.com 的 URL，例如地域为广州，则参数值为 `https://cos.ap-guangzhou.myqcloud.com`。

其他安装参数可以使用命令 `velero install --help` 查看。例如，不备份存储卷数据，可以设置 `--use-volume-snapshots=false` 来关闭存储卷快照备份。

执行安装命令之后查看安装过程，如下图所示：



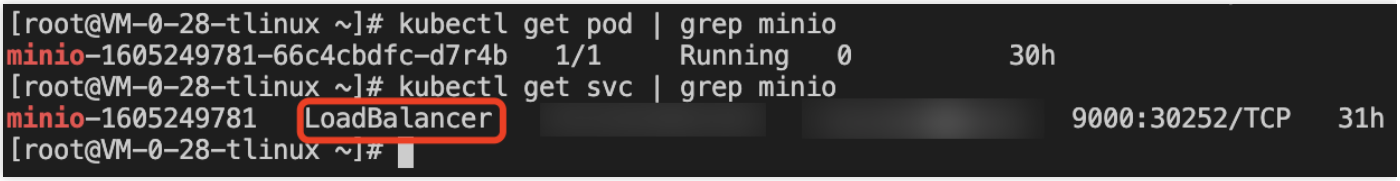
5. 安装完成后，等待 Velero 和 Restic 工作负载就绪。执行以下命令，查看配置的存储位置是否可用，若显示“Avaliable”，则说明集群可正常访问对象存储 COS，如下图所示：



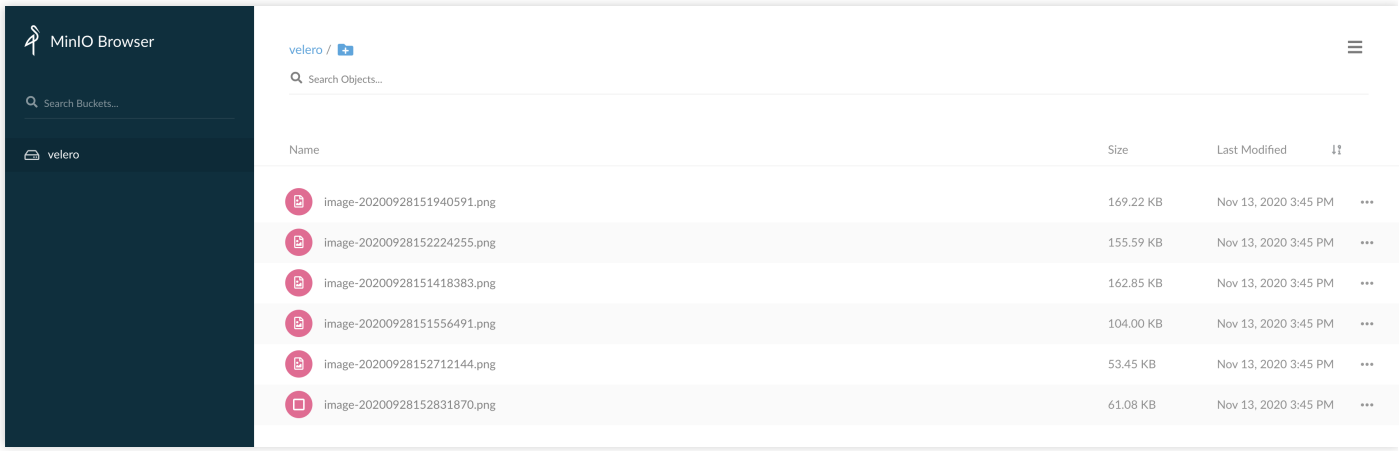
至此，Velero 安装完成。了解 Velero 更多安装介绍，请参见 Velero 官网文档。

Velero 备份还原测试

1. 在集群中使用 Helm 工具，创建一个具有持久卷的 MinIO 测试服务，MinIO 安装方式请参见 MinIO 安装。在此示例中，已经为 MinIO 服务绑定了负载均衡器，可以在浏览器中使用公网地址访问管理页面。



2. 登录 MinIO Web 管理页面，上传用于测试的图片， 如下图所示：



3. 使用 Velero 备份，可以直接备份集群中的所有对象，也可以按类型，名称空间和/或标签过滤对象。您可以执行以下命令仅备份 default 命名空间下所有资源。示例如下：

```
velero backup create default-backup --include-namespaces default
```

4. 执行以下命令查看备份任务是否完成，当备份任务状态是 “Completed” 且 “ERRORS” 为0时，说明备份任务完成且未发生任何错误。

```
velero backup get
```

备份过程如下图所示：

```
[root@VM-0-28-tlinux ~]# velero backup create default-backup --include-namespaces default
Backup request "default-backup" submitted successfully.
Run `velero backup describe default-backup` or `velero backup logs default-backup` for more details.
[root@VM-0-28-tlinux ~]# velero backup get
NAME             STATUS       ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
default-backup   InProgress   0        0          2020-11-14 22:34:18 +0800 CST   29d       default            <none>
[root@VM-0-28-tlinux ~]# velero backup get
NAME             STATUS       ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
default-backup   InProgress   0        0          2020-11-14 22:34:18 +0800 CST   29d       default            <none>
[root@VM-0-28-tlinux ~]# velero backup get
NAME             STATUS       ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
default-backup   Completed    0        0          2020-11-14 22:34:18 +0800 CST   29d       default            <none>
```

5. 执行以下命令，删除 MinIO 下所有资源，包括 PVC 持久卷。如下图所示：

```
[root@VM-0-28-tlinux ~]# helm list
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /root/.kube/config
WARNING: Kubernetes configuration file is world-readable. This is insecure. Location: /root/.kube/config
NAME                    NAMESPACE       REVISION        UPDATED                                 STATUS          CHART           APP VERSION
minio-1605249781        default         1               2020-11-13 14:43:02.437981822 +0800 CST deployed        minio-8.0.3     master
[root@VM-0-28-tlinux ~]#  helm uninstall minio-1605249781
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /root/.kube/config
WARNING: Kubernetes configuration file is world-readable. This is insecure. Location: /root/.kube/config
release "minio-1605249781" uninstalled
[root@VM-0-28-tlinux ~]# kubectl get pvc | grep minio
No resources found in default namespace.
[root@VM-0-28-tlinux ~]# kubectl get pod | grep minio
[root@VM-0-28-tlinux ~]#
```

6. 删除 MinIO 资源后，使用之前的备份测试是否可以成功还原被删除的 MinIO 资源。执行以下命令，将备份存储位置临时更新为只读模式（可以防止在还原过程中，Velero 在备份存储位置中创建或删除备份对象）。

```
kubectl patch backupstoragelocation default --namespace velero \
--type merge \
--patch '{"spec":{"accessMode":"ReadOnly"}}'
```

执行过程如下图所示：

```
[root@VM-0-28-tlinux ~]# kubectl patch backupstoragelocation default --namespace velero \
>     --type merge \
>     --patch '{"spec":{"accessMode":"ReadOnly"}}'
backupstoragelocation.velero.io/default patched
[root@VM-0-28-tlinux ~]# velero backup-location get
NAME      PROVIDER   BUCKET/PREFIX        PHASE       LAST VALIDATED                  ACCESS MODE
default   aws        jok[redacted]        Available   2020-11-14 22:55:41 +0800 CST   ReadOnly
```

7. 执行以下命令，使用上述 步骤3 Velero 创建的备份 “default-backup” 来创建还原任务。示例如下：

```
velero restore create --from-backup default-backup
```

通过命令 `velero restore get` 查看还原任务的状态，若还原状态是 “Completed” 且 “ERRORS” 为0时，则说明还原任务完成，如下图所示：

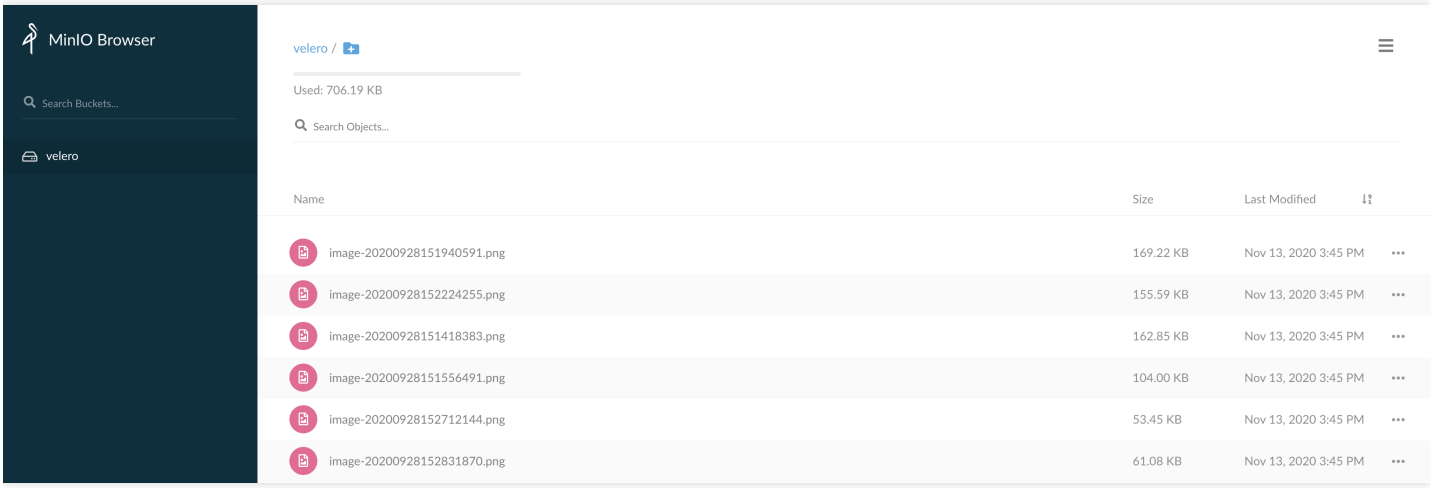
```
[root@VM-0-28-tlinux ~]# velero restore get
NAME                            BACKUP          STATUS      STARTED                         COMPLETED                       ERRORS  WARNINGS  CREATED                         SELECTOR
default-backup-20201114225846   default-backup  Completed   2020-11-14 22:58:46 +0800 CST   2020-11-14 23:00:01 +0800 CST   0       4         2020-11-14 22:58:46 +0800 CST   <none>
[root@VM-0-28-tlinux ~]#
```

8. 还原完成后，执行以下命令，可以查看到之前被删除的 MinIO 相关资源已经还原成功。如下图所示：

```
[root@VM-0-28-tlinux ~]# kubectl get pod | grep minio
minio-1605249781-66c4cbdfc-d7r4b   1/1     Running   0          6m2s
[root@VM-0-28-tlinux ~]# kubectl get pvc | grep minio
minio-1605249781   Bound    pvc-c3988f43-1a53-4071-b369-8d276bd7b636   500Gi      RWO            cbs            6m7s
[root@VM-0-28-tlinux ~]# kubectl get svc | grep minio
minio-1605249781   LoadBalancer   [redacted]   [redacted]   9000:31112/TCP   7m44s
[root@VM-0-28-tlinux ~]#
```

9. 在浏览器上登录 MinIO 的管理页面，可以查看到之前上传的图片，说明持久卷的数据还原成功。如下图所示：

> 注意：
> 本文使用 Restic 来备份和还原持久卷，但 Restic 不支持 `hostPath` 类型卷，详情请参见 Restic 限制。



10. 另外在还原完成后，可以执行以下命令，将备份存储位置恢复为读写模式，以便在下次可以正常备份。示例如下：

```
kubectl patch backupstoragelocation default --namespace velero \
--type merge \
--patch '{"spec":{"accessMode":"ReadWrite"}}'
```

Velero 卸载

执行以下命令，可以在集群中卸载 Velero。示例如下：

```
kubectl delete namespace/velero clusterrolebinding/velero
kubectl delete crds -l component=velero
```

总结

本文主要介绍 Kubernetes 集群资源备份工具 Velero，展示了如何配置腾讯云 COS 对象存储来作为 Velero 的后端存储，并成功实践 MinIO 服务资源和数据的备份和还原操作。

参考文档

- Velero 官网
- Restic 工具介绍
- Minio 安装
- restic 限制

在 TKE 中使用 Velero 迁移复制集群资源

最近更新时间：2023-05-23 15:42:12

操作场景

开源工具 Velero（旧版本名称为 Heptio Ark）可以安全地备份和还原、执行灾难恢复以及迁移 Kubernetes 集群资源和持久卷。在容器服务 TKE 集群或自建 Kubenetes 集群中部署 Velero 可以实现以下功能：

- 备份集群资源并在丢失的情况下进行还原。
- 将集群资源迁移到其他集群。
- 将生产集群资源复制到开发和测试集群。

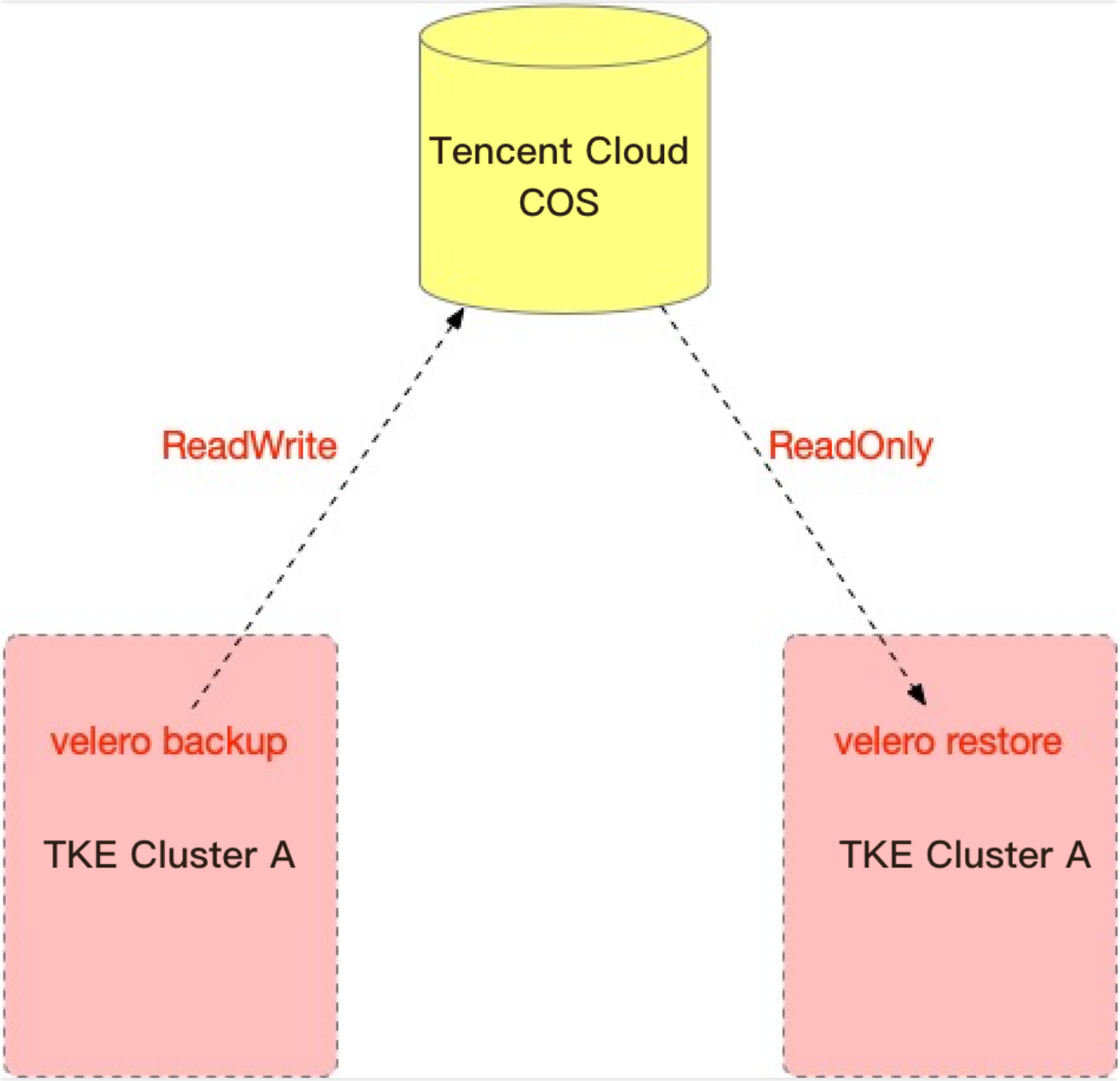
更多关于 Velero 介绍，请参见 Velero 官网文档。本文将介绍如何使用 Velero 实现 TKE 集群间的无缝迁移复制集群资源。

迁移原理

在需要被迁移的集群和目标集群上都安装 Velero 实例，并且两个集群的 Velero 实例指向相同的腾讯云 对象存储 COS 位置，流程如下：

1. 使用 Velero 在需要被迁移的集群执行备份操作，生成备份数据存储到对象存储 COS。
2. 在目标集群上使用 Velero 执行数据的还原操作实现迁移。

迁移原理如下图示：



前提条件

- 已 注册腾讯云账号。
- 已开通腾讯云 对象存储 COS 服务。
- 已有需要被迁移的 TKE 集群（以下称作集群 A），已创建迁移目标的 TKE 集群（以下称作集群 B），创建 TKE 集群请参见 创建集群。

- 集群 A 和 集群 B 都需要安装 Velero 实例（1.5版本以上），并且共用同一个对象存储 COS 存储桶作为 Velero 后端存储，安装步骤请参见 配置存储和安装 Velero。

注意事项

- 从 Velero 1.5版本开始，Velero 可以使用 Restic 备份所有 Pod 卷，无需单独注释每个 Pod。默认情况下，此功能允许用户使用 Restic 备份所有 Pod 卷，但以下卷情况除外：
  - 挂载默认 `Service Account Secret` 的卷
  - 挂载 `hostPath` 的类型卷
  - 挂载 Kubernetes `secrets` 和 `configmaps` 的卷

    本示例需要 Velero 1.5以上版本且启用 Restic 来备份持久卷数据，请确保在安装 Velero 阶段开启 `--use-restic` 和 `--default-volumes-to-restic` 参数，安装步骤请参见 配置存储和安装 Velero。
- 在执行迁移过程中，禁止对两边集群资源进行任何 CRUD 操作，以免在迁移过程中造成数据差异，导致最终迁移后的数据不一致。
- 尽量保证集群 A 和集群 B 的CPU、内存等规格配置相同或不要相差太大，以免出现迁移后的 Pods 因资源原因无法调度导致 Pending 的情况。

操作步骤

在集群 A 创建备份

备份前查看集群 A 资源

在备份集群 A 之前，您可以查看集群 A 资源和服务情况，以便在还原集群之后用于 迁移结果核验。

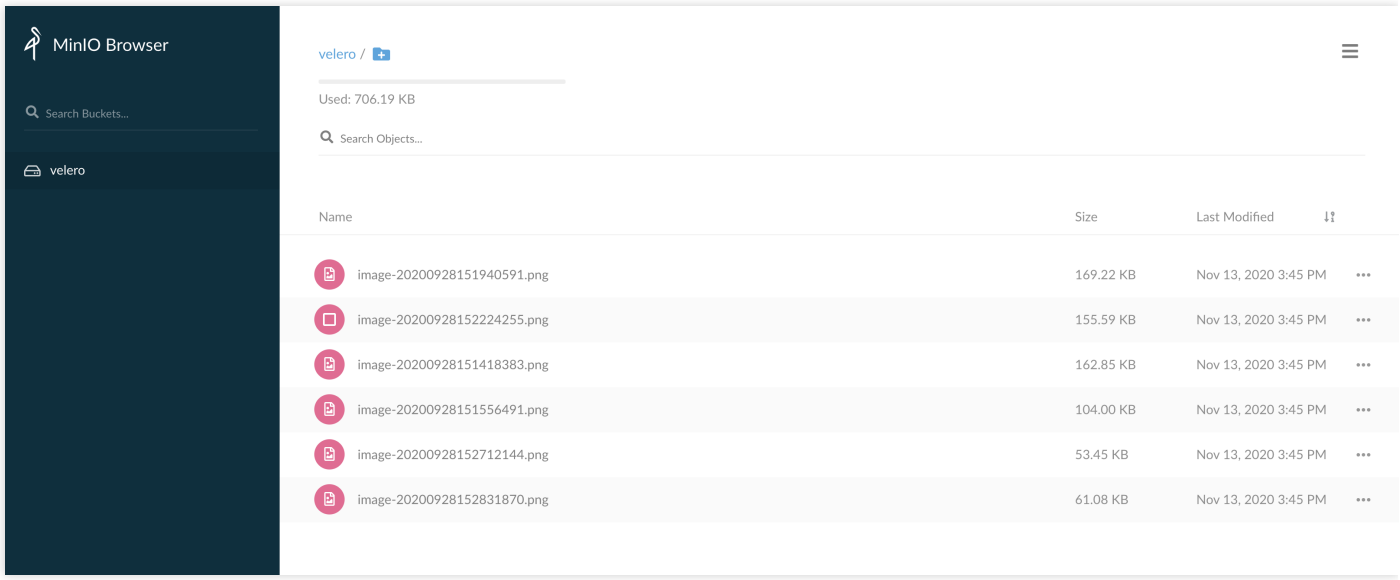
1. 本文将以 default 、default2 命名空间的资源情况作比较验证。执行以下命令，可以查看集群 A 中两个命名空间下的 Pods 和 PVC 资源情况。如下图所示：

```
[root@VM-0-28-tlinux ~]# kubectl get pod
NAME                              READY   STATUS    RESTARTS   AGE
migrate-test-6bf6d577f4-7g7b9     1/1     Running   0          5d1h
minio-1605249781-66c4cbdfc-d7r4b  1/1     Running   0          21h
[root@VM-0-28-tlinux ~]# kubectl get pod -n default2
NAME                          READY   STATUS    RESTARTS   AGE
default2-58856dc878-kprhk   1/1     Running   0          21h
default2-58856dc878-zgnrw   1/1     Running   0          21h
[root@VM-0-28-tlinux ~]# kubectl get pvc
NAME               STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
minio-1605249781   Bound    pvc-d9b13f5f-6478-4c13-ae4e-942c1a7bf175   500Gi      RWO            cbs            21h
[root@VM-0-28-tlinux ~]#
```

> 说明：

> 您可以在备份期间指定执行一些自定义 Hook 操作。例如，需要在备份之前将运行应用程序的内存中的数据持久化到磁盘。 了解 Hook 更多信息，请参见 备份 Hook。

2. 其中，集群 A 中的 MinIO 对象存储服务使用了持久卷，并且已上传一些图片数据，如下图所示：

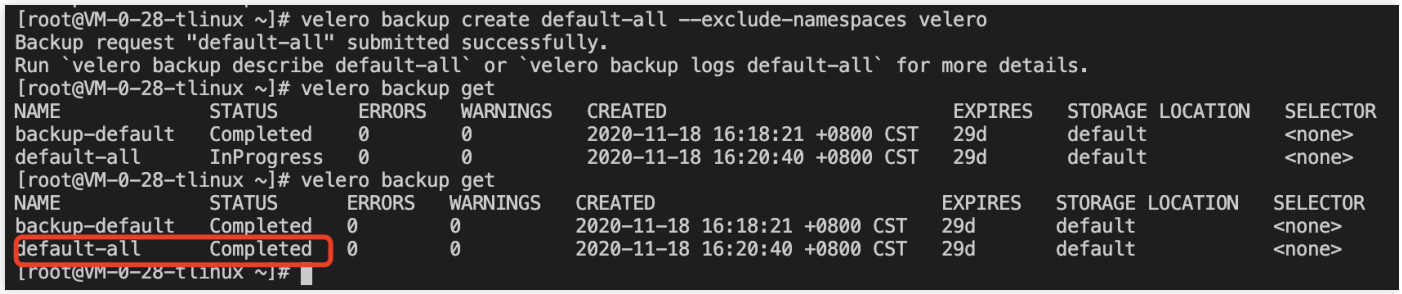


备份集群

1. 执行以下命令，备份集群中不包含 Velero 命名空间（Velero 安装的默认命名空间）资源的其他所有资源，如果需要自定义备份的集群资源范围，可使用命令 `velero create backup -h` 查看支持的资源筛选参数。

```
velero backup create <BACKUP-NAME> --exclude-namespaces <NAMESPACE>
```

本示例以创建一个 “default-all” 集群备份为例，备份过程如下图所示。若备份任务状态显示 “Completed” 时，说明备份成功。



> 说明：
> 您还可以为 Velero 设置定期自动备份，设置方法可以使用命令 `velero schedule -h` 查看。

2. 执行以下命令，检查是否有备份操作发生错误，若命令无任何输出结果，则说明备份过程未发生任何错误。示例如下：

```
velero backup logs <BACKUP-NAME> | grep error
```

> 注意：
> 请确保备份过程未发生任何错误，若 Velero 在执行备份过程中发生错误，请排查解决后重新执行备份。

3. 备份完成后执行以下命令，将备份存储位置临时更新为只读模式（非必须，可以防止在还原过程时， Velero 在备份存储位置中创建或删除备份对象）。示例如下：

```
kubectl patch backupstoragelocation default --namespace velero \
--type merge \
--patch '{"spec":{"accessMode":"ReadOnly"}}'
```

在集群 B 执行还原

还原前查看集群 B 资源

在集群 B 执行还原前，您可以查看集群 B 资源和服务情况，以便在还原集群之后用于 迁移结果核验。

在执行还原操作前，集群 B 中 default 、default2 命名空间下无任何工作负载资源。执行以下命令，可以查看集群 B 中两个命名空间下的 Pods 和 PVC 资源情况。如下图所示：

```
[root@VM-1-14-tlinux ~]# kubectl  get pod
No resources found in default namespace.
[root@VM-1-14-tlinux ~]# kubectl  get pod  -n default2
No resources found in default2 namespace.
[root@VM-1-14-tlinux ~]# kubectl  get pvc
No resources found in default namespace.
[root@VM-1-14-tlinux ~]#
```

还原集群

1. 执行以下命令，将集群 B 中 Velero 备份存储位置临时也更新为只读模式（非必须，可以防止在还原过程时 Velero 在备份存储位置中创建或删除备份对象）。示例如下：

```
kubectl patch backupstoragelocation default --namespace velero \
--type merge \
--patch '{"spec":{"accessMode":"ReadOnly"}}'
```

> 说明：
> 您可以指定在还原期间或还原资源后执行自定义 Hook 操作。例如，可能需要在数据库应用程序容器启动之前执行自定义数据库还原操作。了解 Hook 更多信息，请参见 还原 Hook。

2. 在还原操作之前，需确保集群 B 中 的 Velero 资源与对象存储 COS 中的备份文件同步。默认同步间隔是1分钟，可以使用 `--backup-sync-period` 来配置同步间隔。可以执行以下命令，查看集群 A 的备份是否已同步。

```
velero backup get <BACKUP-NAME>
```

3. 获取备份成功检查无误后，执行以下命令还原所有内容到集群 B 中。

```
velero restore create --from-backup <BACKUP-NAME>
```

还原过程如下图所示：

```
[root@VM-1-14-tlinux ~]# velero backup-location get
NAME      PROVIDER   BUCKET/PREFIX        PHASE       LAST VALIDATED                  ACCESS MODE
default   aws        jo[redacted]         Available   2020-11-18 16:49:43 +0800 CST   ReadOnly
[root@VM-1-14-tlinux ~]# velero backup get
NAME             STATUS      ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
backup-default   Completed   0        0          2020-11-18 16:18:21 +0800 CST   29d       default            <none>
default-all      Completed   0        0          2020-11-18 16:20:40 +0800 CST   29d       default            <none>
[root@VM-1-14-tlinux ~]# velero restore create --from-backup default-all
Restore request "default-all-20201118165141" submitted successfully.
Run `velero restore describe default-all-20201118165141` or `velero restore logs default-all-20201118165141` for more details.
[root@VM-1-14-tlinux ~]#
```

4. 等待还原任务完成后查看还原日志，执行以下命令查看还原是否有报错和跳过信息。示例如下：

```
# 查看迁移时是否有错误的还原信息
velero restore logs <BACKUP-NAME> | grep error
# 查看迁移时跳过的还原操作
velero restore logs <BACKUP-NAME> | grep skip
```

如下图所示，可以查看还原步骤未发生错误，但出现部分 “skipped” 步骤，因为在备份集群资源时备份了不包含 Velero 命名空间的所有集群资源，有一些同类型同名的集群资源已经存在，例如 kube-system下的集群资源。当还原过程中有资源冲突时，Velero 会跳过该还原步骤，实际上该还原过程正常，可以忽略 “skipped” 日志（在特殊情况可以分析该日志）。

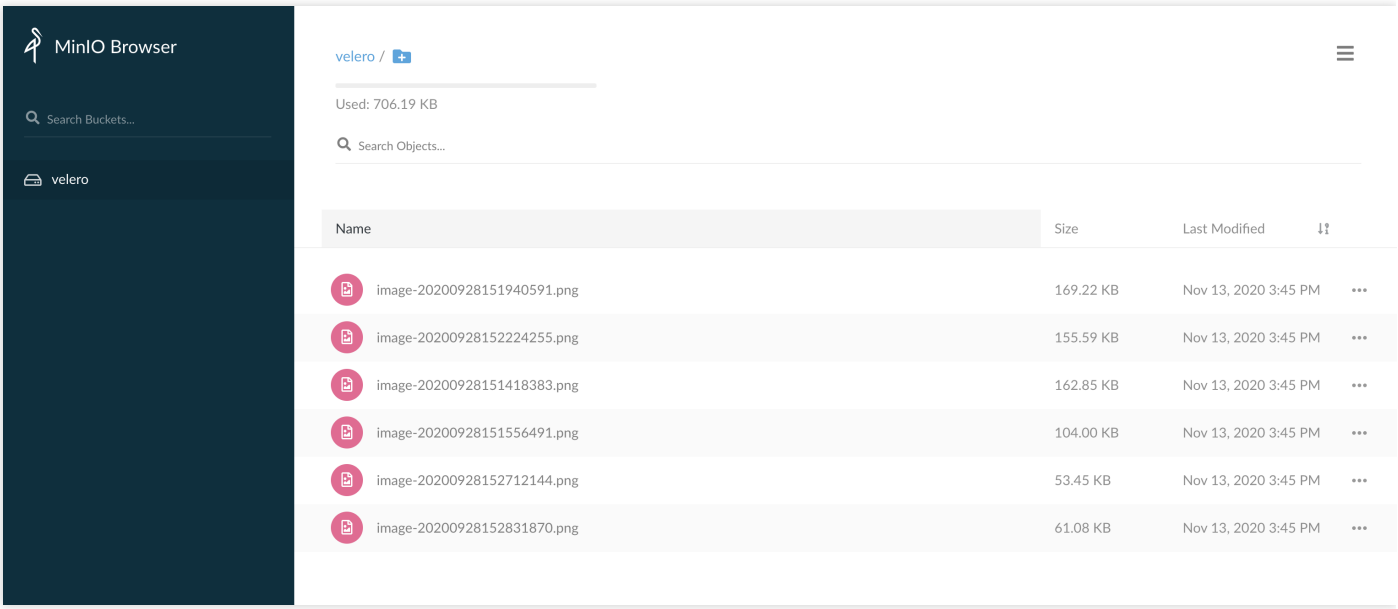
```
[root@VM-1-14-tlinux ~]# velero restore logs default-all-20201118165141 | grep error
[root@VM-1-14-tlinux ~]# velero restore logs default-all-20201118165141 | grep skip
time="2020-11-18T08:51:43Z" level=info msg="Restore of ConfigMap, coredns skipped: it already exists in the cluster and is the same as the backed up version" logSource
e="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:43Z" level=info msg="Restore of ConfigMap, tke-cni-agent-conf skipped: it already exists in the cluster and is the same as the backed up versio
n" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:44Z" level=info msg="Restore of ConfigMap, tke-ingress-controller-config skipped: it already exists in the cluster and is the same as the backe
d up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, lb-ingress-clusterrole-nisa-binding skipped: it already exists in the cluster and is the sa
me as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:attachdetach-controller skipped: it already exists in the cluster and is
the same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:certificate-controller skipped: it already exists in the cluster and is t
he same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:clusterrole-aggregation-controller skipped: it already exists in the clus
ter and is the same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:cronjob-controller skipped: it already exists in the cluster and is the s
ame as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:daemon-set-controller skipped: it already exists in the cluster and is th
e same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:deployment-controller skipped: it already exists in the cluster and is th
e same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:disruption-controller skipped: it already exists in the cluster and is th
e same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:endpoint-controller skipped: it already exists in the cluster and is the
same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:endpointslice-controller skipped: it already exists in the cluster and is
 the same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:expand-controller skipped: it already exists in the cluster and is the sa
me as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:generic-garbage-collector skipped: it already exists in the cluster and i
s the same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:horizontal-pod-autoscaler skipped: it already exists in the cluster and i
s the same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:job-controller skipped: it already exists in the cluster and is the same
as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:namespace-controller skipped: it already exists in the cluster and is the
 same as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
time="2020-11-18T08:51:52Z" level=info msg="Restore of ClusterRoleBinding, system:controller:node-controller skipped: it already exists in the cluster and is the same
 as the backed up version" logSource="pkg/restore/restore.go:1164" restore=velero/default-all-20201118165141
```

迁移结果核验

1. 执行以下命令，校验集群 B 执行迁移操作后的集群资源，可以看到 default 、default2 命名空间下的 Pods 和 PVC 资源已按预期迁移成功。如下图所示：

```
[root@VM-1-14-tlinux ~]# kubectl get pod
NAME                              READY   STATUS    RESTARTS   AGE
migrate-test-6bf6d577f4-7g7b9     1/1     Running   0          43m
minio-1605249781-66c4cbdfc-d7r4b  1/1     Running   0          43m
[root@VM-1-14-tlinux ~]# kubectl get pod -n default2
NAME                        READY   STATUS    RESTARTS   AGE
default2-58856dc878-kprhk   1/1     Running   0          44m
default2-58856dc878-zgnrw   1/1     Running   0          44m
[root@VM-1-14-tlinux ~]# kubectl get pvc
NAME               STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
minio-1605249781   Bound    pvc-3da9c96a-99aa-4367-b9d3-cb953ba4a57d   500Gi      RWO            cbs            44m
[root@VM-1-14-tlinux ~]#
```

2. 登录集群 B 中的 MinIO 服务，可以看到 MinIO 服务中的图片数据未丢失，说明持久卷数据已按预期迁移成功。



3. 至此已完成了 TKE 集群间资源的迁移。迁移操作完成后，执行以下命令，将集群 A 和 集群 B 的备份存储位置恢复为读写模式，以便在下次备份任务可以正常备份。示例如下：

```
kubectl patch backupstoragelocation default --namespace velero \
--type merge \
--patch '{"spec":{"accessMode":"ReadWrite"}}'
```

总结

本文主要介绍了在 TKE 集群间使用 Velero 迁移集群资源的原理、注意事项和操作方法，成功的将集群 A 中的集群资源无缝迁移到集群 B 中，整个迁移过程简单快捷，是一种非常友好的集群资源迁移方案。

使用 Velero 跨云平台迁移集群资源到 TKE

最近更新时间：2021-12-06 11:20:04

操作场景

开源工具 Velero（旧版本名称为 Heptio Ark）可以安全地备份和还原、执行灾难恢复以及迁移 Kubernetes 集群资源和持久卷。容器服务 TKE 支持使用 Velero 备份、还原和迁移集群资源，详情请参见 使用对象存储 COS 作为 Velero 存储实现集群资源备份和还原 和 在 TKE 中使用 Velero 迁移复制集群资源。本文将介绍如何使用 Velero 将自建或其他云平台 Kubernetes 集群无缝迁移到容器服务 TKE 平台。

迁移原理

使用 Velero 迁移自建或其他云平台集群架构的原理与 使用 Velero 迁移复制集群资源 过程的原理类似，迁移集群和被迁移集群需要都安装 Velero 实例，且指定同一个腾讯云 对象存储 COS 存储桶，被迁移集群按需执行备份，目标集群按需还原集群资源实现资源迁移。
不同的是，自建或其他云平台的集群资源迁移到 TKE 时，需要考虑和解决因跨平台导致集群环境差异问题，为此需要通过 Velero 提供的众多实用备份和还原策略帮助解决问题。

前提条件

- 已有自建或其他云平台 Kubernetes 集群（以下称作集群 A ），且集群版本需1.10以上。
- 已创建迁移目标的容器服务 TKE 集群（以下称作集群 B ），创建 TKE 集群请参见 创建集群。
- 集群 A 和 集群 B 都需要安装 Velero 实例（1.5版本以上），并且共用同一个腾讯云 COS 存储桶作为 Velero 后端存储，安装步骤请参见 配置存储和安装 Velero。
- 确保镜像资源在迁移后可以正常拉取。
- 确保两个集群的 Kubernetes 版本的 API 兼容，建议使用相同版本。

迁移指导

在进行迁移工作前，建议先理清迁移思路，制定详细的迁移计划，迁移过程中需要考虑以下几点：

展开全部

分析筛选哪些集群资源需要进行迁移

展开&收起
根据实际情况筛选分类出需要迁移资源清单和不需要迁移的资源清单。

根据业务场景考虑是否需要自定义 Hook 操作

展开&收起

- 在备份集群资源时，考虑是否需要在备份期间执行 备份 Hooks。例如，需要将正在运行的应用的内存数据落盘场景。
- 在还原（迁移）集群资源时，考虑是否需要在还原期间执行 还原 Hooks。例如，需要在还原前准备一些初始化工作。

按需编写备份和还原的命令或资源清单

展开&收起
根据筛选归类的资源清单编写备份和还原策略，推荐在复杂场景下使用创建资源清单的方式来执行备份和还原，YAML 资源清单比较直观且方便维护，参数指定的方式可以在简单迁移场景或测试时使用。

处理跨云平台资源的差异性

展开&收起
由于是跨云平台迁移，动态创建 PVC 的存储类等关系可能不同，需要提前规划动态 PVC/PV 存储类关系是否需要重新映射。需在还原操作前，创建相关映射的 ConfigMap 配置。如需解决更加个性化的差异，可以手动修改备份后的资源清单。

操作完成后核查迁移资源

展开&收起
检查迁移的集群资源是否符合预期且数据完整可用。

操作步骤

以下将介绍某云平台集群 A 中的资源迁移到 TKE 集群 B 中的详细操作步骤，其中涉及到 Velero 备份和还原基础知识，您可以查看本文 Velero 备份/还原实用知识 章节深入了解。

创建集群 A 示例资源

在某云平台集群 A 中部署 Velero 实例中含有 PVC 的 Nginx 工作负载，为方便起见可直接使用动态存储类来创建 PVC 和 PV。

1. 执行以下命令，查看当前集群支持的动态存储类信息。示例如下：

```
# 获取当前集群支持的存储类信息，其中 xxx-StorageClass 为存储类代名，xxx-Provider 为提供商代名，下同。
```

```
$ kubectl get sc
NAME PROVISIONER RECLAIMPOLICY VOLUMEBINDINGMODE ALLOWVOLUMEEXPANSION AGE
xxx-StorageClass xxx-Provider Delete Immediate true 3d3h
...
```

2. 修改 with-pv.yaml 文件中的 PVC 资源清单，使用集群中存储类名为 “xxx-StorageClass” 的存储类来动态创建。示例如下：

```
...
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: nginx-logs
namespace: nginx-example
labels:
app: nginx
spec:
# Optional: 修改 PVC 的存储类的值为某云平台
storageClassName: xxx-StorageClass
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 20Gi # 由于该云平台限制存储最小为20Gi，本示例需要同步修改此值为20Gi
...
```

3. 执行以下命令，应用示例中的 with-pv.yaml，创建如下的集群资源（nginx-example 命名空间）。示例如下：

```
$ kubectl apply -f with-pv.yaml
namespace/nginx-example created
persistentvolumeclaim/nginx-logs created
deployment.apps/nginx-deployment created
service/my-nginx created
```

4. 创建的 PVC “nginx-logs” 已挂载至 Nginx 容器的 `/var/log/nginx` 目录，作为服务的日志存储。本文示例通过在浏览器测试访问 Nginx 服务，为挂载的 PVC 生产日志数据，以便后续还原后进行数据比对。示例如下：

```
$ kubectl exec -it nginx-deployment-5ccc99bffb-6nm5w bash -n nginx-example
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future ver
sion. Use kubectl kubectl exec [POD] -- [COMMAND]
Defaulting container name to nginx.
```

```
Use 'kubectl describe pod/nginx-deployment-5ccc99bffb-6nm5w -n nginx-example'
to see all of the containers in this pod

$ du -sh /var/log/nginx/
84K /var/log/nginx/

# 查看 accss.log 和 error.log 前两条日志
$ head -n 2 /var/log/nginx/access.log
192.168.0.73 - - [29/Dec/2020:03:02:31 +0000] "GET /?spm=5176.2020520152.0.0.2
2d016ddHXZumX HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10
_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.3
6" "-"
192.168.0.73 - - [29/Dec/2020:03:02:32 +0000] "GET /favicon.ico HTTP/1.1" 404
555 "http://47.242.233.22/?spm=5176.2020520152.0.0.22d016ddHXZumX" "Mozilla/5.
0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) C
hrome/87.0.4280.88 Safari/537.36" "-"

$ head -n 2 /var/log/nginx/error.log
2020/12/29 03:02:32 [error] 6#6: *597 open() "/usr/share/nginx/html/favicon.ic
o" failed (2: No such file or directory), client: 192.168.0.73, server: localh
ost, request: "GET /favicon.ico HTTP/1.1", host: "47.242.233.22", referrer: "h
ttp://47.242.233.22/?spm=5176.2020520152.0.0.22d016ddHXZumX"
2020/12/29 03:07:21 [error] 6#6: *1172 open() "/usr/share/nginx/html/0bef" fai
led (2: No such file or directory), client: 192.168.0.73, server: localhost, r
equest: "GET /0bef HTTP/1.0"
```

确认需要迁移的资源清单

1. 执行以下命令，输出集群 A 中所有的资源清单列表。

```
kubectl api-resources --verbs=list -o name | xargs -n 1 kubectl get --show-kind
--ignore-not-found --all-namespaces
```

您也可以执行以下命令，根据资源区分命名空间，缩小输出的资源范围：

- 查看不区分命名空间的资源清单列表：

```
kubectl api-resources --namespaced=false --verbs=list -o name | xargs -n 1 ku
bectl get --show-kind --ignore-not-found
```

- 查看区分命名空间的资源清单列表：

```
kubectl api-resources --namespaced=true --verbs=list -o name | xargs -n 1 kub
ectl get --show-kind --ignore-not-found --all-namespaces
```

2. 可以根据实际情况筛选出需要被迁移的资源清单。本文示例将直接从该云平台迁移 “nginx-example” 命名空间下 Nginx 工作负载相关的资源到容器服务 TKE，涉及资源如下所示：

```
$ kubectl get all -n nginx-example
NAME READY STATUS RESTARTS AGE
pod/nginx-deployment-5ccc99bffb-tn2sh 2/2 Running 0 2d19h

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/my-nginx LoadBalancer 172.21.1.185 x.x.x.x 80:31455/TCP 2d19h

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/nginx-deployment 1/1 1 1 2d19h

NAME DESIRED CURRENT READY AGE
replicaset.apps/nginx-deployment-5ccc99bffb 1 1 1 2d19h

$ kubectl get pvc -n nginx-example
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
nginx-logs Bound d-j6ccrq4k1moziu1l6l5r 20Gi RWO xxx-StorageClass 2d19h

$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
d-j6ccrq4k1moziu1l6l5r 20Gi RWO Delete Bound nginx-example/nginx-logs xxx-Stor
ageClass 2d19h
```

确认 Hook 策略

本文示例在 with-pv.yaml 中已配置“备份 Nginx 工作负载前将文件系统设置为只读，在备份后恢复读写”的 Hook 策略，YAML 文件如下所示：

```
...
annotations:
# 备份 Hook 策略的注解表示：在开始备份之前将 nginx 日志目录设置为只读模式，备份完成后恢复读写
模式
pre.hook.backup.velero.io/container: fsfreeze
pre.hook.backup.velero.io/command: '["/sbin/fsfreeze", "--freeze", "/var/log/ngi
nx"]'
post.hook.backup.velero.io/container: fsfreeze
post.hook.backup.velero.io/command: '["/sbin/fsfreeze", "--unfreeze", "/var/log/
nginx"]'
```

```
spec:
volumes:
- name: nginx-logs
persistentVolumeClaim:
claimName: nginx-logs
containers:
- image: nginx:1.17.6
name: nginx
ports:
- containerPort: 80
volumeMounts:
- mountPath: "/var/log/nginx"
name: nginx-logs
readOnly: false
- image: ubuntu:bionic
name: fsfreeze
securityContext:
privileged: true
volumeMounts:
- mountPath: "/var/log/nginx"
name: nginx-logs
...
```

开始迁移操作

以下将根据实际情况编写备份和还原策略，开始迁移该云平台的 Nginx 工作负载相关资源。

在集群 A 执行备份

1. 创建如下 YAML 文件，备份需要迁移的资源。

```
apiVersion: velero.io/v1
kind: Backup
metadata:
name: migrate-backup
# 必须得是 velero 安装的命名空间
namespace: velero
spec:
# 仅包含 nginx-example 命名空间的资源
includedNamespaces:
- nginx-example
# 包含不区分命名空间的资源
includeClusterResources: true
# 备份数据存储位置指定
storageLocation: default
# 卷快照存储位置指定
```

```
volumeSnapshotLocations:
- default
# 使用 restic 备份卷
defaultVolumesToRestic: true
```

2. 执行备份过程如下所示，当备份状态为 “Completed” 且 errors 数为0时表示备份过程完整无误。示例如下：

```
$ kubectl apply -f backup.yaml
backup.velero.io/migrate-backup created
$ velero backup get
NAME STATUS ERRORS WARNINGS CREATED EXPIRES STORAGE LOCATION SELECTOR
migrate-backup InProgress 0 0 2020-12-29 19:24:12 +0800 CST 29d default <none>
$ velero backup get
NAME STATUS ERRORS WARNINGS CREATED EXPIRES STORAGE LOCATION SELECTOR
migrate-backup Completed 0 0 2020-12-29 19:24:28 +0800 CST 29d default <none>
```

3. 备份完成后执行以下命令，将备份存储位置临时更新为只读模式。示例如下：

> 说明
> 非必须，可以防止在还原过程时， Velero 在备份存储位置中创建或删除备份对象。

```
kubectl patch backupstoragelocation default --namespace velero \
--type merge \
--patch '{"spec":{"accessMode":"ReadOnly"}}'
```

处理跨云平台资源的差异性

1. 由于使用的动态存储类存在差异，需要通过如下所示的 ConfigMap 为持久卷 “nginx-logs” 创建动态存储类名映射。示例如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
name: change-storage-class-config
namespace: velero
labels:
velero.io/plugin-config: ""
velero.io/change-storage-class: RestoreItemAction
```

```
data:
# 存储类名映射到腾讯云动态存储类 cbs
xxx-StorageClass: cbs
```

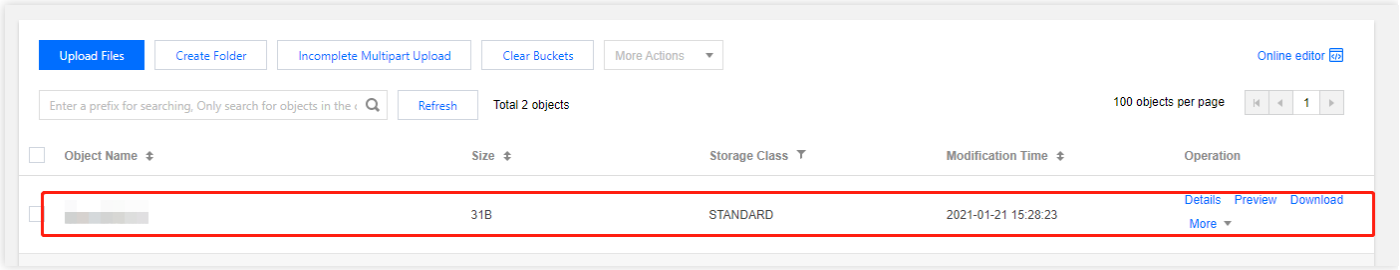
2. 执行以下命令，应用上述的 ConfigMap 配置。示例如下：

```
$ kubectl apply -f cm-storage-class.yaml
configmap/change-storage-class-config created
```

3. Velero 备份的资源清单以 JSON 格式存放在对象存储 COS 中，如有更加个性化的迁移需求，可以直接下载备份文件并自定义修改。本示例将为 Nginx 的 Deployment 资源自定义添加一个 “jokey-test:jokey-test” 注解，修改过程如下：

```
$ Downloads % mkdir migrate-backup
# 解压备份文件
$ Downloads % tar -zxvf migrate-backup.tar.gz -C migrate-backup
# 编辑修改需要自定义的资源，本示例为 Nginx 的 Deployment 资源添加 "jokey-test":"jokey-test" 的注解项
$ migrate-backup % cat resources/deployments.apps/namespaces/nginx-example/nginx-deployment.json
{"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{"jokey-test":"jokey-test",...
# 重新打包修改后的备份文件
$ migrate-backup % tar -zcvf migrate-backup.tar.gz *
```

4. 完成自定义修改并重新打包，登录 对象存储 COS 控制台上传替换原有备份文件。如下图所示：



在集群 B 执行还原

1. 本文示例使用如下所示的资源清单执行还原操作（迁移）：

```
apiVersion: velero.io/v1
kind: Restore
metadata:
```

```
name: migrate-restore
namespace: velero
spec:
backupName: migrate-backup
includedNamespaces:
- nginx-example
# 按需填写需要恢复的资源类型，nginx-example 命名空间下没有想要排除的资源，所以这里直接写
'*'
includedResources:
- '*'
includeClusterResources: null
# 还原时不包含的资源，这里额外排除 StorageClasses 资源类型。
excludedResources:
- storageclasses.storage.k8s.io
# 使用 labelSelector 选择器选择具有特定 label 的资源，由于此示例中无须再使用 label 选择
器筛选，这里先注释。
# labelSelector:
# matchLabels:
# app: nginx
# 设置命名空间关系映射策略
namespaceMapping:
nginx-example: default
restorePVs: true
```

2. 执行还原过程如下所示，当还原状态显示为 “Completed” 且 “errors” 数为0时表示还原过程完整无误。示例如下：

```
$ kubectl apply -f restore.yaml
restore.velero.io/migrate-restore created
$ velero restore get
NAME BACKUP STATUS STARTED COMPLETED ERRORS WARNINGS CREATED SELECTOR
migrate-restore migrate-backup Completed 2021-01-12 20:39:14 +0800 CST 2021-01-
12 20:39:17 +0800 CST 0 0 2021-01-12 20:39:14 +0800 CST <none>
```

迁移资源核查

1. 执行以下命令，查看被迁移的资源的运行状态是否正常。示例如下：

```
# 由于在还原时指定了 "nginx-example" 命名空间映射到 "default" 命名空间，所以还原的资源将
运行在 "default" 命名空间下
$ kubectl get all -n default
NAME READY STATUS RESTARTS AGE
pod/nginx-deployment-5ccc99bffb-6nm5w 2/2 Running 0 49s
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kube-user LoadBalancer 172.16.253.216 10.0.0.28 443:30060/TCP 8d
service/kubernetes ClusterIP 172.16.252.1 <none> 443/TCP 8d
```

```
service/my-nginx LoadBalancer 172.16.254.16 x.x.x.x 80:30840/TCP 49s
NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/nginx-deployment 1/1 1 1 49s
NAME DESIRED CURRENT READY AGE
replicaset.apps/nginx-deployment-5ccc99bffb 1 1 1 49s
```

从命令执行结果可以查看出被迁移的资源的运行状态正常。

2. 核查设置的还原策略是否成功。

i. 执行以下命令，核查动态存储类名映射是否正确。示例如下：

```
# 可以看到 PVC/PV 的存储类已经是 "cbs", 说明存储类映射成功
$ kubectl get pvc -n default
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
nginx-logs Bound pvc-bcc17ccd-ec3e-4d27-bec6-b0c8f1c2fa9c 20Gi RWO cbs 55s
$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AG
E
pvc-bcc17ccd-ec3e-4d27-bec6-b0c8f1c2fa9c 20Gi RWO Delete Bound default/nginx-
logs cbs 57s
```

若 PVC/PV 的存储类为 “cbs”，则说明存储类映射成功。从上述命令执行结果可以查看出存储类映射成功。

ii. 执行以下命令，查看还原前为 “deployment.apps/nginx-deployment” 自定义添加的 “jokey-test” 注解是否成功。示例如下：

```
# 获取注解 “jokey-test” 成功, 说明自定义修改资源成功。
$ kubectl get deployment.apps/nginx-deployment -o custom-columns=annotation
s:.metadata.annotations.jokey-test
annotations
jokey-test
```

若可以正常获取注解，则说明成功修改自定义资源。从上述命令执行结果可以查看出命名空间映射配置成功。

3. 执行以下命令，检查工作负载挂载的 PVC 数据是否成功迁移。

```
# 查看挂载的 PVC 数据目录中的数据大小, 显示为88K, 比迁移前多, 原因是腾讯云 CLB 主动发起健康
检查产生了一些日志
$ kubectl exec -it nginx-deployment-5ccc99bffb-6nm5w -n default -- bash
```

```
Defaulting container name to nginx.
Use 'kubectl describe pod/nginx-deployment-5ccc99bffb-6nm5w -n default' to see
all of the containers in this pod.

$ du -sh /var/log/nginx
88K /var/log/nginx

# 查看前两条日志信息，和迁移前一致，大致说明 PVC 数据未丢失
$ head -n 2 /var/log/nginx/access.log
192.168.0.73 - - [29/Dec/2020:03:02:31 +0000] "GET /?spm=5176.2020520152.0.0.2
2d016ddHXZumX HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10
_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.3
6" "-"
192.168.0.73 - - [29/Dec/2020:03:02:32 +0000] "GET /favicon.ico HTTP/1.1" 404
555 "http://47.242.233.22/?spm=5176.2020520152.0.0.22d016ddHXZumX" "Mozilla/5.
0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) C
hrome/87.0.4280.88 Safari/537.36" "-"

$ head -n 2 /var/log/nginx/error.log
2020/12/29 03:02:32 [error] 6#6: *597 open() "/usr/share/nginx/html/favicon.ic
o" failed (2: No such file or directory), client: 192.168.0.73, server: localh
ost, request: "GET /favicon.ico HTTP/1.1", host: "47.242.233.22", referrer: "h
ttp://47.242.233.22/?spm=5176.2020520152.0.0.22d016ddHXZumX"
2020/12/29 03:07:21 [error] 6#6: *1172 open() "/usr/share/nginx/html/0bef" fai
led (2: No such file or directory), client: 192.168.0.73, server: localhost, r
equest: "GET /0bef HTTP/1.0"
```

从上述命令结果可以查看出，工作负载挂载的 PVC 数据成功迁移。至此，本文示例成功迁移某云平台集群 A 的 Nginx （nginx-example 命名空间）工作负载相关资源和数据到容器服务 TKE 集群 B （default 命名空间）中。

总结

本文主要介绍使用 Velero 迁移自建或其他云平台集群到 TKE 的思路和方法步骤，成功的将集群 A 中的集群资源无缝迁移到集群 B 中。若在实际迁移过程中遇到未覆盖到的场景，欢迎 提交工单 咨询和讨论迁移解决方案。

附录：Velero 备份/还原实用知识

Velero 提供众多非常实用的备份和还原策略，详细介绍如下：

资源过滤相关

当不使用任何筛选选项时，Velero 会将所有对象包括在备份或还原操作中，在**备份和还原**时可以指定参数按需过滤资源。详情请参见 资源过滤。

- **包含关系的过滤参数：**

参数	参数含义
`--include-resources`	指定需要包含的资源对象列表。
`--include-namespaces`	指定需要包含的命名空间列表。
`--include-cluster-resources`	指定是否要包含集群的资源。
`--selector`	指定包含与标签选择器匹配的资源。

- **不包含关系的过滤参数：**

参数	参数含义
`--exclude-namespaces`	指定需要排除的命名空间列表。
`--exclude-resources`	指定需要排除的资源对象列表。
`velero.io/exclude-from-backup=true`	此配置项为资源对象配置 label 属性，添加了此 label 配置项的资源对象将会排除在外。

Hook 操作相关

- 在**备份期间**执行 Hook 操作，例如，需要在备份前将内存数据落盘，详情请参见 备份 Hooks。
- 在**还原期间**执行 Hook 操作，例如，在还原前判断组件依赖是否可用，详情请参见 还原 Hooks。
- 在**还原时**配置 PVC/PV 卷相关映射关系配置可参考以下文档。如需了解更多请参见 还原参考。
  - 配置 PV/PVC 存储类映射
  - 配置 PVC 绑定节点映射

Restic 备份卷配置

从 Velero 1.5版本开始，Velero 默认使用 Restic 备份所有 Pod 卷，而不必单独注释每个 Pod，**推荐使用 Velero 1.5 以上版本**。

在 Velero 1.5版本之前，Velero 使用 Restic 在备份卷时，Restic 提供以下两种方式发现需要备份的 Pod 卷：

- 使用的 Pod 卷备份选择包含注解（默认）：

```
kubectl -n <YOUR_POD_NAMESPACE> annotate <pod/YOUR_POD_NAME> backup.velero.io/backup-volumes=<YOUR_VOLUME_NAME_1,YOUR_VOLUME_NAME_2,...>
```

- 使用的 Pod 卷备份选择不包含注解：

```
kubectl -n <YOUR_POD_NAMESPACE> annotate <pod/YOUR_POD_NAME> backup.velero.io/backup-volumes-excludes=<YOUR_VOLUME_NAME_1,YOUR_VOLUME_NAME_2,...>
```

相关命令

- 备份完成后可执行以下命令，查看备份卷信息：

```
kubectl -n velero get podvolumebackups -l velero.io/backup-name=<YOUR_BACKUP_NAME> -o yaml
```

- 还原完成后可执行以下命令，查看还原卷信息：

```
kubectl -n velero get podvolumerestores -l velero.io/restore-name=<YOUR_RESTORE_NAME> -o yaml
```

其他操作

- 除使用 Velero 命令执行备份操作，也可以通过**创建备份资源来触发（推荐）**，配置示例请参见 备份示例，API 详细字段定义可参见 备份 API 定义。
- 除使用 Velero 命令执行还原操作，也可以通过**创建还原资源来触发（推荐）**，配置示例请参见 还原示例，API 详细字段定义可参见 还原 API 定义。
- 如有 annonations 、label 等其他个性化资源配置差异，可以在还原前手动编辑备份的 JSON 资源清单文件。

TKE 托管集群迁移至 Serverless 集群

最近更新时间：2022-12-13 18:23:37

前提条件

- 已有容器服务 TKE 托管集群（以下称作集群 A ），且集群版本需 >= 1.18 及以上。
- 已创建迁移目标的 TKE Serverless 集群（以下称作集群 B），集群版本需 >= 1.20 及以上，创建 TKE Serverless 集群请参见 创建集群。
- 集群 A 和 集群 B 需要共用同一个腾讯云 COS 存储桶作为 Velero 后端存储，配置 COS 存储桶请参见 配置对象存储。
- 集群 A 和 集群 B 建议在同一 VPC 下（如果需要备份 PVC 中的数据，必须在同一 VPC 下）。
- 确保镜像资源在迁移后可以正常拉取，在 TKE Serverless 集群中配置镜像仓库请参见 镜像仓库相关。
- 确保两个集群的 Kubernetes 版本的 API 兼容，建议使用相同版本。若集群 A 的集群版本较低，建议先升级集群 A 集群版本后，再进行迁移操作。

迁移限制

- 在 TKE 集群中启用固定 IP 特性的工作负载，在迁移到 TKE Serverless 集群后，IP 会发生改变。可以在 Pod Template 中指定 IP 创建 Pod，使用方式示例：`eks.tke.cloud.tencent.com/pod-ip: "xx.xx.xx.xx"`
- TKE Serverless 集群使用 containerd 作为运行时，与 docker 不一致，不兼容 Docker registry v2.5以下版本、harbor v1.10以下的版本的镜像。
- TKE Serverless 集群中，每个 Pod 默认分配20Gi的临时磁盘空间，用于镜像存储，该盘随 Pod 的生命周期创建和销毁。若需使用更大磁盘空间，可以挂载其他类型的 volume 作数据存储，例如使用 PVC。
- 在 TKE Serverless 集群中部署 DaemonSet 类型的工作负载时，需使用 sidecar 方式部署在业务 Pod 中。
- TKE Serverless 集群部署 NodePort 类型的服务时，无法通过 NodeIP:Port 访问服务，需要通过 ClusterIP:Port 访问服务。
- 部署在 TKE Serverless 集群上的 Pod 默认会通过9100端口，对外暴露监控数据。如果业务 Pod 本身需要监听9100端口，则可以在创建 Pod 时，在 Pod Template 中指定其他端口收集监控数据，避免跟业务的9100端口冲突。配置方式示例：`eks.tke.cloud.tencent.com/metrics-port: "9110"`
- 除以上限制外，务必阅读 TKE Serverless 集群其他说明。

迁移步骤

以下将介绍 TKE 集群 A 中的资源迁移到 TKE Serverless 集群 B 中的详细操作步骤。

配置对象存储

操作步骤请参见 创建存储桶。

下载 velero

1. 下载 Velero 最新版本安装包到集群环境中，本文以 v1.8.1 版本为例。

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.8.1/velero-v1.8.1-linux-amd64.tar.gz
```

2. 执行以下命令解压安装包，安装包提供 Velero 命令行执行文件和一些示例文件。

```
tar -xvf velero-v1.8.1-linux-amd64.tar.gz
```

3. 执行以下命令，将 Velero 可执行文件从解压后的目录迁移到系统环境变量目录下直接使用，本文以迁移至 `/usr/bin` 目录为例。示例如下：

```
cp velero-v1.8.1-linux-amd64/velero /usr/bin/
```

在集群 A 和集群 B 中安装 velero

1. 配置 velero 客户端，开启 CSI 特性。

```
velero client config set features=EnableCSI
```

2. 执行以下命令在集群 A 和集群 B 中安装 Velero ，创建 Velero 工作负载以及其他必要的资源对象。

- 使用 CSI 备份 PVC 的示例如下：

```
velero install --provider aws \
--plugins velero/velero-plugin-for-aws:v1.1.0,velero/velero-plugin-for-csi:v0.2.0 \
--features=EnableCSI \
--features=EnableAPIGroupVersions \
--bucket <BucketName> \
--secret-file ./credentials-velero \
--use-volume-snapshots=false \
```

```
--backup-location-config region=ap-guangzhou,s3ForcePathStyle="true",s3Url=http
s://cos.ap-guangzhou.myqcloud.com
```

注意：
TKE Serverless 集群不支持部署 Daemonset，因此本文示例都不支持使用 restic 插件。

- 如不需要备份 PVC，安装示例如下：

```
./velero install --provider aws --use-volume-snapshots=false --bucket gtest-125
1707795 --plugins velero/velero-plugin-for-aws:v1.1.0 --secret-file ./credentia
ls-velero --backup-location-config region=ap-guangzhou,s3ForcePathStyle="true",
s3Url=https://cos.ap-guangzhou.myqcloud.com
```

安装参数说明详情见 velero 安装参数，您也可以使用命令 `velero install --help` 查看。
其他安装参数说明：

安装参数	参数说明
--plugins	使用 AWS S3 兼容 API 插件 “velero-plugin-for-aws”；使用 CSI 插件 velero-plugin-for-csi 对 csi-pv 进行备份，建议开启。
--features	启用可选功能：启用 API 组版本功能 该功能用于兼容不同 API 组版本，建议开启；启用 CSI 快照功能 该功能用于备份 CSI 支持的 PVC，建议开启。
--use-restic	Velero 支持使用免费开源备份工具 Restic 备份和还原 Kubernetes 存储卷数据 （不支持 `hostPath` 卷，详情请参见 Restic 限制），该集成是 Velero 备份功能的补充，在迁移 TKE Serverless 集群的场景下，开启该参数会导致备份失败。
--use-volume-snapshots=false	关闭默认存储卷快照备份

3. 安装完成后，等待 Velero 工作负载就绪。执行以下命令，查看配置的存储位置是否可用，若显示 “Avaliable”，则说明集群可正常访问对象存储 COS。

```
velero backup-location get
NAME PROVIDER BUCKET/PREFIX PHASE LAST VALIDATED ACCESS MODE DEFAULT
default aws <BucketName> Available 2022-03-24 21:00:05 +0800 CST ReadWrite true
```

至此，Velero 安装完成。了解 Velero 更多安装介绍，请参见 Velero 官网文档。

（可选） 在集群 A 和集群 B 中安装 VolumeSnapshotClass 对象

说明：

- 如不需要备份 PVC，可跳过该步骤。
- 更多存储快照相关功能，请参见 使用 CBS CSI 插件对 PVC 进行备份与恢复。

1. 确认已安装 CBS-CSI 插件。

2. 在 访问管理 控制台完成对 `TKE_QCSRole` 角色授予 CBS 快照操作的相关权限，详情请参考 快照授权。

3. 使用以下 YAML，创建 VolumeSnapshotClass 对象。示例如下：

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
labels:
velero.io/csi-volumesnapshot-class: "true"
name: cbs-snapclass
driver: com.tencent.cloud.csi.cbs
deletionPolicy: Delete
```

4. 执行以下命令，检查 VolumeSnapshotClass 是否创建成功。示例如下：

```
$ kubectl get volumesnapshotclass
NAME DRIVER DELETIONPOLICY AGE
cbs-snapclass com.tencent.cloud.csi.cbs Delete 17m
```

（可选） 创建集群 A 示例资源

说明：
如不需要备份 PVC，可跳过该步骤。

在集群 A 中部署 Velero 实例中含有 PVC 的 minio 工作负载，这里使用 cbs-csi 动态存储类来创建 PVC 和 PV。

1. 使用集群中 provisioner 为 `com.tencent.cloud.csi.cbs` 的存储类来动态创建 pv。pvc 示例如下：

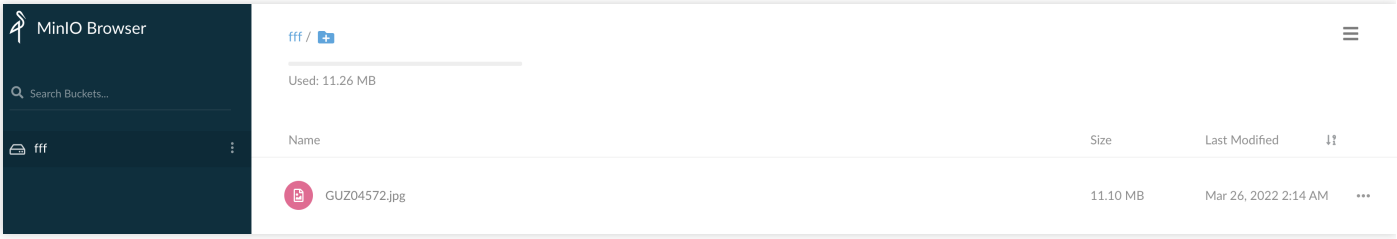
```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
annotations:
volume.beta.kubernetes.io/storage-provisioner: com.tencent.cloud.csi.cbs
name: minio
spec:
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 10Gi
storageClassName: cbs-csi
volumeMode: Filesystem
```

2. 使用 Helm 工具，创建一个引用上述 pvc 的 MinIO 测试服务，MinIO 安装方式请参见 MinIO 安装。在此示例中，已经为 MinIO 服务绑定了负载均衡器，可以在浏览器中使用公网地址访问管理页面。

```
[root@VM-0-28-tlinux ~]# kubectl get pod | grep minio
minio-1605249781-66c4cbdfc-d7r4b   1/1     Running   0          30h
[root@VM-0-28-tlinux ~]# kubectl get svc | grep minio
minio-1605249781   LoadBalancer   [redacted]   9000:30252/TCP   31h
[root@VM-0-28-tlinux ~]#
```

3. 登录 MinIO Web 管理页面，上传用于测试的图片。如下图所示：

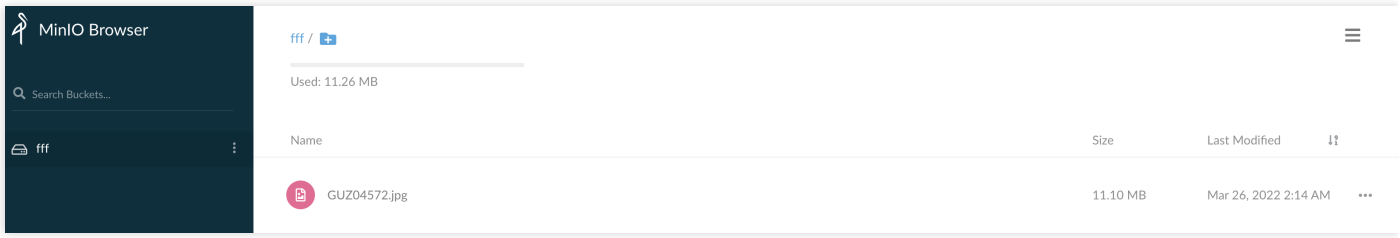


备份与还原

1. 在集群 A 创建备份，请参见**集群迁移**操作步骤中的 集群 A 创建备份。
2. 在集群 B 还原备份，请参见**集群迁移**操作步骤中的 集群 B 执行还原。
3. 迁移结果核验：
   - 如不需要备份 PVC，请参见**集群迁移**操作步骤中的 迁移结果核验。
   - 如需要备份 PVC，参照以下步骤进行核验：
4. 执行以下命令，校验集群 B 执行迁移操作后的集群资源，可以看到 Pods、PVC、Service 资源已按预期迁移成功。如下图所示：

```
[redacted] ~]$ k get po
NAME                     READY   STATUS    RESTARTS   AGE
minio[redacted]          1/1     Running   0          59s
[redacted] ~]$ k get pvc
NAME    STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
minio   Bound    pvc-b608cce1-[redacted]                    10Gi       RWO            cbs            5d16h
[redacted] ~]$ k get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubernetes   ClusterIP      192.168.0.1     <none>           443/TCP          5d19h
minio        LoadBalancer   192.168.1.103   [redacted]       9000:[redacted]/TCP   52s
```

5. 登录集群 B 中的 MinIO 服务，可以看到 MinIO 服务中的图片数据未丢失，说明持久卷数据已按预期迁移成功。



6. 至此已完成了 TKE 集群与 TKE Serverless 集群间资源的迁移。
   迁移操作完成后，执行以下命令，将集群 A 和 集群 B 的备份存储位置恢复为读写模式，以便在下次备份任务可以正常备份。示例如下：

```
kubectl patch backupstoragelocation default --namespace velero \
    --type merge \
    --patch '{"spec":{"accessMode":"ReadWrite"}}'
```

使用 Serverless 集群常见问题

- 拉取镜像失败：请参见 镜像仓库。
- 域名解析失败：常见于 Pod 镜像拉取失败、投递日志到自建 kafka 失败，请参见 Serverless 集群自定义 DNS 服务。
- 日志投递到 CLS 失败：首次使用 TKE Serverless 集群投递日志到 CLS，需要为服务授权，请参见 首次授权。
- 每个集群默认仅可创建 100 个 Pod，若需要创建超过配额的资源，请参见 默认配额。
- Pod 频繁被销毁重建，报错 `Timeout to ensure pod sandbox` ：TKE Serverless 集群 Pod 内的组件会与管控面通讯以保持健康检测，当 Pod 创建完后，Pod 持续 6 分钟网络不通，则会被管控面发起销毁重建。此时需检查 Pod 关联的安全组是否放通了 169.254 路由的访问。

- Pod 端口访问不通 / not ready：
  - 业务容器端口是否与 TKE Serverless 集群管控面端口有冲突，请参见 端口限制
  - Pod 可以 ping 成功，但是 telnet 失败，检查安全组。
- 创建实例时，可以使用如下特性加快拉取镜像速度：请参见 镜像缓存 与 镜像复用。
- 业务日志转存：Serverless 容器服务 job 类型的业务在退出后，底层资源就被回收，此时 Kubectl logs 无法查看容器日志，对于需要 debug 的场景不友好。可通过延迟销毁或者设置 terminationMessage 字段将业务日志转存，请参见 设置容器终止消息。
- Pod 频繁重启，报错 `ImageGCFailed` ：TKE Serverless 集群 Pod 默认磁盘大小为 20GiB, 如果磁盘使用空间达到 80%，TKE Serverless 集群管控面就会触发容器镜像的回收流程，尝试回收未使用的容器镜像来释放磁盘空间。如果未能释放任何空间，则会有一条事件提醒： `ImageGCFailed: failed to garbage collect required amount of images` ，提醒用户磁盘空间不足。常见磁盘空间不足的原因有：
  - 业务有大量临时输出。
  - 业务持有已删除的文件描述符，导致磁盘空间未释放。

参考文档

- 在 Velero 中使用 csi
- 在 Velero 中启用 API 组版本功能
- 使用应用市场安装 Minio

Serverless 集群
通过 NAT 网关访问外网

最近更新时间：2023-05-04 10:46:00

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。

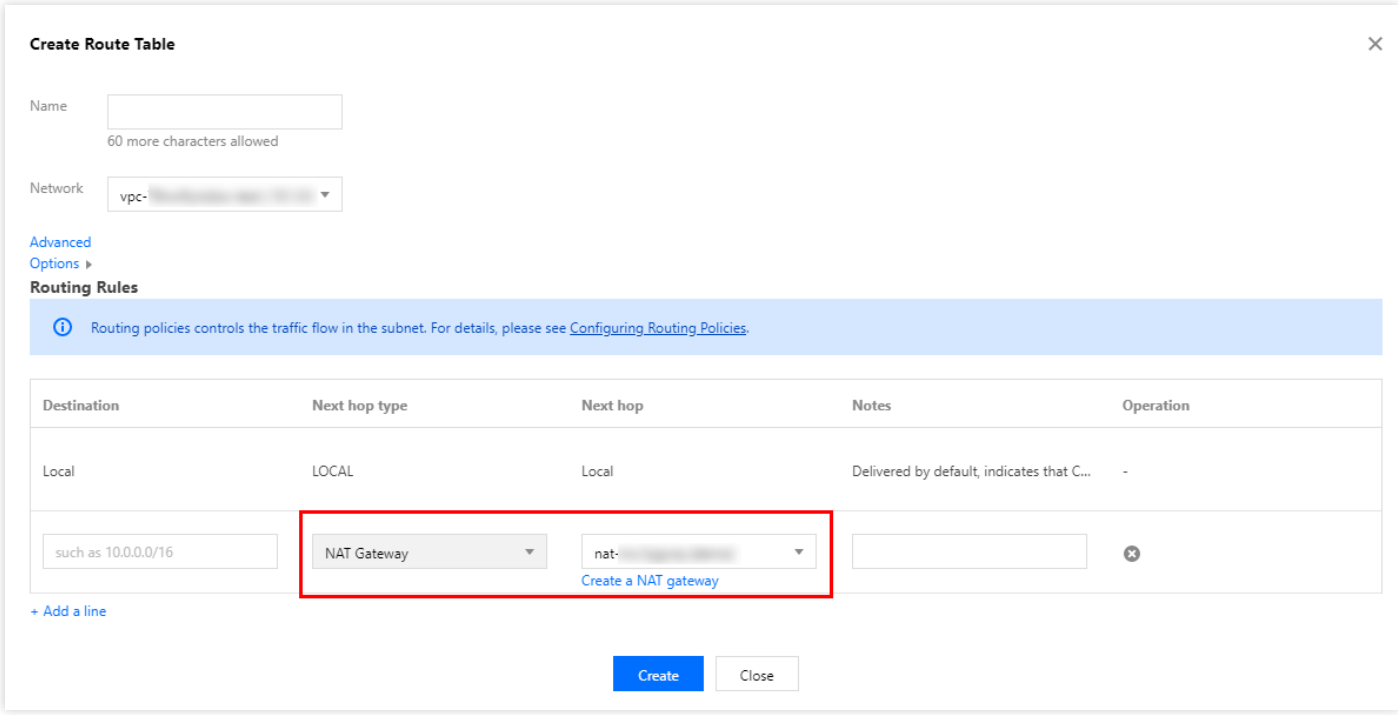
操作步骤

创建 NAT 网关

1. 登录腾讯云私有网络控制台，选择左侧导航栏中的 **NAT 网关** 。
2. 在 “NAT网关”页面中，单击 **+新建** 。
3. 在弹出的“新建NAT网关”窗口中参考 创建 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. 选择左侧导航栏中的 **路由表** ，进入“路由表”管理页面。
2. 在“路由表”管理页面，单击 **+新建** 。
3. 在弹出的“新建路由表”窗口中，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

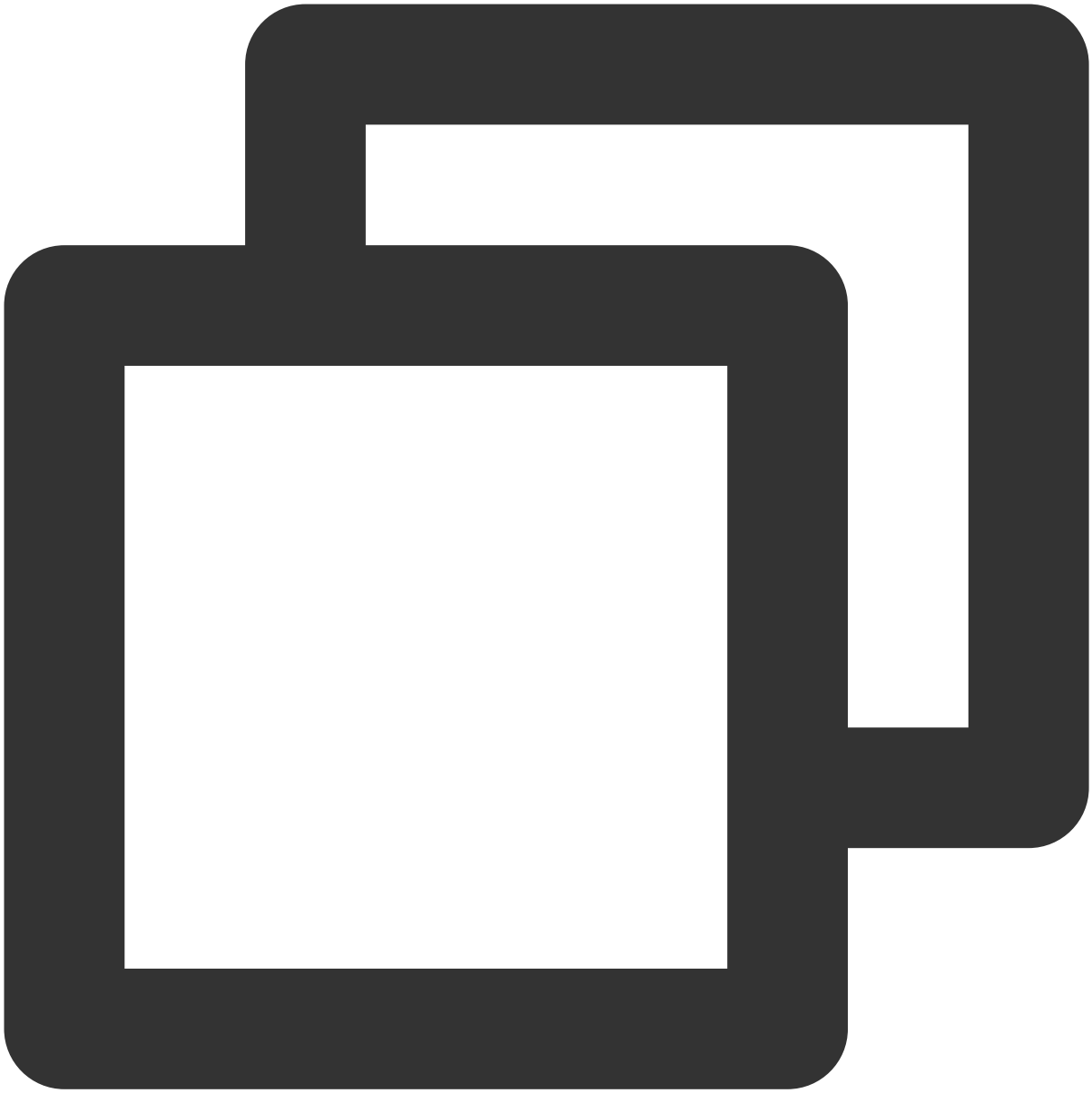
最近更新时间：2023-03-14 18:19:11

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：

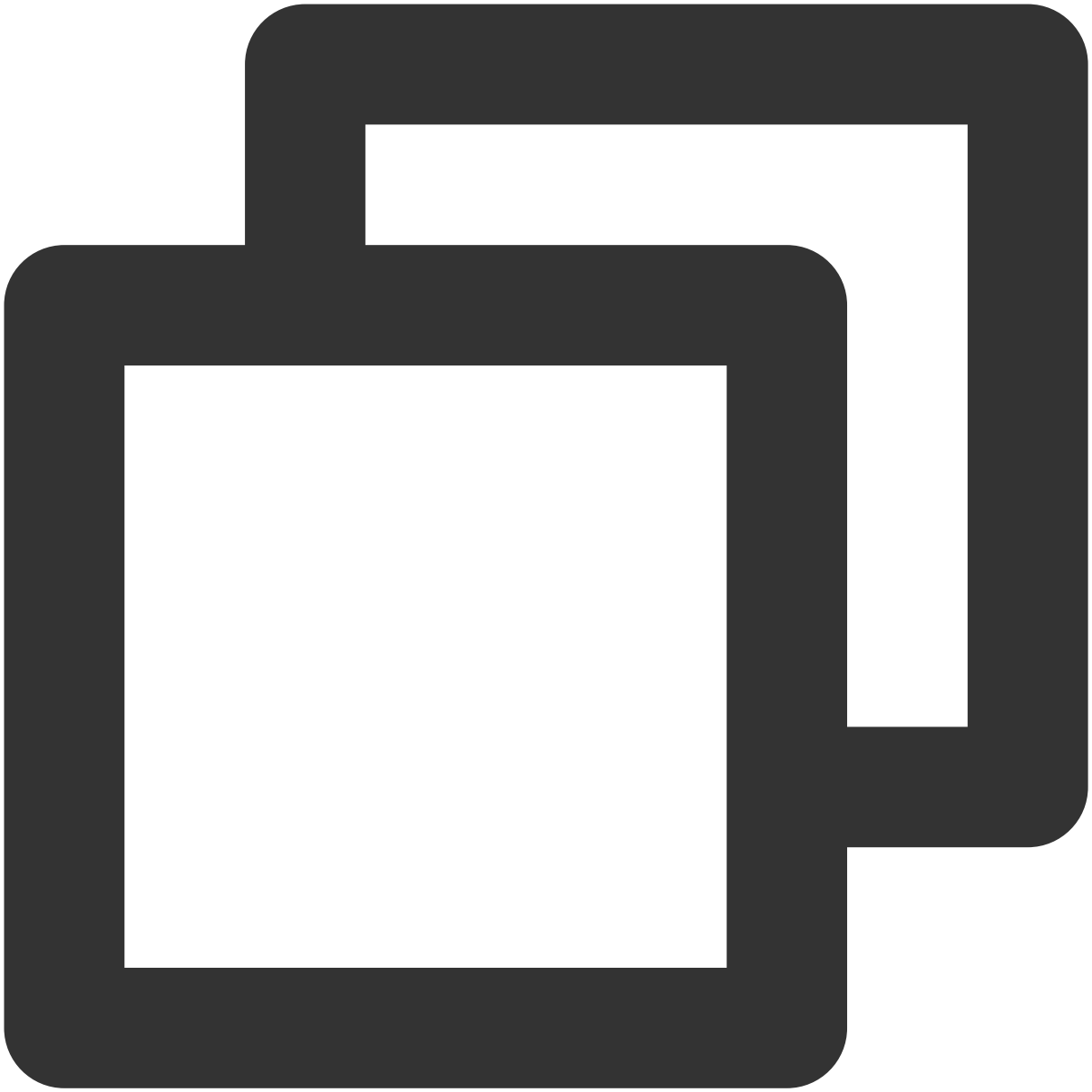
Annotation Key	Annotation Value 及描述	是否必填
eks.tke.cloud.tencent.com/eip-attributes	表明该 Workload 的 Pod 需要关联 EIP，值为 `""` 时表明采用 EIP 默认配置创建。`""` 内可填写 EIP 云 API 参数 json，实现自定义配置。	如需绑定 EIP，则此项为必填项
eks.tke.cloud.tencent.com/eip-claim-delete-policy	Pod 删除后，EIP 是否自动回收，Never 不回收，默认回收。	否
eks.tke.cloud.tencent.com/eip-id-list	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
```

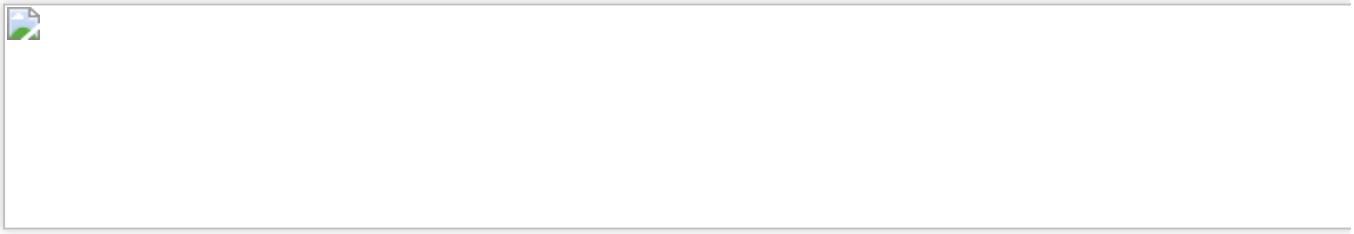
2. 运行后执行以下命令查看事件：



```
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。

3. 查看 log 文件也发现能正常下载数据集。如下图所示：



注意

EIP 的申请每天有限额，不适用于需要多次运行的任务。

在 Serverless 集群上玩转深度学习
构建深度学习容器镜像

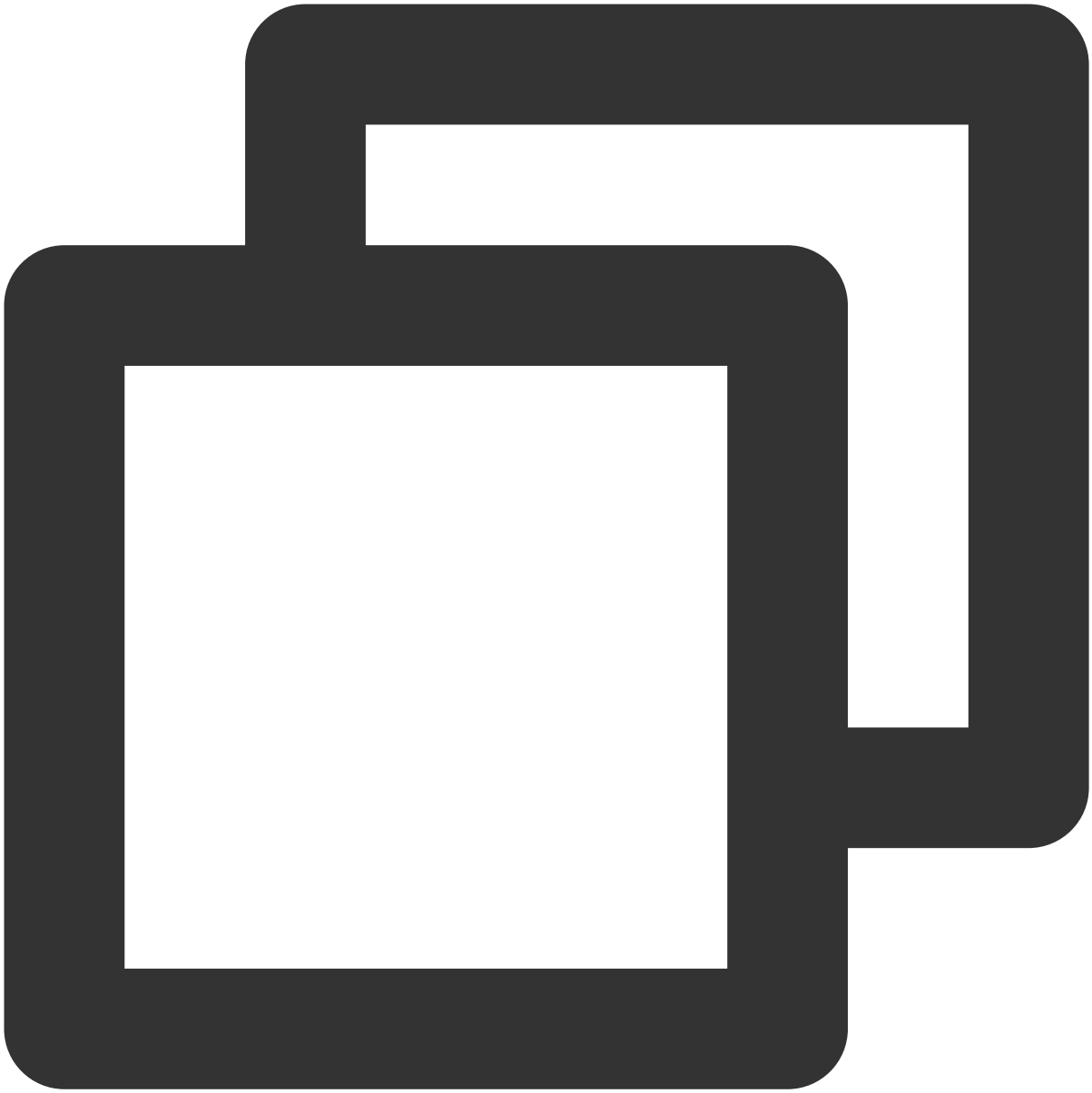
最近更新时间：2023-05-06 17:36:46

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

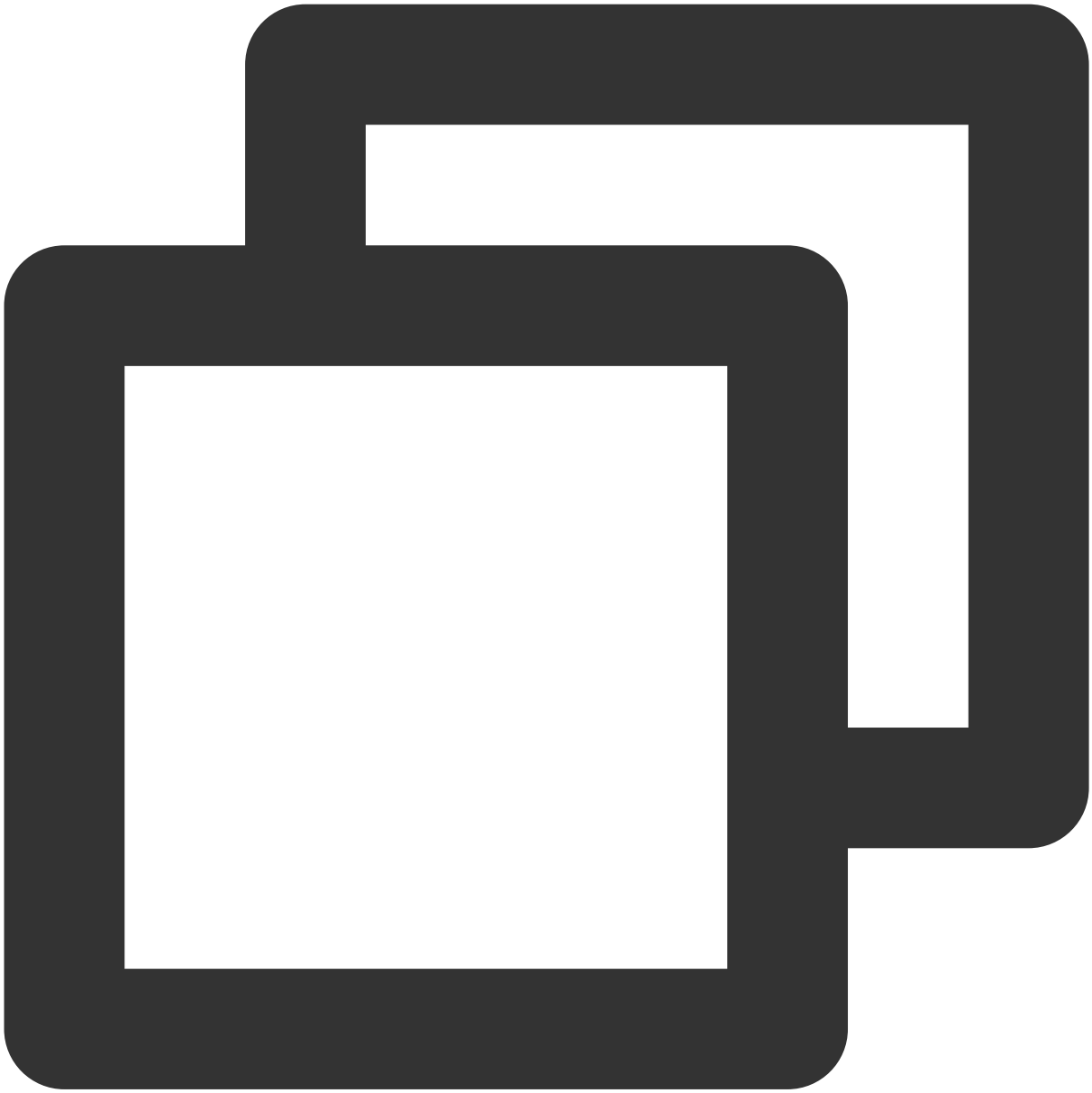
1. 本文示例通过 Docker 容器 构建镜像。准备 Dockerfile 文件，示例如下：



```
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 \\
 && apt-get --purge remove -y git \\    #不需要的组件及时卸载（可选）
 && rm -rf /var/lib/apt/lists/*         #删除apt安装用的安装包（可选）
 && mkdir /tf /tf/models /tf/data       #新建存储模型和数据的路径，可作为挂载点（可选）
ENV PYTHONPATH $PYTHONPATH:/models
ENV LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/local/cuda-11.3/lib64:/usr/lib/x86_64-lin
```

```
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。

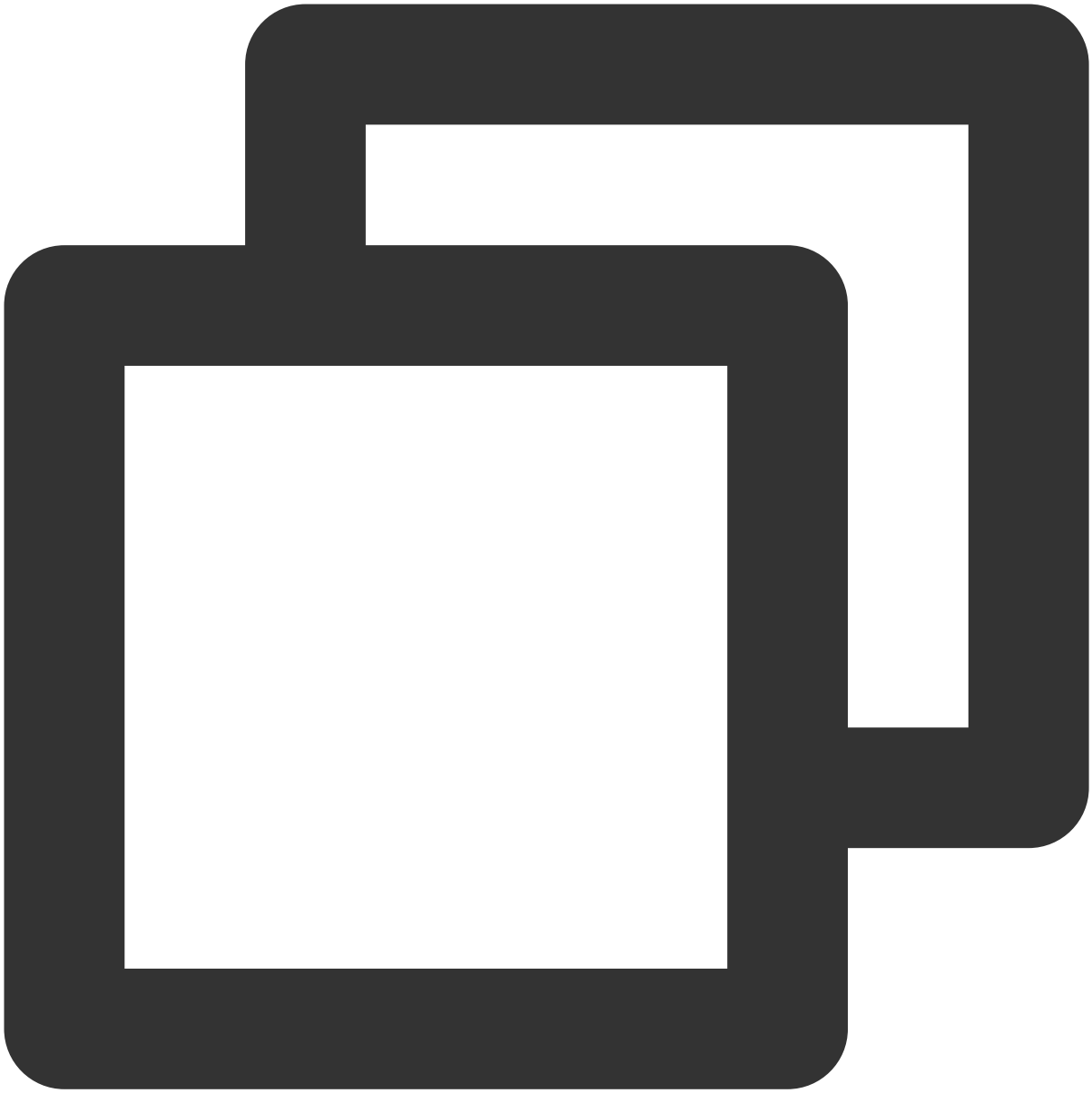
环境变量

在进行本文最佳实践时，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]、ibcusolver.so.[version]、libcudnn.so.[version] 等。您可以执行 `ll` 命令进行查看，如下图所示：

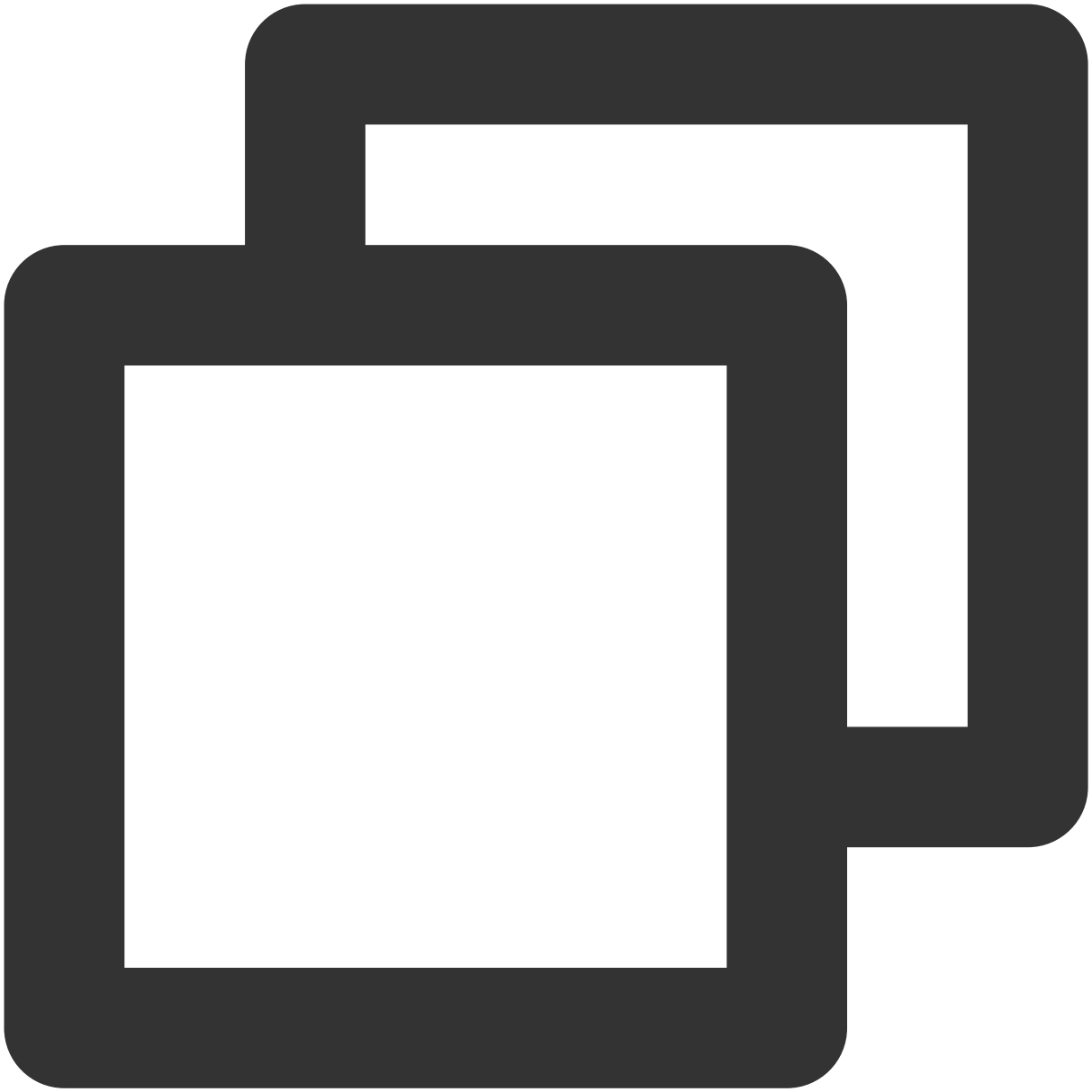
```
root@5a949761c669:/usr/local/cuda-11.3/lib64# ll
total 1534336
drwxr-xr-x 1 root root      4096 Jul  2 03:57 ./
drwxr-xr-x 1 root root      4096 Jul  2 03:57 ../
lrwxrwxrwx 1 root root        16 May  4 02:30 libOpenCL.so.1 -> libOpenCL.so.1.0
lrwxrwxrwx 1 root root        18 May  4 02:30 libOpenCL.so.1.0 -> 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
-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
-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
-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.10.4.2.1
-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
-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：



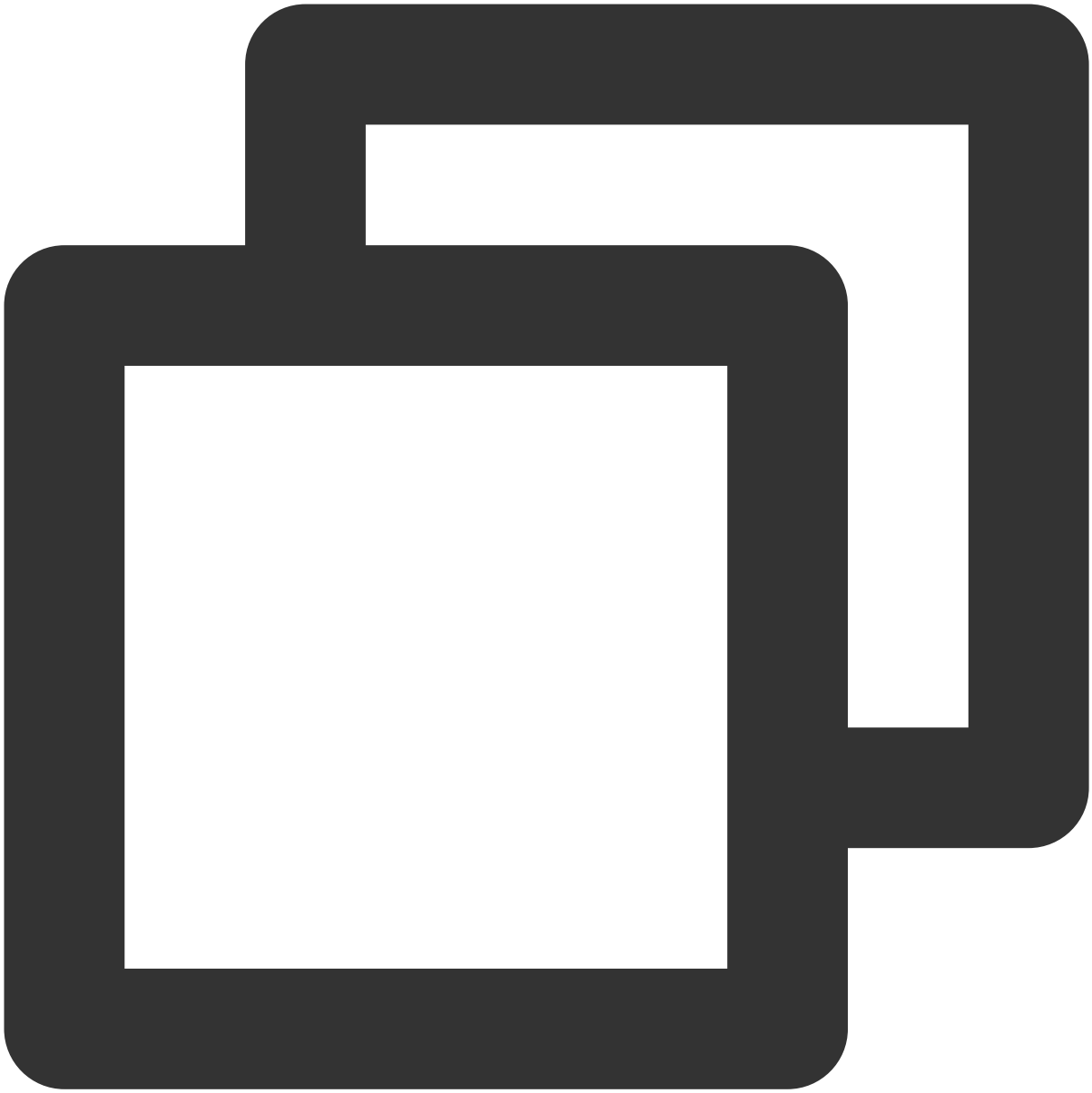
```
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64
```

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
Cannot dlopen some GPU libraries. Please make sure the missing libraries mentioned
```

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：2023-05-22 15:48:11

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

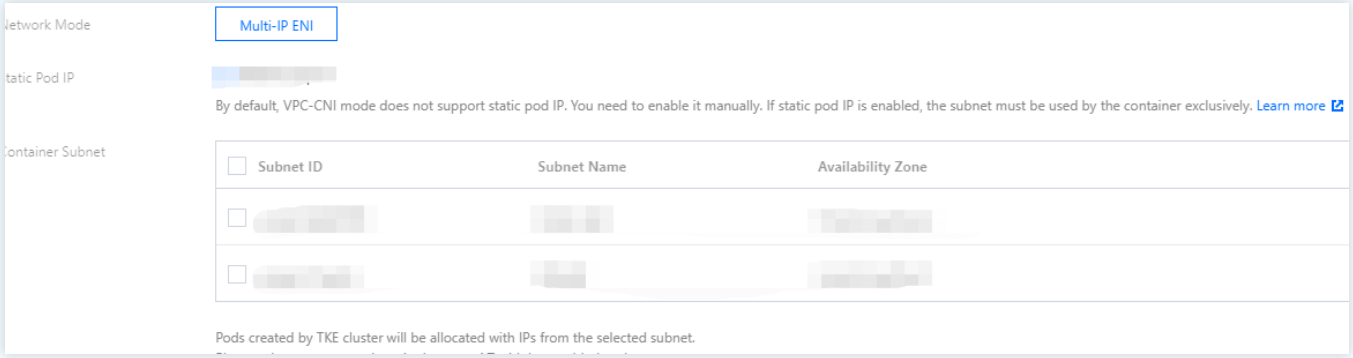
本文将在上一篇文档 构建深度学习容器镜像 基础上继续操作，利用自建集群，在 TKE Serverless 上运行深度学习任务。自建镜像已上传到镜像仓库中：`ccr.ccs.tencentyun.com/carltk/tensorflow-model`，无需重新构建，可以直接拉取使用。

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：



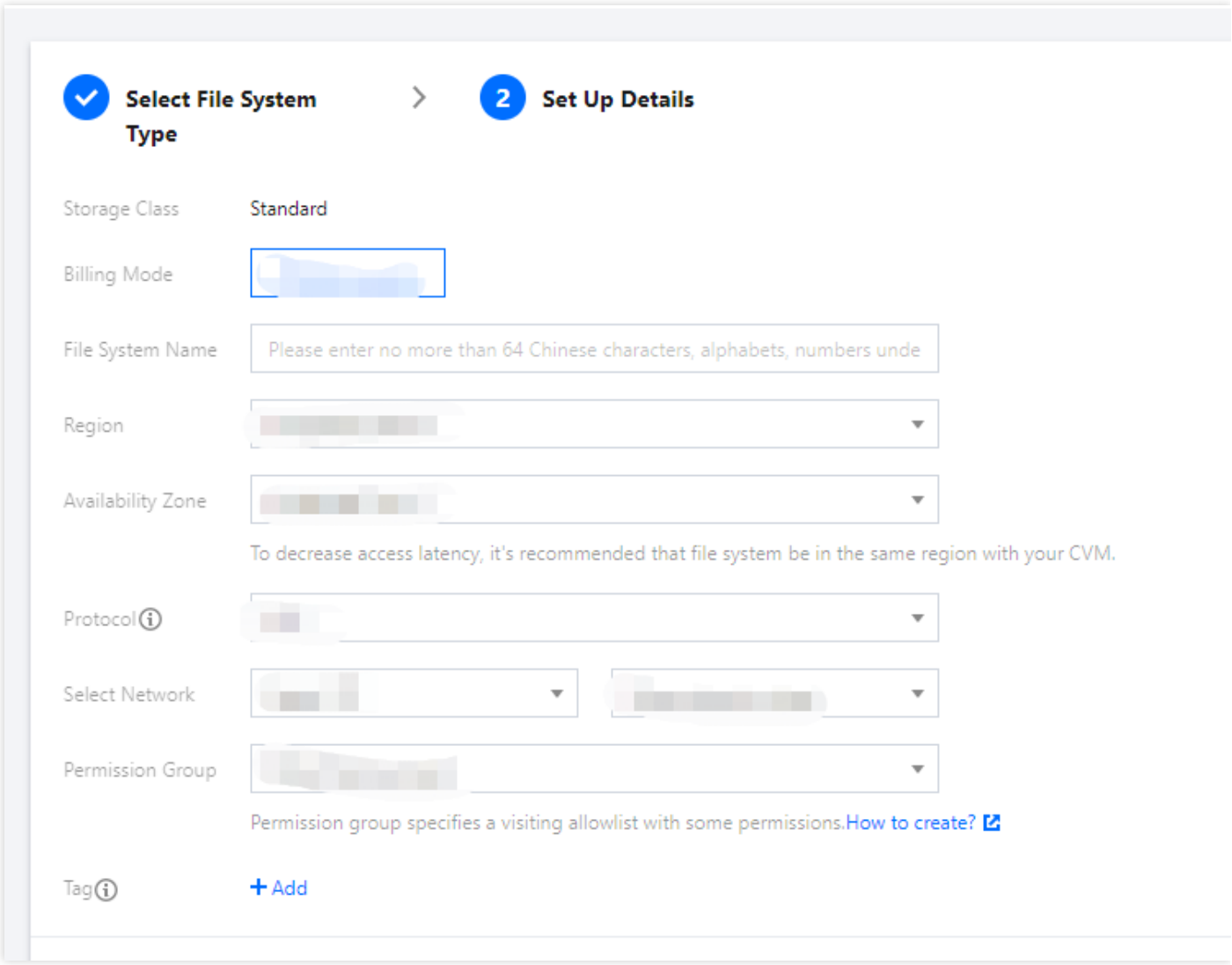
创建 CFS 文件系统（可选）

容器将在任务结束后，自动删除容器并且释放资源。因此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持久存储数据。目前支持 云硬盘 CBS、文件存储 CFS、对象存储 COS 等方式。

本文示例将利用 NFS 盘的方式，使用 CFS，实现于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**，在弹出的“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。如下图所示：

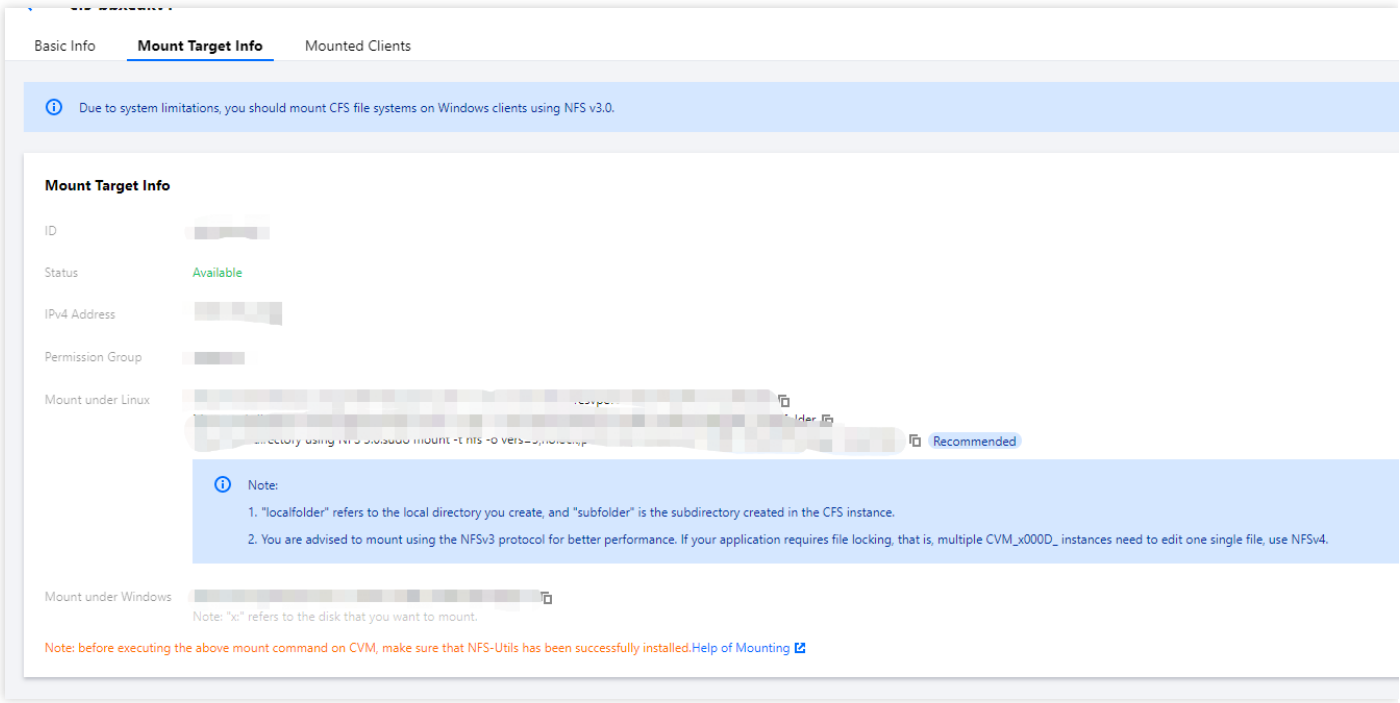


> 注意：
> 创建的 CFS 地域，需确保与集群在同一地域。

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:/` 。

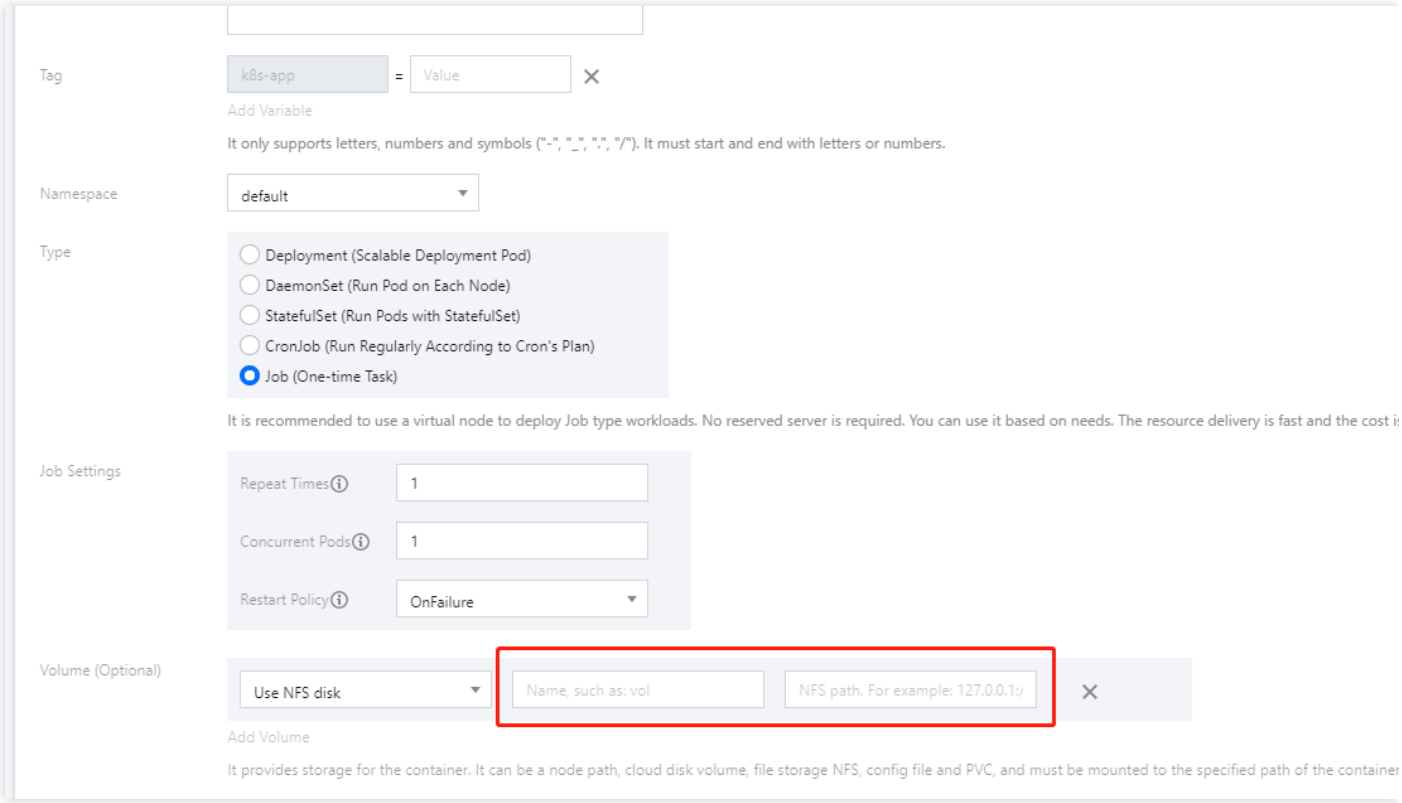
创建训练任务

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
- Kubectl 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。

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）。

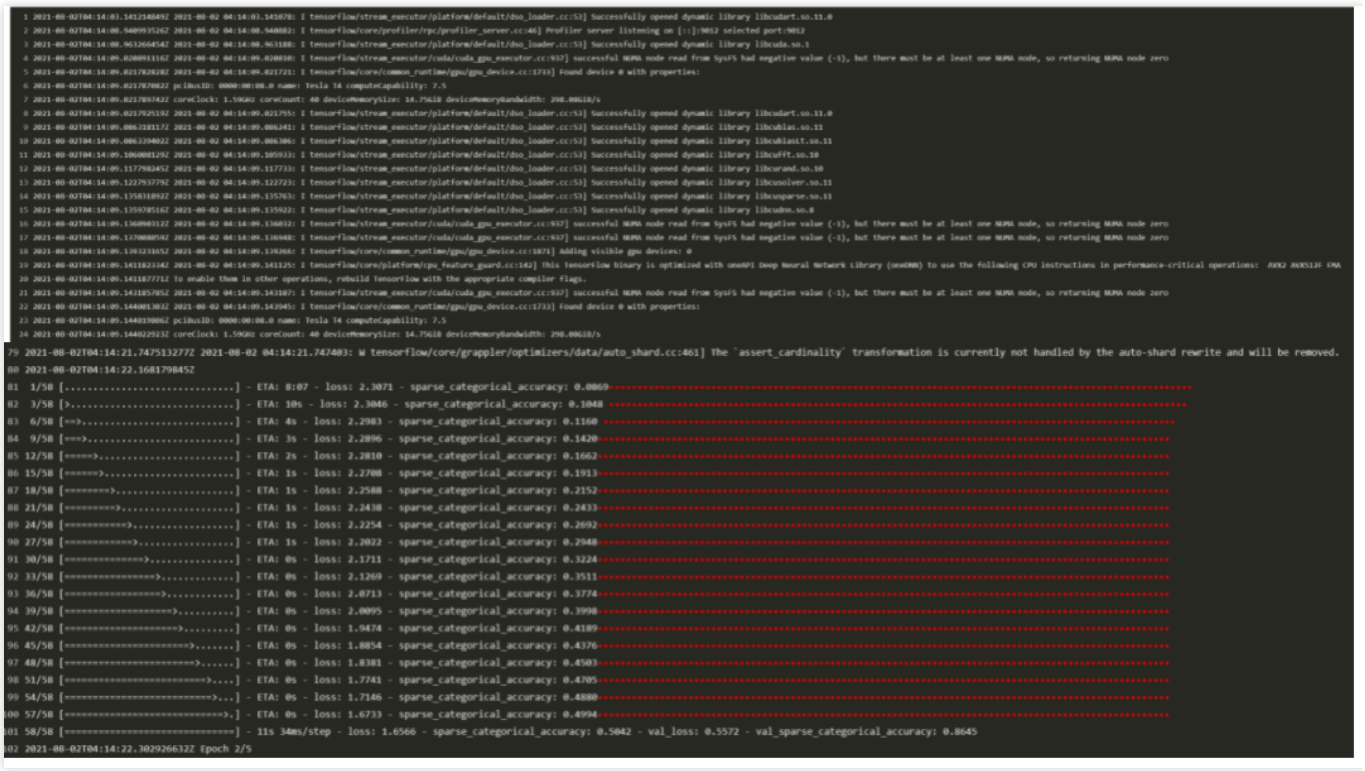
查看运行结果

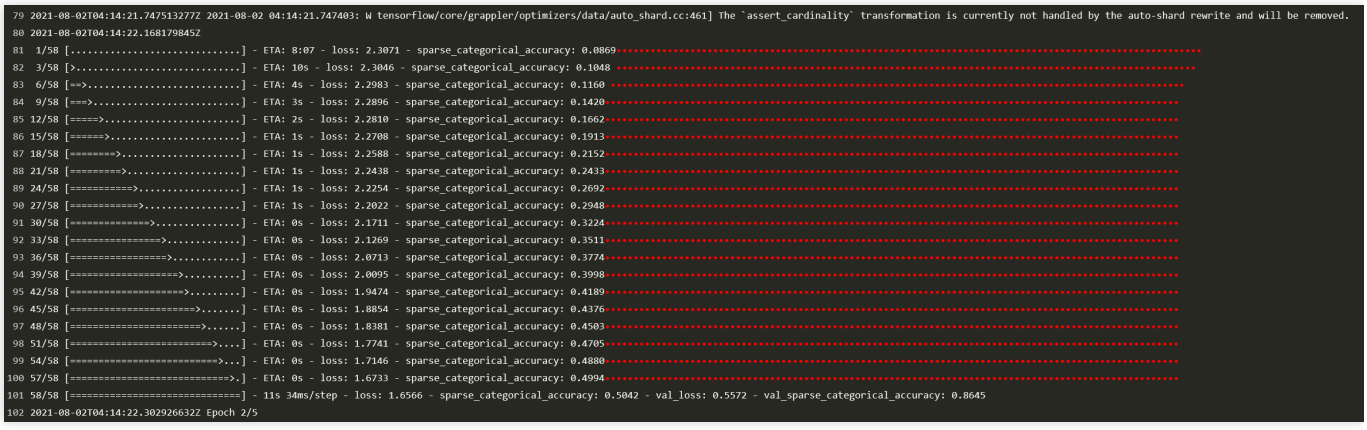
以下提供控制台和命令行两种方式查看运行结果：

- 控制台查看
- 命令行查看

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. 在弹性集群列表中，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**页签在查看事件
   - 选择**日志**页签查看日志，如下图所示：





相关操作

在 TKE 上使用 GPU 部署深度学习任务

在 TKE 上部署和 TKE Serverless 的部署几乎没有区别。以 kubectl 通过 YAML 部署为例，有以下两点改动：

- 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 文档。
- 因为节点自带 GPU 资源，因此无需 Annotations 和 Resources。在实践操作汇总，您可以保留 Annotations，TKE 不会处理这部分。Resources 则建议注释掉，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。

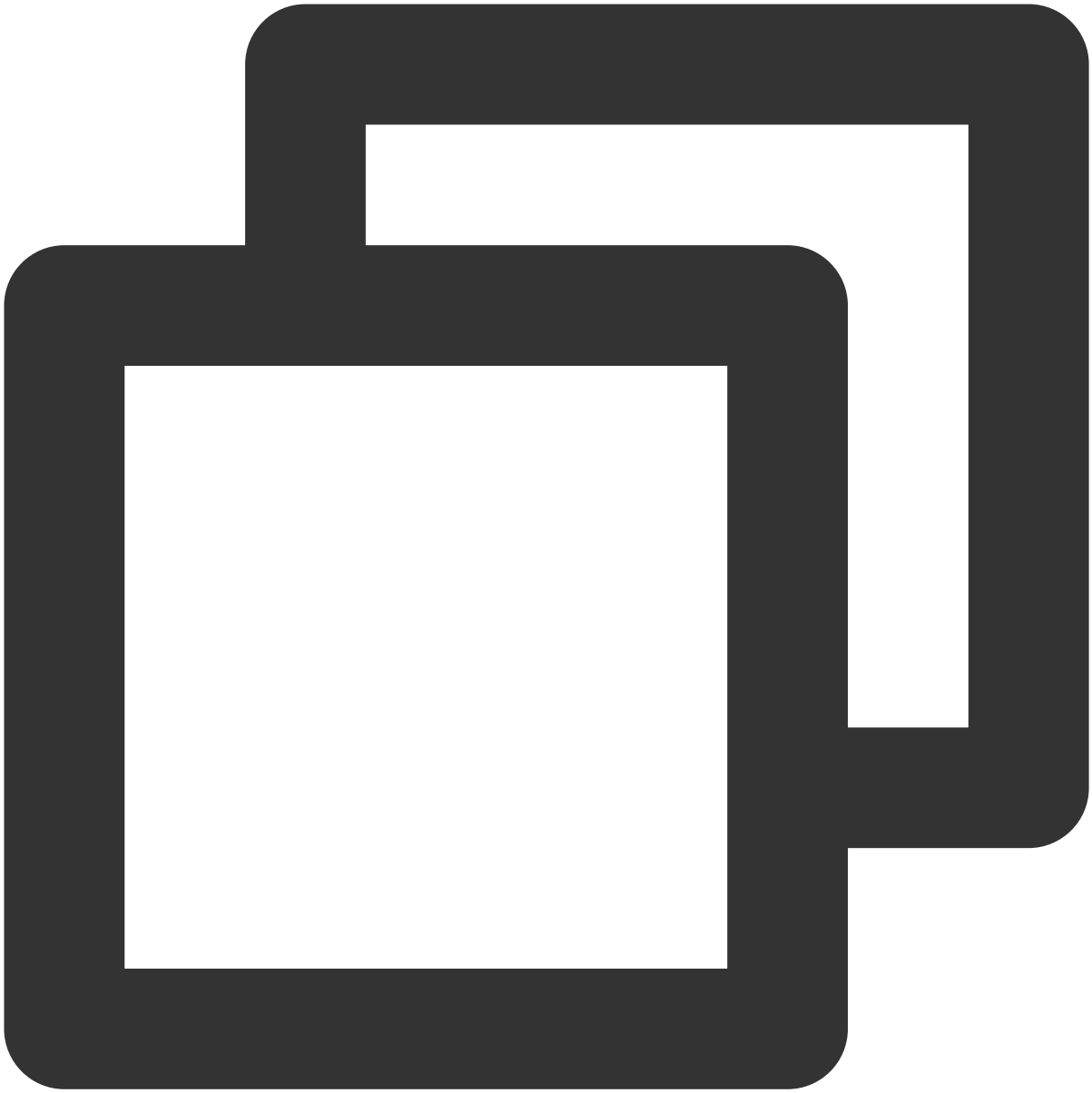
常见问题
公网访问相关

最近更新时间：2023-05-06 17:36:46

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
E tensorflow/core/platform/cloud/curl_http_request.cc:614] The transmission  of req
```

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：2023-05-22 15:15:24

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 connect
ion to pod ***
```

以下为您介绍能将日志持久存储的方法：

- 重定向
- 配置日志采集

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。

配置日志采集

在 TKE Serverless 集群中，可以通过环境变量和自定义资源（CRD）两种方式配置日志采集。

- 使用环境变量配置日志采集
- 通过 CRD 配置日志采集（推荐）

1. 按照 使用环境变量配置日志采集 文档配置日志采集
2. 如果使用密钥授权，可以新建一个 Opaque 类型的 Secret ，创建两个 key（SecretId 和 SecretKey），值分别是在 API 密钥 中获得的 SecretId 和 SecretKey
3. 即可在开启日志采集中找到创建的 Secret ，并关联 SecretId 和 SecretKey 。
4. 在控制台得到原始日志，切换表格展示并将 JSON 格式化。

使用此方法将会出现一个问题，TKE Serverless 开启日志采集功能的原理，是将采集到的日志以 JSON 的形式发送到指定的消费端。而采集提供的 JSON 时间戳是秒级。

将导致在控制台上查看的日志时间的粒度也是秒级，在检索分析端上看到的日志只能按秒级时间顺序，而更小尺度不能实现顺序输出。而有时候日志会在短时间大量输出，通常需要微秒级的精度。因此我们更推荐使用 CRD 配置方式。

Serverless 集群自定义 DNS 服务

最近更新时间：2022-09-26 17:15:46

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。

边缘集群
边缘容器 ServiceGroup 功能
通过 yaml 使用 ServiceGroup 功能

最近更新时间：2022-06-10 16:48:45

操作场景

边缘容器服务 TKE Edge 提供 ServiceGroup 特性，只需两个 yaml 文件即可轻松实现上百地域的服务部署，且无需进行应用适配或改造。本文以在边缘部署 nginx 为例。若您希望在多个节点组内分别部署 nginx 服务，请参考本文依次执行以下步骤。

操作步骤

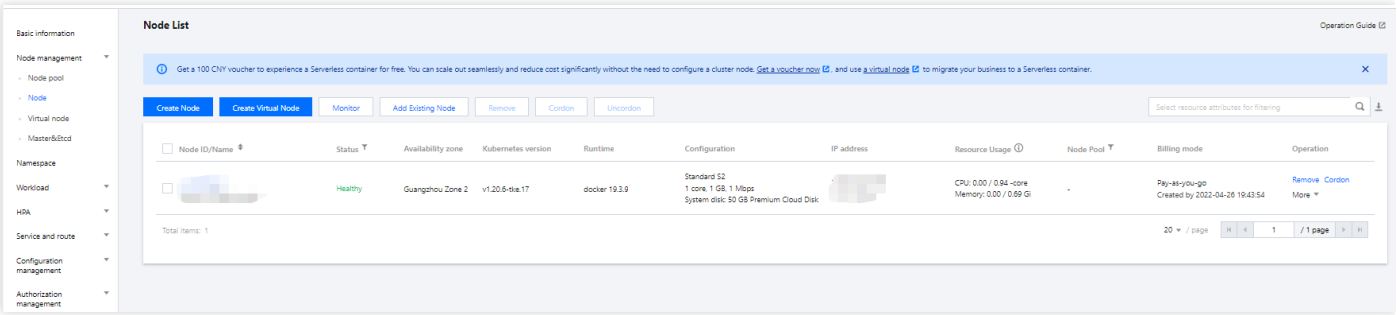
确定 ServiceGroup 唯一标识

该步骤进行逻辑规划，无需任何实际操作。边缘容器将目前要创建的 ServiceGroup 逻辑标记使用的 UniqKey 设置为 zone。

通过 Label 将边缘节点分组

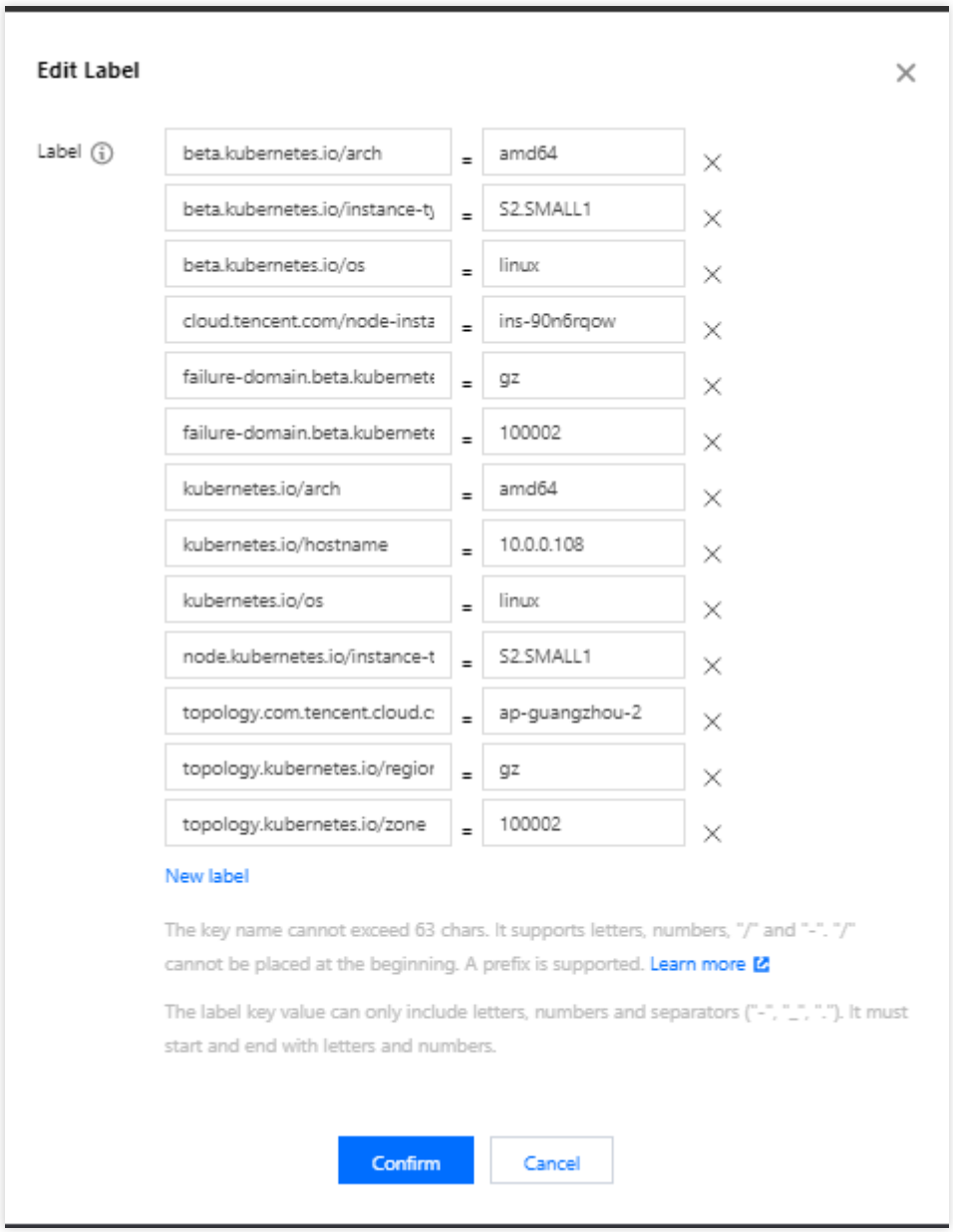
该步骤需要通过 TKE Edge 控制台或者 kubectl 对边缘节点打 Label。TKE Edge 控制台操作步骤如下：

1. 登录 容器服务控制台 ，选择左侧导航栏中的**边缘集群**。
2. 选择需要编辑标签的节点所在的集群 ID，进入该集群管理页面。
3. 选择**节点管理** > **节点**，进入节点列表页，如下图所示：



4. 选择需要编辑标签的节点所在行右侧的**更多** > **编辑标签**。

5. 在弹出的“编辑标签”窗口，参考以下信息新增 Label。如下图所示：



- 参考整体架构章节，选择 Node12及 Node14编辑 Label： `zone=nodeunit1` 。Node21及 Node23编辑 Label： `zone=nodeunit2` 。
- Label 的 key 需与 ServiceGroup 的 UniqKey 一致，value 是 NodeUnit 的唯一key。value 相同的节点表示属于同一个 NodeUnit。
- 如果同一个集群中有多个 ServiceGroup，请为每一个 ServiceGroup 分配不同的 Uniqkey。

6. 单击**确定**即可。

部署 DeploymentGrid

```
apiVersion: superedge.io/v1
kind: DeploymentGrid
```

```
metadata:
name: deploymentgrid-demo
namespace: default
spec:
gridUniqKey: zone
template:
selector:
matchLabels:
appGrid: nginx
replicas: 2
template:
metadata:
labels:
appGrid: nginx
spec:
containers:
- name: nginx
image: nginx:1.7.9
ports:
- containerPort: 80
protocol: TCP
```

部署 ServiceGrid

```
apiVersion: superedge.io/v1
kind: ServiceGrid
metadata:
name: servicegrid-demo
namespace: default
spec:
gridUniqKey: zone
template:
selector:
appGrid: nginx
ports:
- protocol: TCP
port: 80
targetPort: 80
```

说明：

可查看此处 gridUniqKey 字段设置为 zone，因此对应 通过 Label 将边缘节点分组 步骤中对节点进行分组时 Label 的 key 也应设置为 zone。如果有三组节点，则分别添加 `zone: zone-0` ， `zone: zone-1` ， `zone: zone-2` 的 Label 即可。

这时，每组节点内都有了 nginx 的 Deployment 和对应的 Pod，在节点内访问统一的 service-name 也只会将请求发向本组的节点。验证方式如下：

```
[root@VM_1_34_centos ~]# kubectl get deploy
NAME READY UP-TO-DATE AVAILABLE AGE
deploymentgrid-demo-zone-0 2/2 2 2 85s
deploymentgrid-demo-zone-1 2/2 2 2 85s
deploymentgrid-demo-zone-2 2/2 2 2 85s
[root@VM_1_34_centos ~]# kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 172.19.0.1 <none> 443/TCP 87m
servicegrid-demo-svc ClusterIP 172.19.0.177 <none> 80/TCP 80s
```

另外，对于部署了 DeploymentGrid 和 ServiceGrid 后才添加进集群的节点组，该功能会在新的节点组内自动创建指定的 Deployment 和 Service。

边缘容器分布式节点状态判定机制

最近更新时间：2022-06-10 19:32:52

边缘弱网络会触发 Kubernetes 驱逐机制，引起不符合预期的 Pod 驱逐动作。边缘计算情景下，边缘节点与云端的网络环境十分复杂，网络质量无法保证，容易出现 API Server 和节点连接中断等问题。如果不加改造直接使用原生 Kubernetes，节点状态会经常出现异常，引起 Kubernetes 驱逐机制生效，导致 Pod 的驱逐和 EndPoint 的缺失，最终造成服务的中断和波动。

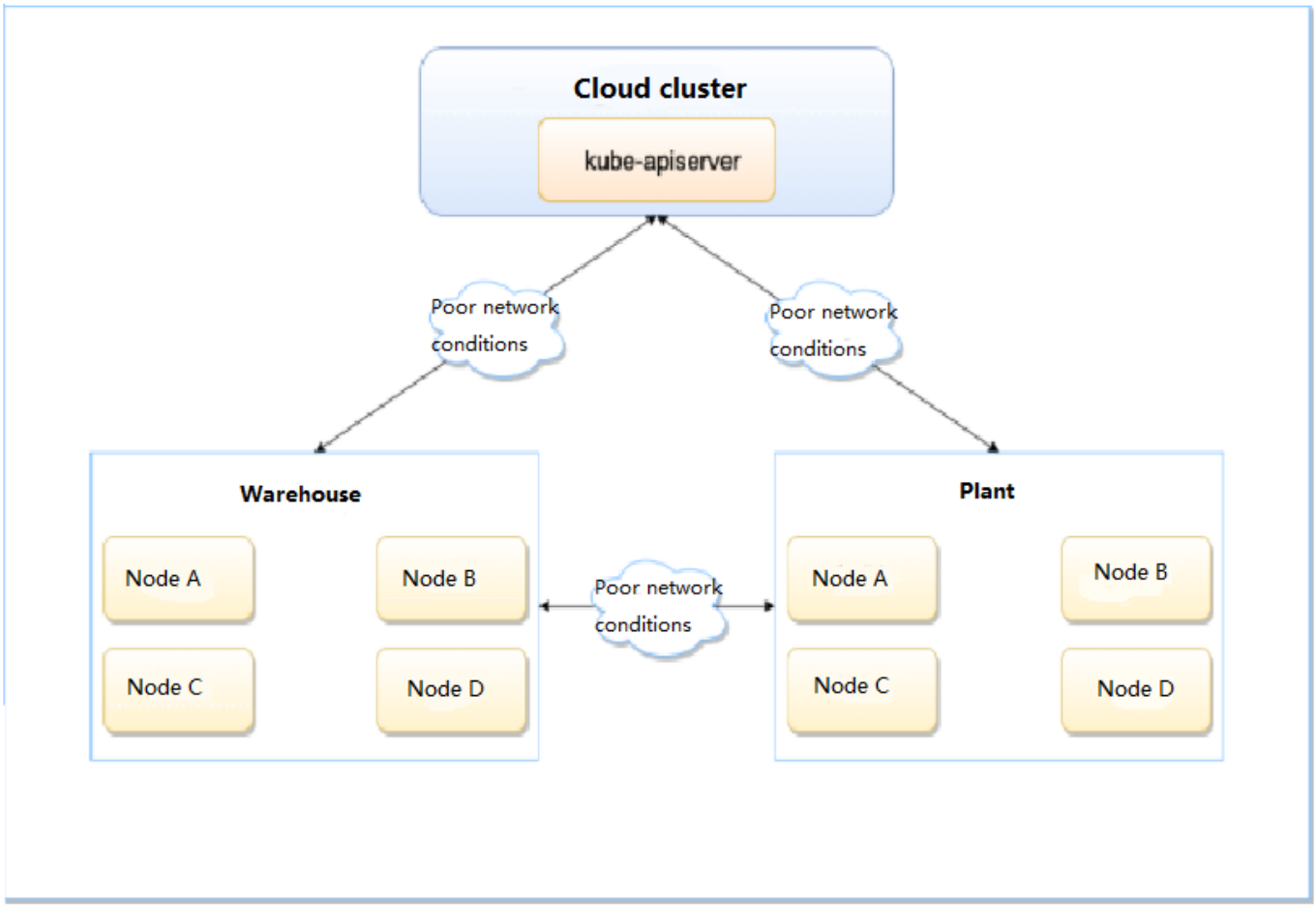
为解决这个问题，边缘容器服务 首创分布式节点状态判定机制。该机制可以更好地识别驱逐时机，保障系统在弱网络下正常运转，避免服务中断和波动。

需求场景

在边缘场景下，需面对云边弱网络的环境。边缘设备位于边缘云机房和移动边缘站点，与云端连接的网络环境较为复杂，既包含云端（控制端）和边缘端的网络环境不可靠，也包含边缘节点之间的网络环境不可靠。

智慧工厂

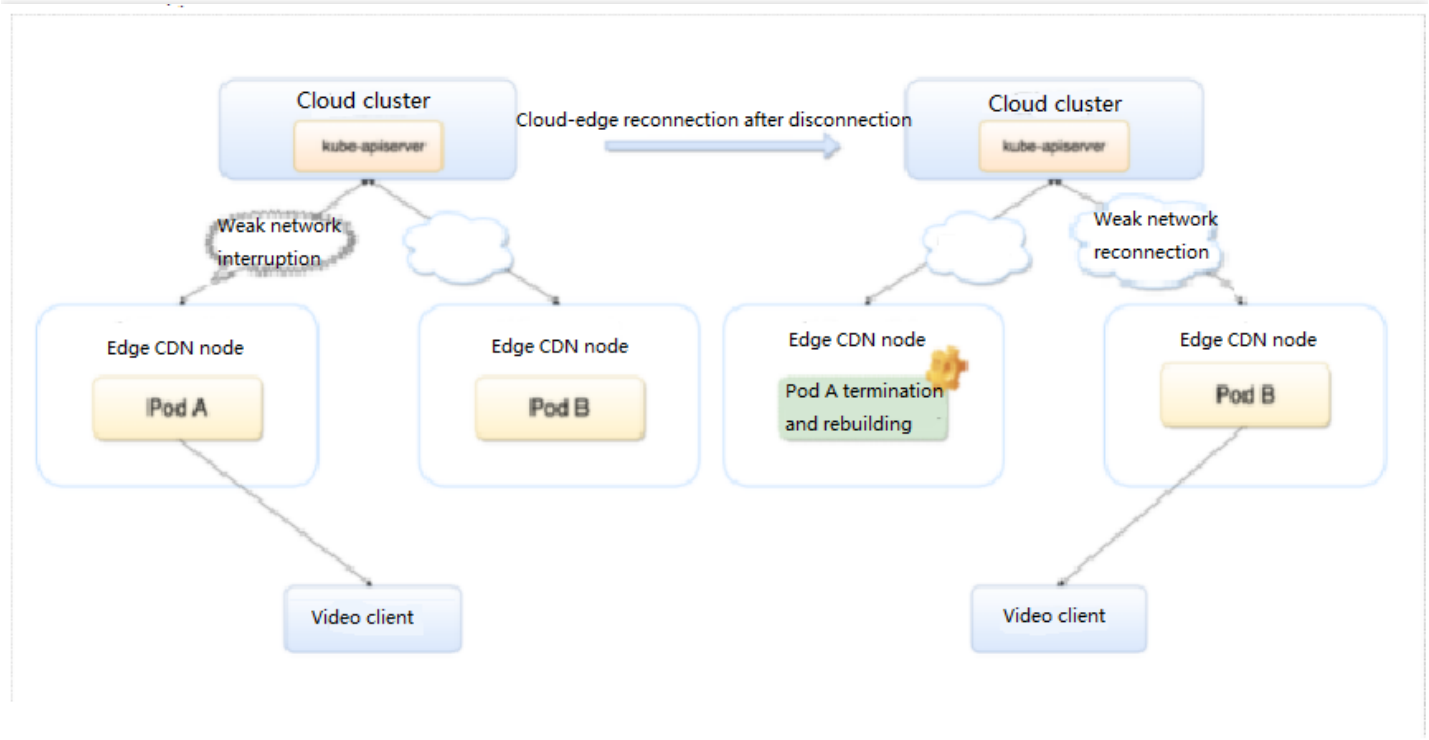
以智慧工厂为例，边缘节点位于厂房仓库和车间，控制端 Master 节点在腾讯云的中心机房内。示意图如下：



- 仓库和车间内的边缘设备同云端集群之间的网络较复杂，因特网、5G、WIFI 等形态均有可能，网络质量差次不齐没有保障。
- 相比于云端的网络环境，仓库和车间内的边缘设备之间是本地网络，网络质量优于同云端集群之间的连接，相对而言更加可靠。

音视频拉流场景

音视频拉流场景如下图所示：



考虑到用户体验及公司成本，音视频拉流经常需要进行提高边缘缓存命中率减少回源、将用户请求的同一文件调度到同一个服务实例以及服务实例缓存文件。

在原生 Kubernetes 的情况下，如果 Pod 因为网络波动而频繁重建，一方面会影响服务实例缓存效果，另一方面会引起调度系统将用户请求调度到其他服务实例。这两点都会对 CDN 效果造成很大甚至不能接受的影响。

事实上，边缘节点完全运行正常，Pod 驱逐或重建其实是完全不必要的。为了克服这个问题，保持服务的持续可用，TKE 边缘容器团队提出了分布式节点状态判定机制。

需求痛点

原生 Kubernetes 处理方式

云边弱网络是影响了运行在边缘节点上的 kubelet 与云端 APIServer 之间通信，云端 APIServer 无法收到 kubelet 的心跳或者进行续租，无法准确获取该节点和节点上 Pod 的运行情况，如果持续时间超过设置的阈值， APIServer 会认为该节点不可用，并做出如下作：

- 失联的节点状态被置为 NotReady 或者 Unknown 状态，并被添加 NoSchedule 和 NoExecute 的 taints。
- 失联的节点上的 Pod 被驱逐，并在其他节点上进行重建。

- 失联的节点上的 Pod 从 Service 的 Endpoint 列表中移除。

解决方案

设计原则

在边缘计算场景中，仅依赖边缘端和 APIServer 的连接情况来判断节点是否正常并不合理，为了让系统更健壮，需要引入额外的判断机制。

相较于云端和边缘端，边缘端节点之间的网络更稳定，可利用更稳定的基础设施提高准确性。边缘容器服务首创了边缘健康分布式节点状态判定机制，除了考虑节点与 APIServer 的连接情况，还引入了边缘节点作为评估因子，以便对节点进行更全面的状态判断。经过测试及大量的实践证明，该机制在云边弱网络情况下提高了系统在节点状态判断上的准确性，为服务稳定运行保驾护航。该机制的主要原理如下：

- 每个节点定期探测其他节点健康状态
- 集群内所有节点定期投票决定各节点的状态
- 云端和边缘端节点共同决定节点状态

首先，节点内部之间进行探测和投票，共同决定具体某个节点是否存在状态异常，保证大多数节点的一致判断才能决定节点的具体状态。其次，即使节点之间的网络状态通常情况下优于云边网络，但也应该考虑边缘节点复杂的网络情况，其网络并非100%可靠。因此，也不能完全信赖节点之间的网络，节点的状态不能只由节点自行决定，云边共同决定才更为可靠。基于这个考虑，做出如下设计：

节点最终状态	云端判定正常	云端判定异常
节点内部判定正常	正常	不再调度新的 Pod 到该节点
节点内部判定异常	正常	驱逐存量 Pod；从 EndPoint 列表摘除； 不再调度新的 Pod 到该节点

方案特性

当云端判定节点异常，但是其他节点认为节点正常的时候，虽然不会驱逐已有 Pod，但为了确保增量服务的稳定性，将不会再将新的 Pod 调度到该节点上，存量节点的正常运行也得益于边缘集群的边缘自治能力。

由于边缘网络和拓扑的特殊性，经常会存在节点组之间网络单点故障的问题，例如 智慧厂房，仓库和车间虽然都属于厂房这个地域内，但他们间的网络连接仅依靠一条关键链路，一旦这条链路发生中断，就会造成节点组之间的分裂，本文提供的方案能够确保两个分裂的节点组失联后互相判定时始终保持多数的一方节点不会被判定为异常，避免被判定为异常造成 Pod 只能被调度到少部分的节点上，造成节点负载过高的情况。

边缘设备可能位于不同的地区且相互不通，本文提供的方案支持多地域内的节点状态判定，可以方便地将节点依据地域或者其他方式进行分组，实现组内的检查。即使重新分组也无需重新部署检测组件或重新初始化，适应边缘计算的网络情况。分组后，节点只会判定同一个组内的节点状态。

前提条件

该功能需要打开节点的51005端口，以便节点之间进行分布式智能健康探测。

操作步骤

> 注意：
> 边缘检查和多地域检查功能需要一定的部署和配置时间，并非即时生效。

开启边缘检查功能

边缘检查功能默认关闭，请参考以下步骤手动开启：

1. 登录 容器服务控制台 。
2. 在集群列表页面，选择目标边缘集群 ID，进入集群详情页面。
3. 选择左侧菜单栏中的**基本信息**，进入“基础信息”页面，
4. 在“基础信息”页面中，单击**开启Edge Health**即可开启边缘检查功能。

开启多地域检查功能

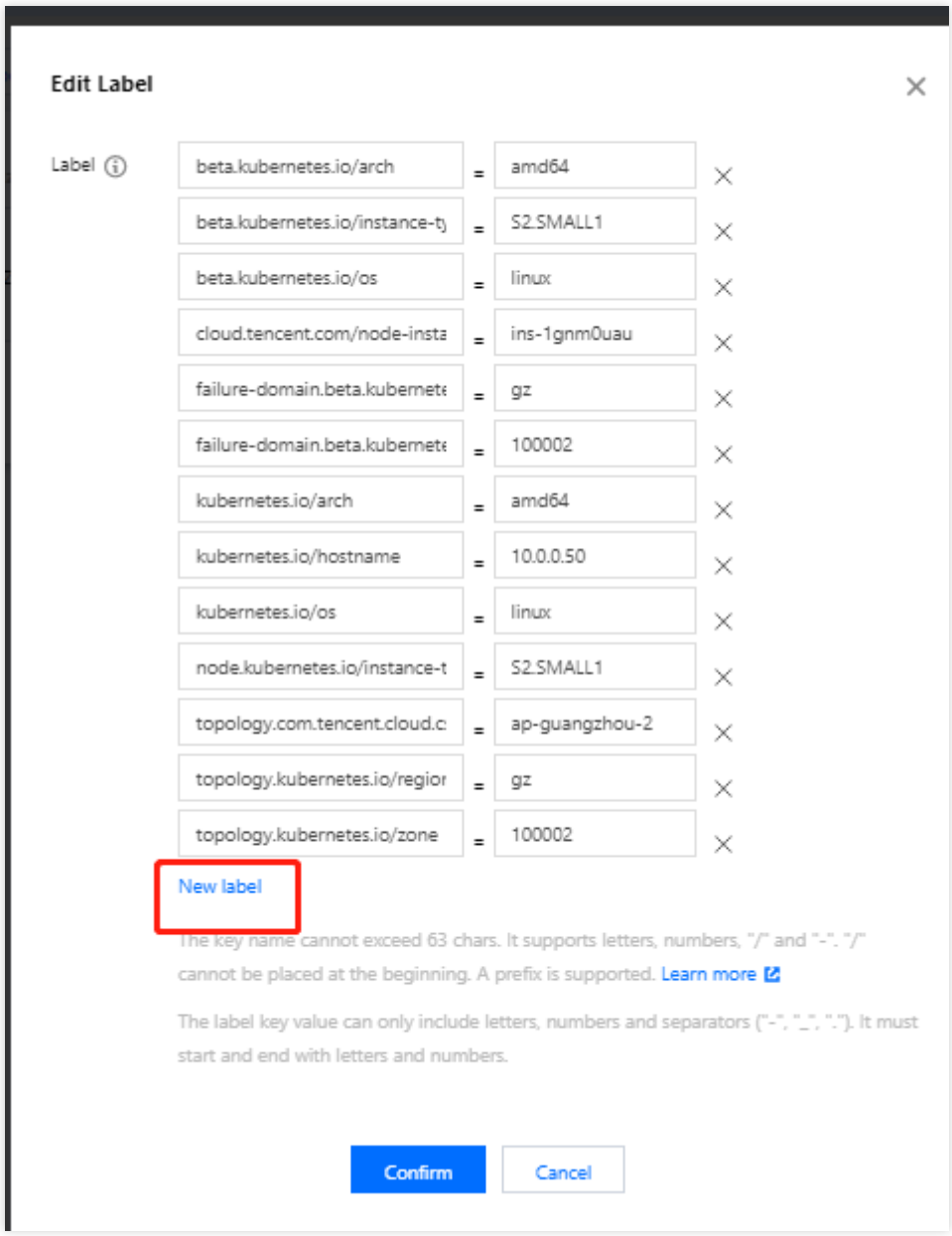
多地域检查功能按地域划分节点：节点地域根据节点上的 `tencent.tkeedgehealth/topology-zone` 标签区分。例如，`tencent.tkeedgehealth/topology-zone: zone0` 表明将节点划分到地域 zone0。标签取值相同的节点视为同一个地域。开启多地域功能时，同一个地域内的节点会相互探测和投票。

> 注意：
> - 如果开启此功能时没有给节点打上 `tencent.tkeedgehealth/topology-zone` 的标签，该节点只会检查自己的健康状态。
> - 如果没有开启此功能，则一个集群内的所有节点会进行相互检查，即使节点上有 `tencent.tkeedgehealth/topology-zone` 标签。

控制台设置节点地域标签

1. 登录 容器服务控制台 。
2. 在集群列表页面，选择目标边缘集群 ID，进入集群详情页面。
3. 选择左侧菜单栏中的**节点管理** > **节点**，进入“节点列表” 页面。
4. 选择需要设置 Label 的节点行，选择**更多** > **编辑标签**。

5. 在弹出的 “编辑 Label” 窗口中，编辑 Label，单击**提交**。如下图所示：



开启多地域检查功能

开启边缘检查功能 后，单击**开启多地域**即可开启多地域检查功能。

安全

Pod 安全组

最近更新时间：2022-11-02 11:52:51

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，以允许进出您部署在多种 TKE 节点类型（目前只支持超级节点，后续会支持普通节点等）上运行的 Pod 的入站和出站网络流量。

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。

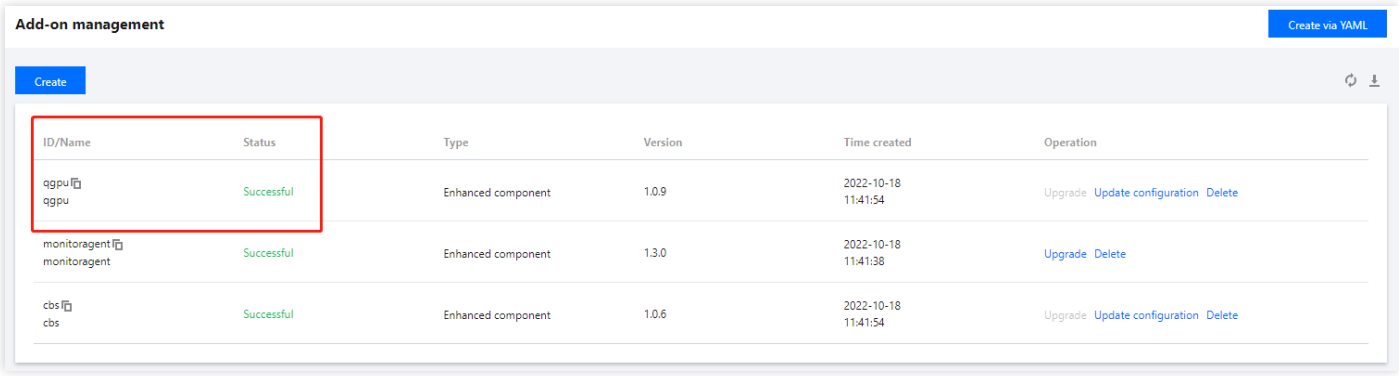
为 Pod 启用安全组能力

安装扩展组件

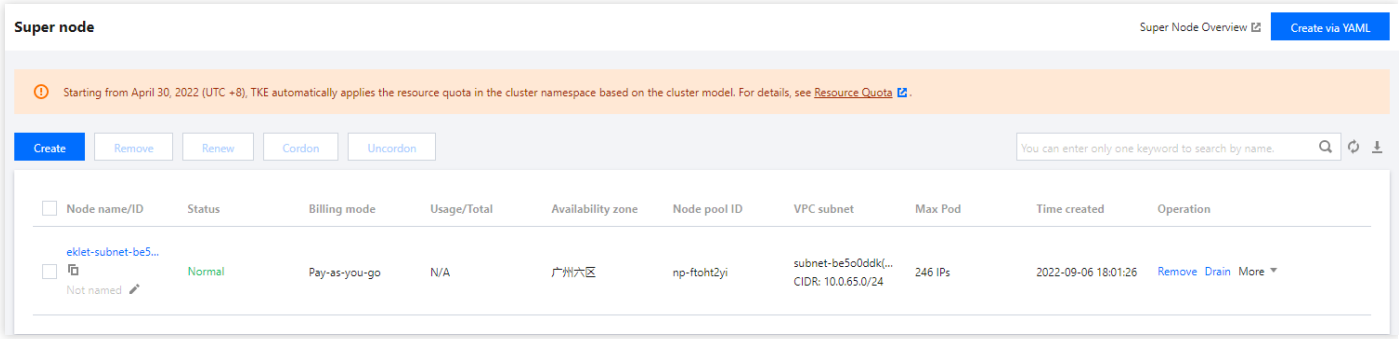
1. 登录 容器服务控制台。
2. 为集群安装 SecurityGroupPolicy（安全组策略）组件。

- 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。
   a. 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-ar
```

```
gument --Filters.0.Name security-group-name --Filters.0.Values tke-worker-secur
ity-for-$my_cluster_name --filter SecurityGroupSet[0].SecurityGroupId | sed 's/
\"//g')
```

b. 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" --GroupDescri
ption "My pod security group"
my_pod_security_group_id=$(tccli vpc DescribeSecurityGroups --cli-unfold-argume
nt --Filters.0.Name security-group-name --Filters.0.Values my-pod-security-grou
p --filter SecurityGroupSet[0].SecurityGroupId | sed 's/\"//g')
echo $my_pod_security_group_id
```

c. 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.Se
curityGroupId $my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Acti
on 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.Se
curityGroupId $my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Acti
on ACCEPT
```

d. 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 --Sec
urityGroupPolicySet.Ingress.0.Port ALL --SecurityGroupPolicySet.Ingress.0.Secur
ityGroupId $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 --Secu
rityGroupPolicySet.Egress.0.Port ALL --SecurityGroupPolicySet.Egress.0.Action A
CCEPT
```

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。

a. 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帐户标签选择 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。

b. 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。

a. 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
```

b. 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。截止现在为止此终端称为 `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-q21ras
u6-8bpgyx9r <none> <none>
my-deployment-866ffd8886-b7gzb 1/1 Running 0 85s 10.0.64.3 eklet-subnet-q21rasu
6-8bpgyx9r <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：

i. 允许与安全组关联的所有 Pod 之间的所有流量。
ii. 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.Se
curityGroupId $my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Acti
on 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.Se
curityGroupId $my_pod_security_group_id --SecurityGroupPolicySet.Ingress.0.Acti
on 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。

0. 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_s
ecurity_group_id
```

容器镜像签名及验证

最近更新时间：2023-05-18 17:05:15

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。

镜像签名验证

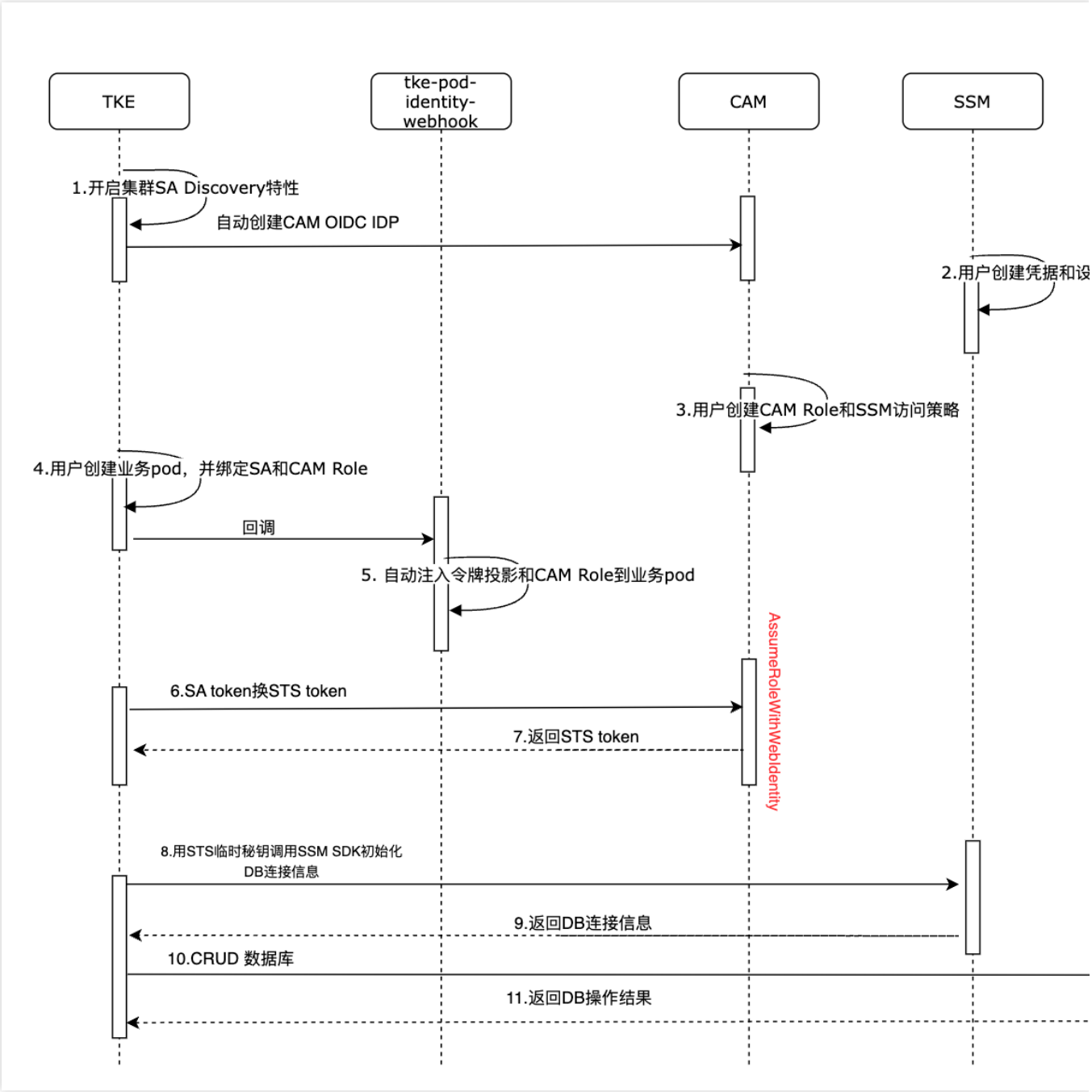
腾讯云容器服务（Tencent Kubernetes Engine，TKE）提供镜像签名验证组件 Cerberus，支持对签名镜像进行可信验证，确保在 TKE 集群中只部署可信授权方签名的容器镜像，降低在容器环境中的镜像安全风险。

Pod 使用 CAM 对数据库身份验证

最近更新时间：2024-02-05 17:23:15

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

操作步骤

步骤1：准备托管集群

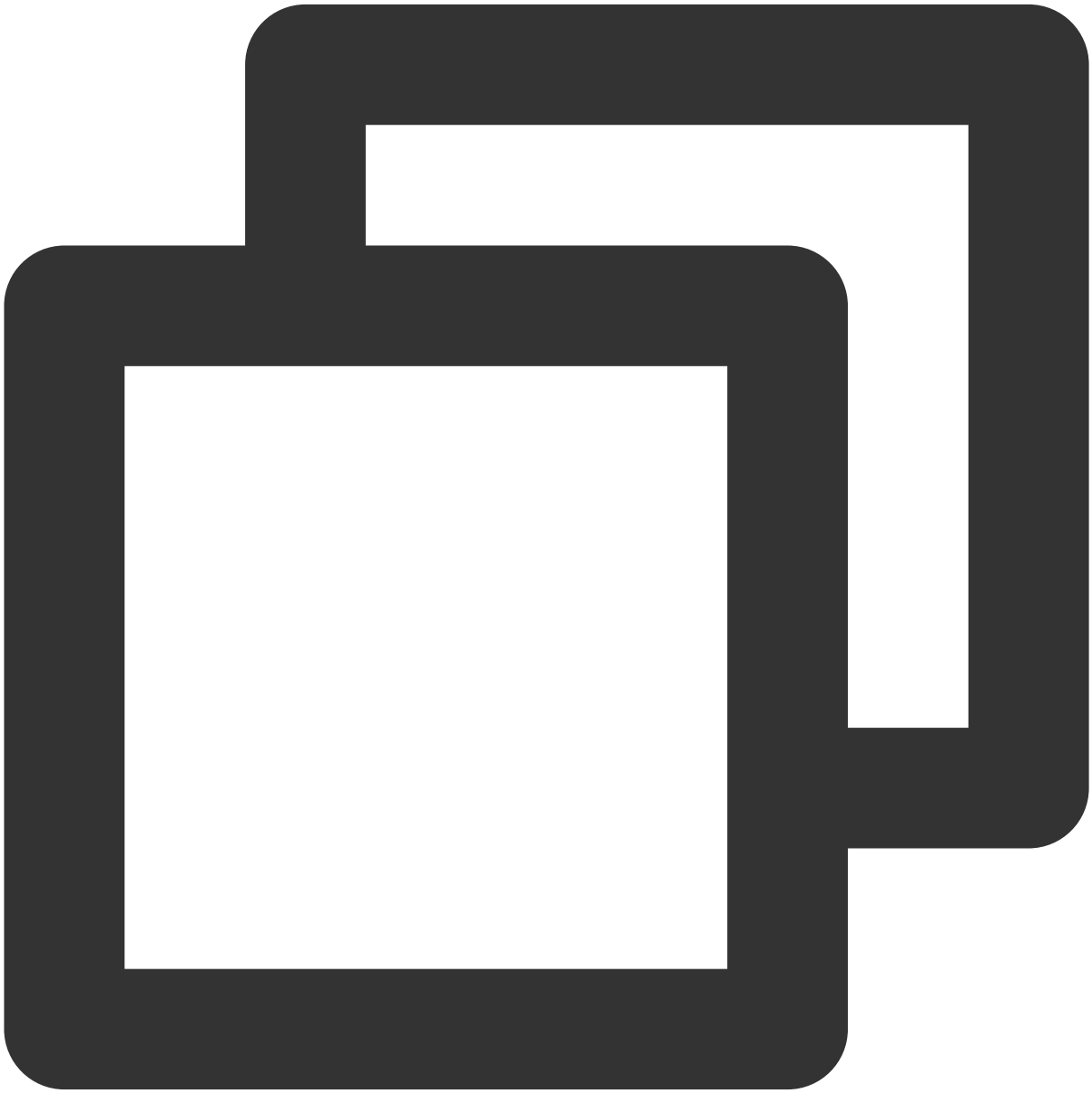
1. 登录 容器服务控制台，新建集群。

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。

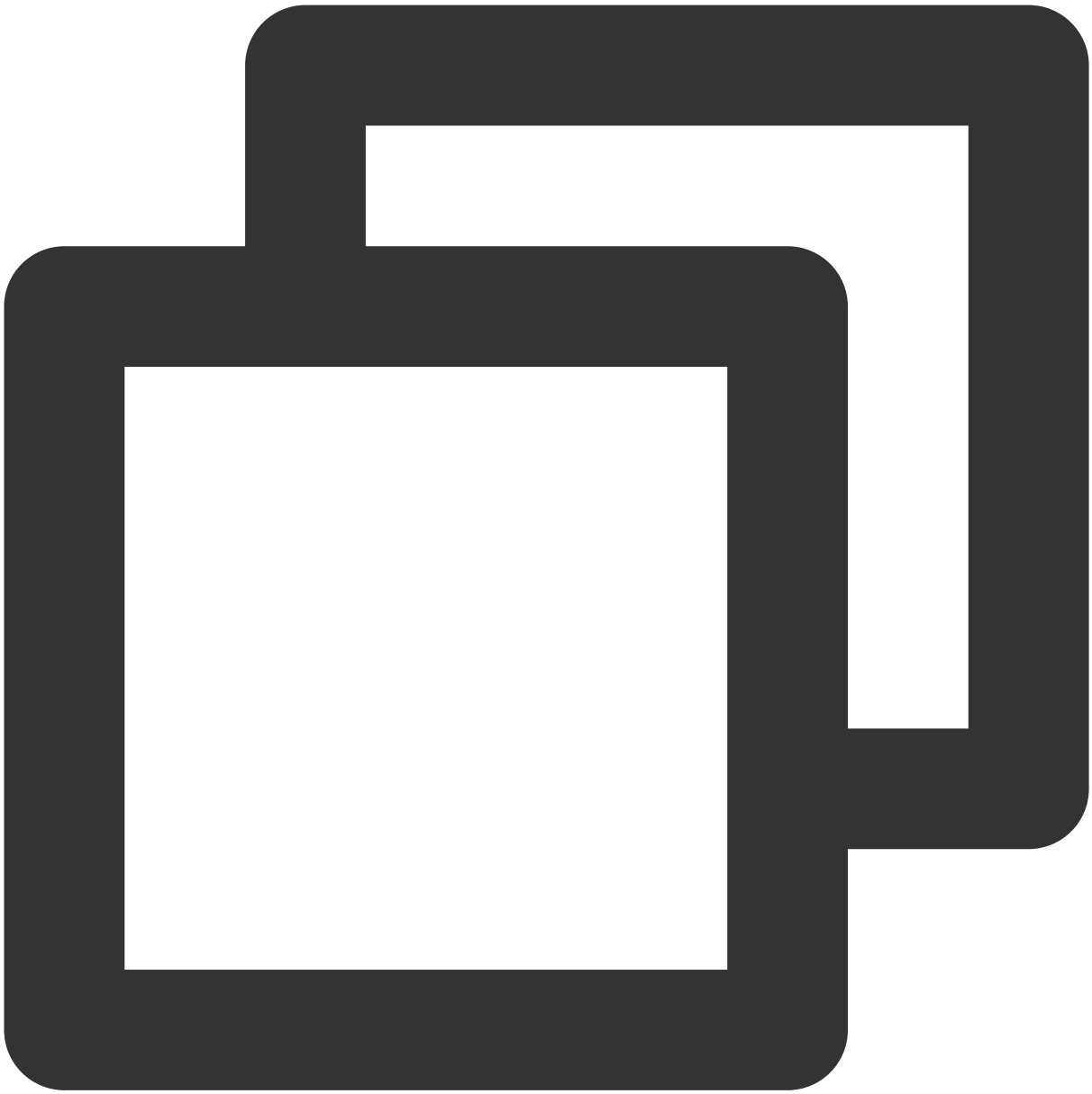
如果您已有托管集群，请在集群详情页检查集群版本，当集群版本不满足要求时，请升级集群。对运行中的 Kubernetes 集群进行升级，详情见 升级集群。

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

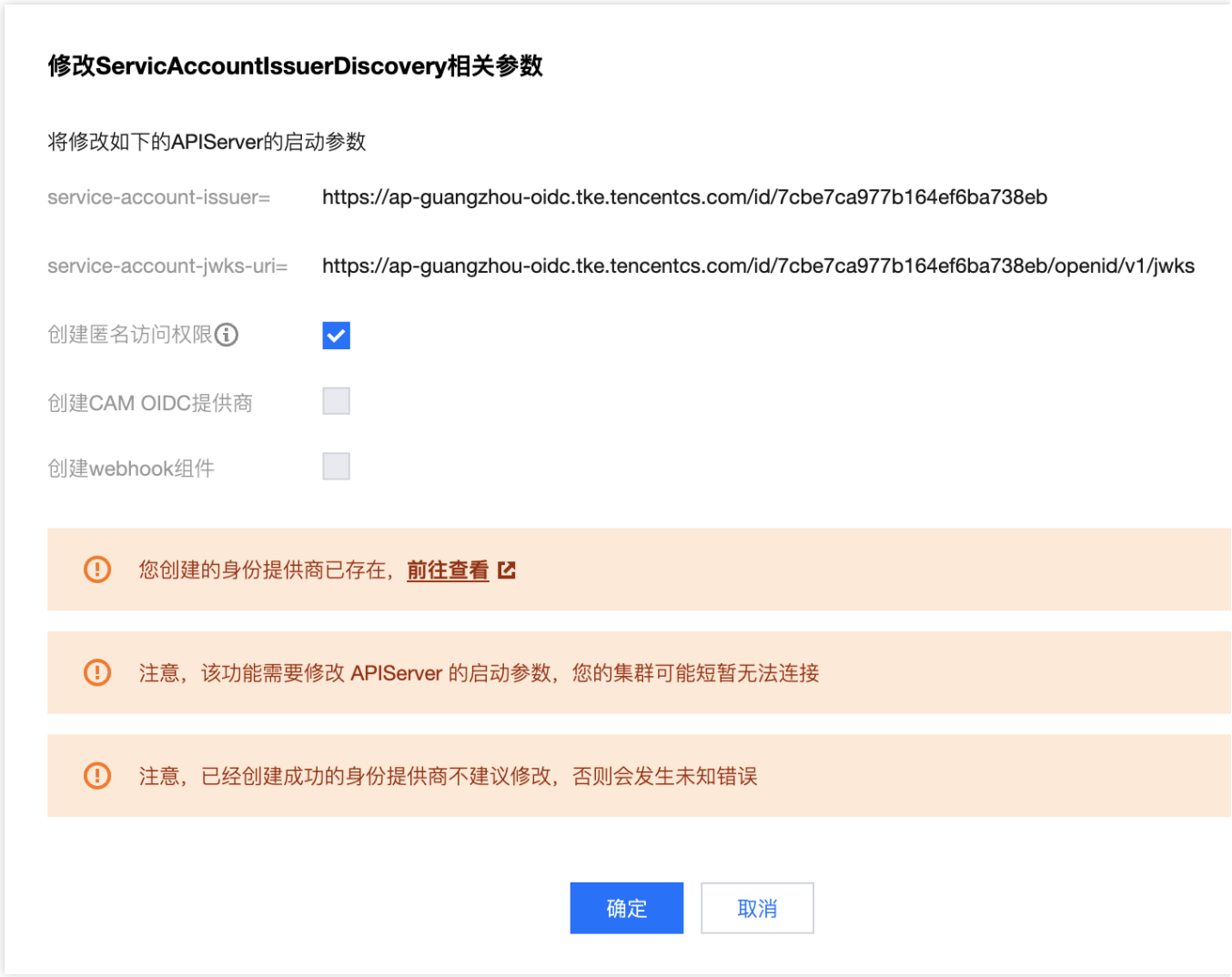
"service-account-issuer" 和 "service-account-jwks-uri" 参数值不允许编辑，采用默认规则。

步骤3：检查 CAM OIDC 提供商和 WEBHOOK 组件是否创建成功

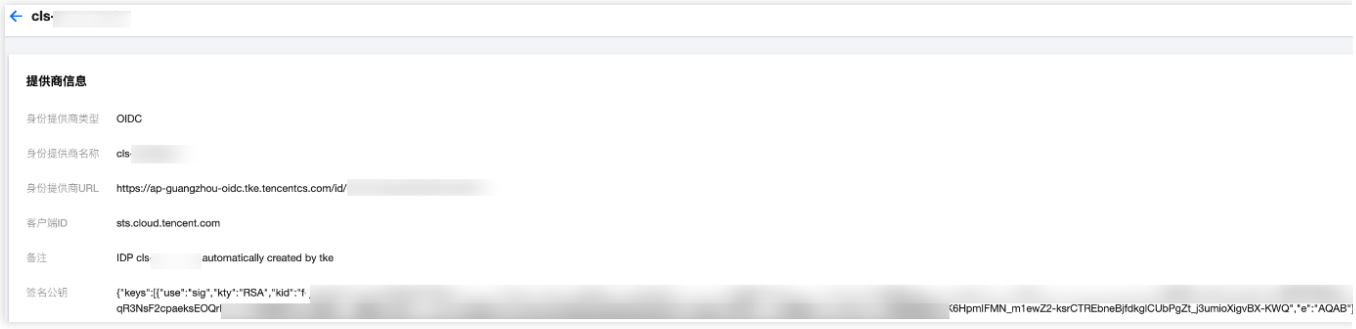
1. 在集群详情页中，单击 ServiceAccountIssuerDiscovery 右侧的

。

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：

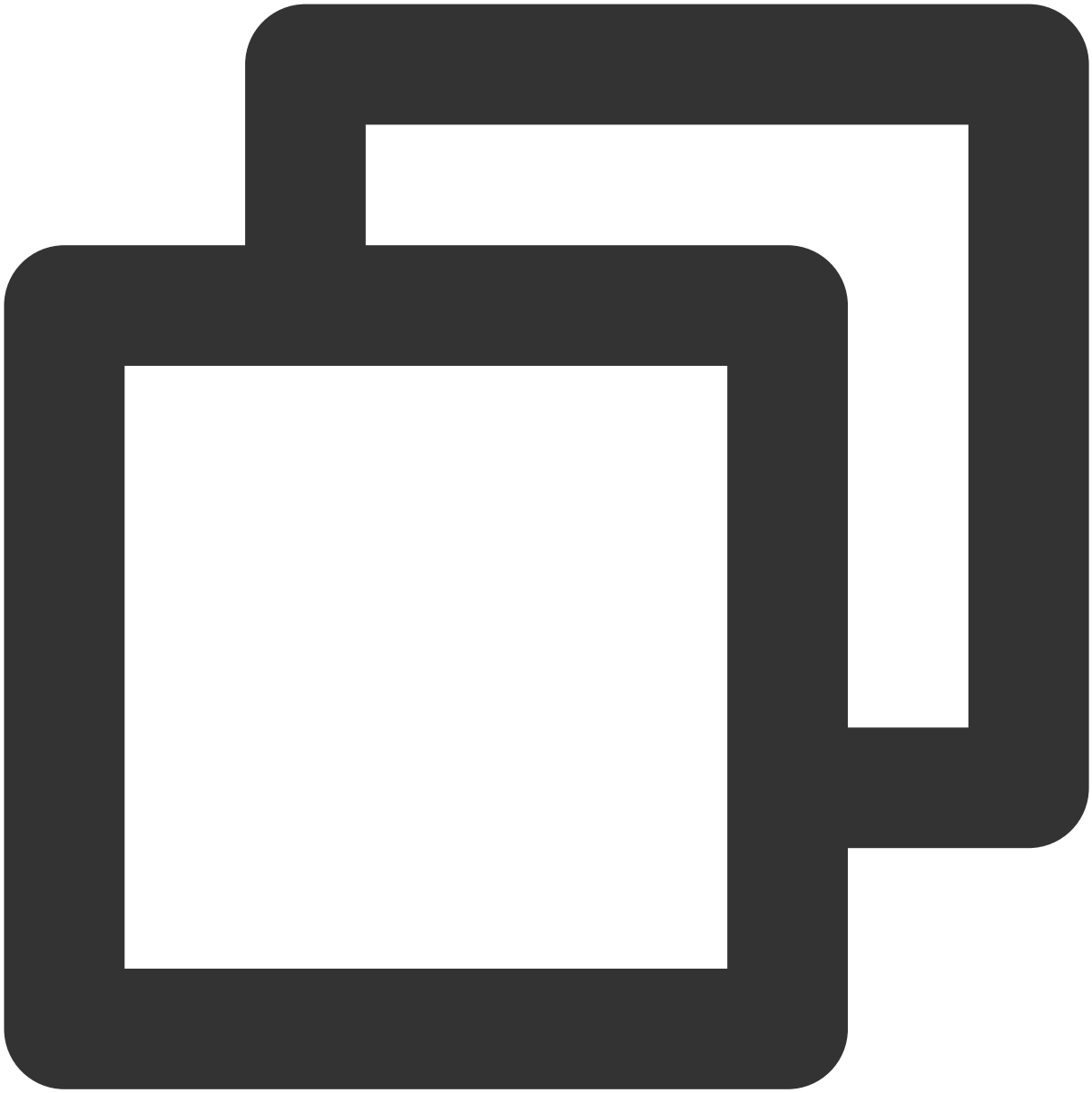


3. 查看您刚创建的 CAM OIDC 提供商详细信息。如下图所示：



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
kube-system   pod-identity-webhook-78c76****-9qrpj   1/1     Running   0
```

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。创建步骤请参见 创建 MySQL 实例。

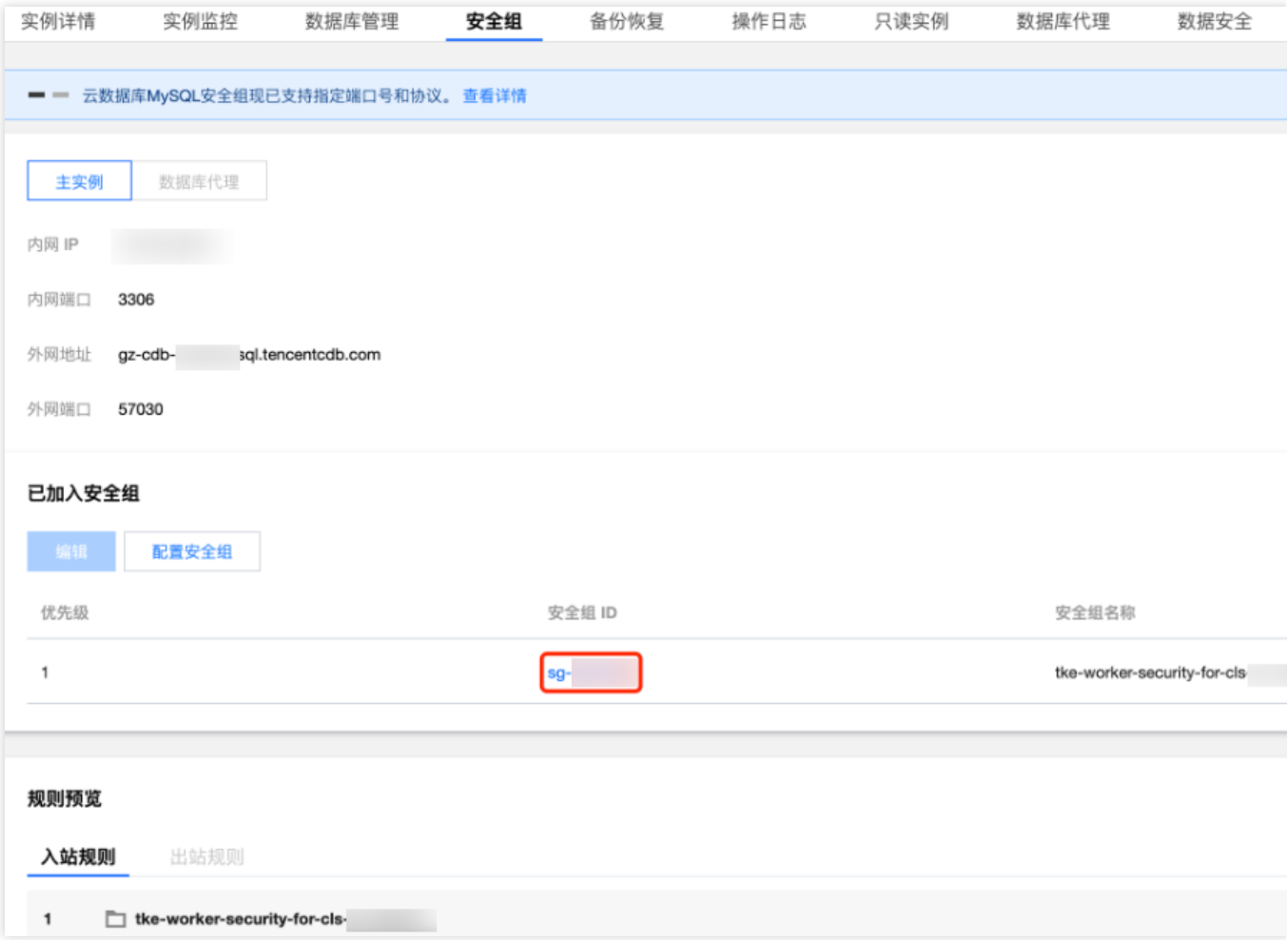
注意

外网地址的 value 值标识为 `$db_address` 。

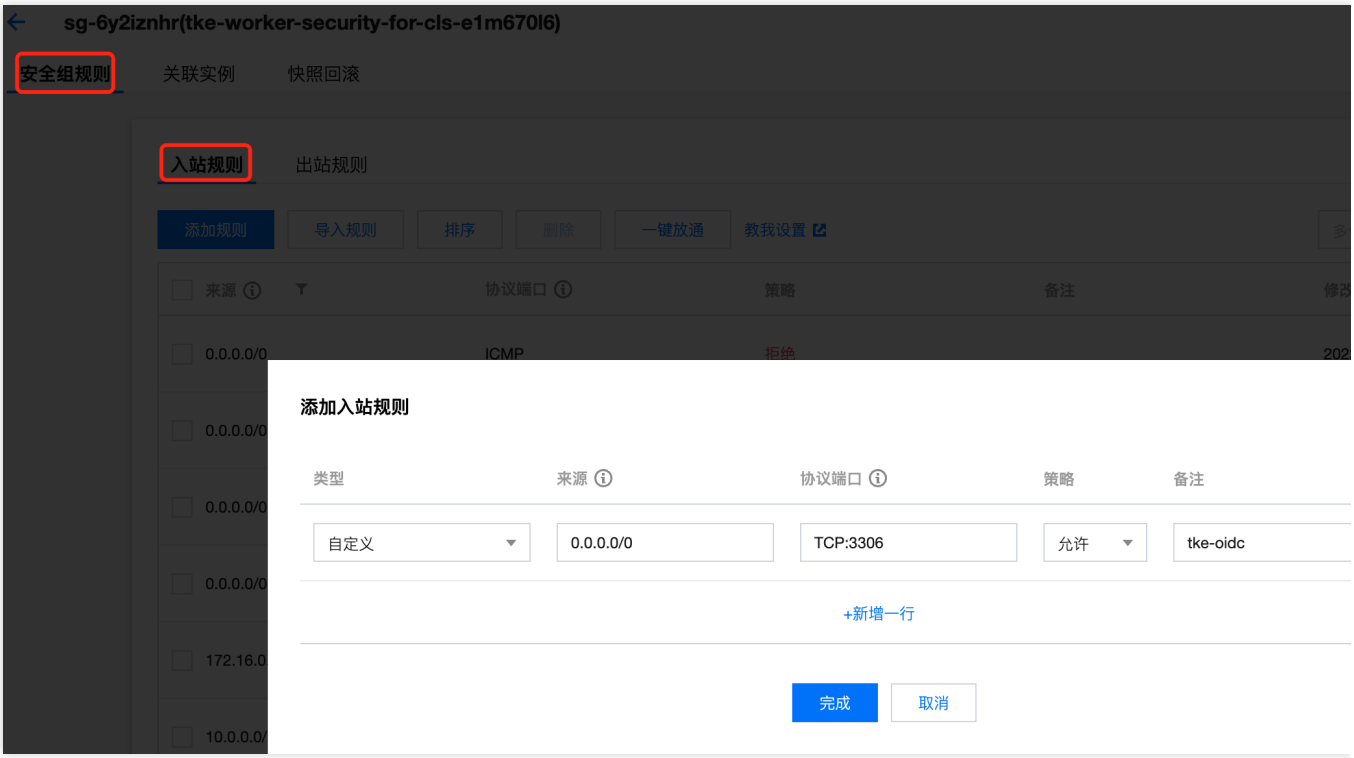
端口的 value 值标识为 `$db_port` 。

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：

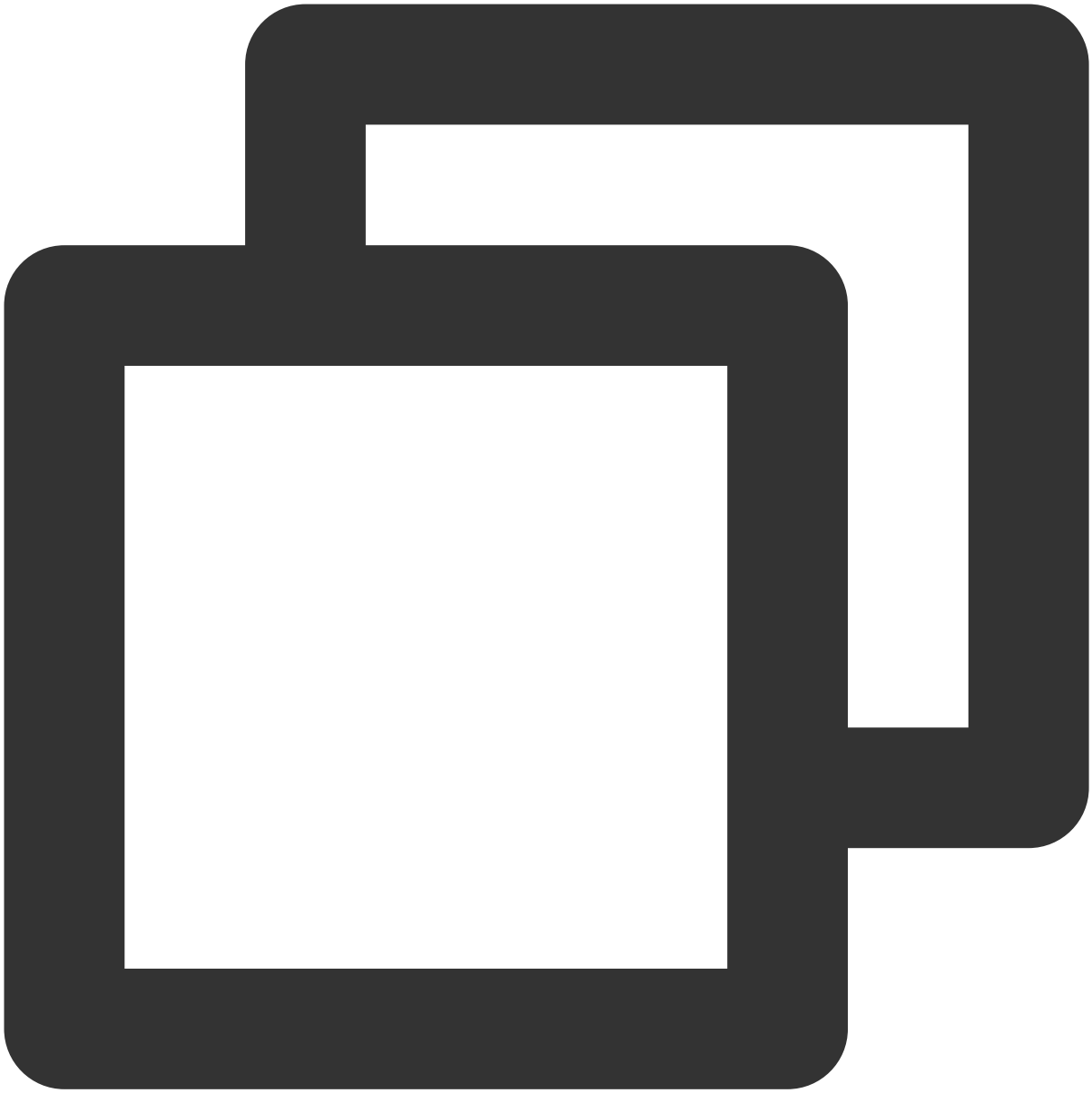


为了给 Kubernetes Pod 创建入站规则，您需要单击**安全组 ID**后跳转到安全组实例页面。在安全组实例详情页，选择**安全组规则 > 入站规则 > 添加规则**。在“添加入站规则”弹窗中，进行入站规则的创建。本示例中使用**来源**为 `0.0.0.0/0`，**协议端口**为 `TCP:3306`。



步骤6：测试数据库连接性

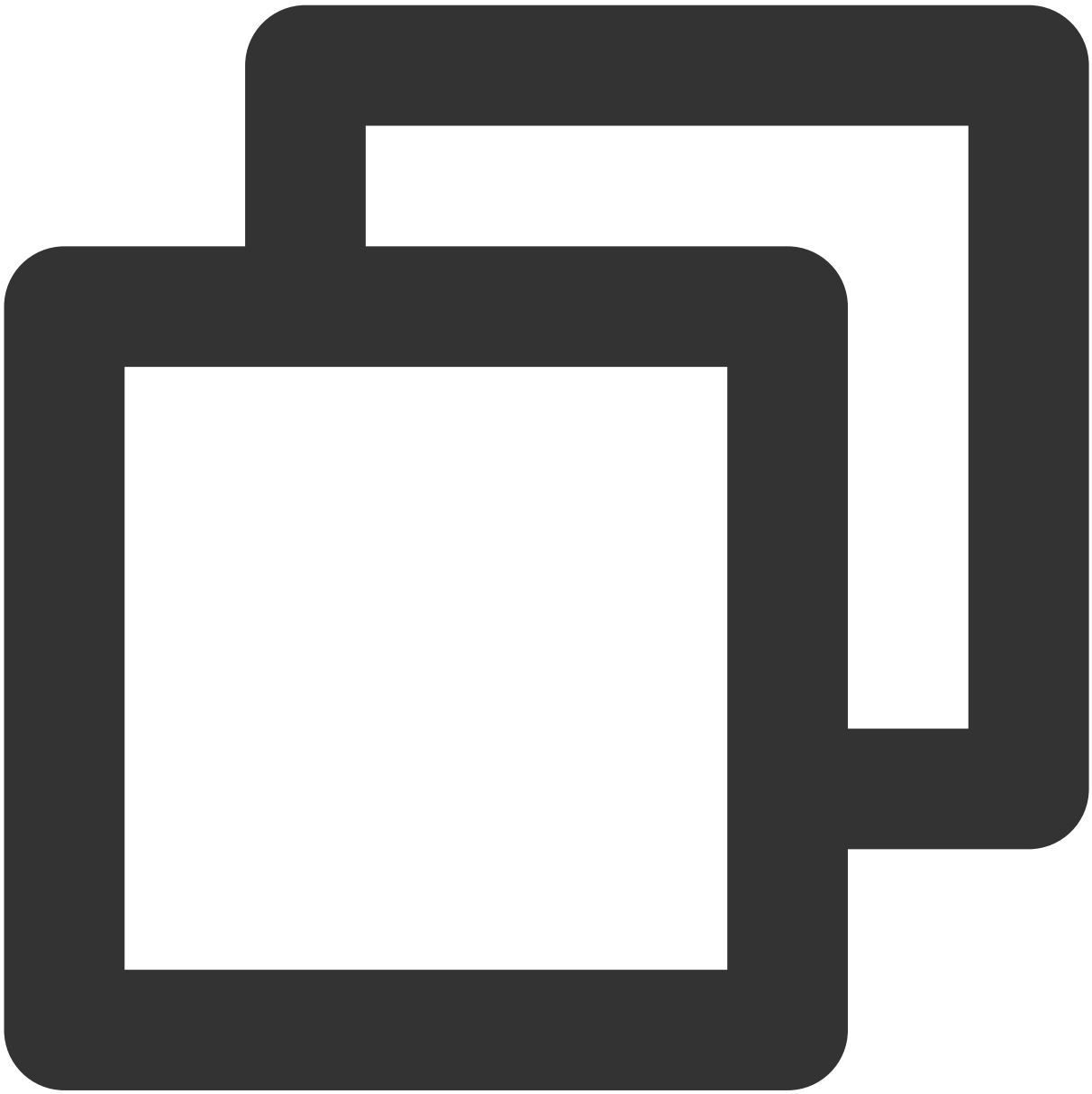
在安装 MySQL 客户端的实例中，确认您使用用户名 root 和您在创建数据库设置的密码连接到数据库。如果无法连接到数据库，请返回查看是否开启 公网 及是否正确配置 安全组。



```
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：

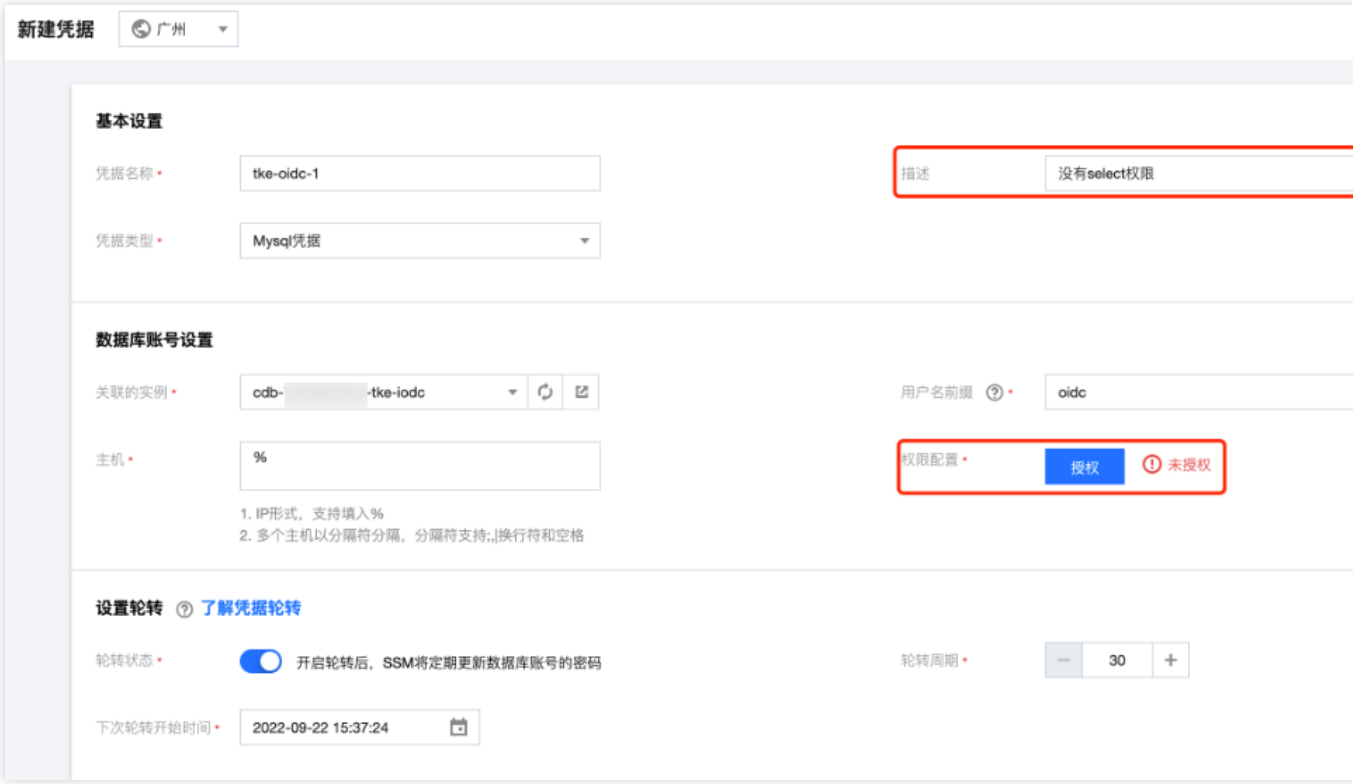
注意

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。

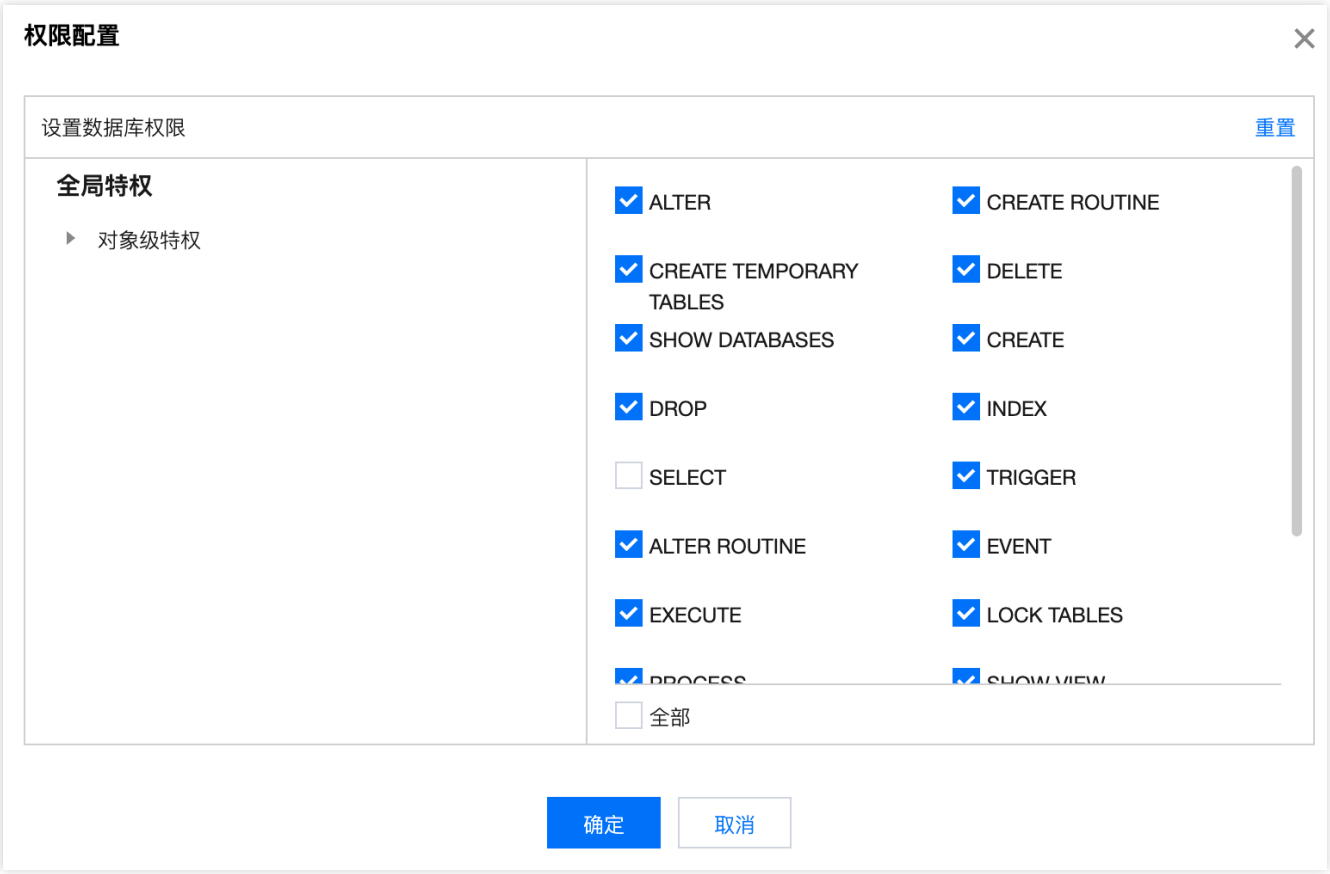
主机：是指客户端 IP，不指定时填写 `%` 。

权限配置：根据对数据库实例的操作需求进行授权。

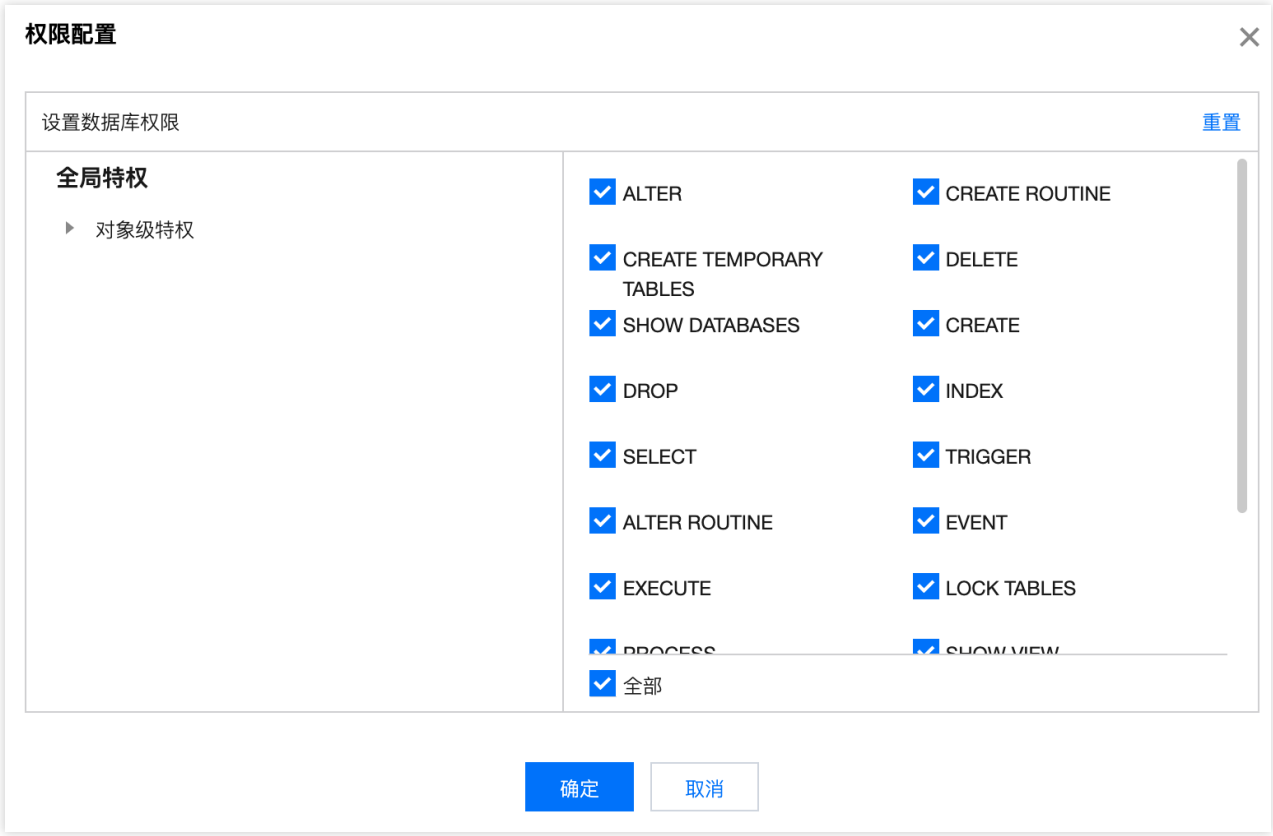
创建第一个数据库凭证

创建第二个数据库凭证

单击**授权**，在“权限配置”页面，勾选如下权限：



单击**授权**，在权限配置页面，勾选**全部**权限，如下图所示：

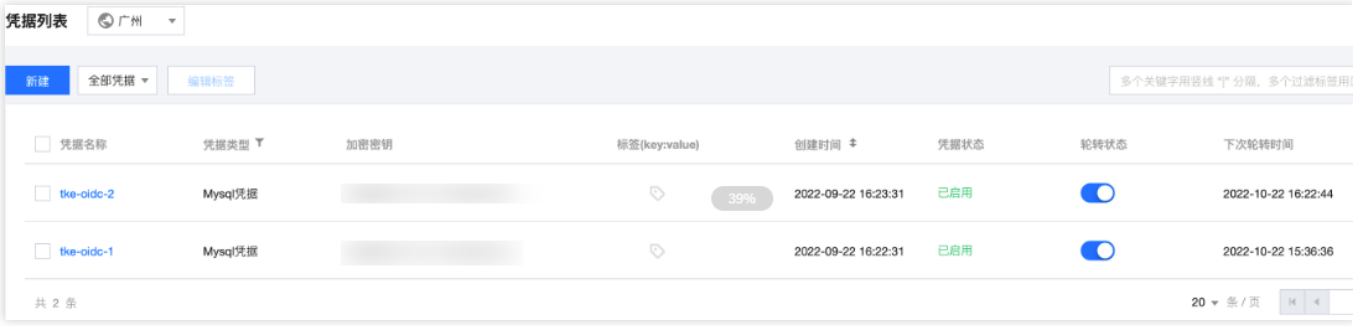


注意

凭据名称的 value 值标识为 `$ssm_name`

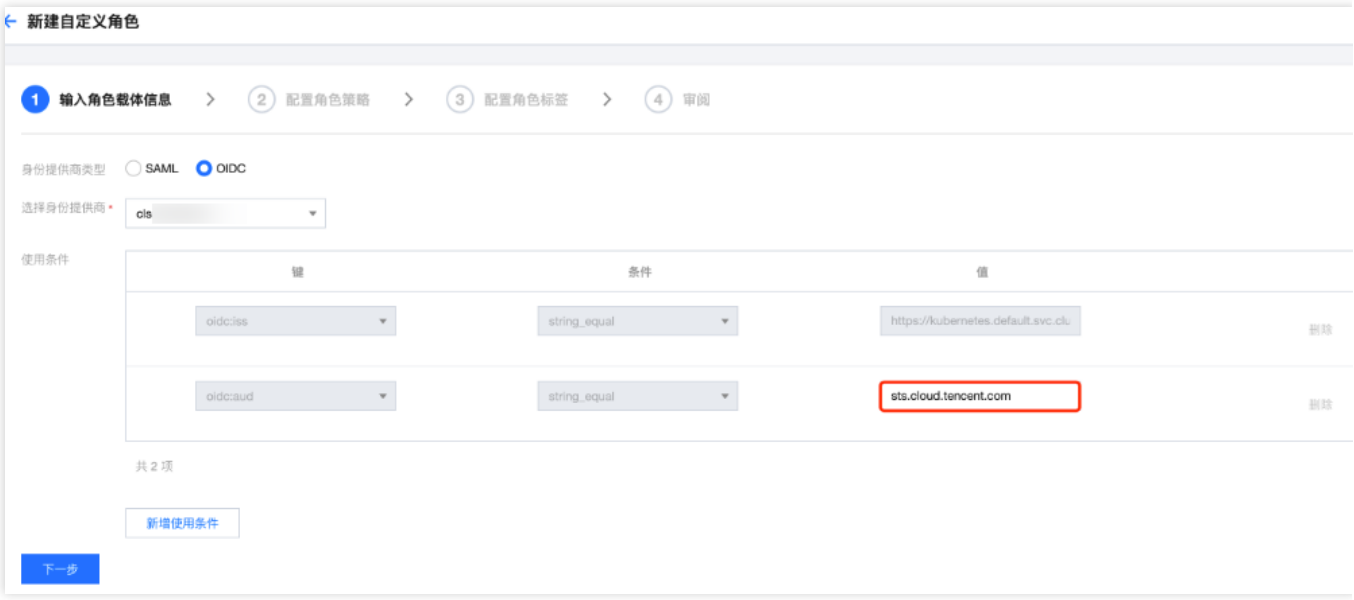
凭据所在地域标识为 `$ssm_region_name`

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

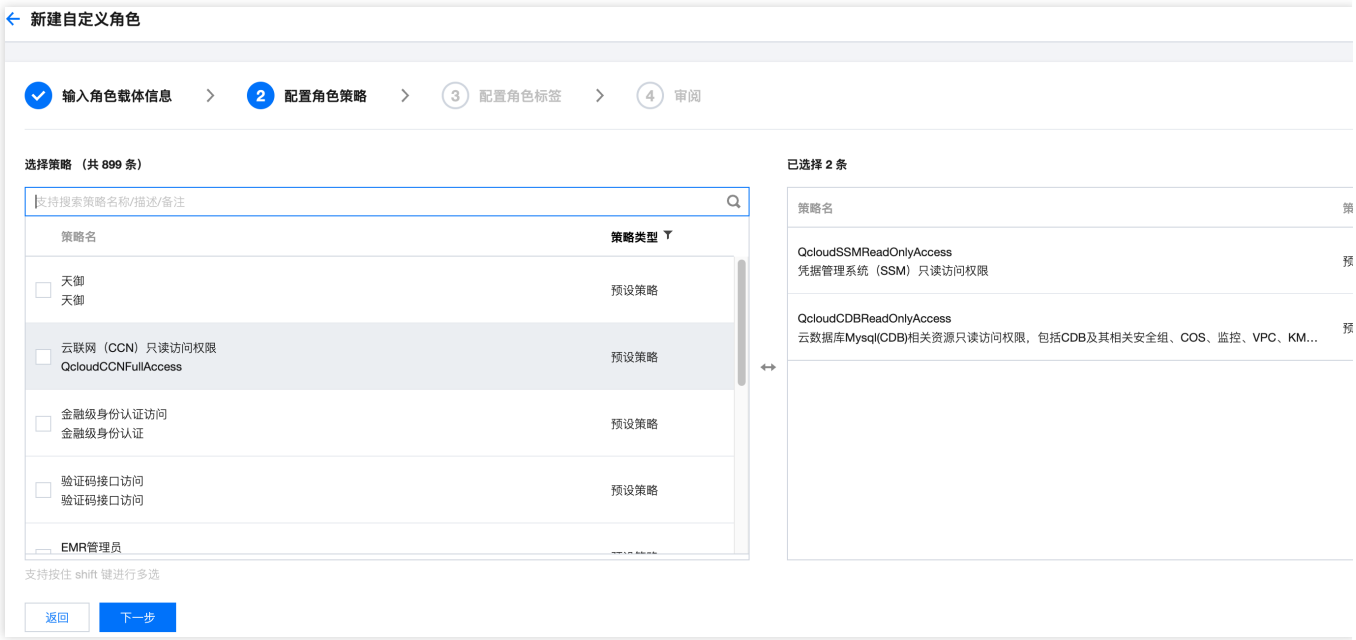
1. 登录 访问管理控制台。
2. 在角色页中，单击**新建角色 > 身份提供商**。
3. 在**新建自定义角色**页，参考以下信息进行设置。



注意：

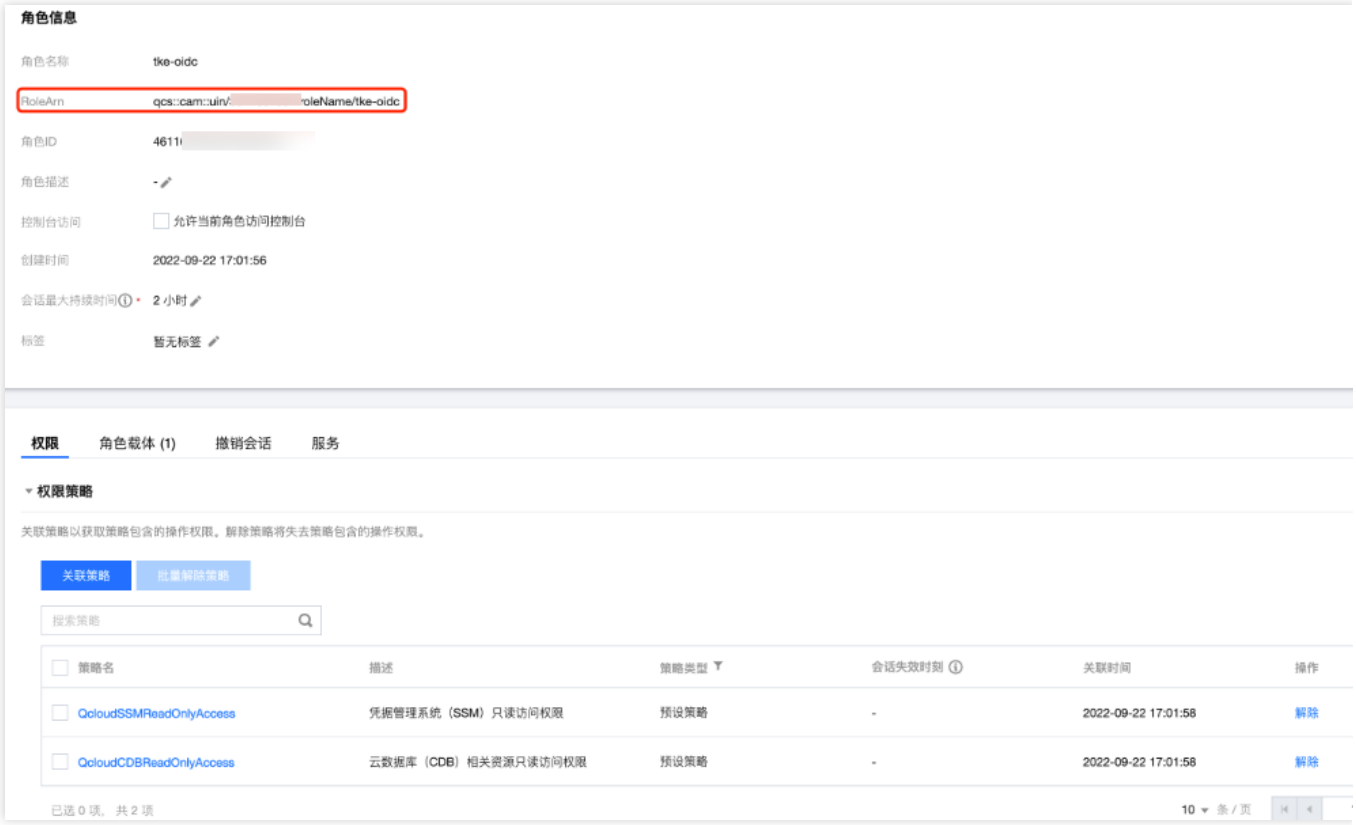
oidc:aud 的 value 值需要和 CAM OIDC 提供商的**客户端 ID** value 值保持一致。

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。

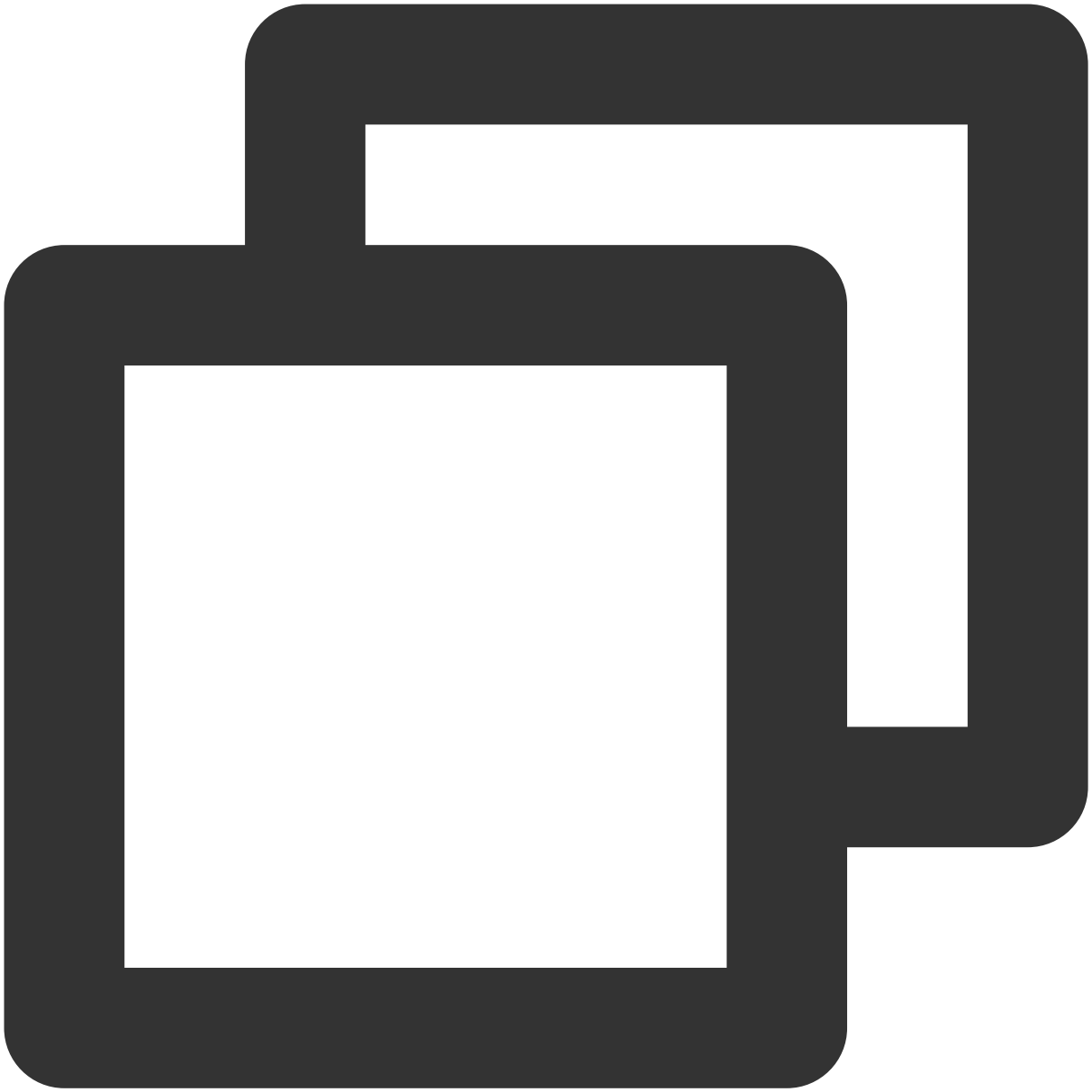


注意：

RoleArn的 value 值标识为 `$my_pod_role_arn` 。

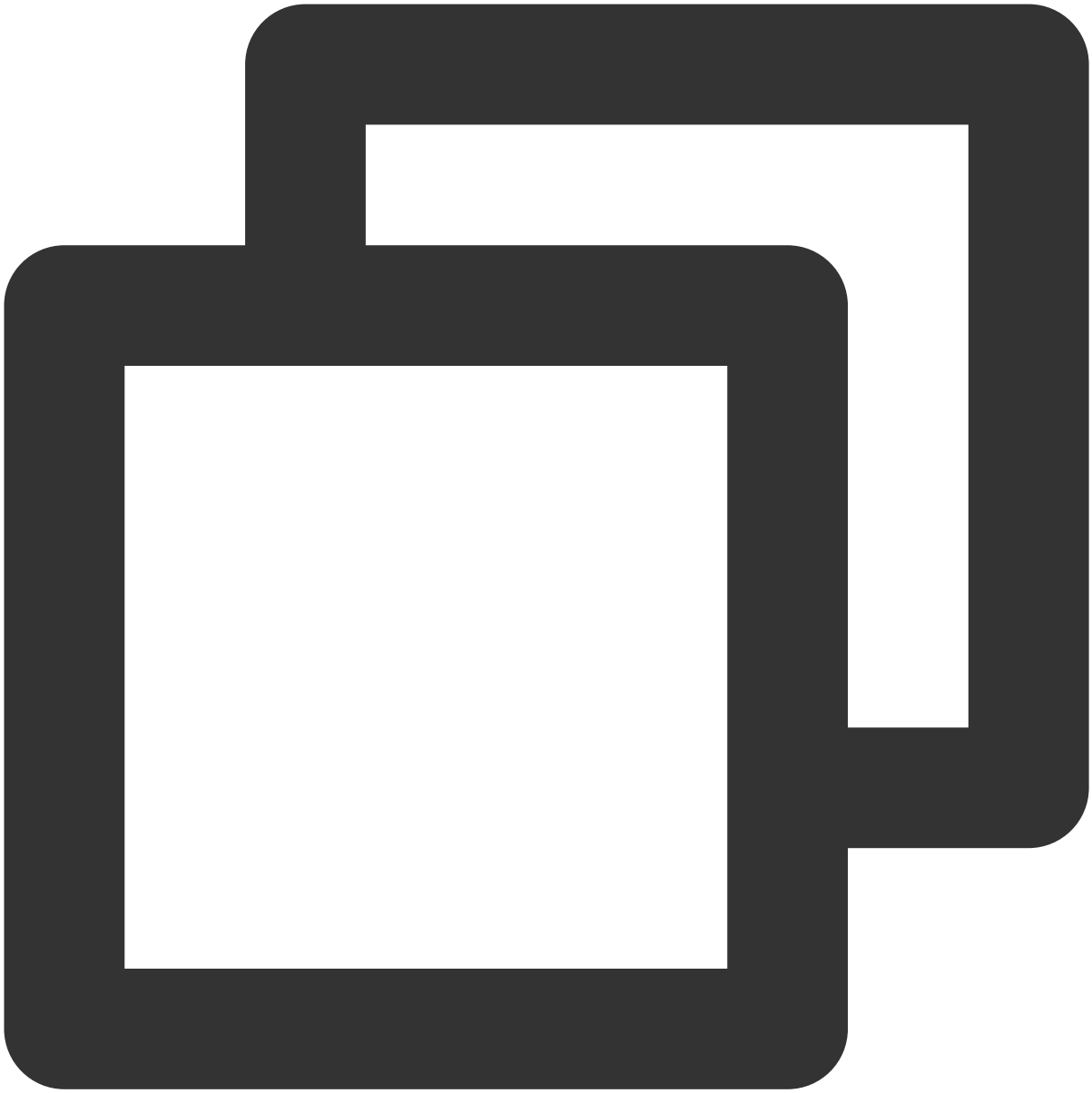
步骤10：部署示例应用程序

1. 创建一个 Kubernetes 命名空间来部署资源。



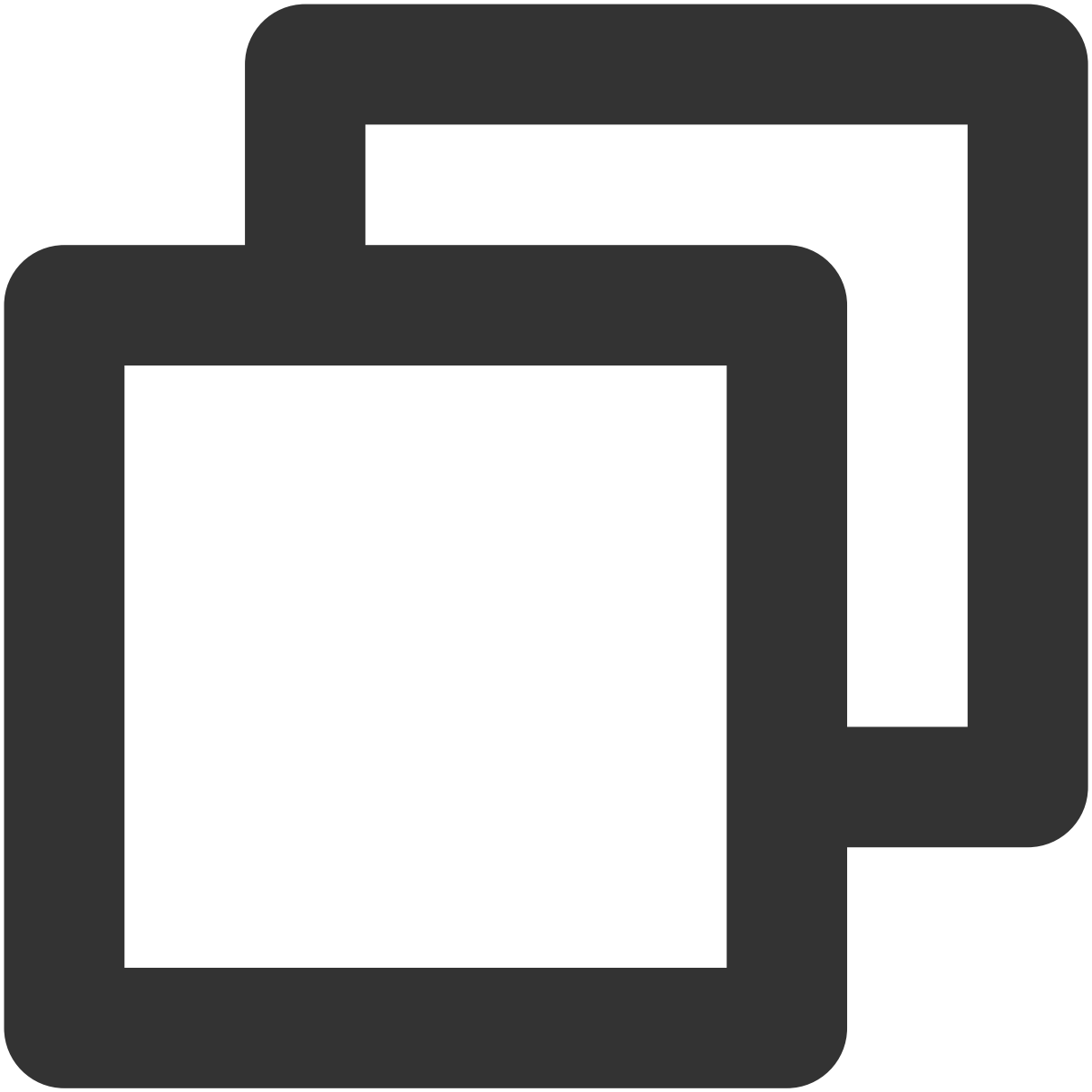
```
kubectl create namespace my-namespace
```

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。

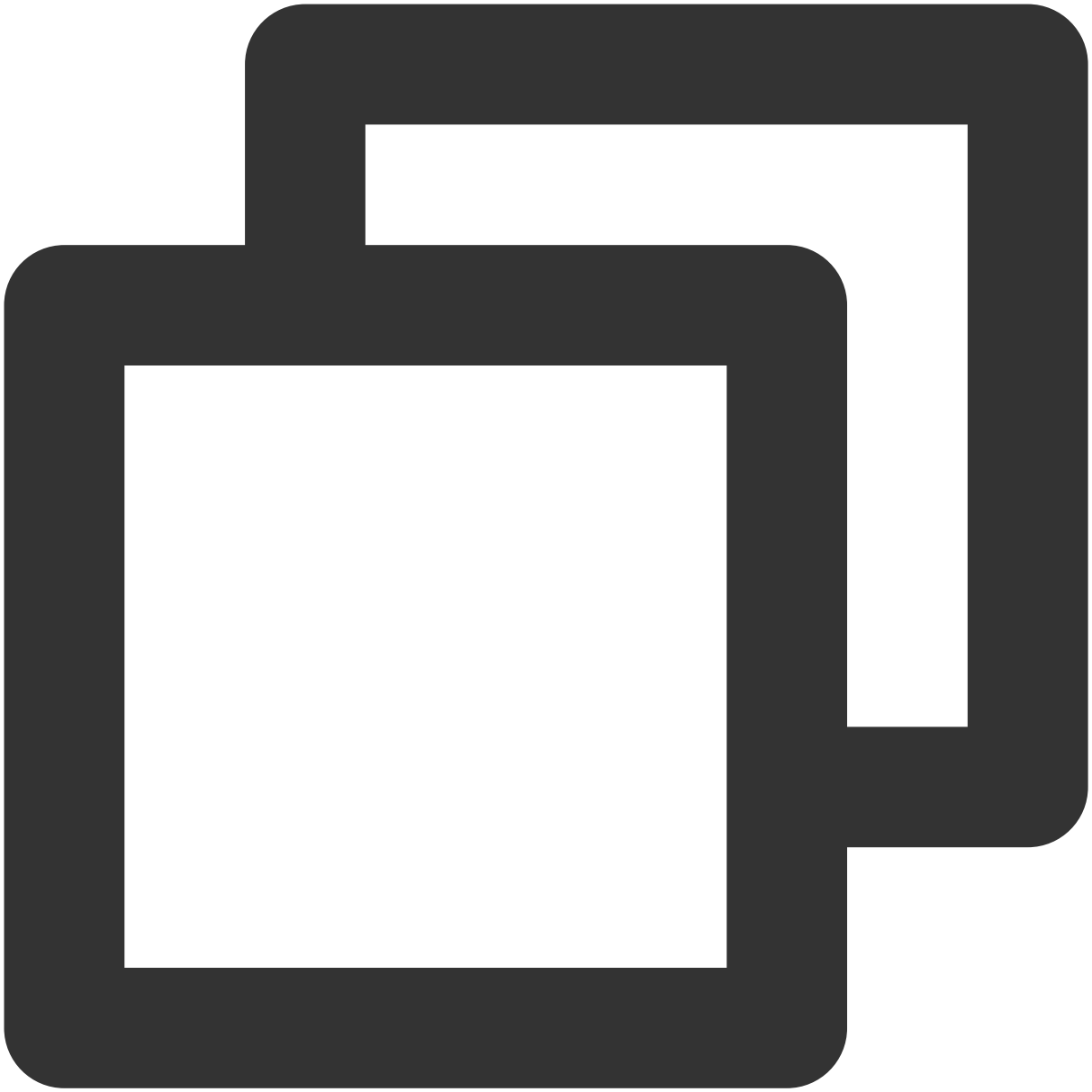


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app: my-app
  replicas: 1
  template:
```

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

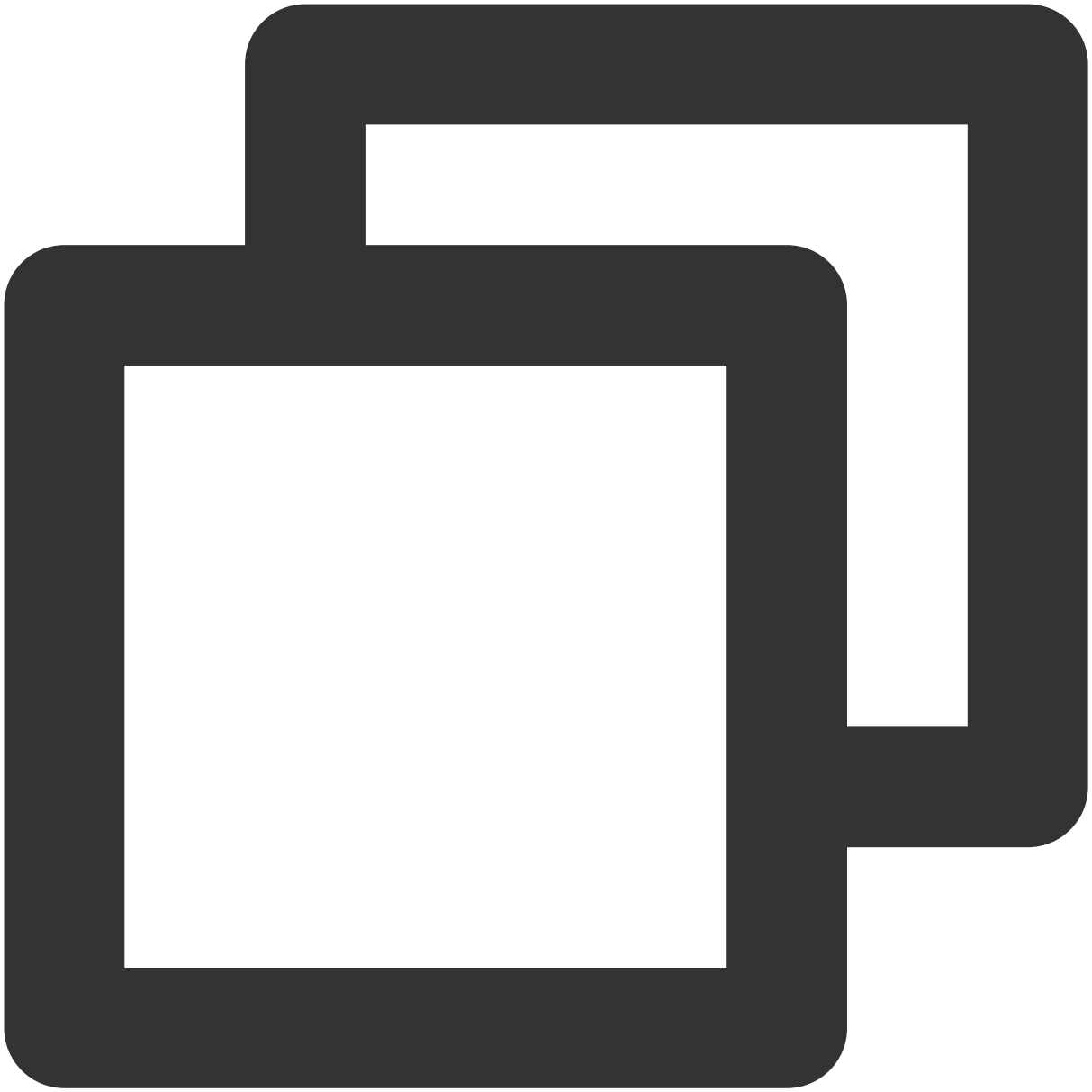
需注意，在本示例中，`$image` 选择 `ccr.ccs.tencentyun.com/tkeimages/sample-application:latest`，该镜像集成了编译的 demo文件，方便进行示例演示。您可以根据自身业务进行填写。

4. 部署示例。



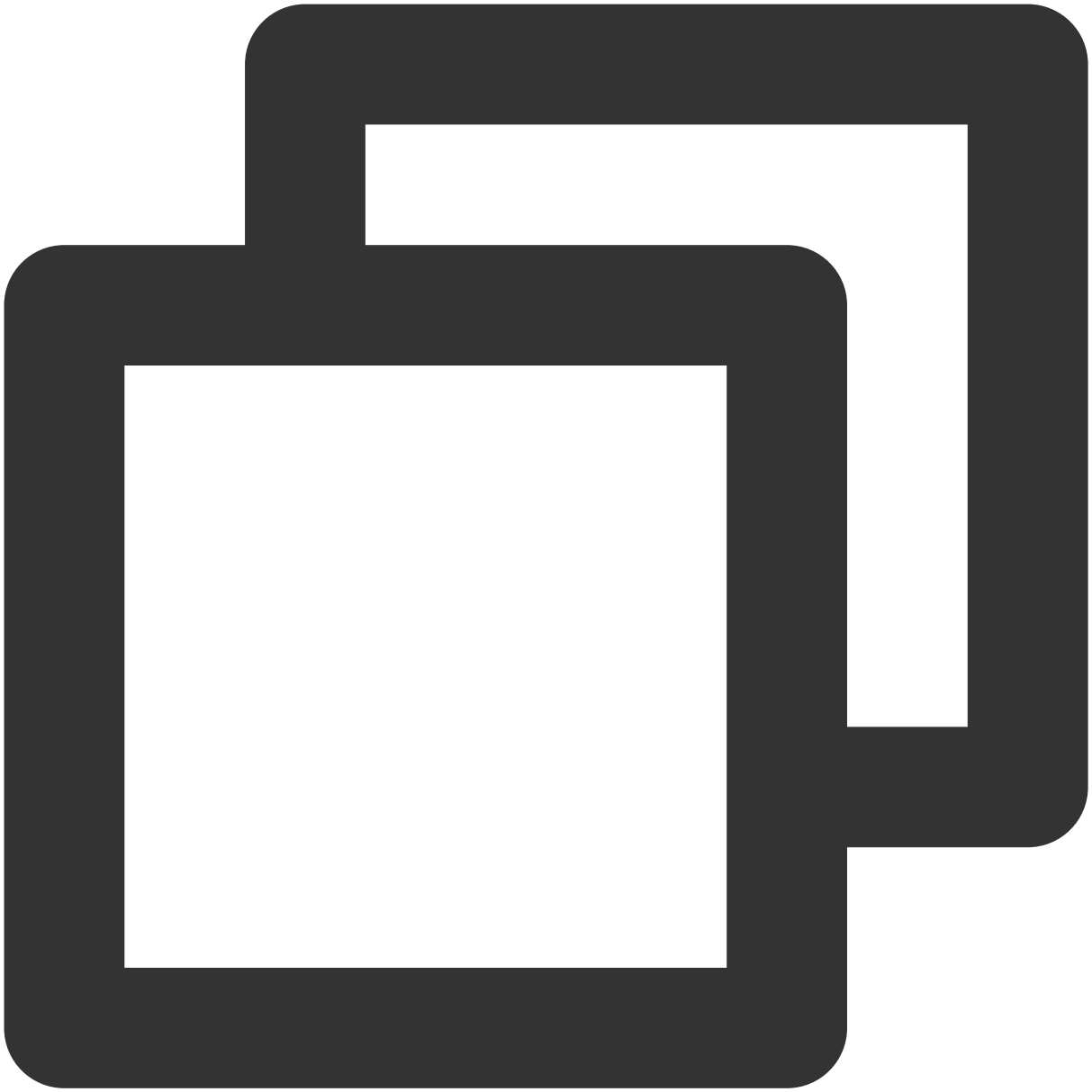
```
kubectl apply -f my-serviceaccount.yaml
kubectl apply -f sample-application.yaml
```

5. 查看使用示例应用程序部署的 Pod。



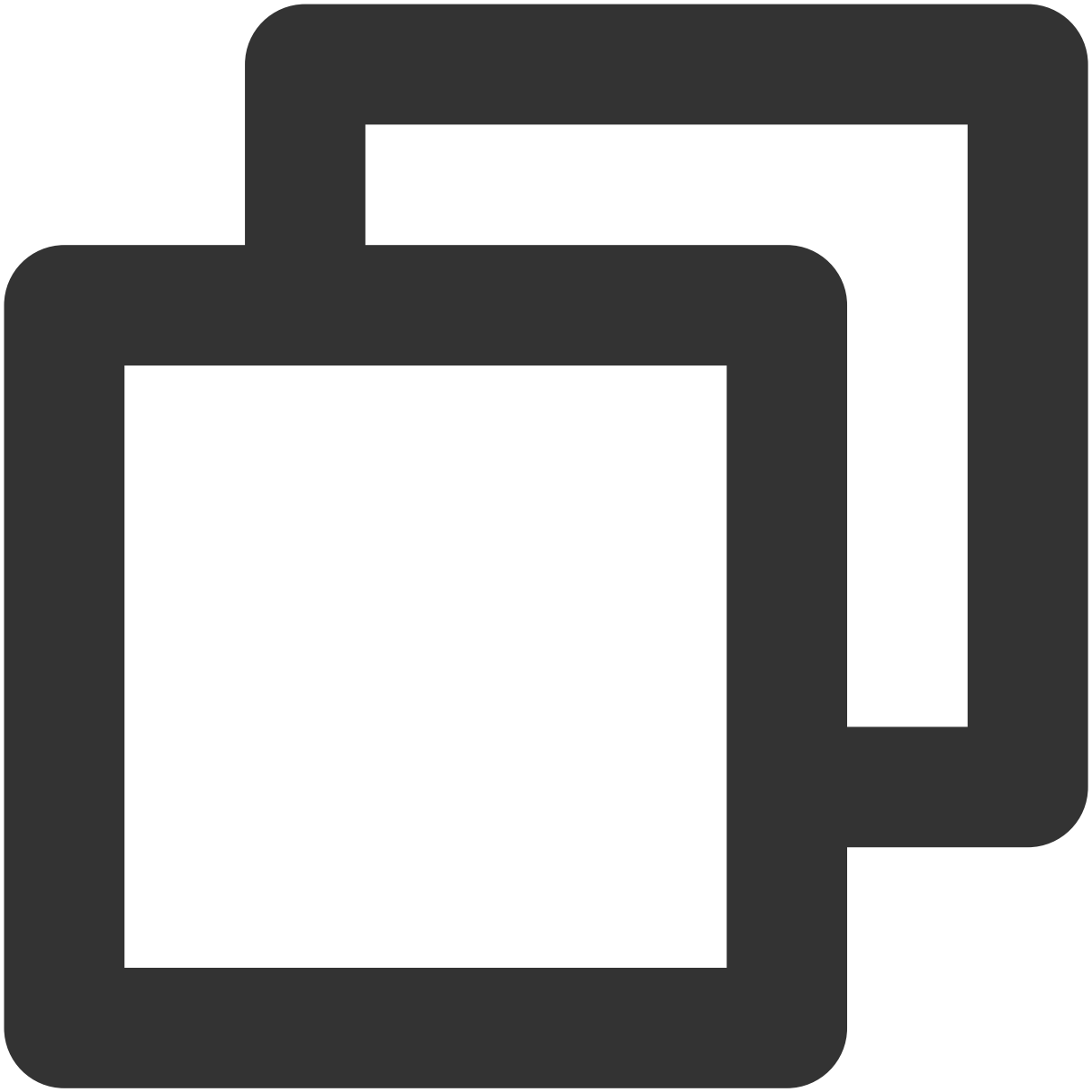
```
kubectl get pods -n my-namespace
```

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。



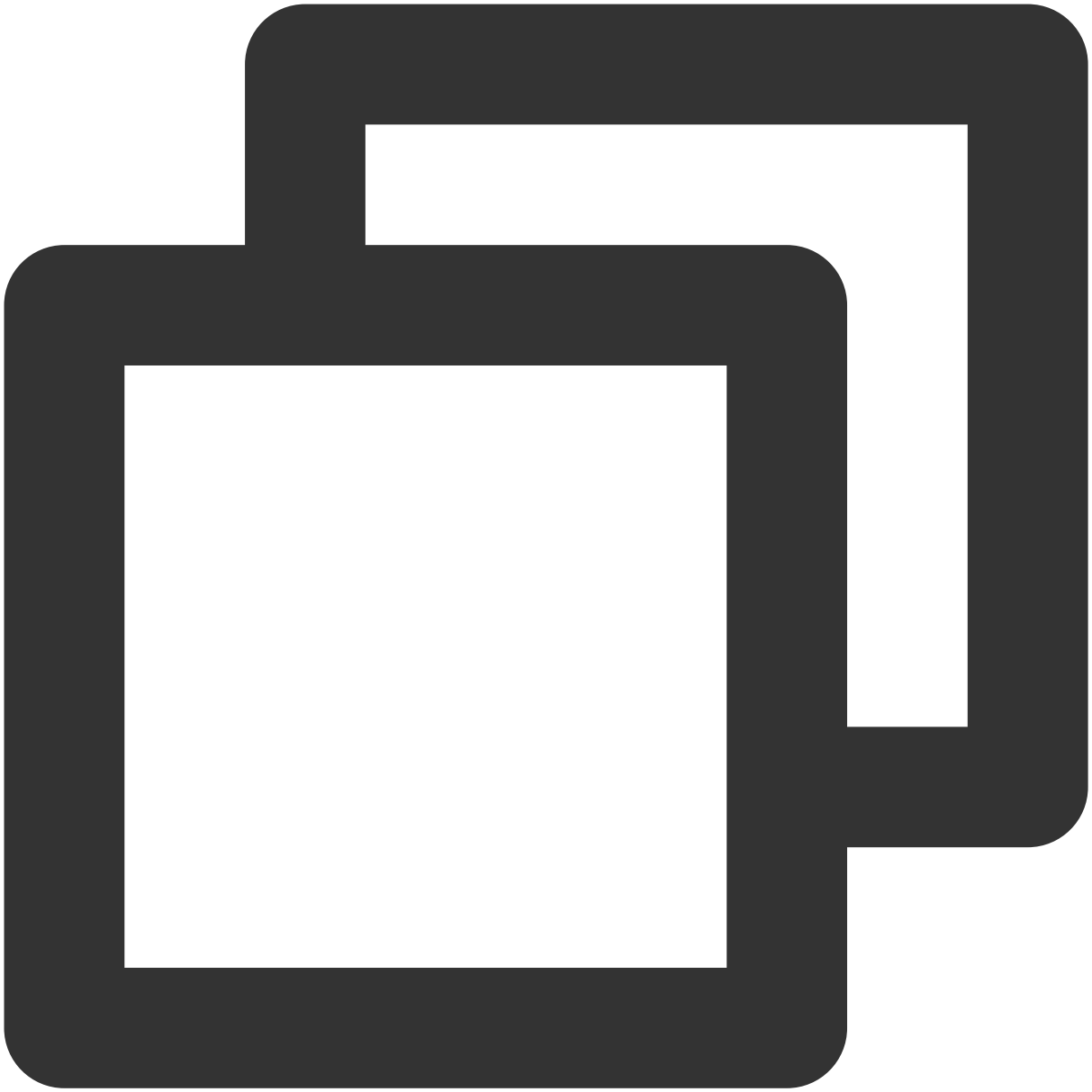
```
kubectl describe pod nginx-deployment-6bfd845f47-9zxld -n my-namespace
```

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@sha256:30e1d52127fe952b044d5606952eaa07fb20536cf195620497
    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)
```

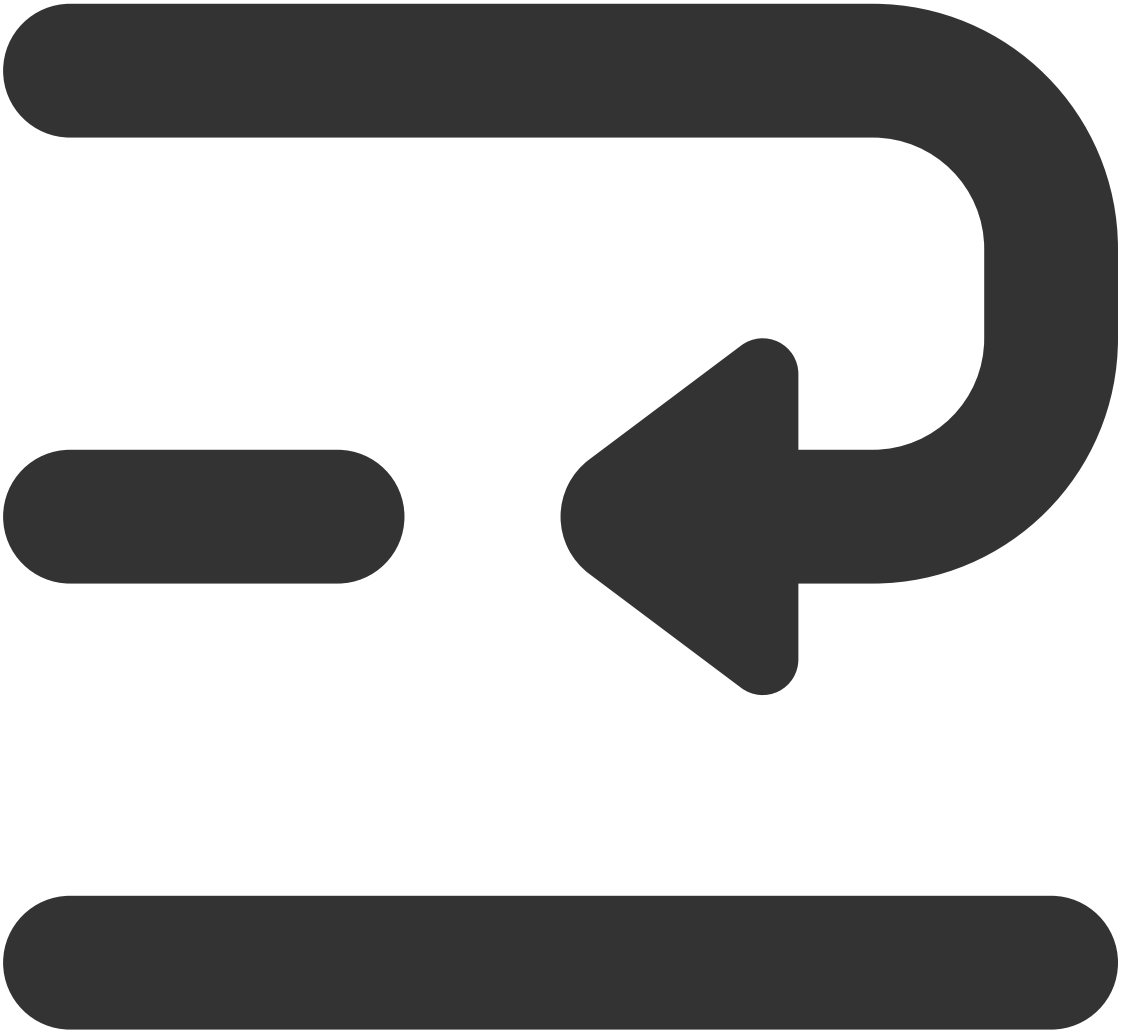
步骤11：访问数据库 demo 伪代码实现

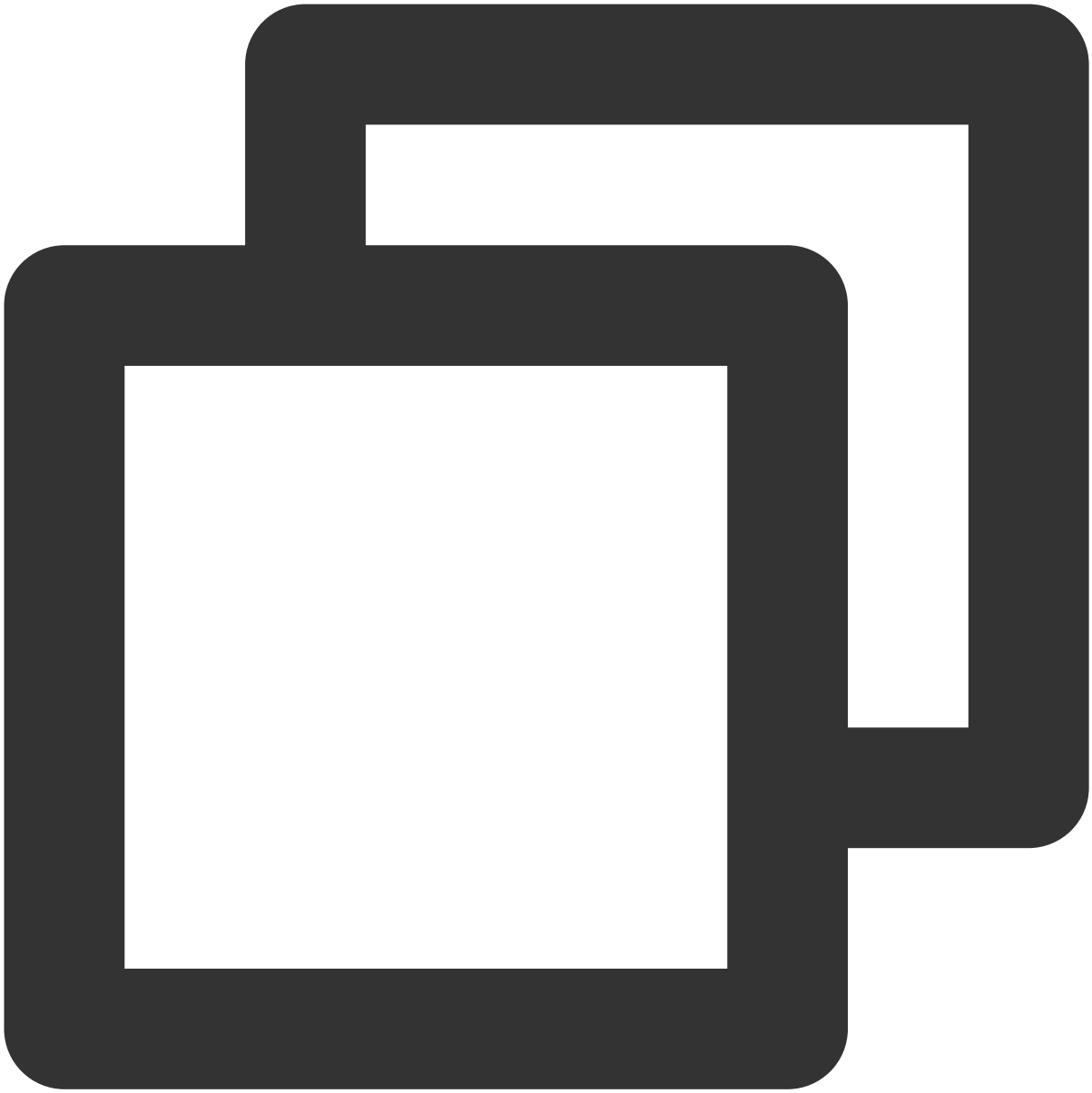
1. 确认子账号所有访问 AssumeRoleWithWebIdentity 接口的权限。如果没有权限请联系管理员添加。
2. 确认有访问 AssumeRoleWithWebIdentity 接口的权限后，请参考 凭证管理 中步骤5获取访问 DB + SSM 的临时密钥。
3. 克隆 ssm-rotation-sdk-golang 代码。



```
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"
```

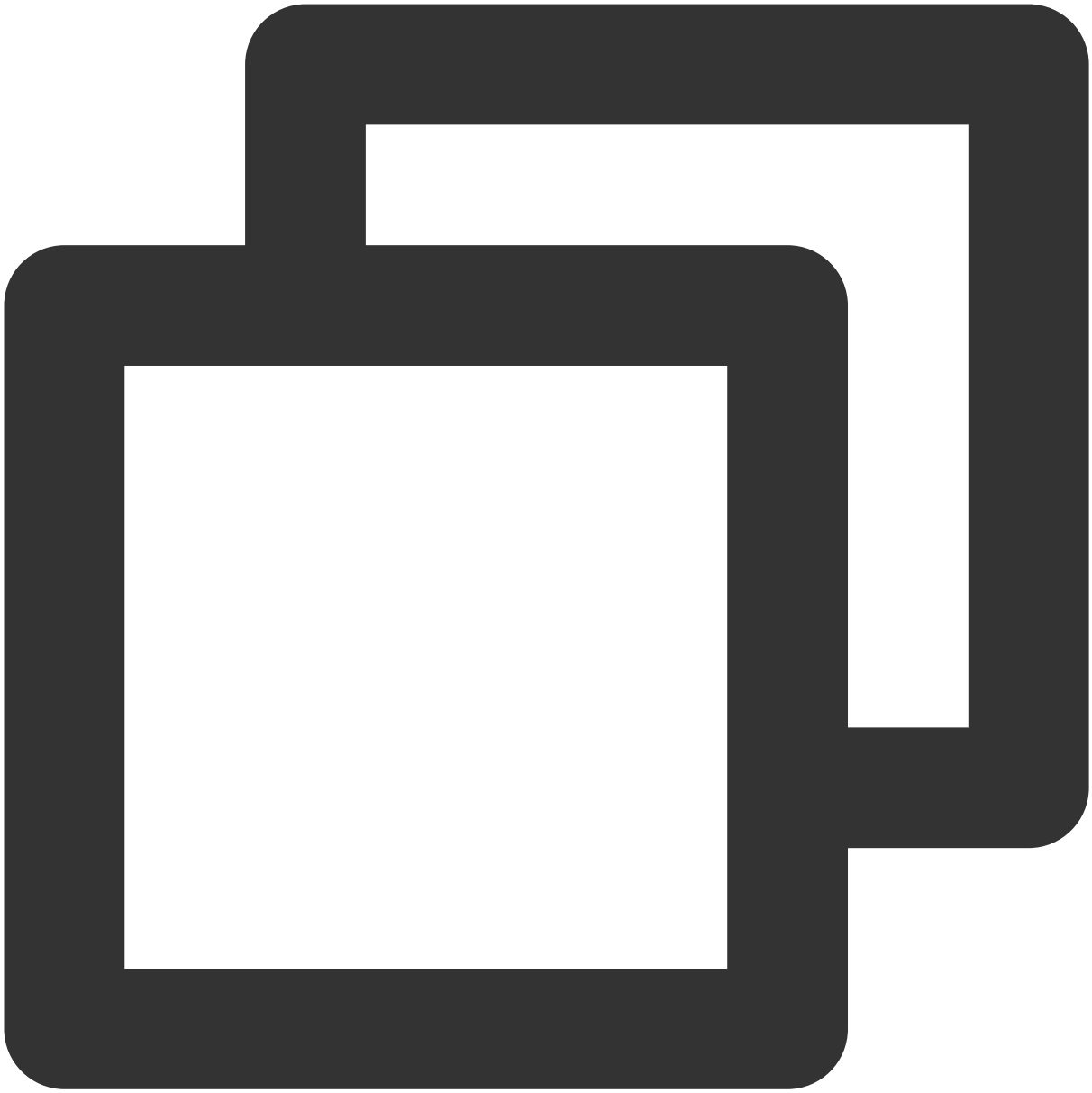
```
)

var (
        roleArn, tokenPath, providerId, regionName, saToken string
        secretName, dbAddress, dbName, ssmRegionName        string
        dbPort                                              uint64
        dbConn                                              *db.DynamicSecretRotati
        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, cre
        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
                        ParamStr:            "charset=utf8&loc=Local",
                },
                SsmServiceConfig: &ssm.SsmAccount{
                        SecretId:  credential.TmpSecretId,  // 需填写实际可用的SecretI
                        SecretKey: credential.TmpSecretKey, // 需填写实际可用的SecretK
                        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
        })
        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")
}
```

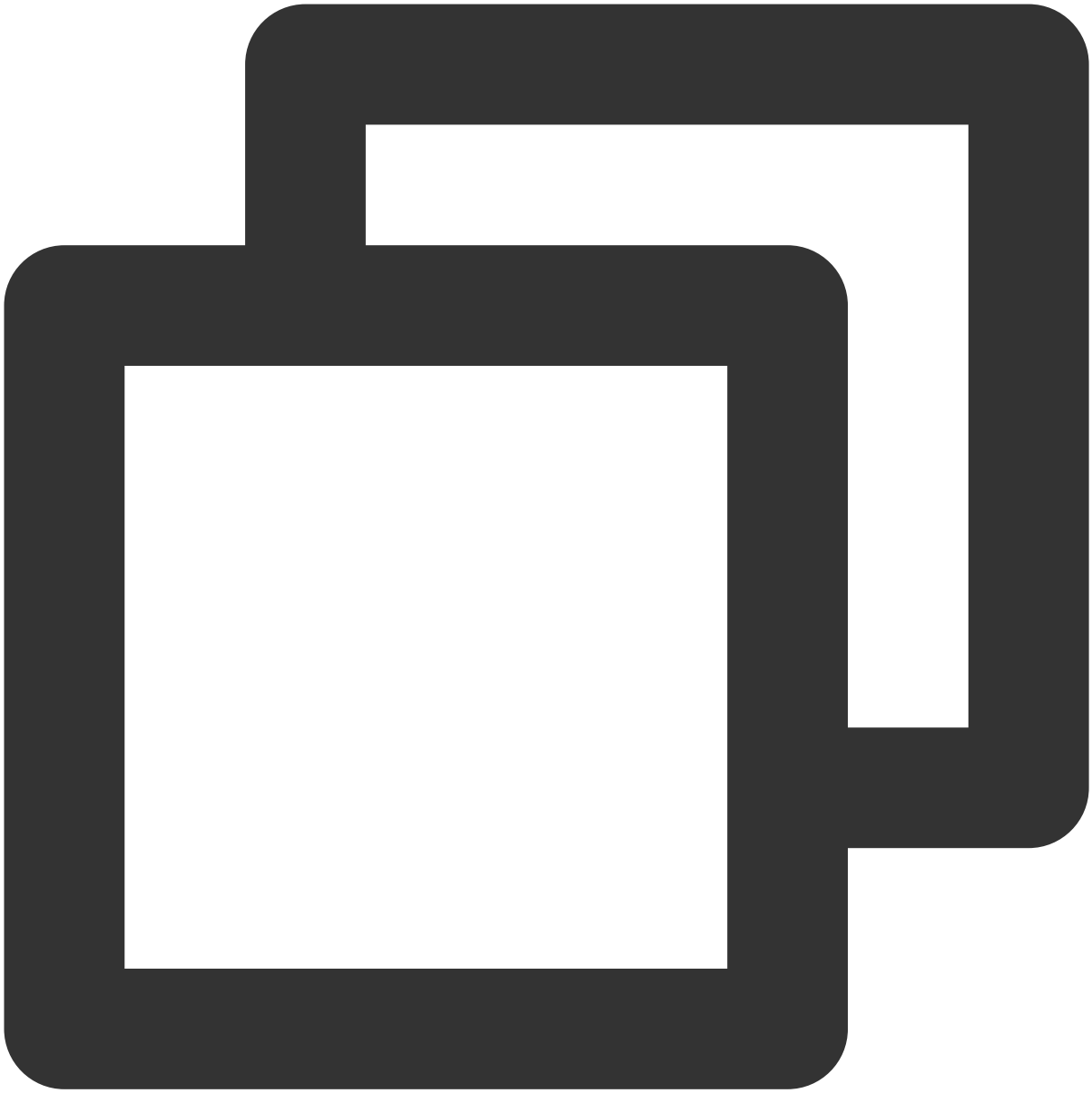
步骤12：测试 demo 示例

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_addres
```

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

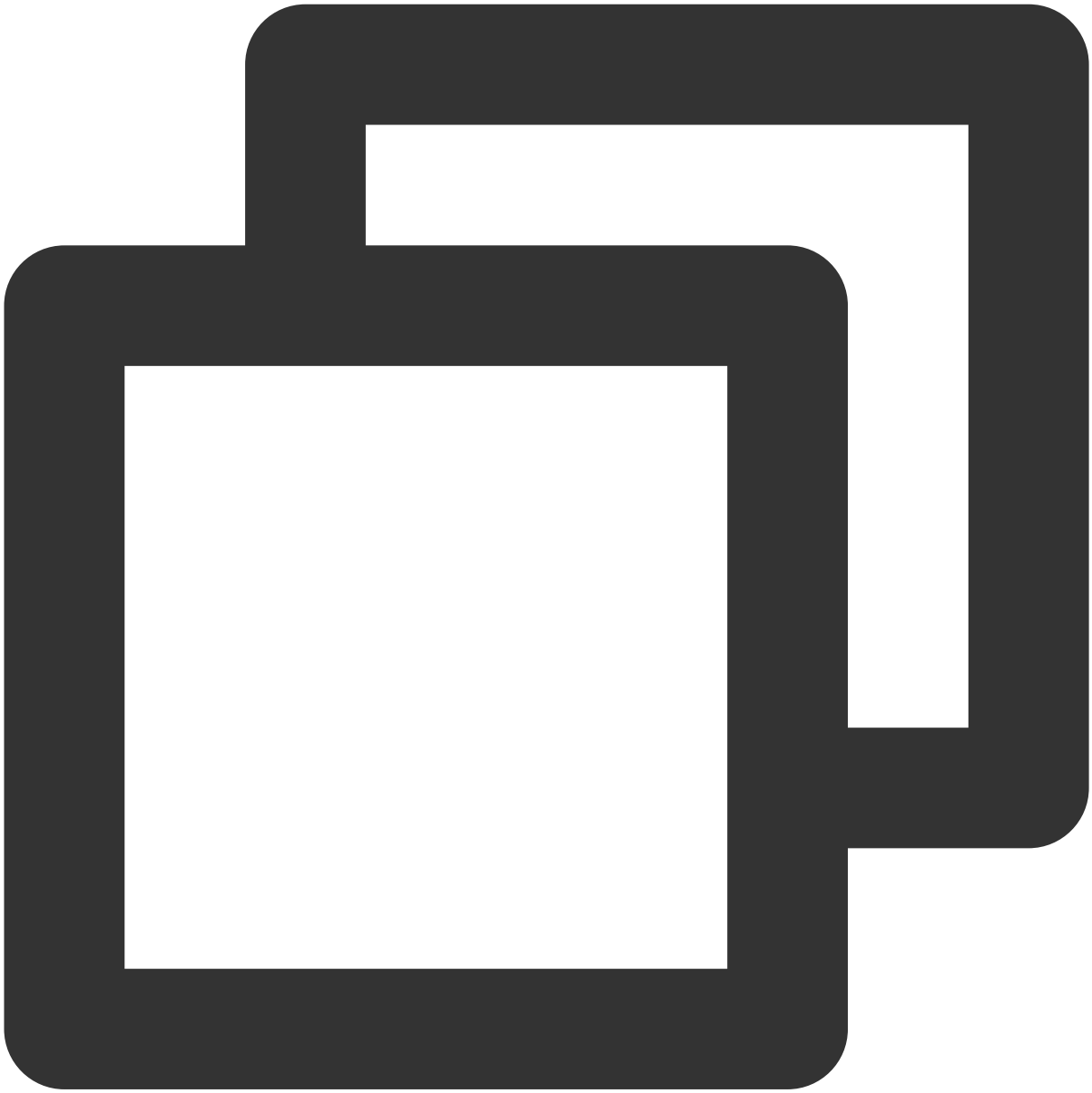
该组件权限是当前功能实现的最小权限依赖。

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	mutatingwebhookconfigurations	get/update

的资源注入客户端的证书。		

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
```

服务部署
合理利用节点资源
概述

最近更新时间：2022-04-18 10:40:59

将已容器化的业务部署至 Kubernetes 的过程并不复杂，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、更好的进行资源调度、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：2022-04-18 10:46:03

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 nameapsace 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：2020-06-29 19:42:52

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：2020-06-29 19:42:52

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：2022-08-02 17:19:05

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最佳实践，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。所以我们通常需要给应用部署多个副本，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。如果无法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 K8S v1.18 默认启用，建议 v1.18 及其以上的集群使用 `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，若有节点因其它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
```

- 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 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
```

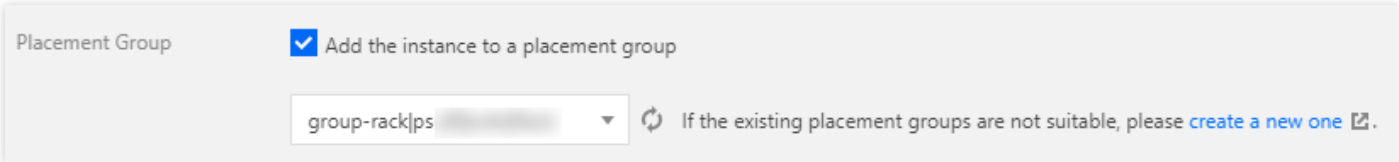
使用置放群组从物理层面实现容灾

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。

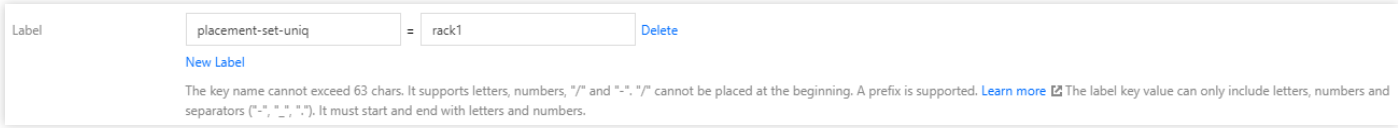
   > 注意：
   > 置放群组需与 TKE 独立集群在同一地域。

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”页面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
   - 示例2

   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
```

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: extensions/v1beta1
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。

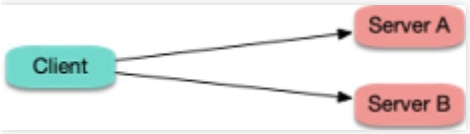
工作负载平滑升级

最近更新时间：2022-10-12 16:05:09

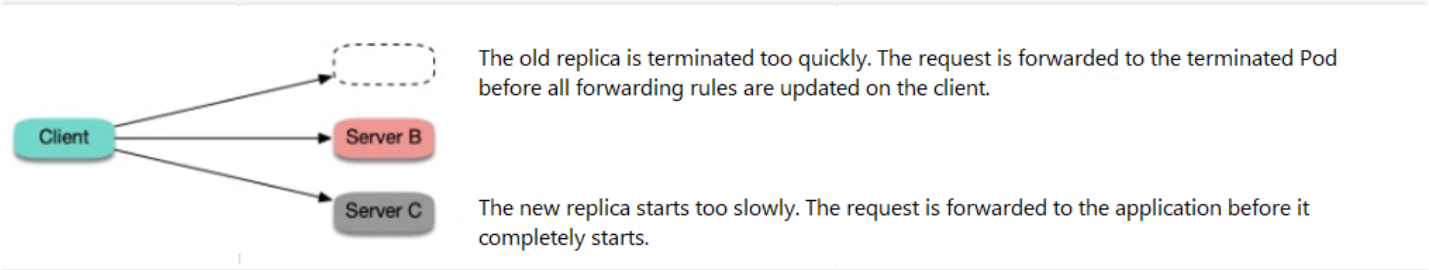
解决了服务单点故障和驱逐节点时导致的可用性降低问题后，我们还需要考虑一种可能导致可用性降低的场景，那就是滚动更新。为什么服务正常滚动更新也可能影响服务的可用性呢？可能存在以下原因。

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：2020-07-06 17:37:24

本文将介绍如何把在本地的 docker 中已经调试完毕的容器，在迁移到腾讯云容器服务平台中运行时，对于 docker run 中的参数如何跟腾讯云容器控制台的参数进行对应。这里我们以创建一个简单的 gitlab 服务为例。

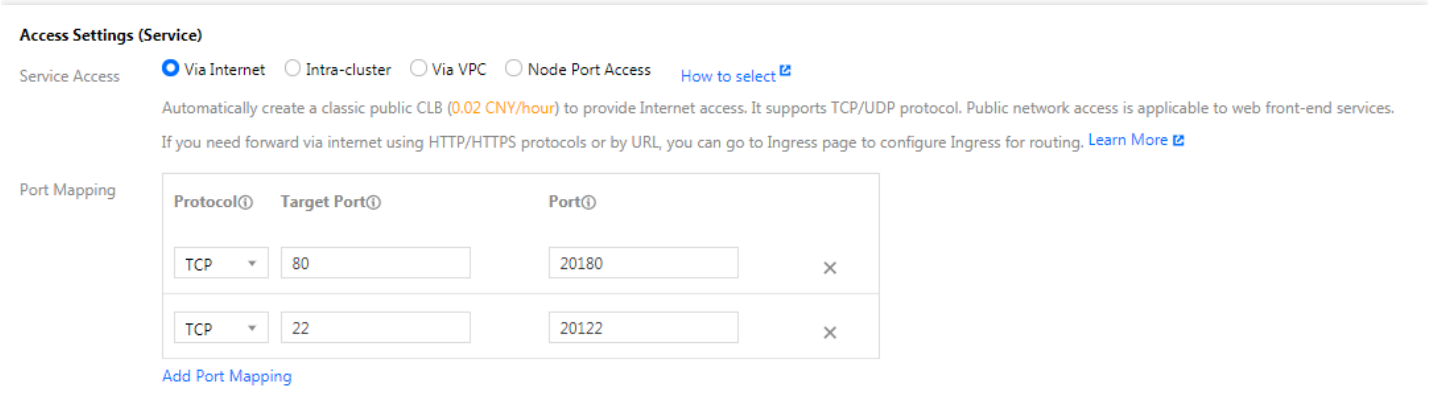
gitlab 容器的参数示例

您可以使用以下的 docker run 命令可以创建出一个简单的 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。

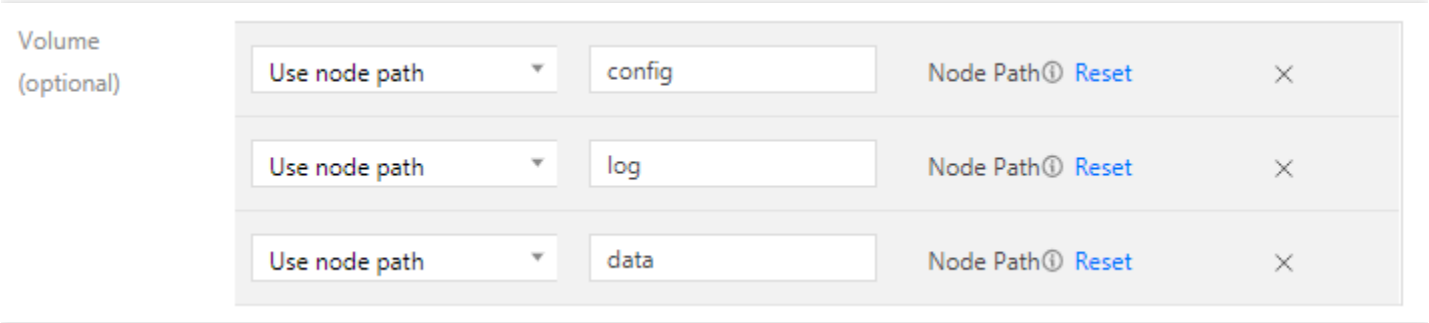
`-p` ：指定端口映射。这里映射了两个端口，容器端口分别是80和22，对外暴露的端口分别是20180和20122，对应到控制台，添加两条端口映射规则，并填写对应的容器端口和服务端口。由于 gitlab 需要提供外网访问，采用了**提供公网访问**访问方式。如下图所示：



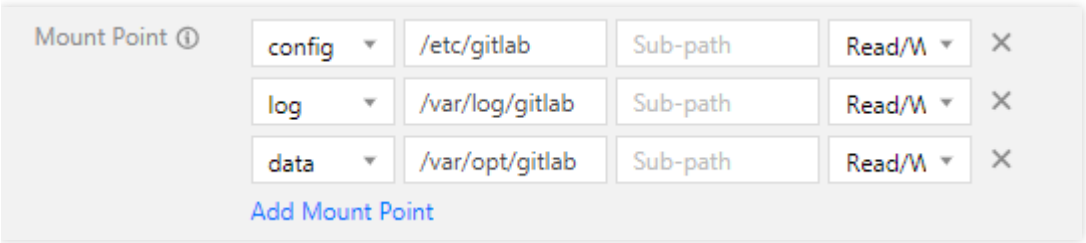
`--restart` ：本参数用于指定在容器退出时，是否重启容器。容器平台创建的所有容器退出时，都会重启容器，所以本参数不需要在容器控制台指定。

`-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 ：

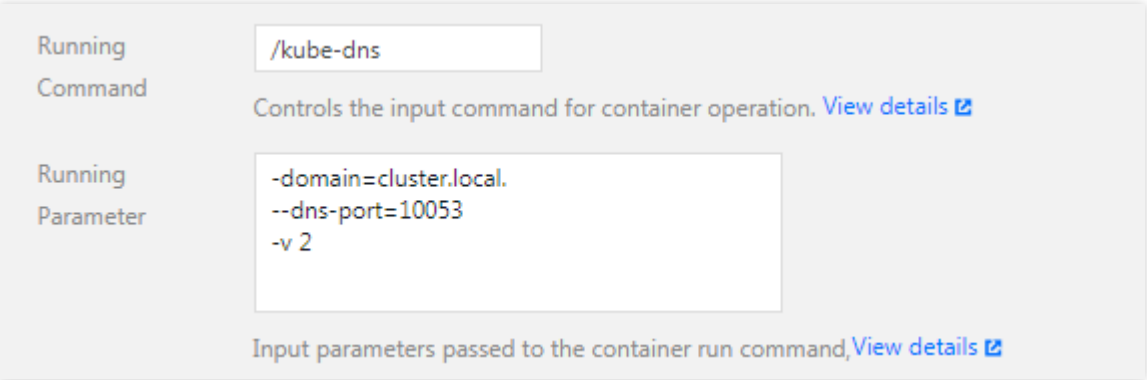
- 变量名：FOO，变量值：foo。
- 变量名：BAR，变量值：bar。

Command 和 Args

您可以在 docker run 的时候，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：



解决容器内时区不一致问题

最近更新时间：2020-03-25 17:01:29

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，请对应您实际情形结合文档解决问题：

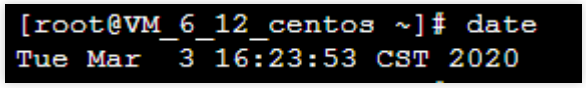
角色	地域	配置	操作系统	Kubernetes 版本信息
节点	华南地区（广州）	CPU：1核，内存:：1GB，带宽：1 Mbps 系统盘：50 GB（普通云硬盘）	CentOS Linux 7 （Core）	1.16.3

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：

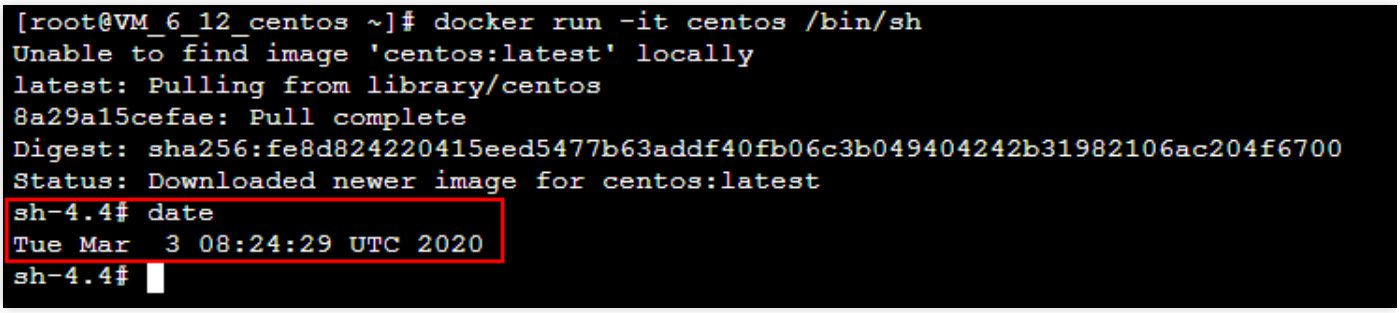


3. 依次执行以下命令，查看容器内 CentOS 系统默认时区。

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

```
date
```

返回结果如下图所示：



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
```

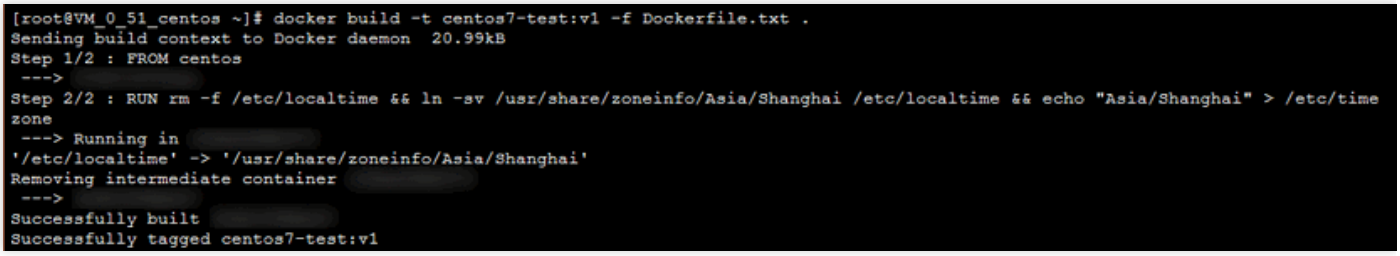
2. 按 **i** 切换至编辑模式，写入以下内容，配置时区文件。

```
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：



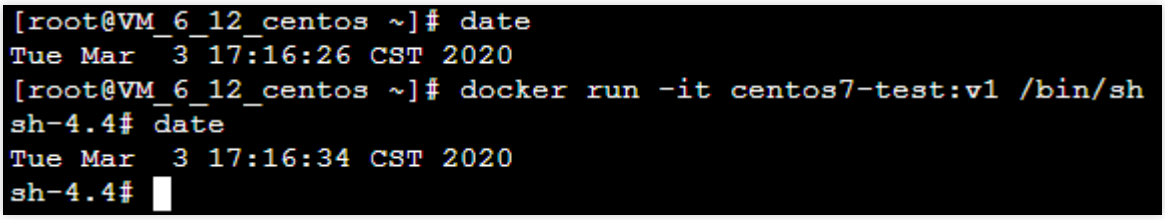
5. 依次执行以下命令，启动容器镜像并查看容器内时区。

```
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：

- 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。

- 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。

请对应实际情况，选择以下方式，进行挂载主机时间配置到容器：

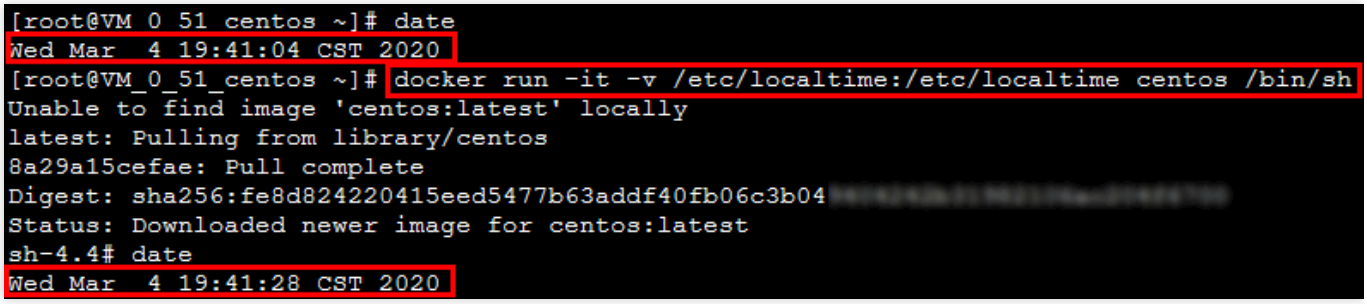
- 方式1：挂载本地 `/etc/localtime` ：
  i. 依次执行以下命令，查看本地时间并挂载本地 `/etc/localtime` 到容器内。

```
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：



ii. 执行以下命令，退出容器。

```
exit
```

- 方式2： 挂载本地 `/usr/share/zoneinfo/Asia/Shanghai` ：
  i. 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：

ii. 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：2022-01-25 10:41:47

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
```

ii. 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。

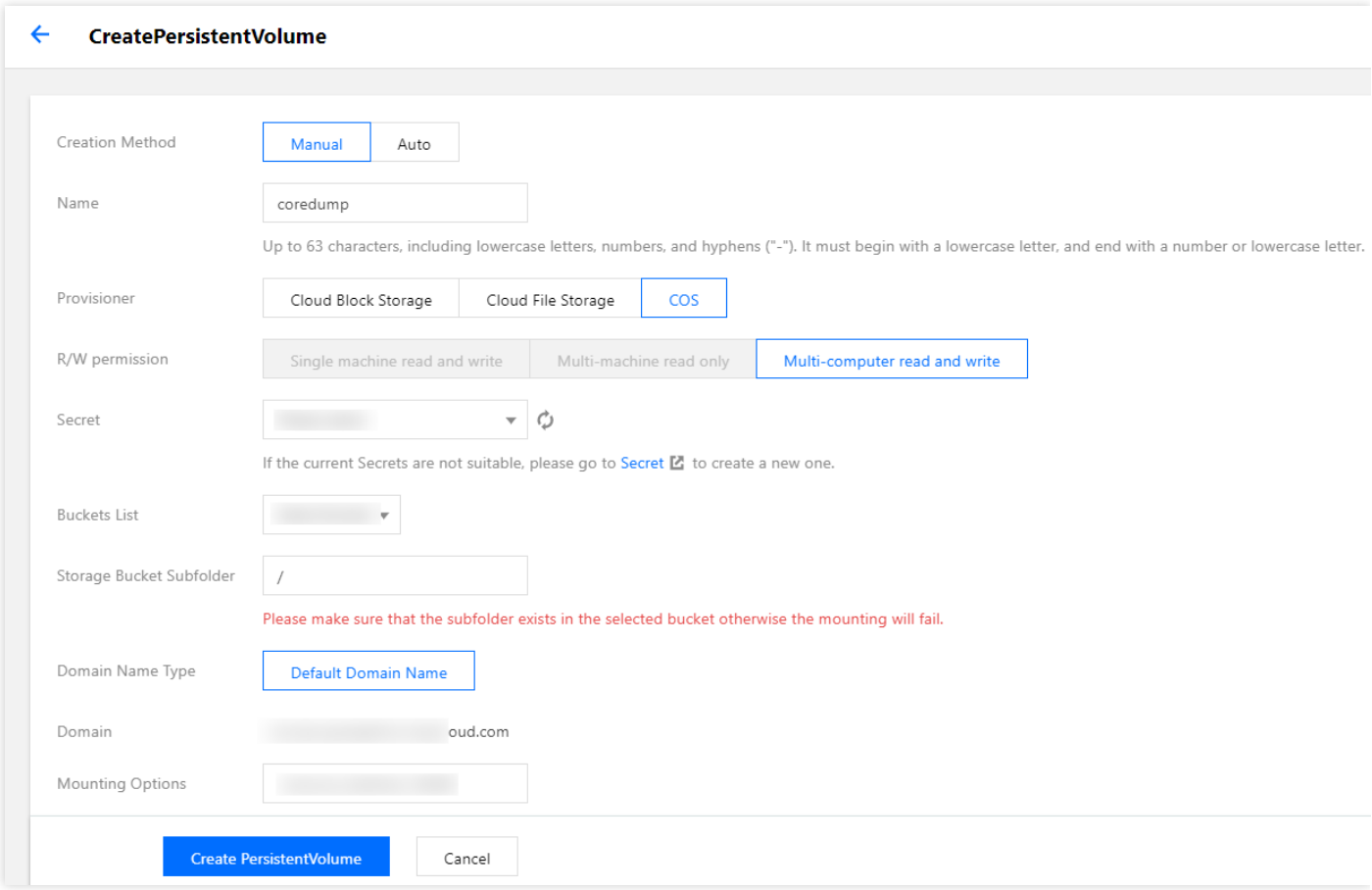
创建 PV 和 PVC

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：

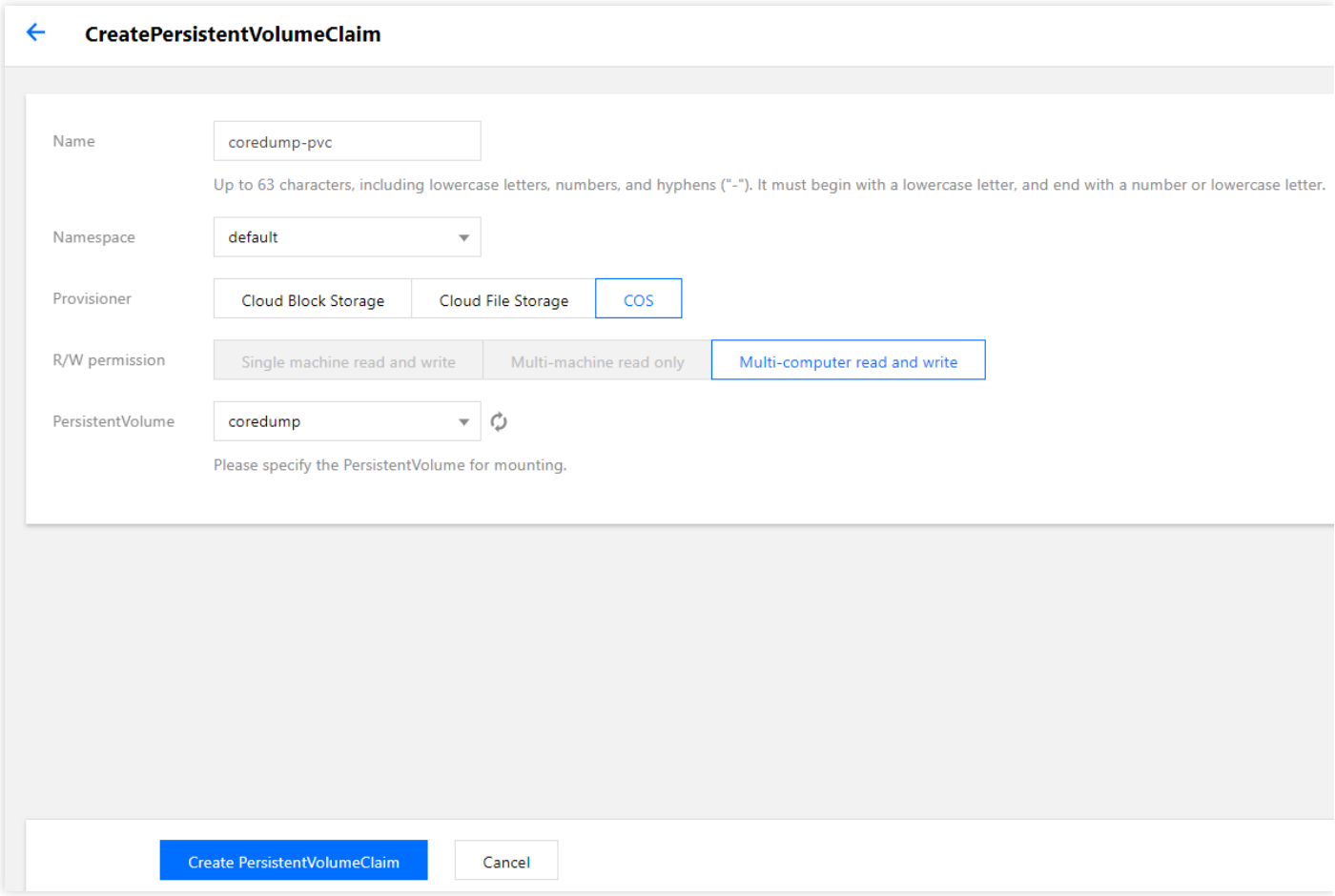
- **来源设置**：选择**静态创建**。
- **Secret**：选择已在 创建 Secret 中创建的 Secret，本文以 coredump 为例（kube-system 命名空间下）。
- **存储桶列表**：选中已创建的用于存储 coredump 文件的存储桶。
- **存储桶子目录**：此处指定根目录，如果需要指定子目录，请提前在存储桶中创建。

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。

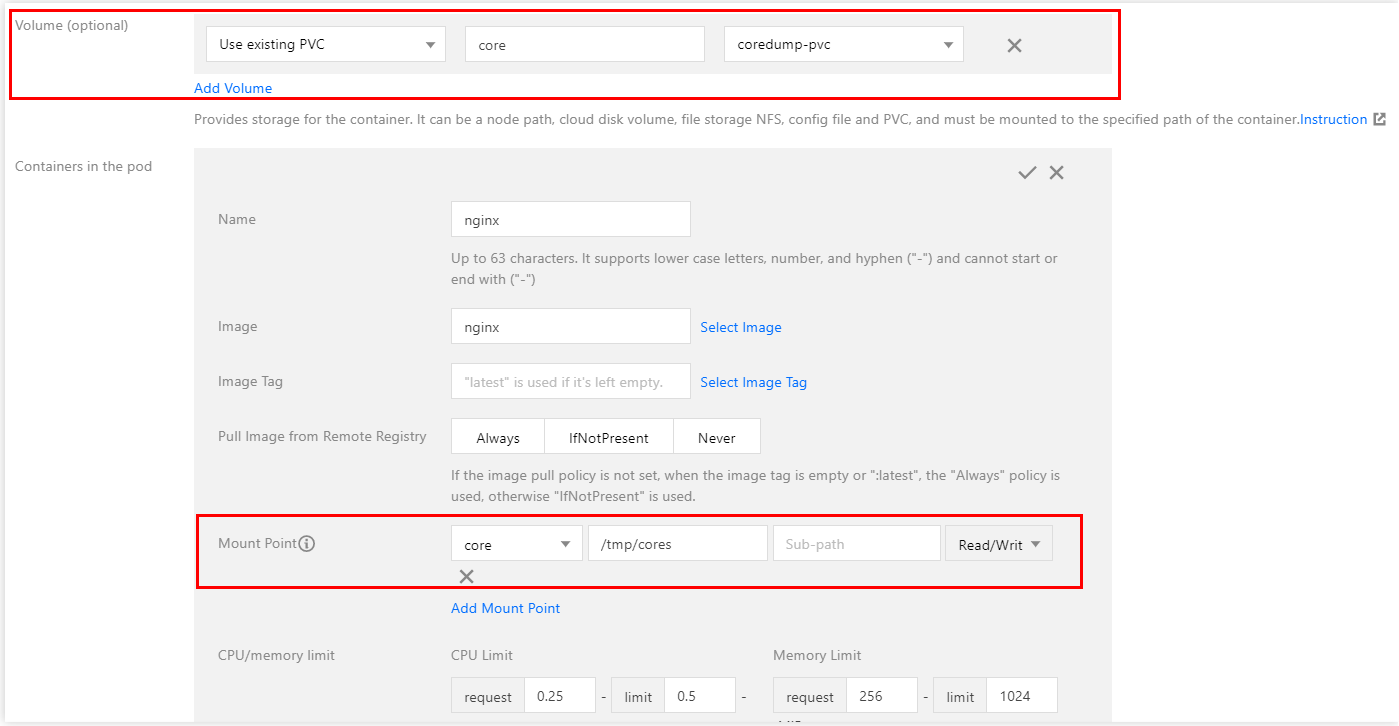
挂载 COS 存储

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

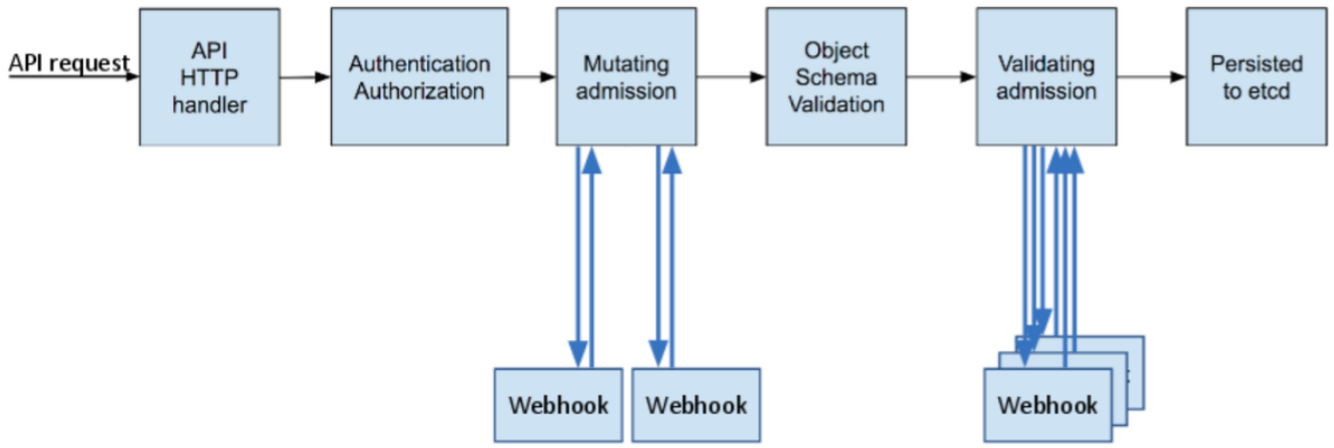
使用对象存储 COS

在 TKE 中使用动态准入控制器

最近更新时间：2022-04-21 12:23:28

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。

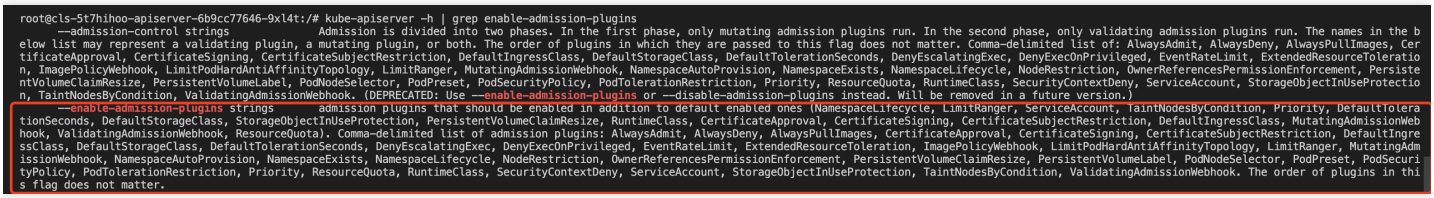
操作步骤

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" -da
ys 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：

  i. 登录容器服务控制台，选择左侧导航栏中的**集群**。
  ii. 在“集群管理”页面，选择集群 ID。
  iii. 在集群详情页面，选择左侧的**基本信息**。
  iv. 从“基本信息”页面的“集群APIServer信息”模块的 “Kubeconfig” 中的 `clusters.cluster[].certificate-authority-data` 字段进行获取，该字段已进行 `base64` 编码，无需再进行处理。

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：

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: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURFekNDQWZ1Z0F3SUJBZ0lKQU52Mkg3WlYxb1lFTUEwR0NTcUdTSWIzRFFFQkN3VUFNQ0F4SGpBY0JnTlYKQkFNTUZYZGxZbk5sY25abGNpNWtaV1poZFd4MExuTjJZekFlRncweU1ERXhNVGN3TkRVeU1URmFGdzAwT
0RBMApNRFF3TkRVeU1URmFNQ0F4SGpBY0JnTlZCQU1NRlhkbFluTmxjblpsY2k1a1pXWmhkV3gwTG5OMll6Q0NBU0l3CkRRWUpLb1pJaHZjTkFRRUJCUUFEZ2dFUEFEQ0NBUW9DZ2dFQkFNRnl5aXpjYU1uZ3EvanRTWVhwdDQwY2dzK0EKQ3pKTnl6a0VTT2l2ZEJ0alB6WHpKTEFCYlh0UkE3U
XYwWkNJaktZOTZLZ1dGUmluVFRZVjBHVVV2T3BsWEh1aApiUnBzZXRVUGFwZUp1ZDRuOFozY09xUTk2T0JRM2ZiNEt1SFdCYXI5UU1sYjFjVDZ3amk0bTZWSklUVmhiQzhjCkFJRWpKSG5uL2pyMkVwTkVIdjhkL1J4S3kyVFVrc3UvYVBhU0I2Sm84Ui9JVkFzNmNHeWwzbnFZdEJ3NFBQSDcKe
TltUXJsQko3VHNsazR2aW1xMEVOUXhDWDA2ZVBjVFlyendwTGtqZHMyWU5pSmRiM3pVVEhpZWJTUzVWWkRxWApPT0NWWUt4VnNSUkRwRWFNMDFxR0NiNWJnc2xwTXpHZk5FRUF1K2dLSVM1VGFsTkRhVVhhcWRHT0xRRUNBd0VBCkFhTlFNRTR3SFFZRFZSME9CQllFRkZiWkVzeEhaSGdseXVqa
WFPZ2w5eS9rT1Nrck1COEdBMVVkSXdRWU1CYUEKRkZiWkVzeEhaSGdseXVqaWFPZ2w5eS9rT1Nrck1Bd0dBMVVkRXdRRk1BTUJBZjh3RFFZSktvWklodmNOQVFFTApCUUFEZ2dFQkFIVzI4c01RWFNFUnZYMFJDbG1vT3oydFFhVkVhaWd2aU1SR09SZithd3dUNWpWdktNRHhvREtRCnhRUHh6e
nhhWjFNRDd4QmRUTlQ2VFArZWNTeC9qdEh2RkpPQk10NHQrdHRlb3h5T1l1YlVWZjdNTHRyb2dUYW0KeVE4b0JZN3ppOWtkaGt5cnhmanEraFdZWis5d0hxWFE2U2NFaktzeWxRMjVsdllGNG1FOVB0ZFNXcFpURWtmegpFSklFUjBPZnZxd2F4VHBISEN1NkpucjNDM2NtMDJpWkgyYStMWWVDZ
k1EOTU2YTJqbU4xWCs2cWg0WGVGaHRmCjJCZzNHVUtyVkpJOGUwc01PcUVUVzYrbEtTZ3VQS0FzWStKRXhRbmhzbWJHcjZNNk5MNjR6Zzh5dVl6aVY4UnYKRDc3M0V4a0tzVXdrMU9SQ1hySm4rWWtXWk53VlJpQT0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo="
  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}/`);
});

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

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：

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

混合云
IDC 集群添加超级节点用于弹性扩容

最近更新时间：2022-10-19 16:28:22

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。

简称	组件名称	描述
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 EKS，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:
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-xxxxx
xxx <none> <none>
nginx-deployment-77b9b9bc97-sd4z5 1/1 Running 0 27s 10.0.1.7 eklet-subnet-xxxxxxx
x <none> <none>
nginx-deployment-77b9b9bc97-z86tx 1/1 Running 0 27s 10.0.1.133 eklet-subnet-xxxxx
xxx <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 <non
e> <none>
nginx-deployment-77b9b9bc97-s9vzc 1/1 Running 0 7m38s 10.0.1.118 eklet-subnet-xxx
xxxxx <none> <none>
nginx-deployment-77b9b9bc97-sd4z5 1/1 Running 0 7m38s 10.0.1.7 eklet-subnet-xxxxx
xxx <none> <none>
```

网络
DNS 相关
TKE DNS 最佳实践

最近更新时间：2023-05-06 17:36:46

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。

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
v1.22	v1.8.4
v1.20	v1.8.4
v1.18	v1.7.0
v1.16	v1.6.2
v1.14	v1.6.2

由于历史原因，可能会有 v1.18 及以上版本的集群仍然部署 v1.6.2 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)

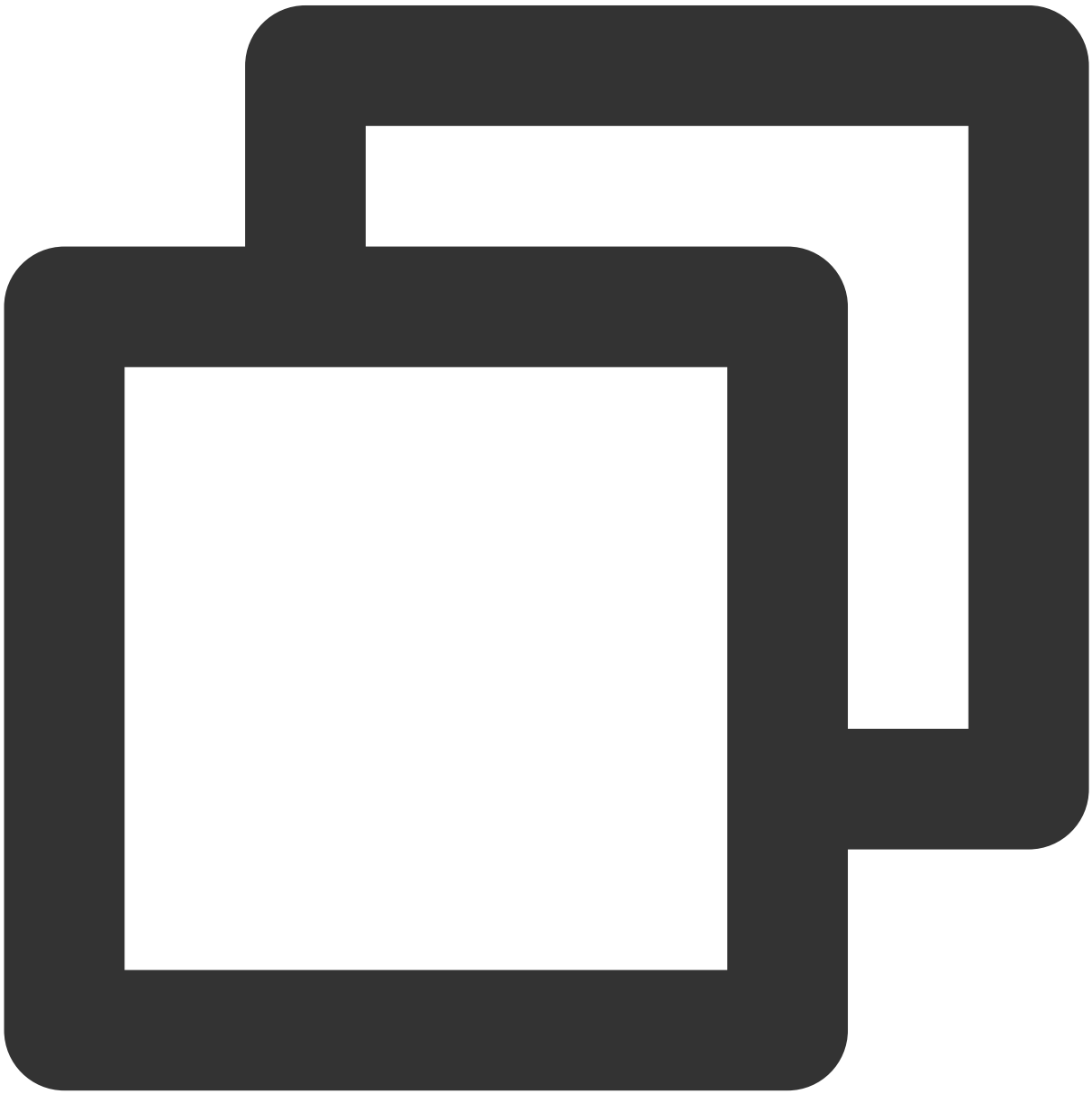
示例：

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，参见：在 TKE 集群中使用 NodeLocal DNS Cache

配置 CoreDNS 平滑升级

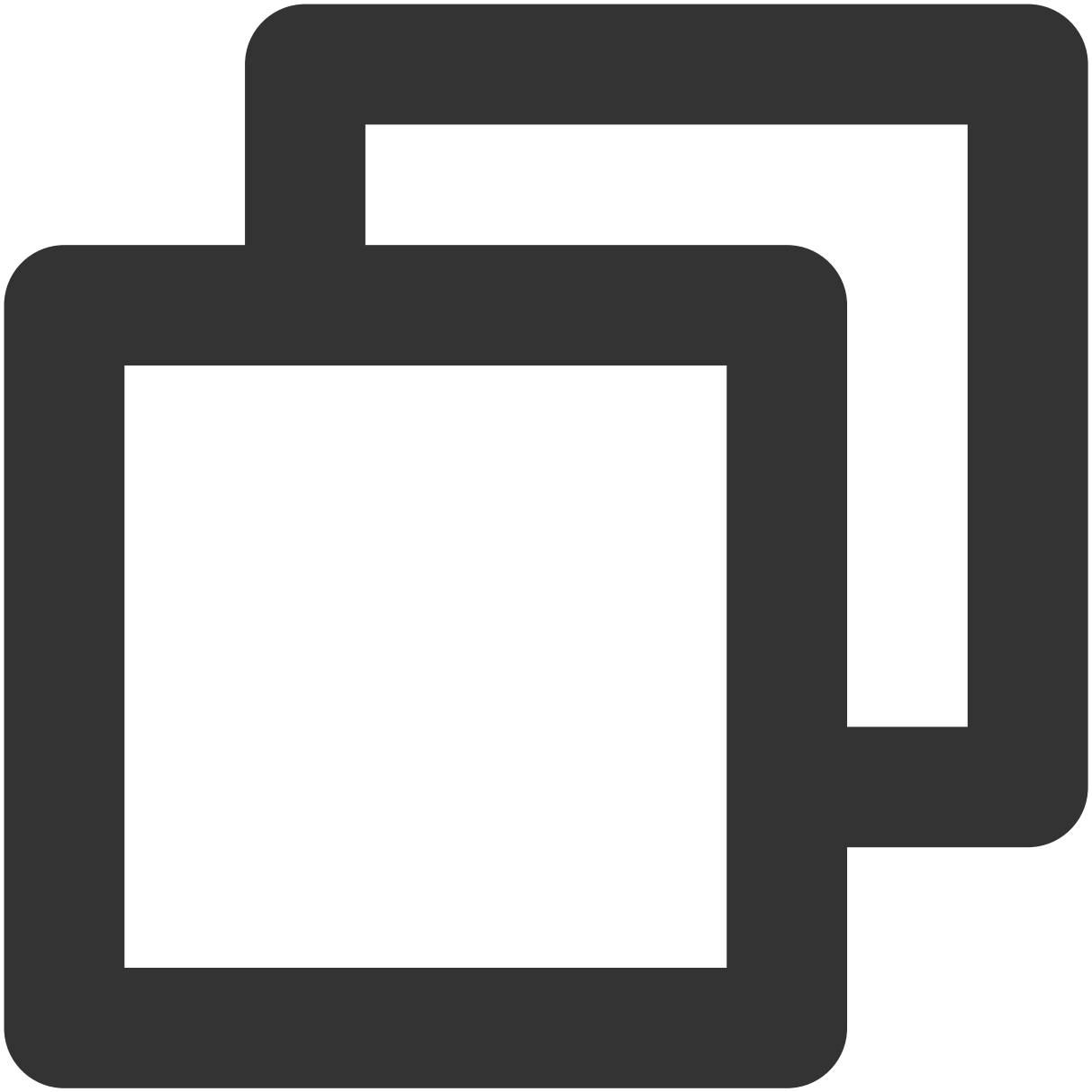
当重启节点或者升级 CoreDNS 时，可能导致 CoreDNS 部分副本在一段时间不可用，可以通过以下配置，最大程度保证 DNS 服务的可用性，实现平滑升级。

kube-proxy 为 iptables 模式，无需配置

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，配置 IPVS UDP 协议的会话保持超时时间

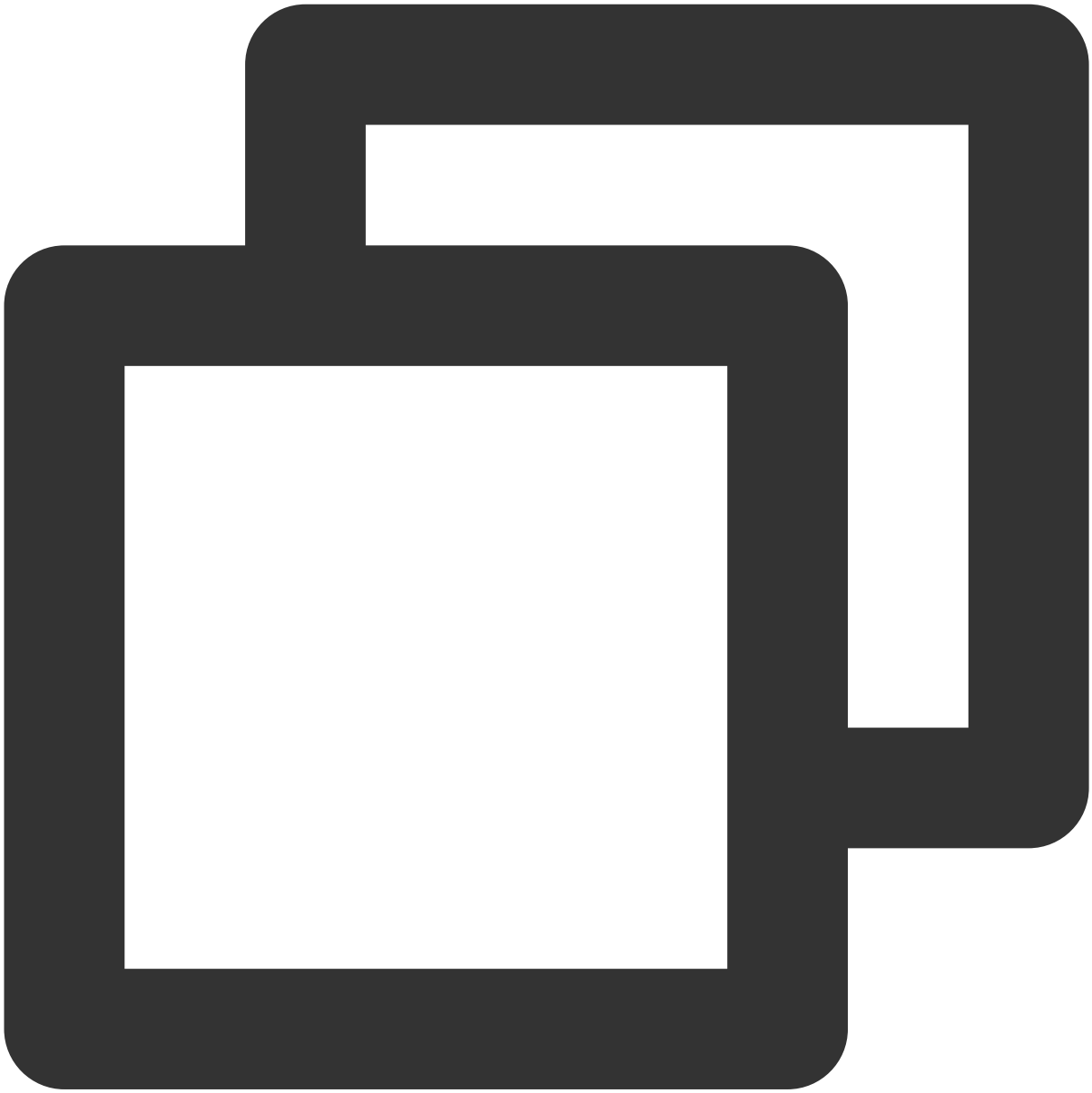
IPVS 模式下，如果业务自身没有 UDP 服务，可以通过降低 IPVS UDP 协议的会话保持超时时间来尽量减少服务不可用的时间。
1. 集群版本大于等于1.18，kube-proxy 提供参数 `--ipvs-udp-timeout` ，默认为0s，也即使用系统默认值：300s，推荐配置为 `--ipvs-udp-timeout=10s` 。按如下方式配置 kube-proxy DaemonSet：



```
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:
```

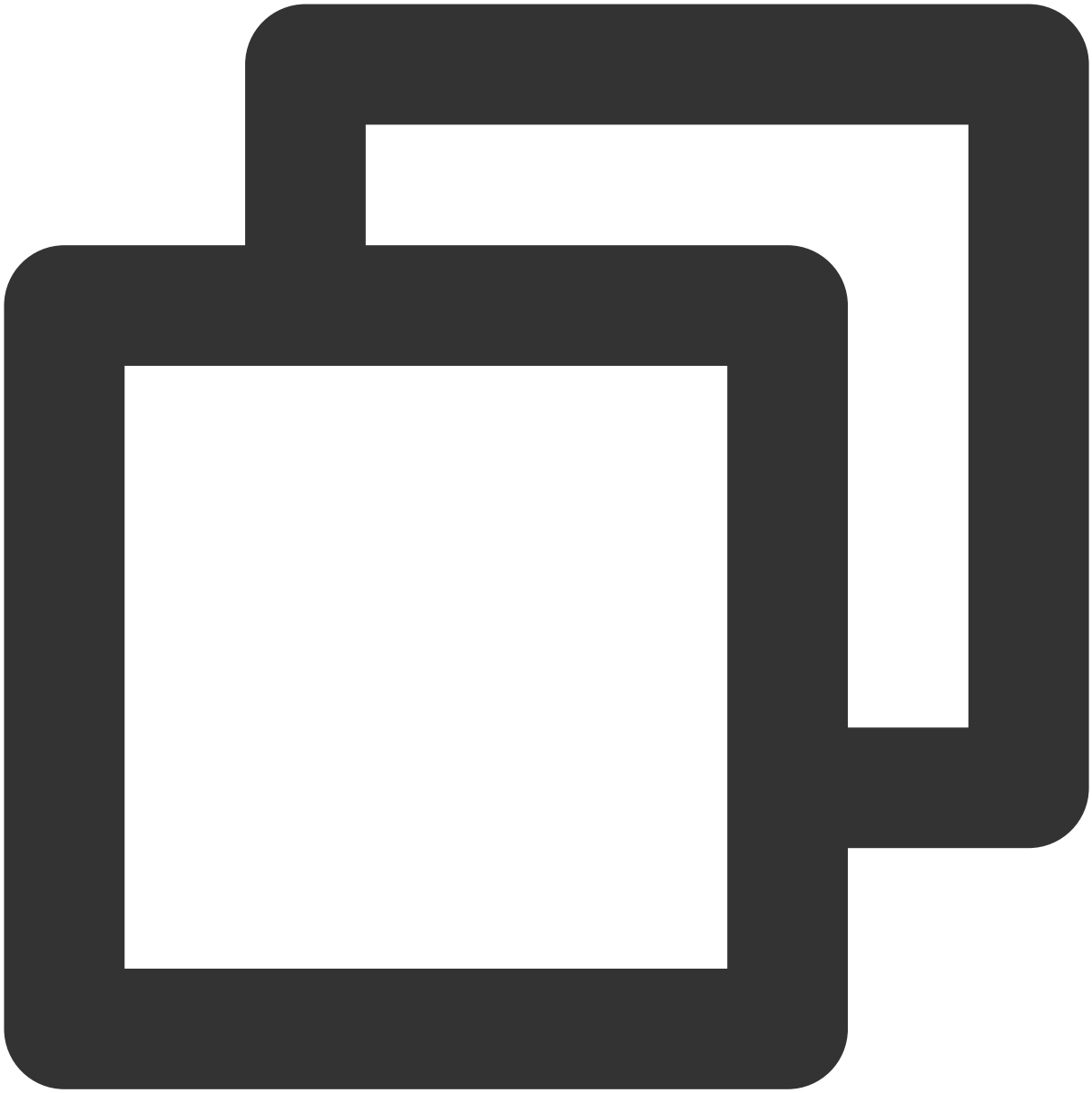
```
      - kube-proxy
      name: kube-proxy
```

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：



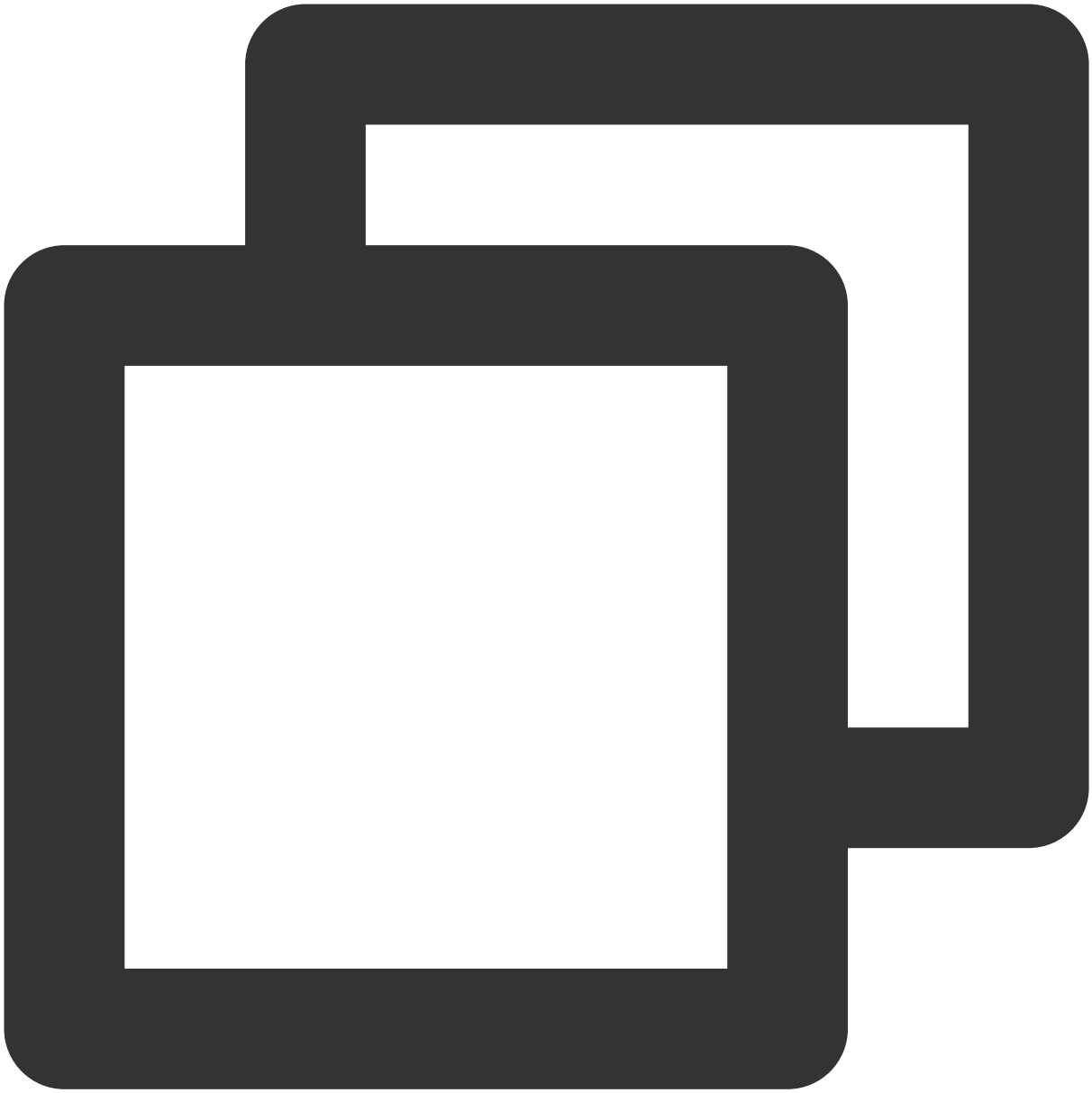
```
ipvsadm -L --timeout
Timeout (tcp tcpfin udp): 900 120 10
```

注意

在配置完成后，需要等待5min，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）：

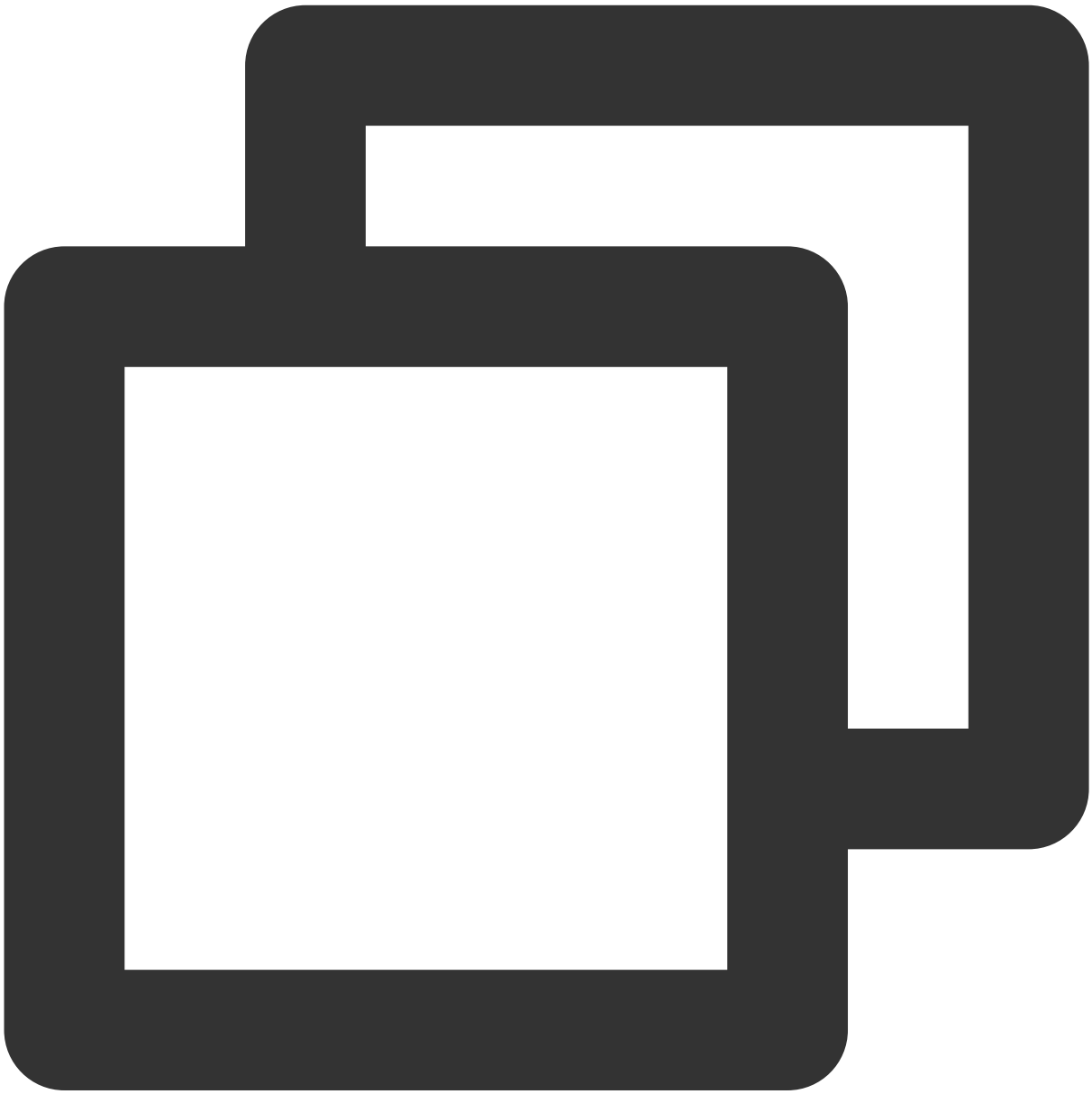


```
.:53 {
    health {
        lameduck 30s
    }
    kubernetes cluster.local. in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
    }
}
```

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。

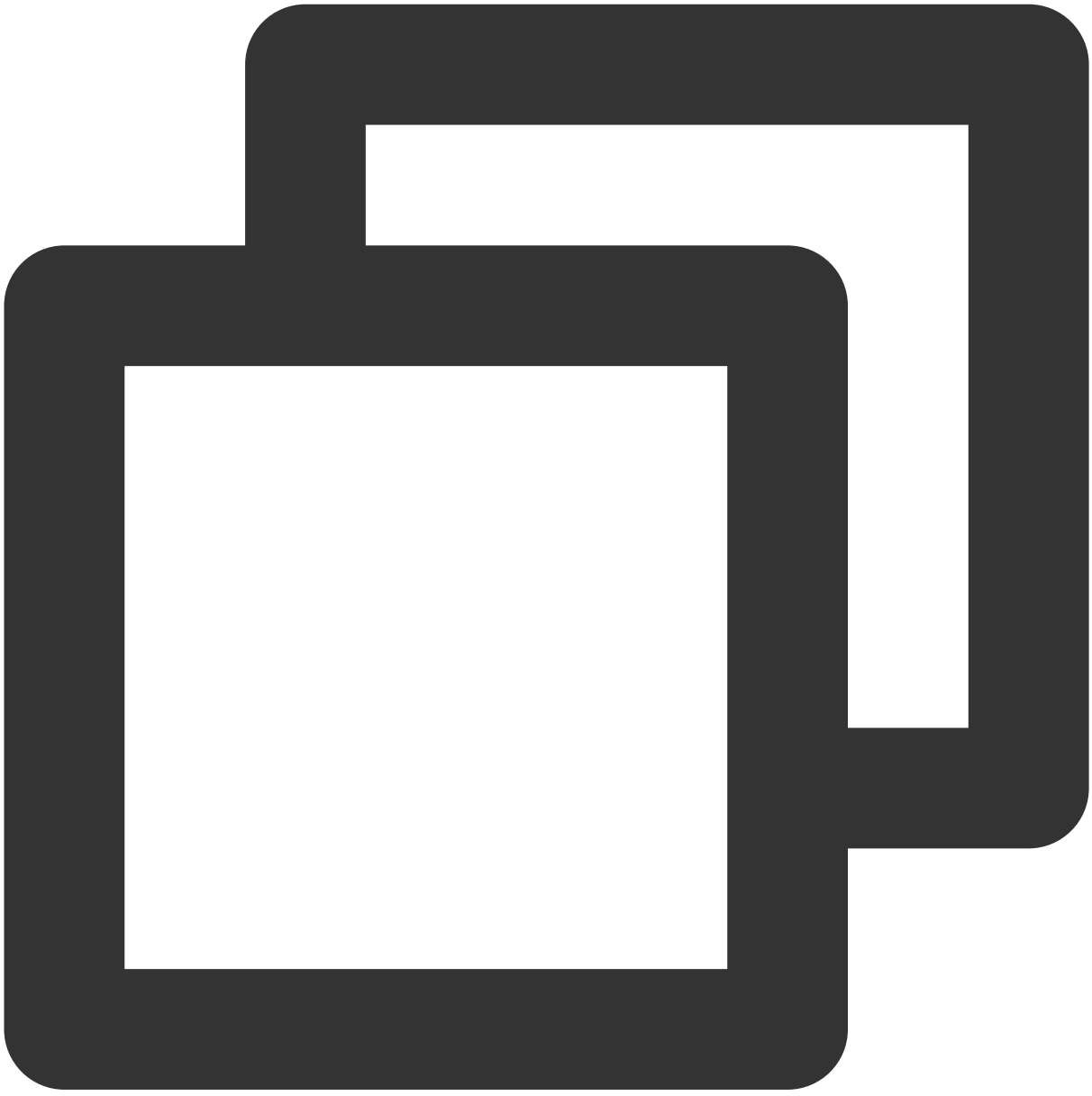
1. 打开 ready 插件，按如下方式配置 CoreDNS 的 configmap（仅展示 CoreDNS 1.6.2版本的部分配置，其它版本配置参见 手动升级 CoreDNS）：



```
        .: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：

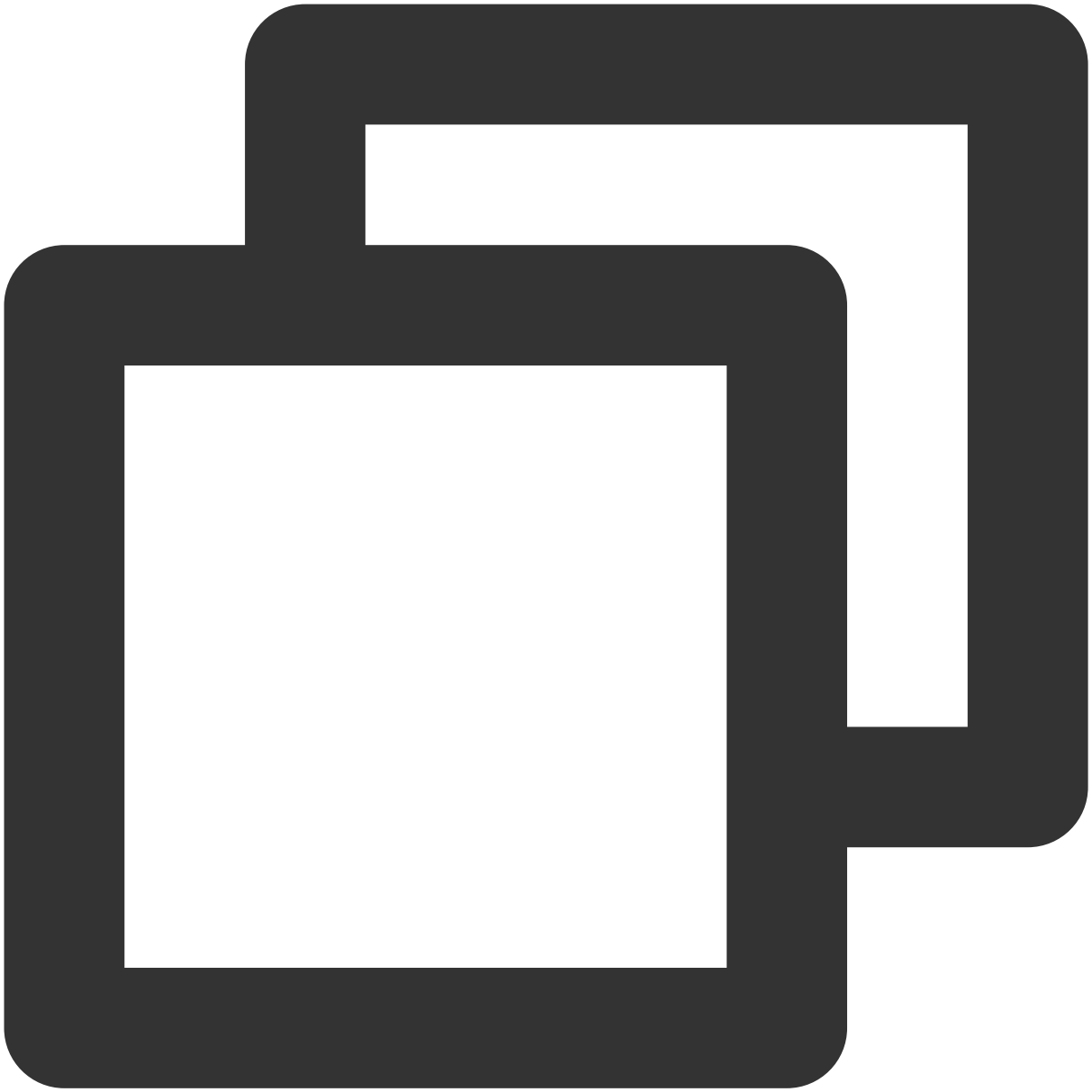


```
readinessProbe:
  failureThreshold: 5
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
```

```
    initialDelaySeconds: 30
    periodSeconds: 10
    successThreshold: 1
    timeoutSeconds: 5
```

配置 CoreDNS 使用 UDP 访问上游 DNS

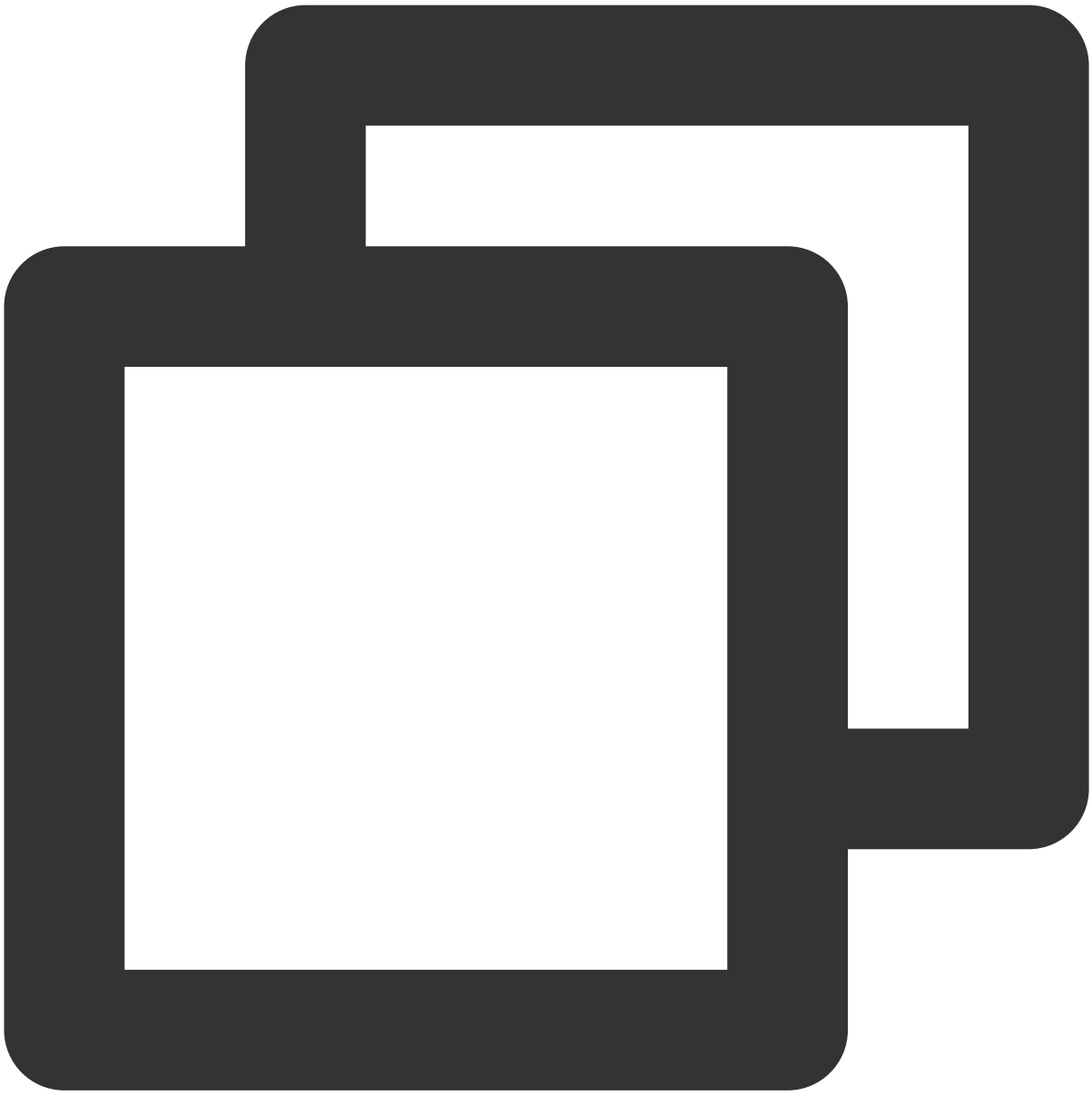
当 CoreDNS 需要与上游 DNS Server 通信时，它将默认使用客户端请求的协议（UDP 或者 TCP），而 TKE 中 CoreDNS 的上游默认为 VPC 内的 DNS 服务，该服务对 TCP 的支持在性能上比较有限，因此推荐做如下配置，显示指定 UDP（尤其在安装了 NodeLocal DNSCache 时）：



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

配置 CoreDNS 过滤 HINFO 请求

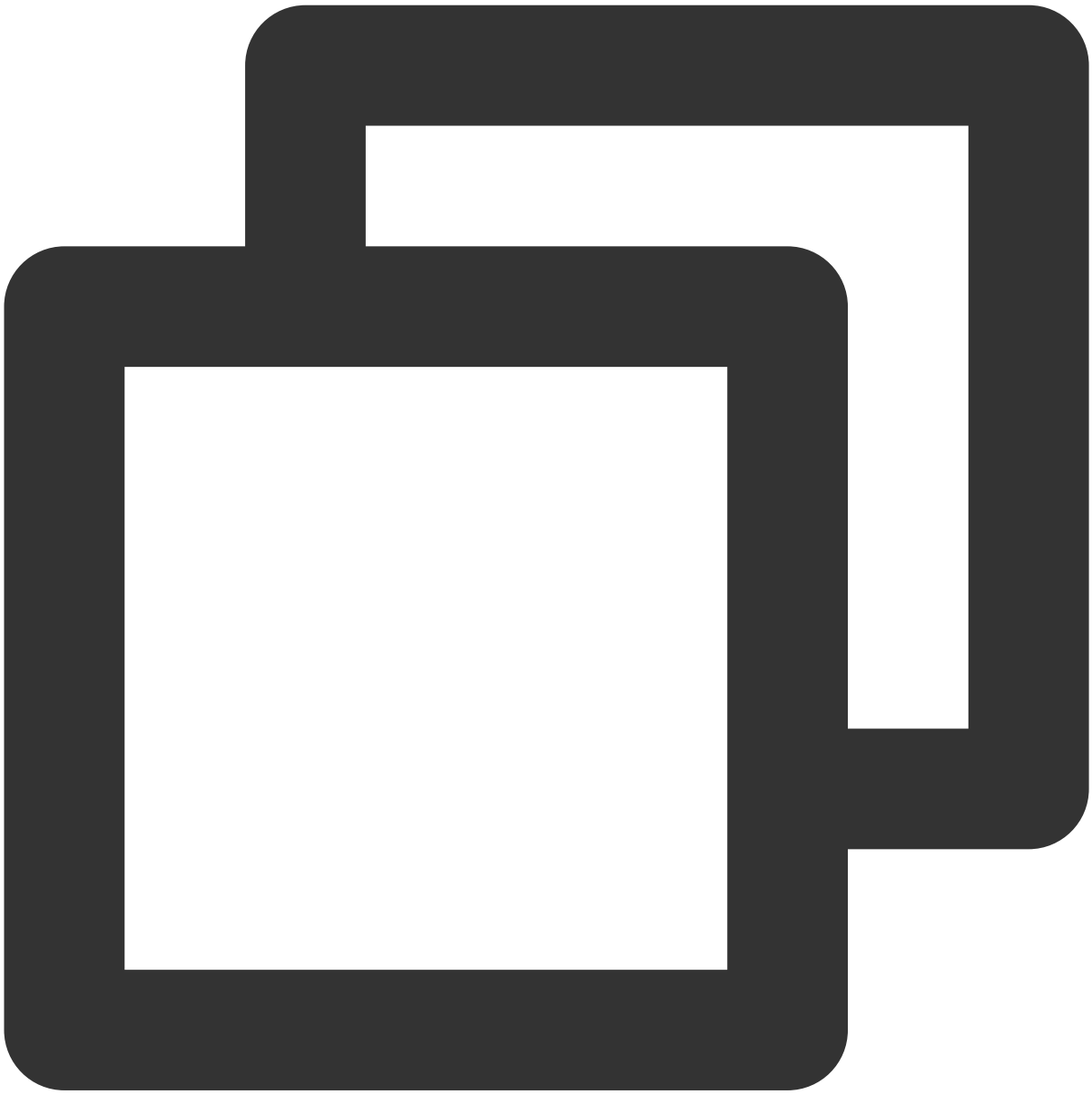
VPC 内的 DNS 服务不支持 HINFO 类型的 DNS 请求，因此推荐做如下配置，在 CoreDNS 侧过滤此类请求（尤其在安装了 NodeLocal DNSCache 时）：



```
.: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。

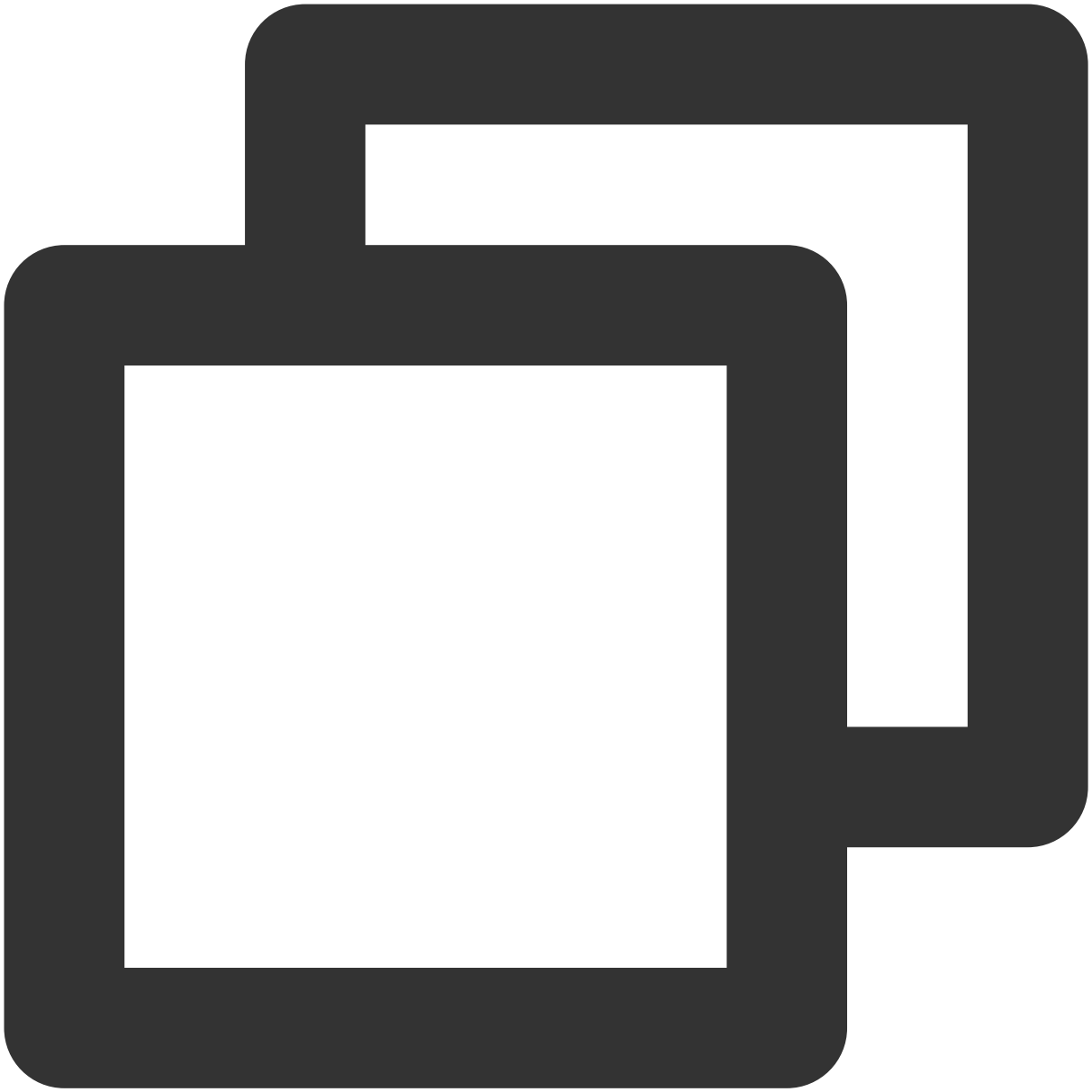
配置自定义域名解析

详情参见：在 TKE 中实现自定义域名解析

手动升级

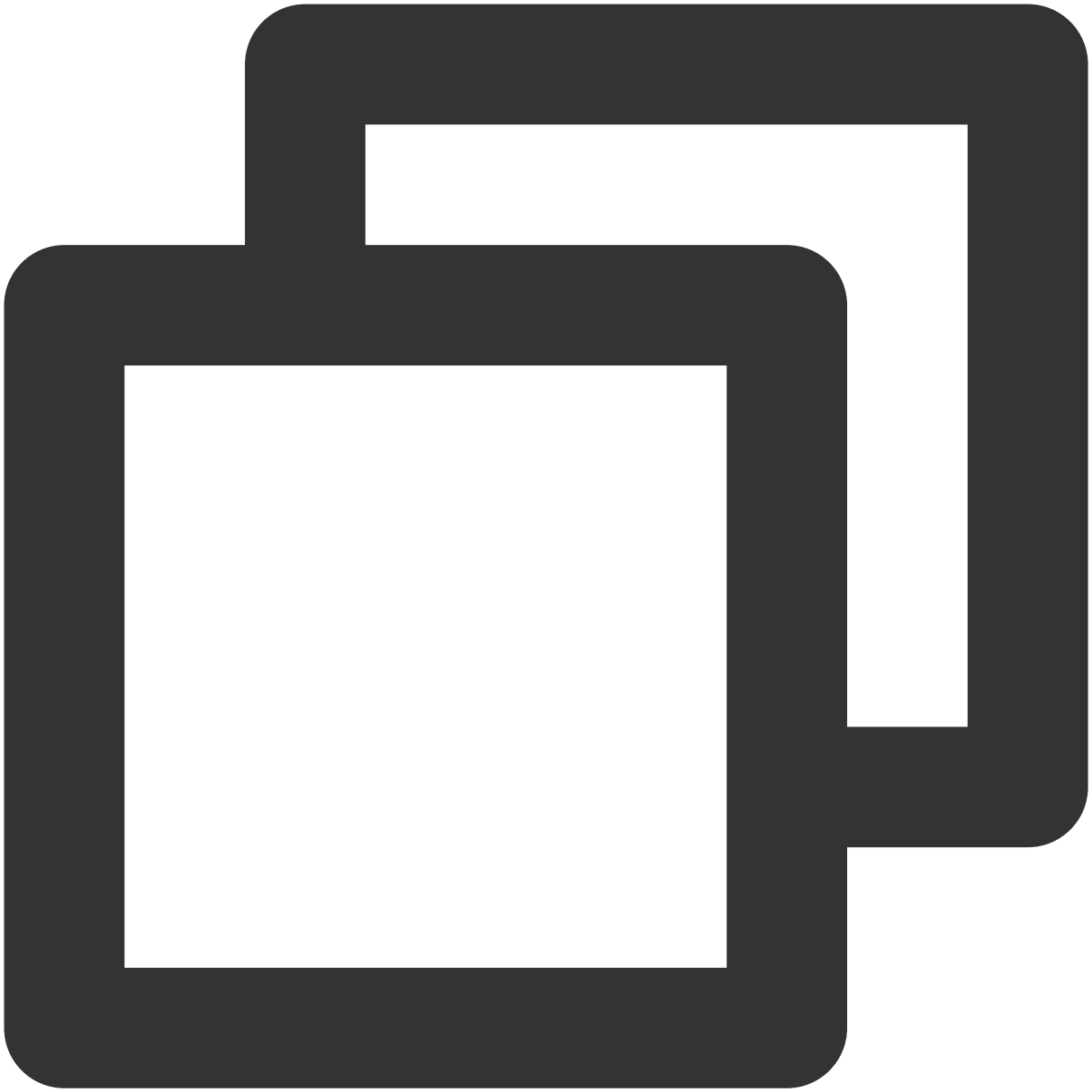
升级到1.7.0

1. 编辑 coredns configmap



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

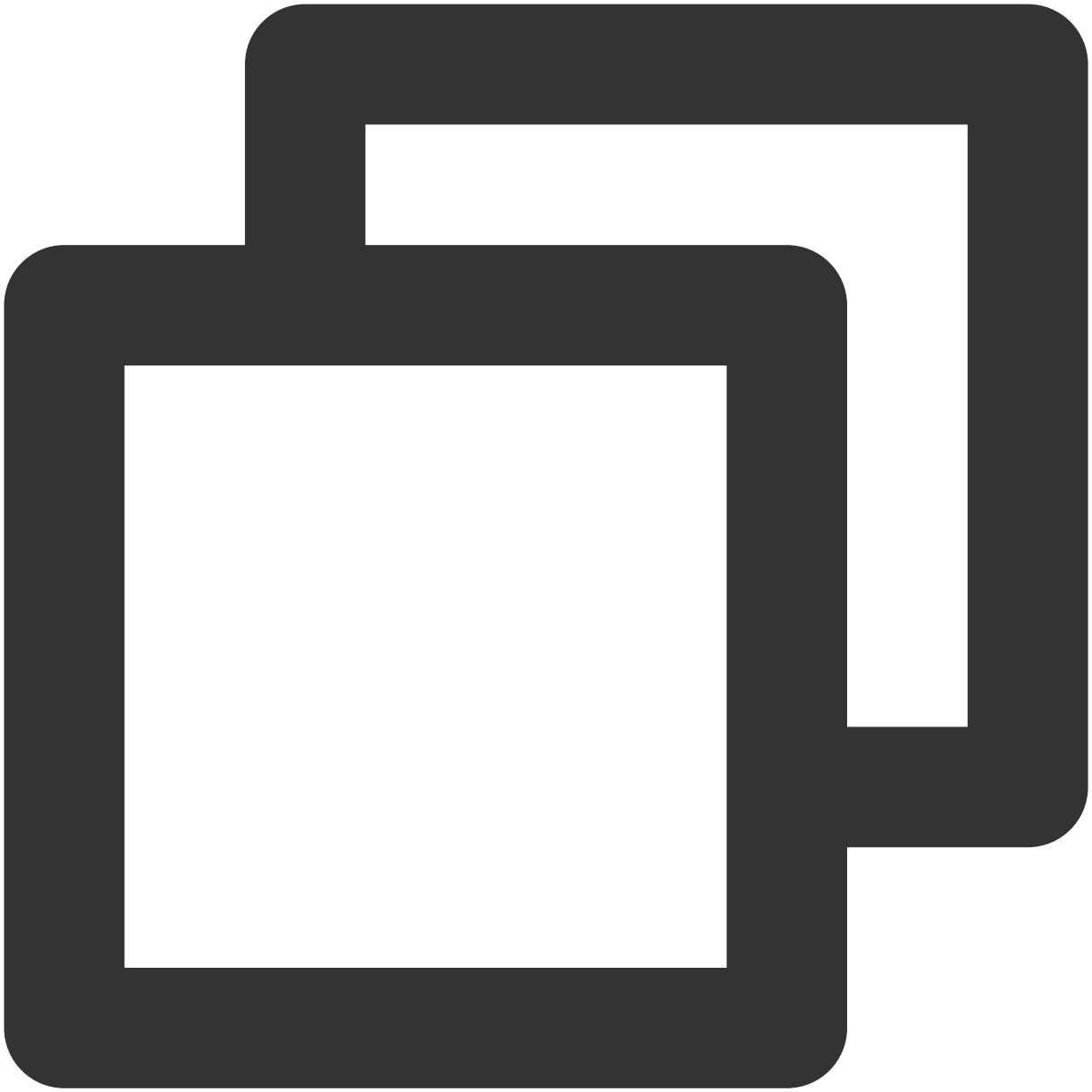
修改为以下内容：



```
.:53 {
    template ANY HINFO . {
        rcode NXDOMAIN
    }
    errors
    health {
        lameduck 30s
    }
    ready
    kubernetes cluster.local. in-addr.arpa ip6.arpa {
        pods insecure
```

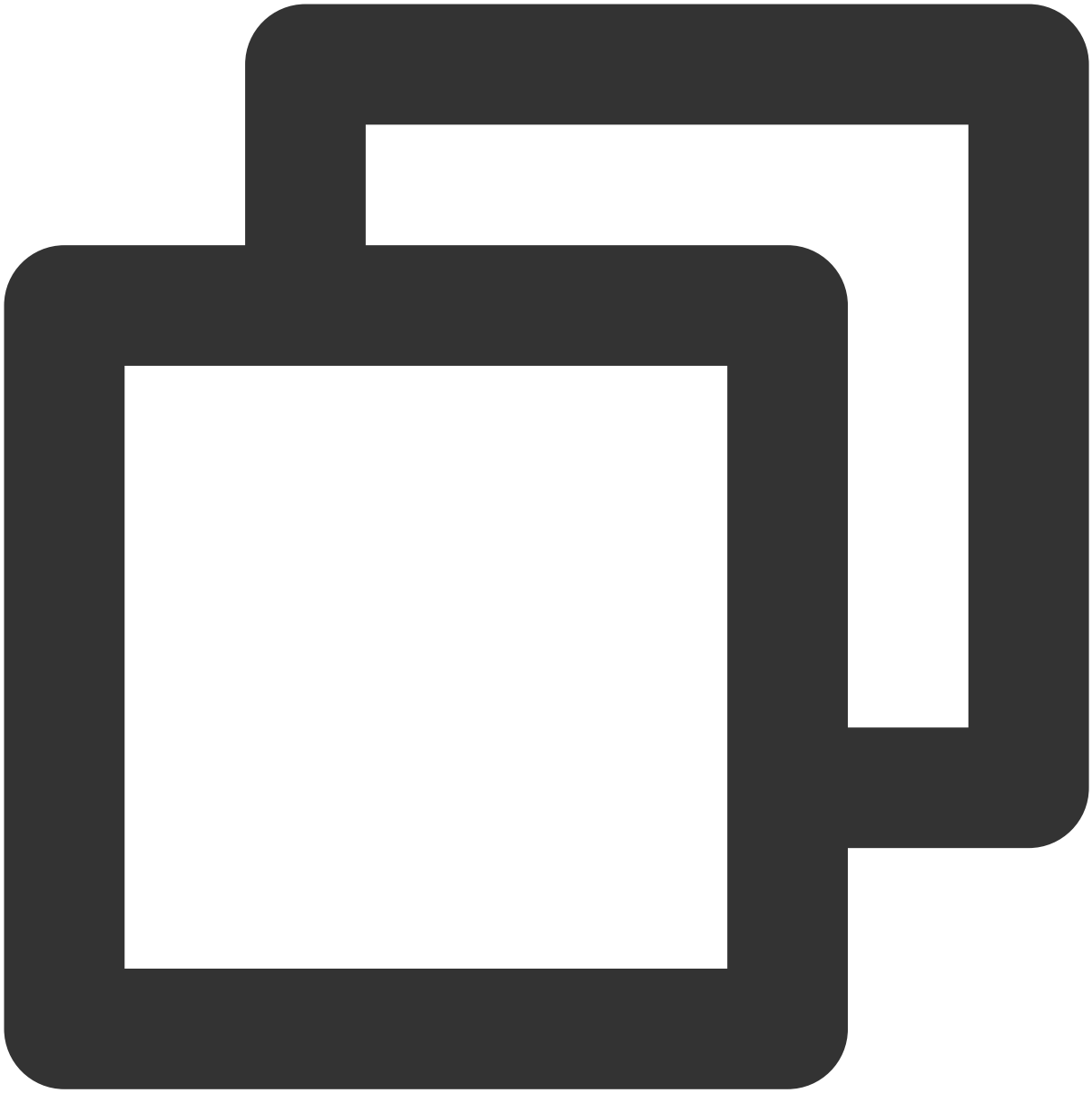
```
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            prefer_udp
        }
        cache 30
        reload
        loadbalance
    }
```

2. 编辑 coredns deployment



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

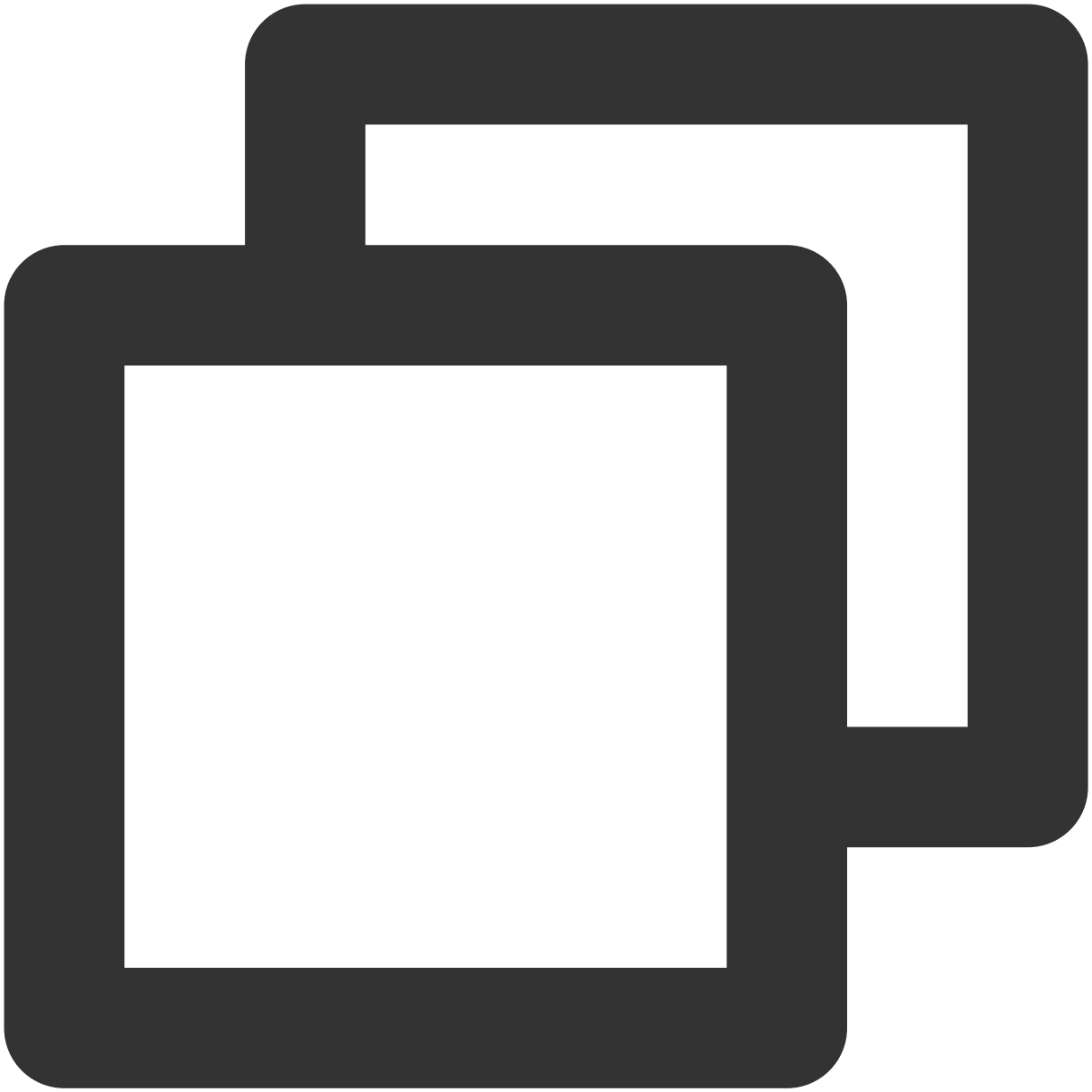
替换镜像为



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

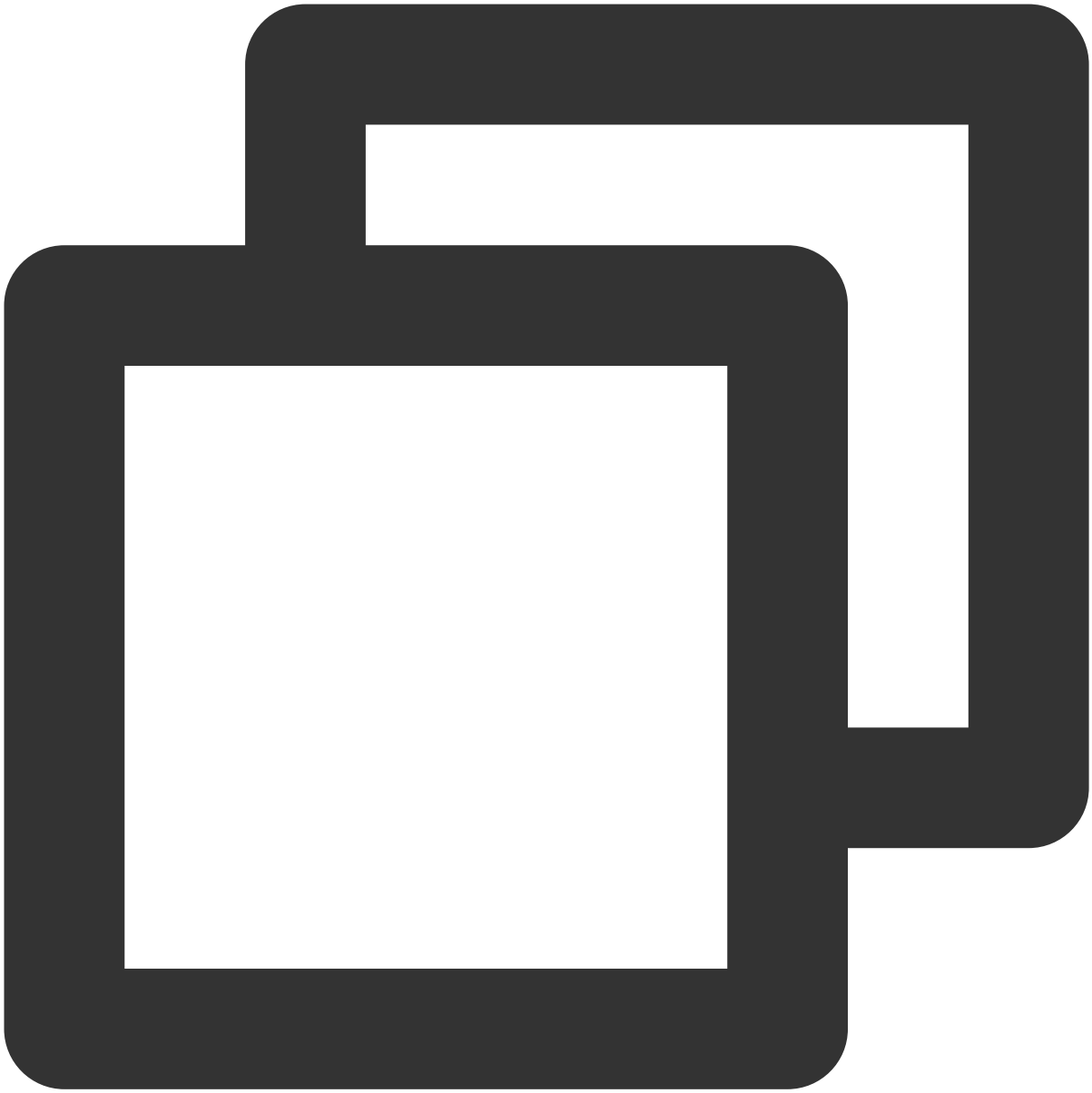
升级到1.8.4

1. 编辑 coredns clusterrole



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

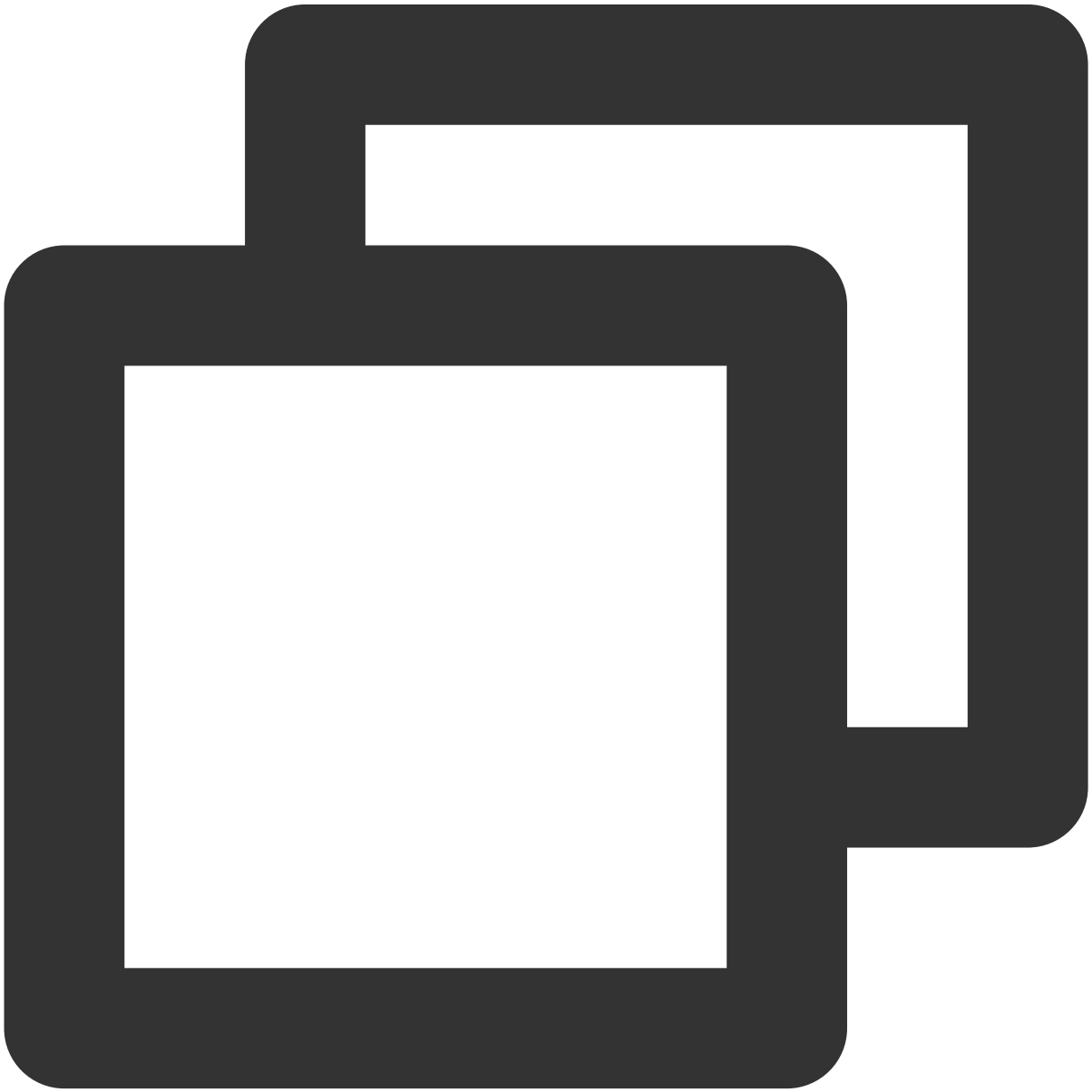
修改为以下内容：



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

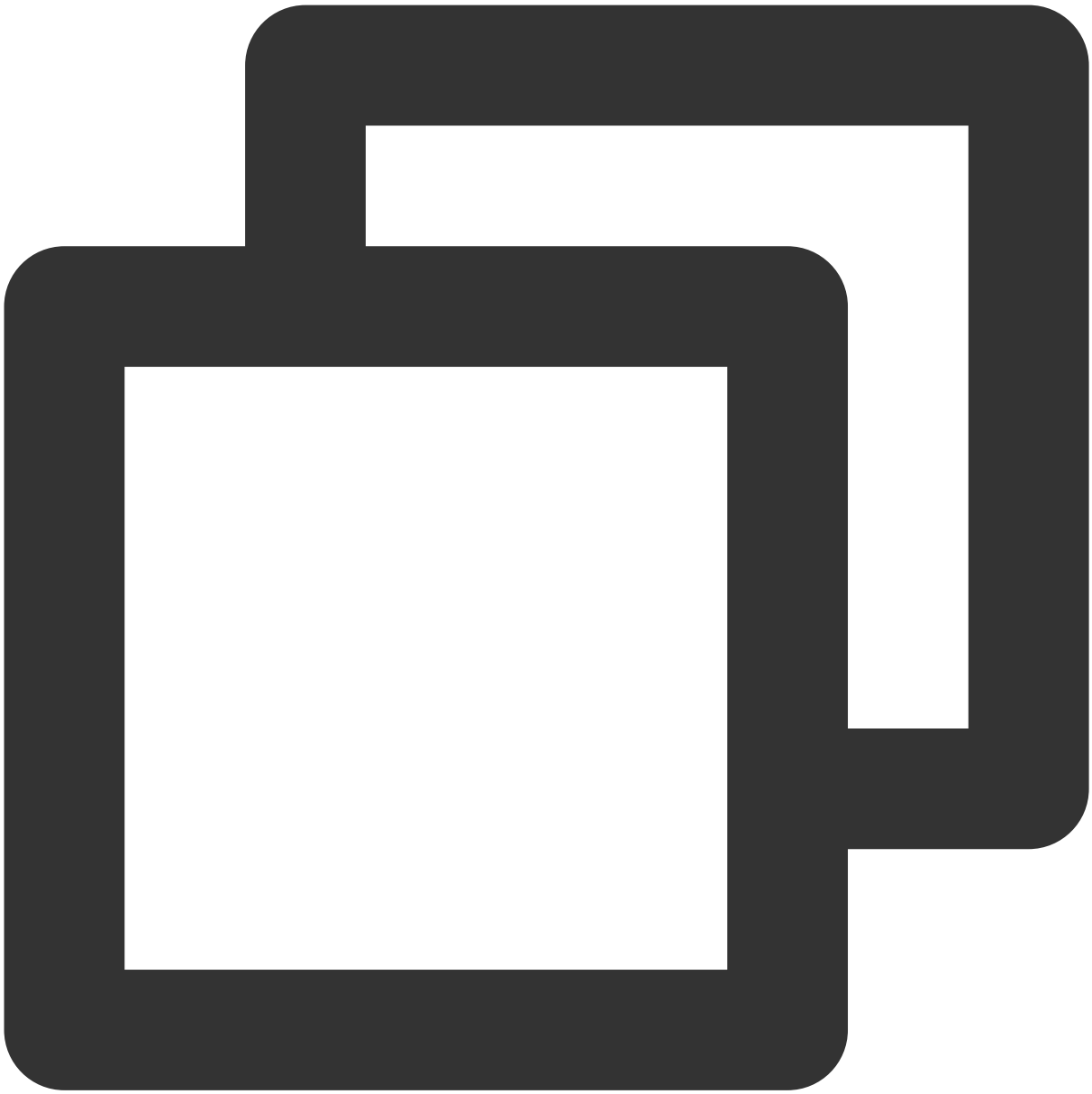
```
- apiGroups:
  - discovery.k8s.io
  resources:
  - endpointslices
  verbs:
  - list
  - watch
```

2. 编辑 coredns configmap



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

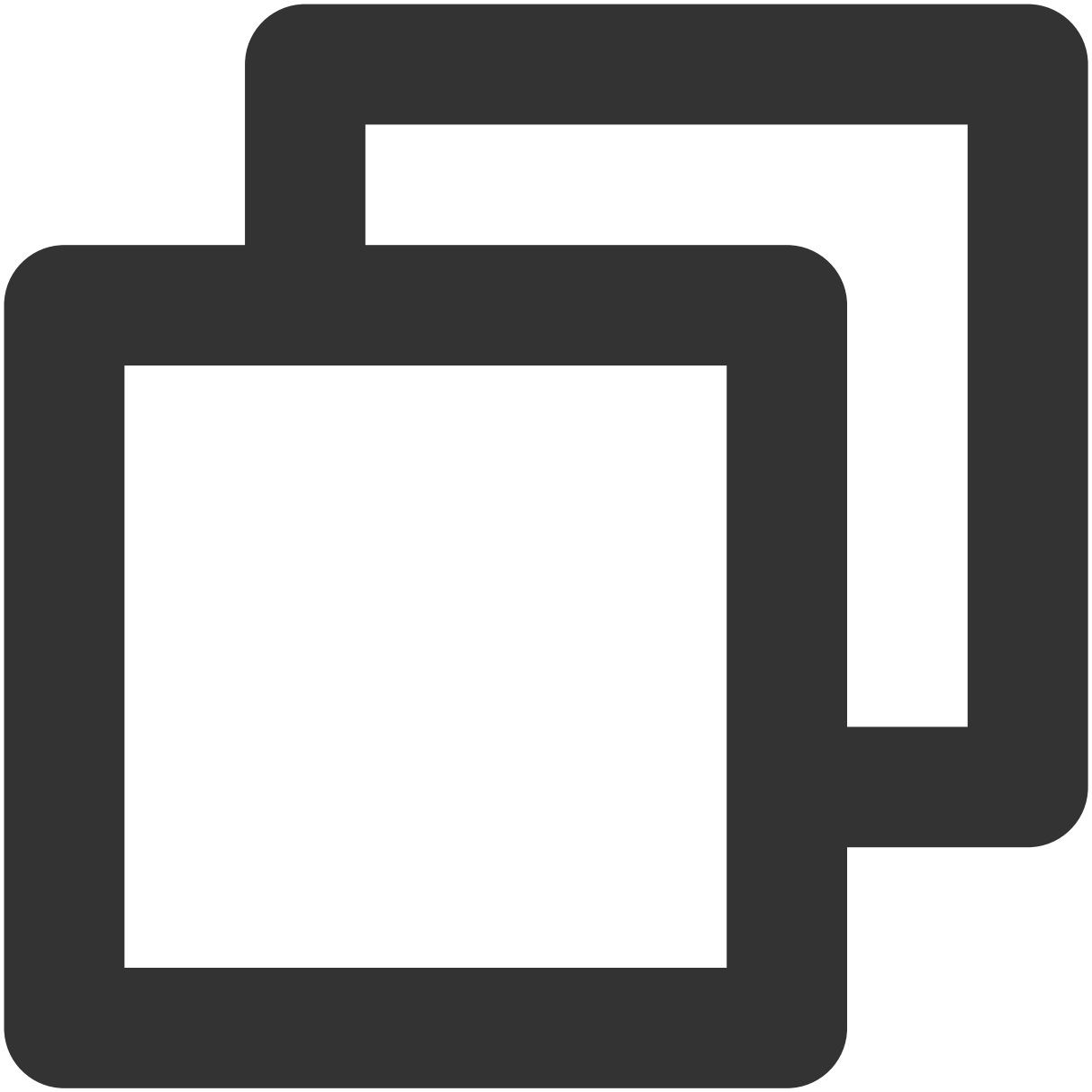
修改为以下内容：



```
.:53 {
    template ANY HINFO . {
        rcode NXDOMAIN
    }
    errors
    health {
        lameduck 30s
    }
    ready
```

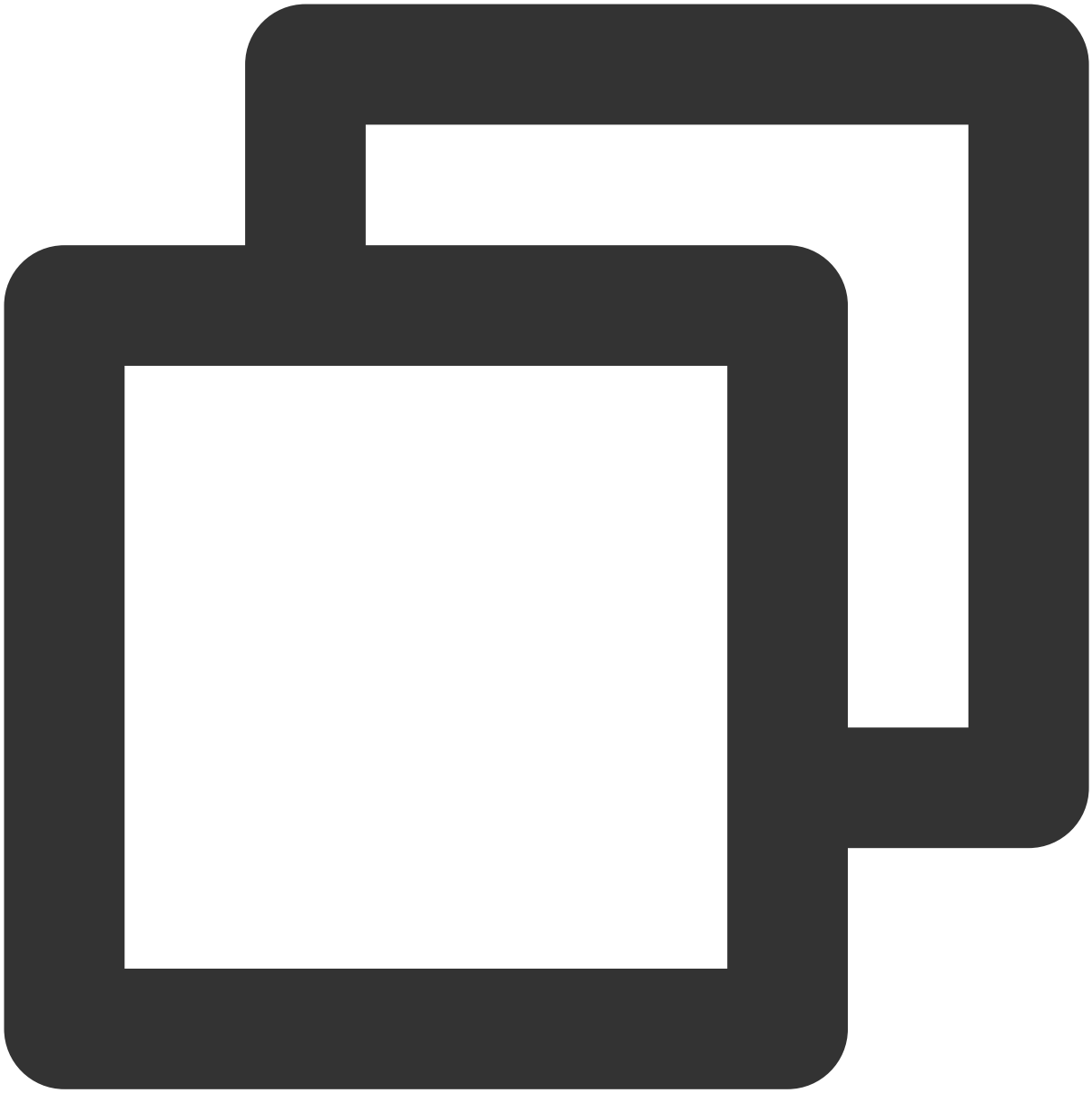
```
        kubernetes cluster.local. in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            prefer_udp
        }
        cache 30
        reload
        loadbalance
    }
```

3. 编辑 coredns deployment



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

替换镜像为



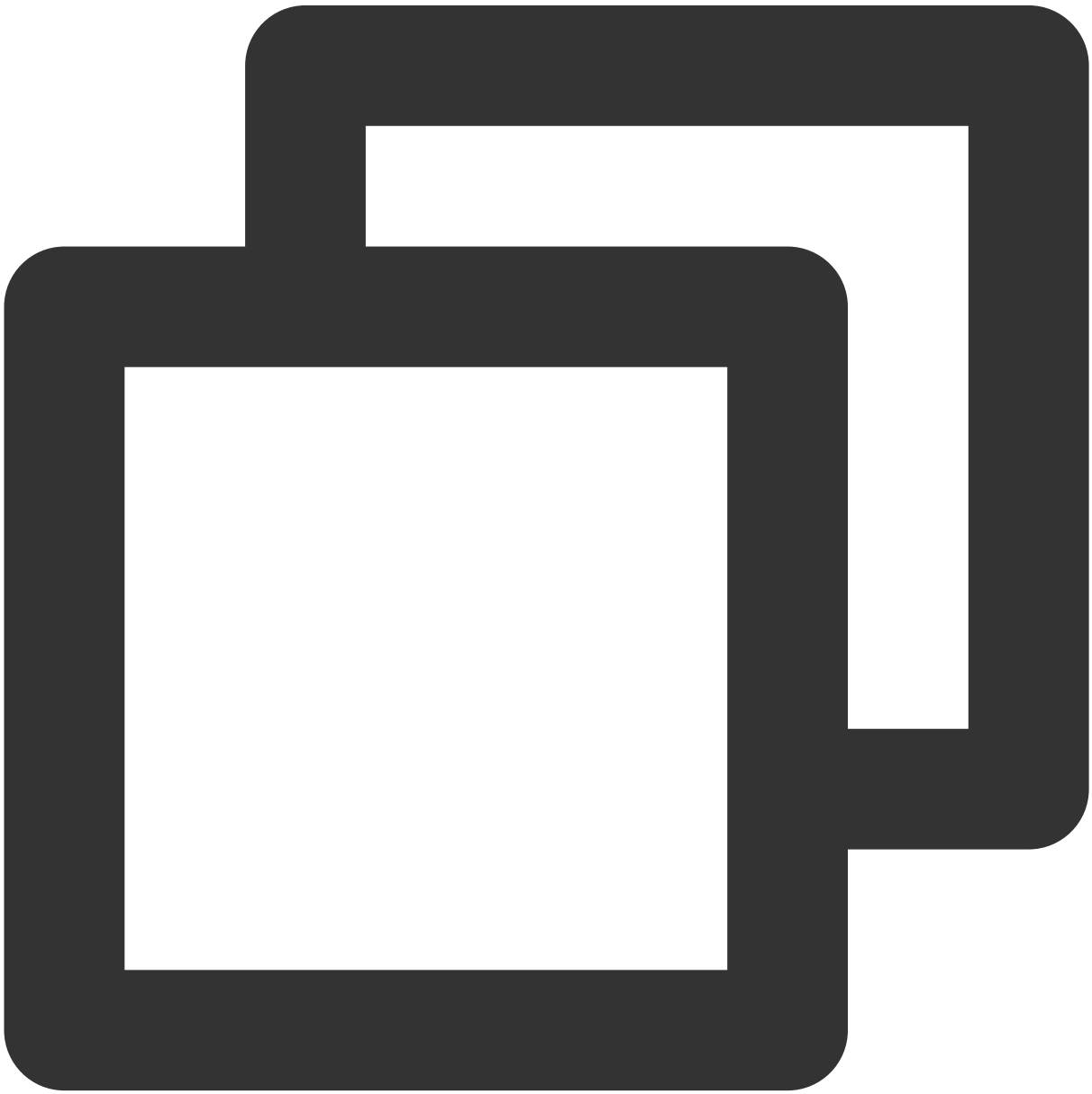
```
image: ccr.ccs.tencentyun.com/tkeimages/coredns:1.8.4
```

配置业务建议

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。

相关内容

配置介绍

errors

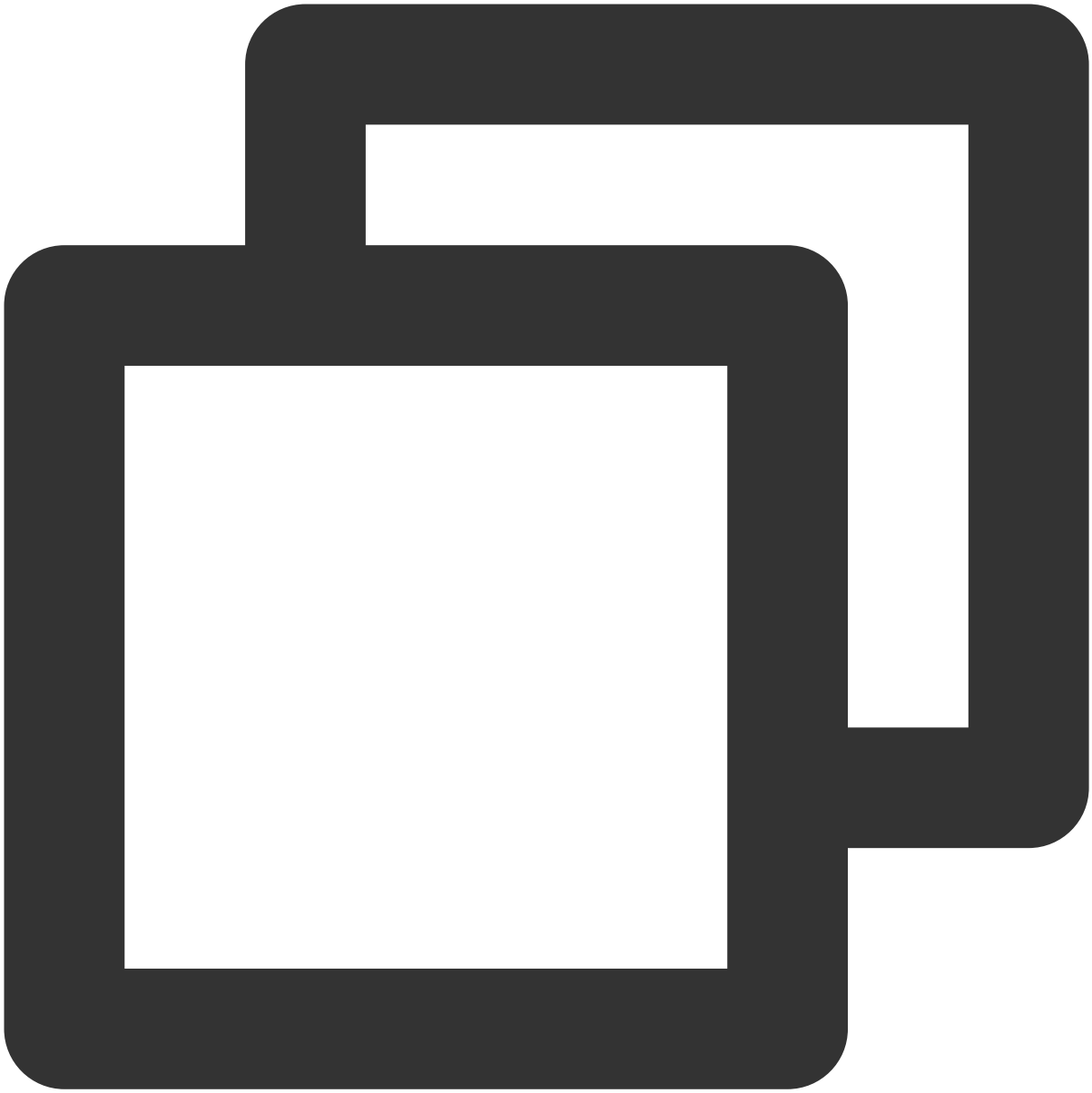
输出错误信息。

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

CPU

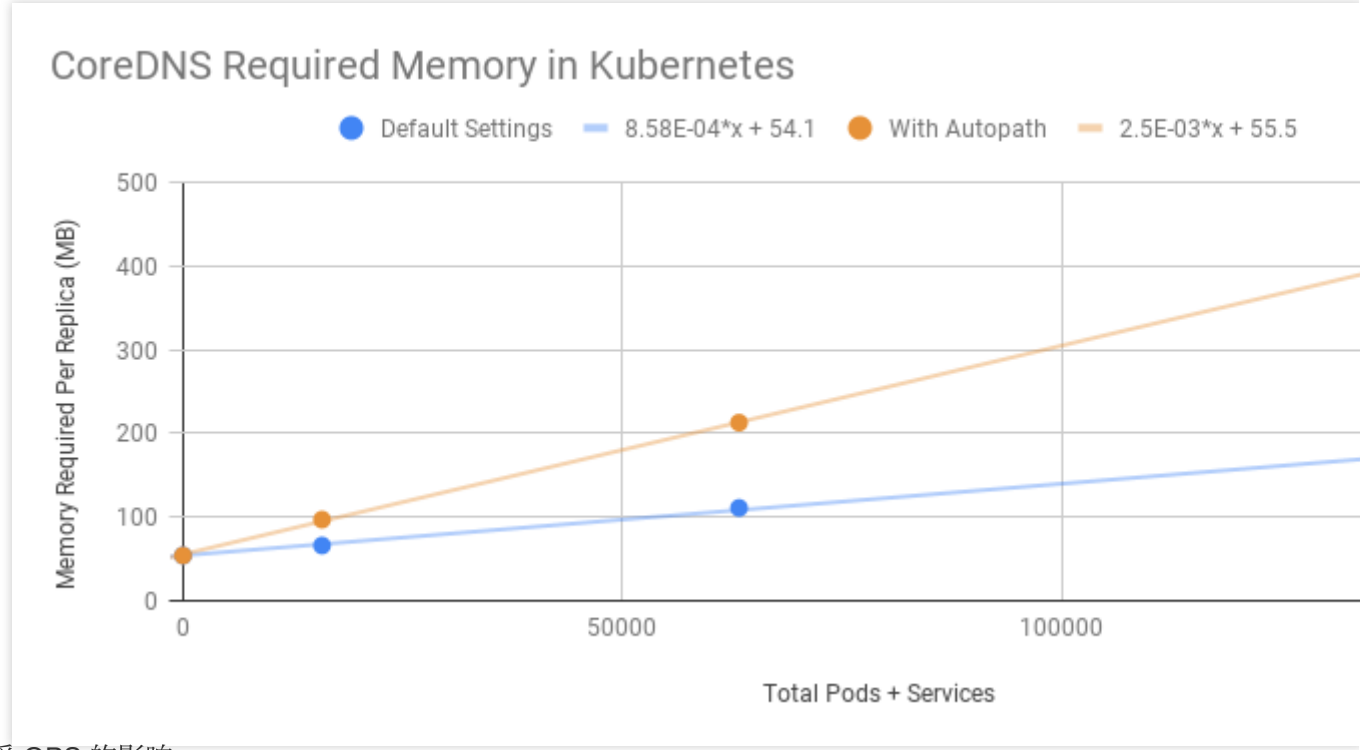
主要取决于集群内 Pod 数和 Service 数。

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



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

在 TKE 集群中使用 NodeLocal DNS Cache

最近更新时间：2024-08-16 15:25:15

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。

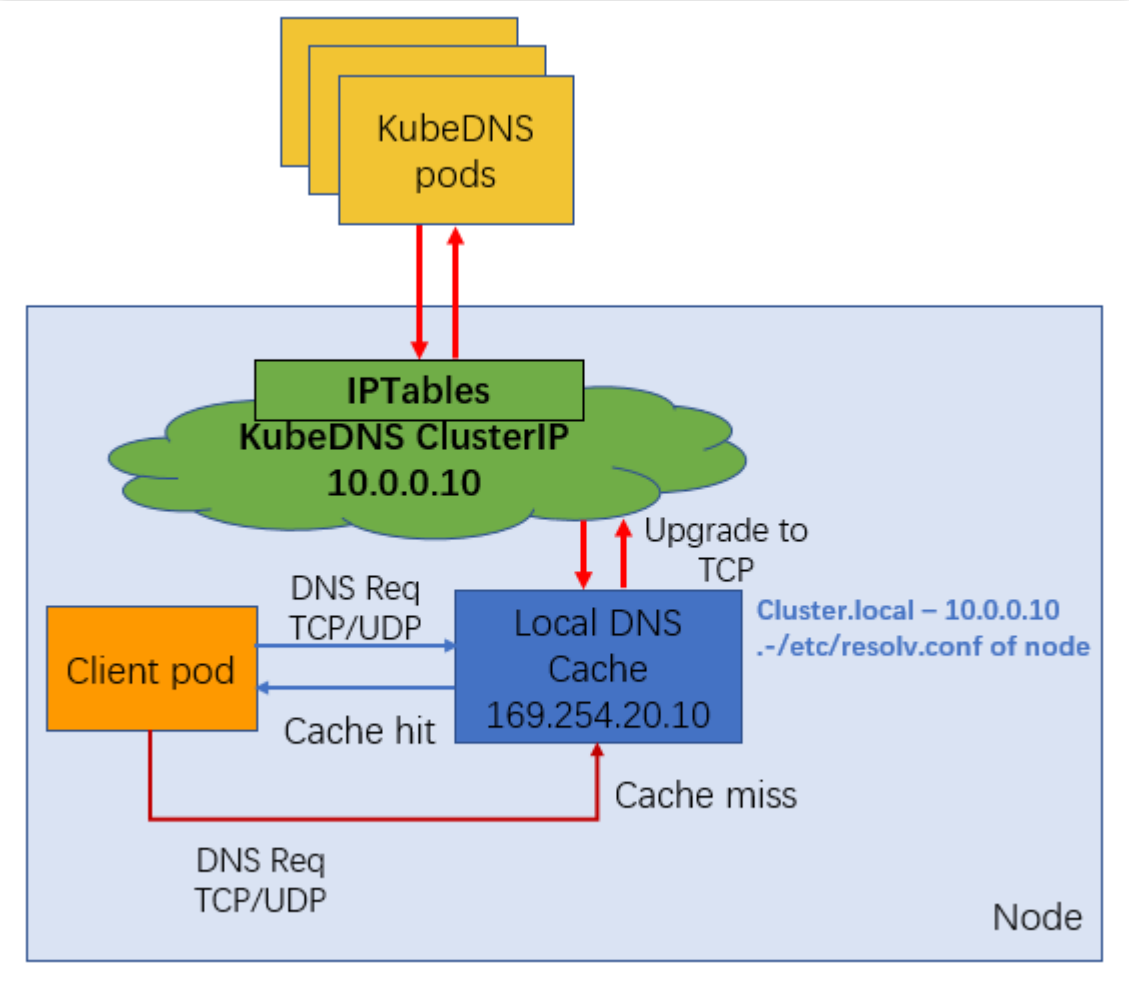
使用限制

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：

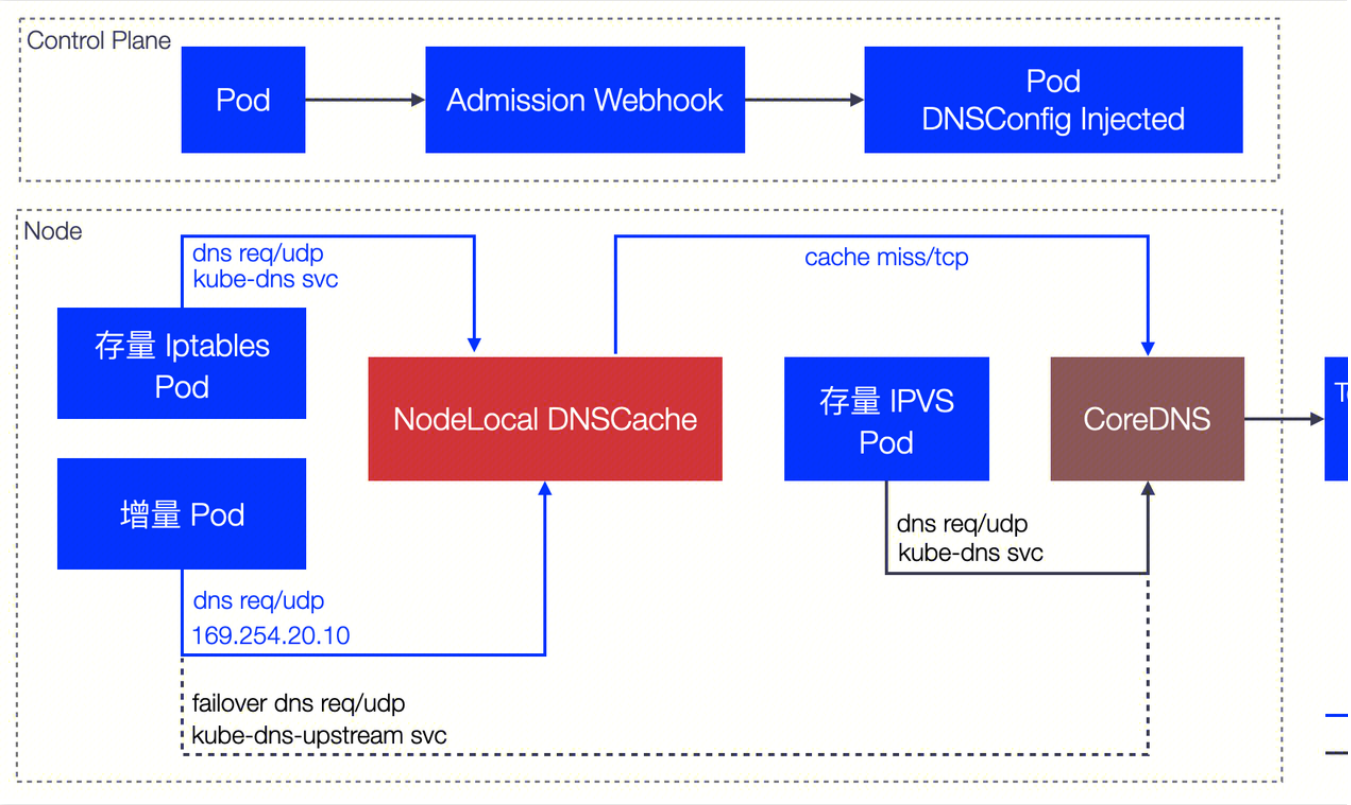
方式1：修改 kubelet 参数 `--cluster-dns`，指向 `169.254.20.10`，然后重启 kubelet 服务。此操作方式存在业务中断的风险。

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，操作方式如下：

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 **NodeLocalDNSCache**。如下图所示：

5. 单击**完成**。
6. 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

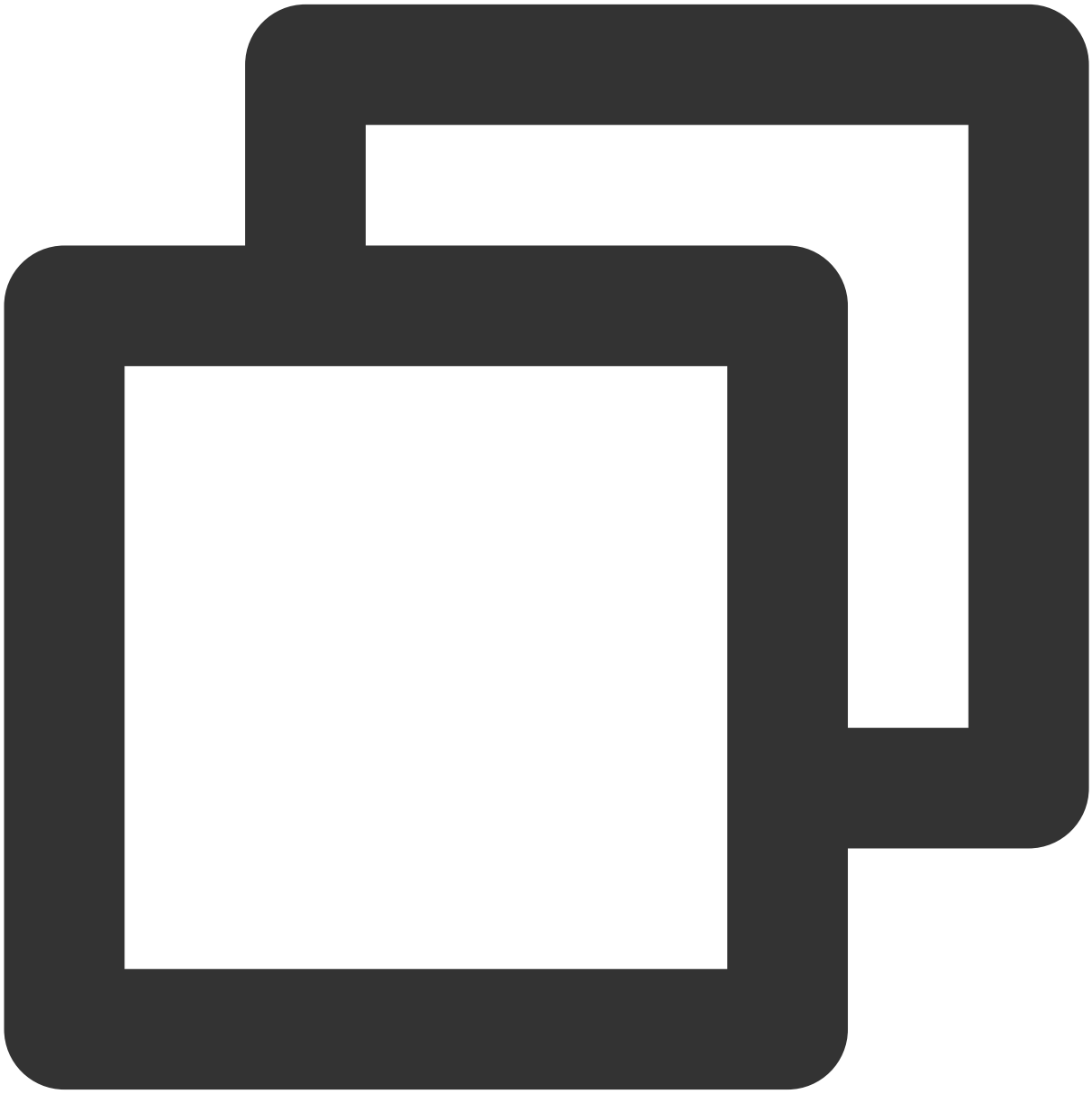
iptables 集群和 ipvs 集群下，对 NodeLocal DNS Cache 的使用方式不同，具体描述如下：

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:
```

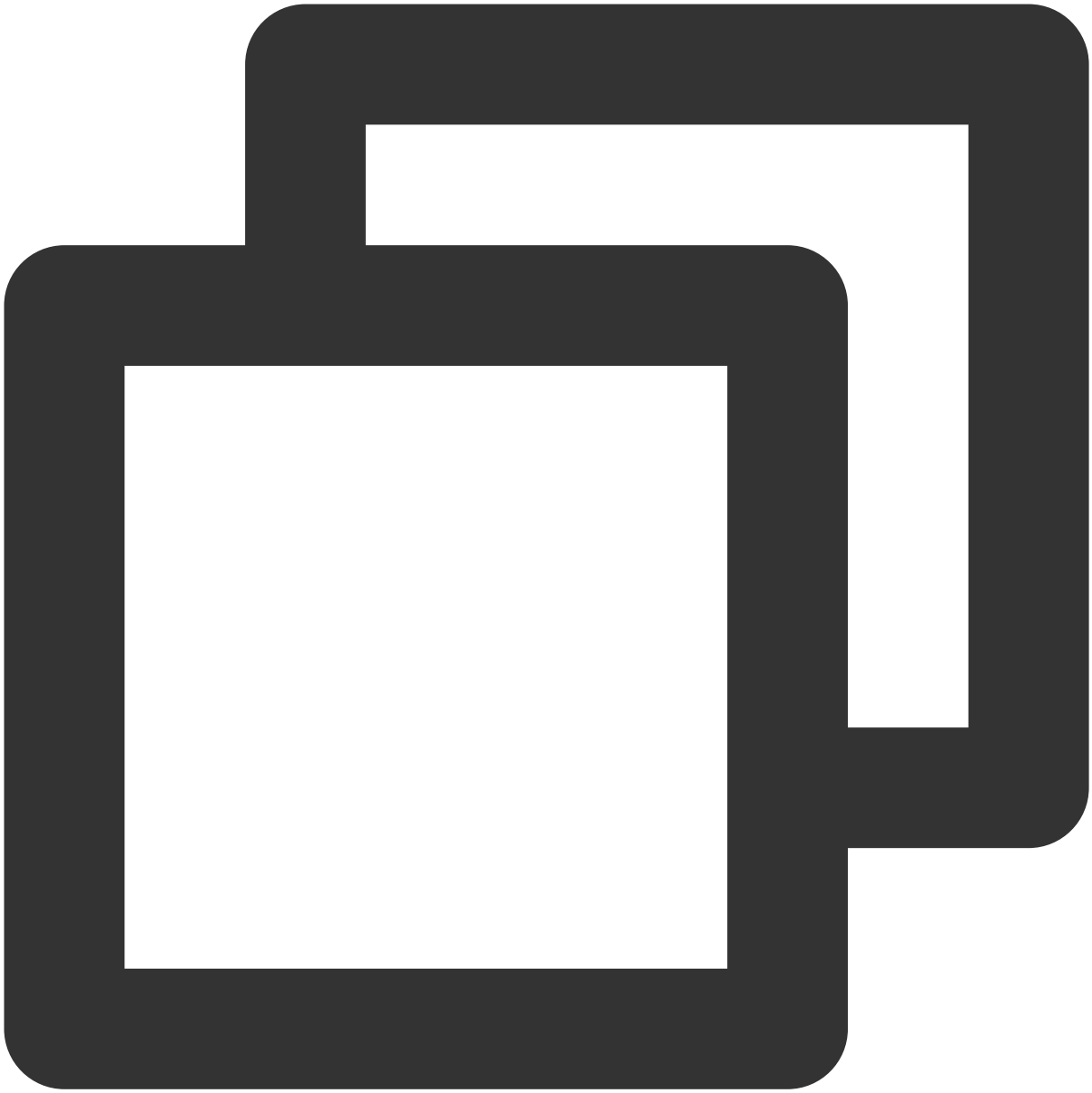
```
    - 169.254.20.10
    - 10.23.1.234
    options:
    - name: ndots
      value: "3"
    - name: attempts
      value: "2"
    - name: timeout
      value: "1"
    searches:
    - default.svc.cluster.local
    - svc.cluster.local
    - cluster.local
  dnsPolicy: None
```

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。

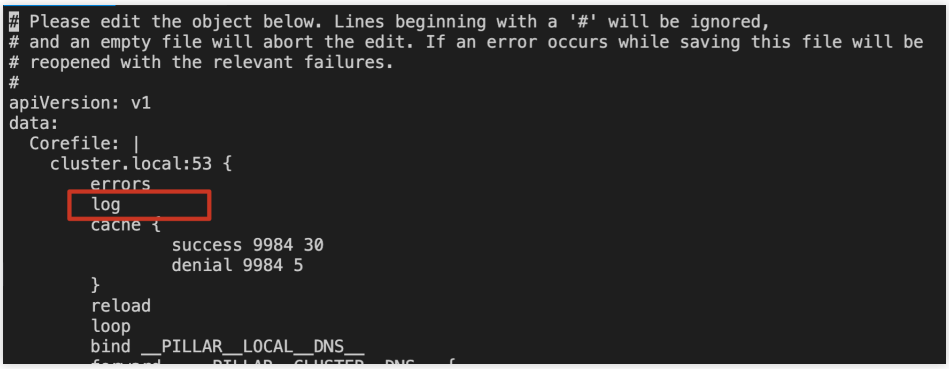
增量 Pod：当满足上述注意事项后，新建 Pod 会自动注入 DNSConfig 配置，访问本节点 `169.254.20.10:53`，使用本地 DNS Cache 能力解析 DNS 请求。

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：



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
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
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 fal
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" NXD
 0.00215277s
[INFO] 10.99.21.3:48989 - 39939 "AAAA IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512"
153 0.003375326s
[INFO] 10.99.21.3:49403 - 2811 "A IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR
135301s
[INFO] 10.99.21.3:49403 - 3664 "AAAA IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERR
```

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"
 153 0.000454866s
[INFO] 10.99.10.29:53379 - 11046 "AAAA IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOE
0.000248217s
[INFO] 10.99.10.29:53379 - 10135 "A IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERRO
002900195s
[INFO] 10.99.10.29:35234 - 52022 "A IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" NX
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 [
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
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
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" NXDO
 0.000706419s
[INFO] 10.99.10.2:50240 - 25235 "AAAA IN kube-dns.kube-system.svc.svc.cluster.local. udp 60 false 512" N
153 0.000955811s
[INFO] 10.99.10.2:35072 - 55625 "A IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERROR
0513704s
[INFO] 10.99.10.2:35072 - 56505 "AAAA IN kube-dns.kube-system.svc.cluster.local. udp 56 false 512" NOERR
.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

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。

在 TKE 中实现自定义域名解析

最近更新时间：2023-03-27 11:08:16

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：

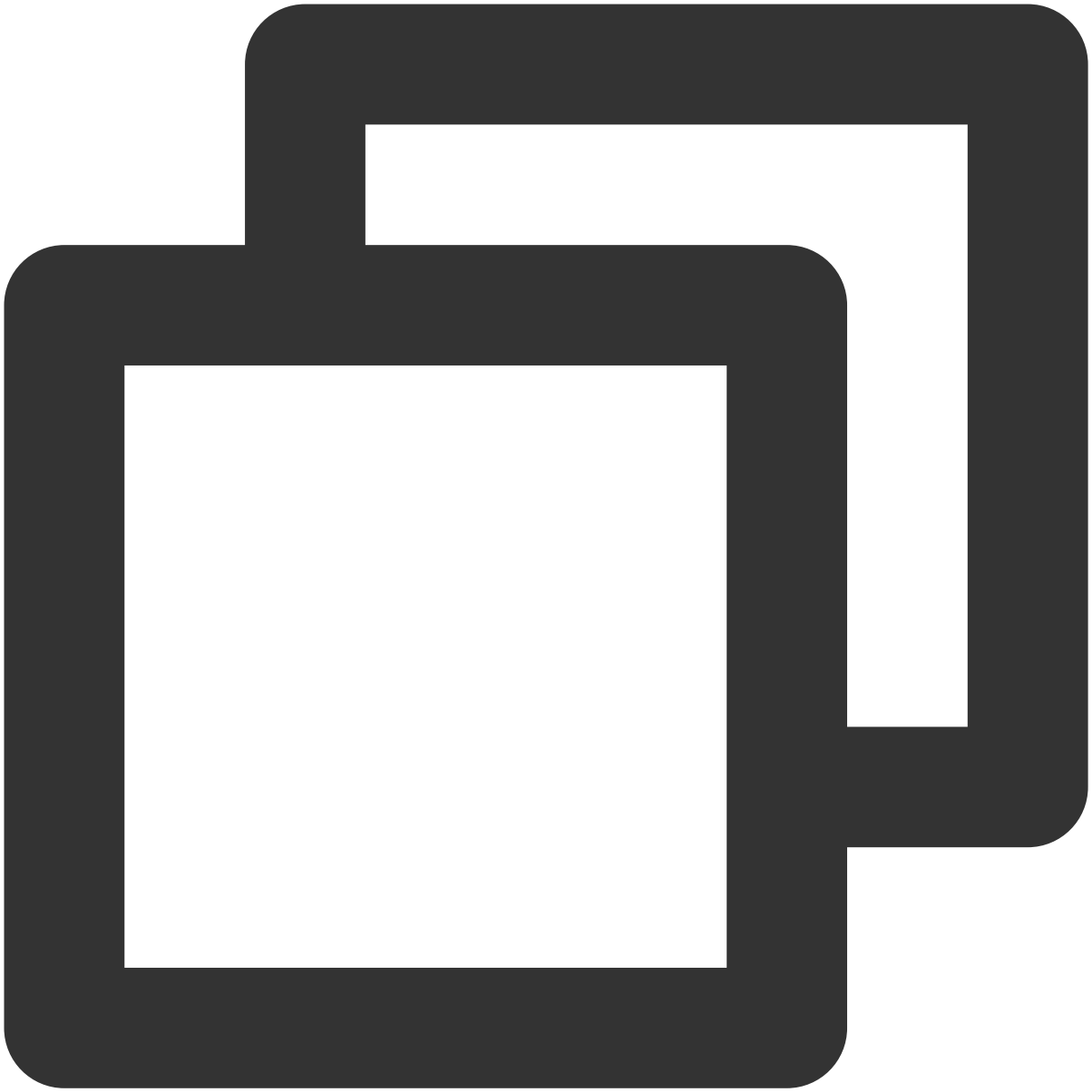
方案	优势
方案1：使用 CoreDNS Hosts 插件配置任意域名解析	简单直观，可以添加任意解析记录。
方案2：使用 CoreDNS Rewrite 插件指向域名到集群内服务	无需提前知道解析记录的 IP 地址，但要求解析记录指向的地址必须部署在集群中。
方案3：使用 CoreDNS Forward 插件将自建 DNS 设为上游 DNS	可以管理大量的解析记录，记录的管理都在自建 DNS 中，增删记录无需修改 CoreDNS 配置。

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

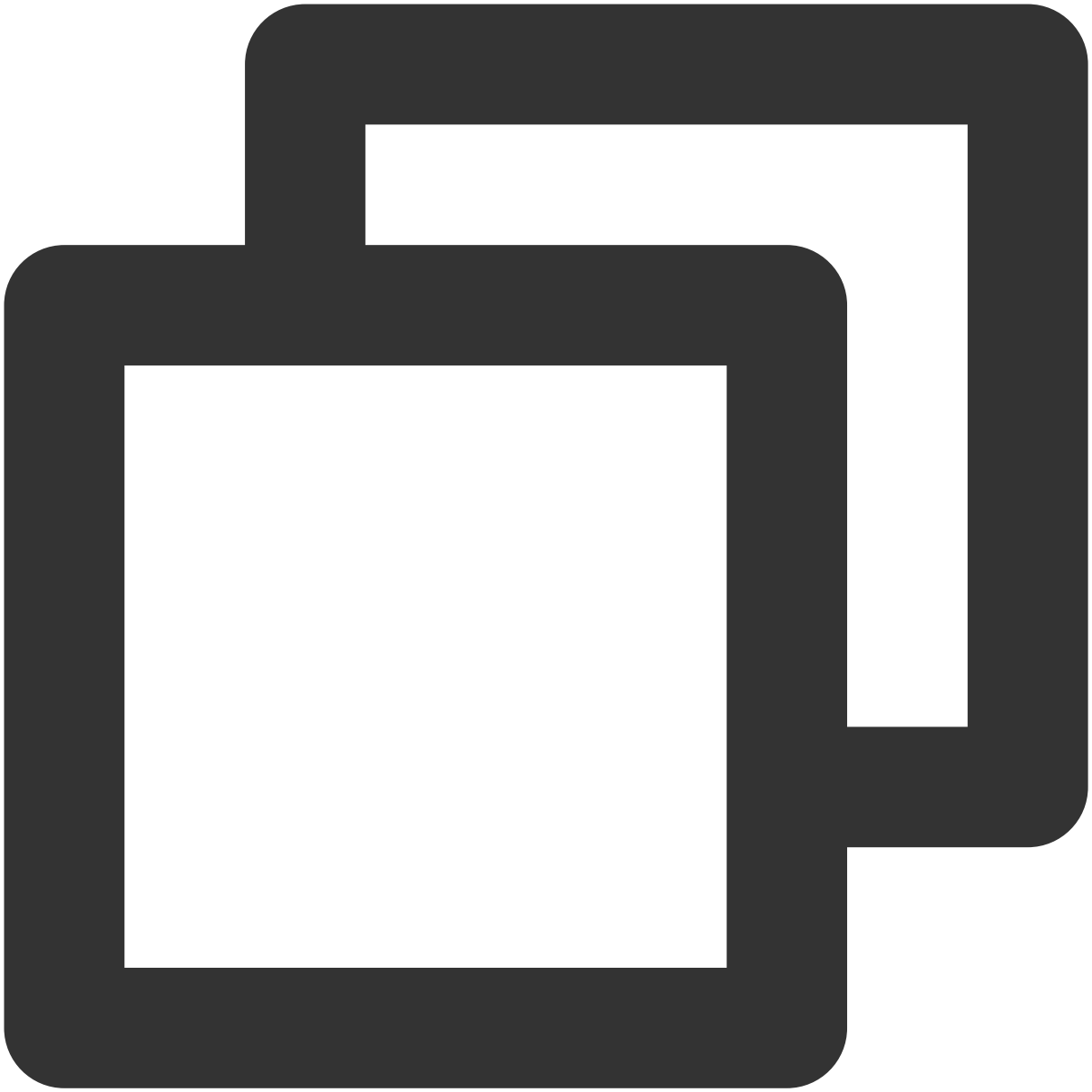
方案1：使用 CoreDNS Hosts 插件配置任意域名解析

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：

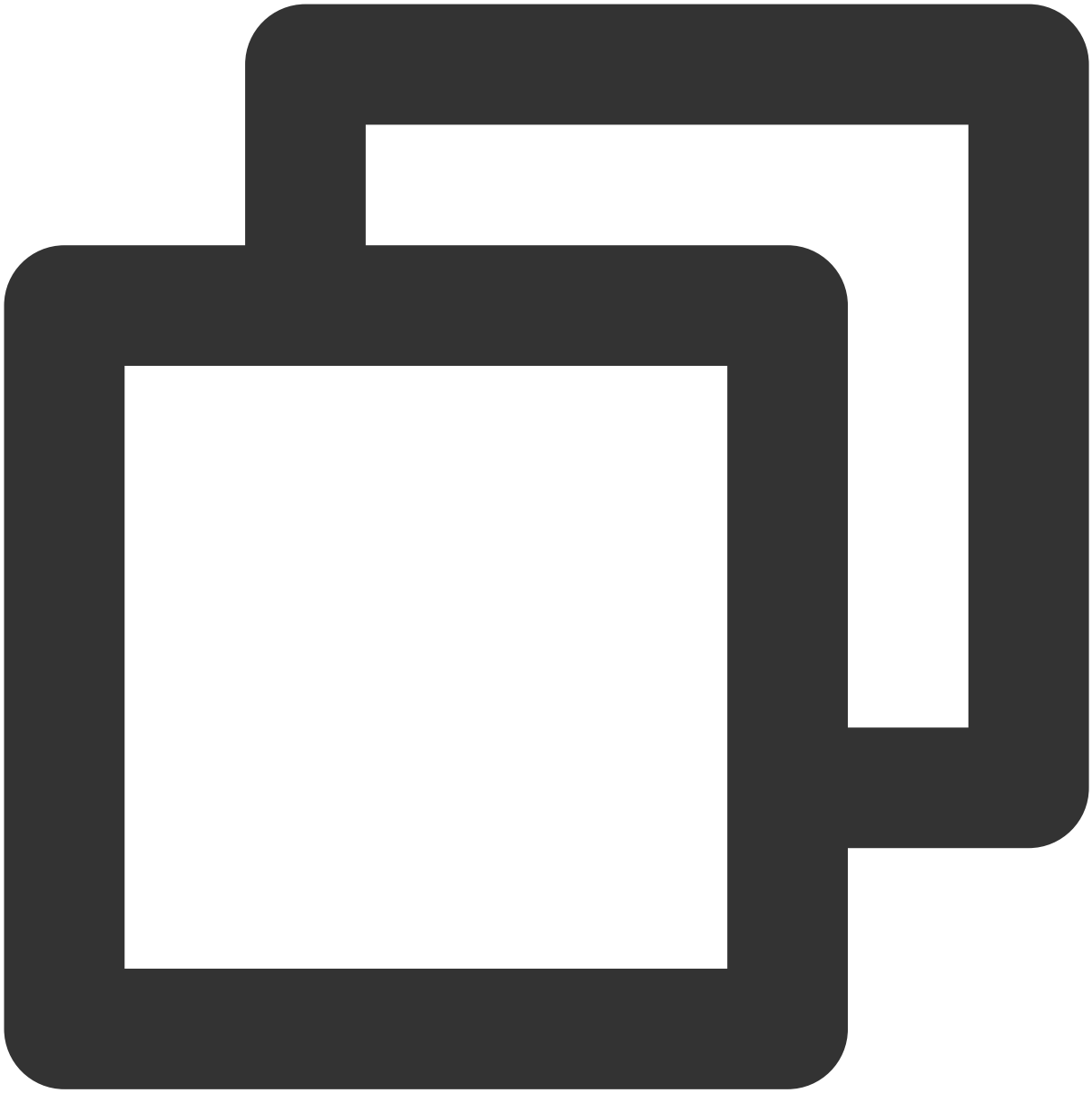


```
hosts {
      192.168.1.6     harbor.example.com
      192.168.1.8     es.example.com
      fallthrough
}
```

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：



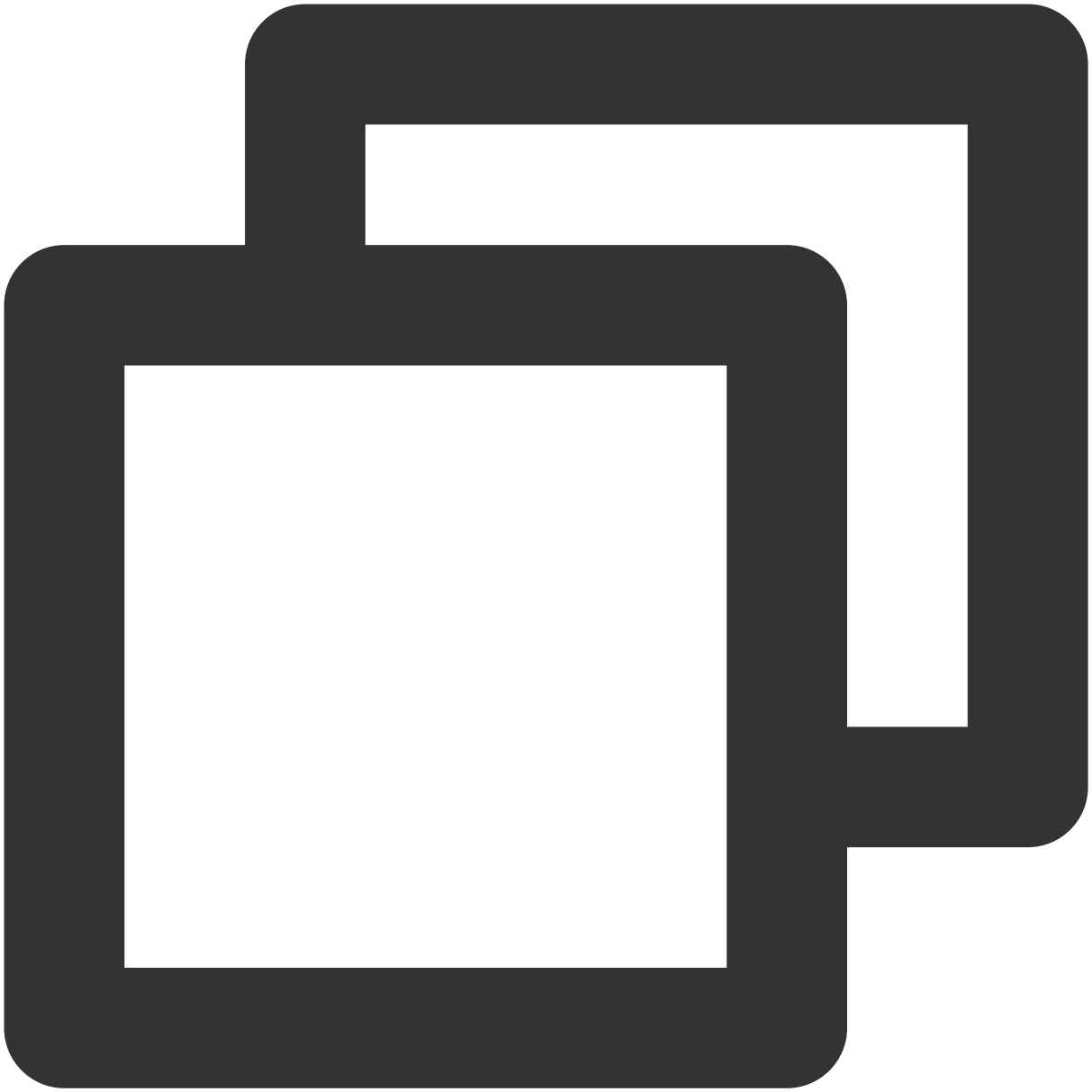
```
apiVersion: v1
data:
      Corefile: |2-
        .:53 {
            errors
            health
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
```

```
            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

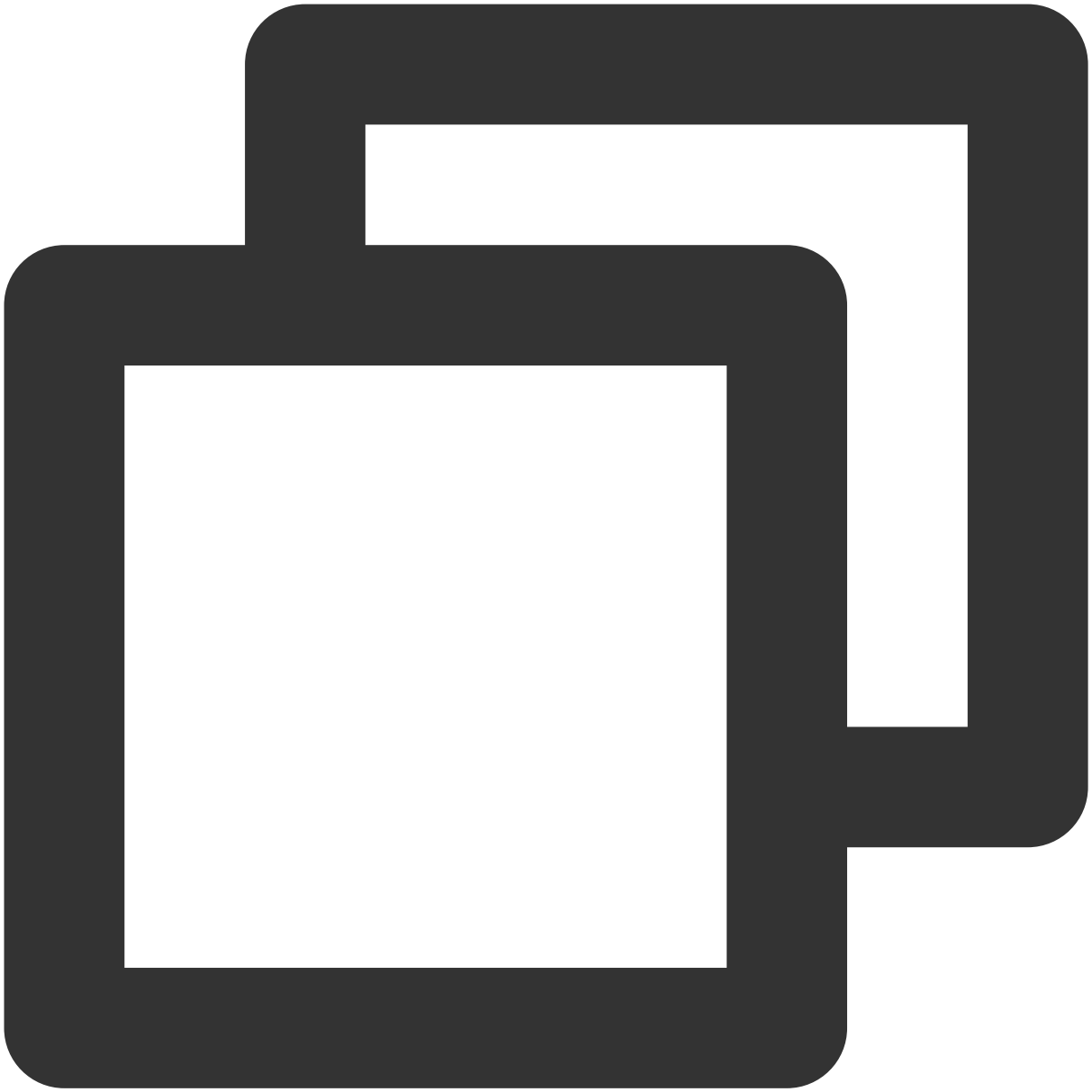
如果需要使用自定义域名的服务部署在集群中，可以使用 CoreDNS 的 Rewrite 插件，将指定域名解析到某个 Service 的 ClusterIP。

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：

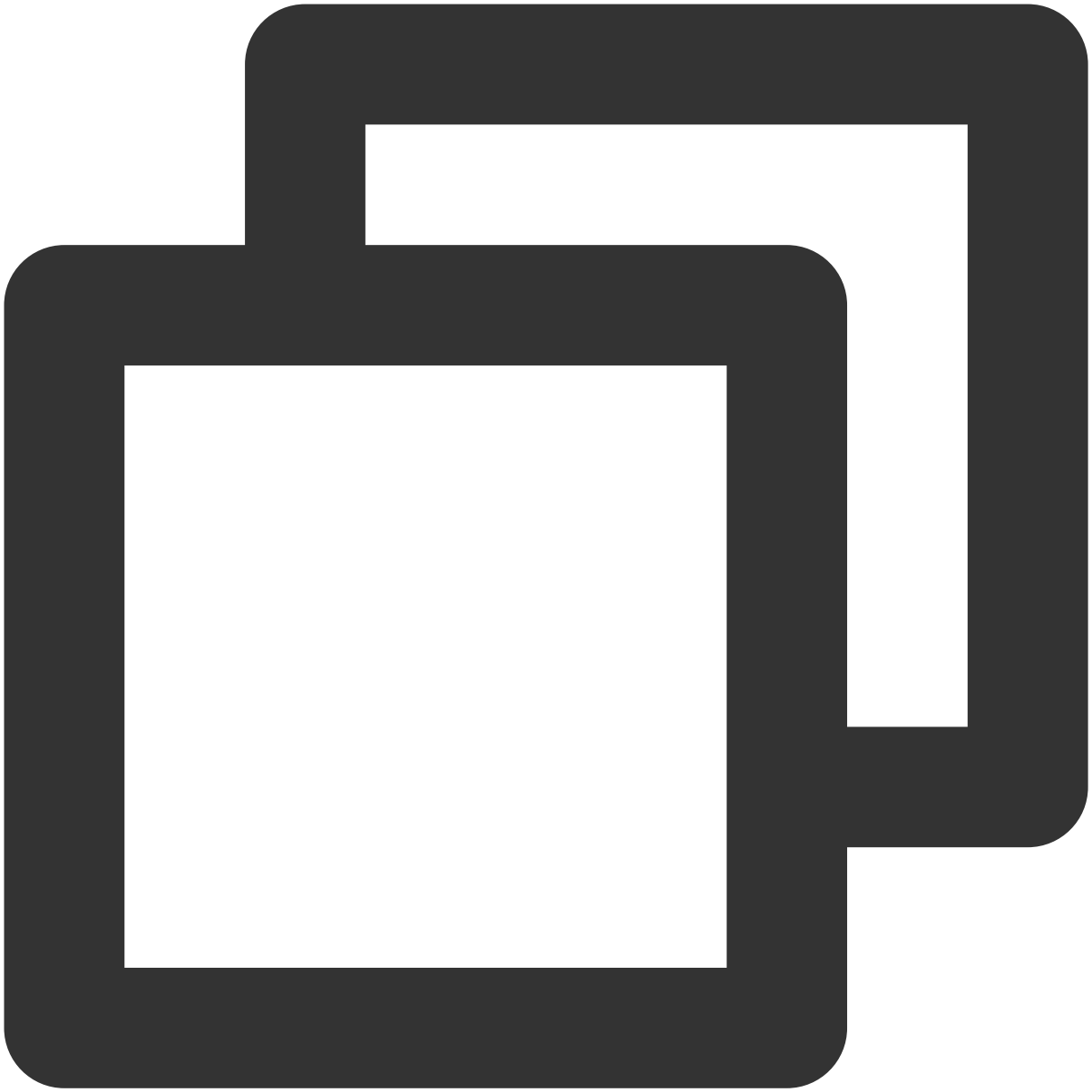


```
rewrite name es.example.com es.logging.svc.cluster.local
```

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：

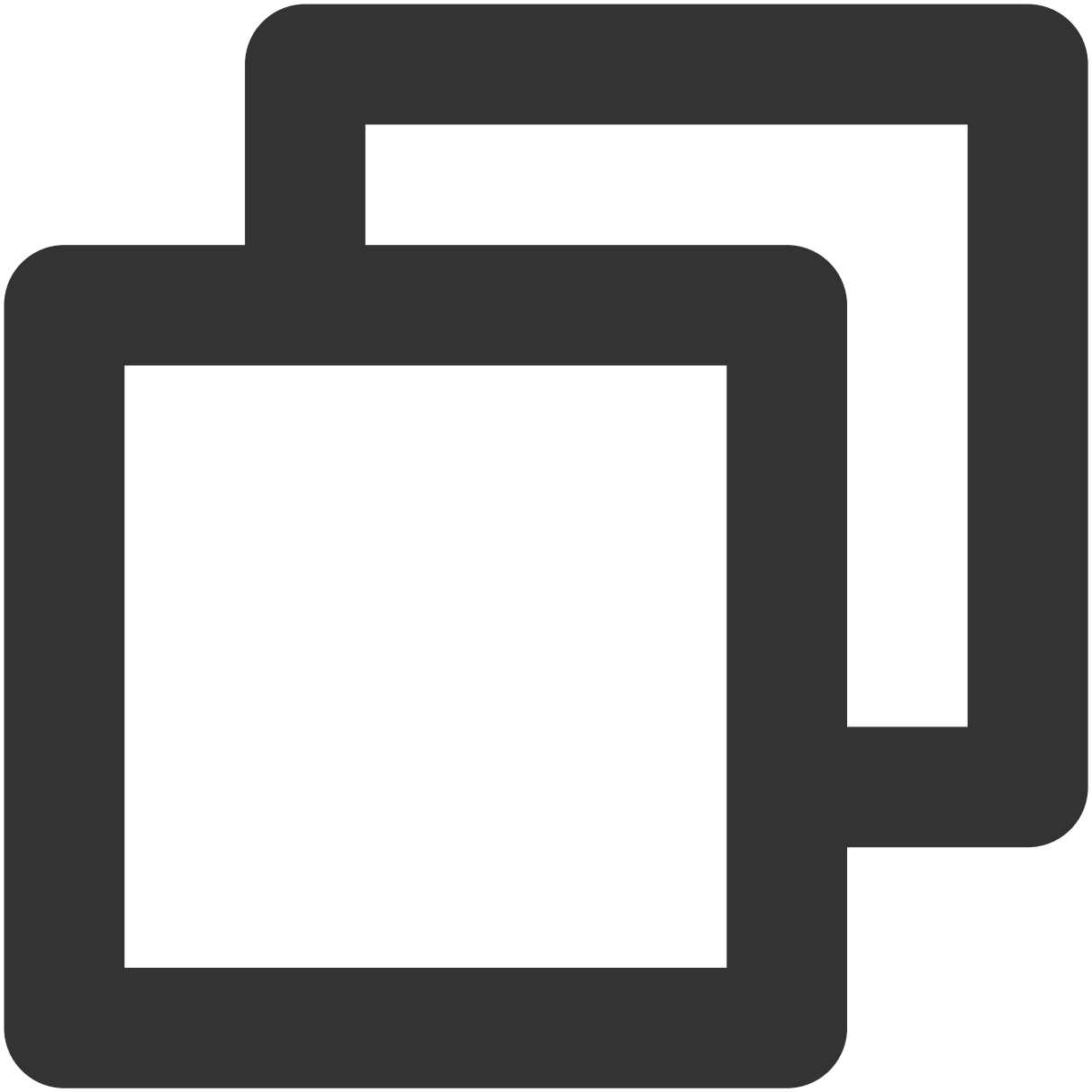


```
apiVersion: v1
data:
      Corefile: |2-
        .:53 {
            errors
            health
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
```

```
            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
```

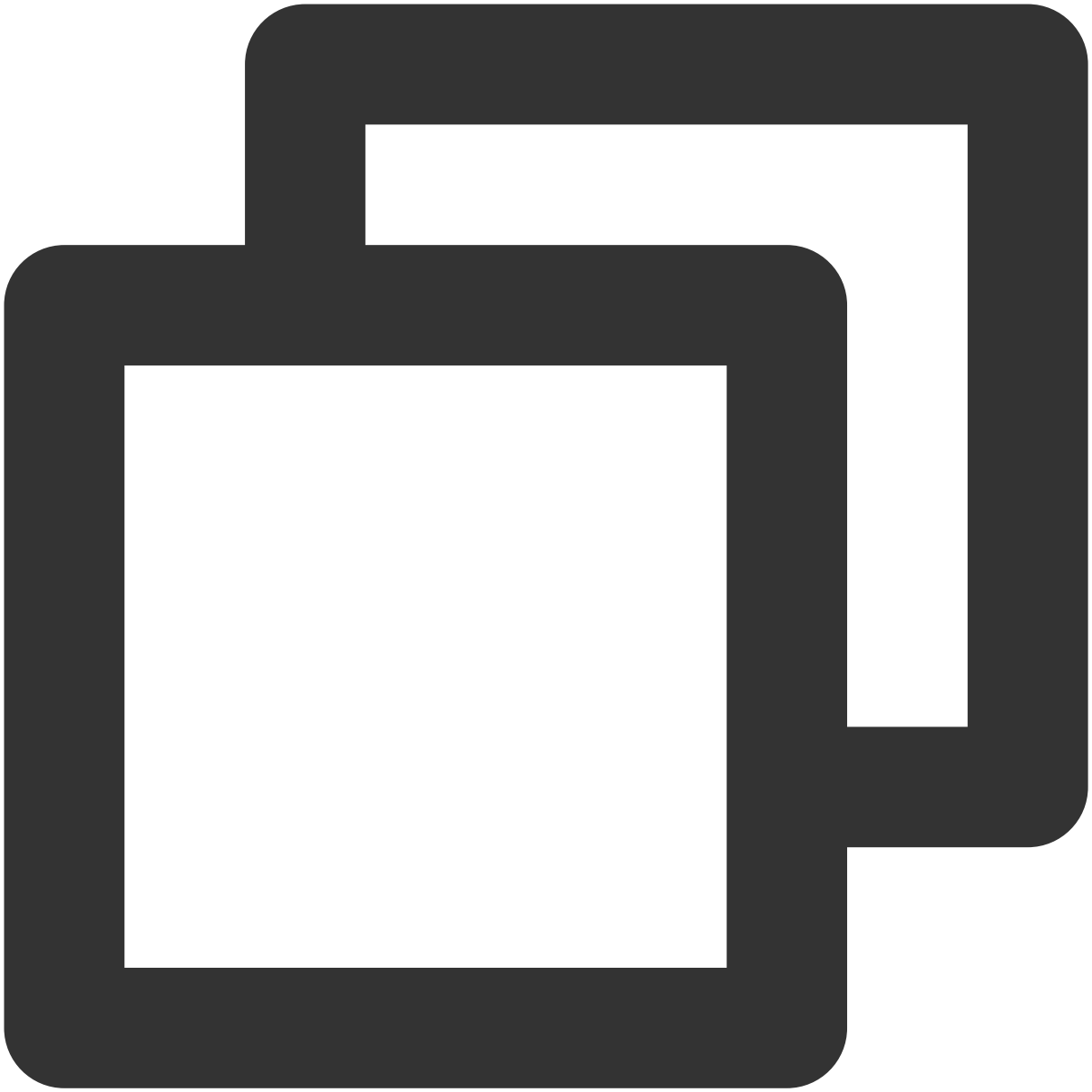
方案3：使用 CoreDNS Forward 插件将自建 DNS 设为上游 DNS

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。



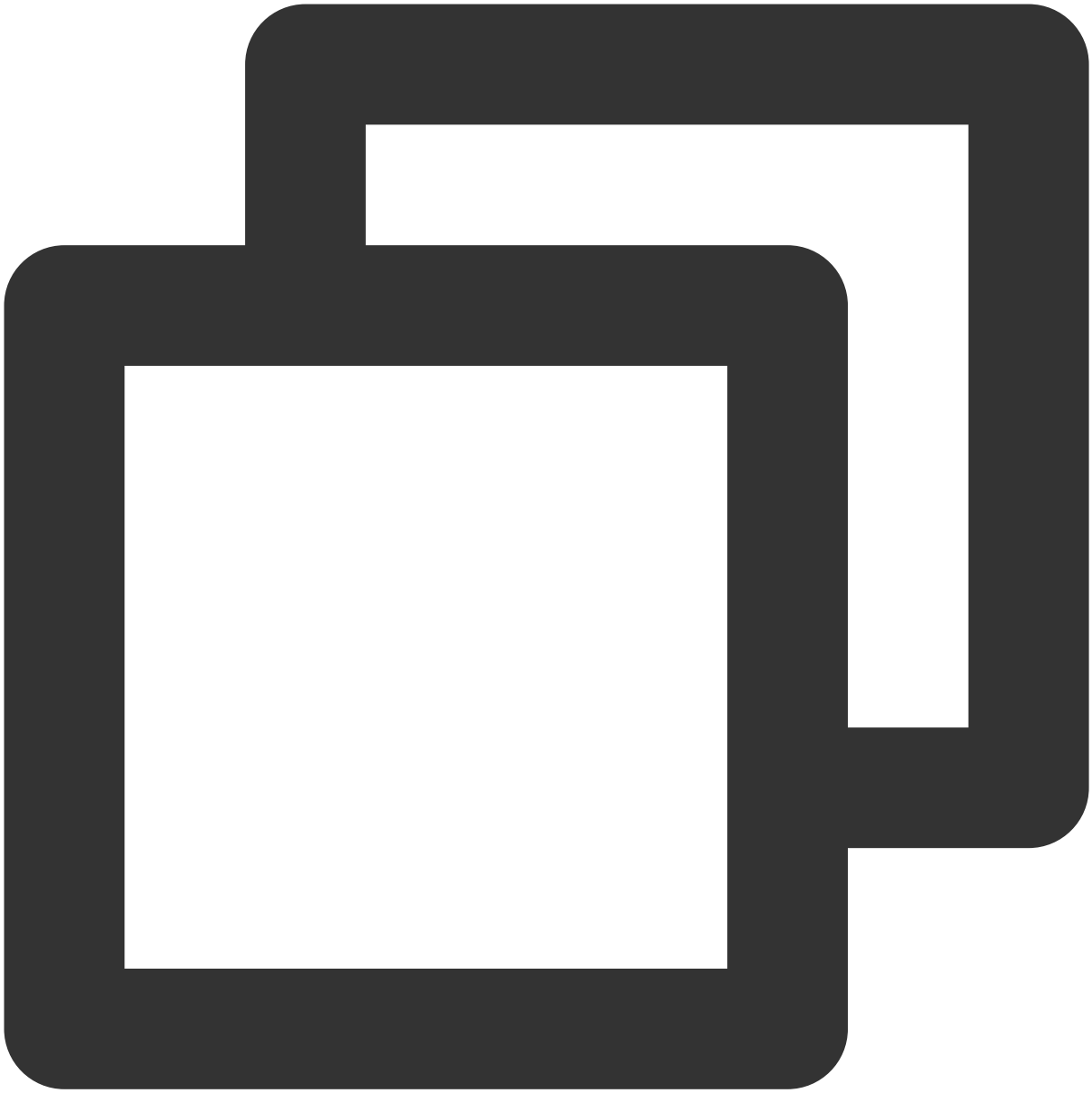
```
forward . /etc/resolv.conf
```

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：



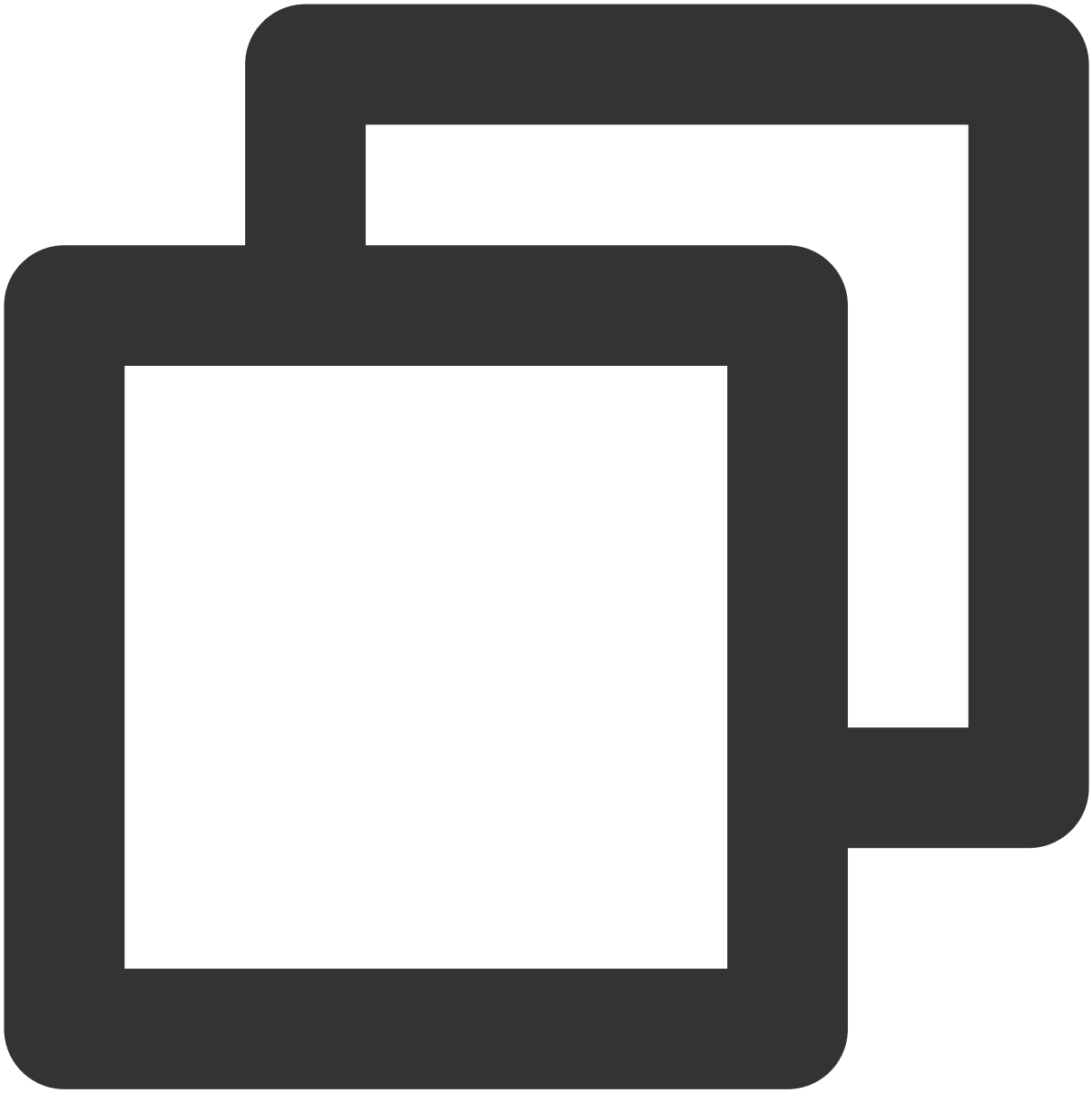
```
apiVersion: v1
data:
   Corefile: |2-
     .:53 {
         errors
         health
         kubernetes cluster.local. in-addr.arpa ip6.arpa {
             pods insecure
             upstream
             fallthrough in-addr.arpa ip6.arpa
         }
```

```
        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：2023-11-22 10:47:56

本文介绍如何在腾讯云容器服务的集群里面配置 ExternalDNS。

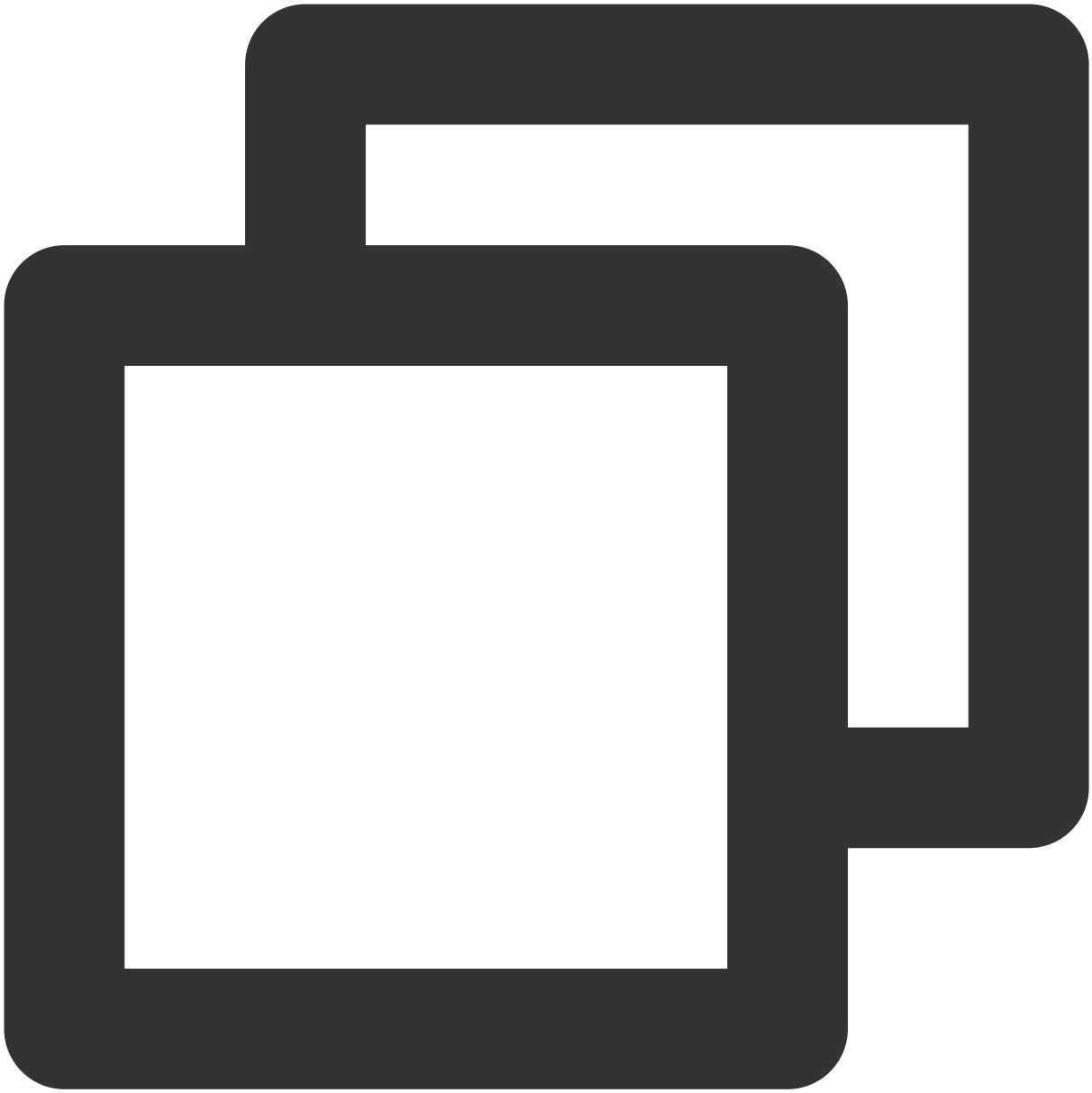
什么是 External DNS

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

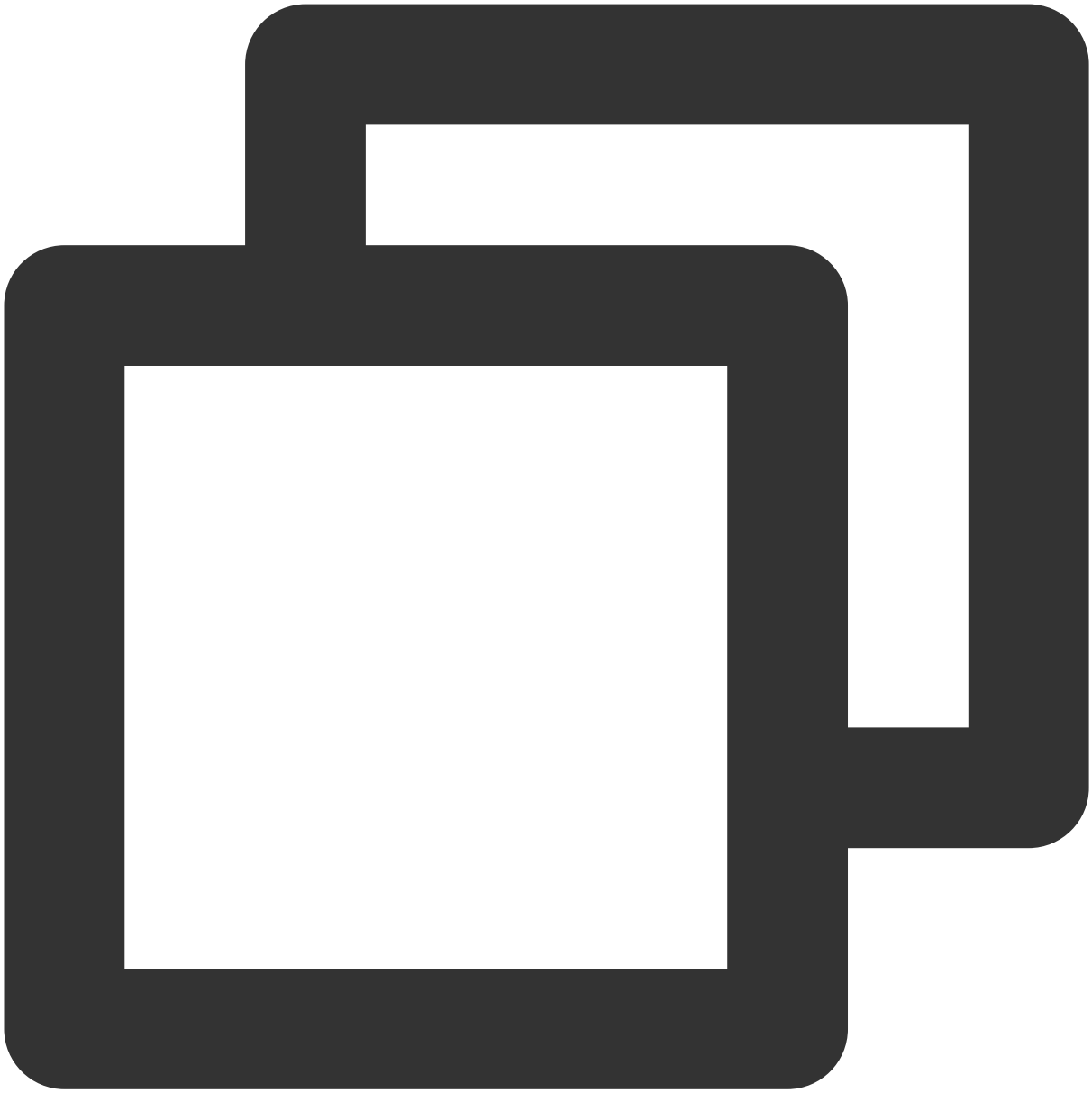
在 PrivateDNS 控制台创建 DNS 域名。DNS 域名记录中将会包含 DNS 记录。

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 Kuberentes 集群中部署相关资源对象



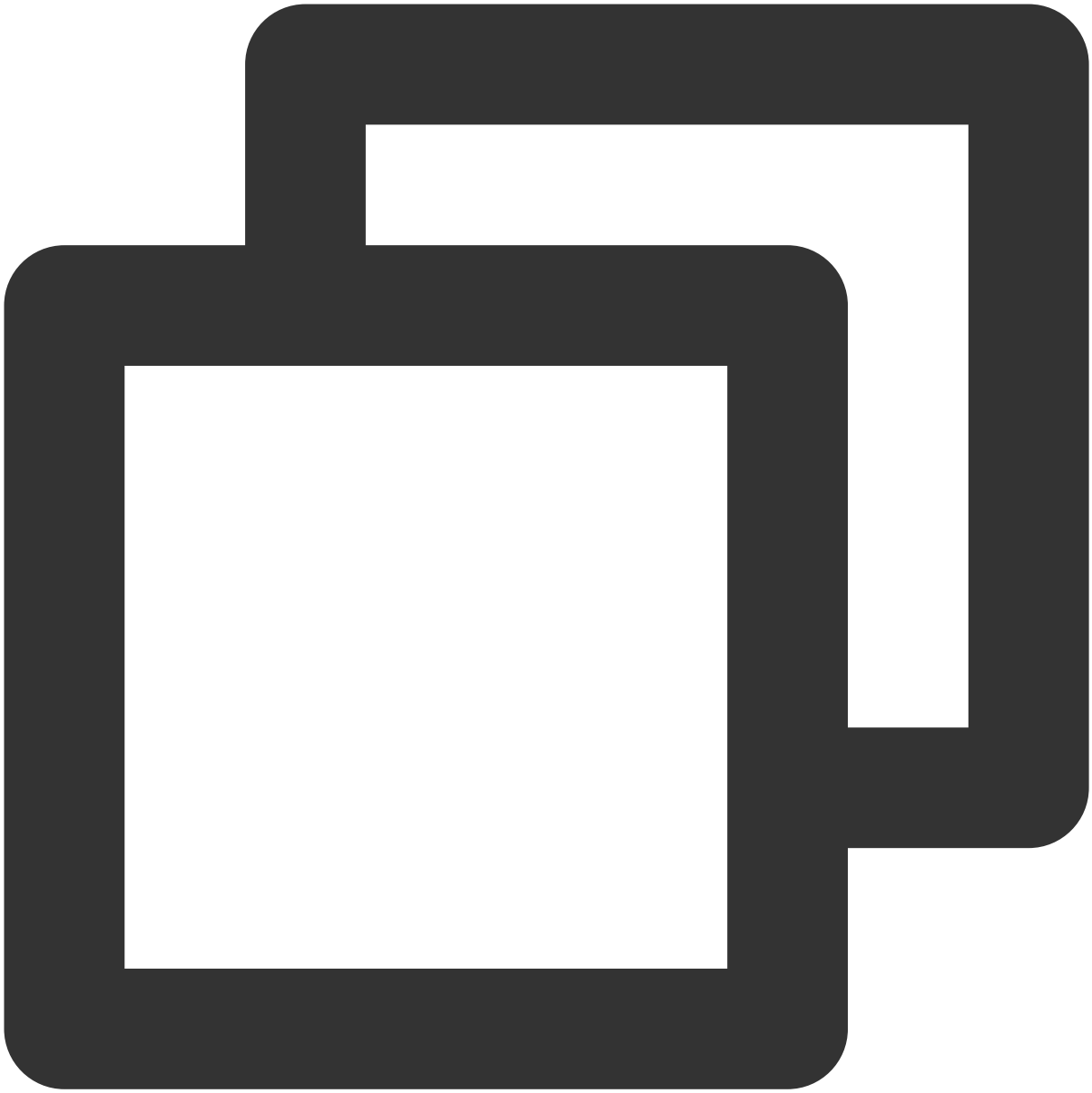
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: external-dns
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: external-dns
rules:
- apiGroups: [""]
```

```
  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
    }
---
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
        - --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
        - --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
    external-dns.alpha.kubernetes.io/internal-hostname: nginx-internal.external-dns
    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
```

说明：

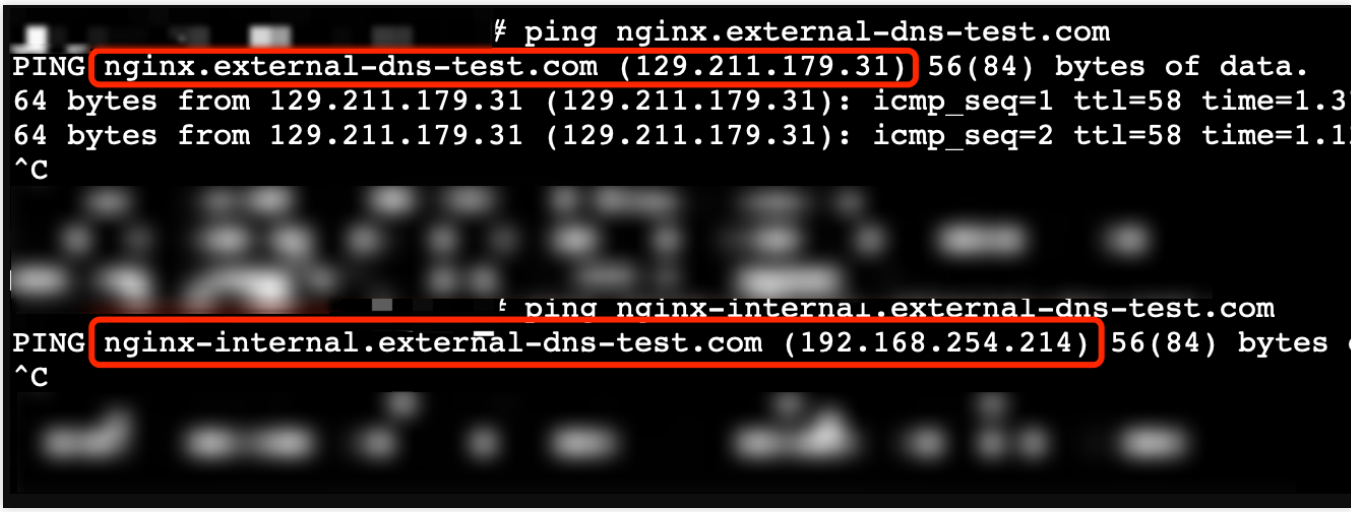
nginx.external-dns-test.com 将记录服务的 Loadbalancer VIP。

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：

当您在与集群所在同一个 VPC 内节点上， ping 名为 nginx 的 Service 里 annotation 的域名声明时，会自动解析成 ClusterIP 和 Loadbalancer VIP，如下图所示：


ping nginx.external-dns-test.com
PING nginx.external-dns-test.com (129.211.179.31) 56(84) bytes of data.
64 bytes from 129.211.179.31 (129.211.179.31): icmp_seq=1 ttl=58 time=1.3
64 bytes from 129.211.179.31 (129.211.179.31): icmp_seq=2 ttl=58 time=1.1
^C
ping nginx-internal.external-dns-test.com
PING nginx-internal.external-dns-test.com (192.168.254.214) 56(84) bytes
^C

使用 Network Policy 进行网络访问控制

最近更新时间：2022-12-12 17:25:28

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

NetworkPolicy Feature	是否支持
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。

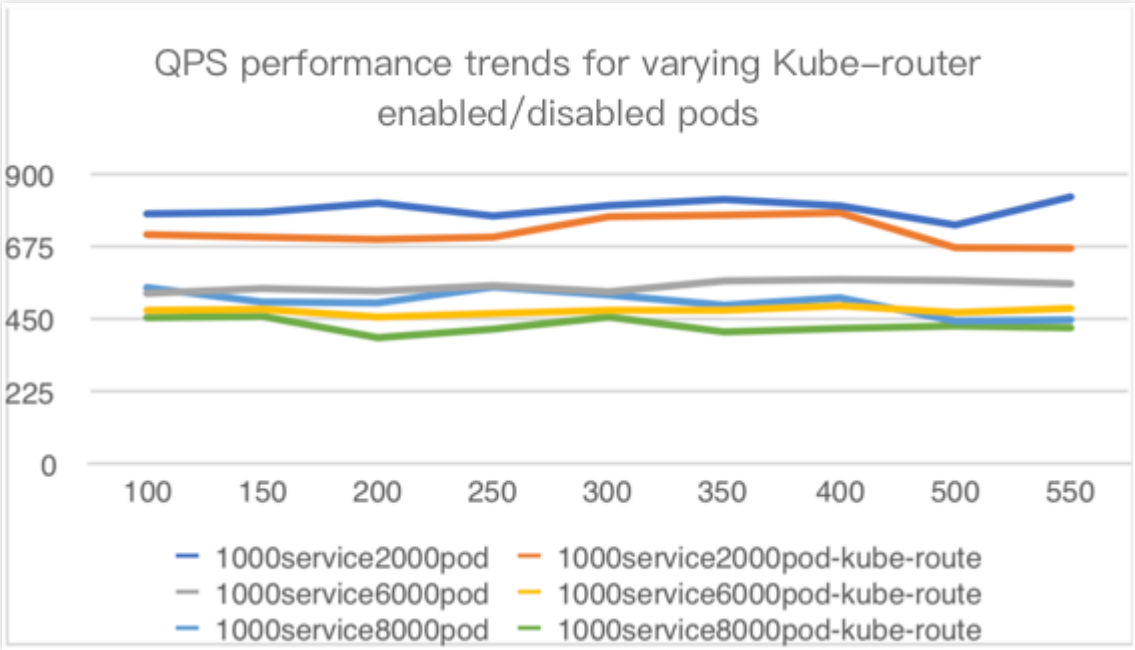
2. 部署1000个 service，每个分别对应 2/6/8 个 Pod（Nginx），作为干扰组。

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

在 TKE 上部署 Nginx Ingress

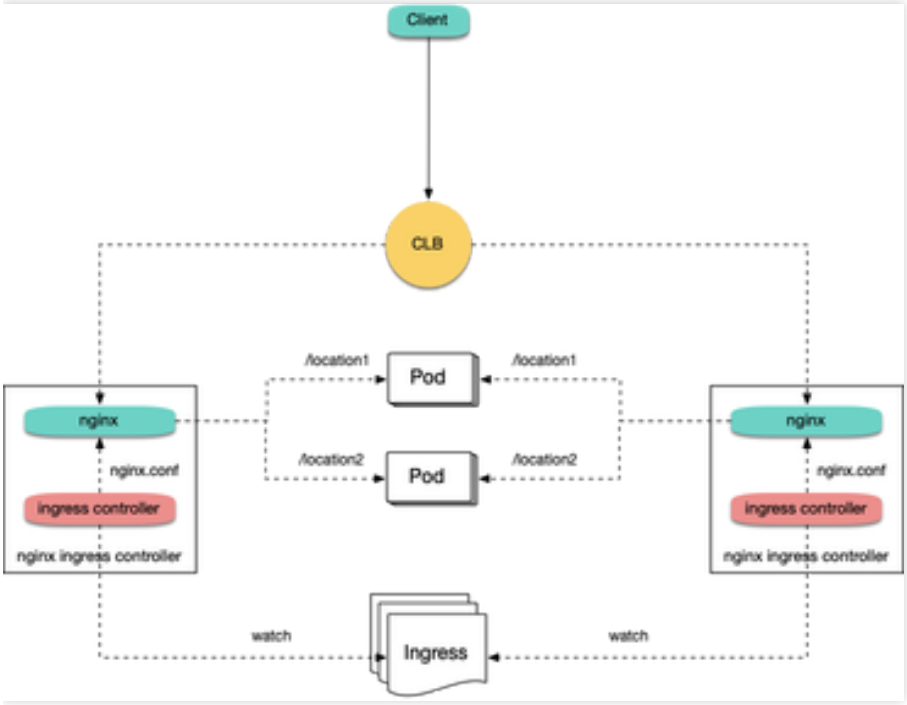
最近更新时间：2020-12-16 12:17:46

概述

Nginx Ingress 功能强大且性能极高，有多种部署方式。本文将介绍 Nginx Ingress 在腾讯云容器服务（Tencent Kubernetes Engine，TKE）上 Deployment + LB、Daemonset + HostNetwork + LB 和 Deployment + LB 直通 Pod 三种部署方案及其部署方法。

Nginx Ingress 简介

Nginx Ingress 是 Kubernetes Ingress 的一种实现。它通过 watch Kubernetes 集群的 Ingress 资源，将 Ingress 规则转换成 Nginx 的配置，让 Nginx 进行7层的流量转发。如下图所示：



Nginx Ingress 有以下两种实现方式，本文重点对 Kubernetes 开源社区的实现进行介绍：

- Kubernetes 开源社区的实现
- Nginx 官方的实现

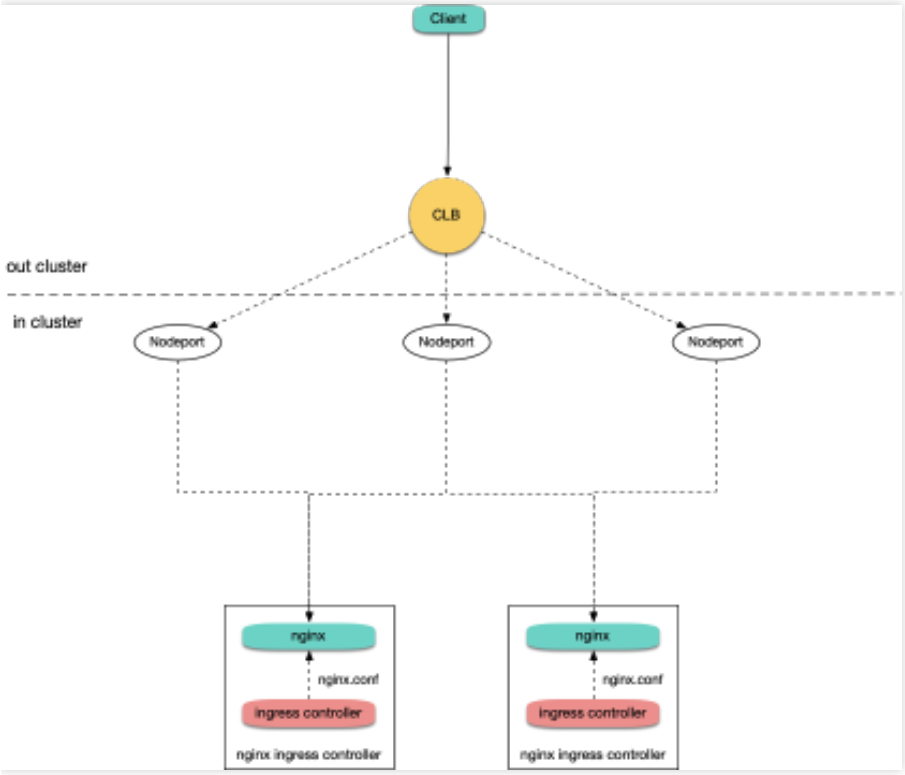
部署方案选型建议

对 Nginx Ingress 在 TKE 上部署的三种方案进行比较，本文向您提出以下选型建议：

1. Deployment + LB：较为简单通用，但在大规模和高并发场景存在性能问题。如果对性能要求低，可以考虑使用此方案。
2. Daemonset + HostNetwork + LB：使用 hostNetwork 性能好，但需要手动维护 CLB 和 Nginx Ingress 节点，也无法实现自动扩缩容，不太建议用此方案。
3. Deployment + LB 直通 Pod：性能好，而且不需要手动维护 CLB，是理想解决方案。但在此方案中需要集群支持 VPC-CNI，如果已有集群本身用的 VPC-CNI 网络插件，或者用的 Global Router 网络插件并开启了 VPC-CNI 的支持（两种模式混用），建议使用此方案。

方案1： Deployment + LB

在 TKE 上部署 Nginx Ingress 最简单的方式是将 Nginx Ingress Controller 以 Deployment 的方式部署，并且为其创建 LoadBalancer 类型的 Service（自动创建负载均衡 CLB 或绑定已有 CLB），使 CLB 接收外部流量，再转发到 Nginx Ingress 内部。如下图所示：



当前 TKE 上 LoadBalancer 类型的 Service 默认实现是基于 NodePort：CLB 会绑定各节点的 NodePort 作为后端 RS（Real Server），将流量转发到节点的 NodePort，节点再通过 Iptables 或 IPVS 将请求路由到 Service 对应的后端 Pod（即 Nginx Ingress Controller 的 Pod）。后续如有节点的增删，CLB 也会自动更新节点 NodePort 的绑定。
执行以下命令安装 Nginx Ingress：

```
kubectl create ns nginx-ingress
```

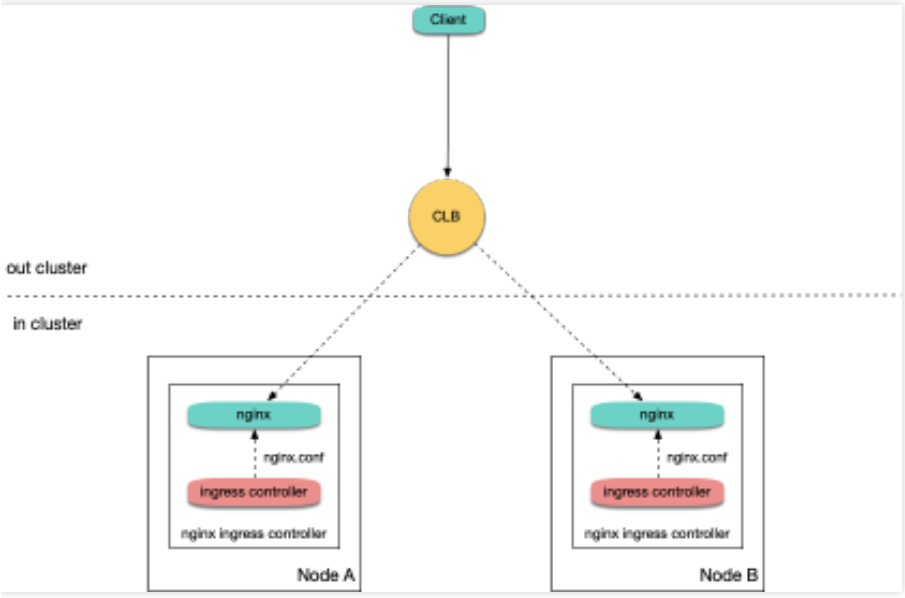
```
kubectl apply -f https://raw.githubusercontent.com/TencentCloudContainerTeam/manifest/master/nginx-ingress/nginx-ingress-deployment.yaml -n nginx-ingress
```

方案2： Daemonset + HostNetwork + LB

在方案1中，流量会经过一层 NodePort，会多一层转发。因此存在以下问题：

- 转发路径较长，流量到 NodePort 后会再经过 Kubernetes 内部 LB，通过 Iptables 或 IPVS 转发到 Nginx，会增加网络耗时。
- 经过 NodePort 必然发生 SNAT，如果流量过于集中则容易导致源端口耗尽或 conntrack 插入冲突导致丢包，引发部分流量异常。
- 每个节点的 NodePort 也充当一个负载均衡器，CLB 如果绑定大量节点的 NodePort，LB 的状态就分散在每个节点上，容易导致全局负载不均。
- CLB 会对 NodePort 进行健康探测，探测包最终会被转发到 Nginx Ingress 的 Pod，如果 CLB 绑定的节点多，而 Nginx Ingress 的 Pod 少，会导致探测包对 Nginx Ingress 造成较大的压力。

在方案2中，提出以下解决方法：
让 Nginx Ingress 使用 hostNetwork，CLB 直接绑节点 IP + 端口（80,443），不用经过 NodePort。由于使用 hostNetwork，Nginx Ingress 的 pod 就不能被调度到同一节点，为避免端口监听冲突，可提前选取部分节点作为边缘节点，专门用于部署 Nginx Ingress，并为这些节点打上 label，然后 Nginx Ingress 以 DaemonSet 方式部署在这些节点上。架构如下图所示：



如需安装 Nginx Ingress，请执行以下步骤：

1. 执行以下命令，将规划好的用于部署 Nginx Ingress 的节点打上 label（注意替换节点名称）：

```
kubectl label node 10.0.0.3 nginx-ingress=true
```

2. 执行以下命令，将 Nginx Ingress 部署在这些节点上:

```
kubectl create ns nginx-ingress
```

```
kubectl apply -f https://raw.githubusercontent.com/TencentCloudContainerTeam/manifest/master/nginx-ingress/nginx-ingress-daemonset-hostnetwork.yaml -n nginx-ingress
```

3. 手动创建 CLB，及创建80和443端口的 TCP 监听器，分别绑定已部署 Nginx Ingress 节点的80和443端口。

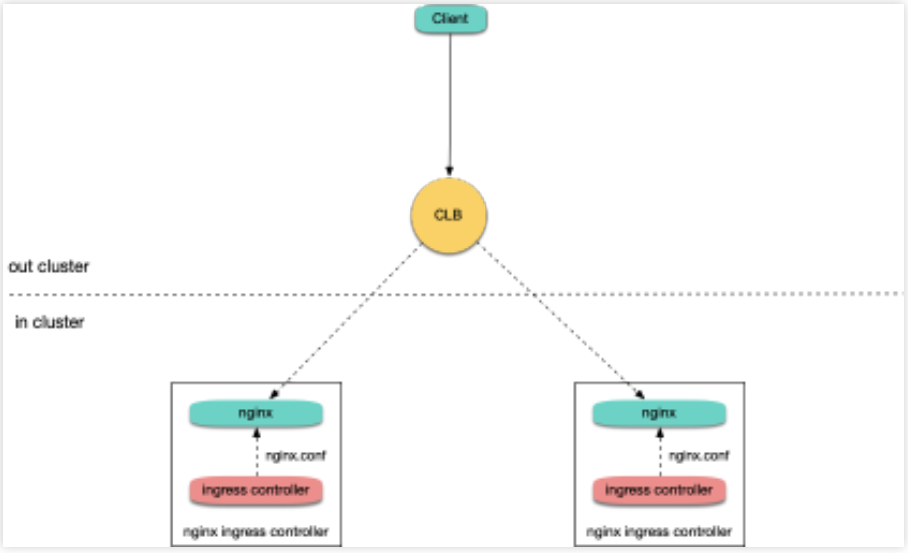
方案3：Deployment + LB 直通 Pod

方案2相比方案1更有优势，但仍存在以下问题：

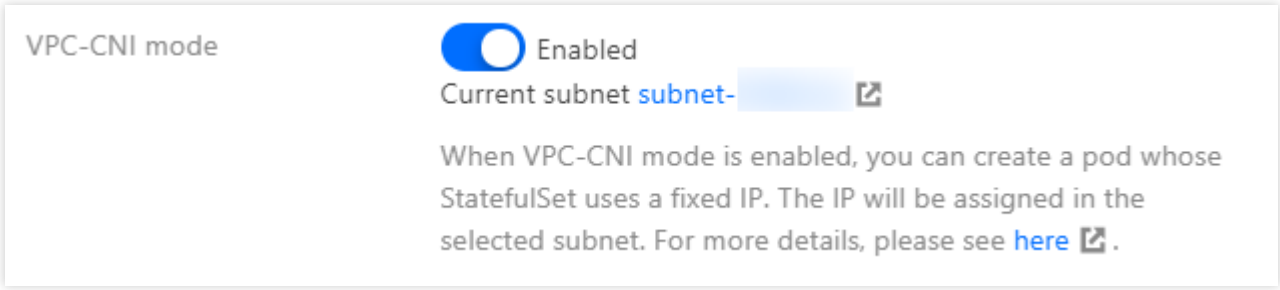
- 提高了手动维护 CLB 和 Nginx Ingress 节点的运维成本。
- 需要提前规划好 Nginx Ingress 的节点，增删 Nginx Ingress 节点时需要手动在 CLB 控制台绑定和解绑节点。
- 无法支持自动扩、缩容。

在方案3中，提出以下解决方法：

- 若网络模式是 VPC-CNI，且所有的 Pod 都使用弹性网卡，您可以使用 CLB 直接绑定弹性网卡的 Pod，即绕过 NodePort，不用手动管理 CLB且支持自动扩、缩容。如下图所示：



- 若网络模式是 Global Router，您可以在集群信息页 为集群开启 VPC-CNI 支持，即两种网络模式混用。如下图所示：



确保集群支持 VPC-CNI 之后，依次执行以下命令，即可安装 Nginx Ingress：

```
kubectl create ns nginx-ingress
```

```
kubectl apply -f https://raw.githubusercontent.com/TencentCloudContainerTeam/manifest/master/nginx-ingress/nginx-ingress-deployment-eni.yaml -n nginx-ingress
```

常见问题

如何支持内网 Ingress？

方案2：Daemonset + HostNetwork + LB 是手动管理 CLB，在自行创建 CLB 时可以选择用公网或内网。方案1：Deployment + LB 和 方案3：Deployment + LB 直通 Pod 默认创建公网 CLB。

如果要用内网，可以重新部署 YAML，给 nginx-ingress-controller 中的 Service 添加 key，例如 `service.kubernetes.io/qcloud-loadbalancer-internal-subnetid`，value 为内网 CLB 创建的子网 id 的 annotation。请参考以下代码：

```
apiVersion: v1
kind: Service
metadata:
annotations:
service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: subnet-xxxxxx # value 替换为集群所在 vpc 的其中一个子网 id
labels:
app: nginx-ingress
component: controller
name: nginx-ingress-controller
```

如何复用已有 LB？

方案1：Deployment + LB 和 方案3：Deployment + LB 直通 Pod 默认自动创建新的 CLB，Ingress 的流量入口地址取决于新创建 CLB 的 IP 地址。如果业务对入口地址有依赖，可以让 Nginx Ingress 绑定已有的 CLB。

操作方法为重新部署 YAML，给 nginx-ingress-controller 中的 Service 添加 key，例如 `service.kubernetes.io/tke-existed-lbid`，value 为 CLB ID 的 annotation。请参考以下代码：

```
apiVersion: v1
kind: Service
metadata:
annotations:
service.kubernetes.io/tke-existed-lbid: lb-6swtxxxx # value 替换为 CLB 的 ID
labels:
app: nginx-ingress
component: controller
name: nginx-ingress-controller
```

Nginx Ingress 公网带宽有多大？

腾讯云账号有标准账户和传统账户两种类型：

> ⚠ **注意：**
>
> 您可参考文档 区分腾讯云账户类型 来区分自己账号的类型。

- **标准账户类型：**指带宽上移到 CLB 或 IP 上管理。
  当您的账号是标准账户类型时，Nginx Ingress 的带宽等于已购 CLB 的带宽，默认是 10Mbps（按量计费），可按需调整。
- **传统账户类型：**指带宽在云服务器（CVM）上管理。
  当您的账号是传统账户类型时，Nginx Ingress 使用公网 CLB，Nginx Ingress 的公网带宽是 CLB 所绑定的 TKE 节点的带宽之和。如果使用 方案3：Deployment + LB 直通 Pod，CLB 直通 Pod，即 CLB 直接绑定弹性网卡，那么此时 Nginx Ingress 的公网带宽是所有 Nginx Ingress Controller Pod 被调度到的节点上的带宽之和。

如何创建 Ingress?

当您在 TKE 上自行部署 Nginx Ingress，需使用 Nginx Ingress 管理 Ingress 时，在容器服务控制台上无法创建 Ingress 时，可通过 YAML 的方式来创建 Ingress 并且需要给每个 Ingress 都指定 Ingress Class 的 annotation。请参考以下代码：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
name: test-ingress
annotations:
kubernetes.io/ingress.class: nginx # 这里是重点
spec:
```

```
rules:
- host: *
http:
paths:
- path: /
backend:
serviceName: nginx-v1
servicePort: 80
```

如何监控？

通过 如何创建 Ingress 安装的 Nginx Ingress，已经暴露了 metrics 端口，可以被 Prometheus 采集。如果集群内安装了 prometheus-operator，可以使用 ServiceMonitor 来采集 Nginx Ingress 的监控数据。请参考以下代码：

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
name: nginx-ingress-controller
namespace: nginx-ingress
labels:
app: nginx-ingress
component: controller
spec:
endpoints:
- port: metrics
interval: 10s
namespaceSelector:
matchNames:
- nginx-ingress
selector:
matchLabels:
app: nginx-ingress
component: controller
```

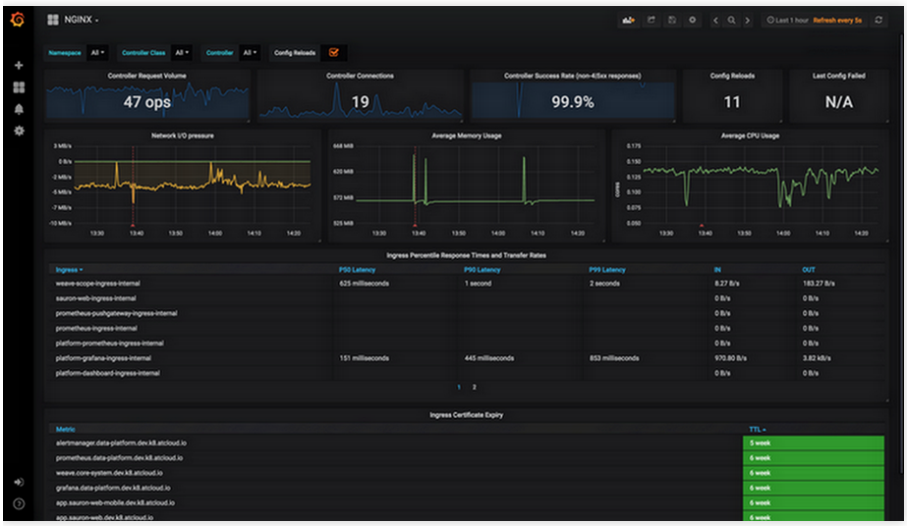
原生 Prometheus 配置请参考以下代码：

```
- job_name: nginx-ingress
scrape_interval: 5s
kubernetes_sd_configs:
- role: endpoints
namespaces:
names:
- nginx-ingress
relabel_configs:
- action: keep
source_labels:
```

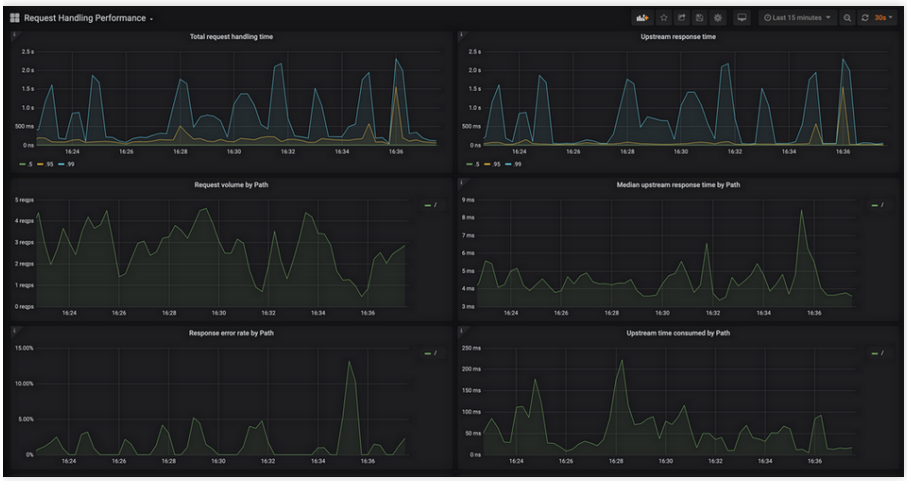
```
- __meta_kubernetes_service_label_app
- __meta_kubernetes_service_label_component
regex: nginx-ingress;controller
- action: keep
source_labels:
- __meta_kubernetes_endpoint_port_name
regex: metrics
```

采集监控数据后，可为 grafana 配置 Nginx Ingress 社区提供的面板，并展示数据。

实际操作中，直接复制 json 导入 grafana，即可导入面板。其中，`nginx.json` 是展示 Nginx Ingress 各种常规监控的面板。如下图所示：



`request-handling-performance.json` 是展示 Nginx Ingress 性能方面的监控面板。如下图所示：



参考资料

- TKE Service YAML 示例
- TKE Service 使用已有 CLB
- 区分腾讯云账户类型

Nginx Ingress 高并发实践

最近更新时间：2022-08-02 10:01:07

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

您可通过以下方式对 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
- 扩大源端口范围
- TIME_WAIT 复用
- 调大最大文件句柄数
- 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

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：2023-05-06 17:36:46

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**组件管理**，进入组件列表页面。
4. 单击已安装好的 Nginx-ingress 扩展组件，进入组件页面。
5. 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。

6. 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：

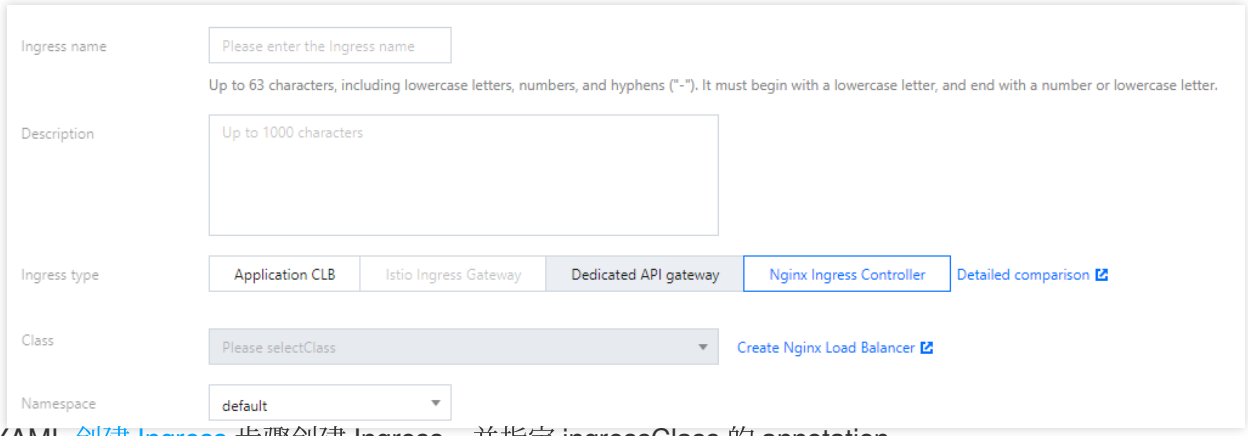
通过控制台创建 Ingress

通过 YAML 创建 Ingress

参考控制台 创建 Ingress 步骤创建 Ingress。其中：

Ingress类型：选择**Nginx均衡负载器**。

Class：选择上述步骤创建的 Nginx Ingress 实例。



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：

```
1 apiVersion: networking.k8s.io/v1beta1
2 kind: Ingress
3 metadata:
4   annotations:
5     ingress.cloud.tencent.com/direct-access: "false"
6     kubernetes.io/ingress.class: nginx-external
```

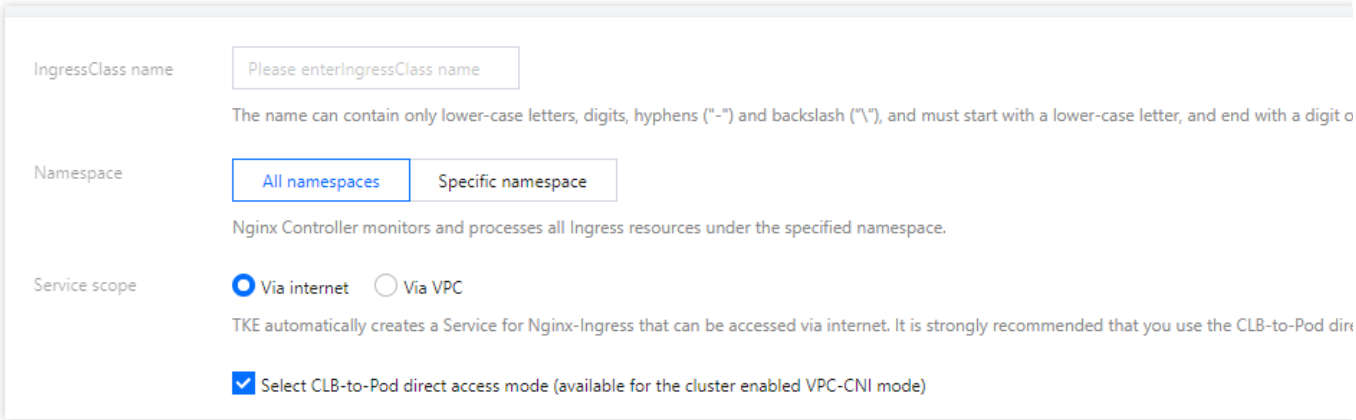
性能优化

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：

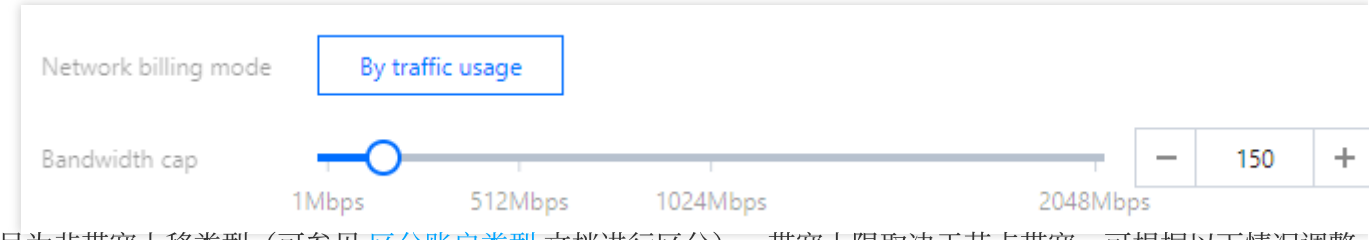


说明

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。

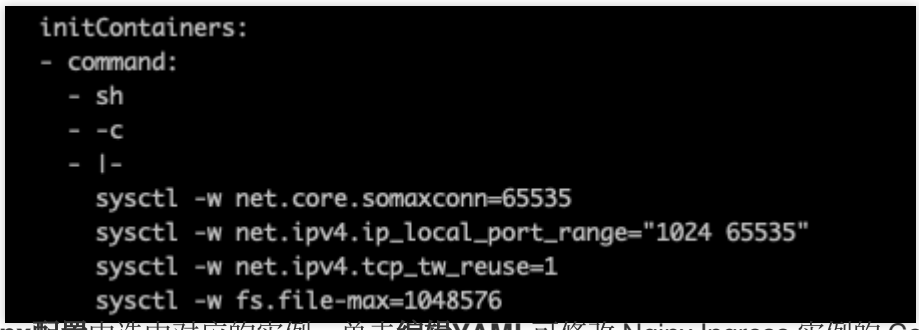
优化 Nginx Ingress 参数

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

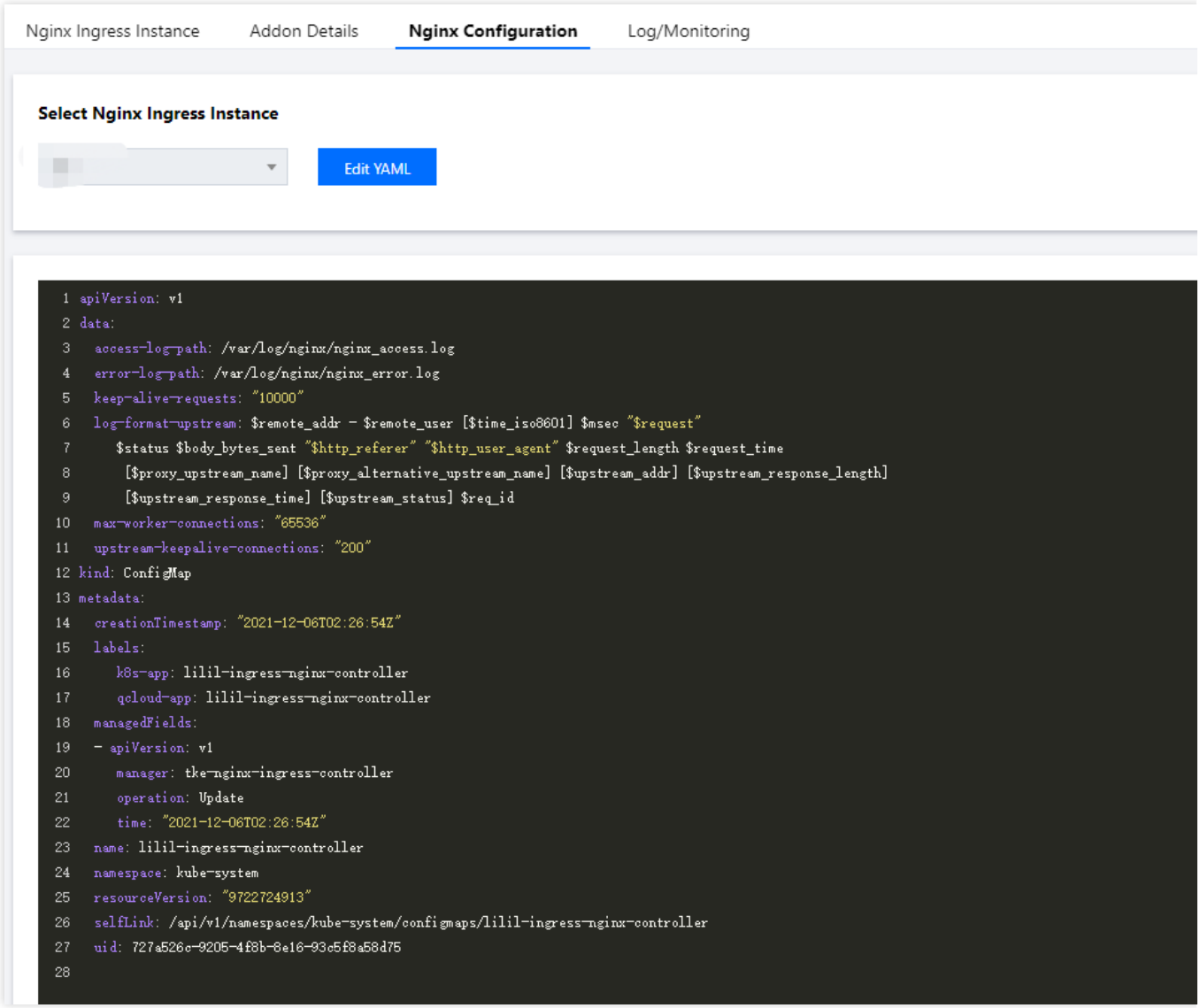
修改 Nginx Ingress 自身配置

编辑部署好的 nginx-ingress-conntroller 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
```

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。

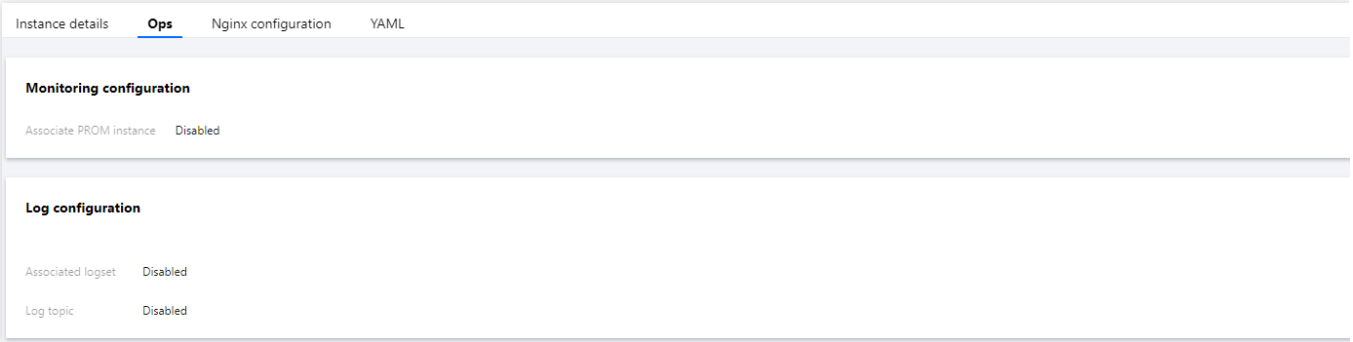
提升 Nginx Ingress 可观测性

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/再打开改日志采集功能，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，请按照文档进行配置。

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**前往CLS查看访问日志**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**查看访问日志仪表盘**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。

自建 Nginx Ingress 实践教程
快速开始

最近更新时间：2024-08-12 17:49:23

概述

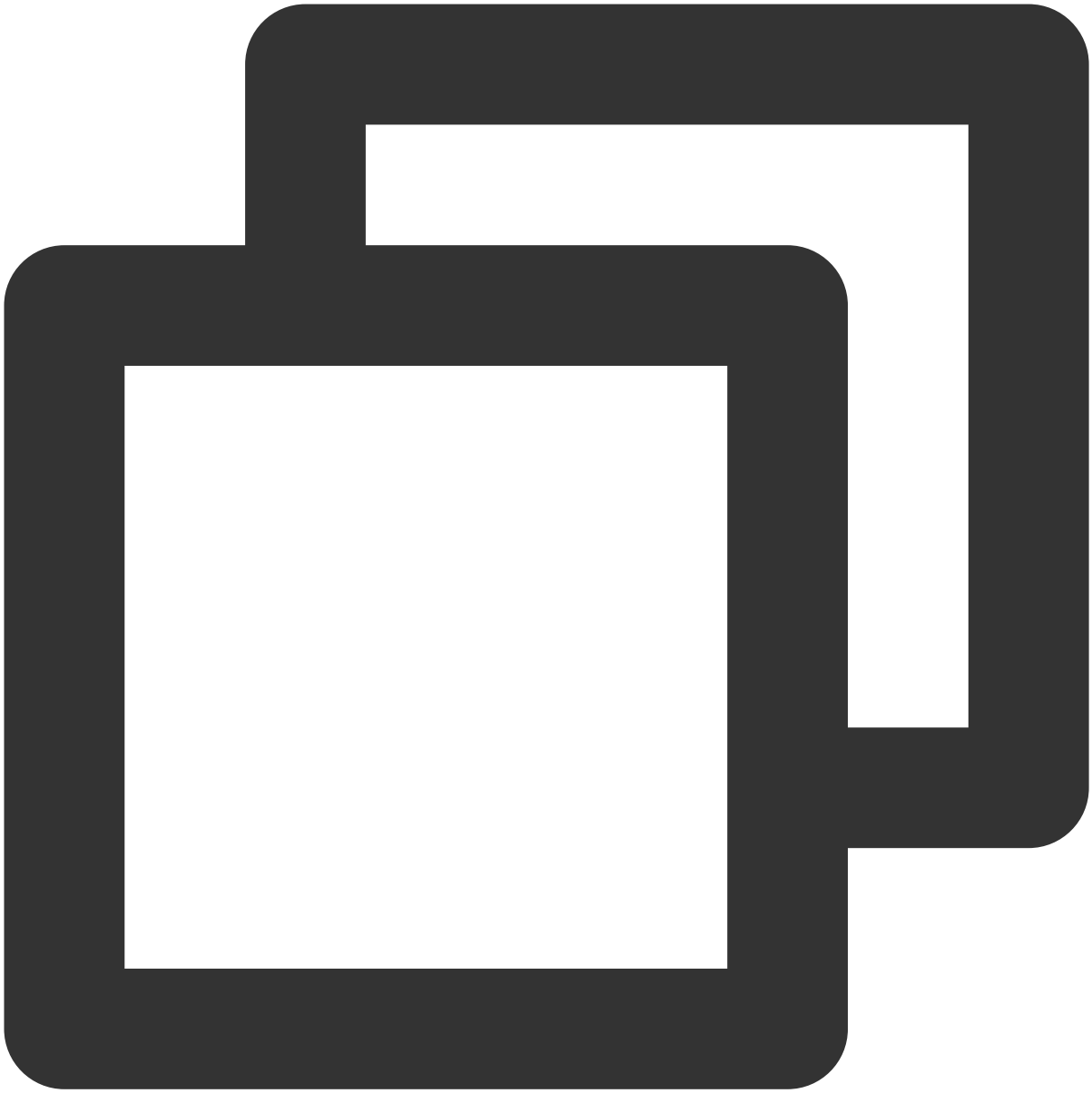
Nginx Ingress Controller 是基于高性能 NGINX 反向代理实现的 Kubernetes Ingress 控制器，也是最常用的开源 Ingress 实现。本文介绍如何在 TKE 环境中自建 Nginx Ingress Controller，主要使用 helm 进行安装，提供一些 `values.yaml` 配置指引。

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。

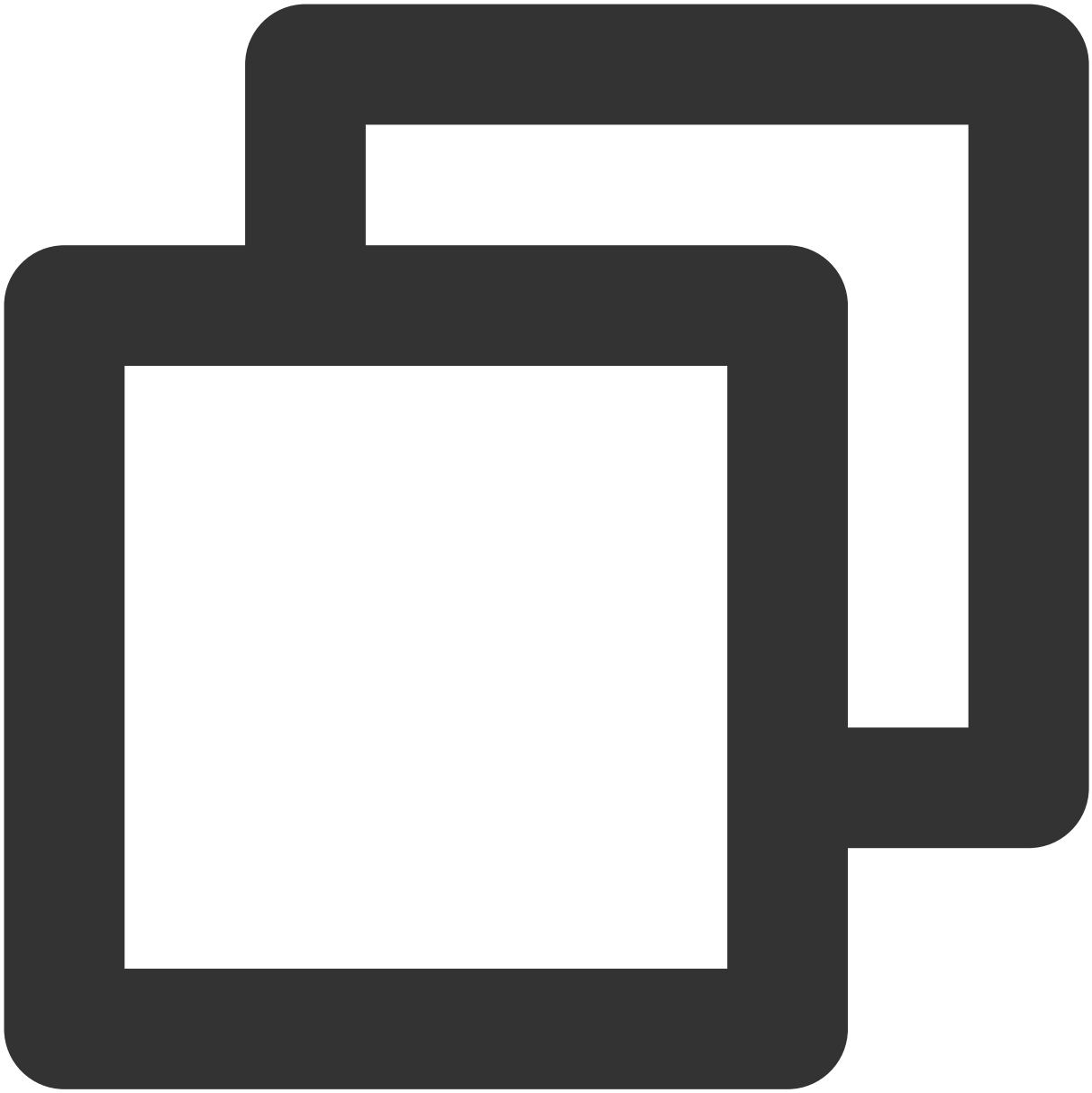
使用 helm 安装

添加 helm repo：



```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

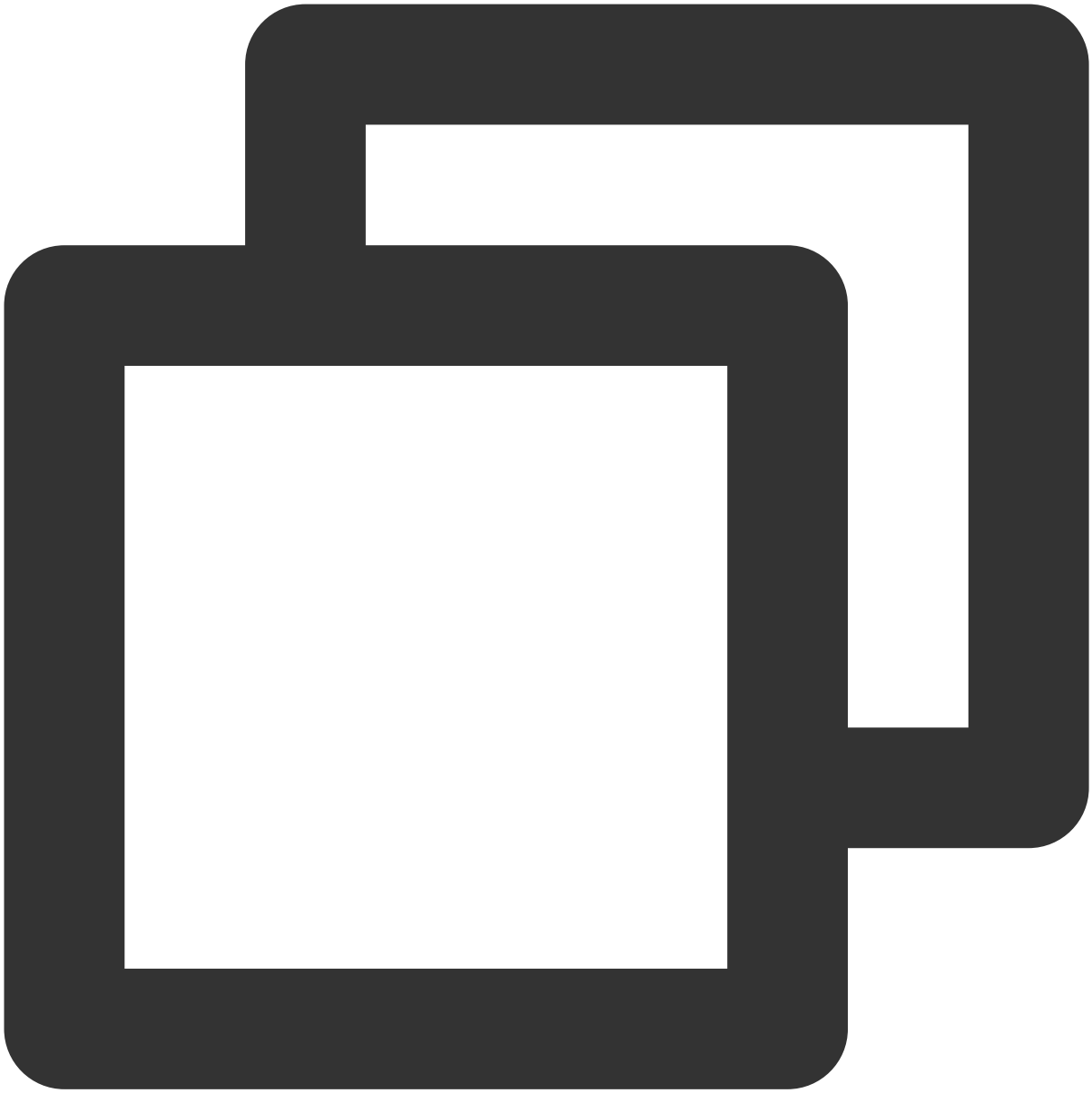
查看默认配置：



```
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` ：



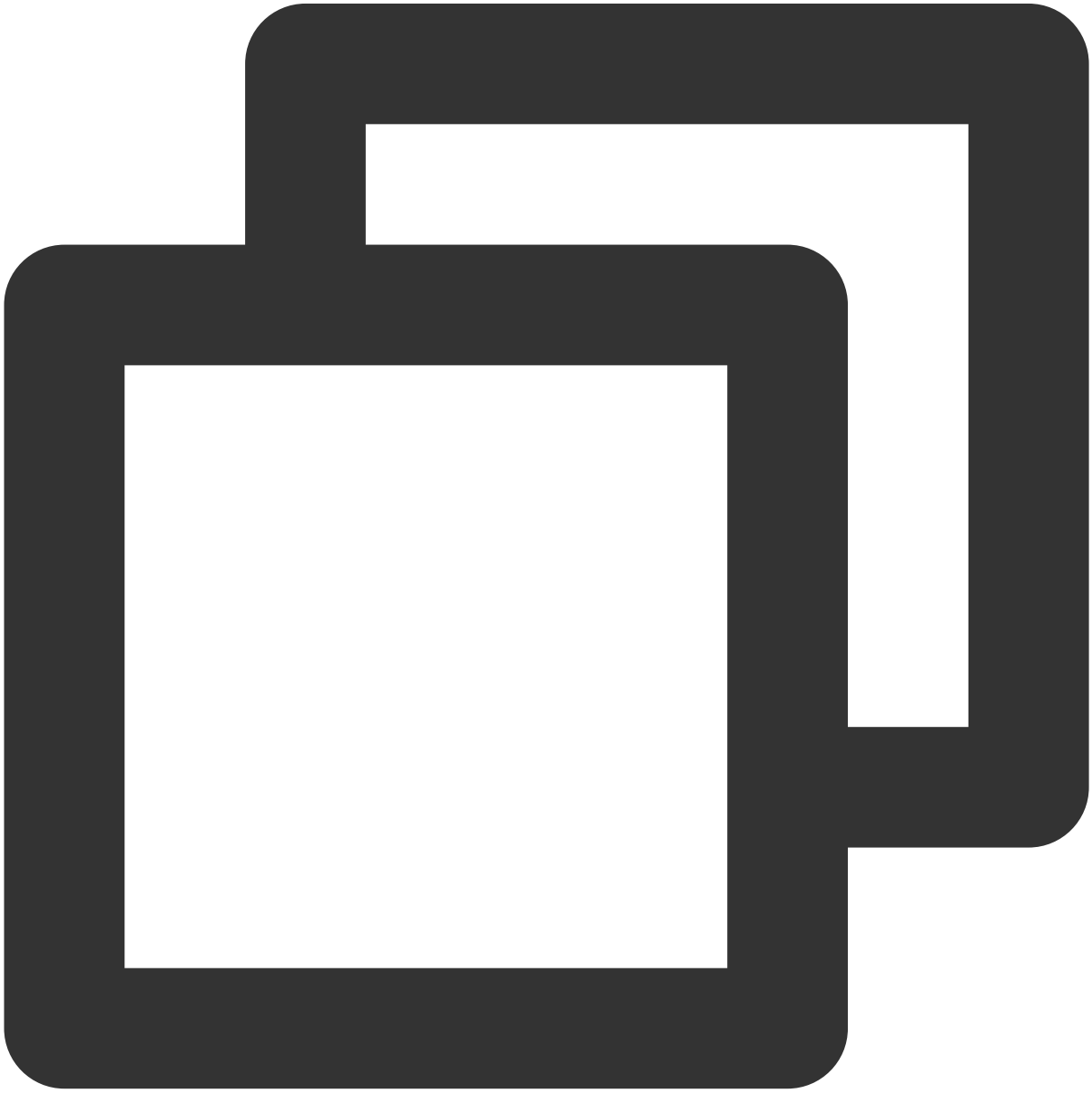
```
controller: # 以下配置将依赖镜像替换为了 docker hub 上的 mirror 镜像以保证在国内环境能正常拉取
  image:
    registry: docker.io
    image: k8smirror/ingress-nginx-controller
  admissionWebhooks:
    patch:
      image:
        registry: docker.io
        image: k8smirror/ingress-nginx-kube-webhook-certgen
  defaultBackend:
    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：

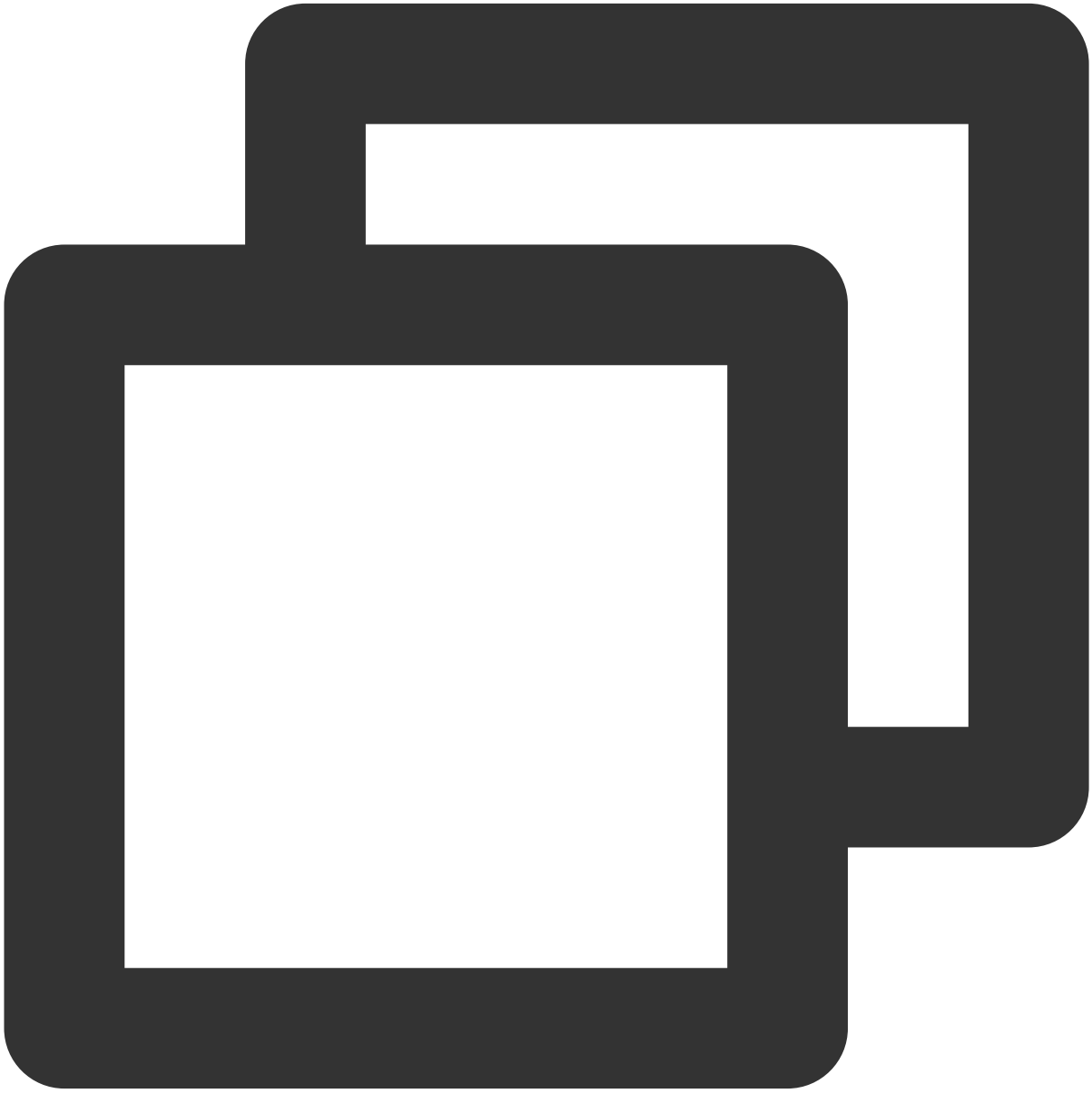


```
helm upgrade --install ingress-nginx ingress-nginx/ingress-nginx \\
  --namespace ingress-nginx --create-namespace \\
  -f values.yaml
```

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
```

```
ingress-nginx-controller             LoadBalancer   xxx.xx.xxx.xxx   xxx.xx.xx.xxx
ingress-nginx-controller-admission   ClusterIP      xxx.xx.xxx.xxx   <none>
```

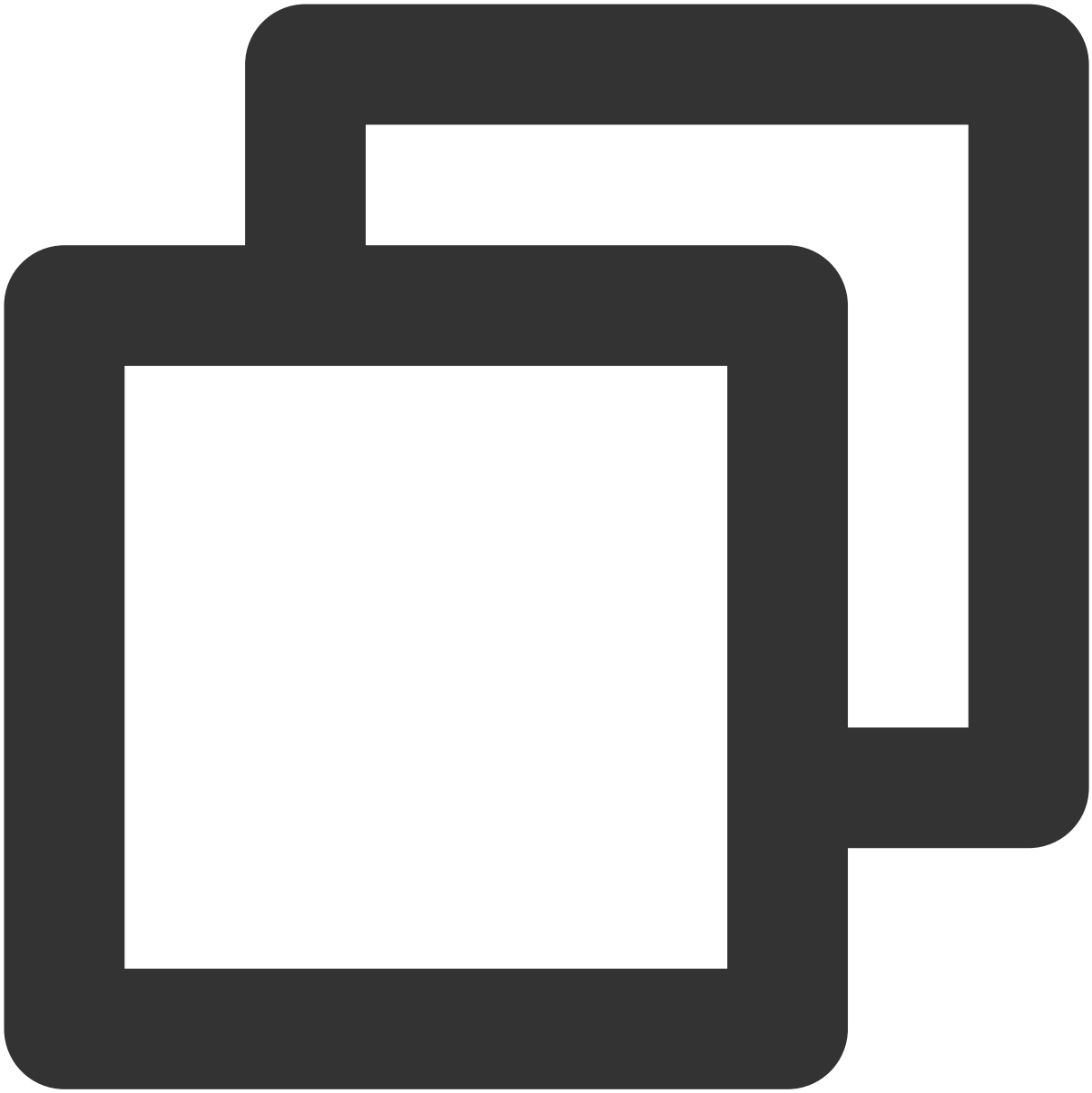
说明：

`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。

版本与升级

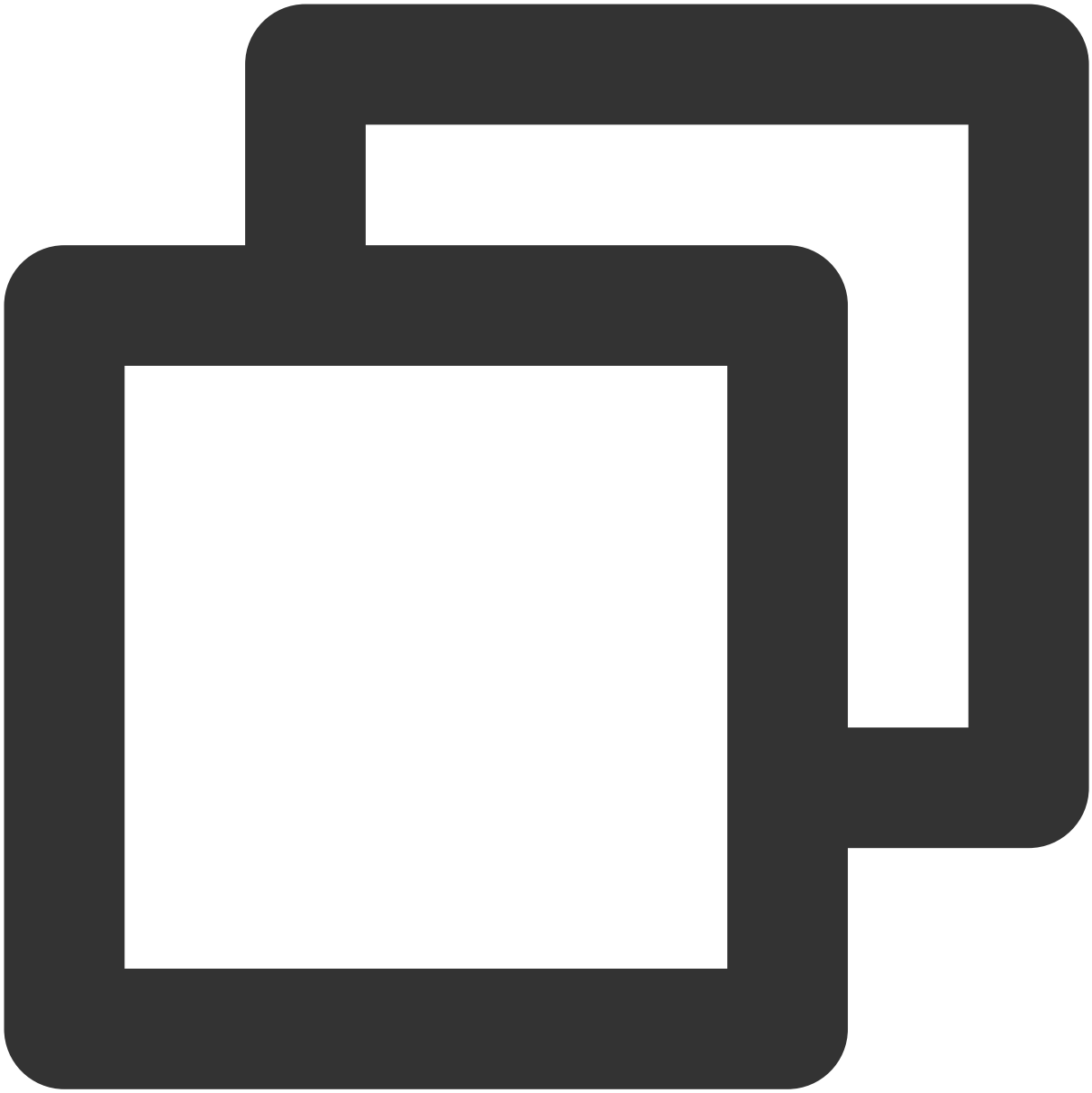
Nginx Ingress 的版本需要与 Kubernetes 集群版本能够兼容，可参考官方 Supported Versions table 确认当前集群版本能否支持最新的 nginx ingress，如果不支持，安装的时候需指定 chart 版本。

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
ingress-nginx/ingress-nginx     4.7.3           1.8.4           Ingress controller
ingress-nginx/ingress-nginx     4.7.2           1.8.2           Ingress controller
ingress-nginx/ingress-nginx     4.7.1           1.8.1           Ingress controller
ingress-nginx/ingress-nginx     4.7.0           1.8.0           Ingress controller
```

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
```

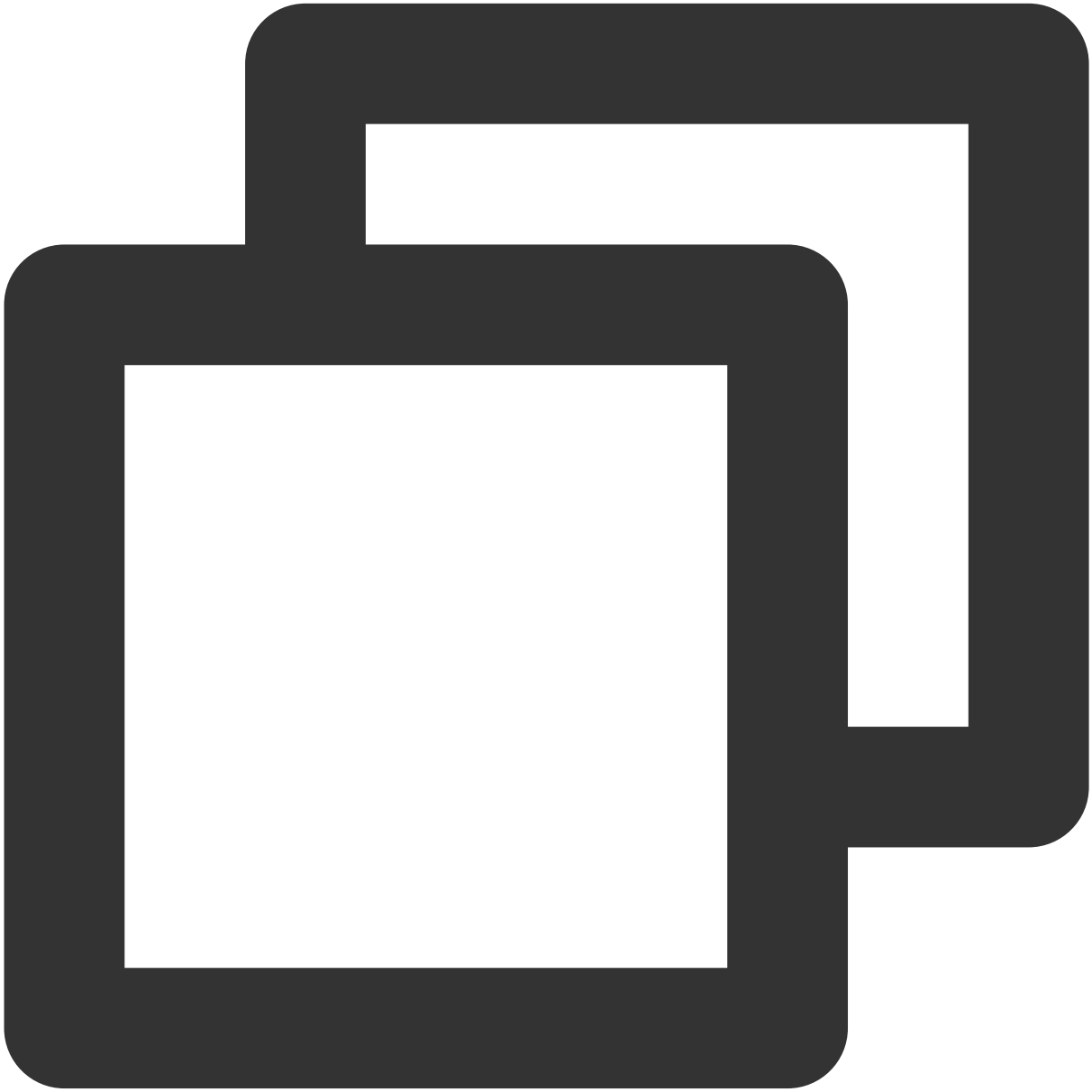
注意：

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，根据自己需求合并下 `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。

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

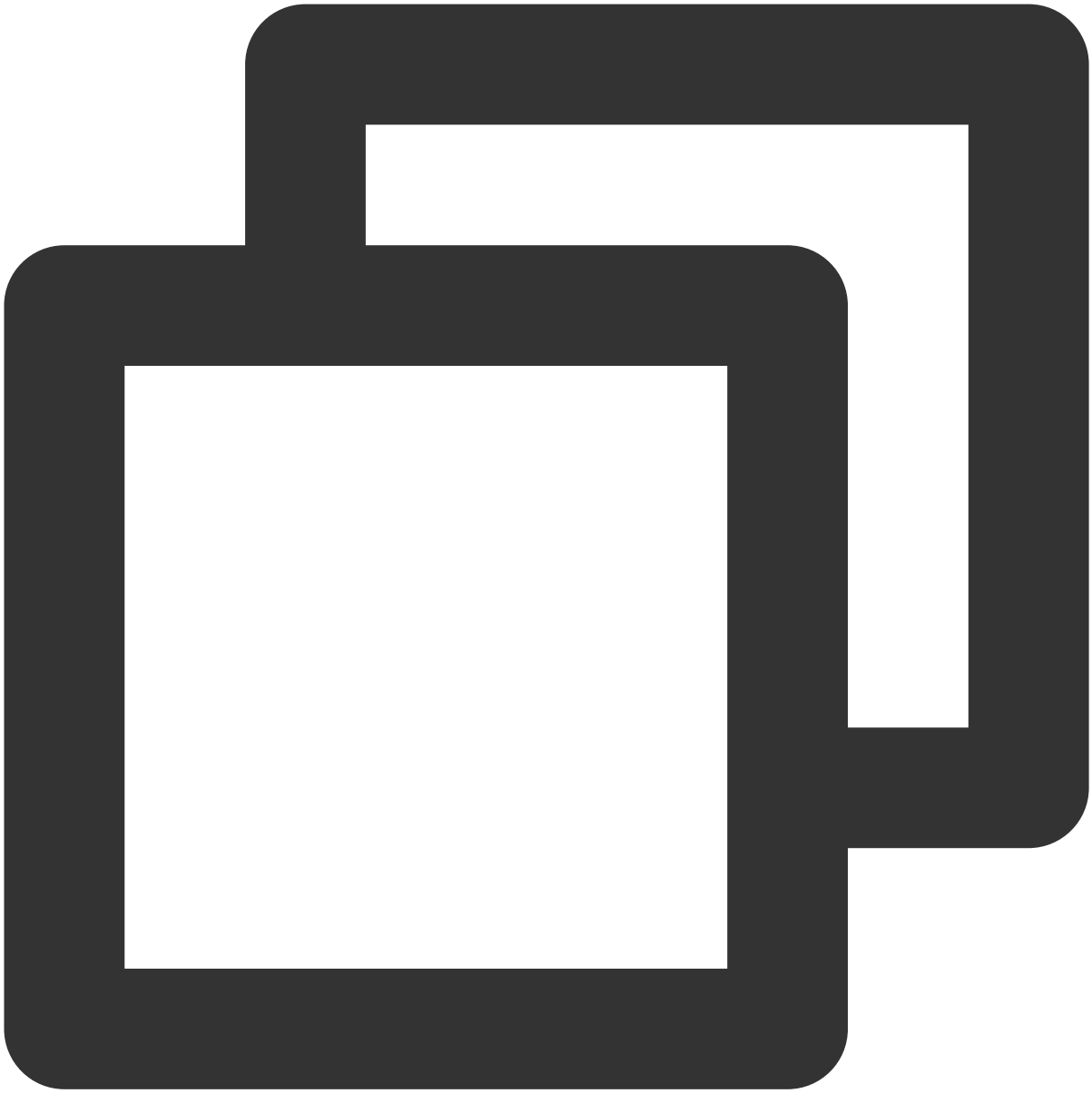
最近更新时间：2024-08-12 17:49:23

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

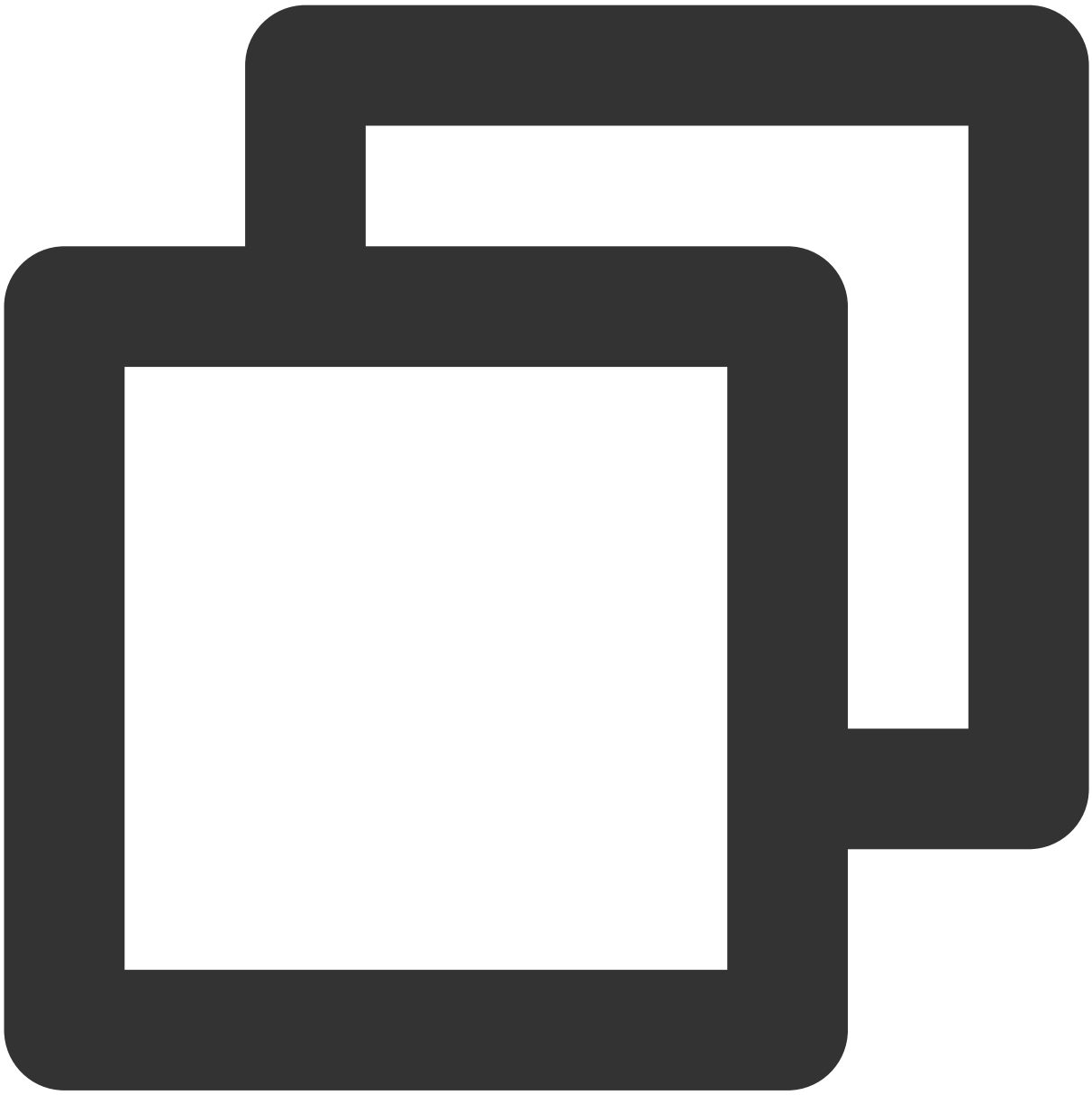
例如改成内网 CLB，在 `values.yaml` 中的示例代码如下：



```
controller:
  service:
    annotations:
      service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: 'subnet-xxxxxx'
```

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。



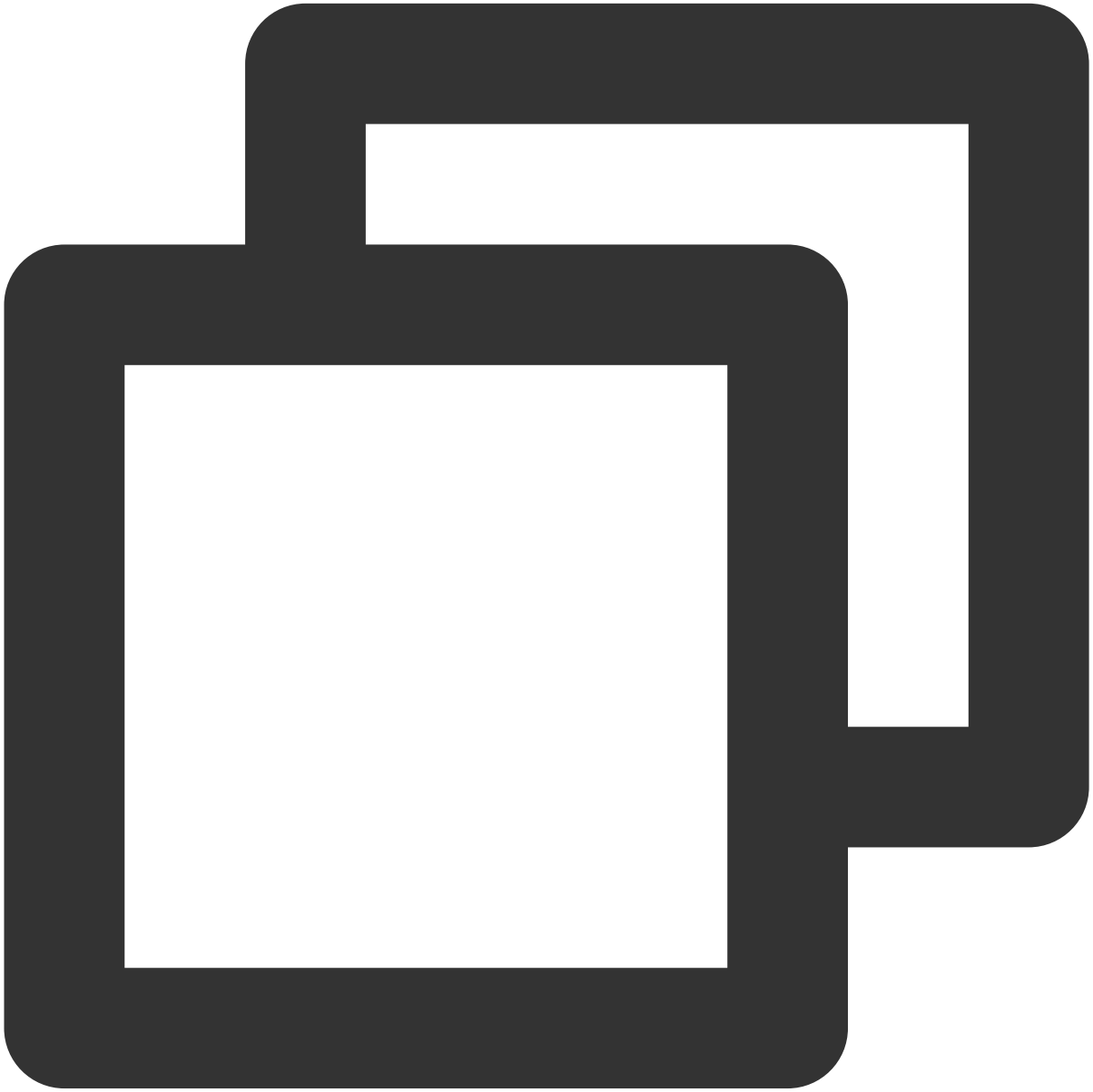
```
controller:
  service:
    annotations:
      service.kubernetes.io/tke-existed-lbid: 'lb-xxxxxxxx' # 指定已有 CLB 的实例 ID
```

注意：

在 CLB 控制台创建 CLB 实例时，选择的 VPC 需与集群一致。

使用内外网双 CLB

如果您需要让 nginx ingress 同时使用公网和内网 CLB 接入流量，您可以配置 nginx ingress 使用两个 service。默认情况下，会创建一个公网 CLB Service。如果您还需要一个内网 CLB 的 Service，可以按照以下步骤配置 internal service：



```
controller:
  service:
    internal:
      enabled: true # 创建内网 CLB Service
```

```
    annotations:
      service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: "subnet-xxxxxx
```

启用 CLB 直连

最近更新时间：2024-08-12 17:49:23

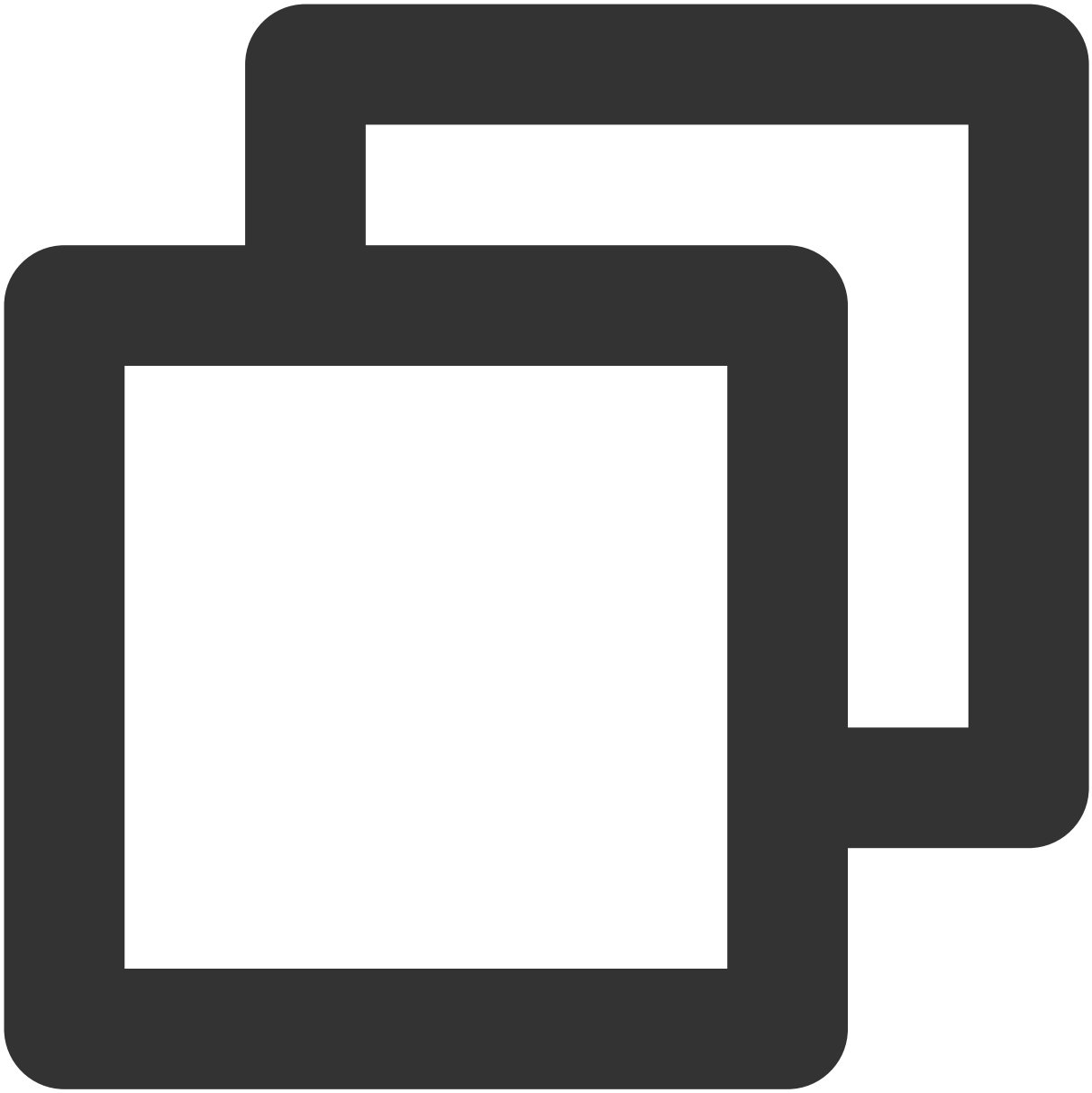
概述

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

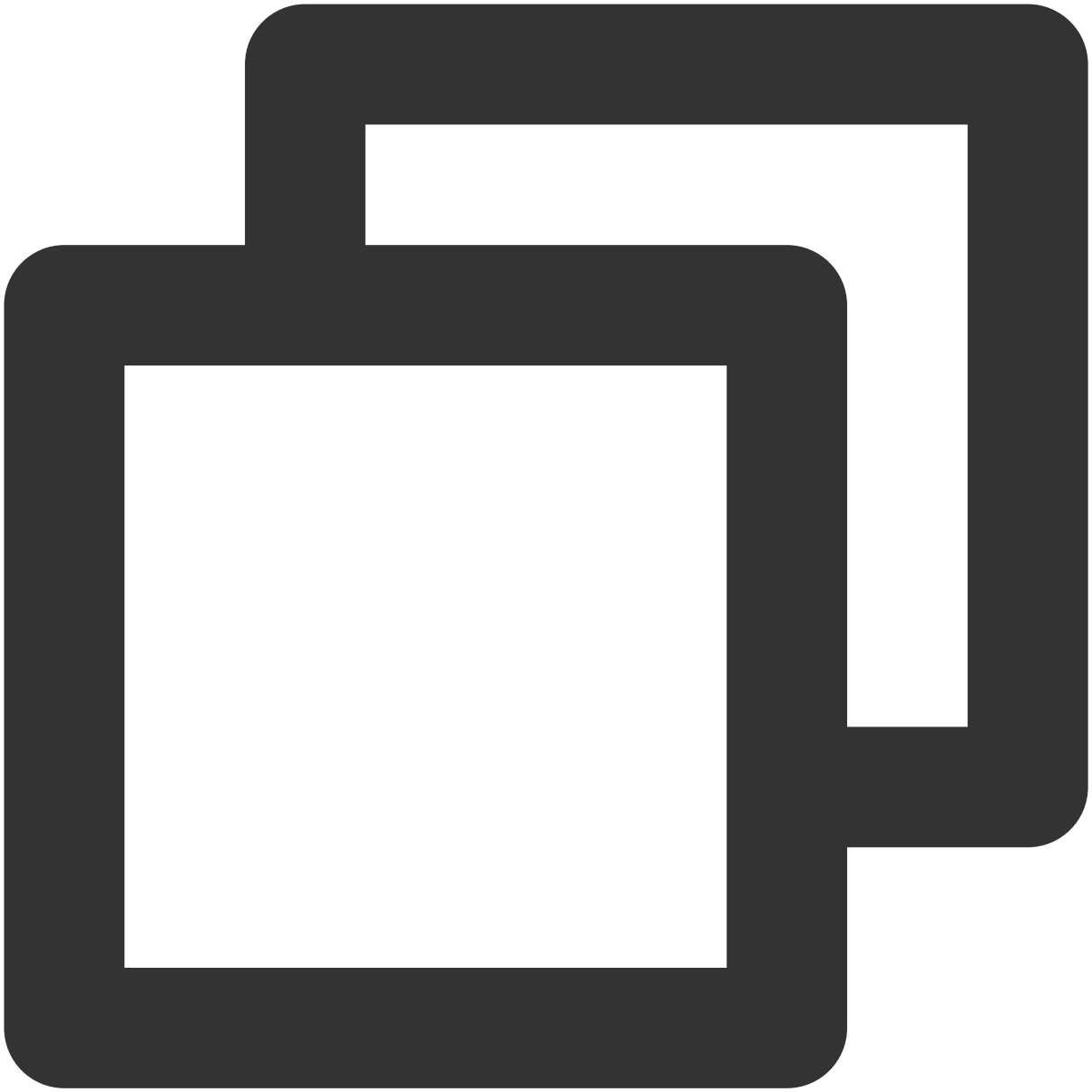
如果集群网络是 GlobalRouter，但没有启用 VPC-CNI，建议为集群开启 VPC-CNI，详情见 GlobalRouter + VPC-CNI 网络模式启用直连 启用 CLB 直连。

如果不希望开启 VPC-CNI，可以根据以下步骤启用直连，但是需接受 使用限制。

注意：

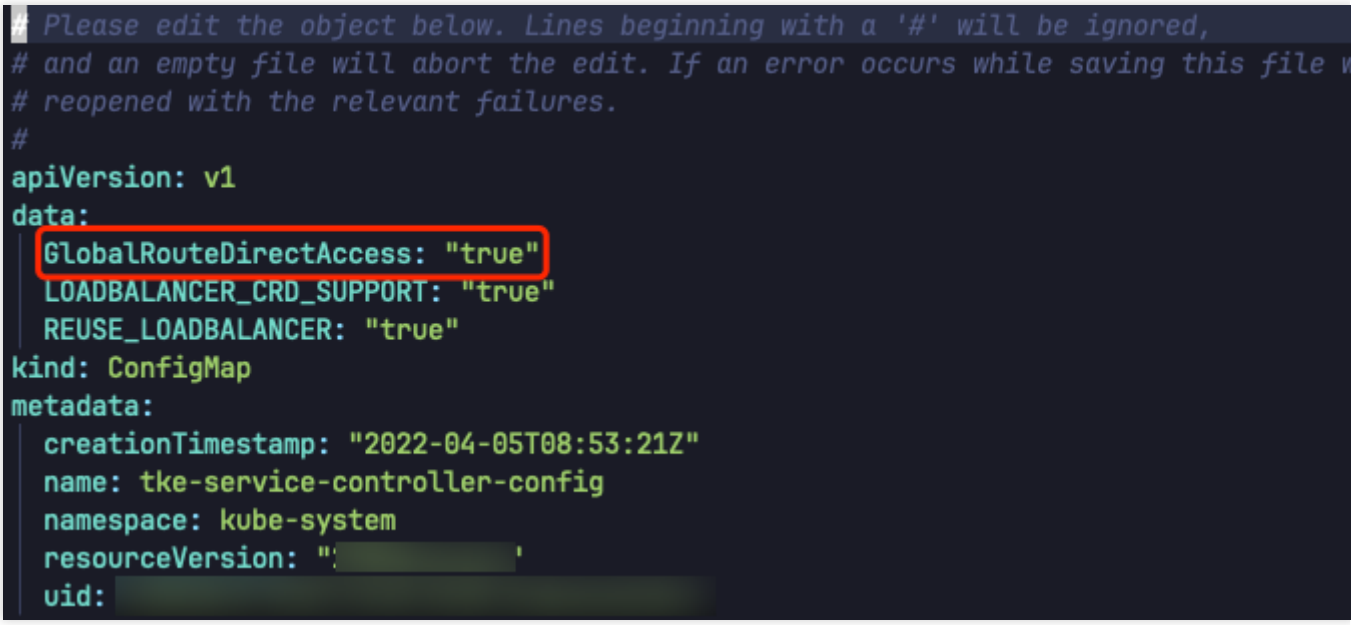
请确认您的账号满足上述条件，并接受使用限制。

1. 修改 configmap 开启 GlobalRouter 集群维度的直连能力：

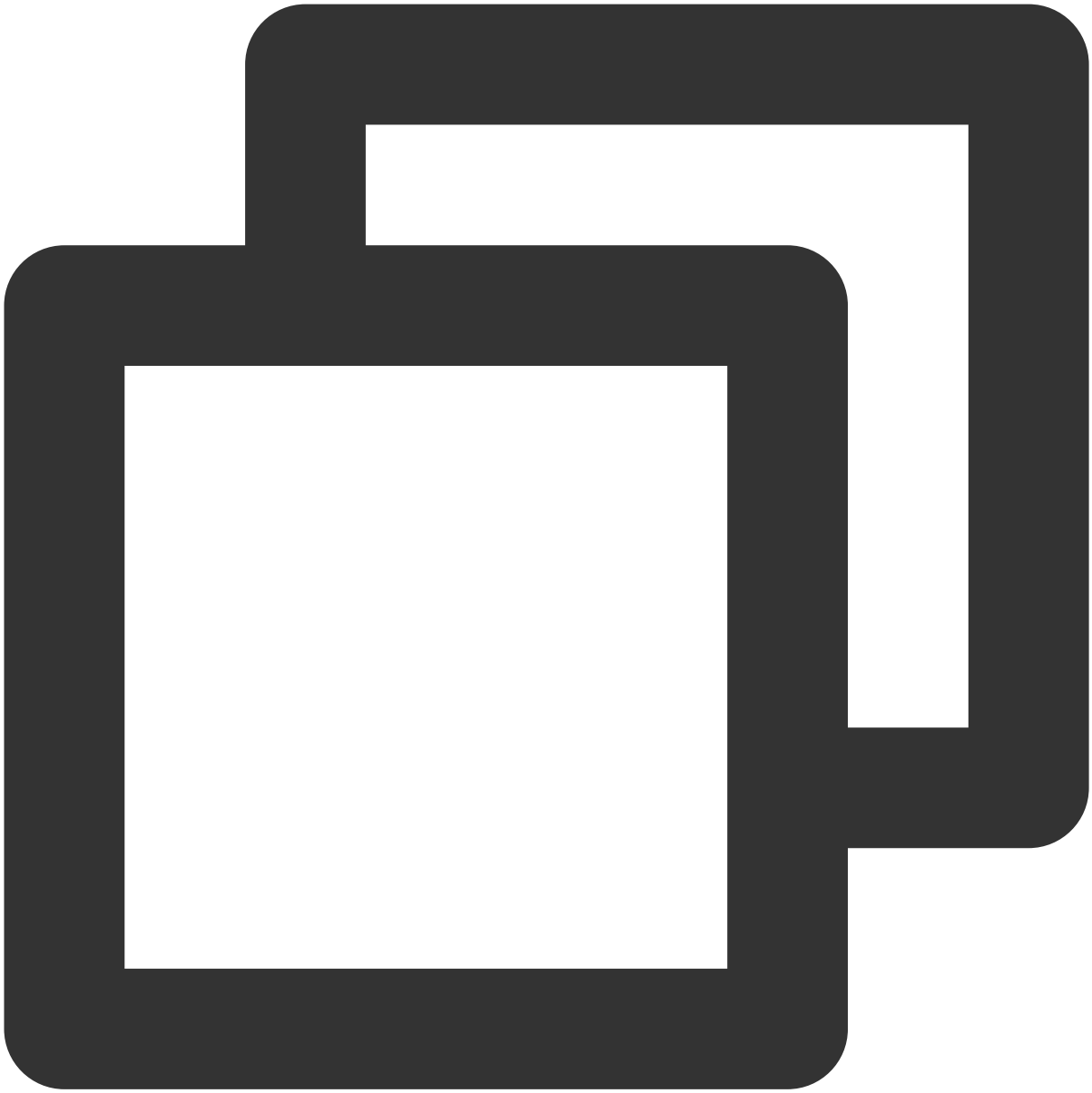


```
kubectl edit configmap tke-service-controller-config -n kube-system
```

将 `GlobalRouteDirectAccess` 置为 true：



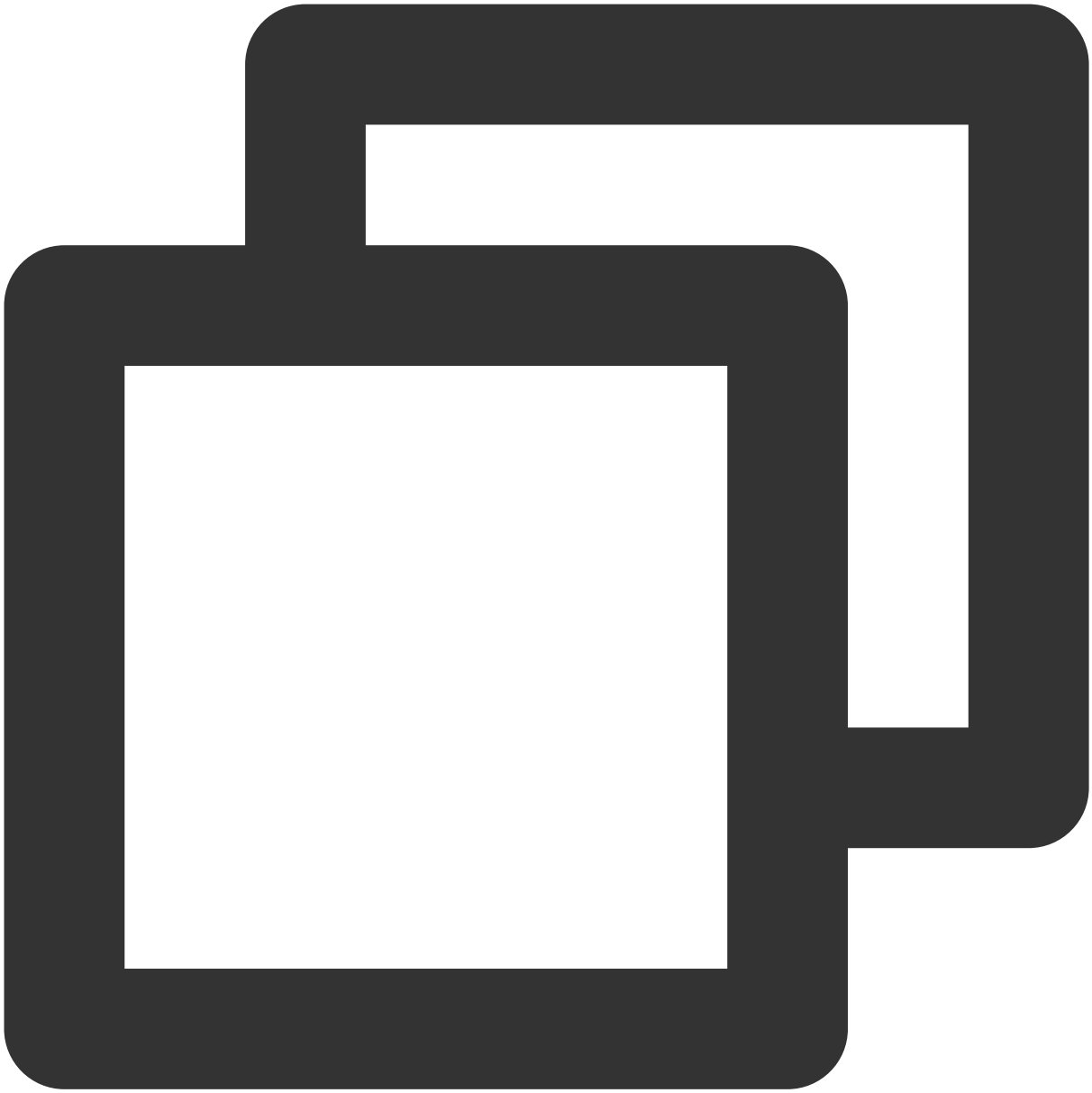
2. 配置 `values.yaml` 启用 CLB 直连：



```
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：2024-08-12 17:49:23

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。

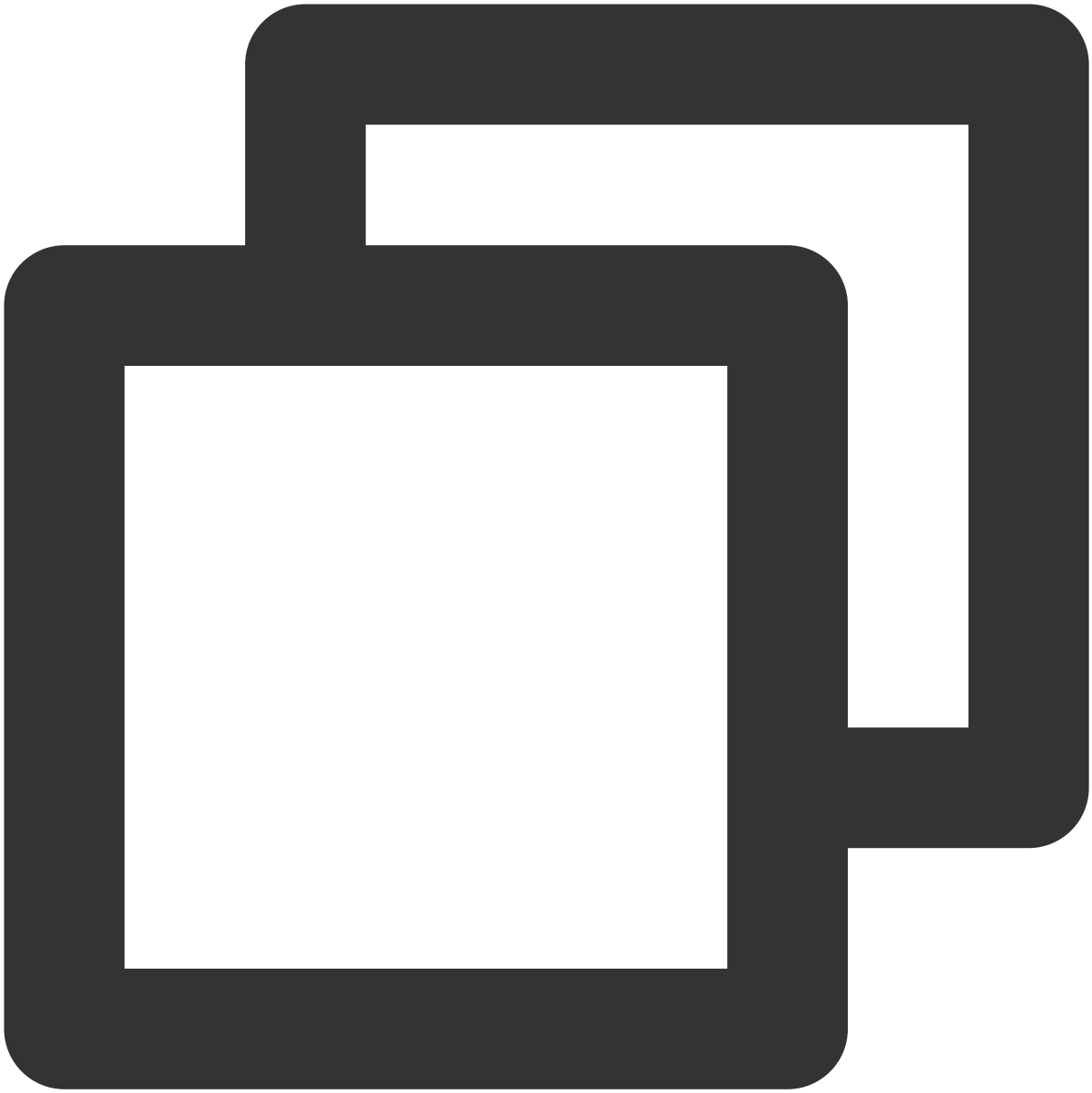
操作指南

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
            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
      # 参考: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuratic
      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
      # 参考: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuratic
      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-configuratic
      max-worker-connections: "65536"
```

说明：

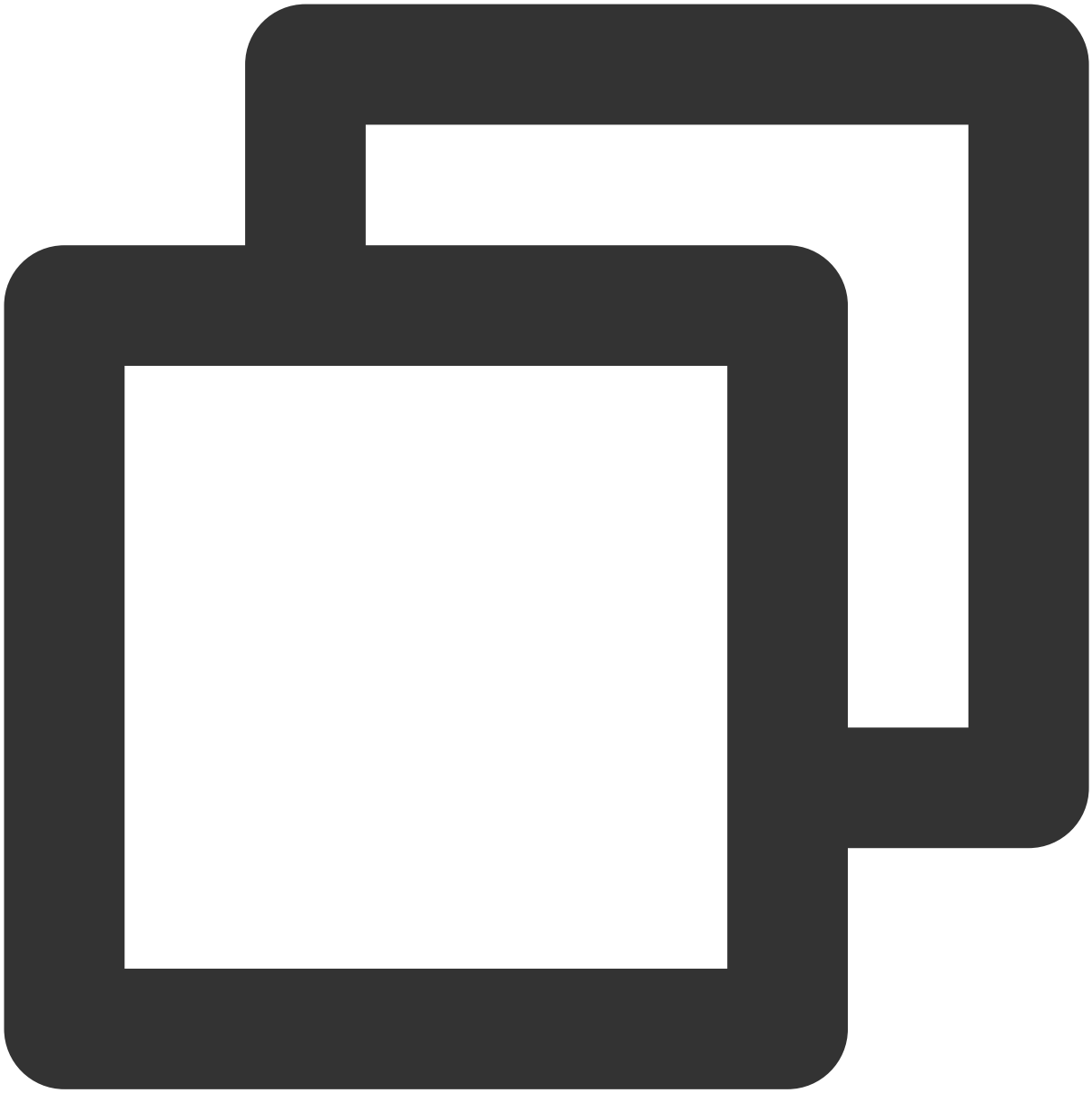
请参见 Nginx Ingress 高并发实践。

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
          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

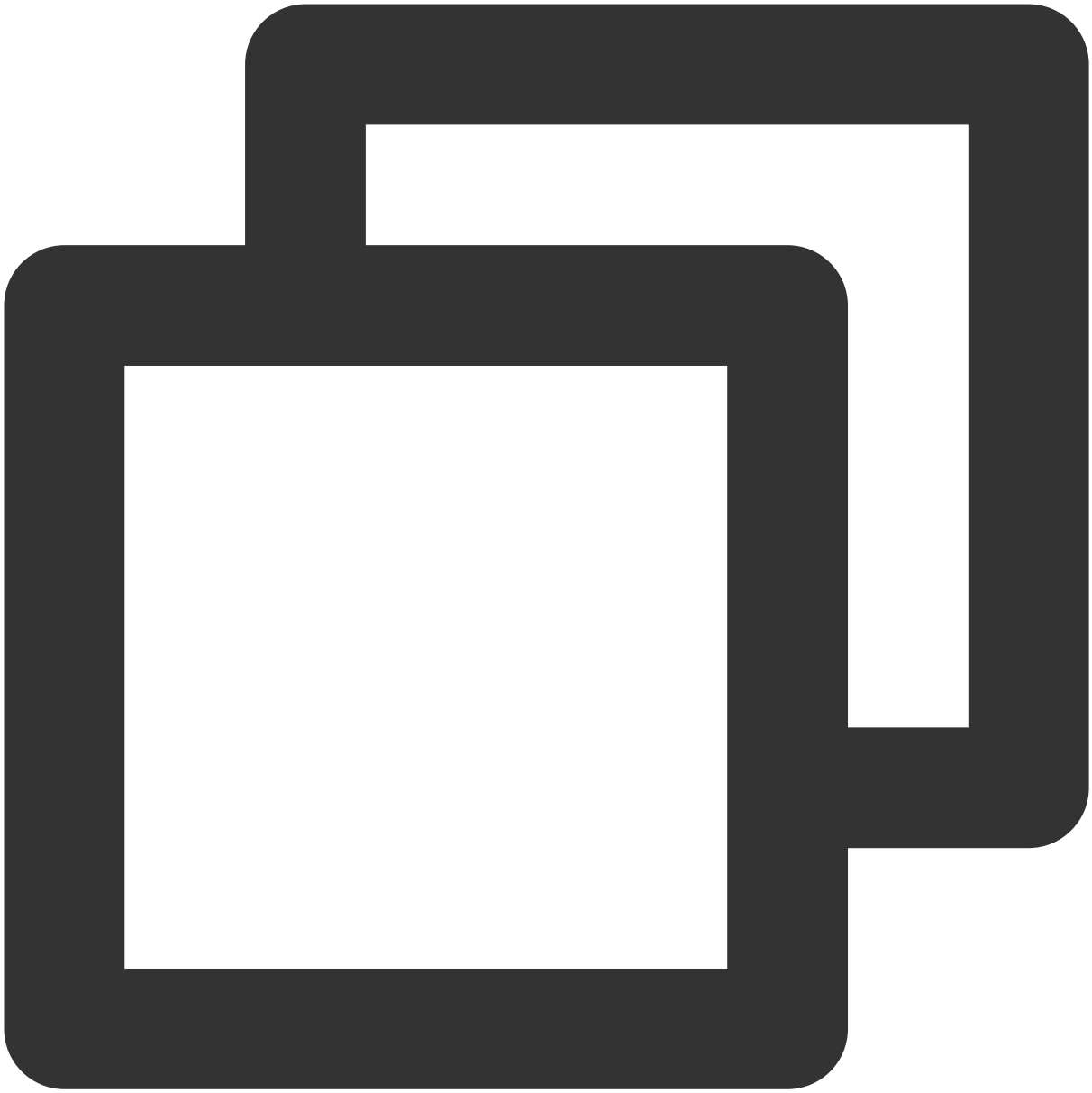
最近更新时间：2024-08-12 17:49:23

概述

本文介绍 Nginx Ingress 的高可用部署配置方法。

调高副本数

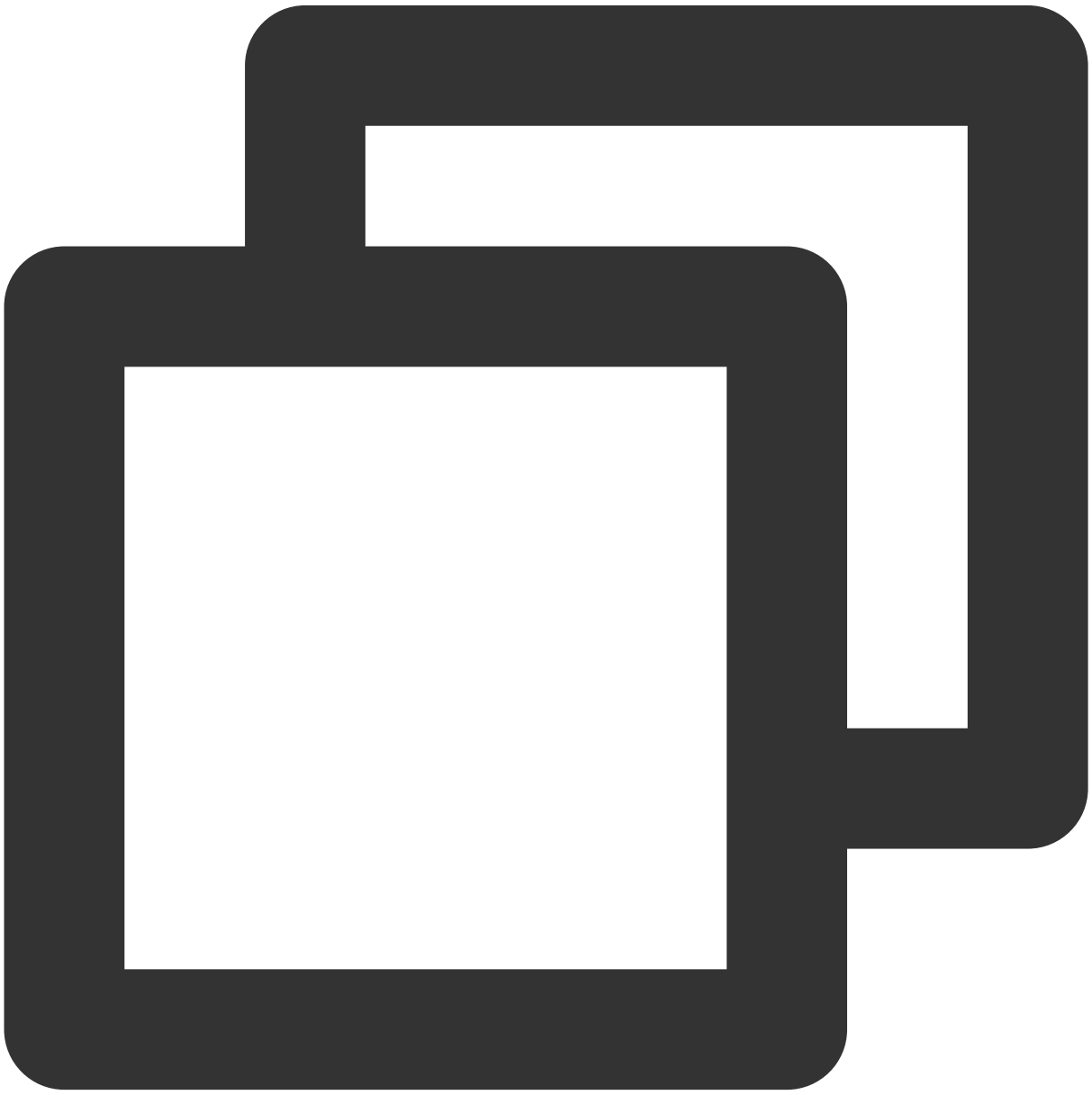
配置自动扩缩容：



```
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
```

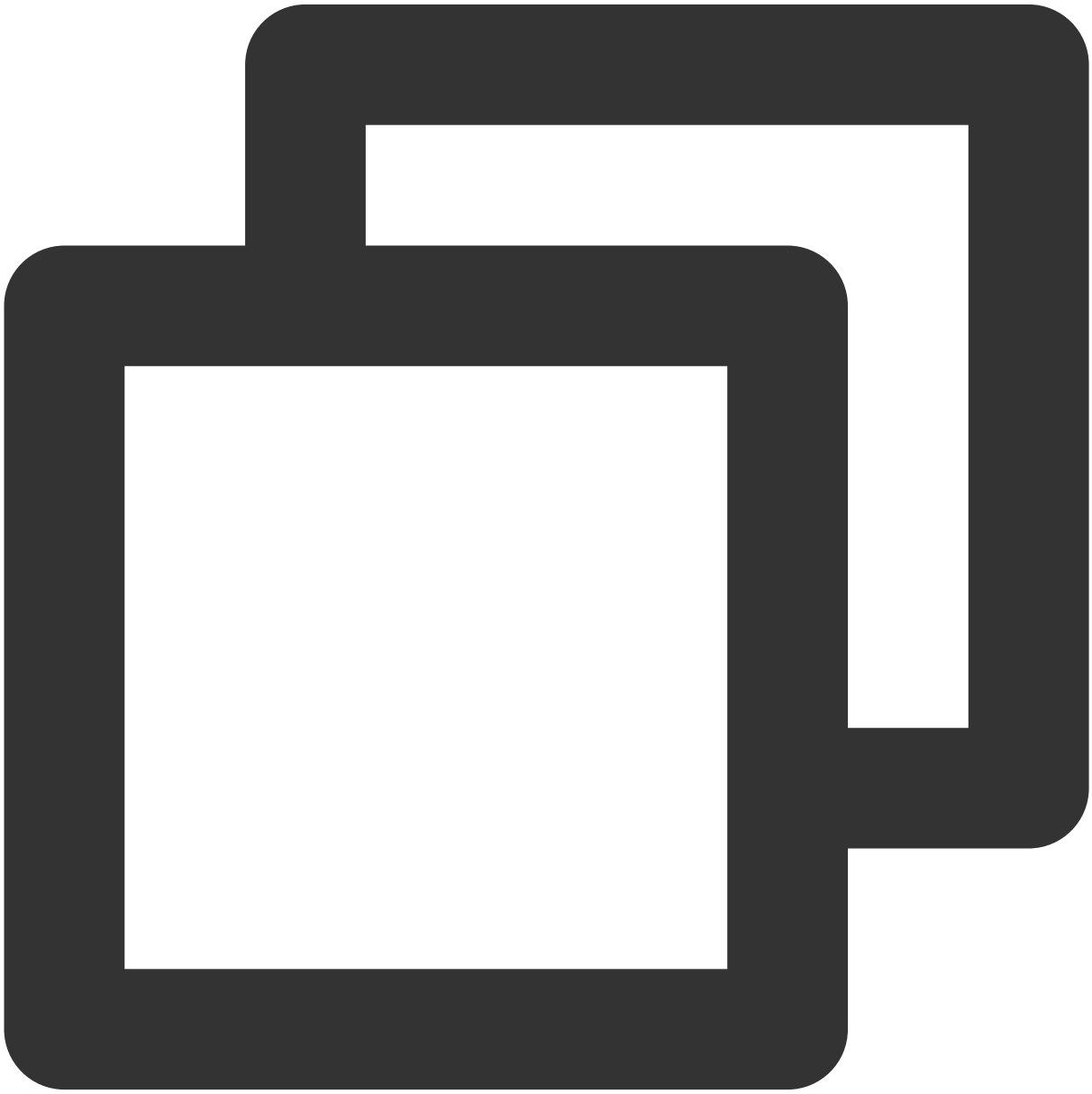
如果希望固定副本数，直接配置 `replicaCount`：



```
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：

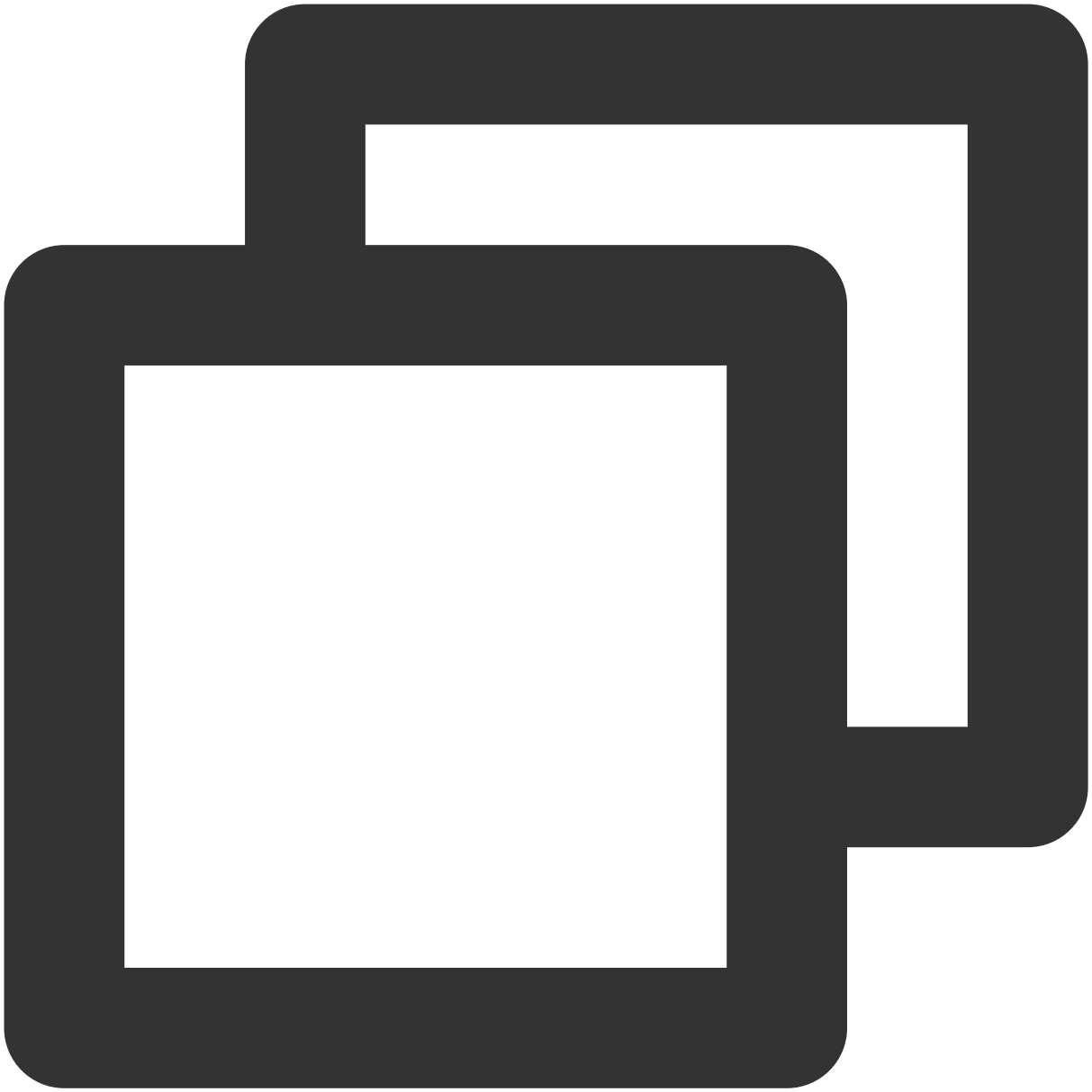


```
controller:
  topologySpreadConstraints: # 尽量打散的策略
```

```
    - 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：



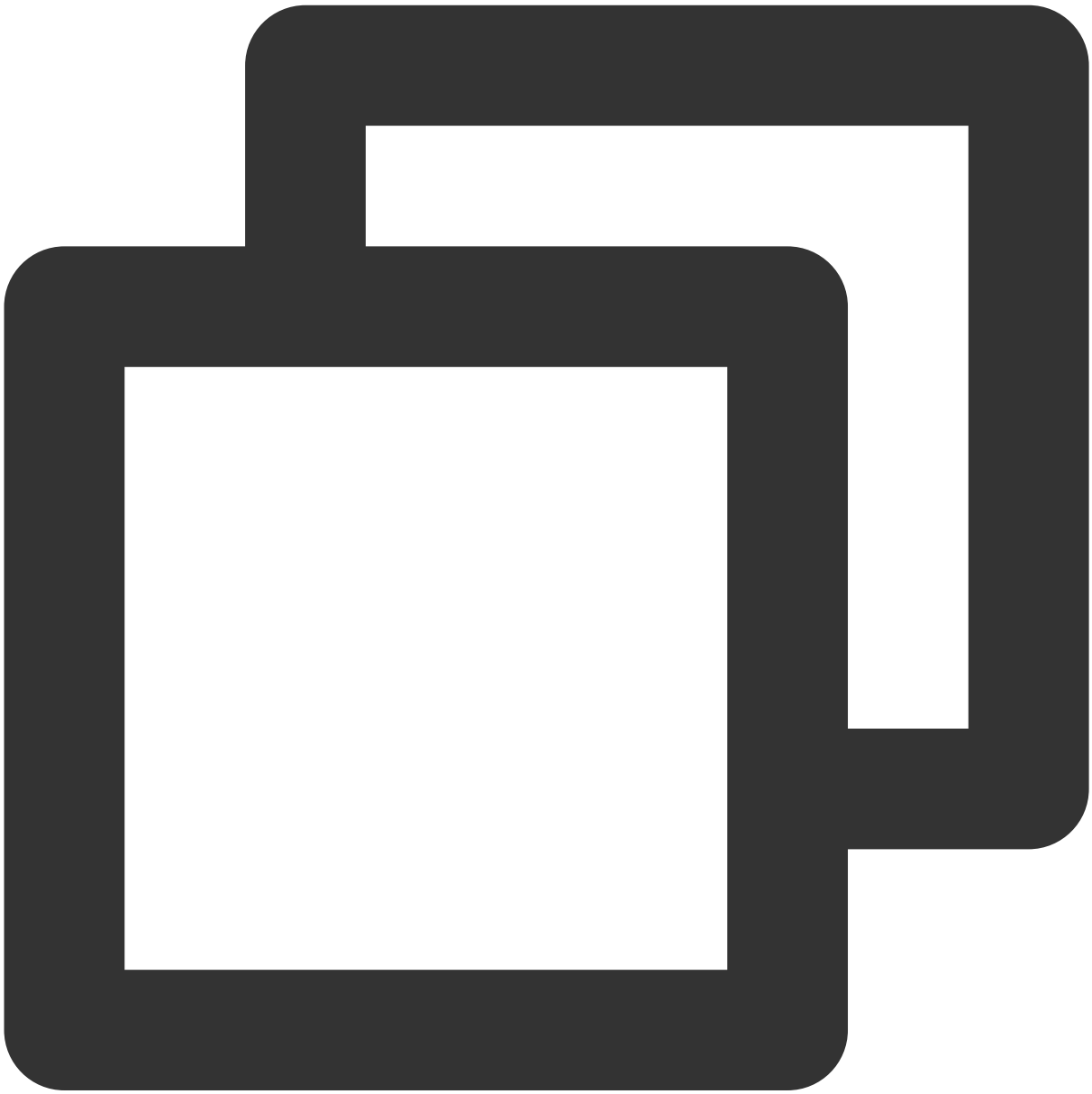
```
controller:
  nodeSelector:
    tke.cloud.tencent.com/nodepool-id: np-********
```

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

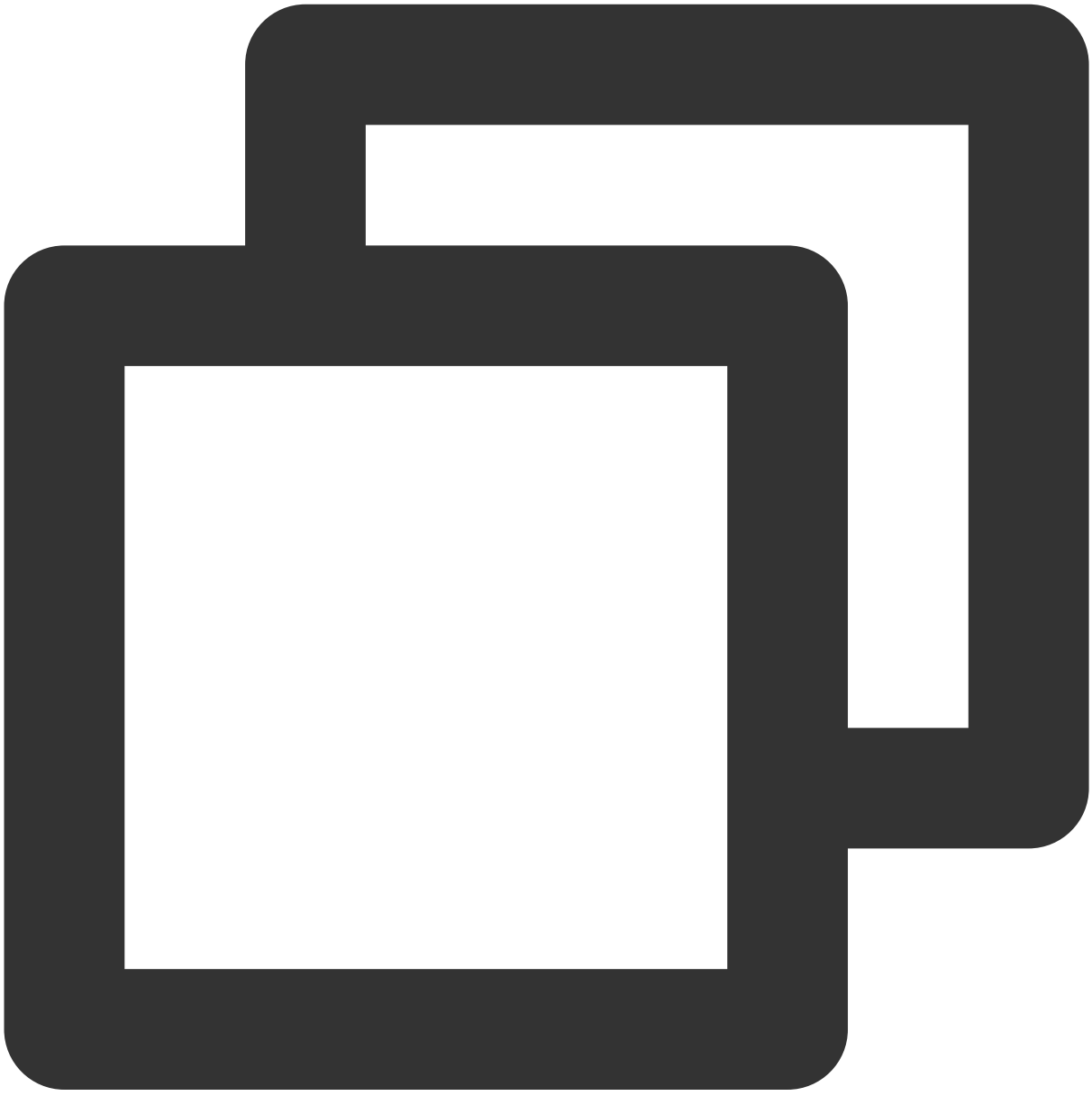
如果 Nginx Ingress 不是调度到超级节点，需合理设置 request 和 limit，既要确保有足够的资源，也要避免使用过多资源导致节点负载过高：



```
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

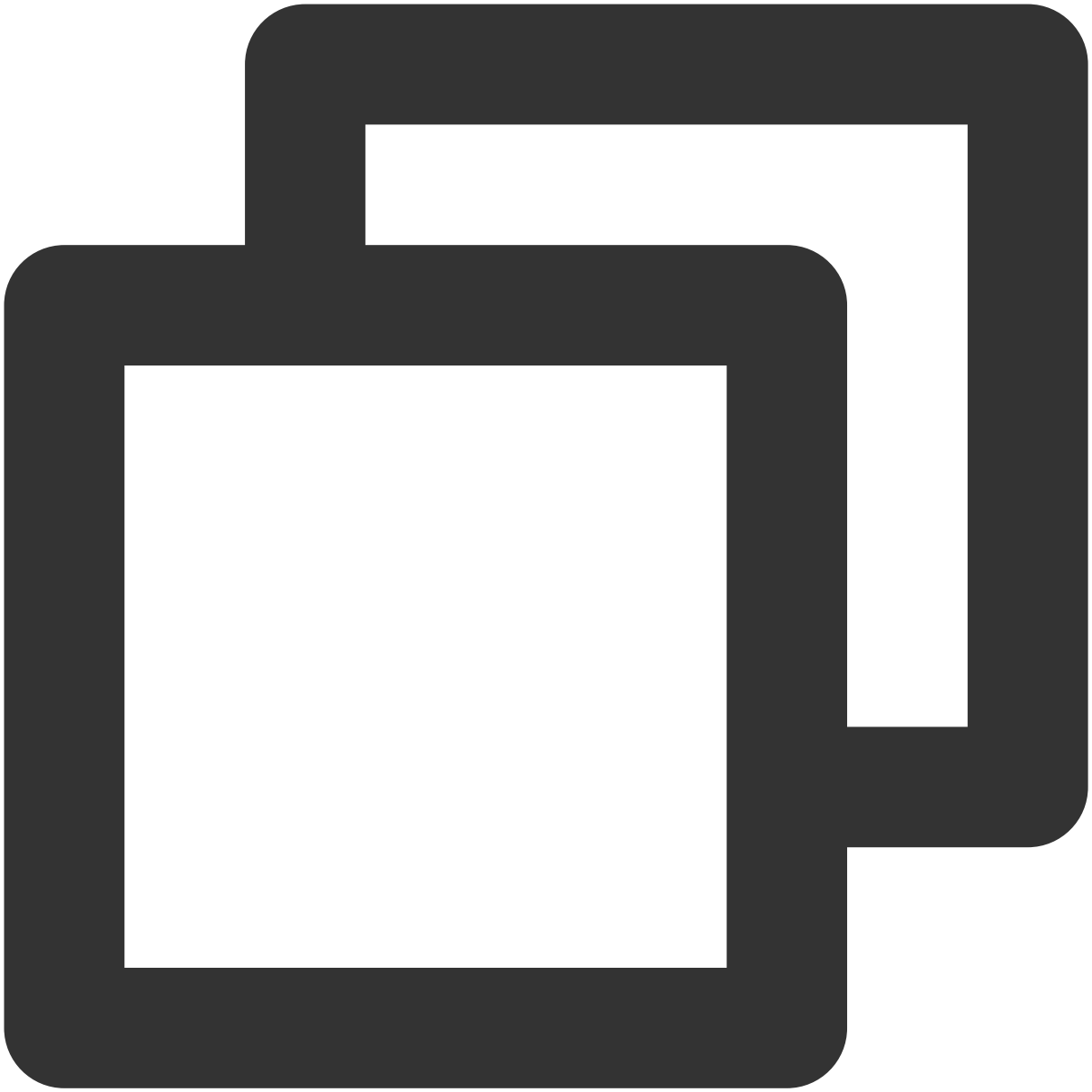
最近更新时间：2024-08-12 17:49:23

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：



```
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
```

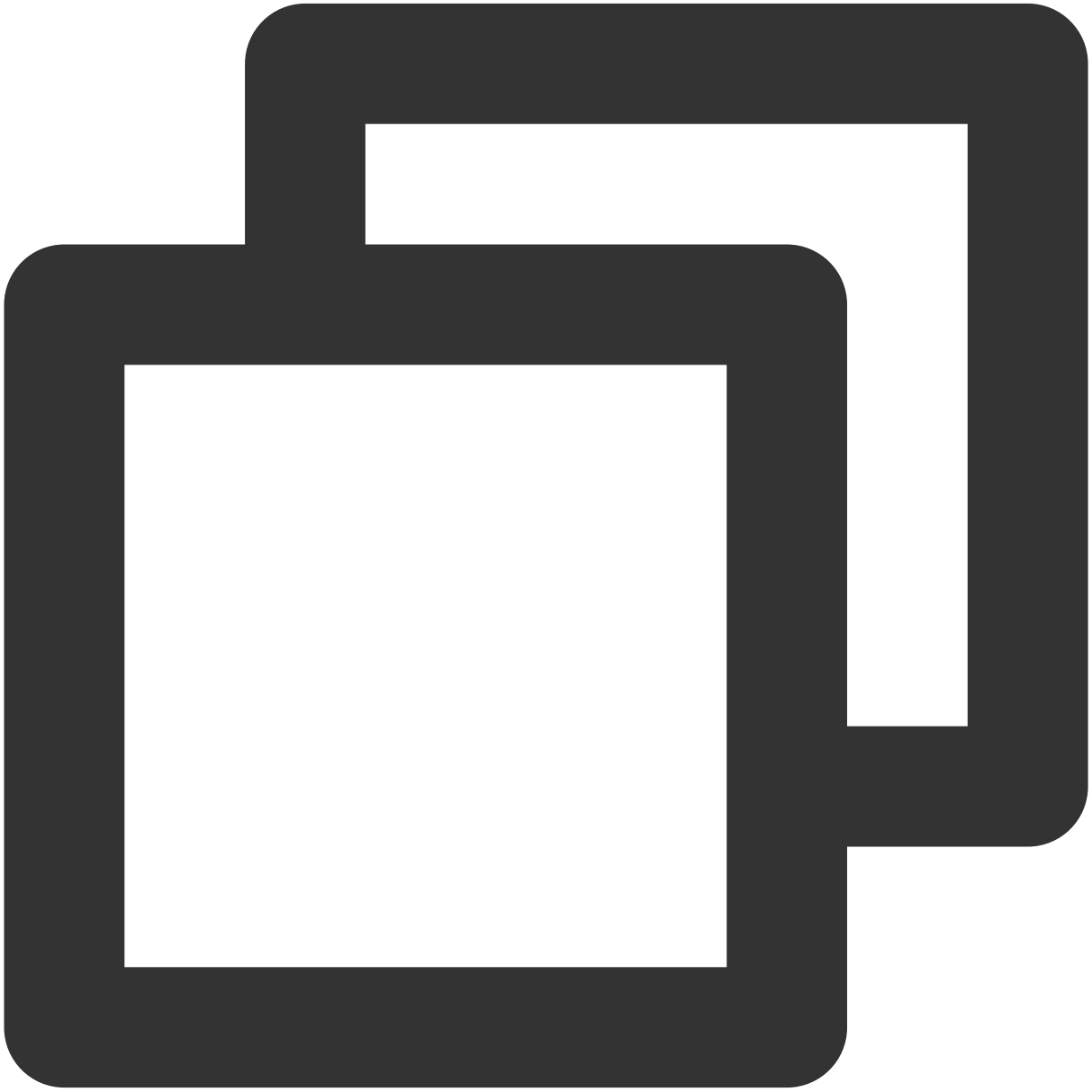
集成 Grafana 监控面板

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，或者如果您有自建的 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_upst
      $upstream_response_length $upstream_response_time $upstream_status $req_id $h
  extraEnvs:
    - name: TZ
      value: Asia/Shanghai
```

2. 确保集群启用了日志采集功能。

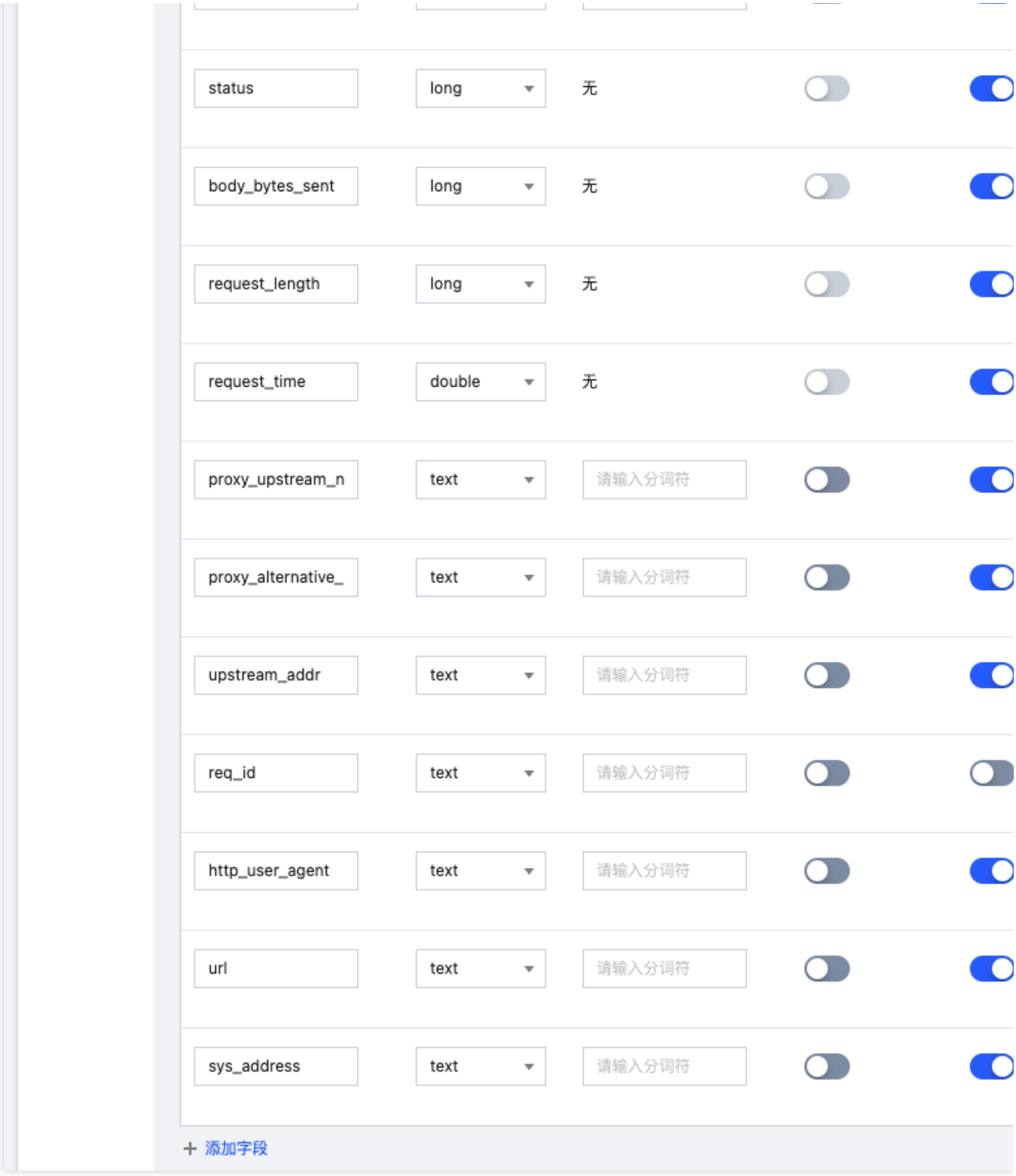
3. 为 Nginx Ingress Controller 准备好 CLS 日志集和日志主题，如果没有，请前往 CLS 控制台 根据自己的需求来创建，并记录日志主题的 ID。

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)：


status
long
无
body_bytes_sent
long
无
request_length
long
无
request_time
double
无
proxy_upstream_n
text
请输入分词符
proxy_alternative_
text
请输入分词符
upstream_addr
text
请输入分词符
req_id
text
请输入分词符
http_user_agent
text
请输入分词符
url
text
请输入分词符
sys_address
text
请输入分词符
添加字段

高级设置：

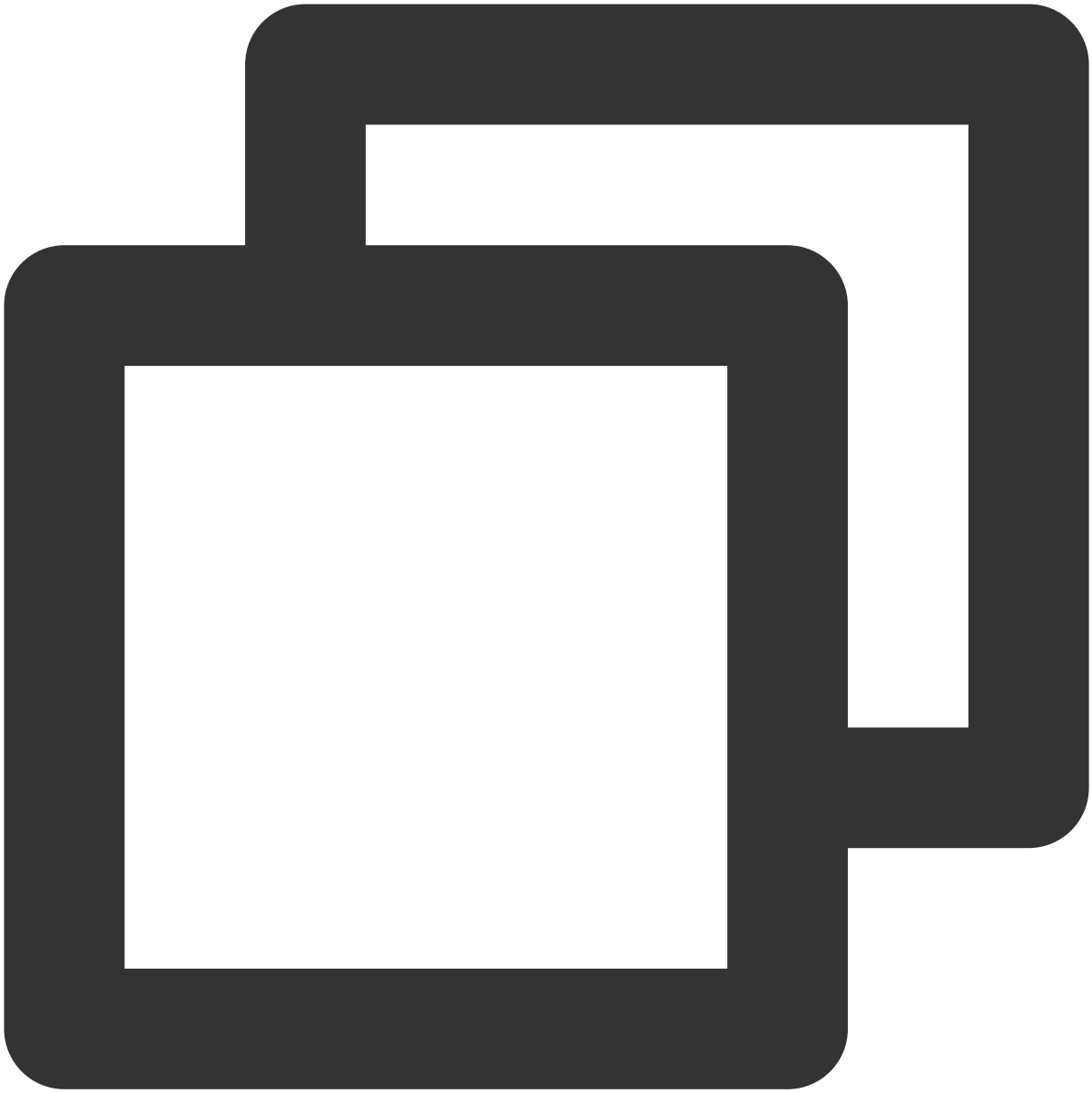
5. 创建 TKE 日志采集规则（根据实际情况二选一）：

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。

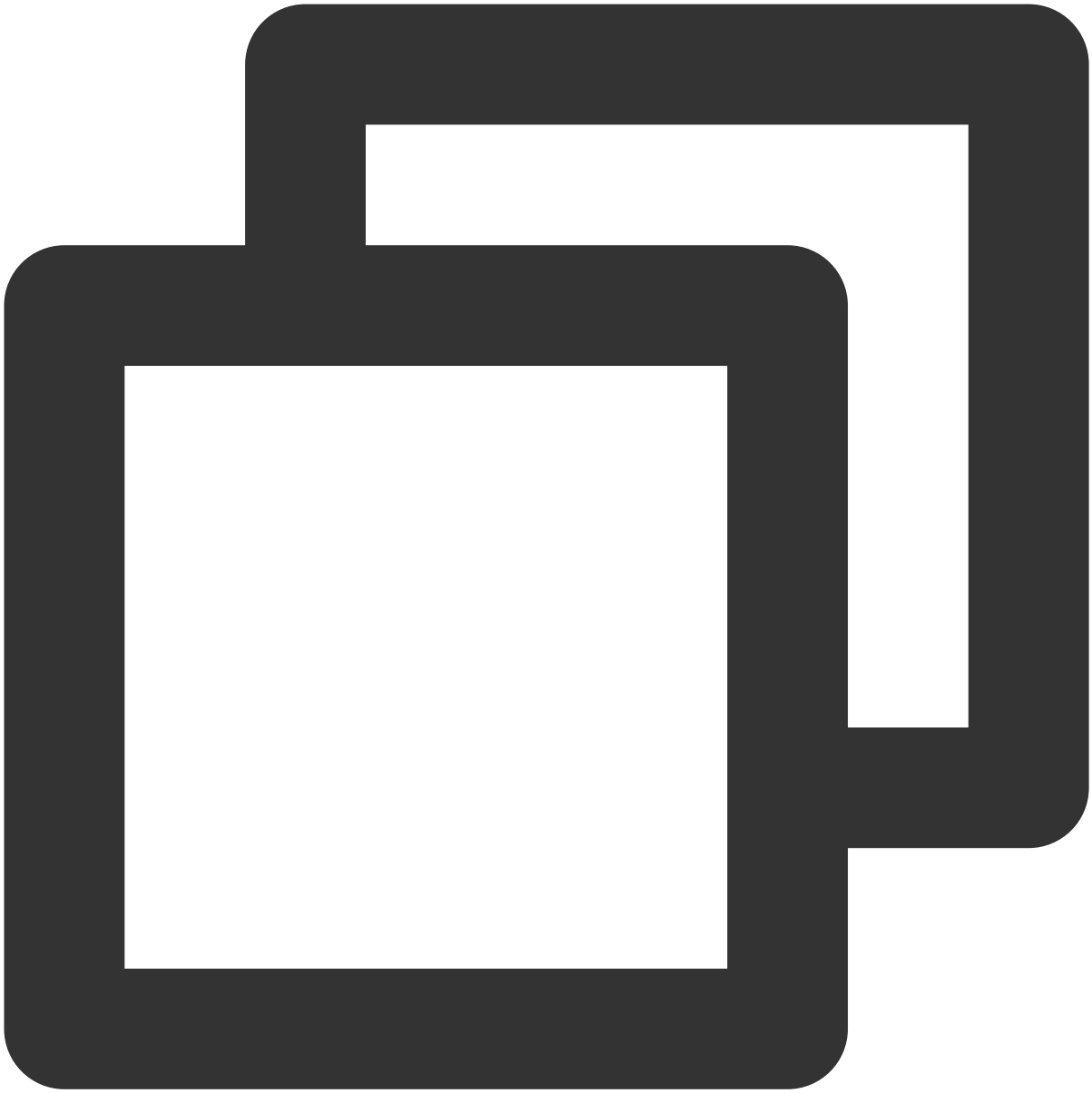
采集标准输出：



```
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
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
      logRegex: (\\S+)\\s-\\s(\\S+)\\s\\[([^\\]]+)\\]\\s\\"(\\w+)\\s(\\S+)\\s([^\\"
```

```
      keys:
        - remote_addr
        - remote_user
        - time_local
        - 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
        - sys_address
  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
```

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
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
      logRegex: (\\S+)\\s-\\s(\\S+)\\s\\[([^\\]]+)\\]\\s\\"(\\w+)\\s(\\S+)\\s([^\\"
```

```
      keys:
        - remote_addr
        - remote_user
        - time_local
        - 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
        - sys_address
  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
      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.

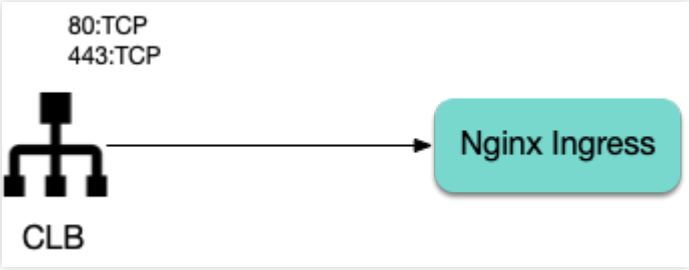
9. 在**日志服务**的 Nginx 访问大盘 和 Nginx 监控大盘 可以直接通过面板来设置监控告警规则，详情请参见 监控告警概述。

接入腾讯云 WAF

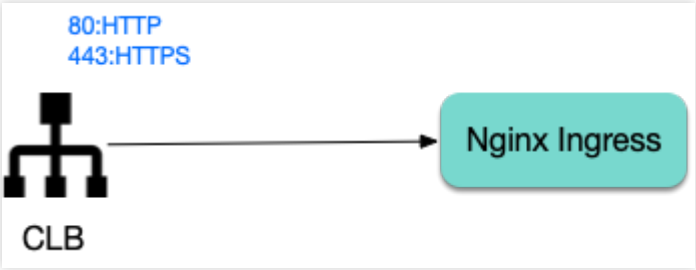
最近更新时间：2024-08-12 17:49:23

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：

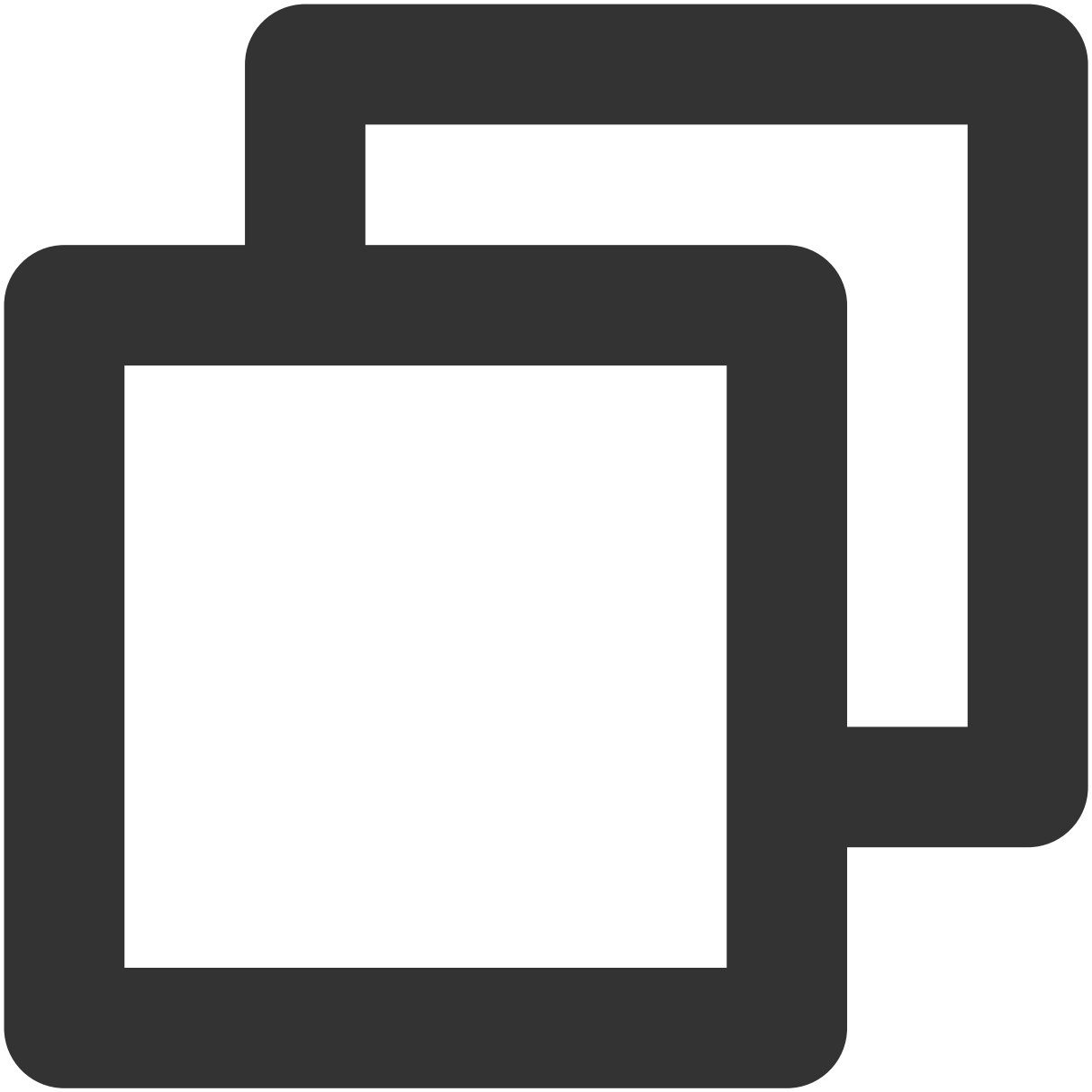


本文为您介绍将 Nginx Ingress 所使用的 CLB 监听器改为七层监听器。

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）。

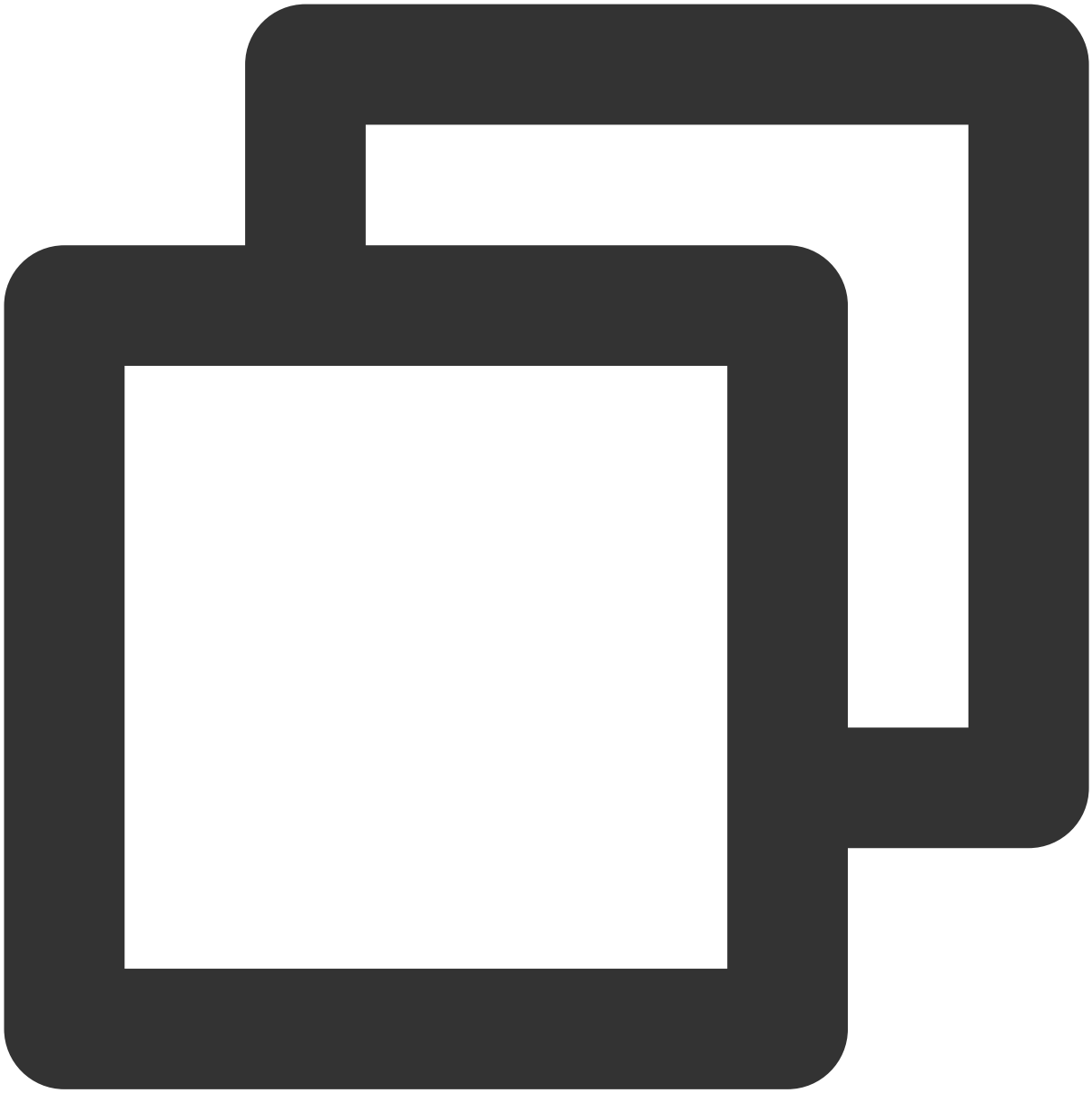
不需要 HTTP 流量可以将 `enableHttp` 置为 false。

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：

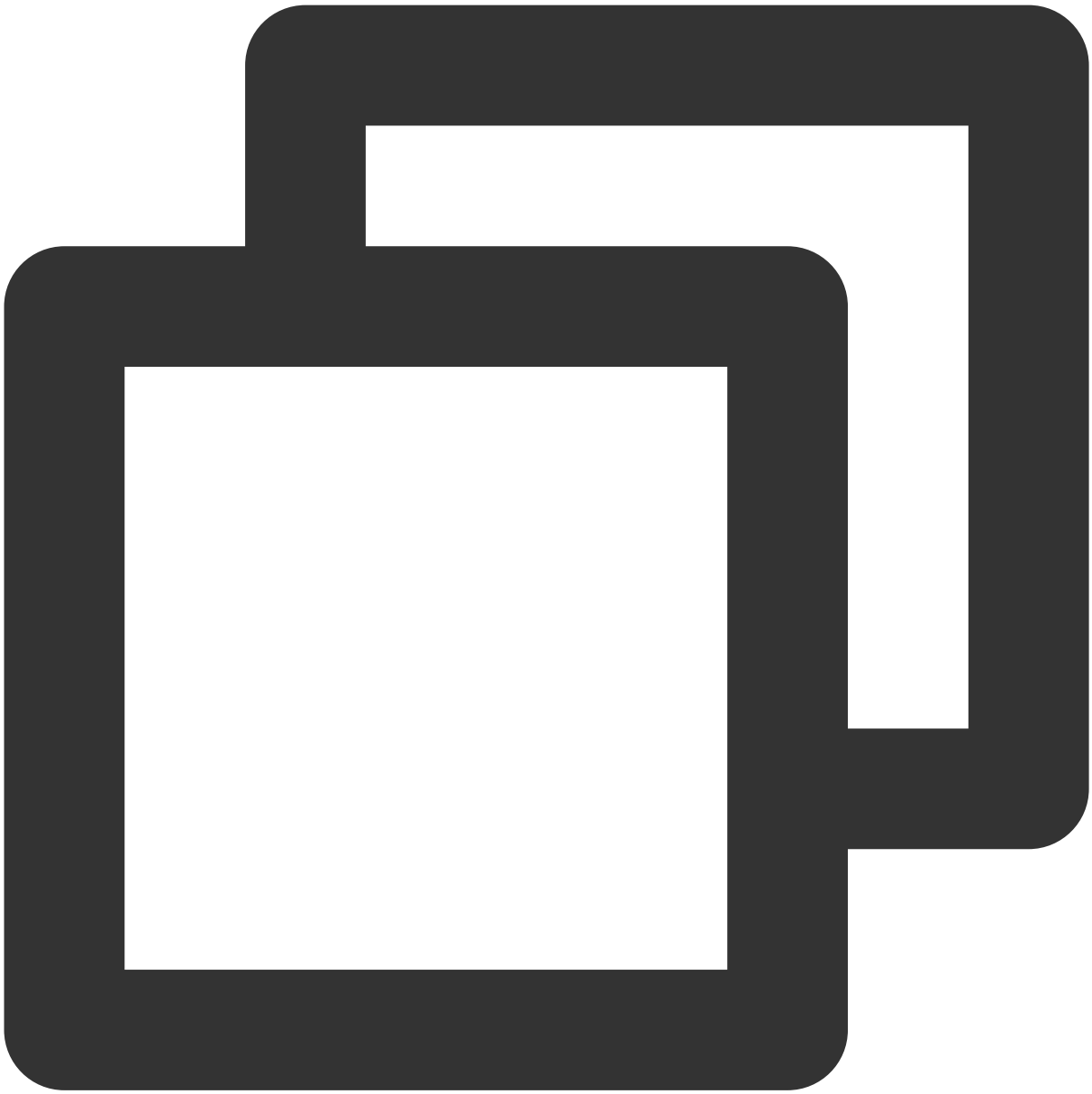
操作步骤

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
```

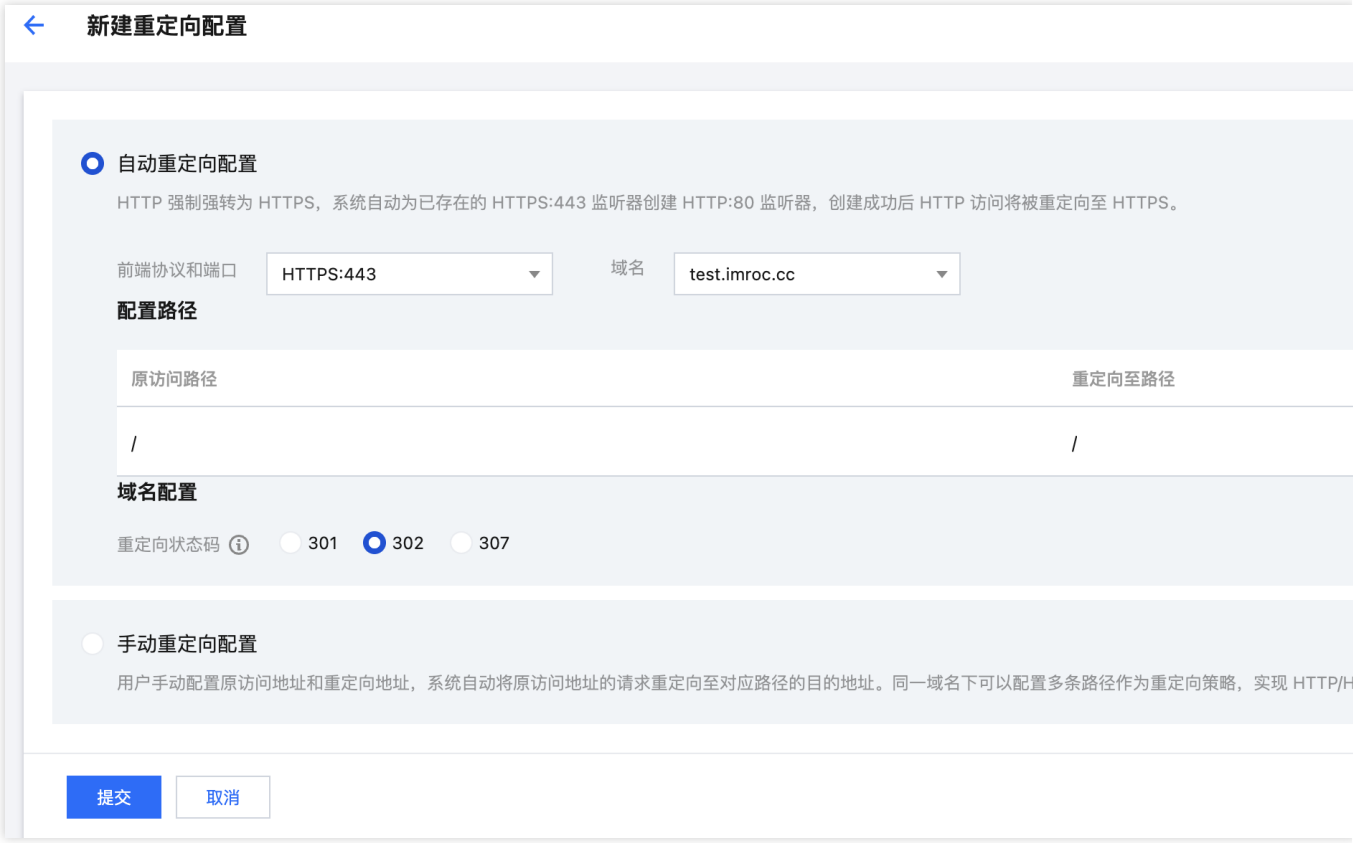
3. 配置 `values.yaml` ：



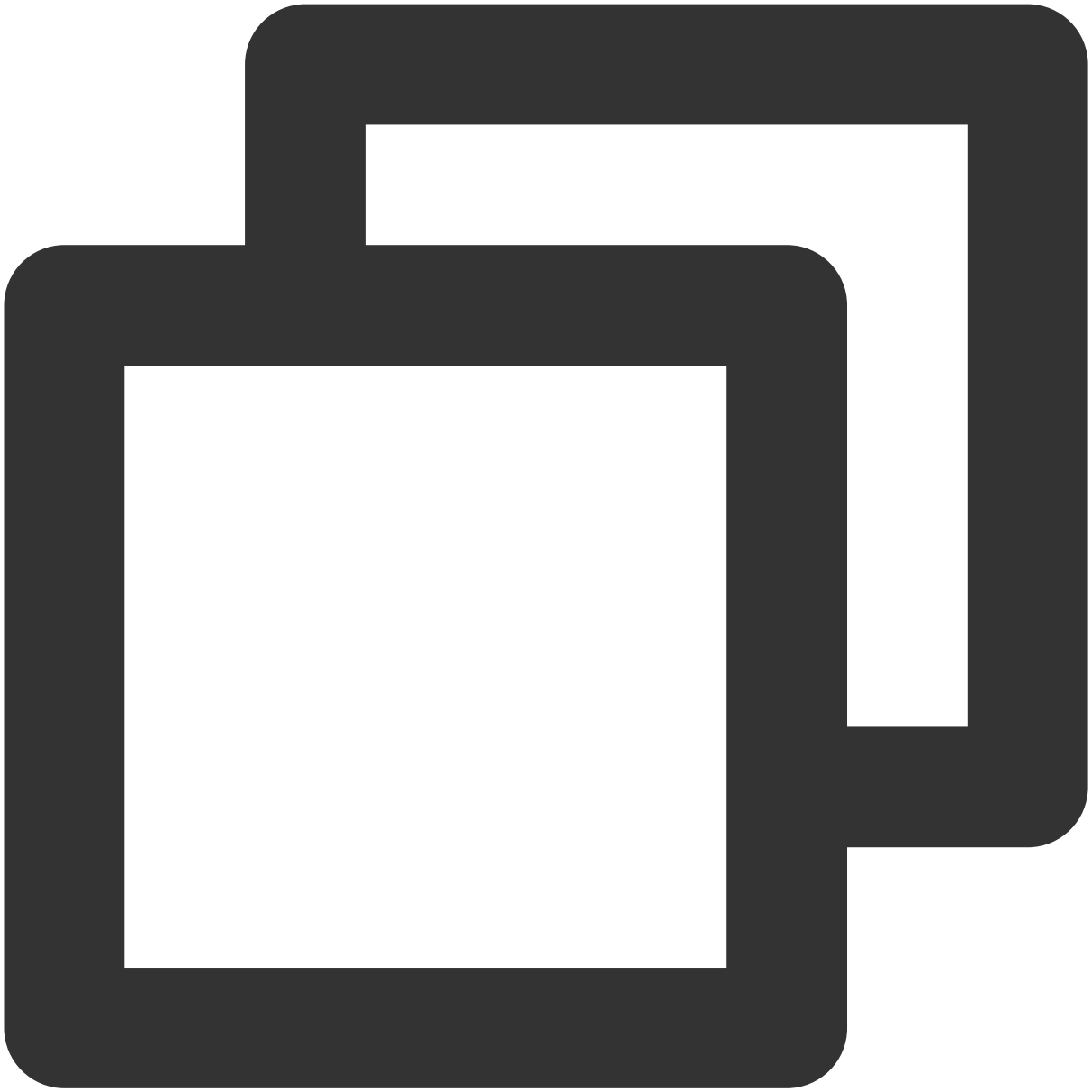
```
controller: # 以下配置将依赖镜像替换为了 docker hub 上的 mirror 镜像以保证在国内环境能正常拉取
  image:
    registry: docker.io
    image: k8smirror/ingress-nginx-controller
  admissionWebhooks:
    patch:
      image:
        registry: docker.io
        image: k8smirror/ingress-nginx-kube-webhook-certgen
```

```
  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：



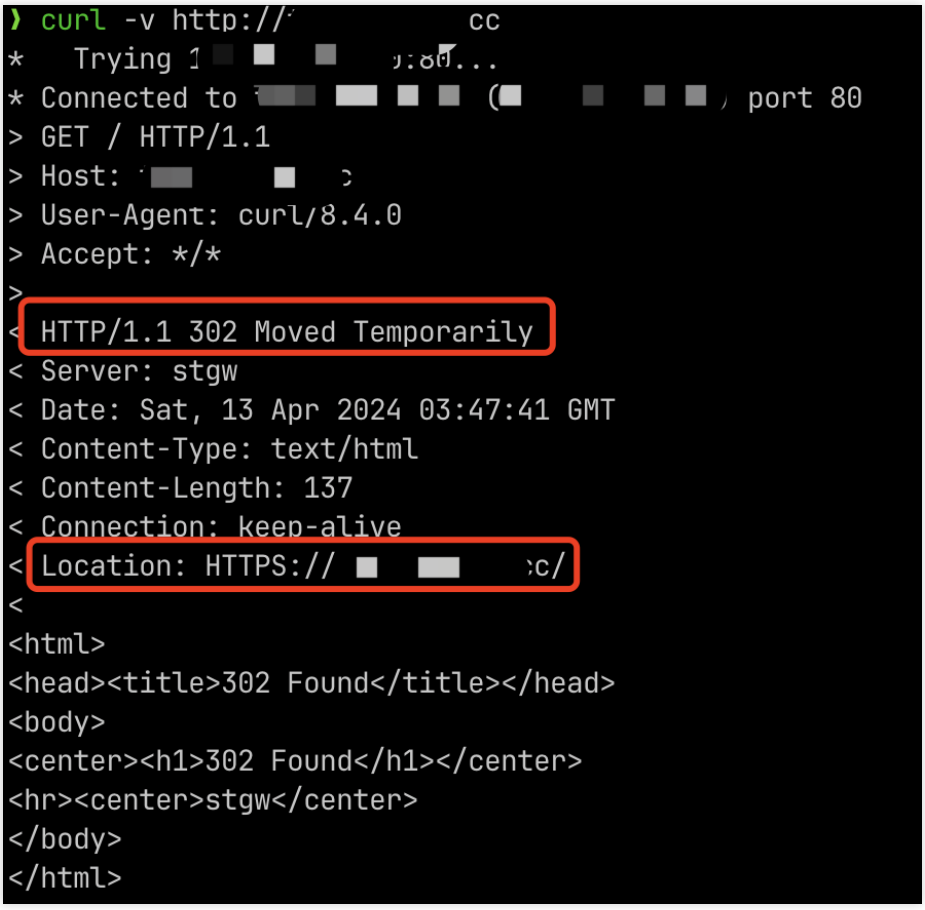
5. 部署测试应用和 Ingress 规则：

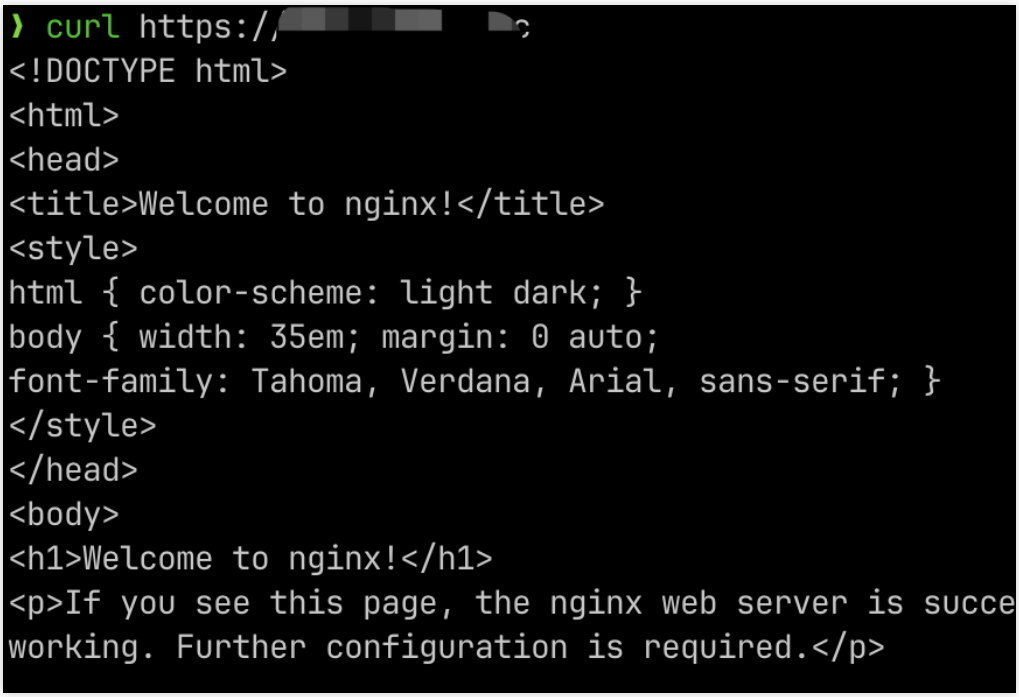


```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
```

```
  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://
Trying
Connected to
port 80
GET / HTTP/1.1
Host:
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://
<html>
<head><title>302 Found</title></head>
<body>
<h1>302 Found</h1>
<hr> stgw
</body>
</html>


curl https://
<!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>
If you see this page, the nginx web server is succe
working. Further configuration is required.

配置 WAF

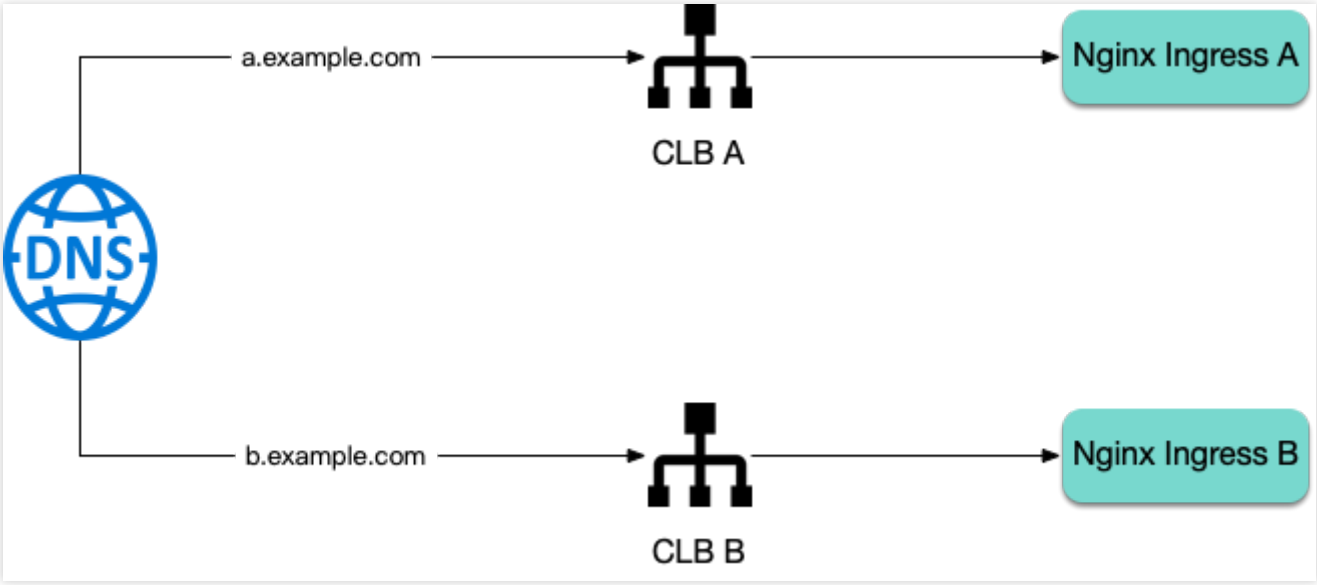
Nginx Ingress 配置好后，确认对应的 CLB 监听器已经改为了 HTTP/HTTPS，则已满足 Nginx Ingress 接入 WAF 的前提条件，可以根据 WAF 官方文档 的指引来进行配置，最终完成 Nginx Ingress 的 WAF 接入。

安装多个 Nginx Ingress Controller

最近更新时间：2024-08-12 17:49:23

概述

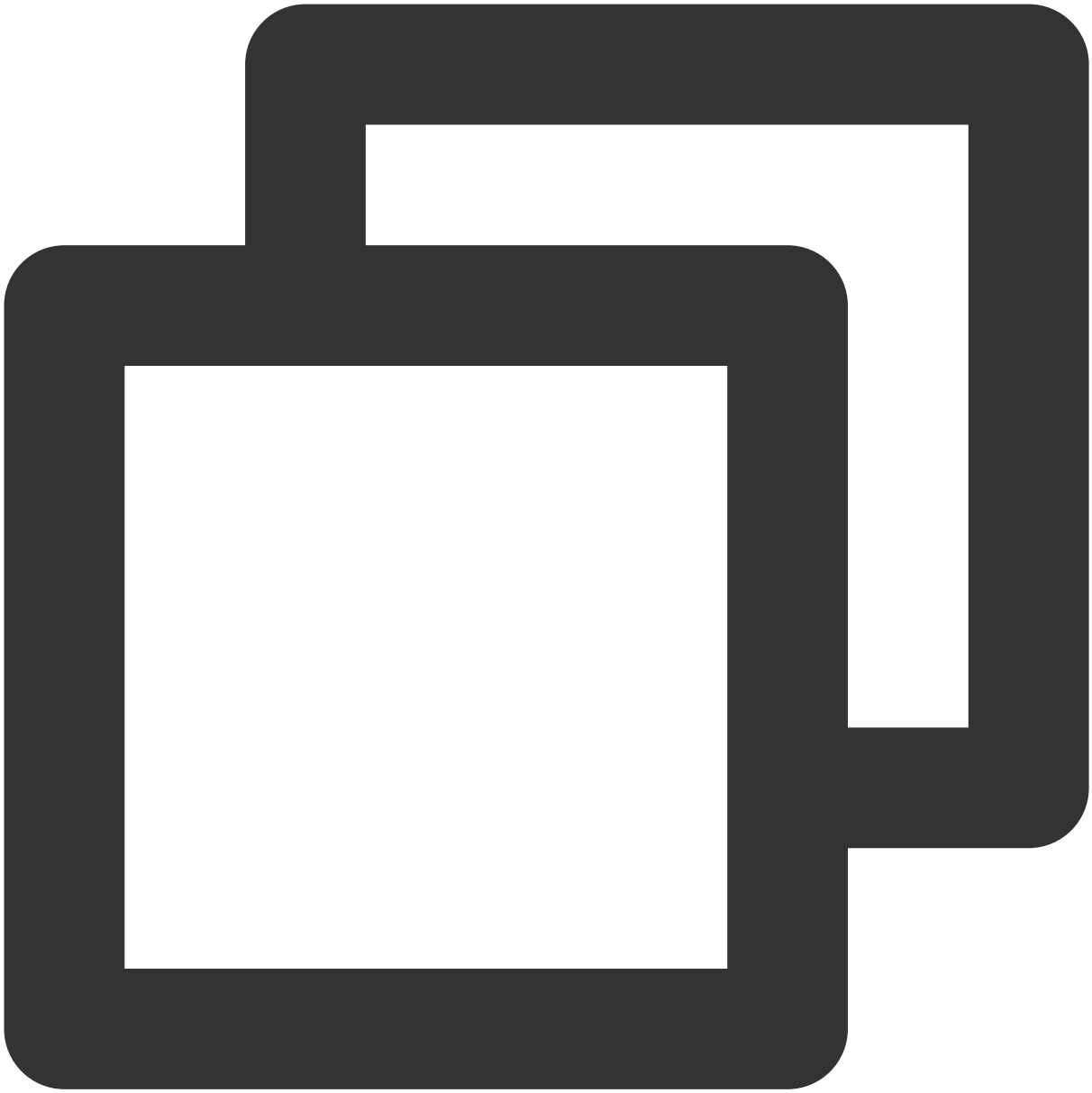
如果您需要部署多个 Nginx Ingress Controller，即希望不同的 Ingress 规则使用不同的流量入口：



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 `ingressClassName` （注意不要冲突）：

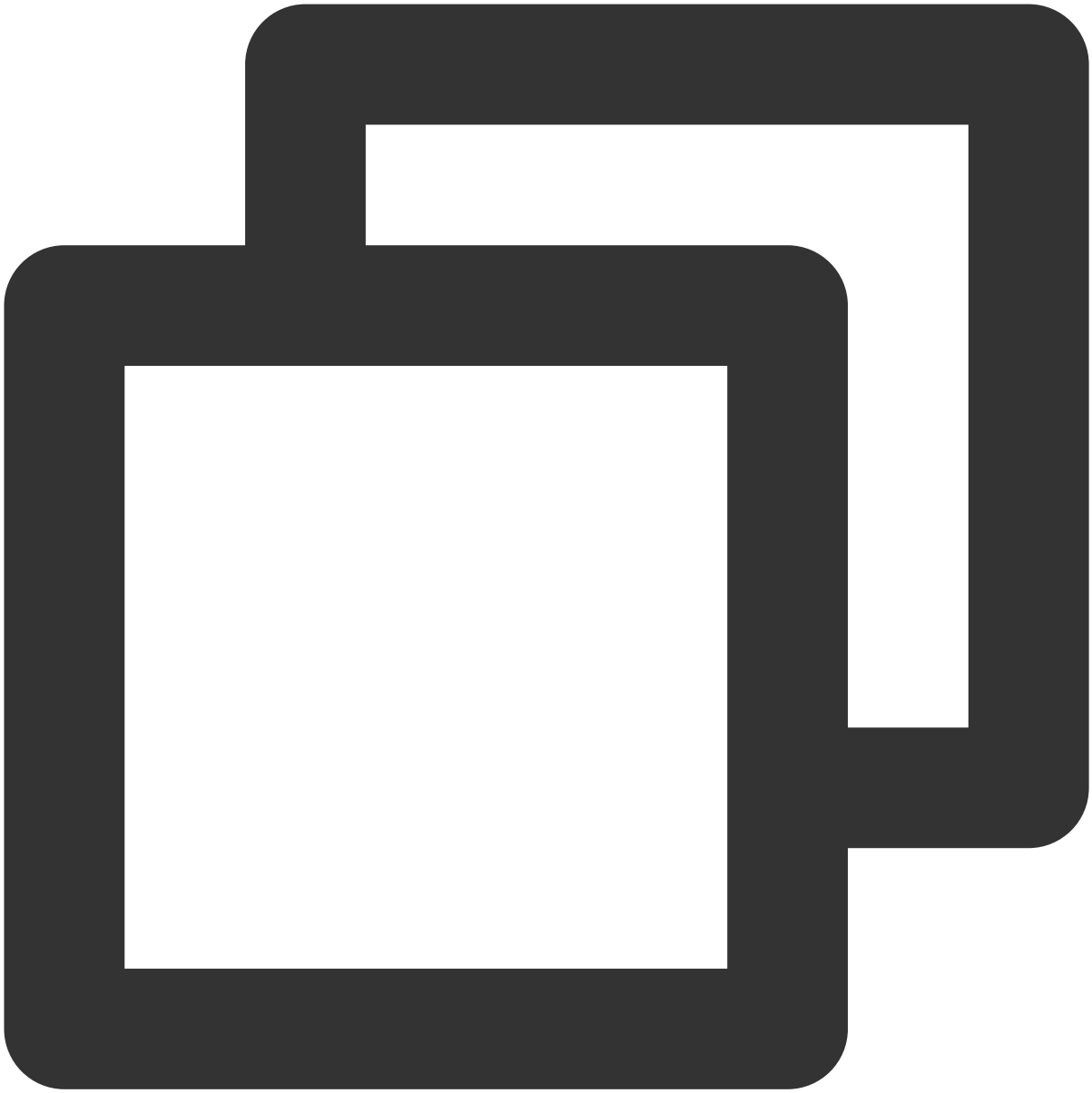


```
controller:
  ingressClassName: prod
  ingressClassResource:
    name: prod
    controllerValue: k8s.io/ingress-prod
```

说明：

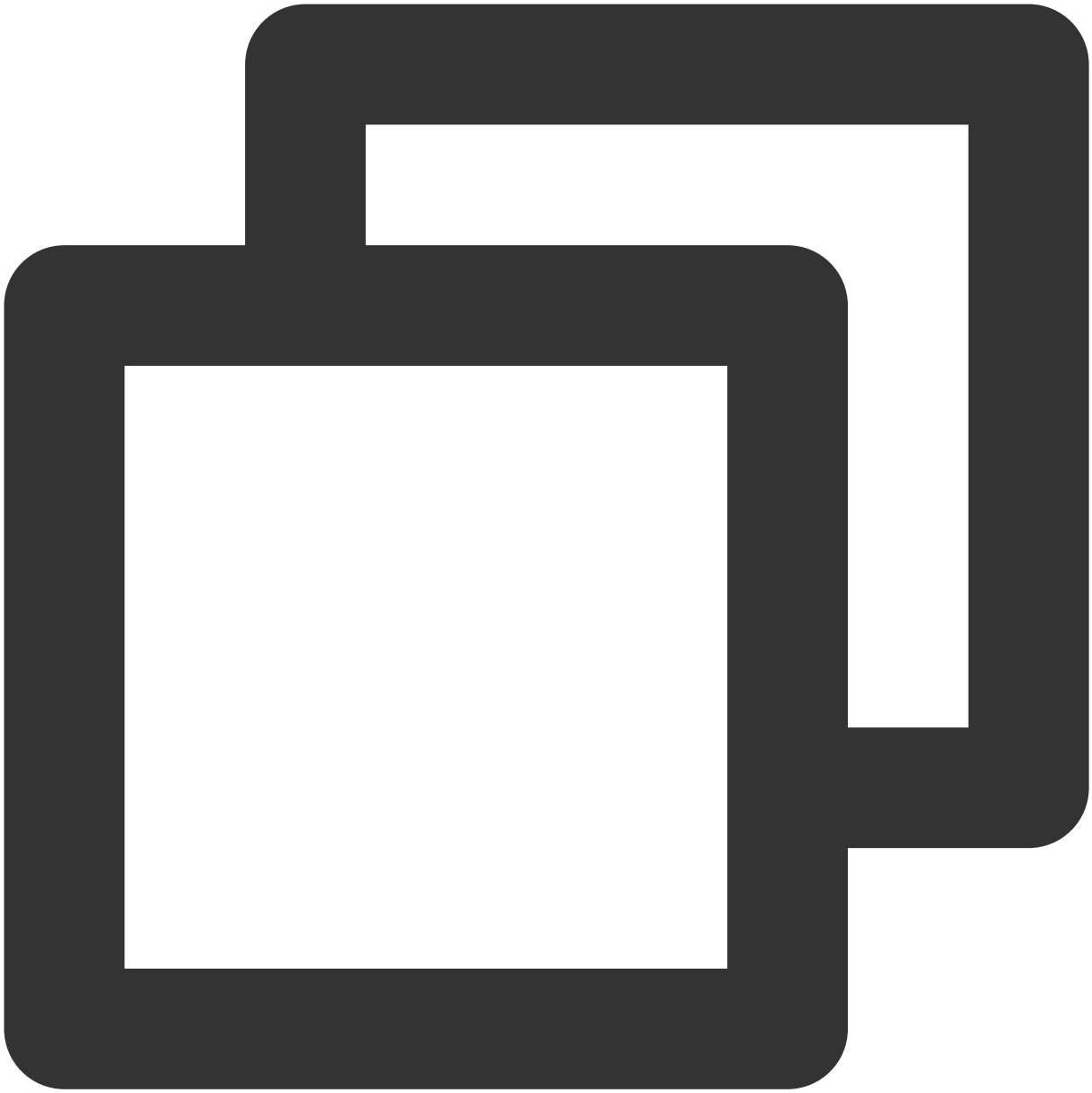
三个字段需同时改。

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

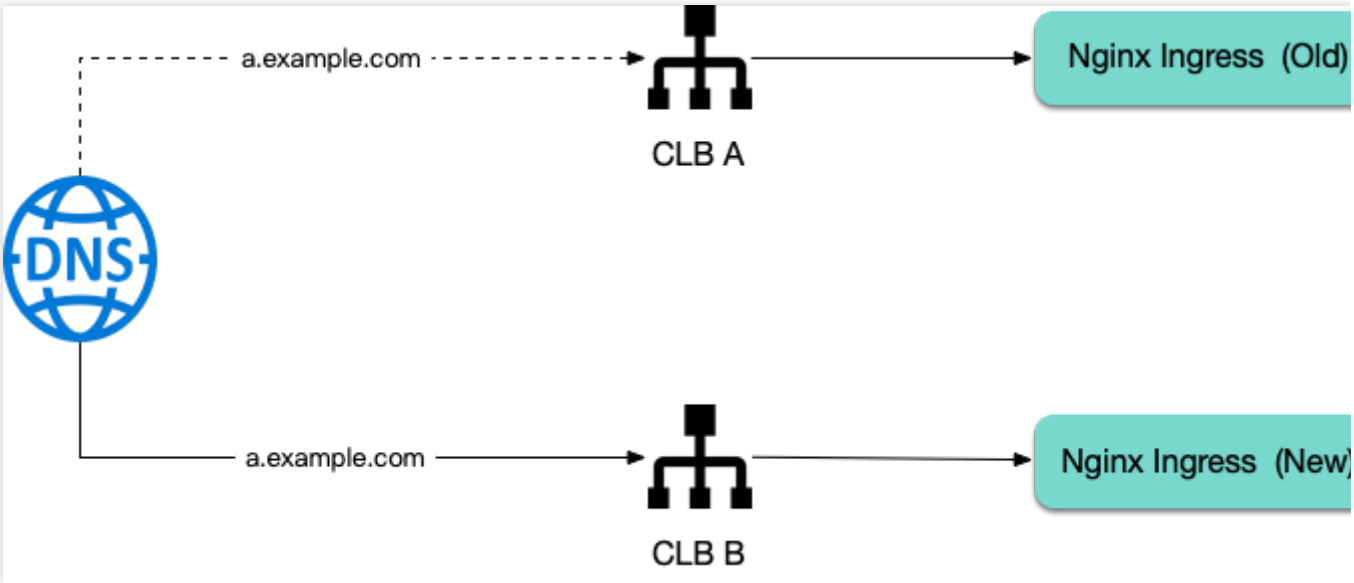
最近更新时间：2024-08-12 17:49:23

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。

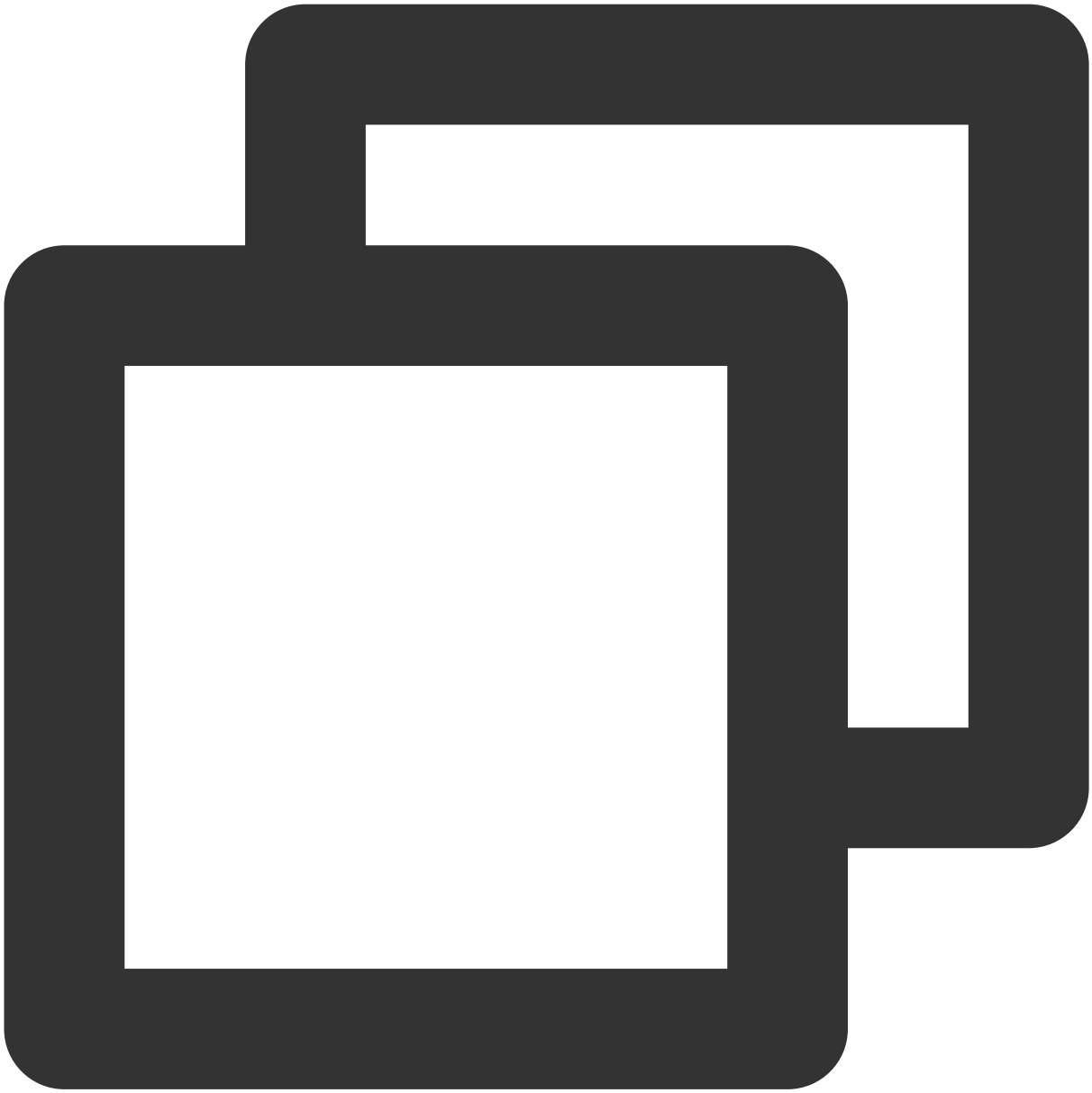
迁移思路

使用本文中自建的方法创建一套新的 Nginx Ingress 实例，与旧的实例共享同一个 IngressClass，也就会共享相同的 Ingress 转发规则，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

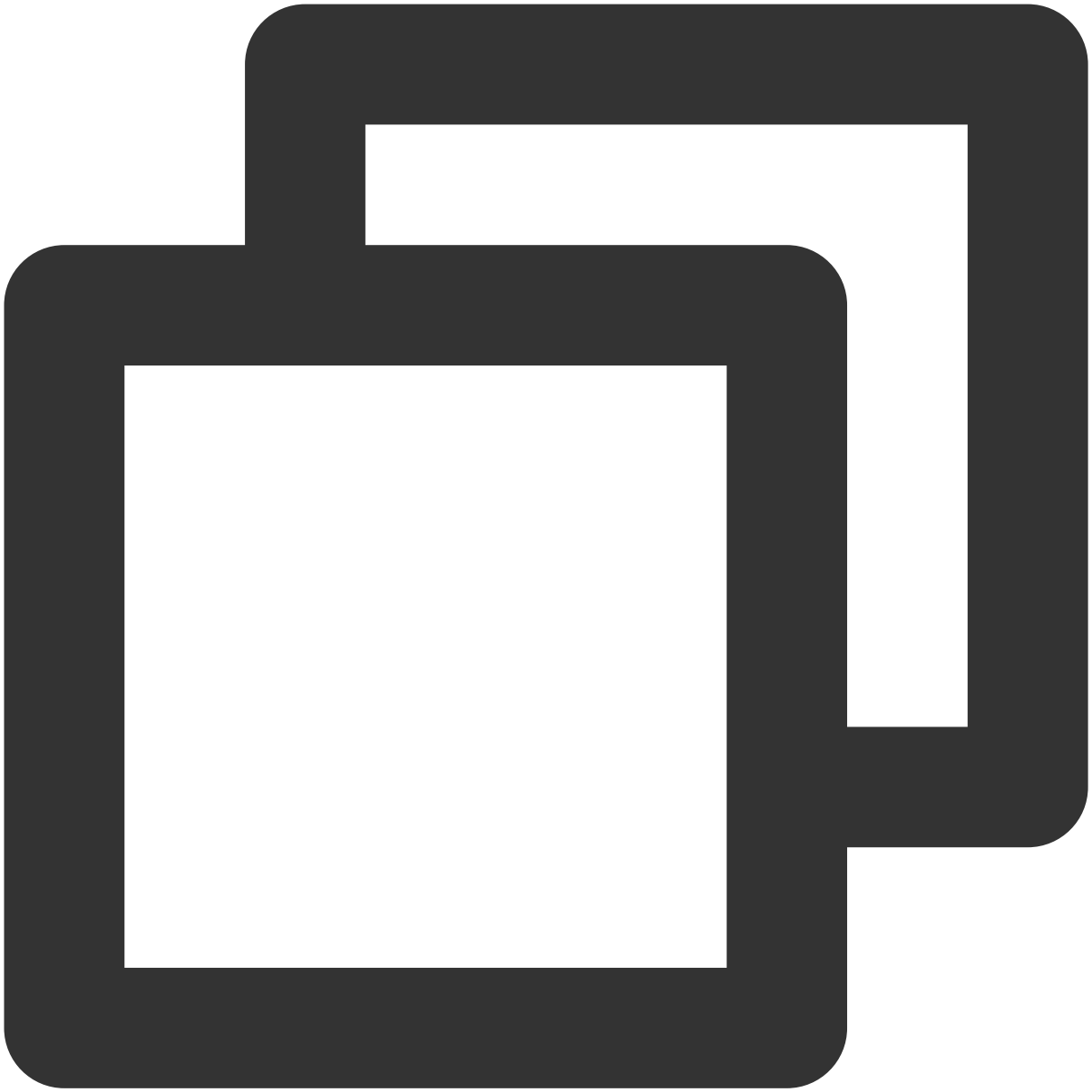
1. 先确认已安装的 Nginx Ingress 实例的 IngressClass 名称，例如：



```
$ kubectl get deploy -A | grep nginx
kube-system            extranet-ingress-nginx-controller           1/1     1
```

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`。

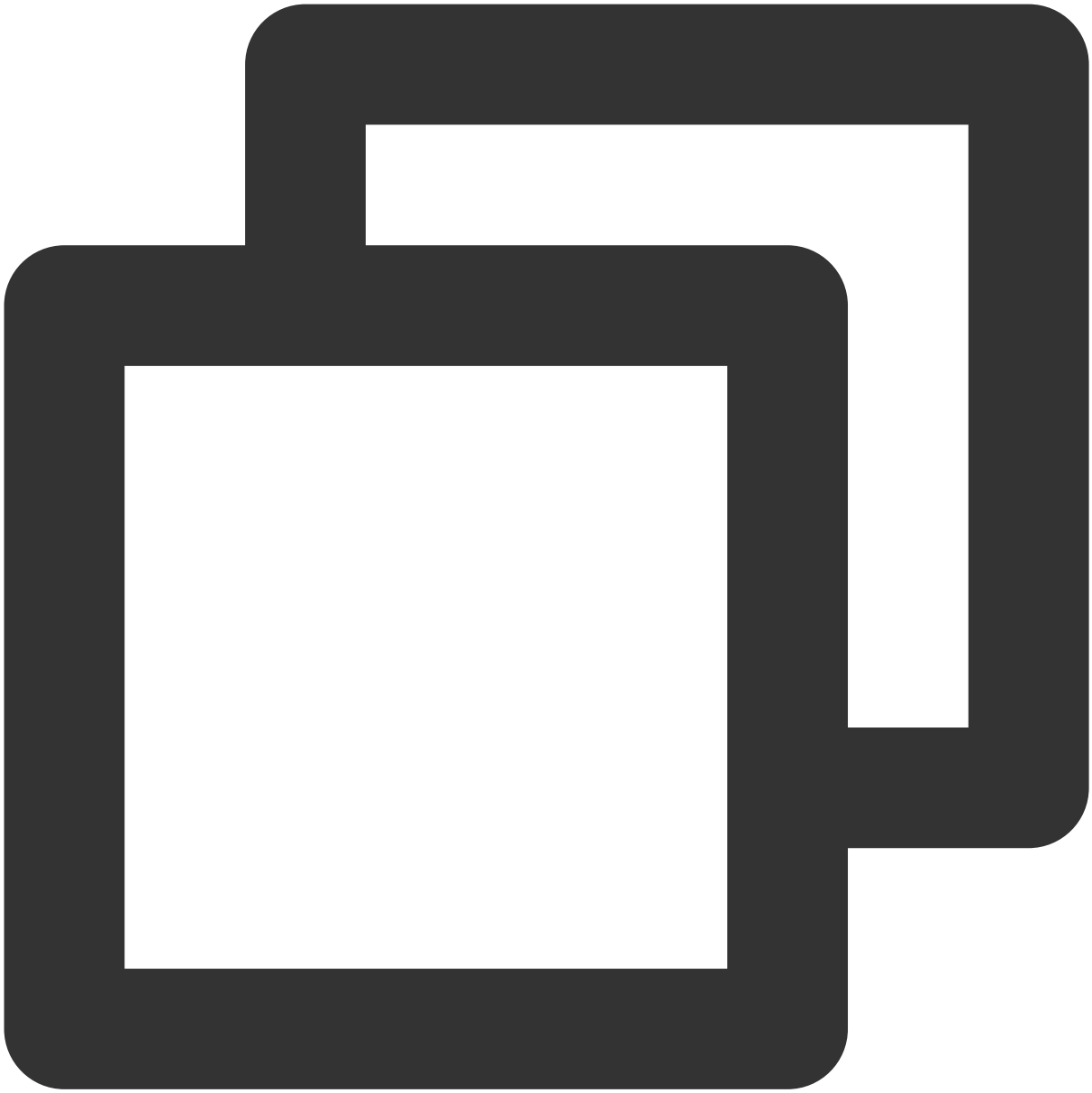
2. 确认当前使用的 nginx ingress 的镜像版本：



```
$ kubectl -n kube-system get deploy extranet-ingress-nginx-controller -o yaml | gre
        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：



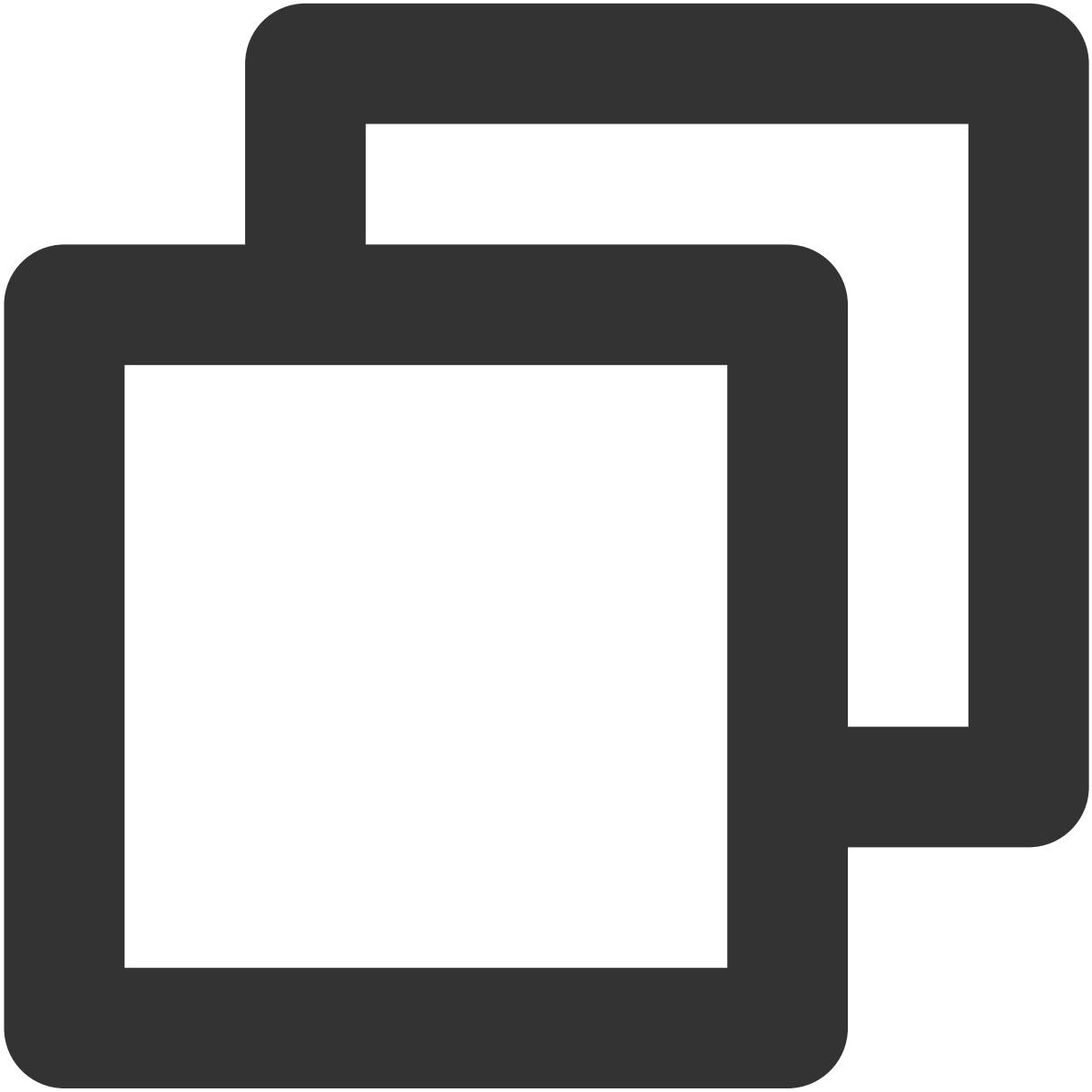
```
$ helm search repo ingress-nginx/ingress-nginx --versions  | grep 1.9.5
ingress-nginx/ingress-nginx     4.9.0           1.9.5           Ingress controller
```

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

确保 helm 新创建的 Nginx Ingress 实例和 TKE 插件创建 Nginx Ingress 实例共用一个 IngressClass，即让 Ingress 规则在两边同时生效。

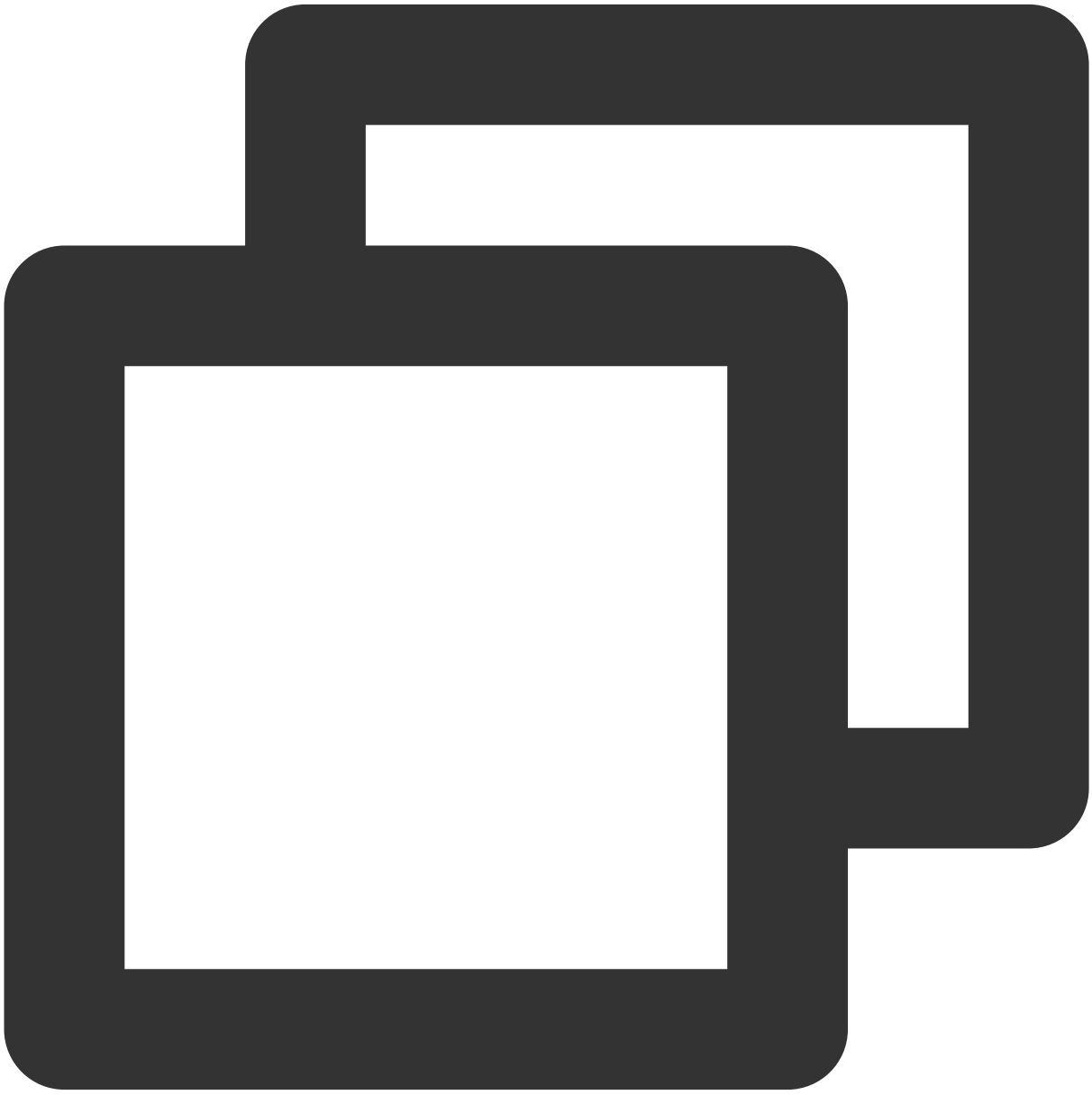
查看当前的 IngressClass 定义：



```
$ kubectl get ingressclass extranet -o yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  creationTimestamp: "2024-03-27T10:47:49Z"
  generation: 1
  labels:
```

```
    app.kubernetes.io/component: controller
  name: extranet
  resourceVersion: "27703380423"
  uid: 5e2de0d1-8eae-4b55-afde-25c8fe37d478
spec:
  controller: k8s.io/extranet
```

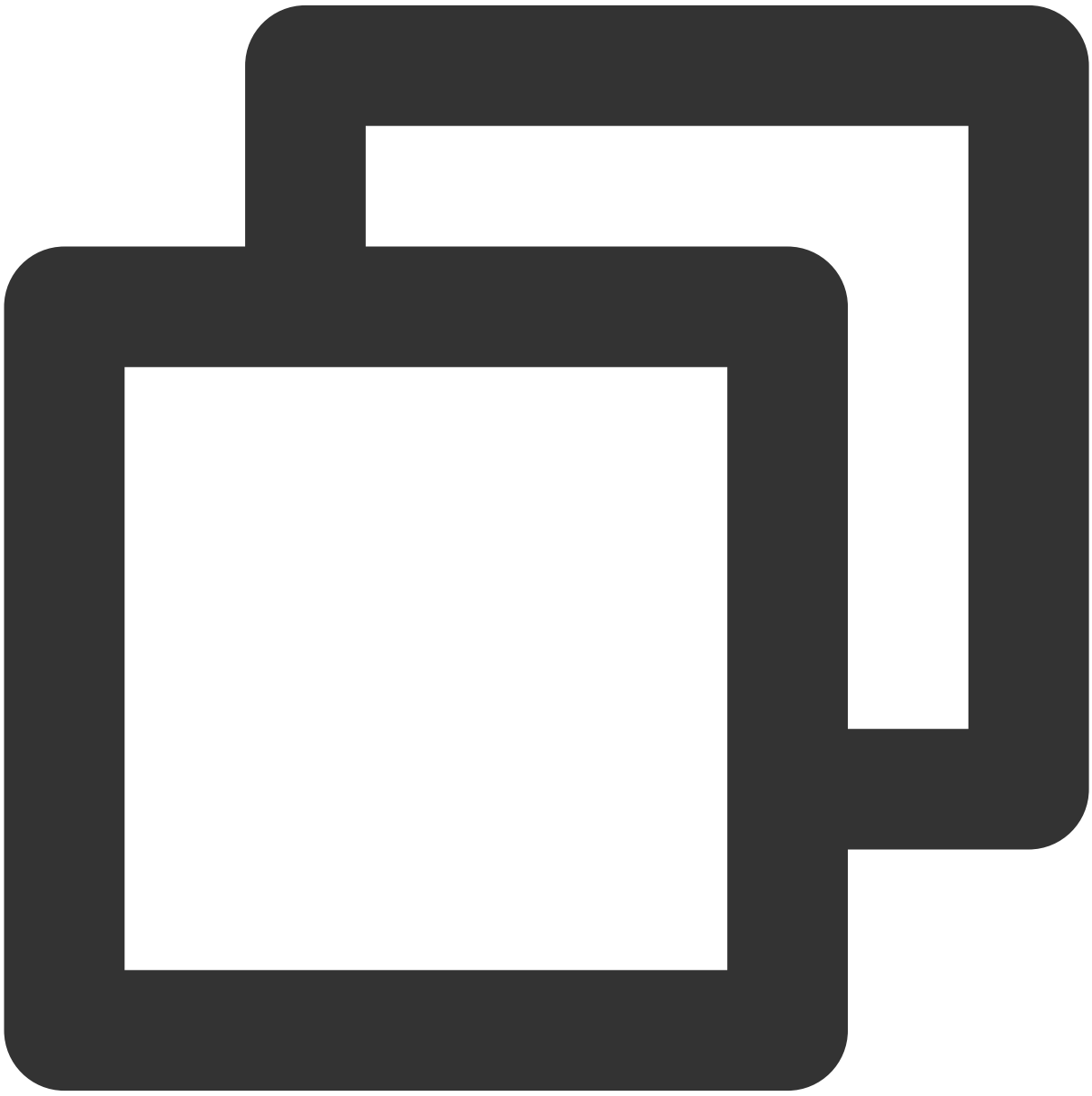
获取到 controller 的值 `k8s.io/extranet` ，与 IngressClass 名称一起，配置到 `values.yaml` 中：



```
controller:
  ingressClassName: extranet # IngressClass 名称
```

```
  ingressClassResource:
    enabled: false # 不自动创建 IngressClass 资源，避免冲突
    controllerValue: k8s.io/extranet # 新 Nginx Ingress 复用已有的 IngressClass
```

安装新的 Nginx Ingress Controller



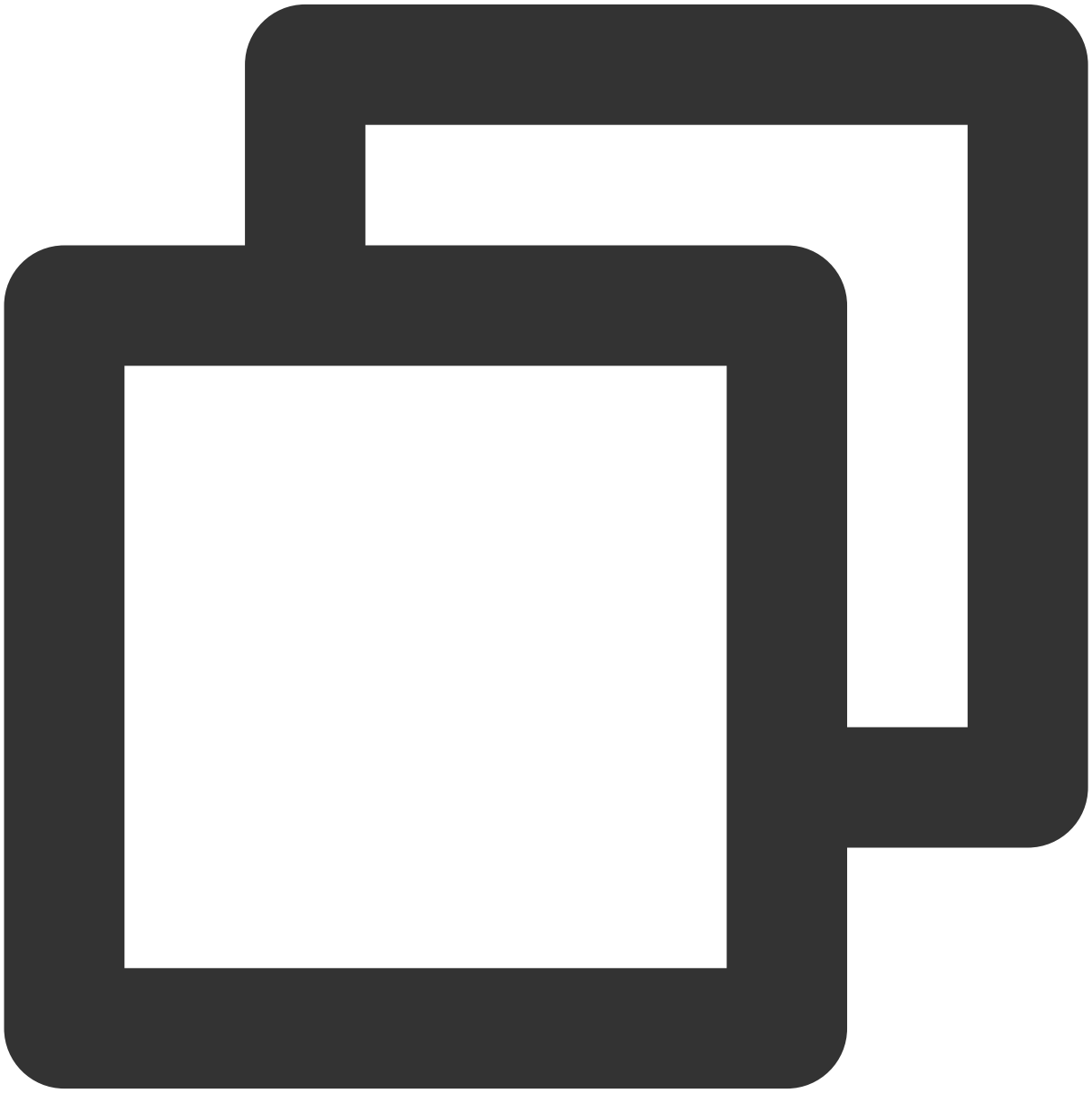
```
helm upgrade --install new-extranet-ingress-nginx ingress-nginx/ingress-nginx \\
  --namespace ingress-nginx --create-namespace \\
```

```
  --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       E
new-extranet-ingress-nginx-controller         LoadBalancer   172.16.165.100   4
```

`EXTERNAL-IP` 是新的流量入口，请验证确认能够正常转发。

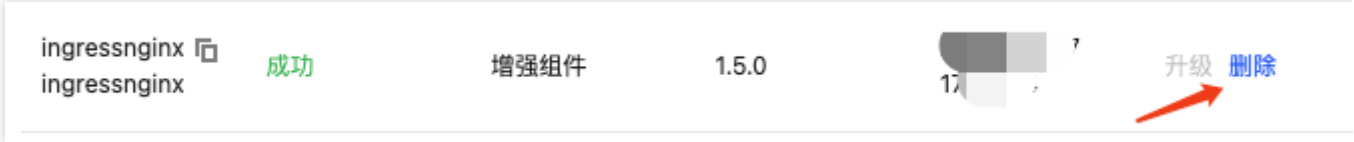
切换 DNS

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，再前往 TKE 控制台删除 Nginx Ingress 实例：

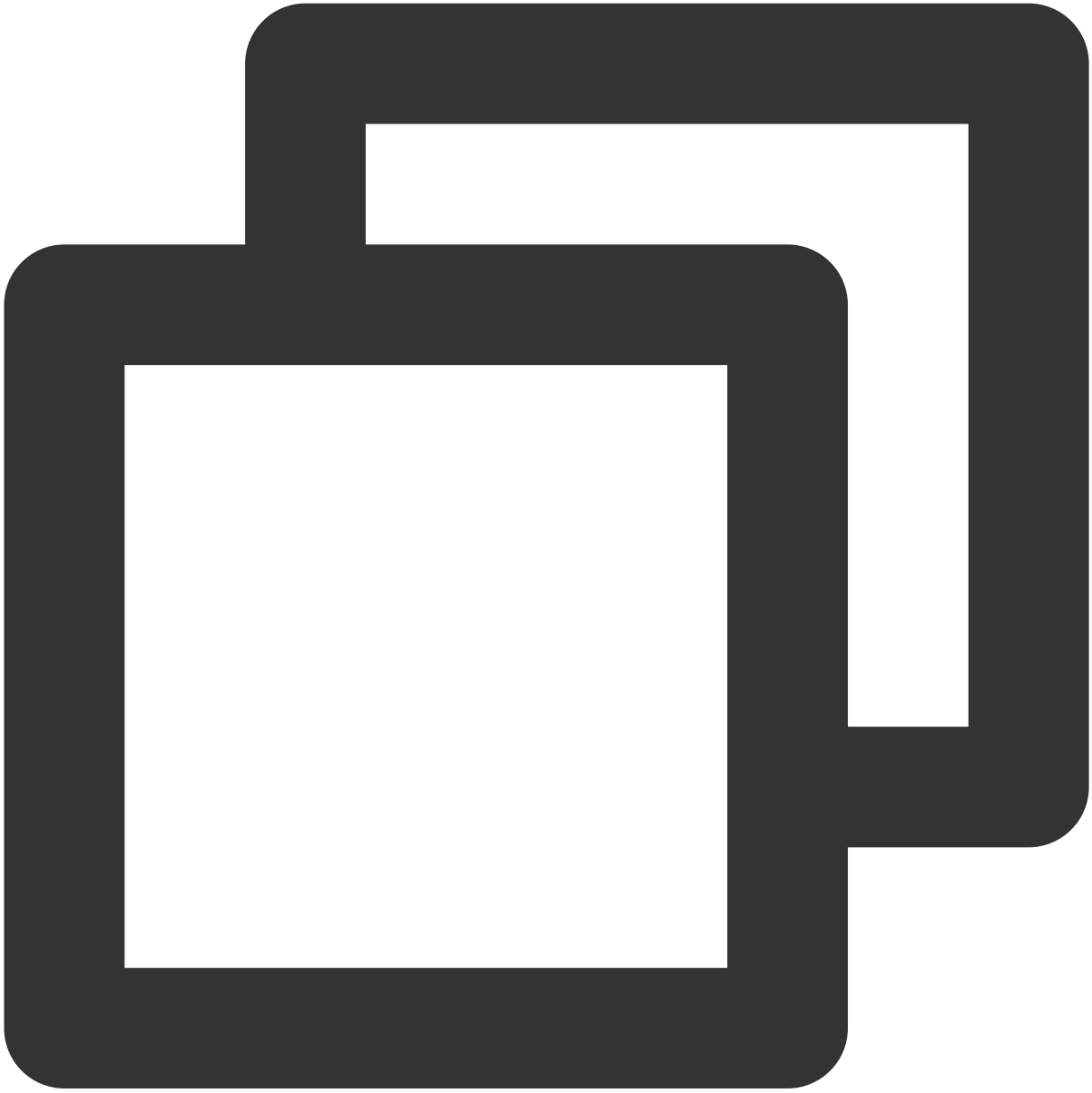
2. 在**组件管理**中删除 `ingressnginx` ，彻底完成迁移。



values.yaml 完整配置示例

最近更新时间：2024-08-12 17:49:23

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
          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
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuratio
    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
    # 参考：https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuratio
    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-configuratio
    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_upst
      $upstream_response_length $upstream_response_time $upstream_status $req_id $h
    # 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
          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
      scaleDown:
        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
      scaleUp:
        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
```

在 TKE 上对 Pod 进行带宽限速

最近更新时间：2022-03-23 18:17:30

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，使用详细介绍请参阅 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. 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-eni-agent` 配置。

```
kubectl edit daemonset tke-eni-agent -n kube-system
```

之后添加 args `--bandwidth`，开启 bandwidth 插件支持。

> 说明：
> tke-eni-agent 加入以上参数即可开启该特性，去掉即可关闭。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/se
```

TKE 基于弹性网卡直连 Pod 的网络负载均衡

最近更新时间：2023-05-24 10:16:25

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。但因为转发的随机性，一个节点上部署多个副本时会话保持依旧无法支持，且在 Local 转发在滚动更新时，容易出现服务的闪断，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（）

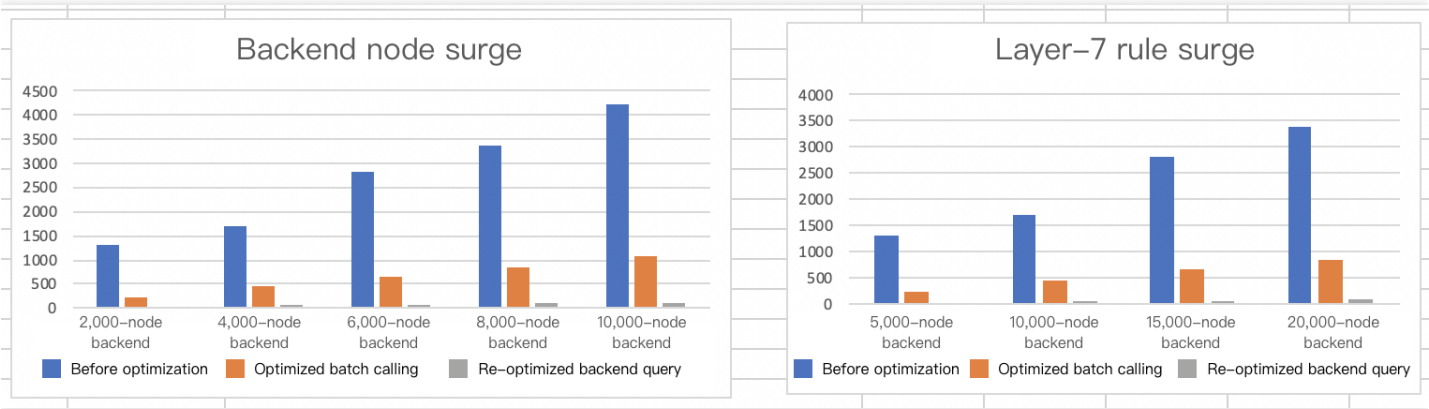
七层规则数量	集群节点数量	集群节点数量（更新）	优化前（秒）	优化批量调用（秒）	再优化后端实例查询（秒）	耗时（
200	1	10	1313.056	227.908	31.548	97.
200	1	20	1715.053	449.795	51.248	97.
200	1	30	2826.913	665.619	69.118	97.
200	1	40	3373.148	861.583	90.723	97.
200	1	50	4240.311	1085.03	106.353	97.

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（
1	100	50	1631.787	451.644	68.63	95.
1	200	50	3399.833	693.207	141.004	95.
1	300	50	5630.398	847.796	236.91	95.
1	400	50	7562.615	1028.75	335.674	95.

对比图如下：



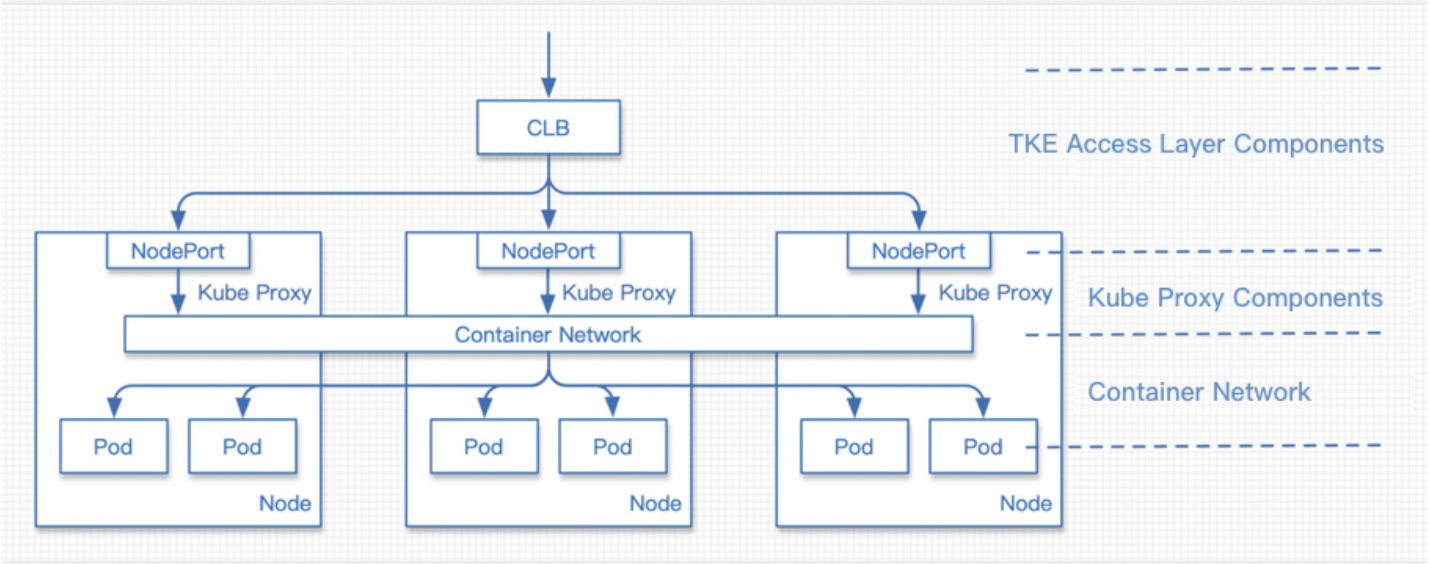
除控制面性能优化外，负载均衡能够直接访问容器网络的 Pod 即为组件业务能力最重要的组成部分，不仅避免了 NAT 转发性能上的损失，同时避免了 NAT 转发带来的各种对集群内业务功能影响，但在启动该项目时还不具备最优

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

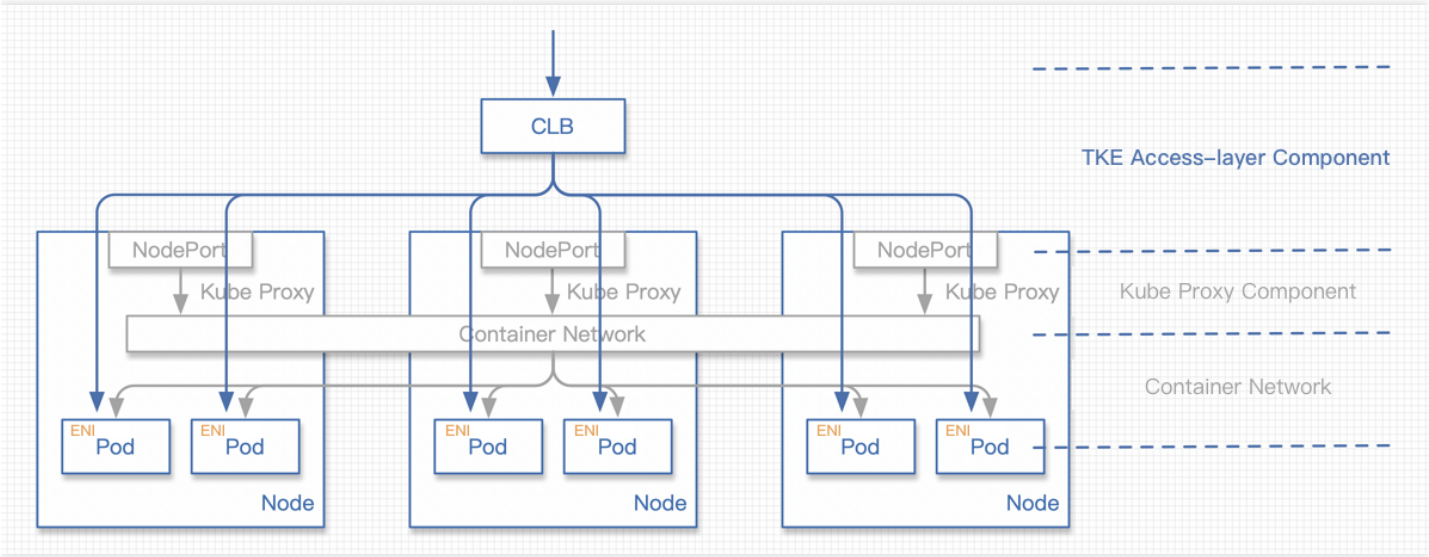
传统 NodePort 模式



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的时，`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` 。

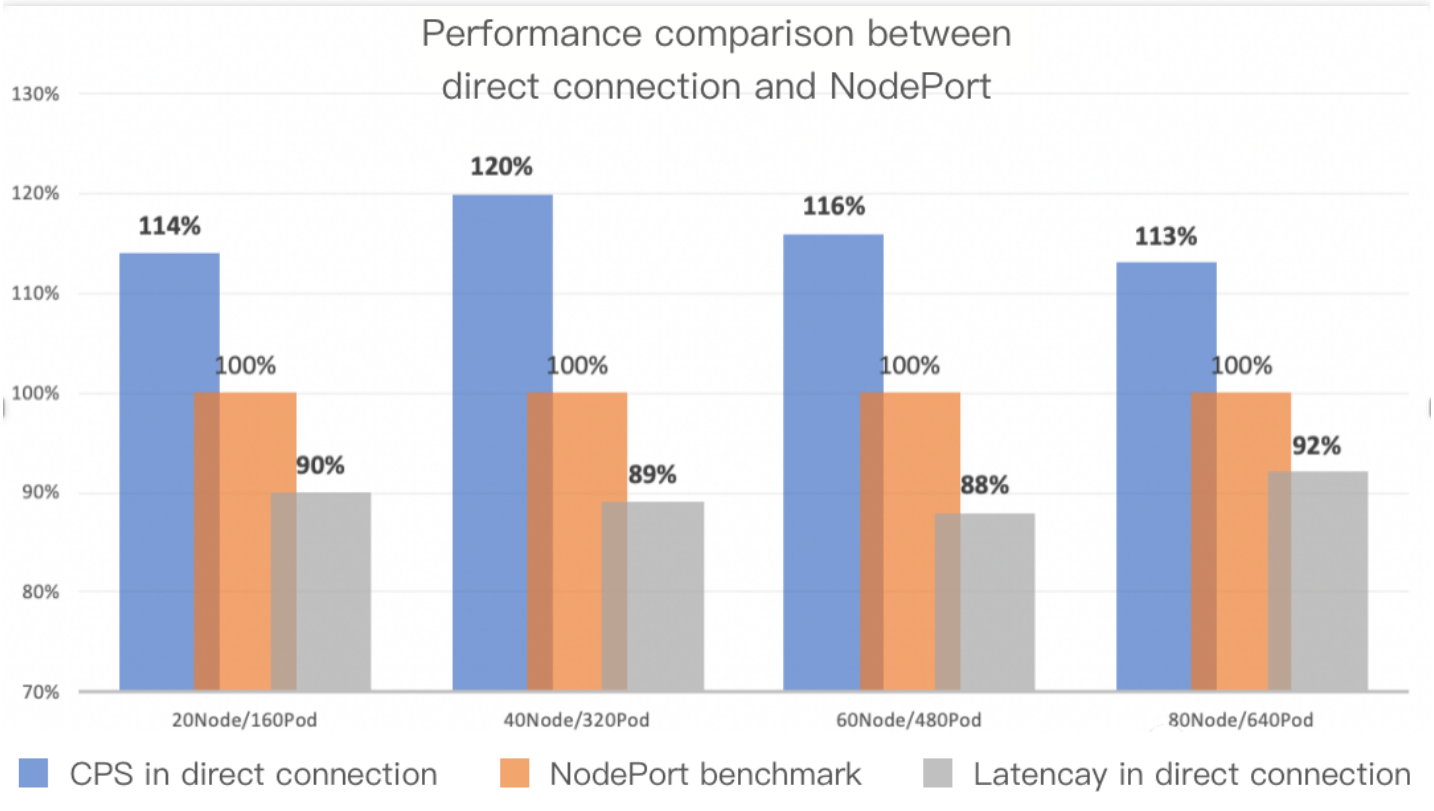
注意：
此回调过程必须是 HTTPS 的，即需要在 `MutatingWebhookConfigurations` 中配置签发请求的 CA，并在服务端配置该 CA 签发的证书。

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。

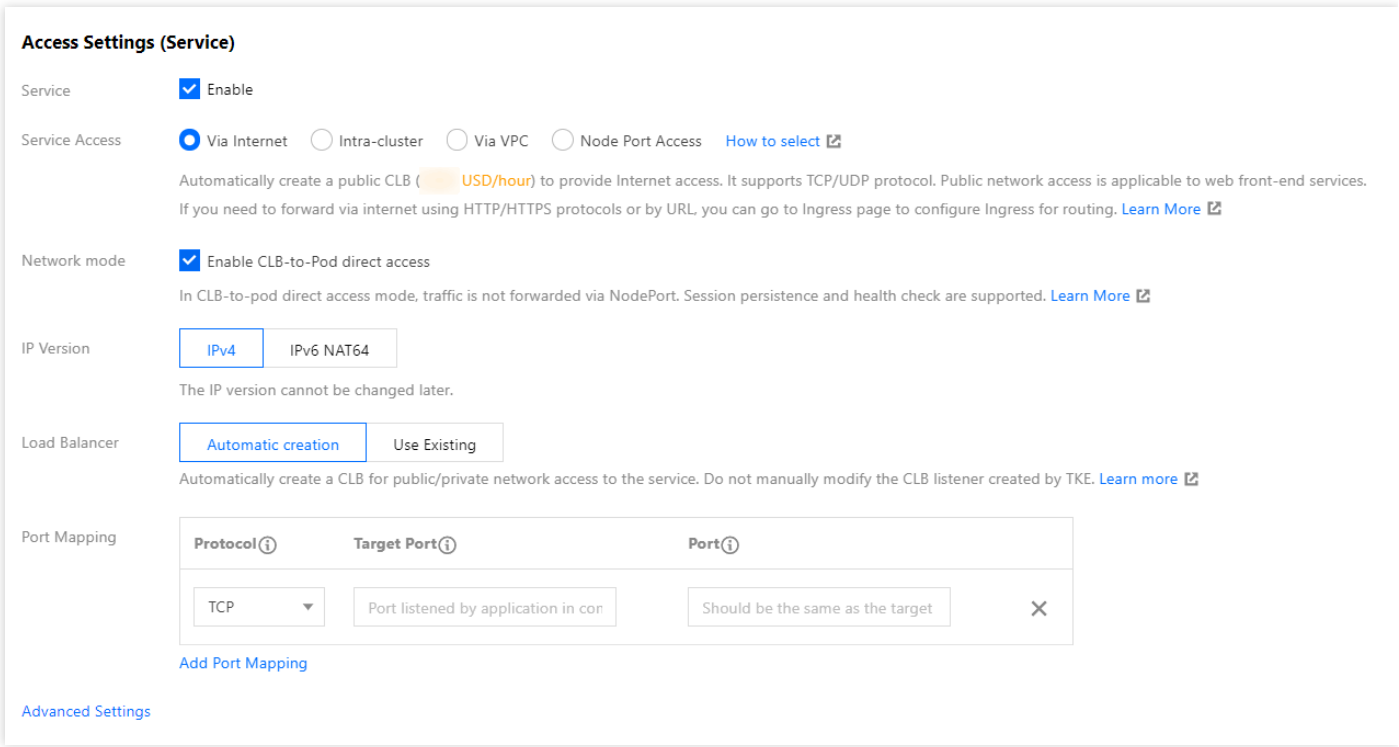
新模式使用指引

前置要求

- `Kubernetes` 集群版本需高于 1.12。
- 集群网络模式需开启 `VPC-CNI` 弹性网卡模式。
- 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，进入 “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，如下图所示：



- **服务访问方式**：选择为**提供公网访问**或**VPC内网访问**。
- **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**创建服务**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 a
pp=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

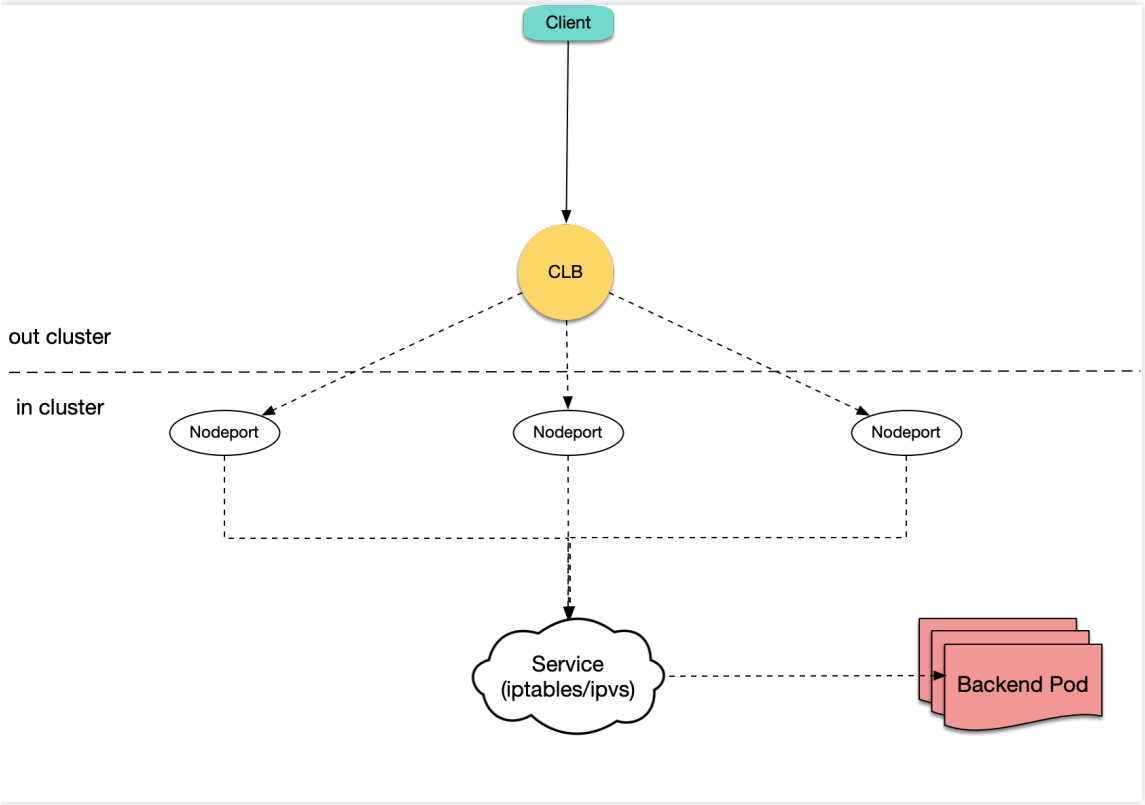
2. Kubernetes Ingress 介绍
3. Kubernetes Deployments 滚动更新策略
4. Kubernetes Pods ReadinessGate 特性
5. Kubernetes 通过 Local 转发获取来源 IP
6. TKE 容器服务 网络模式选型
7. TKE 容器服务 VPC-CNI网络模式
8. TKE 容器服务 配置kubectl并连接集群
9. AWS ALB Ingress Controller
0. GKE 通过独立 NEG 配置容器原生负载平衡

在 TKE 上使用负载均衡直通 Pod

最近更新时间：2023-06-06 16:30:12

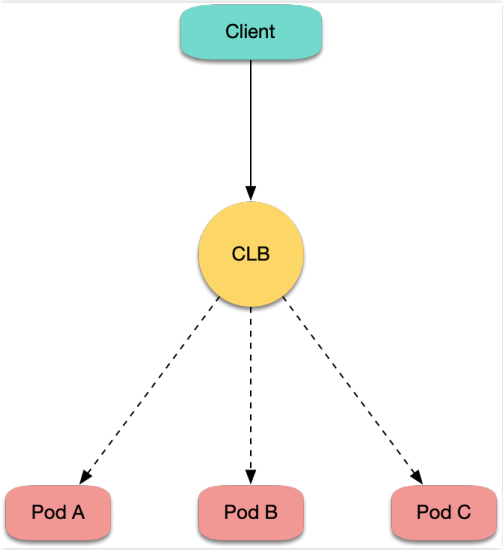
概述

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：



容器服务 TKE 使用相同的方式实现默认 CLB 类型 Service 与 Ingress，但目前还支持 CLB 直通 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 直通 Pod 方式优势

使用 CLB 直通 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 直通 Pod 通常有如下场景：

- 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。
- 集群网络模式必须开启 `VPC-CNI` 弹性网卡模式。可参考 确认是否开启弹性网卡 步骤进行确认。
  目前 CLB 直通 Pod 的实现基于弹性网卡，暂不支持普通的网络模式。

操作步骤

确认是否开启弹性网卡

请对应您的实际情况，按照以下步骤进行操作：

- 若您在创建集群时，“容器网络插件”选择为**VPC-CNI**，则创建的 Pod 已默认使用了弹性网卡，请跳过此步骤。
- 若您在创建集群时，“容器网络插件”选择为**Global Router**后开启了 VPC-CNI 支持。则为两种模式混用，创建的 Pod 默认不使用弹性网卡，需使用 YAML 创建工作负载，为 Pod 指定 `tke.cloud.tencent.com/networks: tke-route-eni` 该 annotation 来声明使用弹性网卡，并为其中一个容器添加例如 `tke.cloud.tencent.com/eni-ip: "1"` 的 requests 与 limits。YAML 示例如下：

```
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
```

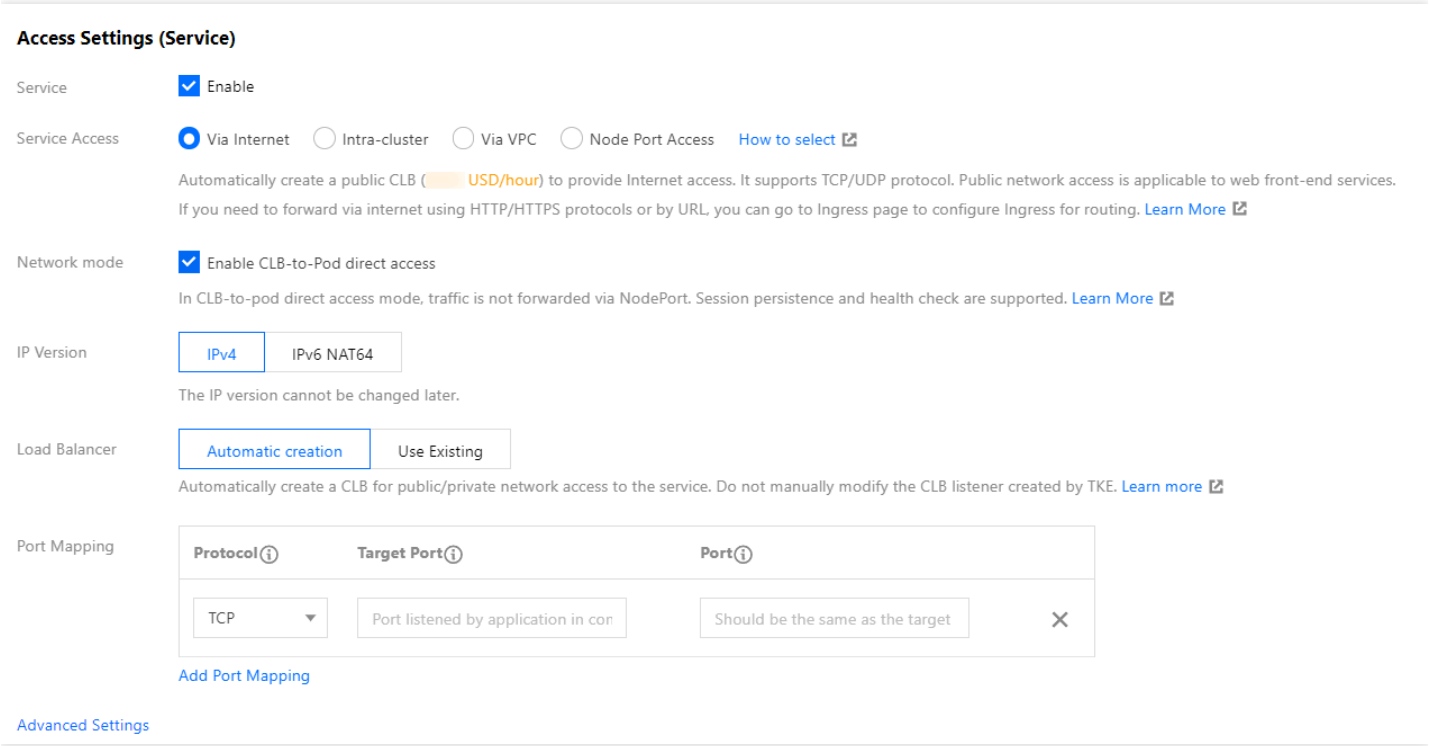
```
spec:
containers:
- image: nginx
name: nginx
resources:
requests:
tke.cloud.tencent.com/eni-ip: "1"
limits:
tke.cloud.tencent.com/eni-ip: "1"
```

创建 Service 时声明直连模式

当使用 CLB 的 Service 暴露服务时，需要声明使用直连模式。步骤如下：

通过控制台创建 Service

如果通过控制台创建 Service，可以勾选“采用负载均衡直连Pod模式”，详情请参见 创建 Service。如下图所示：



通过 YAML 创建 Service

如果通过 YAML 创建 Service，需要为 Service 加上 `service.cloud.tencent.com/direct-access: "true"` 的 annotation。示例如下：

> 说明：
> 如何使用 YAML 创建 Service 请参见 创建 Service。

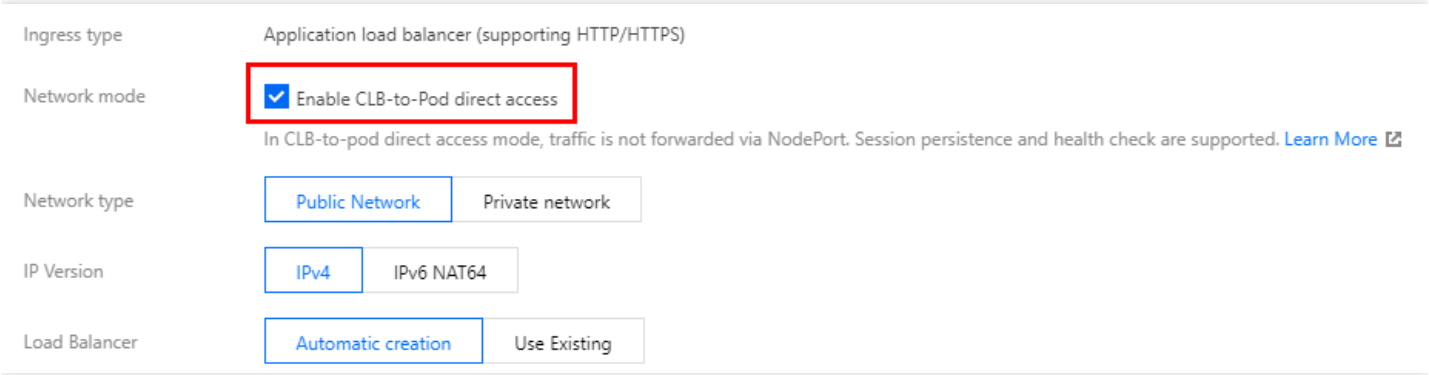
```
apiVersion: v1
kind: Service
metadata:
annotations:
service.cloud.tencent.com/direct-access: "true"
labels:
app: nginx
name: nginx-service-eni
spec:
externalTrafficPolicy: Cluster
ports:
- name: 80-80-no
port: 80
protocol: TCP
targetPort: 80
selector:
app: nginx
sessionAffinity: None
type: LoadBalancer
```

创建 Ingress 时声明直连模式

当使用 Ingress 暴露服务时，同样也需要声明使用直连模式。步骤如下：

通过控制台创建 Ingress

如果通过控制台创建 Ingress，可以勾选“采用负载均衡直连Pod模式”，详情请参见 创建 Ingress。如下图所示：



通过 YAML 创建 Ingress

如果通过 YAML 创建 Ingress，需要为 Ingress 加上 `ingress.cloud.tencent.com/direct-access: "true"` 的 annotation。示例如下：

说明：
如何使用 YAML 创建 Ingress 请参见 创建 Ingress。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
annotations:
ingress.cloud.tencent.com/direct-access: "true"
kubernetes.io/ingress.class: qcloud
name: test-ingress
namespace: default
spec:
rules:
- http:
paths:
- backend:
serviceName: nginx
servicePort: 80
path: /
```

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

在 TKE 中获取客户端真实源 IP

最近更新时间：2022-11-10 10:20:50

> 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 iptbales 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4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。

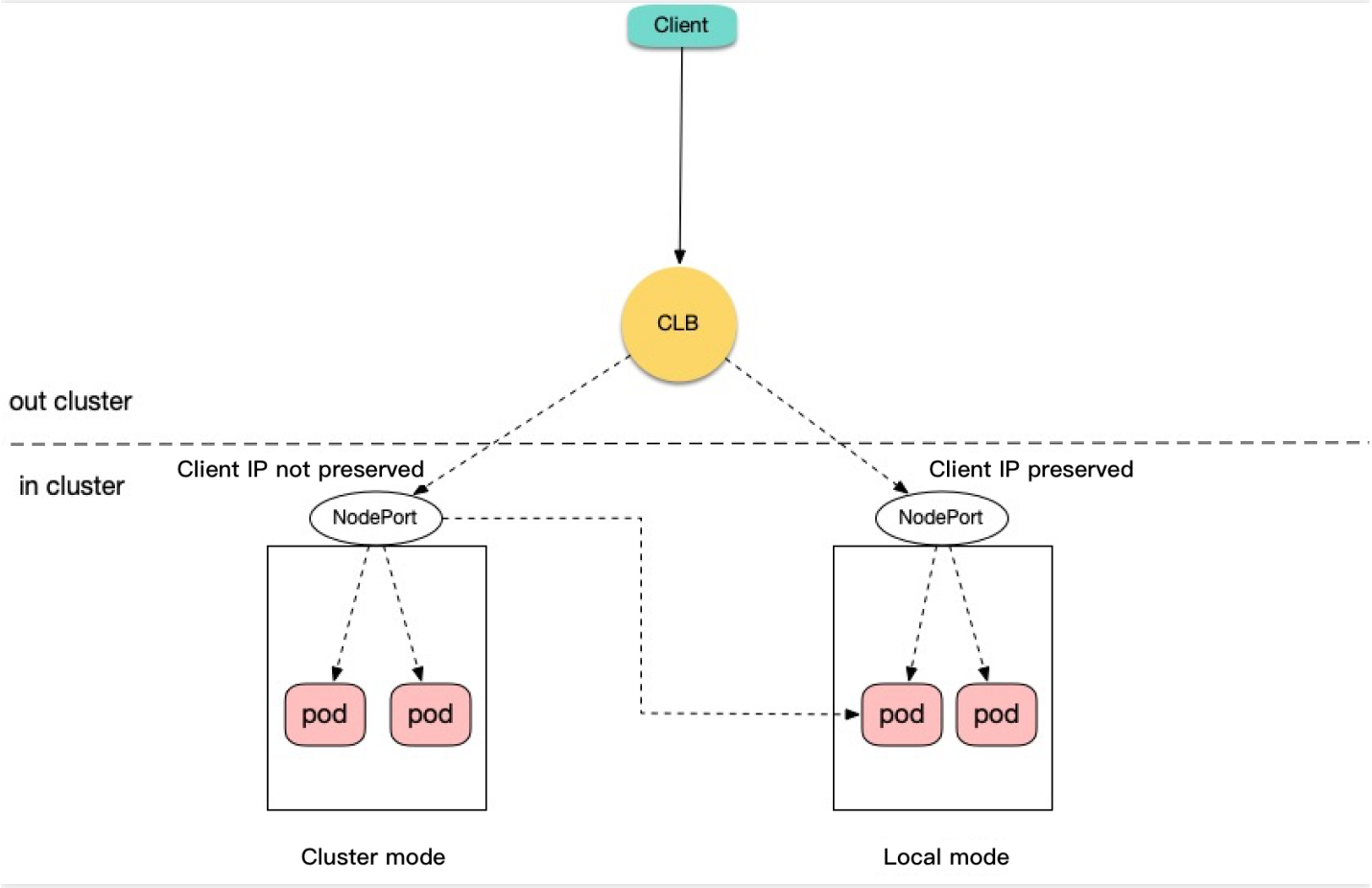
通过 Service 资源的配置选项保留客户端源 IP

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。

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：

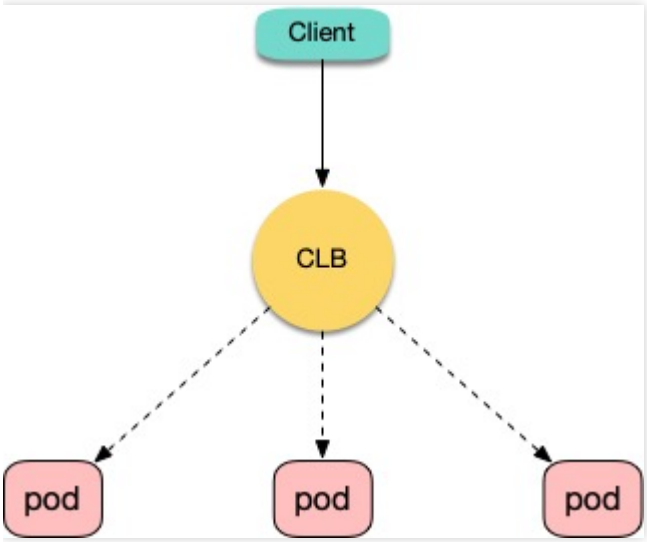
```
apiVersion: v1
kind: Service
metadata:
name: example-Service
spec:
selector:
app: example-Service
ports:
- port: 8765
targetPort: 9376
externalTrafficPolicy: Local
type: LoadBalancer
```

通过 TKE 原生 CLB 直通 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**：集群需开启 VPC-CNI 网络模式。

使用 TKE 原生支持的 CLB 直通 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：



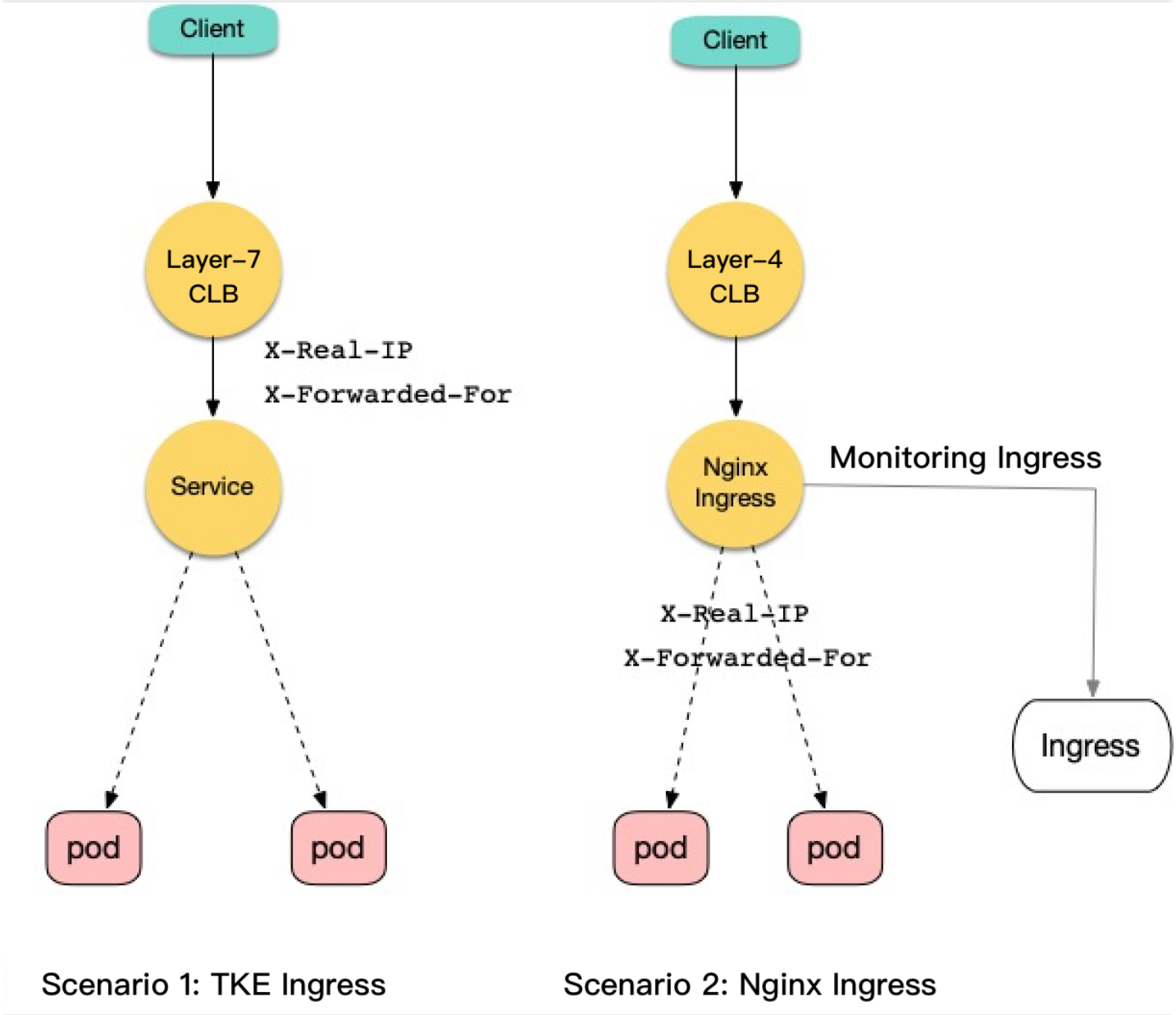
详细介绍和配置请参见 在 TKE 上使用负载均衡直通 Pod。

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。

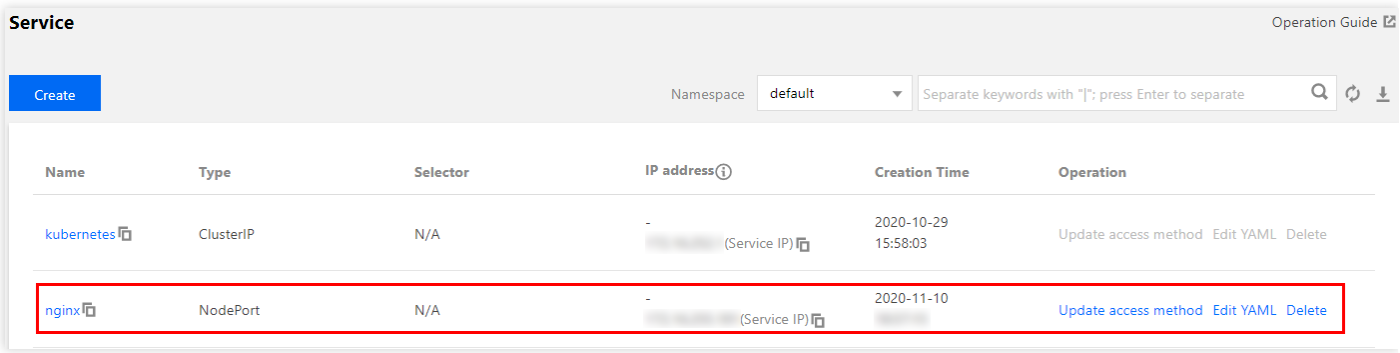
在七层（HTTP/HTTPS）服务转发场景下，可以通过获取 HTTP Header 中 `X-Forwarded-For` 和 `X-Real-IP` 字段的值来获取客户端真实源 IP。TKE 中有两种场景使用方式，原理介绍图如下所示：



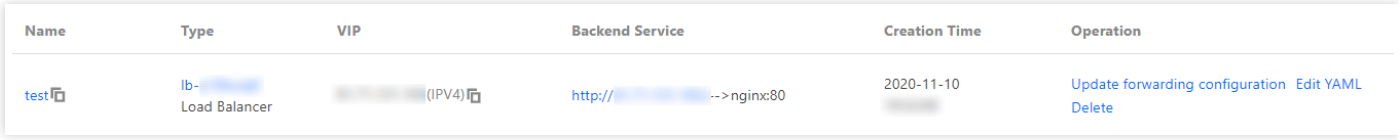
场景一：使用 TKE Ingress 获取真实源 IP

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：

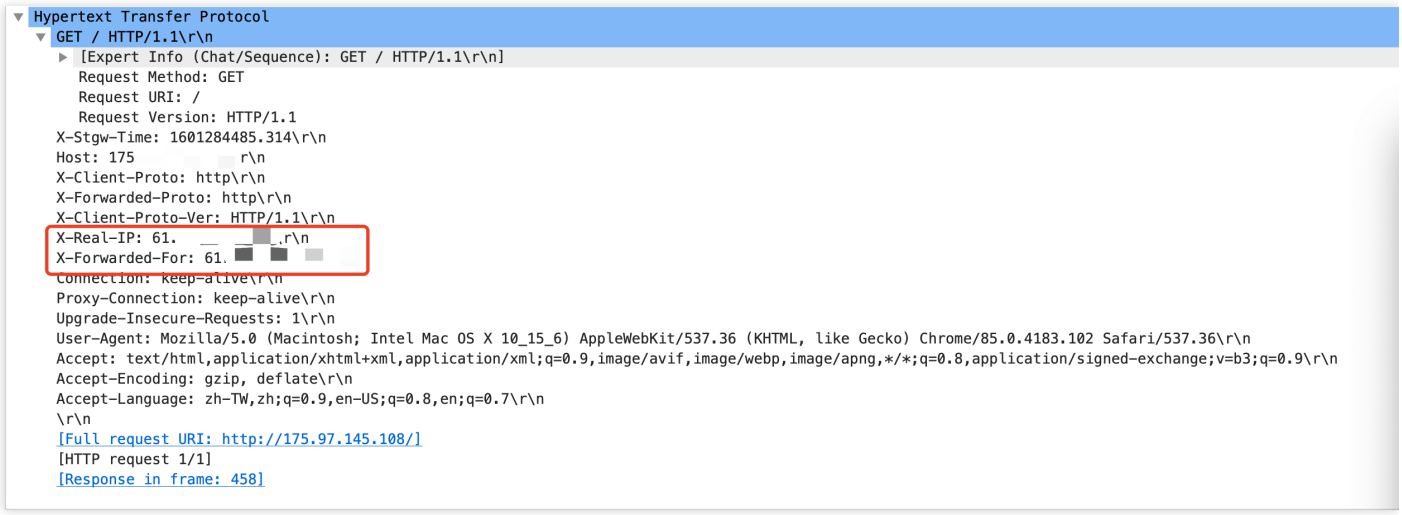
1. 为工作负载创建一个主机端口访问方式的 Service 资源，本文以 nginx 为例。如下图所示：



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：



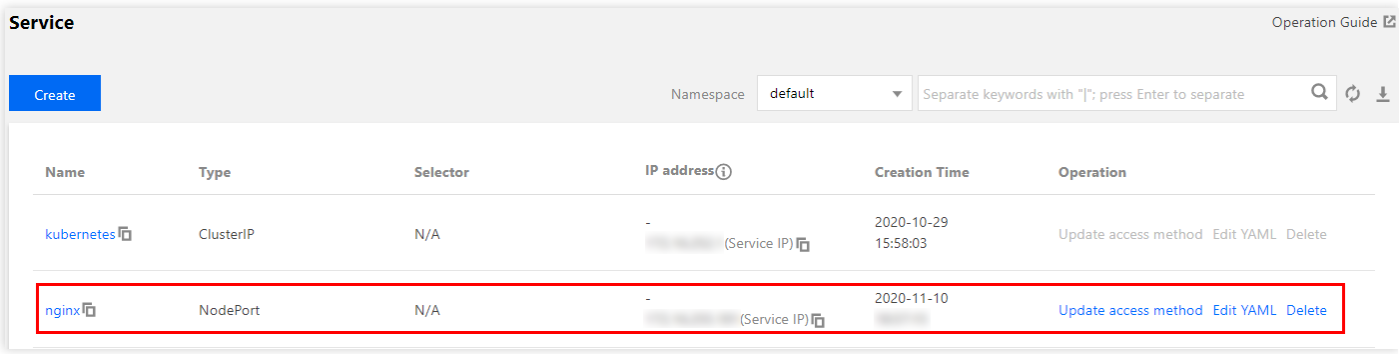
3. 待配置生效后，在后端通过获取 HTTP Header 中的 `X-Forwarded-For` 或 `X-Real-IP` 字段值得到客户端真实源 IP。后端抓包测试结果示例如下图所示：



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：

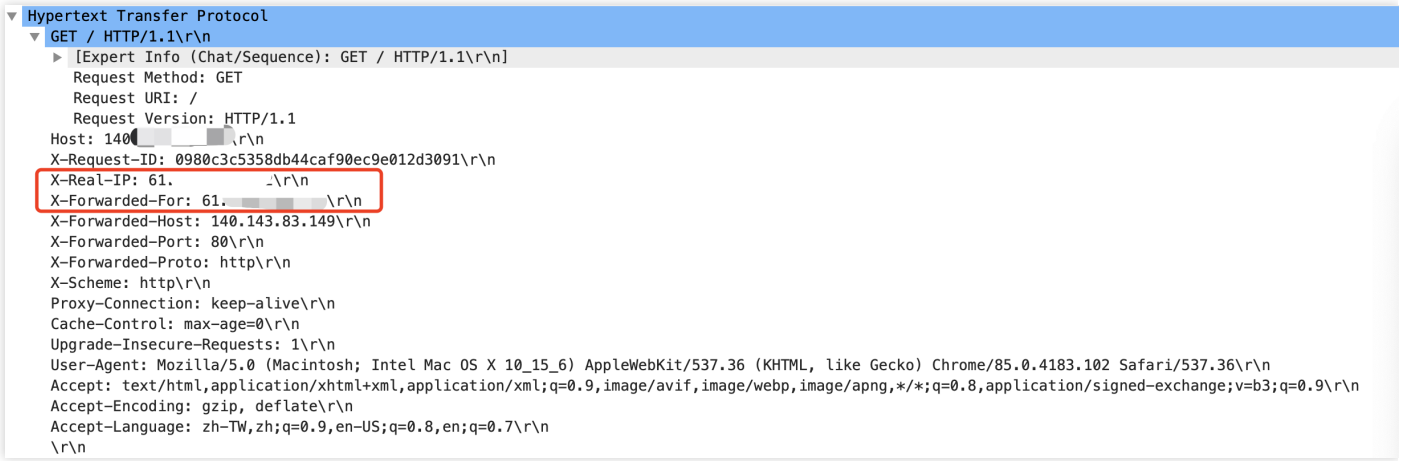
1. Nginx Ingress 可以通过 TKE 应用商店、自定义 YAML 配置或使用官方（helm 安装）方式安装，原理和部署方法请参见 在 TKE 上部署 Nginx Ingress 中的部署方案1或方案3。若选择方案1部署，则需要修改 Nginx Ingress Controller Service 的 `externalTrafficPolicy` 字段值为 `Local` 。
安装完成后，会在容器服务控制台自动为 Nginx Ingress Controller 服务创建一个 CLB（四层）访问入口，如下图所示：



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：

```
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：



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。

参考资料

- 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：2024-02-06 15:42:34

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。

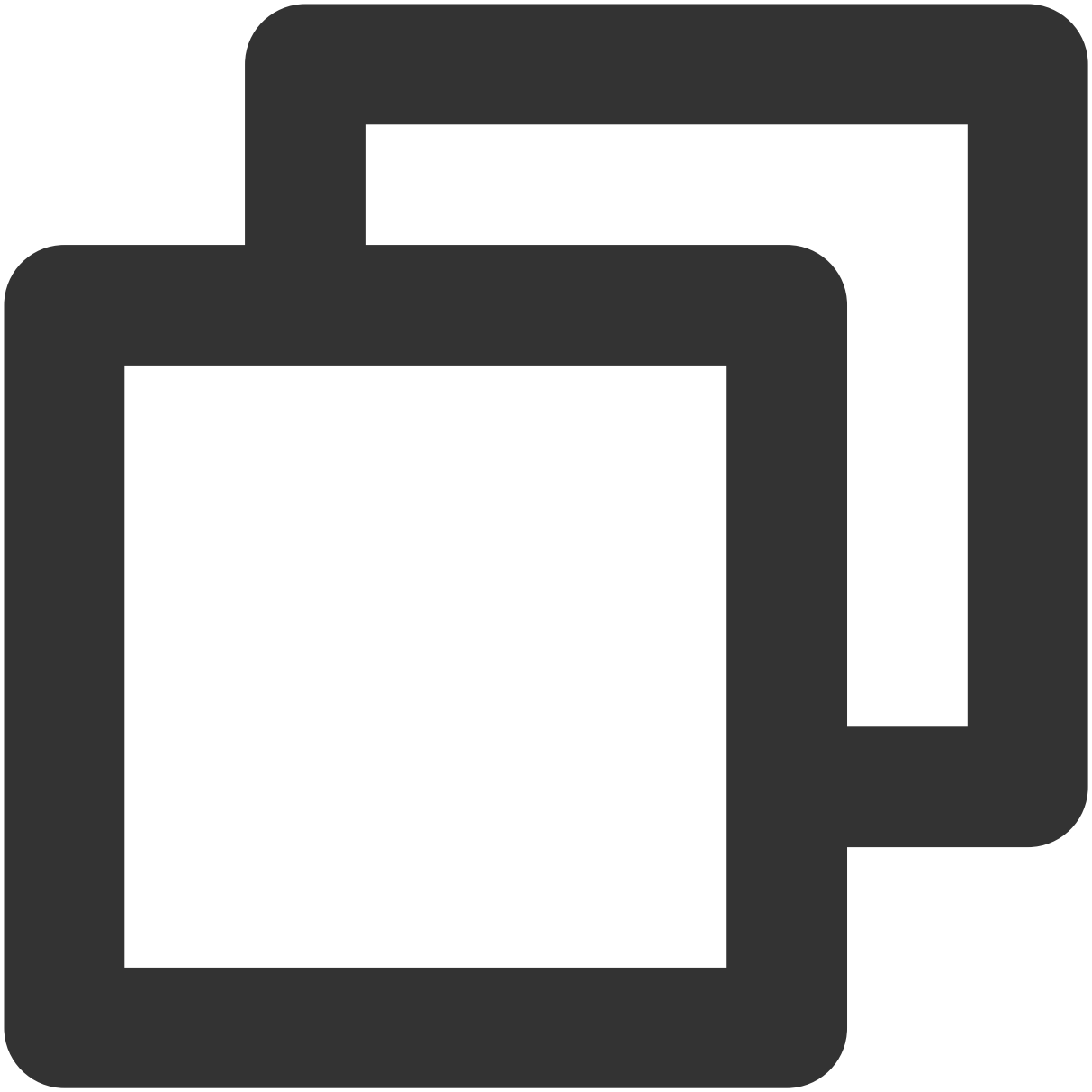
前提条件

已创建 TKE 集群 并能够通过 Kubectl 连接集群。
已安装 Helm。

操作步骤

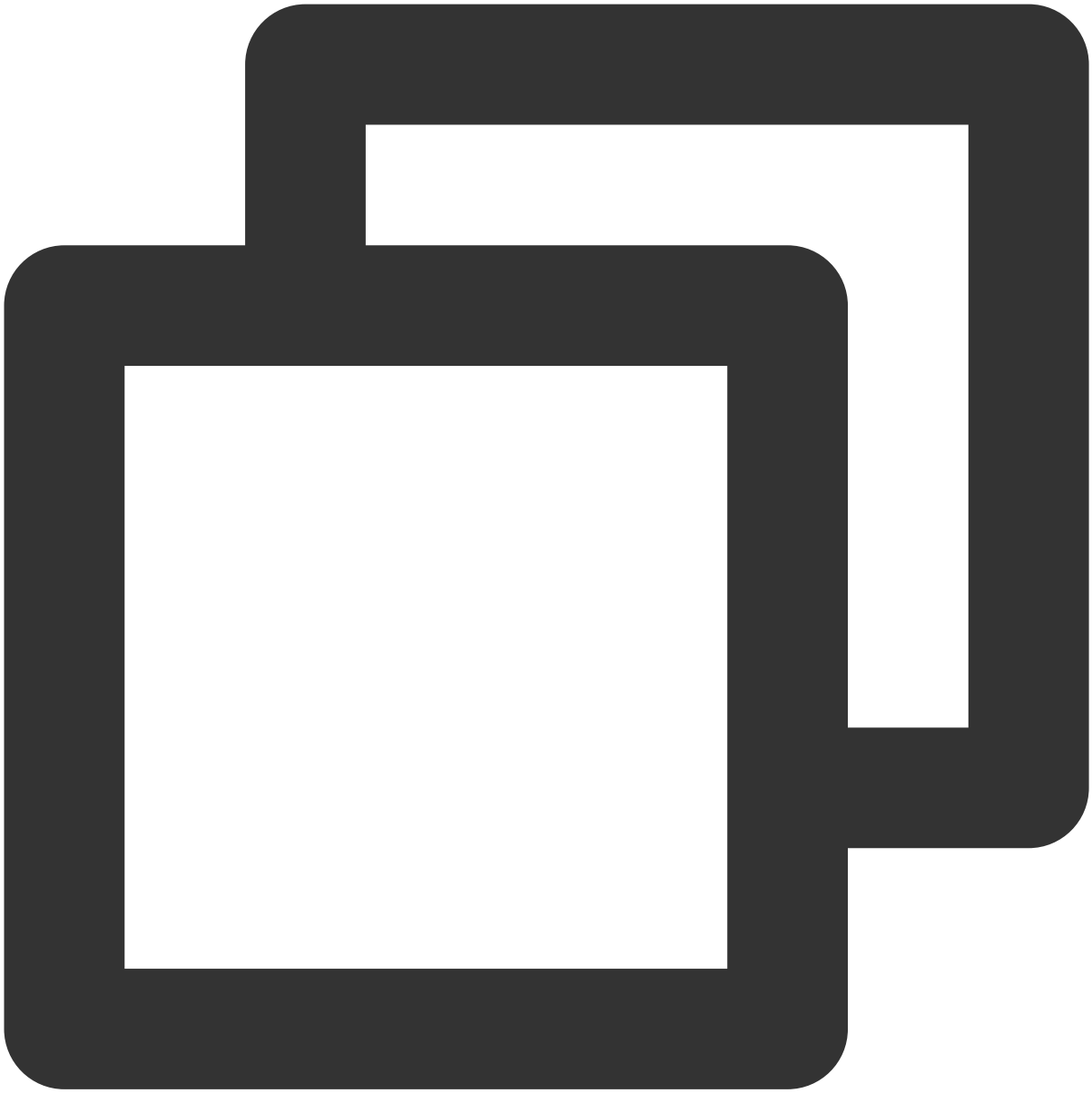
安装 Traefik

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
    service.kubernetes.io/tke-existed-lbid: lb-lb57hvgl # 用此注解绑定提前创建好的 LB,
```

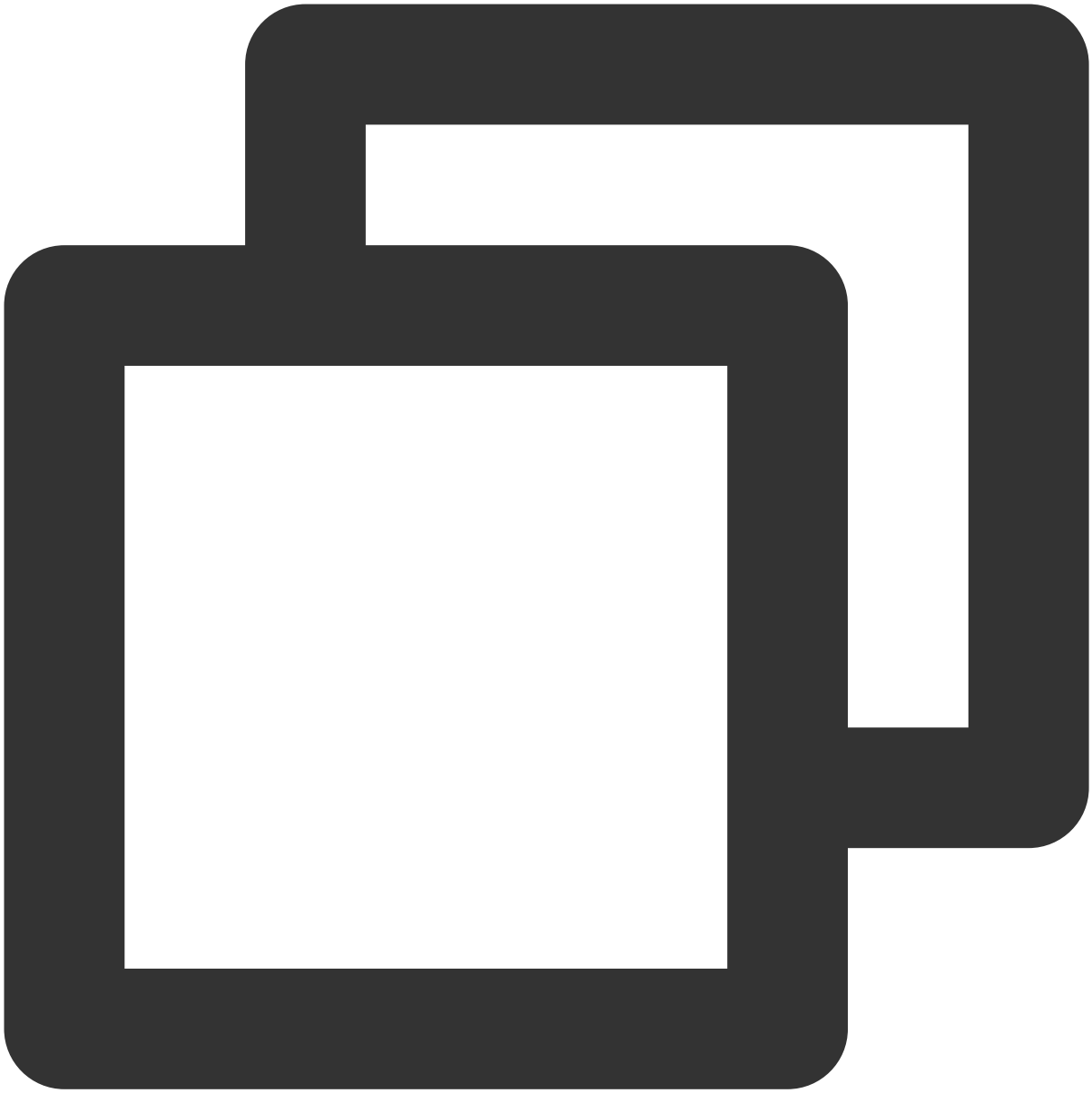
```
ports:
  web:
    expose: true
    exposedPort: 80 # 对外的 HTTP 端口号，使用标准端口号在国内需备案
  websecure:
    expose: true
    exposedPort: 443 # 对外的 HTTPS 端口号，使用标准端口号在国内需备案
deployment:
  enabled: true
  replicas: 1
podAnnotations:
  tke.cloud.tencent.com/networks: "tke-route-eni" # 在 VPC-CNI 与 Global Router 两种
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。

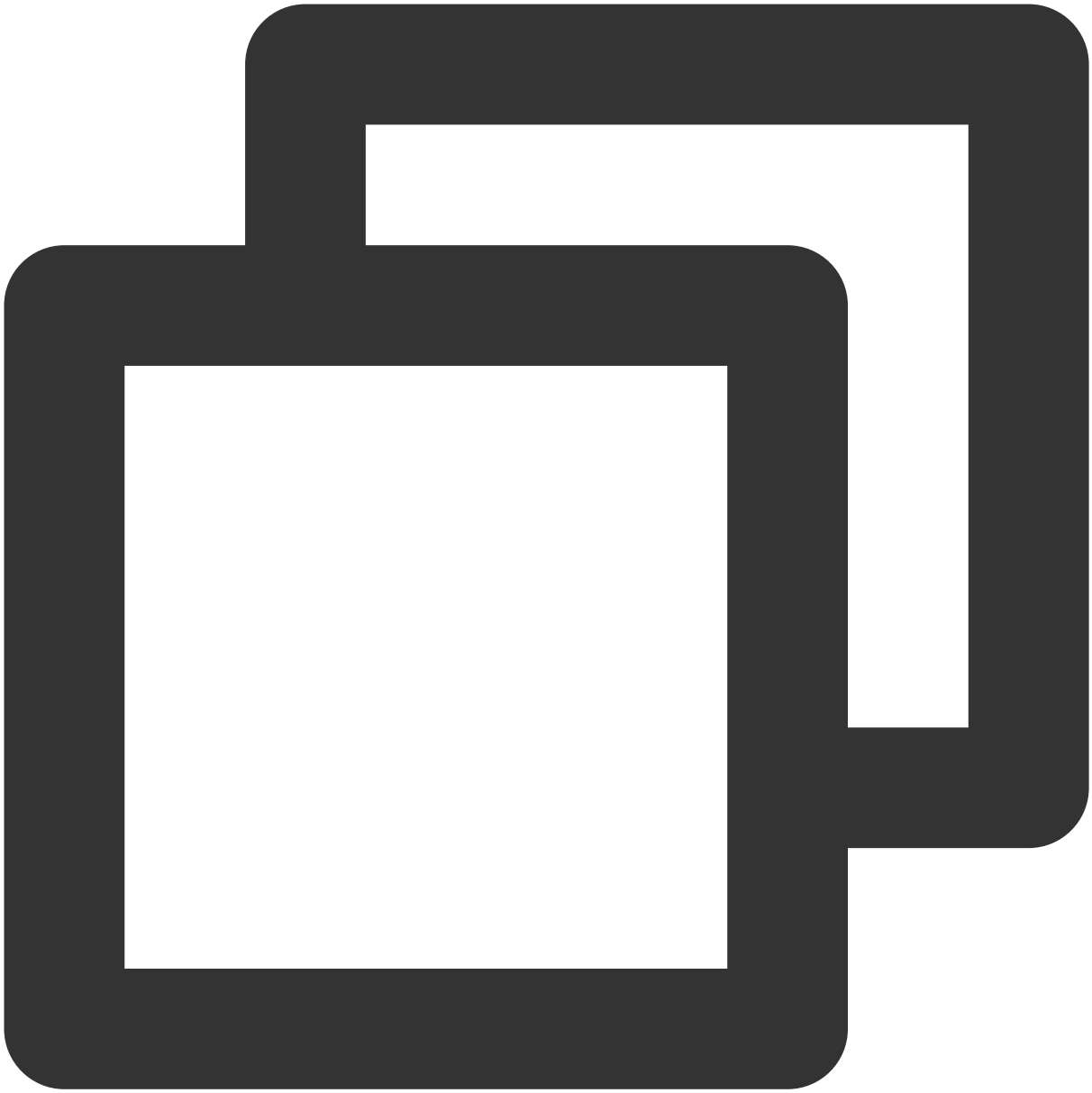
3. 执行以下命令将 Traefik 安装到 TKE 集群。示例如下：

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
```

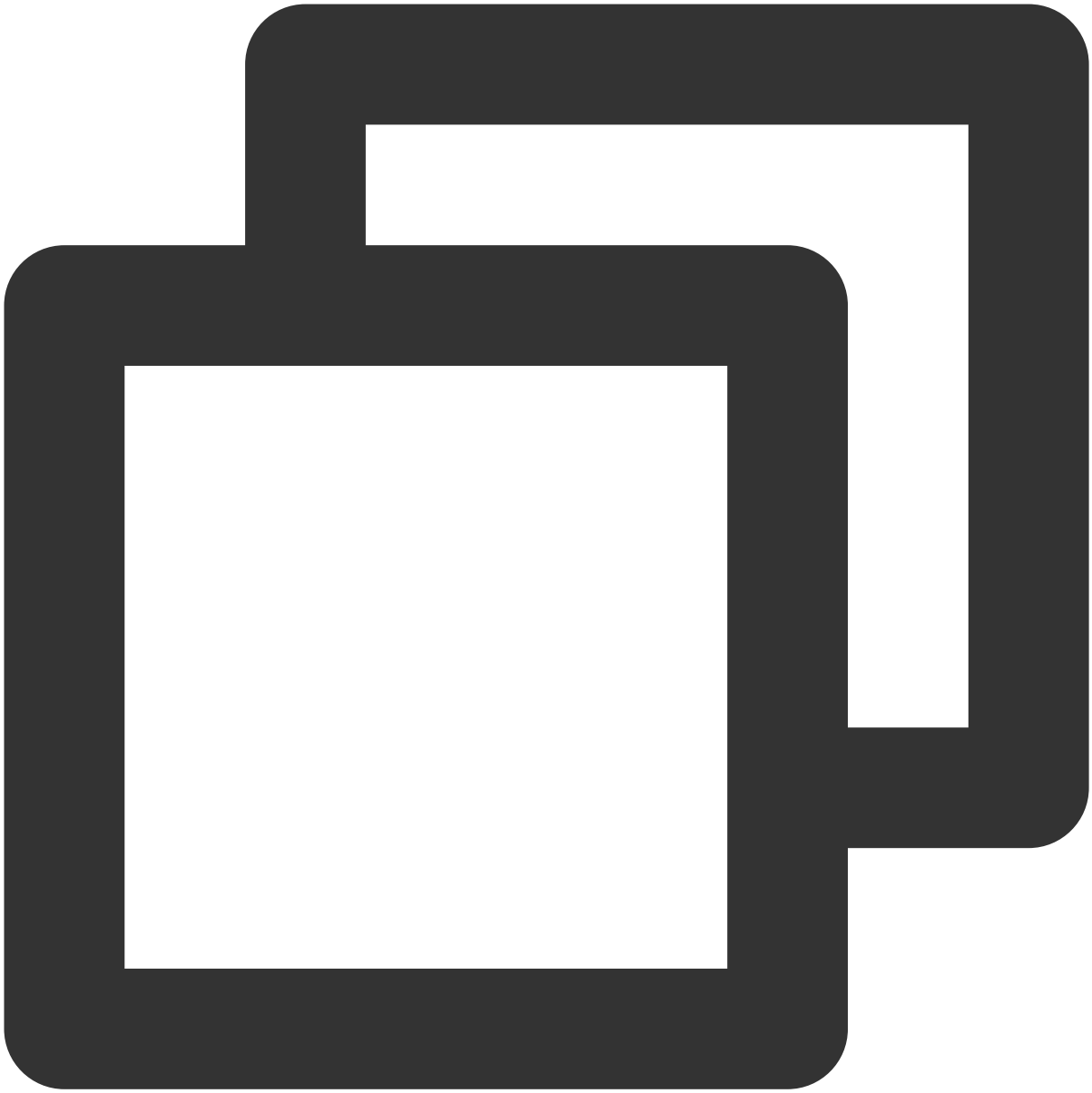
4. 执行以下命令，获取流量入口的 IP 地址（如下为 EXTERNAL-IP 字段）。示例如下：



```
$ kubectl get service -n ingress
NAME      TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)
traefik   LoadBalancer   172.22.252.242   49.233.239.84   80:31650/TCP,443:32288/TC
```

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:
```

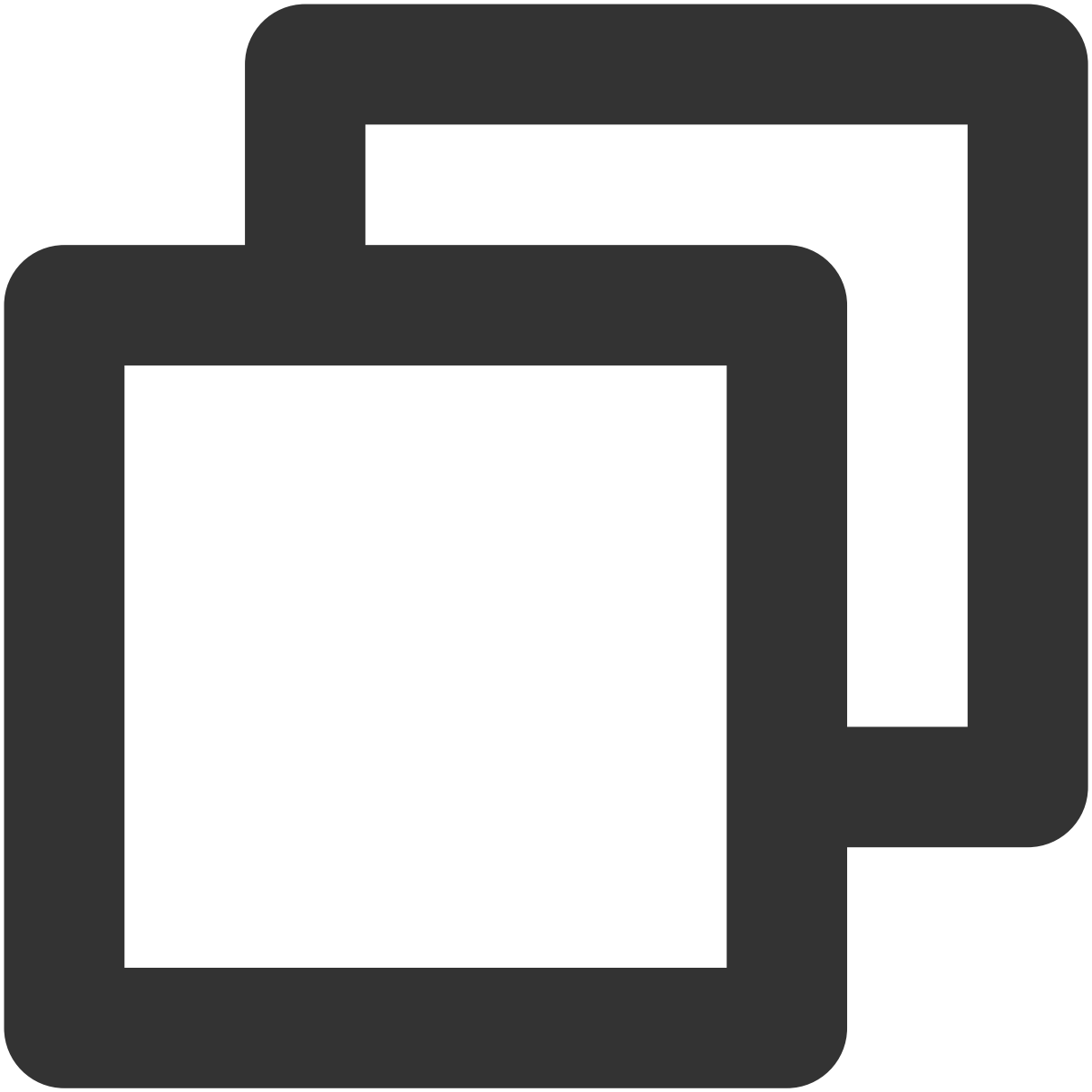
```
      - path: /test
        backend:
          serviceName: nginx
          servicePort: 80
```

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：2023-02-06 15:17:05

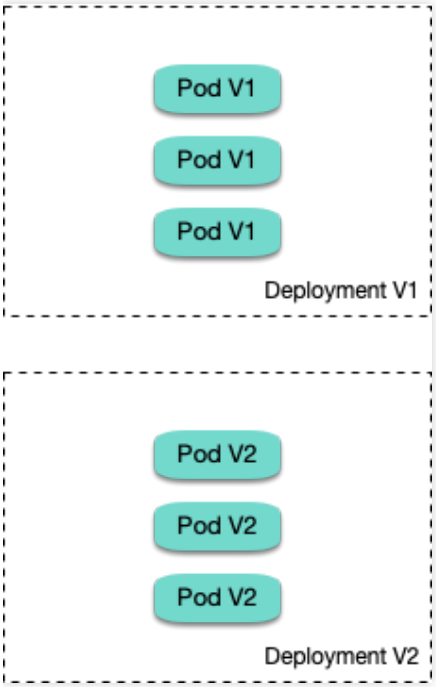
操作场景

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，且不期望集群引入过多的组件或复杂的用法，则可参考本文利用 Kubernetes 原生的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

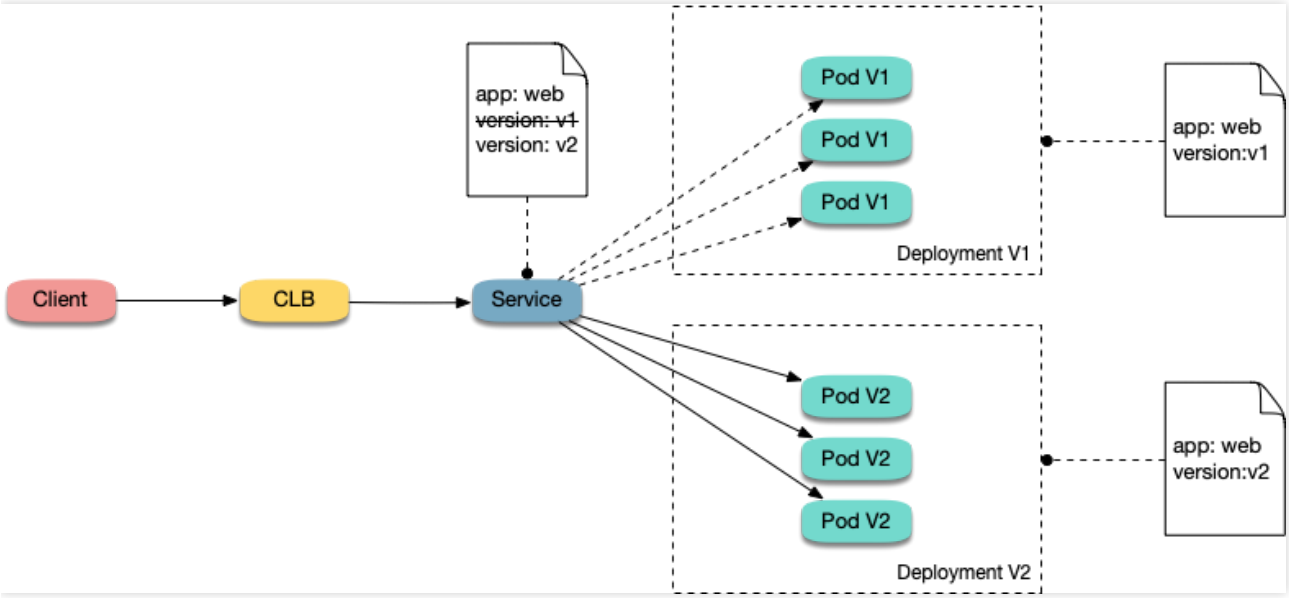
用户通常使用 Deployment、StatefulSet 等 Kubernetes 自带的工作负载来部署业务，每个工作负载管理一组 Pod。以 Deployment 为例，示意图如下：



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

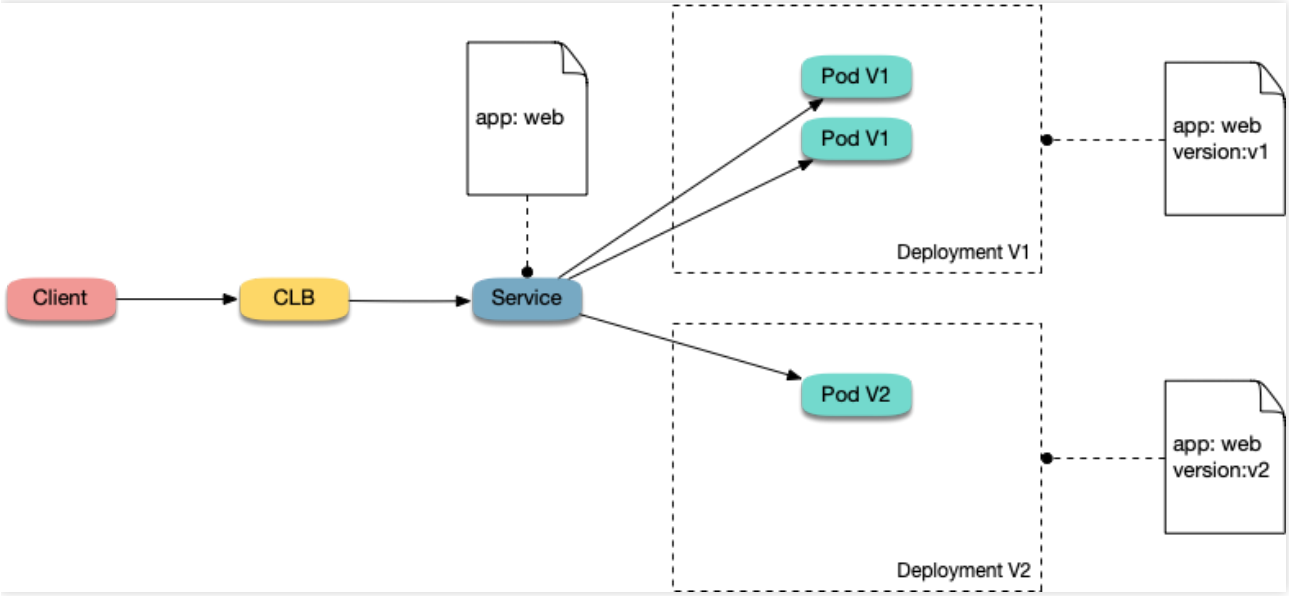
以 Deployment 为例，集群中已部署两个不同版本的 Deployment，其 Pod 拥有共同的 label。但有一个 label 值不同，用于区分不同的版本。Service 使用 selector 选中了其中一个版本的 Deployment 的 Pod，此时通过修改 Service 的 selector 中决定服务版本的 label 的值来改变 Service 后端对应的 Deployment，即可实现让服务从一个版本直接切换到另一个版本。示意图如下：



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 Servcie 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 Servcie：

- 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")
';
}
```

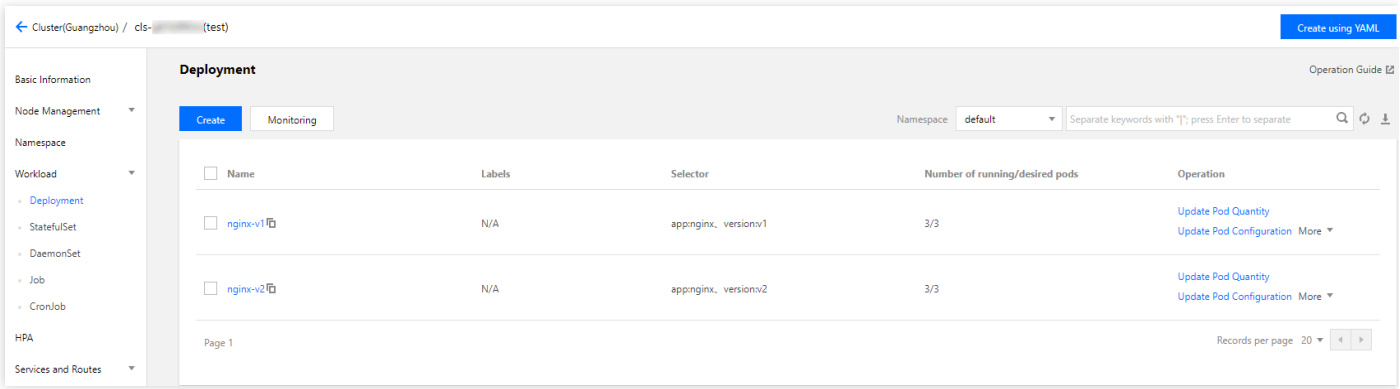
```
}
}
```

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。

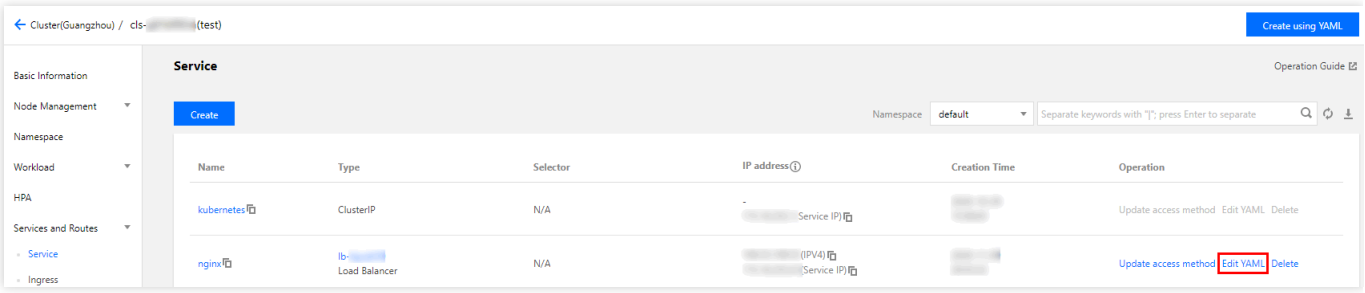
```
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
```

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：

- **通过控制台修改**：

  i. 进入集群详情页，选择左侧**服务与路由** > **Service**。

ii. 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
```

iii. 单击**完成**。

- **通过 kubectl 修改**：

```
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**：
  i. 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**。
  ii. 在 YAML 编辑页面，将 v1 版本的 `.spec.replicas` 修改为1并单击**完成**。
  iii. 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
```

使用 Nginx Ingress 实现金丝雀发布

最近更新时间：2020-11-11 17:22:31

本文将介绍使用 Nginx Ingress 实现金丝雀发布的使用场景、用法详解及实践。

说明：

使用 Nginx Ingress 实现金丝雀发布的集群，需部署 Nginx Ingress 作为 Ingress Controller，并且对外暴露统一的流量入口。详情请参见 在 TKE 上部署 Nginx Ingress。

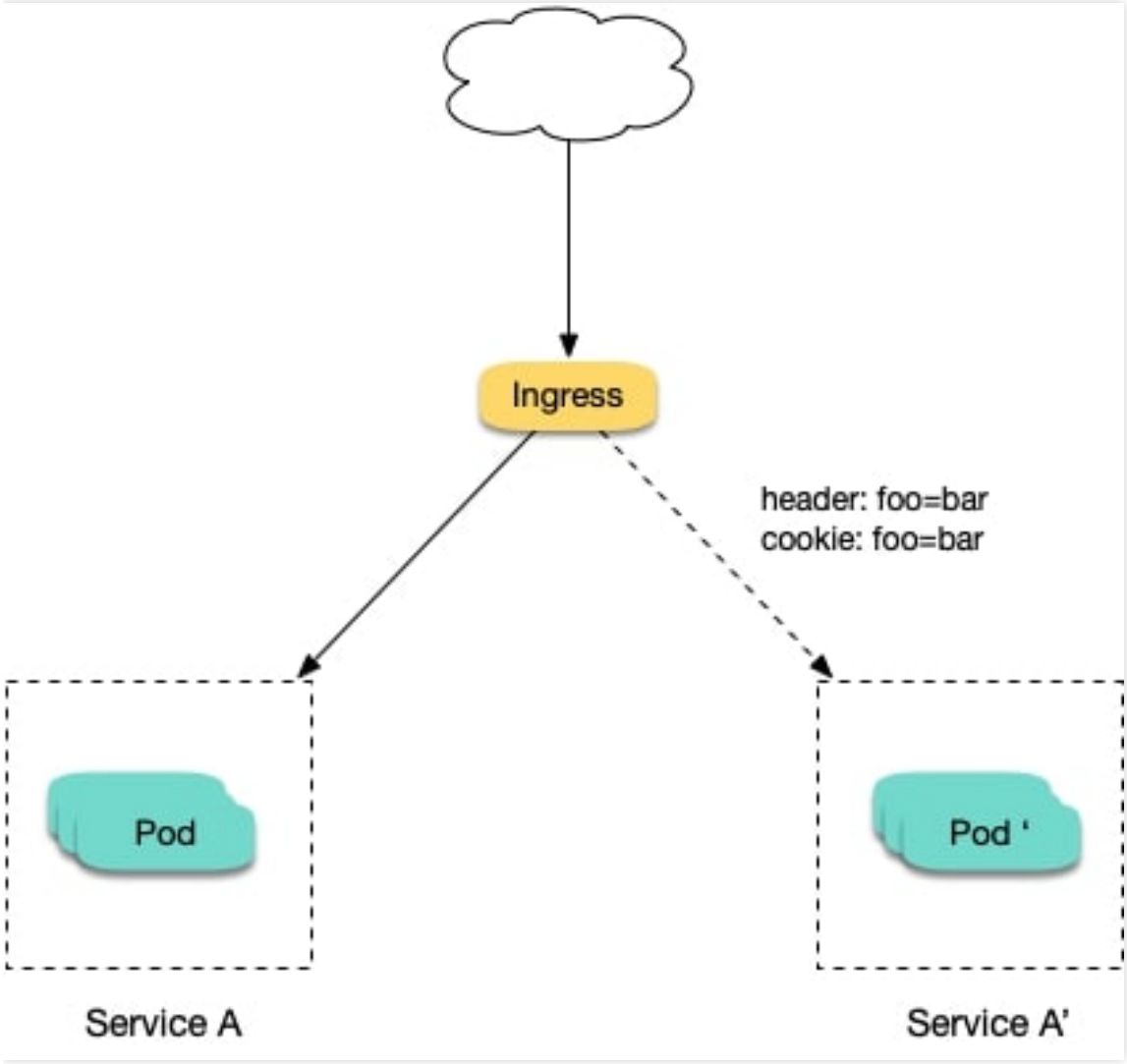
使用场景

使用 Nginx Ingress 实现金丝雀发布适用场景主要取决于业务流量切分的策略，目前 Nginx Ingress 支持基于 Header、Cookie 和服务权重3种流量切分的策略，基于这3种策略可实现以下两种发布场景：

场景1: 灰度新版本到部分用户

假设线上已运行了一套对外提供7层服务的 Service A，此时需上线开发的新版本 Service A'，但不期望直接替换原有的 Service A，仅灰度部分用户，待运行一段时间足够稳定后再逐渐全量上线新版本，平滑下线旧版本。
针对此场景可使用 Nginx Ingress 基于 Header 或 Cookie 进行流量切分的策略来发布，业务使用 Header 或 Cookie 来标识不同类型的用户，并通过配置 Ingress 来实现让带有指定 Header 或 Cookie 的请求被转发到新版本，其它请

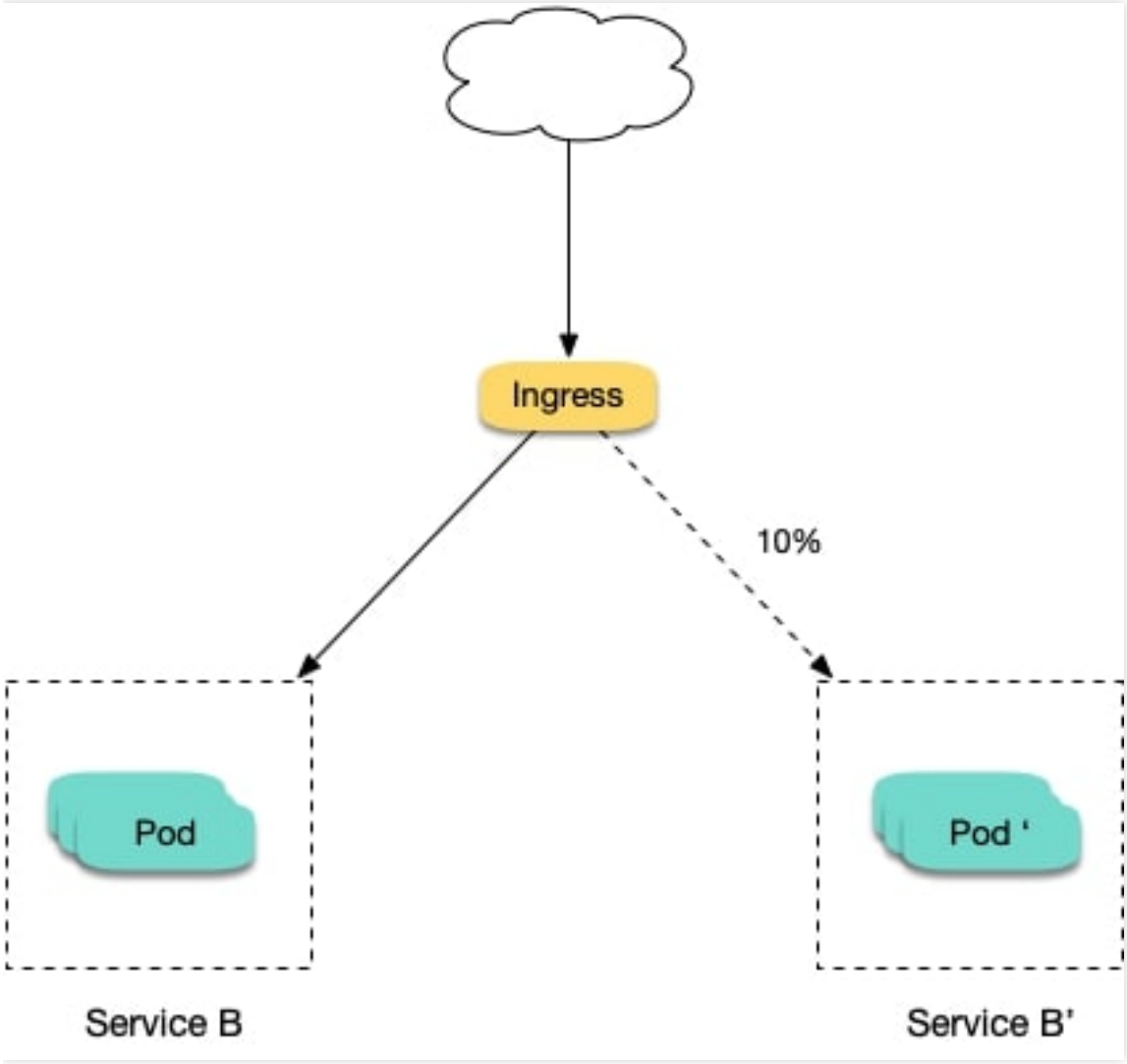
求仍然转发到旧版本，从而将新版本灰度给部分用户。示意图如下：



场景2: 切分一定比例的流量到新版本

假设线上已运行了一套对外提供7层服务的 Service B，此时修复了 Service B 的部分问题，需灰度上线新版本 Service B'。但不期望直接替换原有的 Service B，需先切换10%的流量至新版本，待运行一段时间足够稳定后再逐渐

加大新版本流量比例直至完全替换旧版本，最终平滑下线旧版本。示意图如下：



注解说明

通过给 Ingress 资源指定 Nginx Ingress 所支持的 annotation 可实现金丝雀发布。需给服务创建2个 Ingress，其中1个常规 Ingress，另1个为带 `nginx.ingress.kubernetes.io/canary: "true"` 固定的 annotation 的 Ingress，称为 Canary Ingress。Canary Ingress 一般代表新版本的服务，结合另外针对流量切分策略的 annotation 一起配置即可实现多种场景的金丝雀发布。以下为相关 annotation 的详细介绍：

- **`nginx.ingress.kubernetes.io/canary-by-header`**
  表示如果请求头中包含指定的 header 名称，并且值为 `always` ，就将该请求转发给该 Ingress 定义的对应后端服务。如果值为 `never` 则不转发，可以用于回滚到旧版。如果为其他值则忽略该 annotation。

- **`nginx.ingress.kubernetes.io/canary-by-header-value`**
  该 annotation 可以作为 `canary-by-header` 的补充，可指定请求头为自定义值，包含但不限于 `always` 或 `never` 。当请求头的值命中指定的自定义值时，请求将会转发给该 Ingress 定义的对应后端服务，如果是其它值则忽略该 annotation。

- **`nginx.ingress.kubernetes.io/canary-by-header-pattern`**
  与 `canary-by-header-value` 类似，区别为该 annotation 用正则表达式匹配请求头的值，而不是只固定某一个值。如果该 annotation 与 `canary-by-header-value` 同时存在，该 annotation 将被忽略。

- **`nginx.ingress.kubernetes.io/canary-by-cookie`**
  与 `canary-by-header` 类似，该 annotation 用于 cookie，仅支持 `always` 和 `never` 。

- **`nginx.ingress.kubernetes.io/canary-weight`**
  表示 Canary Ingress 所分配流量的比例的百分比，取值范围 [0-100]。例如，设置为10，则表示分配10%的流量给 Canary Ingress 对应的后端服务。

说明：

- 以上规则会按优先顺序进行评估，优先顺序为： `canary-by-header -> canary-by-cookie -> canary-weight` 。
- 当 Ingress 被标记为 Canary Ingress 时，除了 `nginx.ingress.kubernetes.io/load-balance` 和 `nginx.ingress.kubernetes.io/upstream-hash-by` 外，所有其他非 Canary 注释都将被忽略。

使用示例

注意：

以下示例环境以 TKE 集群为例，您可通过示例快速上手 Nginx Ingress 的金丝雀发布。需注意以下事项：

- 相同服务的 Canary Ingress 仅能够定义一个，导致后端服务最多支持两个版本。

2. Ingress 里必须配置域名，否则不会有效果。
3. 即便流量完全切到了 Canary Ingress 上，旧版服务仍需存在，否在会出现报错。

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 Servcie：

- 方式1：在单击 TKE 或 EKS 集群详情页右上角的【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` 。

部署两个版本的服务

1. 在集群中部署第一个版本的 Deployment，本文以 nginx-v1 为例。YAML 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-v1
spec:
replicas: 1
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
```

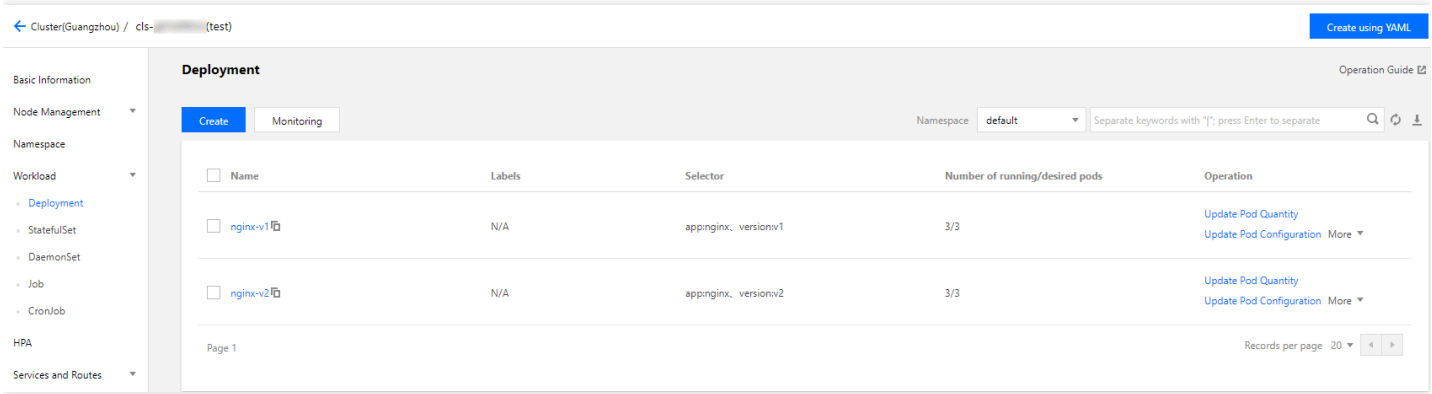
```
metadata:
labels:
app: nginx
version: v1
name: nginx-v1
data:
nginx.conf: |-
worker_processes 1;

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")
';
}
}
}

---

apiVersion: v1
kind: Service
metadata:
name: nginx-v1
spec:
type: ClusterIP
ports:
- port: 80
protocol: TCP
name: http
selector:
app: nginx
version: v1
```

2. 再部署第二个版本的 Deployment，本文以 nginx-v2 为例。YAML 示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-v2
spec:
replicas: 1
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")
';
}
}
}

---

apiVersion: v1
kind: Service
metadata:
name: nginx-v2
spec:
type: ClusterIP
ports:
- port: 80
protocol: TCP
name: http
selector:
app: nginx
version: v2
```

您可登录 容器服务控制台，在集群的工作负载详情页查看部署情况。如下图所示：



3. 创建 Ingress，对外暴露服务，指向 v1 版本的服务。YAML 示例如下：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
name: nginx
annotations:
kubernetes.io/ingress.class: nginx
spec:
rules:
- host: canary.example.com
http:
paths:
- backend:
serviceName: nginx-v1
servicePort: 80
path: /
```

4. 执行以下命令，进行访问验证。

```
curl -H "Host: canary.example.com" http://EXTERNAL-IP # EXTERNAL-IP 替换为 Nginx I
ngress 自身对外暴露的 IP
```

返回结果如下：

```
nginx-v1
```

基于 Header 的流量切分

创建 Canary Ingress，指定 v2 版本的后端服务，并增加 annotation。实现仅将带有名为 Region 且值为 cd 或 sz 的请求头的请求转发给当前 Canary Ingress，模拟灰度新版本给成都和深圳地域的用户。YAML 示例如下：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
annotations:
kubernetes.io/ingress.class: nginx
nginx.ingress.kubernetes.io/canary: "true"
nginx.ingress.kubernetes.io/canary-by-header: "Region"
nginx.ingress.kubernetes.io/canary-by-header-pattern: "cd|sz"
name: nginx-canary
spec:
rules:
```

```
- host: canary.example.com
http:
paths:
- backend:
serviceName: nginx-v2
servicePort: 80
path: /
```

执行以下命令，进行访问测试。

```
$ curl -H "Host: canary.example.com" -H "Region: cd" http://EXTERNAL-IP # EXTERNAL-IP 替换为 Nginx Ingress 自身对外暴露的 IP
nginx-v2
$ curl -H "Host: canary.example.com" -H "Region: bj" http://EXTERNAL-IP
nginx-v1
$ curl -H "Host: canary.example.com" -H "Region: cd" http://EXTERNAL-IP
nginx-v2
$ curl -H "Host: canary.example.com" http://EXTERNAL-IP
nginx-v1
```

可查看当仅有 header `Region` 为 cd 或 sz 的请求才由 v2 版本服务响应。

基于 Cookie 的流量切分

使用 Cookie 则无法自定义 value，以模拟灰度成都地域用户为例，仅将带有名为 `user_from_cd` 的 Cookie 的请求转发给当前 Canary Ingress。YAML 示例如下：

说明：

若您已配置以上步骤创建 Canary Ingress，则请删除后再参考本步骤创建。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
annotations:
kubernetes.io/ingress.class: nginx
nginx.ingress.kubernetes.io/canary: "true"
nginx.ingress.kubernetes.io/canary-by-cookie: "user_from_cd"
name: nginx-canary
spec:
rules:
- host: canary.example.com
http:
paths:
- backend:
```

```
serviceName: nginx-v2
servicePort: 80
path: /
```

执行以下命令，进行访问测试。

```
$ curl -s -H "Host: canary.example.com" --cookie "user_from_cd=always" http://EXT
ERNAL-IP # EXTERNAL-IP 替换为 Nginx Ingress 自身对外暴露的 IP
nginx-v2
$ curl -s -H "Host: canary.example.com" --cookie "user_from_bj=always" http://EXT
ERNAL-IP
nginx-v1
$ curl -s -H "Host: canary.example.com" http://EXTERNAL-IP
nginx-v1
```

可查看当仅有 cookie `user_from_cd` 为 `always` 的请求才由 v2 版本的服务响应。

基于服务权重的流量切分

使用基于服务权重的 Canary Ingress 时，直接定义需要导入的流量比例即可。以导入10%流量到 v2 版本为例，YAML 示例如下：

说明：

若您已配置以上步骤创建 Canary Ingress，则请删除后再参考本步骤创建。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
annotations:
kubernetes.io/ingress.class: nginx
nginx.ingress.kubernetes.io/canary: "true"
nginx.ingress.kubernetes.io/canary-weight: "10"
name: nginx-canary
spec:
rules:
- host: canary.example.com
http:
paths:
- backend:
serviceName: nginx-v2
servicePort: 80
path: /
```

执行以下命令，进行访问测试。

```
$ for i in {1..10}; do curl -H "Host: canary.example.com" http://EXTERNAL-IP; done
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v1
nginx-v2
nginx-v1
nginx-v1
nginx-v1
```

可查看，有十分之一的几率由 v2 版本的服务响应，符合10%服务权重的设置。

参考资料

- Nginx Ingress 金丝雀注解官方文档
- 在 TKE 上部署 Nginx Ingress

日志
TKE 日志采集最佳实践

最近更新时间：2021-07-15 14:18:35

概述

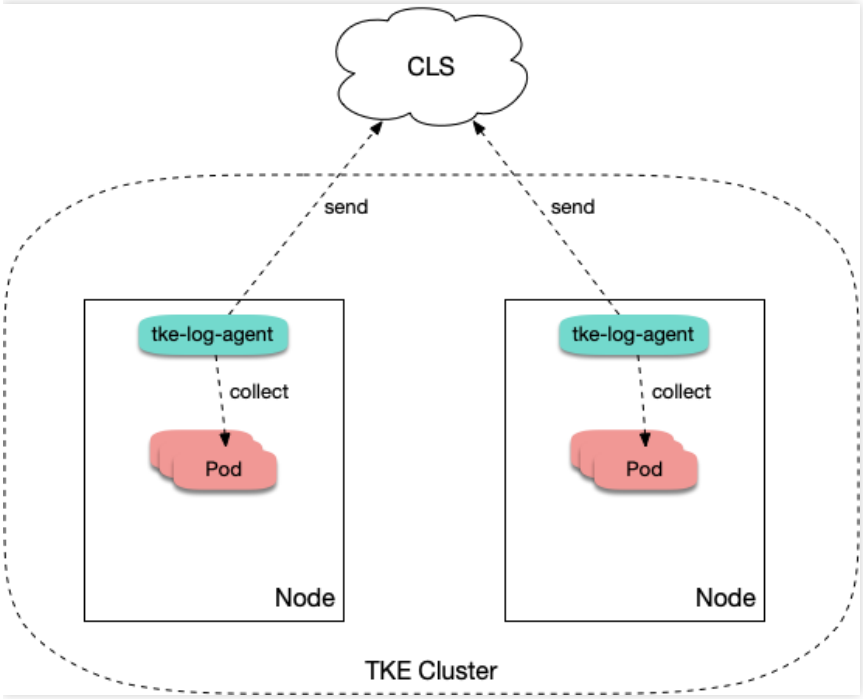
本文介绍容器服务 TKE 的日志功能中的日志采集、存储、查询各种功能的用法，并结合实际应用场景提供建议。您可结合实际情况，参考本文进行日志采集实践。

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。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

当使用 TKE 日志采集功能时，需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”3种采集类型。请参考下文了解各类型使用场景及建议：

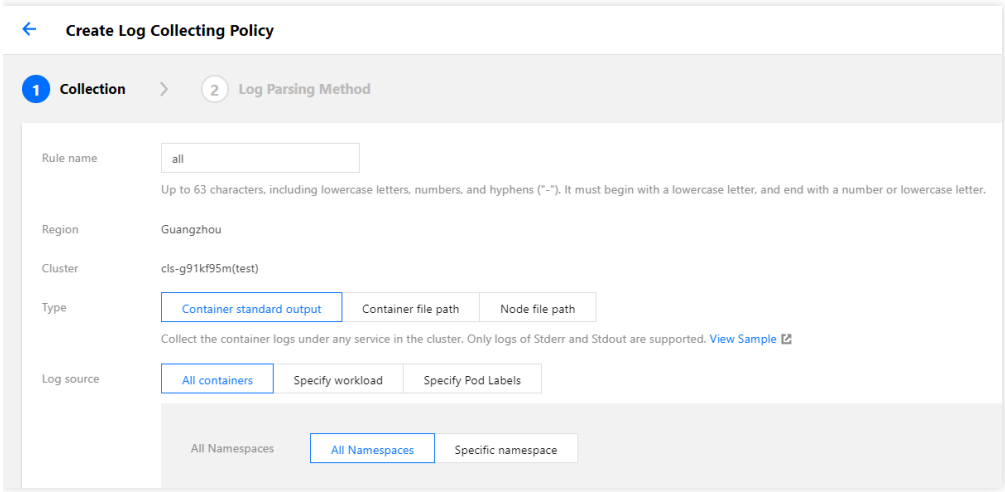
采集标准输出

“采集标准输出”是将 Pod 内容器日志输出到标准输出，日志内容会由容器运行时（docker 或 containerd）来管理。“采集标准输出”较于其他方式最简单，推荐选择。具备以下优势：

- 不需要额外挂载 volume。

2. 可直接通过 `kubectl logs` 查看日志内容。
3. 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。
4. 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。



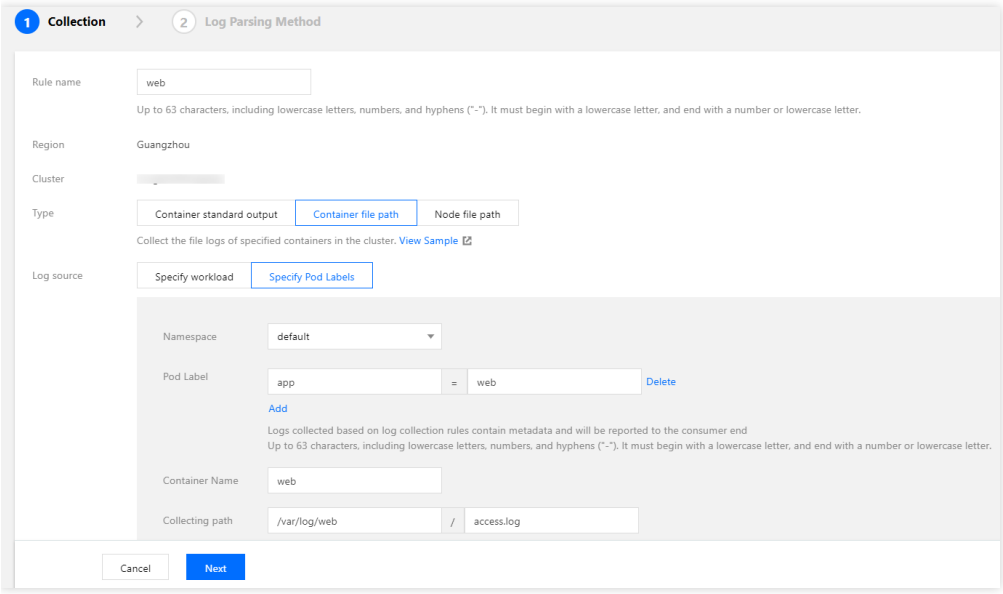
采集容器内的文件

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

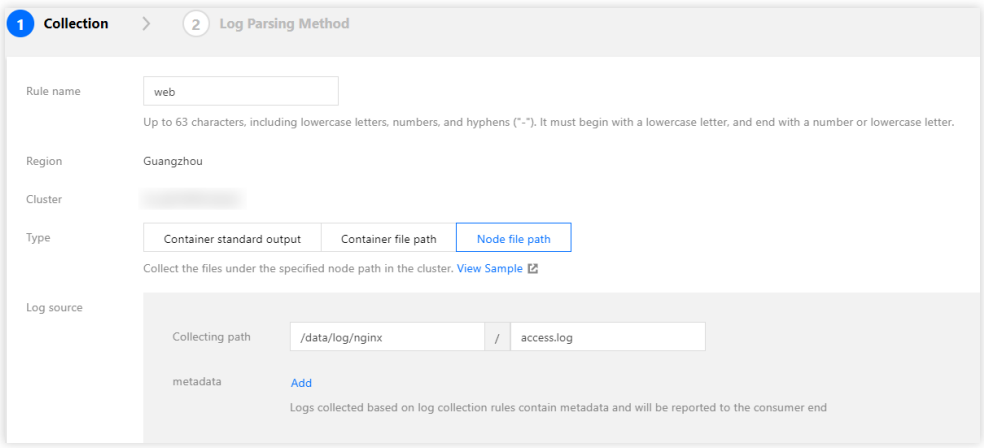
若业务需将日志写入日志文件，但期望在容器停止后仍保留原始日志文件作为备份，避免采集异常时日志完全丢失。此时可以给日志文件路径挂载 hostPath，日志文件会落盘到宿主机指定目录，并且容器停止后不会清理日志文件。
由于不会自动清理日志文件，可能会发生 Pod 调度走再调度回来，日志文件被写在相同路径，从而重复采集的问题。采集分以下两种情况：

- 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：

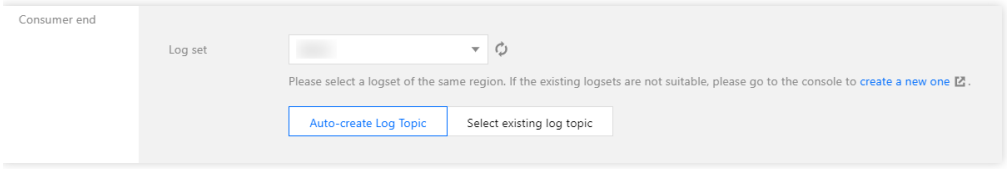
> 通常情况下不会发生重复采集，若日志框架会对日志进行自动轮转，建议采集规则不要使用通配符 `*` 来匹配日志文件。

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：



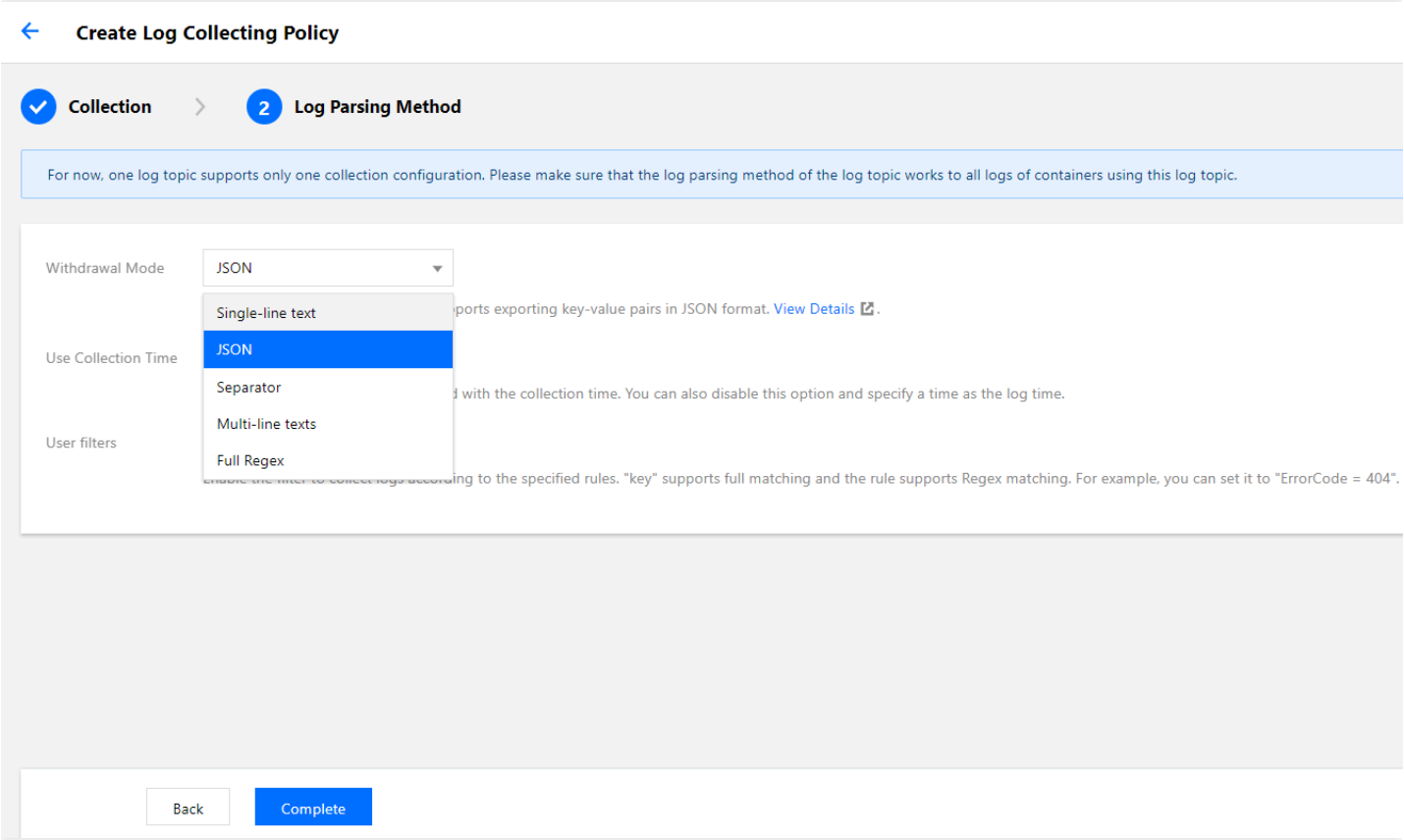
自动创建日志主题后，可前往 日志集管理 的对应日志集详情页面，进行重命名操作，以便后续检索时快速找到日志所在的日志主题。

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、JSON、分隔符、多行文本和完全正则5种提取模式。如下图所示：



- JSON 模式
- 单行文本及多行文本模式
- 分隔符及完全正则模式

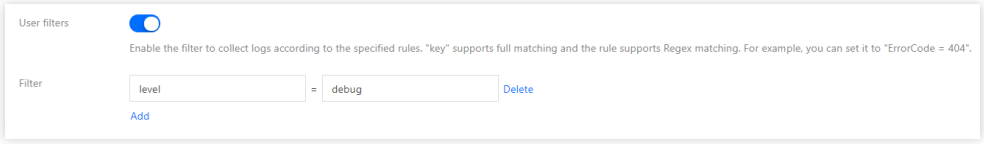
选择 “JSON 模式”需日志本身是以 JSON 格式输出的，推荐选择该模式。JSON 格式本身已将日志结构化，CLS 可以提取 JSON 的 key 作为字段名，value 作为对应的字段值，不再需要根据业务日志输出格式配置复杂的匹配规则。日志示例如下：

```
{"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"}
```

配置过滤内容

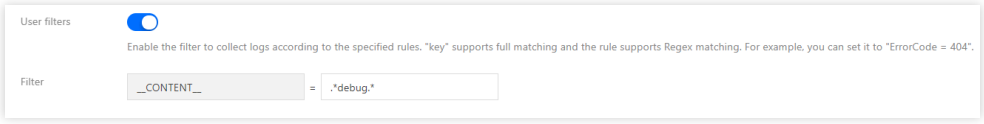
可选择过滤无需使用的日志信息，降低成本。

- 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` 。

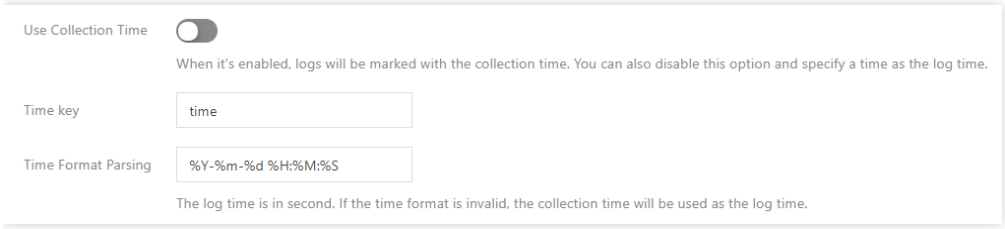


自定义日志时间戳

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，需关闭“使用采集时间”、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。



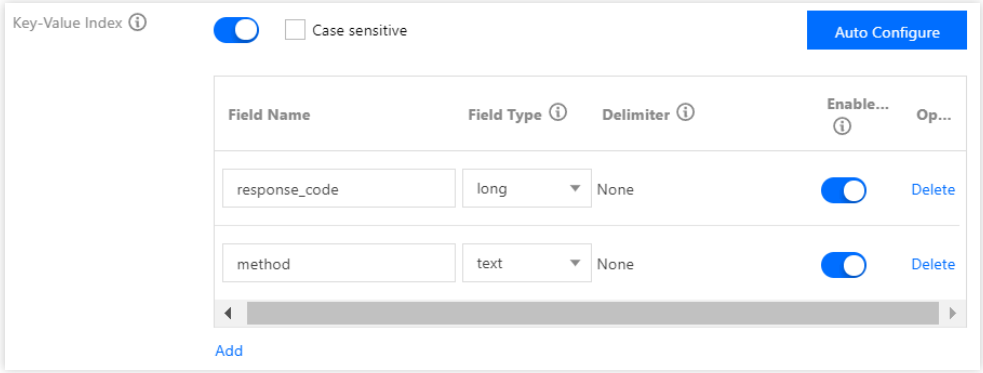
更多时间格式配置信息请参见 配置时间格式。

查询日志

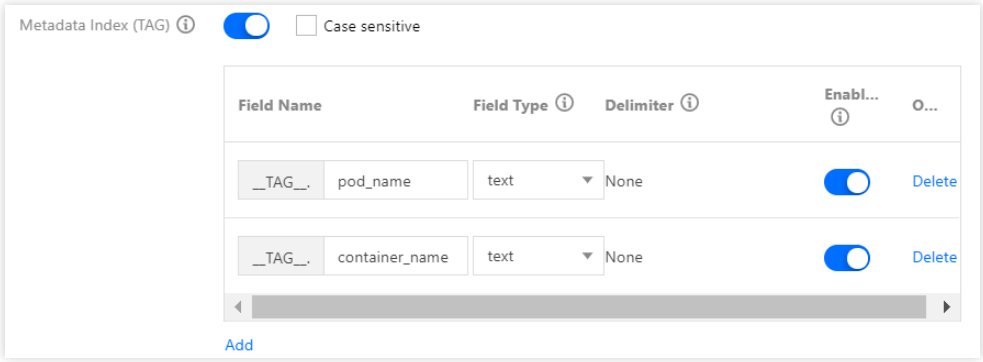
完成日志采集规则配置后，采集器会自动开始采集日志并上报到 CLS。您可在 日志服务控制台 的【检索分析】中查询日志，开启索引后支持 Lucene 语法。有以下3类索引：

- 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：



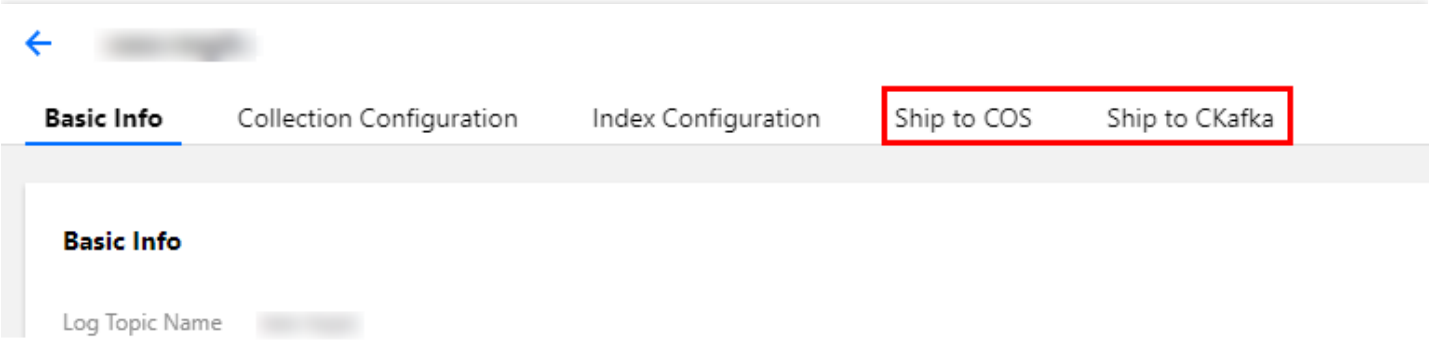
- 元字段索引。上报日志时额外自动附加的一些字段。例如 pod 名称、namespace 等，方便检索时指定这些字段进行检索。如下图所示：



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

- TKE ：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

TKE Serverless 日志采集实现多行日志合并

最近更新时间：2023-05-09 16:52:26

使用场景

使用环境变量去配置 TKE Serverless 日志采集时，默认使用单行提取模式。但当客户程序的日志数据跨占多行时（例如 Java 程序日志），不能以换行符 `\n` 作为日志的结束标识符。为了能让日志系统明确区分开每条日志，需要配置具有首行正则表达式的 configmap，当某行日志匹配了预先设置的正则表达式，则认为是一条日志的开头，而在下一个行首出现则作为该条日志的结束标识符。本文向您介绍使用环境变量的方式开启 TKE Serverless 日志采集时，如何实现多行日志合并。

操作步骤

原始日志示例

```
2020-09-24 16:09:07 ERROR System.out(4844) java.lang.NullPointerException
at com.temp.ttscancel.MainActivity.onCreate(MainActivity.java:43)
at android.app.Activity.performCreate(Activity.java:5248)
at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1110) at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2162) at and
roid.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2257)
at android.app.ActivityThread.access$800(ActivityThread.java:139)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1210)
```

创建 Configmap

针对原始日志文件示例，parser.conf 配置文件内容为：

```
apiVersion: v1
data:
parser.conf: |-
[PARSER]
Name parser_name
Format regex
Regex ^(?<timestamp>[0-9]{2,4}\-[0-9]{1,2}\-[0-9]{1,2} [0-9]{1,2}\:[0-9]{1,2}\:[0
-9]{1,2}) (?<message>.*)
kind: ConfigMap
metadata:
name: cm
namespace: default
```

其中 `parser_name` 需要在创建工作负载的时候设置在 annotation 中（spec.template.metadata.annotations），执行以下命令进行设置：

```
eks.tke.cloud.tencent.com/parser-name: parser_name
```

更多 `parser.conf` 配置文件相关内容可参见 Regular Expression。

创建 Pod 时挂载 Configmap

在创建 Pod 时需要做以下操作：

1. 以 Volume 形式挂载已创建的 Configmap。
2. 通过环境变量的方式开启日志采集，详情可参见 配置日志采集。
3. 指定两个 annotation。

```
eks.tke.cloud.tencent.com/parser-name: "parser_name"
eks.tke.cloud.tencent.com/volume-name-for-parser: "volume-name"
```

- `eks.tke.cloud.tencent.com/parser-name` 是指已创建的 Configmap 的 name。
- `eks.tke.cloud.tencent.com/volume-name-for-parser` 是指 Pod 中挂载的 Volume 名称，可自定义。

Pod yaml 模板

```
apiVersion: apps/v1
kind: Deployment
metadata:
labels:
k8s-app: multiline
qcloud-app: multiline
name: multiline
namespace: default
spec:
replicas: 1
selector:
matchLabels:
k8s-app: multiline
qcloud-app: multiline
template:
metadata:
annotations:
eks.tke.cloud.tencent.com/parser-name: parser_name
eks.tke.cloud.tencent.com/volume-name-for-parser: volume-name
```

```
labels:
k8s-app: multiline
qcloud-app: multiline
spec:
containers:
- env:
- name: EKS_LOGS_OUTPUT_TYPE
value: cls
- name: EKS_LOGS_LOG_PATHS
value: stdout
- name: EKS_LOGS_TOPIC_ID
value: topic-id
- name: EKS_LOGS_LOGSET_NAME
value: eks
- name: EKS_LOGS_SECRET_ID
valueFrom:
secretKeyRef:
key: SecretId
name: cls
optional: false
- name: EKS_LOGS_SECRET_KEY
valueFrom:
secretKeyRef:
key: SecretKey
name: cls
optional: false
image: nginx
imagePullPolicy: Always
name: ng
resources:
limits:
cpu: 500m
memory: 1Gi
requests:
cpu: 250m
memory: 256Mi
volumeMounts:
- mountPath: /mnt
name: volume-name
imagePullSecrets:
- name: qcloudregistrykey
restartPolicy: Always
volumes:
- configMap:
defaultMode: 420
name: cm
name: volume-name
```

结构化处理后日志示例

```
2020-09-24 16:09:07 ERROR System.out(4844) java.lang.NullPointerException \at com
.temp.ttscancel.MainActivity.onCreate(MainActivity.java:43) \at android.app.Activ
ity.performCreate(Activity.java:5248) \at android.app.Instrumentation.callActivit
yOnCreate(Instrumentation.java:1110) \at android.app.ActivityThread.performLaunch
Activity(ActivityThread.java:2162) \at android.app.ActivityThread.handleLaunchAct
ivity(ActivityThread.java:2257) \at android.app.ActivityThread.access$800(Activit
yThread.java:139) \at android.app.ActivityThread$H.handleMessage(ActivityThread.j
ava:1210)
```

NginxIngress 自定义日志

最近更新时间：2023-05-06 17:36:46

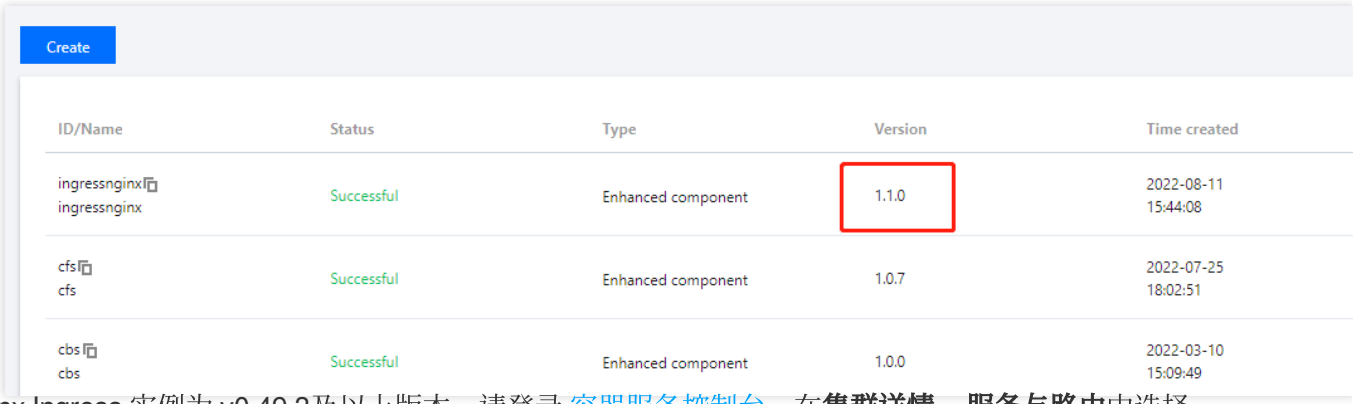
容器服务 TKE 通过集成日志服务 CLS，提供了全套完整的产品化能力，实现 Nginx-ingress 日志采集、消费能力。更多请查看 Nginx-ingress 日志配置。若默认的日志索引不符合您的日志需求，您可以自定义日志索引，本文向您介绍如何更新 Nginx Ingress 的日志索引。

前提条件

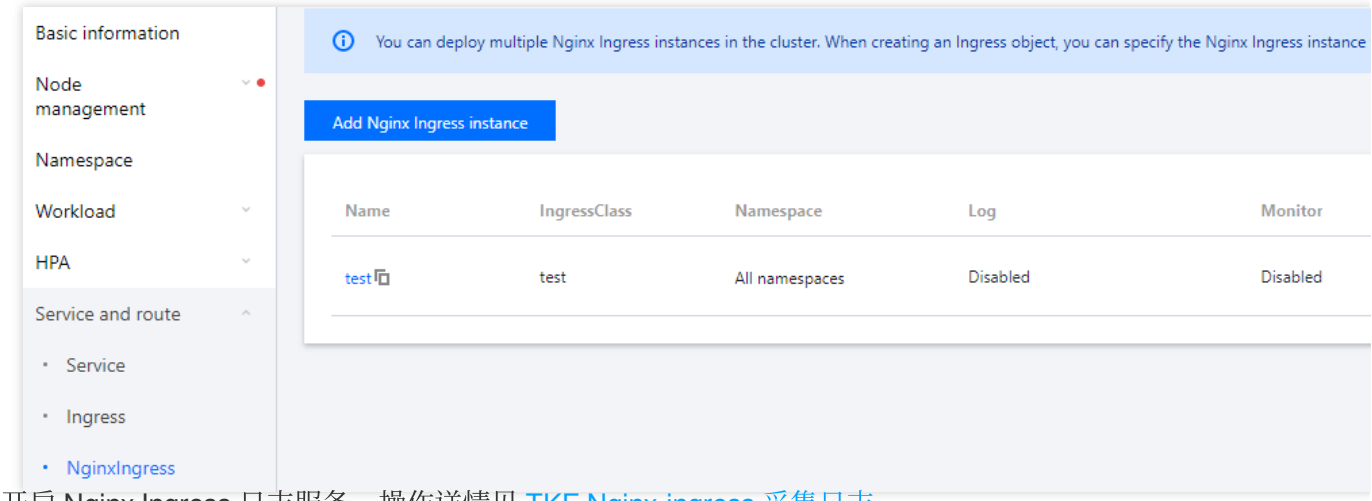
1. Nginx Ingress 为 v1.1.0及以上版本。请登录 容器服务控制台，在**集群详情 > 组件管理**中查看 Nginx Ingress 的组件版本。

注意：

仅Nginx Ingress 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. Nginx Ingress 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 Nginx Ingress 日志服务。操作详情见 TKE Nginx-ingress 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 Nginx Ingress 实例的日志输出格式

Nginx Ingress 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：

```
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] [$upstream_response_
    [$upstream_response_time] [$upstream_status] $req_id $service_name $namespace
  max-worker-connections: "65536"
  upstream-keepalive-connections: "200"
kind: ConfigMap
metadata:
  creationTimestamp: "2022-07-22T02:56:35Z"
  labels:
    k8s-app:        s-ingress-nginx-controller
    qcloud-app:          -ingress-nginx-controller
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:access-log-path: {}
        f:allow-snippet-annotations: {}
        f:error-log-path: {}
        f:keep-alive-requests: {}
        f:max-worker-connections: {}
```

示例

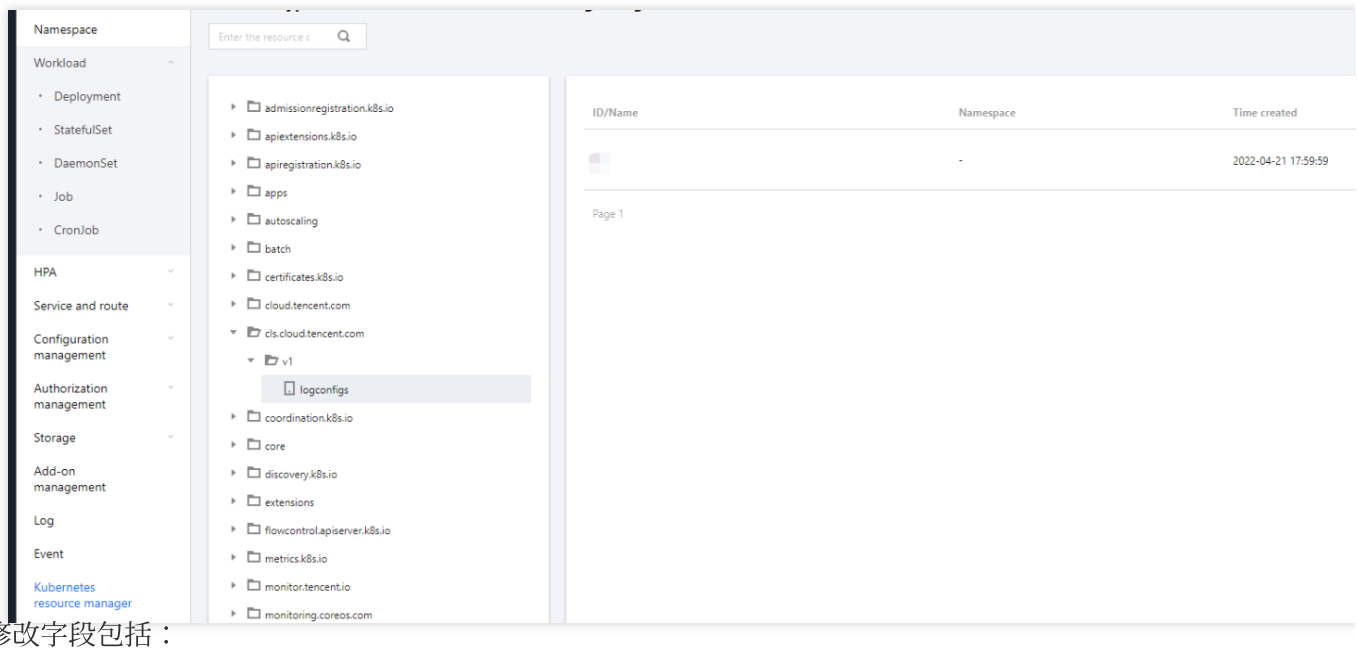
在日志中增加两个连续的字符串：`$namespace` 和 `$service_name`，并放在日志内容的最后，添加位置如下图所示：

```
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] [$upstream_response_le
    [$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 Nginx Ingress 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。

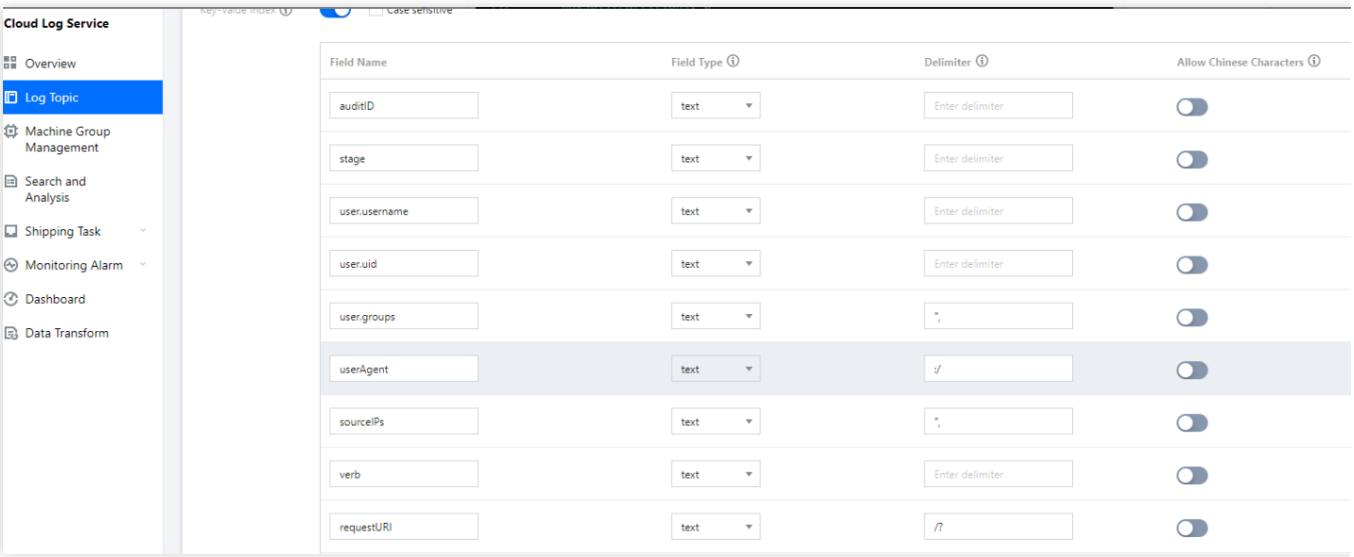
示例

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：

```
      - 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
      - namespace
      - service_name
      logRegex: (\S+)\s-\s(\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+)
    logType: fullregex_log
    maxSplitPartitions: 0
    storageType: ""
    topicId: 3aa9fa69-1595-4fef-ad2d-cf9a0df0beed
  inputDetail:
```

（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。

监控

使用 Prometheus 监控 Java 应用

最近更新时间：2020-09-27 16:31:01

操作场景

Prometheus 社区开发了 JMX Exporter 用于导出 JVM 的监控指标，以便使用 Prometheus 来采集监控数据。当您的 Java 业务容器化至 Kubernetes 后，可通过本文了解如何使用 Prometheus 与 JMX Exporter 来监控 Java 应用。

JMX Exporter 简介

Java Management Extensions，JMX 是管理 Java 的一种扩展框架，JMX Exporter 基于此框架读取 JVM 的运行时状态。JMX Exporter 利用 Java 的 JMX 机制来读取 JVM 运行时的监控数据，然后将其转换为 Prometheus 可辨识的 metrics 格式，以便让 Prometheus 对其进行监控采集。

JMX Exporter 提供**启动独立进程**及 **JVM 进程内启动（in-process）**两种方式暴露 JVM 监控指标：

1. 启动独立进程
JVM 启动时指定参数，暴露 JMX 的 RMI 接口。JMX Exporter 调用 RMI 获取 JVM 运行时状态数据，转换为 Prometheus metrics 格式，并暴露端口让 Prometheus 采集。
2. JVM 进程内启动（in-process）
JVM 启动时指定参数，通过 javaagent 的形式运行 JMX Exporter 的 jar 包，进程内读取 JVM 运行时状态数据，转换为 Prometheus metrics 格式，并暴露端口让 Prometheus 采集。

> **说明：**
>
> 官方不建议使用**启动独立进程**方式，该方式配置复杂且需单独的进程，进程本身的监控又引发了新的问题。本文以 **JVM 进程内启动（in-process）**方式为例，在 Kubernetes 环境下使用 JMX Exporter 暴露 JVM 监控指标。

操作步骤

使用 JMX Exporter 暴露 JVM 监控指标

打包镜像

使用 JVM 进程内启动（in-process）方式，启动 JVM 需指定 JMX Exporter 的 jar 包文件和配置文件。jar 包为二进制文件，不便通过 configmap 挂载，建议直接将 JMX Exporter 的 jar 包和配置文件都打包到业务容器镜像中。步骤如下：

1. 准备一个制作镜像的目录，放入 JMX Exporter 配置文件 `prometheus-jmx-config.yaml` 。

```
ssl: false
lowercaseOutputName: false
lowercaseOutputLabelNames: false
```

> ⚠ **注意：**
>
> 更多配置项请参考 Prometheus 官方文档。

2. 准备 jar 包文件，可前往 jmx_exporter 的 Github 页面获取最新的 jar 包下载地址。执行以下命令，下载到当前目录。

```
wget https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/0.13.0/jmx_prometheus_javaagent-0.13.0.jar
```

3. 准备 Dockerfile 文件，本文以 Tomcat 为例。

```
FROM tomcat:jdk8-openjdk-slim
ADD prometheus-jmx-config.yaml /prometheus-jmx-config.yaml
ADD jmx_prometheus_javaagent-0.13.0.jar /jmx_prometheus_javaagent-0.13.0.jar
```

4. 执行以下命令，编译镜像。

```
docker build . -t ccr.ccs.tencentyun.com/imroc/tomcat:jdk8
```

至此已完成镜像打包，您还可利用 docker 多阶段构建，省略手动下载 jar 包的步骤。Dockerfile 示例如下：

```
FROM ubuntu:16.04 as jar
WORKDIR /
RUN apt-get update -y
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y wget
RUN wget https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/0.13.0/jmx_prometheus_javaagent-0.13.0.jar
```

```
FROM tomcat:jdk8-openjdk-slim
ADD prometheus-jmx-config.yaml /prometheus-jmx-config.yaml
COPY --from=jar /jmx_prometheus_javaagent-0.13.0.jar /jmx_prometheus_javaagent-0.13.0.jar
```

部署 Java 应用

部署应用至 Kubernetes 时，需修改 JVM 启动参数以便启动时加载 JMX Exporter。JVM 启动时会读取 `JAVA_OPTS` 环境变量，作为额外的启动参数，部署时可为应用增加该环境变量。示例如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: tomcat
spec:
replicas: 1
selector:
matchLabels:
app: tomcat
template:
metadata:
labels:
app: tomcat
spec:
containers:
- name: tomcat
image: ccr.ccs.tencentyun.com/imroc/tomcat:jdk8
env:
- name: JAVA_OPTS
value: "-javaagent:/jmx_prometheus_javaagent-0.13.0.jar=8088:/prometheus-jmx-config.yaml"

---

apiVersion: v1
kind: Service
metadata:
name: tomcat
labels:
app: tomcat
spec:
type: ClusterIP
ports:
- port: 8080
protocol: TCP
name: http
```

```
- port: 8088
protocol: TCP
name: jmx-metrics
selector:
app: tomcat
```

- 启动参数格式：`-javaagent:<jar>=<port>:<config>`
- 该示例使用8088端口暴露 JVM 的监控指标，您可按需自行更改。

添加 Prometheus 监控配置

配置 Prometheus，使监控数据可被采集。示例如下：

```
- job_name: tomcat
scrape_interval: 5s
kubernetes_sd_configs:
- role: endpoints
namespaces:
names:
- default
relabel_configs:
- action: keep
source_labels:
- __meta_kubernetes_service_label_app
regex: tomcat
- action: keep
source_labels:
- __meta_kubernetes_endpoint_port_name
regex: jmx-metrics
```

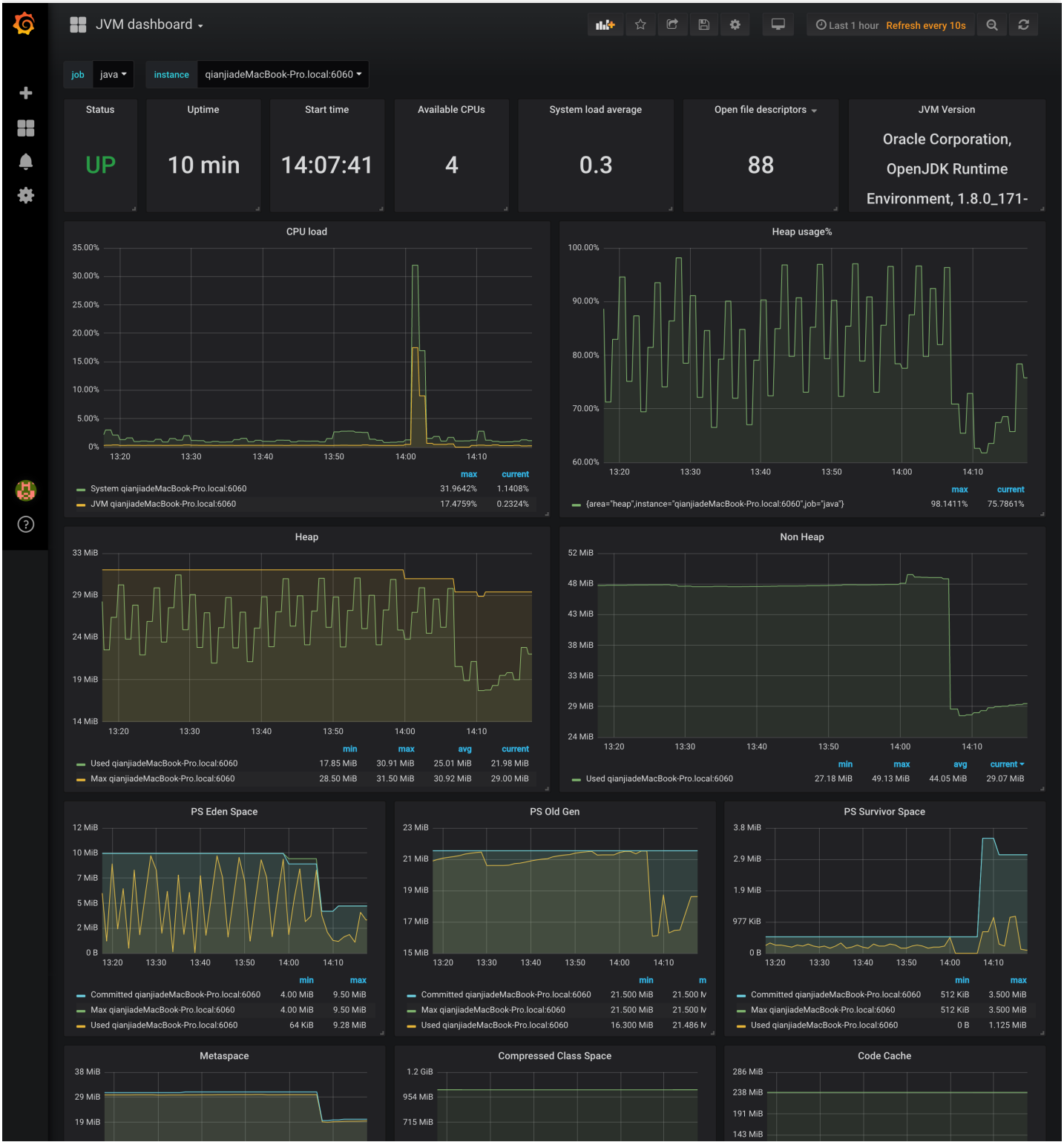
若已安装 prometheus-operator，则可通过创建 ServiceMonitor 的 CRD 对象配置 Prometheus。示例如下：

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
name: tomcat
namespace: default
labels:
app: tomcat
spec:
endpoints:
- port: jmx-metrics
interval: 5s
namespaceSelector:
matchNames:
- default
```

```
selector:
matchLabels:
app: tomcat
```

添加 Grafana 监控面板

采集数据后可进行数据展示。若熟悉 Prometheus 和 Grafana，则可自行根据指标设计所需面板。您也可直接使用社区提供面板，例如 JVM dashboard。可直接导入使用，面板效果图如下：



参考资料

- JMX Exporter 项目地址
- JVM 监控面板

使用 Prometheus 监控 MySQL 与 MariaDB

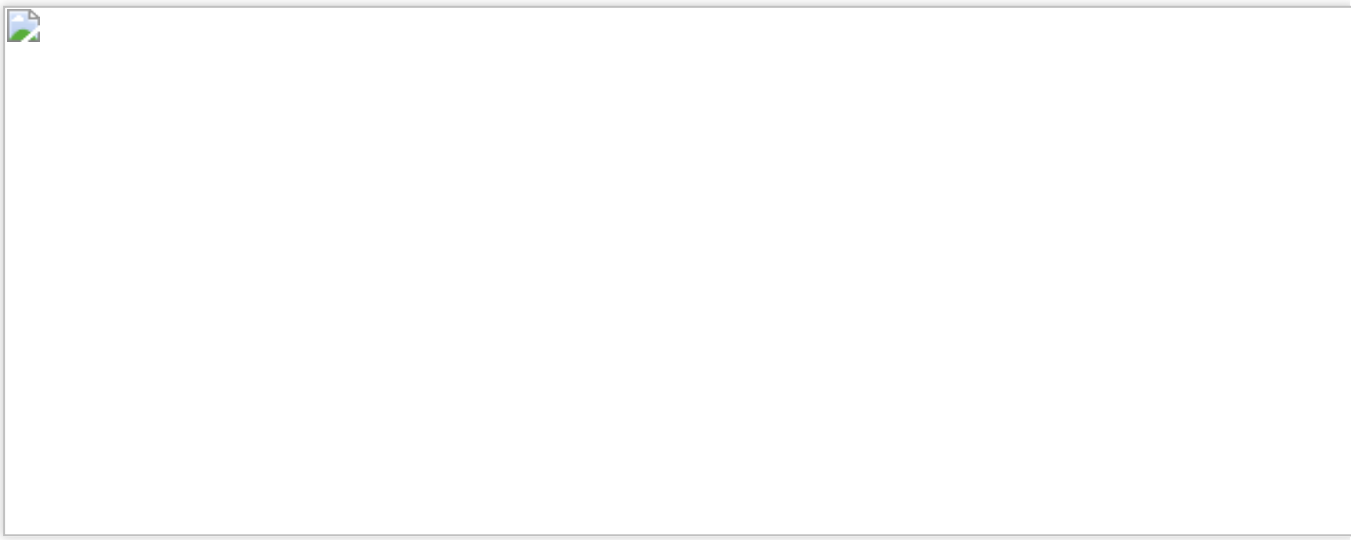
最近更新时间：2023-03-14 18:19:11

操作场景

MySQL 是常用的关系型数据库，MariaDB 作为 MySQL 的分支版本，兼容 MySQL 协议，也越来越流行。在 Kubernetes 环境中，可借助开源的 mysqld-exporter 来使用 Prometheus 监控 MySQL 与 MariaDB。您可通过本文了解 Prometheus 并开始使用。

mysqld-exporter 简介

mysqld-exporter 通过读取 MySQL 或 MariaDB 中某些数据库状态的数据，将其转换为 Prometheus 的指标格式并暴露为 HTTP 接口被 Prometheus 采集，让原本不支持 Prometheus 指标的 MySQL 和 MariaDB 能够被 Prometheus 监控起来。如下图所示：



操作步骤

部署 mysqld-exporter

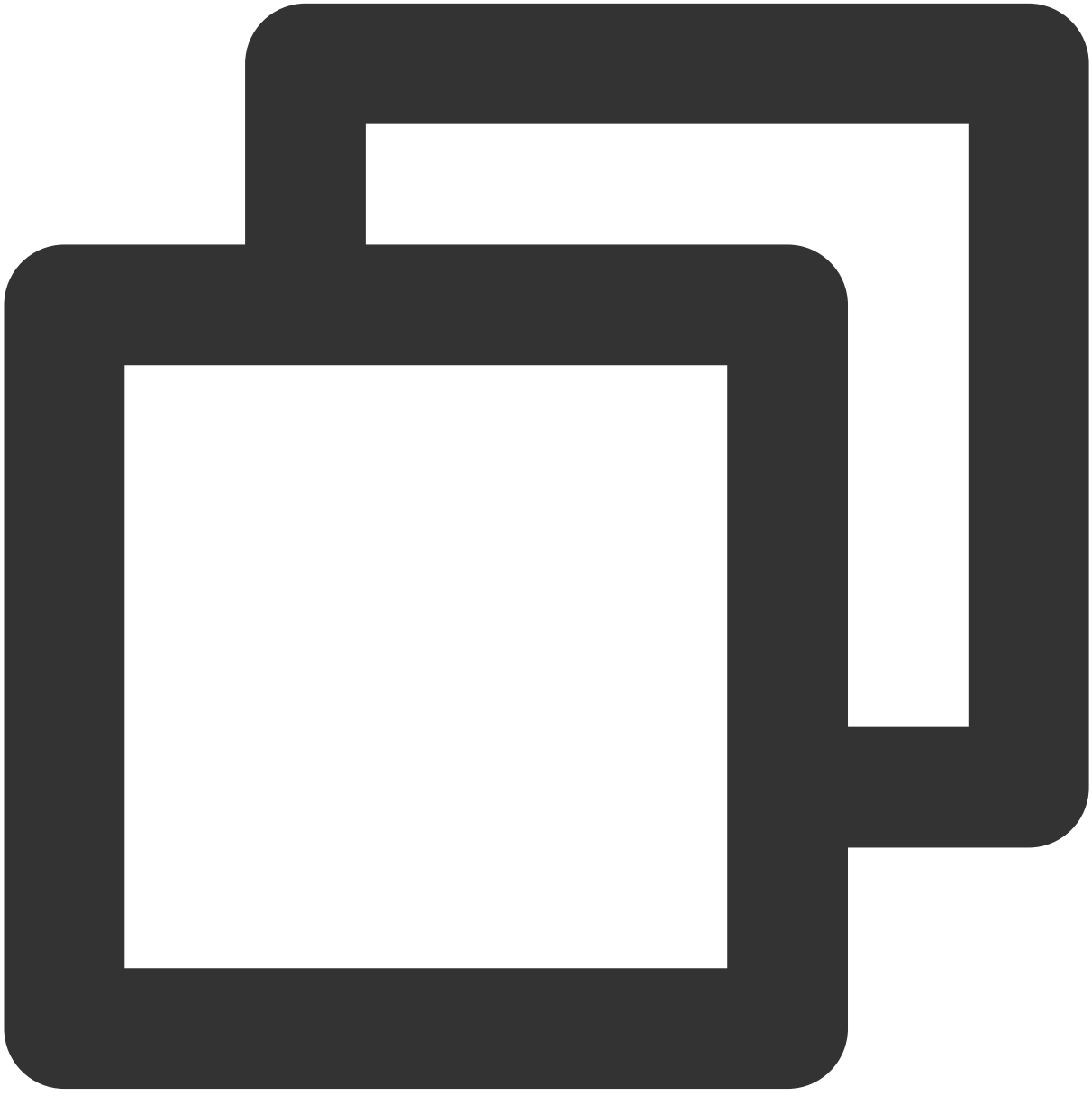
注意

在部署 mysqld-exporter 之前需确保已在集群内、集群外或使用已有的云服务中部署 MySQL 或 MariaDB。

部署 MySQL

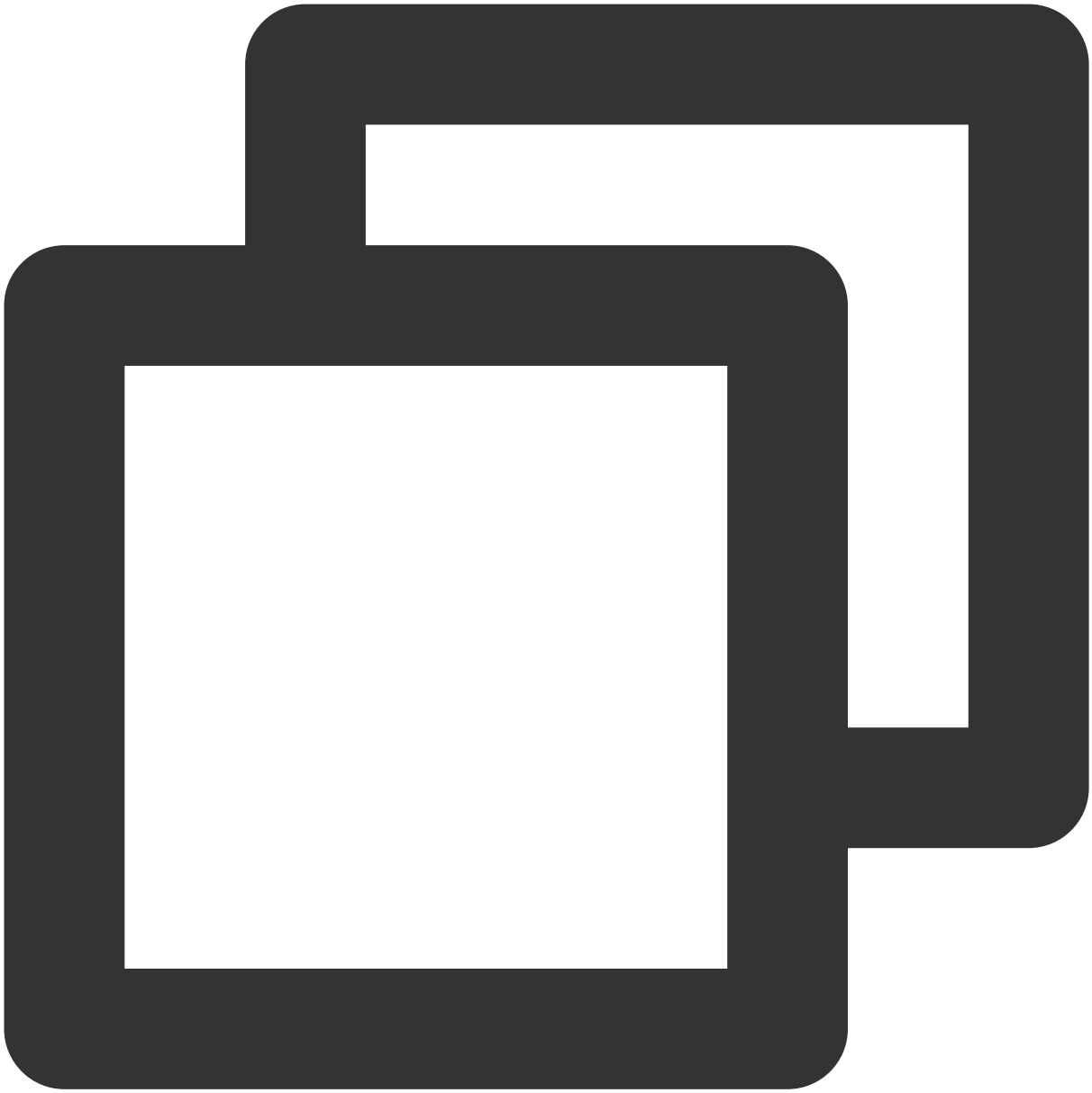
以从应用市场部署 MySQL 到集群为例。步骤如下：

1. 登录 容器服务控制台，在左侧导航栏选择**应用市场**。
2. 在**应用市场**页面，搜索并选择 **MySQL**。
3. 在**应用详情**页面，单击**创建应用**。
4. 在**创建应用**页面，填写应用信息后单击**创建**。
5. 应用创建完成后，在左侧导航栏选择**应用**，在**应用**页面查看应用详情。
6. 执行以下命令，查看 MySQL 是否正常运行。



```
$ kubectl get pods
NAME                     READY   STATUS         RESTARTS   AGE
mysql-698b898bf7-4dc5k   1/1     Running        0          11s
```

7. 执行以下命令，获取 root 密码。

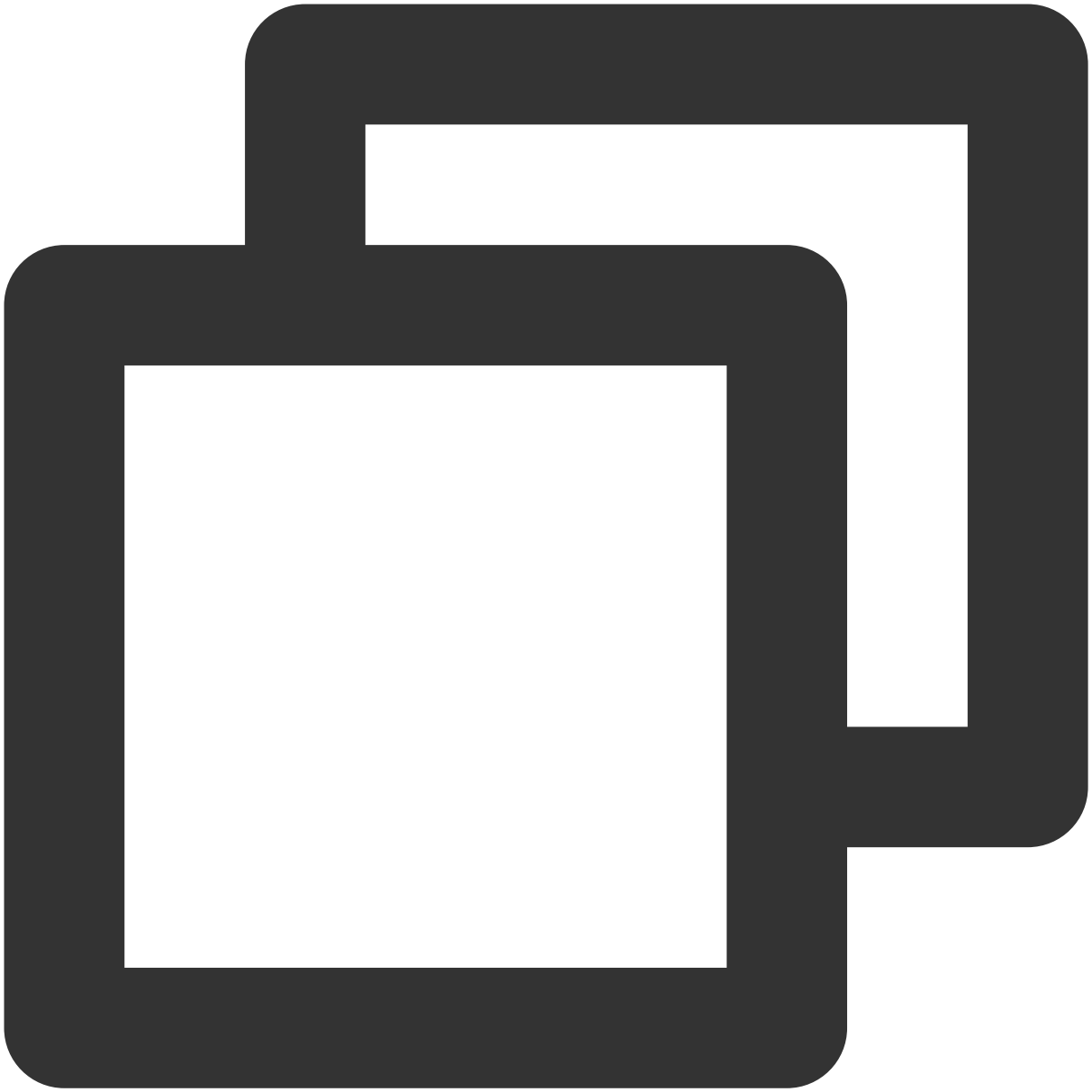


```
$ kubectl get secret -o jsonpath={.data.mysql-root-password} mysql | base64 -d
6ZAj33yLBo
```

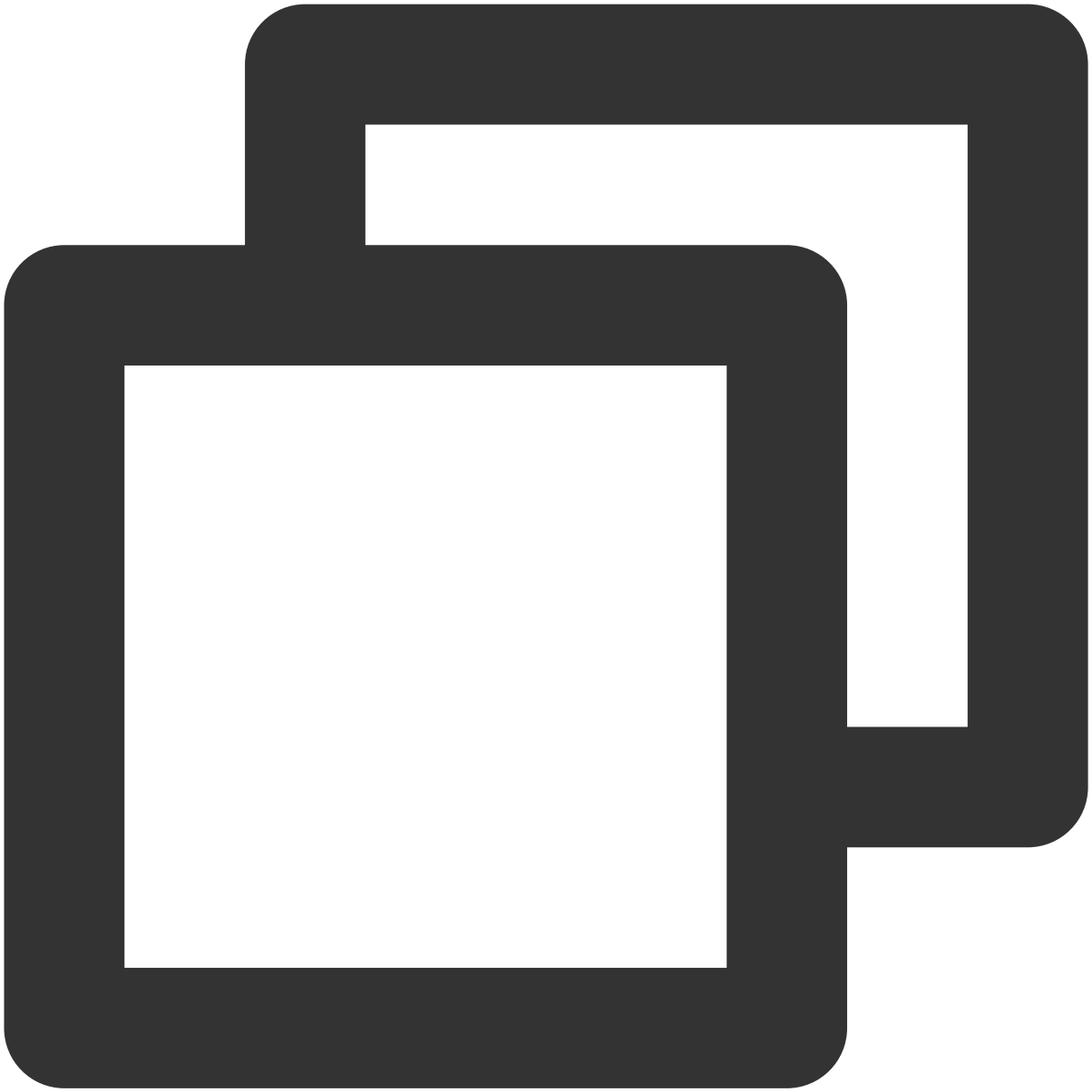
部署 mysqld-exporter

部署 MySQL 后，可以开始部署 mysqld-exporter。步骤如下：

1. 依次执行以下命令，创建 mysqld-exporter 账号并登录 MySQL。示例如下：

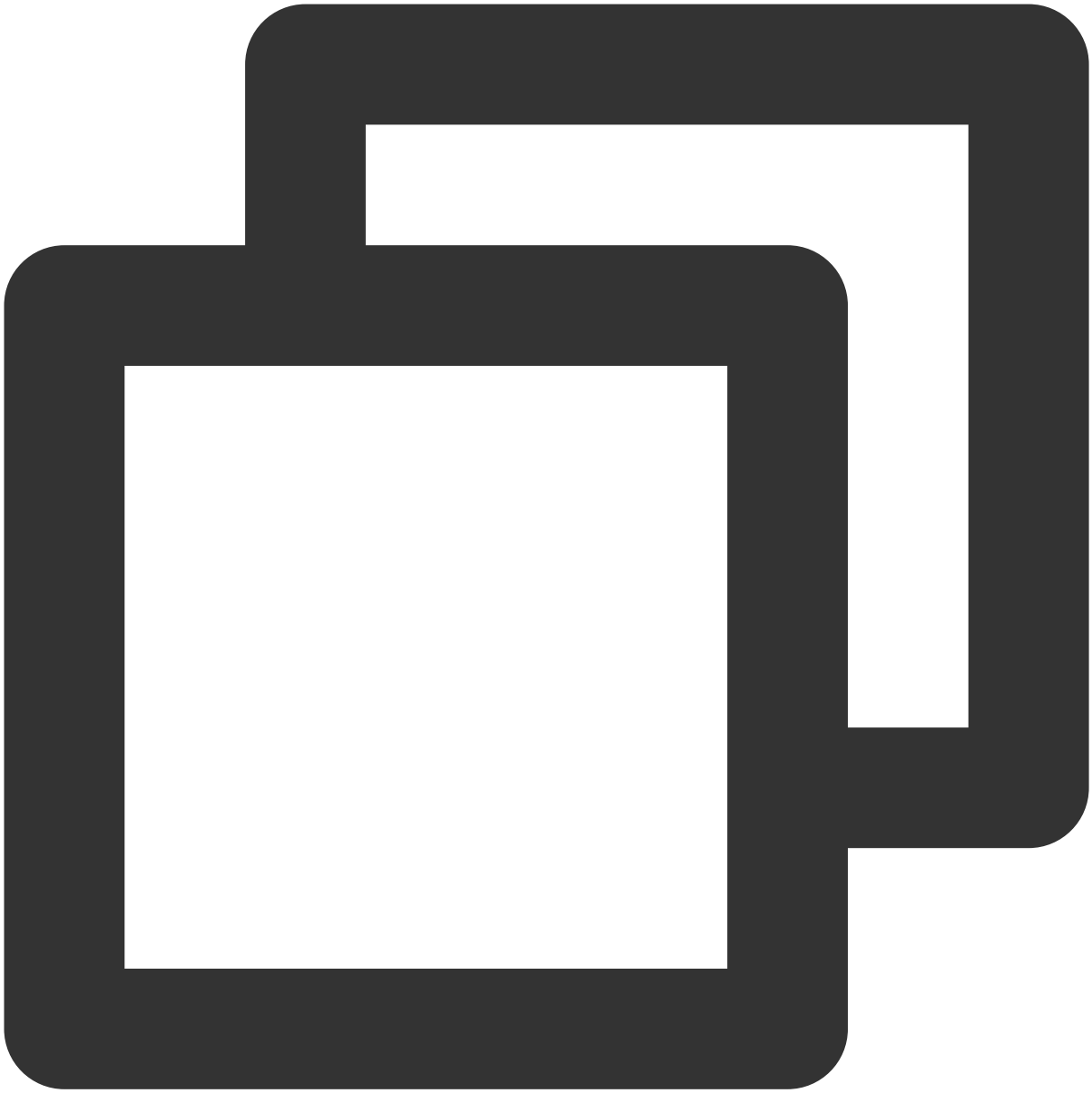


```
$ kubectl exec -it mysql-698b898bf7-4dc5k bash
```



```
$ mysql -uroot -p6ZAj33yLBo
```

2. 执行以下命令，输入 SQL 语句创建账号。以 `mysqld-exporter/123456` 为例，示例如下：

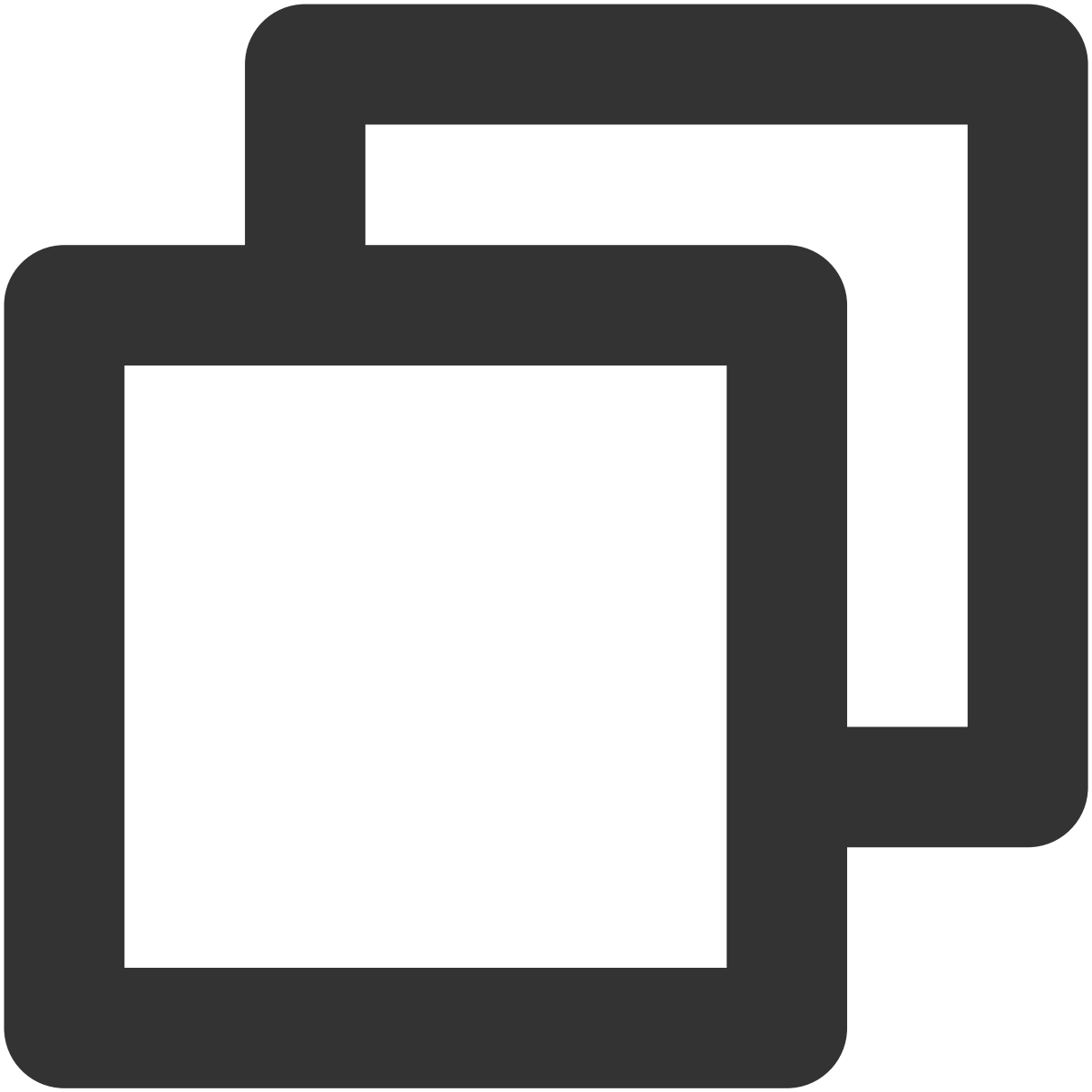


```
CREATE USER 'mysqld-exporter' IDENTIFIED BY '123456' WITH MAX_USER_CONNECTIONS 3;
GRANT PROCESS, REPLICATION CLIENT, REPLICATION SLAVE, SELECT ON *.* TO 'mysqld-expo
flush privileges;
```

3. 使用 yaml 文件部署 mysqld-exporter。示例如下：

注意

需根据实际情况替换 DATA_SOURCE_NAME 中的账号密码，以及 MySQL 的连接地址。



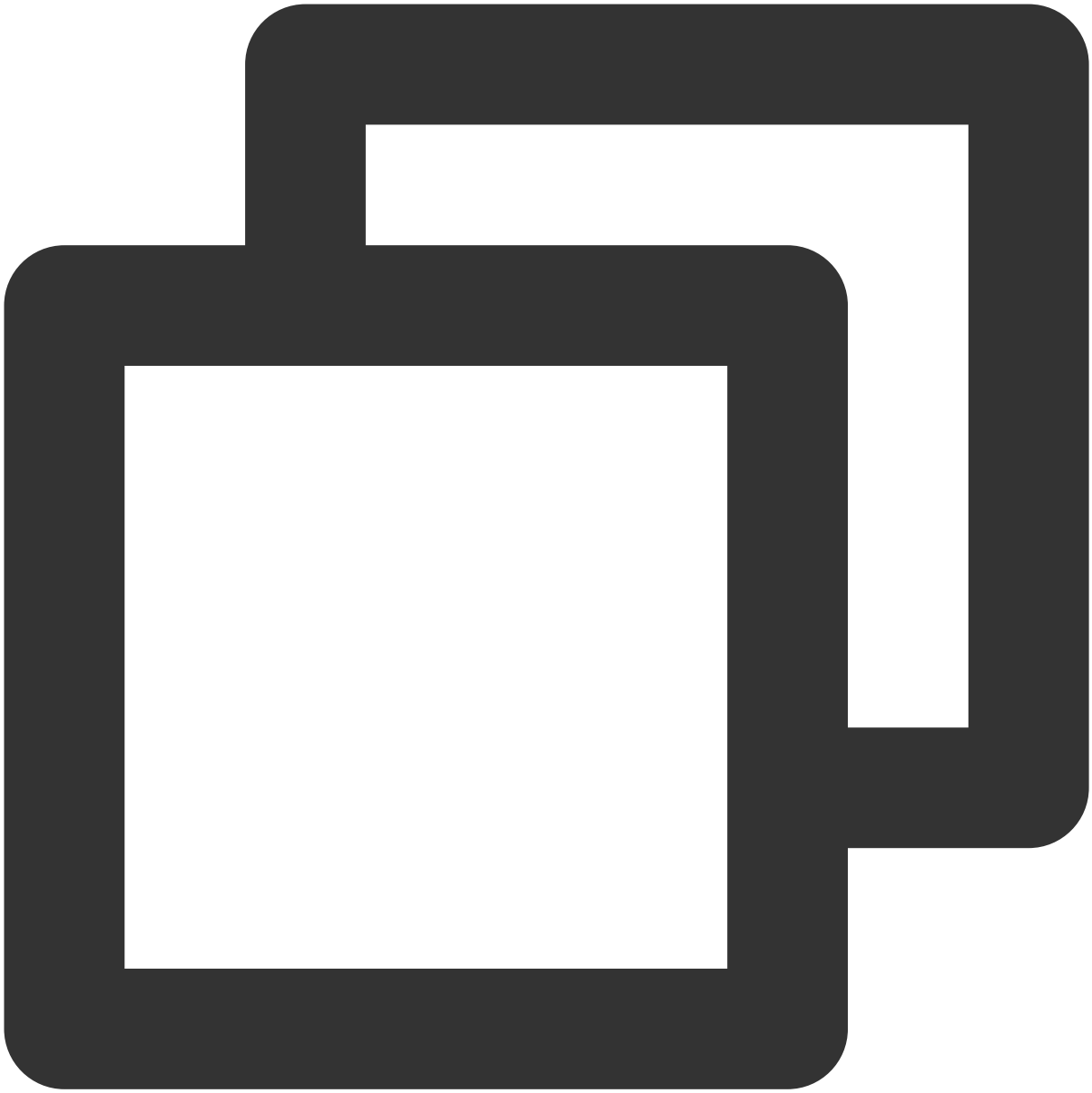
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysqld-exporter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysqld-exporter
  template:
    metadata:
```

```
      labels:
        app: mysqld-exporter
    spec:
      containers:
      - name: mysqld-exporter
        image: prom/mysqld-exporter:v0.12.1
        args:
        - --collect.info_schema.tables
        - --collect.info_schema.innodb_tablespaces
        - --collect.info_schema.innodb_metrics
        - --collect.global_status
        - --collect.global_variables
        - --collect.slave_status
        - --collect.info_schema.processlist
        - --collect.perf_schema.tablelocks
        - --collect.perf_schema.eventsstatements
        - --collect.perf_schema.eventsstatementssum
        - --collect.perf_schema.eventswaits
        - --collect.auto_increment.columns
        - --collect.binlog_size
        - --collect.perf_schema.tableiowaits
        - --collect.perf_schema.indexiowaits
        - --collect.info_schema.userstats
        - --collect.info_schema.clientstats
        - --collect.info_schema.tablestats
        - --collect.info_schema.schemastats
        - --collect.perf_schema.file_events
        - --collect.perf_schema.file_instances
        - --collect.perf_schema.replication_group_member_stats
        - --collect.perf_schema.replication_applier_status_by_worker
        - --collect.slave_hosts
        - --collect.info_schema.innodb_cmp
        - --collect.info_schema.innodb_cmpmem
        - --collect.info_schema.query_response_time
        - --collect.engine_tokudb_status
        - --collect.engine_innodb_status
        ports:
        - containerPort: 9104
          protocol: TCP
        env:
        - name: DATA_SOURCE_NAME
          value: "mysqld-exporter:123456@(mysql.default.svc.cluster.local:3306)/"
--
apiVersion: v1
kind: Service
metadata:
  name: mysqld-exporter
```

```
    labels:
      app: mysqld-exporter
  spec:
    type: ClusterIP
    ports:
    - port: 9104
      protocol: TCP
      name: http
    selector:
      app: mysqld-exporter
```

添加监控采集配置

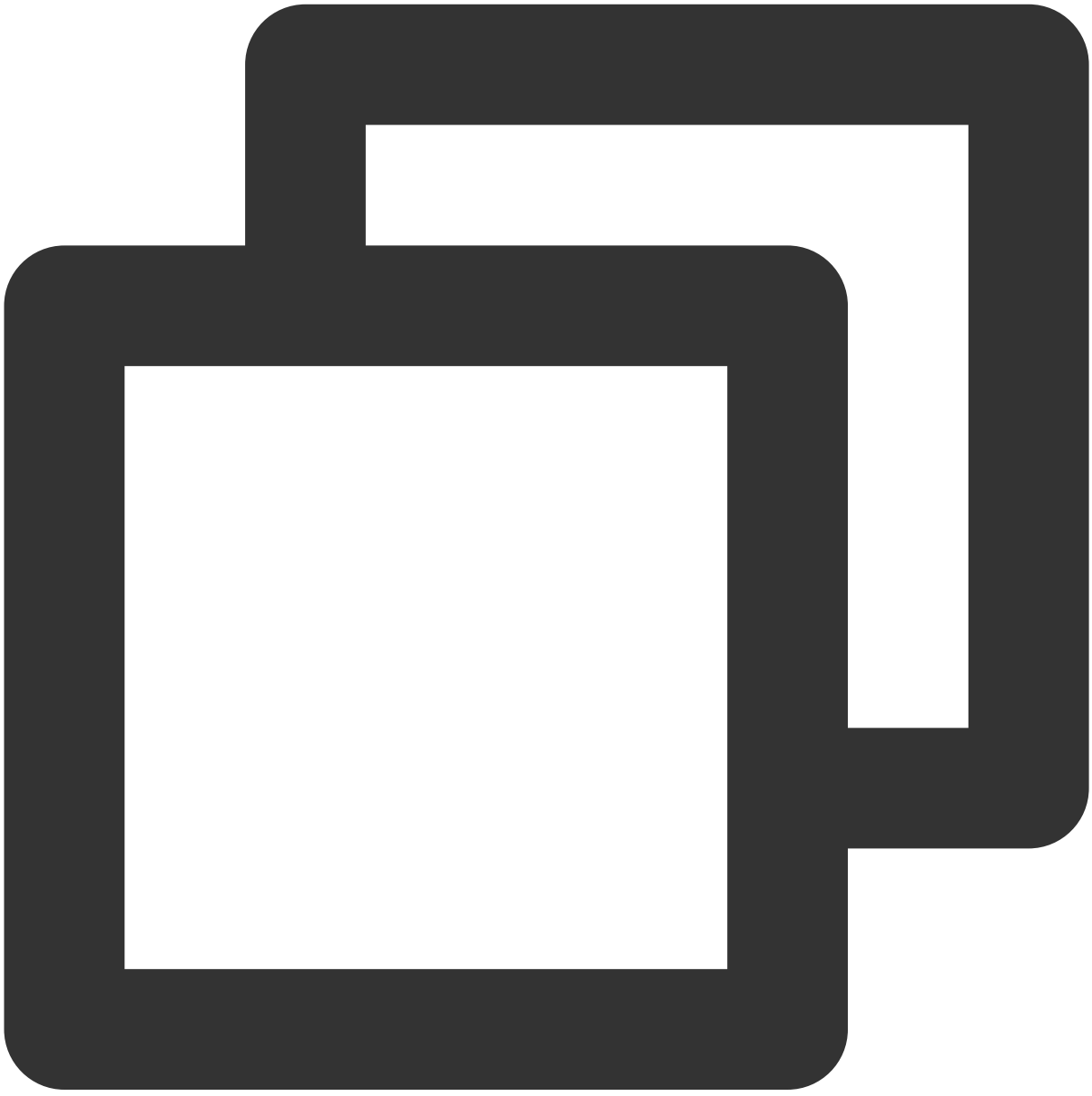
部署 mysqld-exporter 后，添加监控采集配置，让 mysqld-exporter 暴露的数据可被采集。ServiceMonitor 定义示例如下（需要集群中支持）：



```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: mysqld-exporter
spec:
  endpoints:
    interval: 5s
    targetPort: 9104
  namespaceSelector:
    matchNames:
    - default
```

```
    selector:
      matchLabels:
        app: mysqld-exporter
```

Prometheus 原生配置示例如下：



```
    - job_name: mysqld-exporter
      scrape_interval: 5s
      kubernetes_sd_configs:
      - role: endpoints
        namespaces:
          names:
```

```
      - default
    relabel_configs:
    - action: keep
      source_labels:
      - __meta_kubernetes_service_label_app_kubernetes_io_name
      regex: mysqld-exporter
    - action: keep
      source_labels:
      - __meta_kubernetes_endpoint_port_name
      regex: http
```

添加监控面板

监控采集配置能正常采集数据之后，还需要为 Grafana 添加监控面板进行展示。
如果只需观察 MySQL 或 MariaDB 的概览情况，可导入面板 grafana.com。如下图所示：

如果需要更丰富的面板，导入 percona 开源面板 中 `MySQL_` 开头的 json 文件中的内容即可。

自建 Prometheus 迁移到云原生监控

最近更新时间：2022-04-18 16:57:34

操作场景

腾讯云容器服务 云原生监控 兼容 Prometheus 与 Grafana 的 API，同时也兼容主流 prometheus-operator 的 CRD 用法，为云原生监控提供了极大的灵活性与扩展性，结合 Prometheus 开源生态工具可以解锁更多高级用法。
本文将介绍如何通过辅助脚本和迁移工具，快速将自建 Prometheus 迁移到云原生监控。

前提条件

已在自建 Prometheus 集群的一个节点上安装 Kubectl 并配置好 Kubeconfig，保证通过 Kubectl 能够管理集群。

操作步骤

迁移动态采集配置

自建 Prometheus 若使用 prometheus-operator，通常需要通过 ServiceMonitor 和 PodMonitor 这类 CRD 资源来动态添加采集配置，云原生监控同样支持该用法。若只将自建 Prometheus 集群的 prometheus-operator 迁移到云原生监控，并未迁移集群，则无需迁移动态配置，只需使用云原生监控关联自建集群，自建 Prometheus 创建的 ServiceMonitor 和 PodMonitor 资源即可自动在云原生监控中生效。

如需跨集群迁移，可以将自建 Prometheus 的 CRD 资源导出，并选择性的在被关联的云原生监控的集群中重新应用。以下为您介绍如何在自建 Prometheus 集群中批量导出 ServiceMonitor 和 PodMonitor。

1. 创建脚本 `prom-backup.sh`，脚本内容如下：

```
_ns_list=$(kubectl get ns | awk '{print $1}' | grep -v NAME)
count=0
declare -a types=("servicemonitors.monitoring.coreos.com" "podmonitors.monitoring.coreos.com")
for _ns in ${_ns_list}; do
## loop for types
for _type in "${types[@]}"; do
echo "Backup type [namespace: ${_ns}, type: ${_type}]."
_item_list=$(kubectl -n ${_ns} get ${_type} | grep -v NAME | awk '{print $1}' )
## loop for items
for _item in ${_item_list}; do
```

```
_file_name=./${_ns}_${_type}_${_item}.yaml
echo "Backup kubernetes config yaml [namespace: ${_ns}, type: ${_type}, item:
${_item}] to file: ${_file_name}"
kubectl -n ${_ns} get ${_type} ${_item} -o yaml > ${_file_name}
count=$[count + 1]
echo "Backup No.${count} file done."
done;
done;
done;
```

2. 执行以下命令，运行 `prom-backup.sh` 脚本：

```
bash prom-backup.sh
```

3. `prom-backup.sh` 脚本会将每个 ServiceMonitor 与 PodMonitor 资源导出成单独的 YAML 文件。可执行 `ls` 命令查看输出的文件列表，示例如下：

```
$ ls
kube-system_servicemonitors.monitoring.coreos.com_kube-state-metrics.yaml
kube-system_servicemonitors.monitoring.coreos.com_node-exporter.yaml
monitoring_servicemonitors.monitoring.coreos.com_coredns.yaml
monitoring_servicemonitors.monitoring.coreos.com_grafana.yaml
monitoring_servicemonitors.monitoring.coreos.com_kube-apiserver.yaml
monitoring_servicemonitors.monitoring.coreos.com_kube-controller-manager.yaml
monitoring_servicemonitors.monitoring.coreos.com_kube-scheduler.yaml
monitoring_servicemonitors.monitoring.coreos.com_kube-state-metrics.yaml
monitoring_servicemonitors.monitoring.coreos.com_kubelet.yaml
monitoring_servicemonitors.monitoring.coreos.com_node-exporter.yaml
```

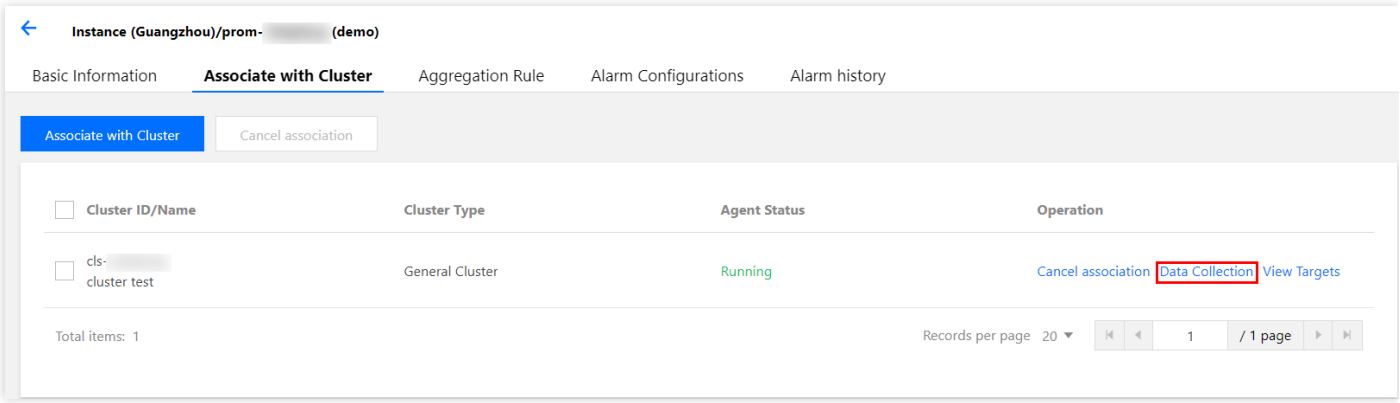
4. 您可以自行筛选和修改，将 YAML 文件重新应用到被关联的云原生监控的集群中（请勿应用已经存在或功能相同的采集规则），云原生监控会自动感知这部分动态采集规则并进行采集。

> 说明：
> 若后续需增加 ServiceMonitor 或 PodMonitor，可以通过 TKE 控制台进行可视化添加，也可脱离控制台直接用 YAML 创建，用法与 Prometheus 社区的 CRD 完全兼容。

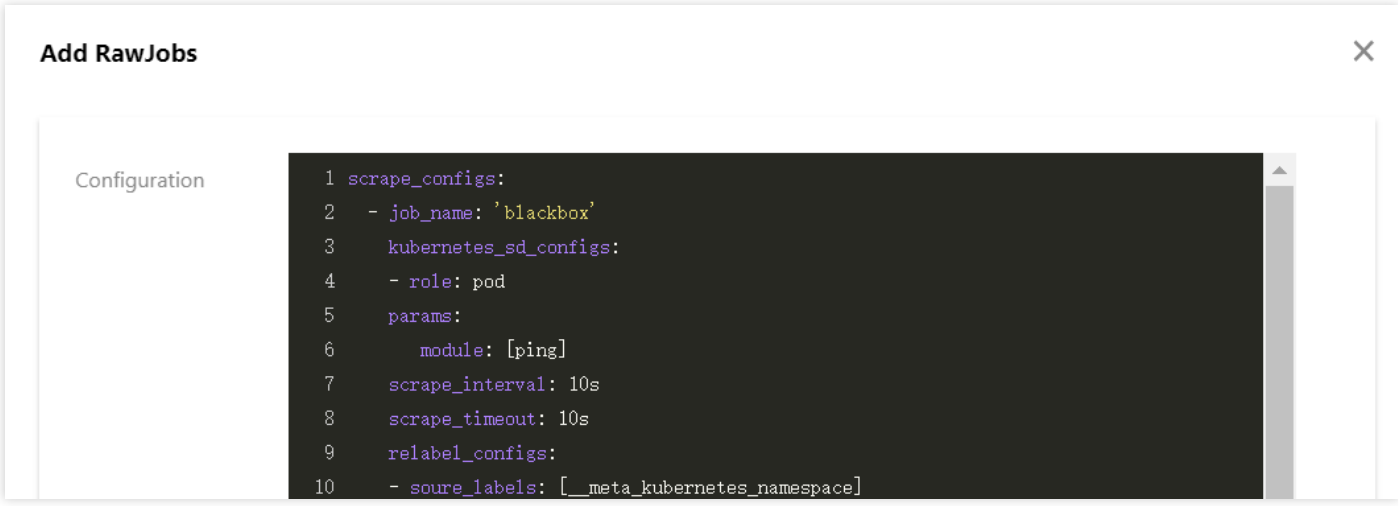
迁移静态采集配置

若自建 Prometheus 系统直接使用 Prometheus 原生配置文件，只需在 TKE 控制台进行简单的几步操作，即可将其转换为云原生监控的 RawJob，使其兼容 Prometheus 原生配置文件的 `scrape_configs` 配置项。

1. 登录 容器服务控制台。
2. 在左侧菜单栏中单击 **云原生监控**进入云原生监控页面。
3. 单击需要配置的云原生监控 ID/名称，进入基本信息页面。
4. 选择**关联集群**页签，在对应的集群右侧**操作**列项下单击**数据采集配置**。



5. 选择**RawJob>新增**，打开添加 RawJobs 窗口。将原生 Prometheus 配置文件中的 Job 配置复制粘贴到此配置窗口中。



6. 可以将所有需要导入的 Job 数组都粘贴到云原生监控，单击**确定**后会自动拆分成多个 RawJob，名称为每个 Job 的 `job_name` 字段。

迁移全局配置

云原生监控提供 Prometheus CRD 资源，可以通过修改该资源来修改全局配置。

1. 执行以下命令，获取 Prometheus 相关信息。

```
$ kubectl get ns
prom-fnc7bvu9 Active 13m
$ kubectl -n prom-fnc7bvu9 get prometheus
NAME VERSION REPLICAS AGE
tke-cls-hha93bp9 11m
$ kubectl -n prom-fnc7bvu9 edit prometheus tke-cls-hha93bp9
```

2. 执行以下命令，修改 Prometheus 相关配置。

```
$ kubectl -n prom-fnc7bvu9 edit prometheus tke-cls-hha93bp9
```

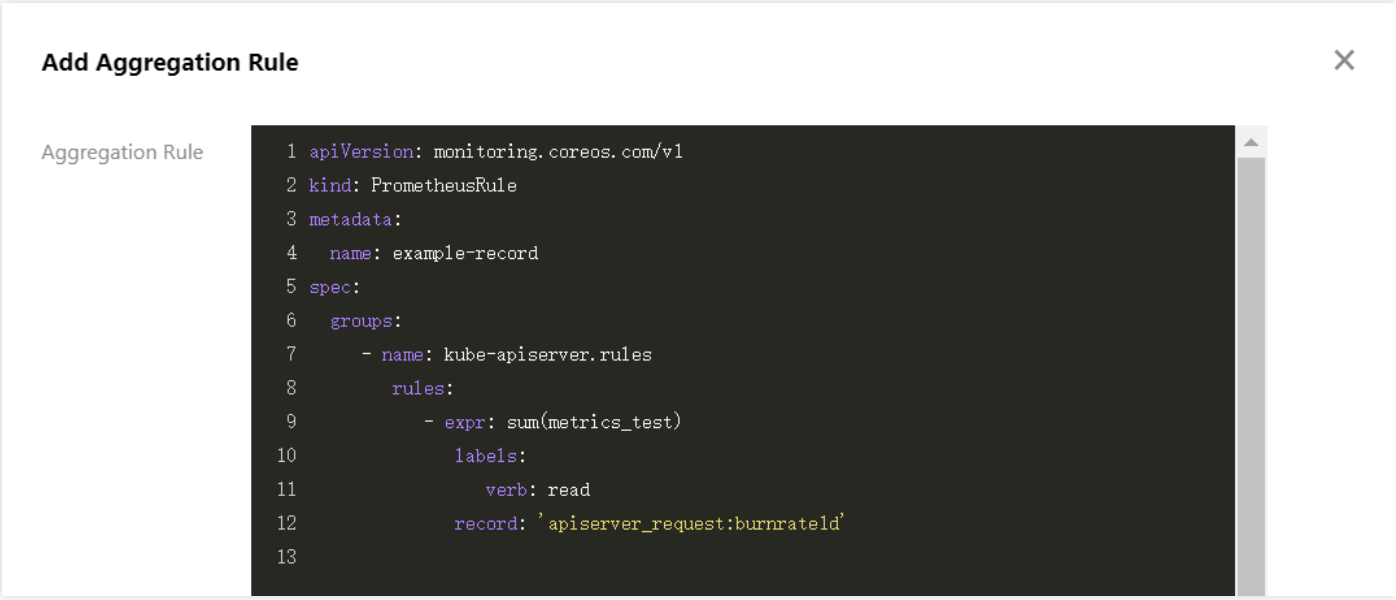
在弹出的编辑页面，您可修改以下参数：

- **scrapeInterval**：采集抓取间隔时长（默认为15s）。
- **externalLabels**：可为所有时序数据增加默认的 label 标识。

迁移聚合配置

Prometheus 的聚合配置，无论是原始 Recording rules 静态配置或是 PrometheusRule 动态配置，每条规则的格式都相同。

1. 登录 容器服务控制台。
2. 在左侧菜单栏中单击 **云原生监控**进入云原生监控页面。
3. 单击需要配置的云原生监控 ID/名称，进入基本信息页面。
4. 选择**聚合规则>新建聚合规则**，打开新建聚合规则窗口。使用 PrometheusRule 格式将每条规则粘贴到 groups 数组中。如下图所示：



说明：
若自建 Prometheus 本身使用 PrometheusRule 定义的聚合规则，仍建议将其按照上述步骤进行迁移。若直接使用 YAML 方式在集群中创建 PrometheusRule 资源，云原生监控暂时无法将其显示到控制台。

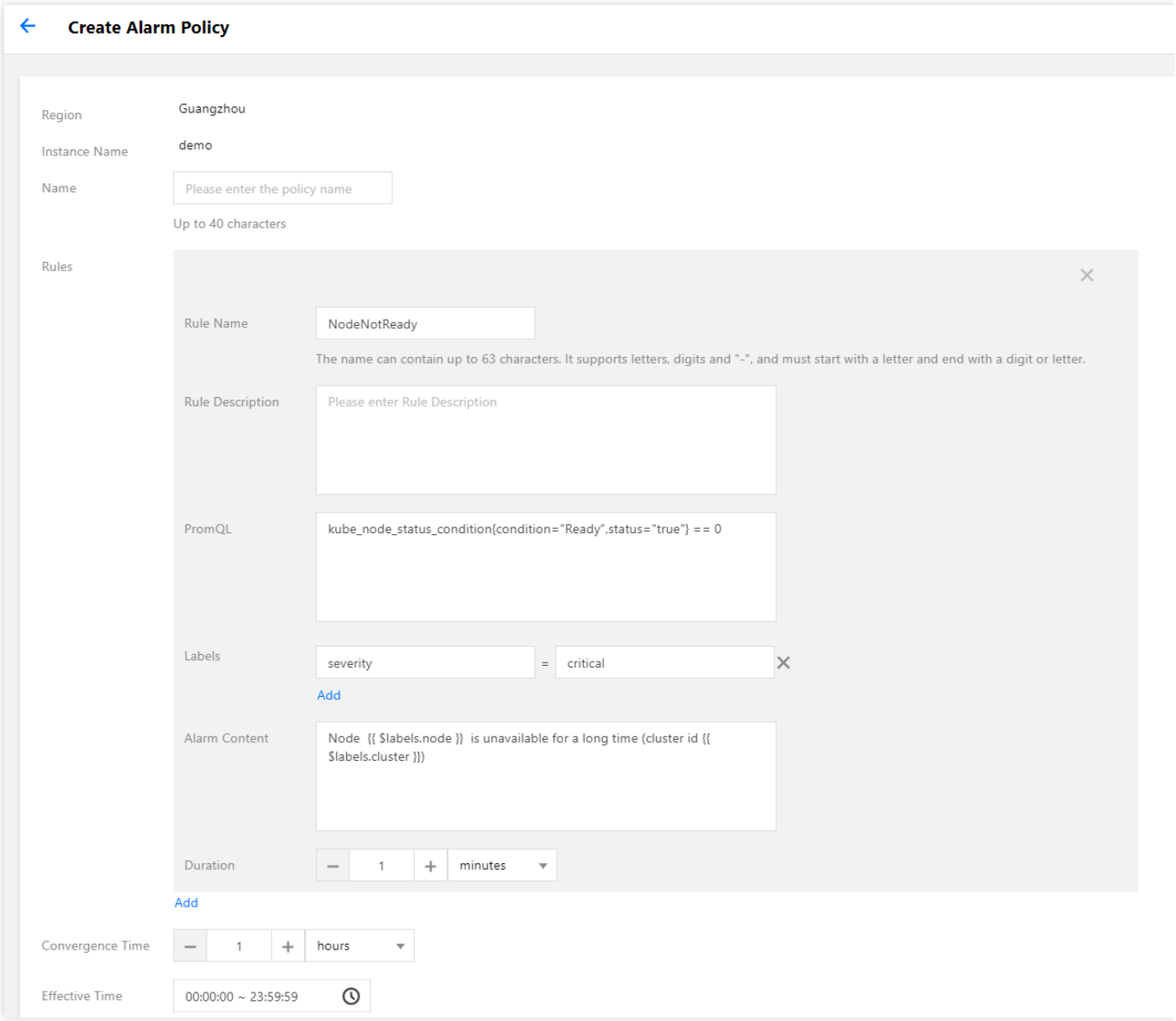
迁移告警配置

本文提供以下自建 Prometheus 告警原始配置 YAML 文件为例，介绍如何将其转换为云原生监控类似的监控配置。

```
- alert: NodeNotReady
expr: kube_node_status_condition{condition="Ready",status="true"} == 0
for: 5m
labels:
severity: critical
annotations:
description: 节点 {{ $labels.node }} 长时间不可用 (集群id {{ $labels.cluster }})
```

1. 登录 容器服务控制台。
2. 在左侧菜单栏中单击 **云原生监控**进入云原生监控页面。
3. 单击需要配置的云原生监控 ID/名称，进入基本信息页面。

4. 选择 **告警配置>新建告警策略**，配置告警策略：



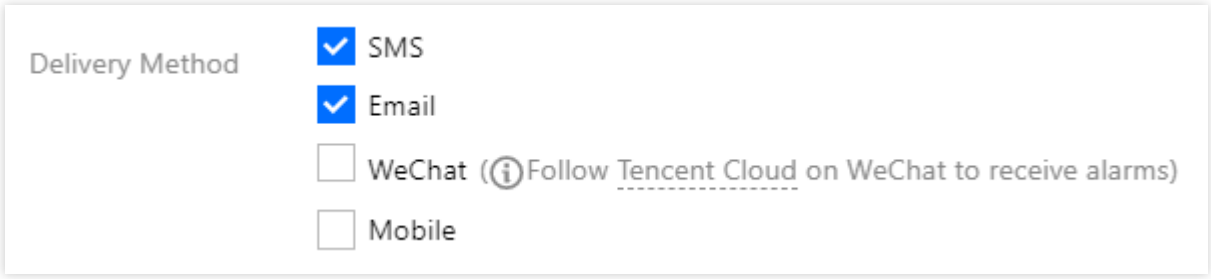
主要参数信息如下：

- **PromQL**：等同于 原始配置 的 expr 字段，为告警的核心配置，用于指示告警触发条件的 PromQL 表达式。
- **Labels**：等同于 原始配置 的 labels 字段，为告警添加额外的 label。
- **告警内容**：表示推送的告警内容，通常使用模板，可插入变量。建议带上集群 ID，可使用变量 `{{ $labels.cluster }}` 表示集群 ID。
- **持续时间**：等同于 原始配置 的 for 字段，表示达到告警条件多久之后还未恢复就推送告警。本文示例配置为5分钟。
- **收敛时间**：等同于 AlertManager 的 repeat_interval 配置，表示某个告警推送之后多久之后还未恢复就再次推送，即相同告警的推送间隔时长。本文示例配置为1小时。

> 说明：
> 上述告警配置示例表示节点状态变为 NotReady 之后，5分钟内未恢复即推送告警，如果长时间未恢复，则间隔1小时再次推送告警。

5. 配置告警渠道，目前支持腾讯云与 WebHook 两类：
   - 腾讯云告警渠道
   - WebHook 告警渠道

   腾讯云告警渠道集成短信、邮件、微信、电话告警方式，可根据自身需求勾选：



迁移 Grafana 面板

自建 Prometheus 通常配置了许多自定义的 Grafana 监控面板，如需迁移到其他平台，在面板数量较多的情况下，依次导出再导入方式效率太低。借助 grafana-backup 工具可以实现 Grafana 面板的批量导出和导入，您可以参考以下批量导出导入面板步骤进行快速迁移。

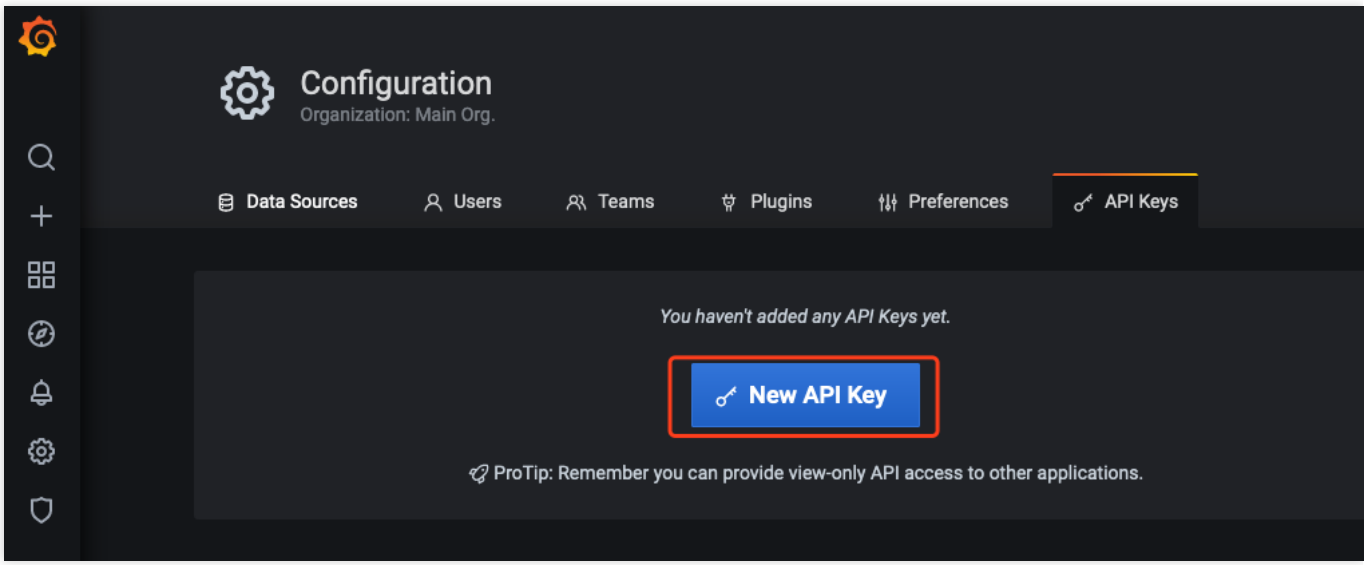
1. 执行以下命令安装 grafana-backup。示例如下：

```
pip3 install grafana-backup
```

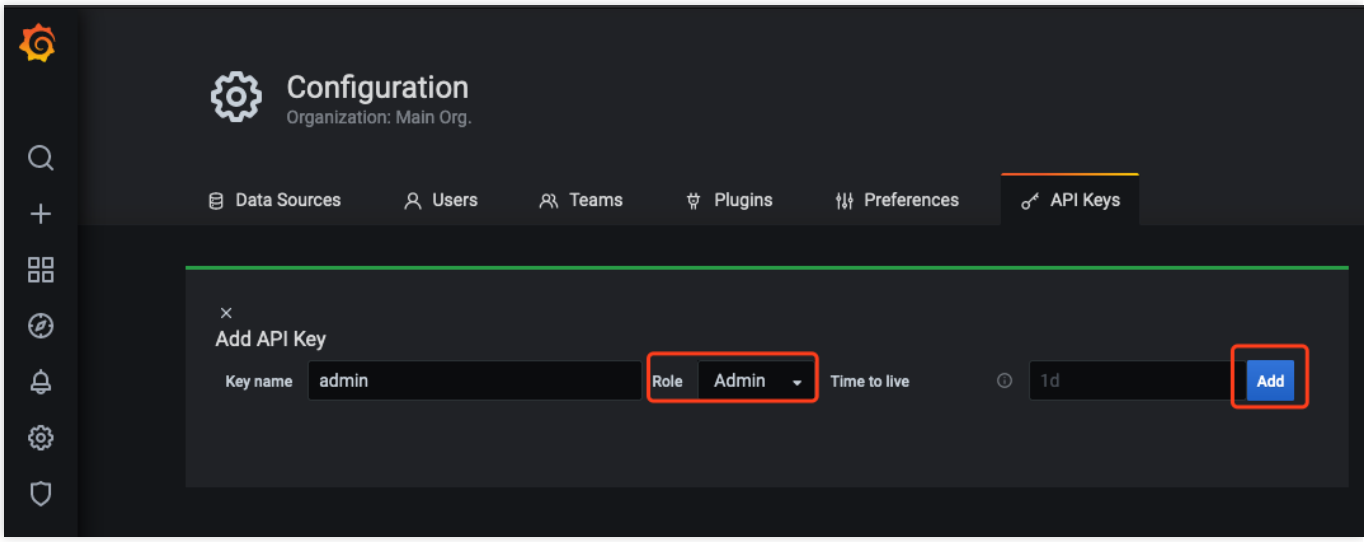
> 说明：
> 推荐使用 Python3，使用 Python2 可能存在兼容性问题。

2. 创建 API Keys。

i. 分别打开自建 Grafana 与云原生监控 Grafana 的配置面板，选择**API Keys>New API Key**，如下图所示：



ii. 在 Add API Key 窗口中，创建一个 Role 为 Admin 的 APIKey，如下图所示：



3. 为需要导出的面板准备备份配置文件。

i. 执行以下命令，获取自建 Grafana 的访问地址。示例如下：

```
$ kubectl -n monitoring get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
grafana ClusterIP 172.21.254.127 <none> 3000/TCP 25h
```

说明：

上述 Grafana 集群内访问地址以 `http://172.21.254.127:3000` 为例。

ii. 执行以下命令，生成 grafana-backup 配置文件（写入 Grafana 地址与 APIKey）。示例如下：

```
export TOKEN=<TOKEN>
cat > ~/.grafana-backup.json <<EOF
{
"general": {
"debug": true,
"backup_dir": "_OUTPUT_"
},
"grafana": {
"url": "http://172.21.254.127:3000",
"token": "${TOKEN}"
}
}
EOF
```

说明：
<TOKEN> 需要替换为自建 Grafana 的 APIKey，url 地址需替换为实际环境地址。

4. 执行以下命令，导出所有面板。示例如下：

```
grafana-backup save
```

面板将以一个压缩文件的形式保存在 `_OUTPUT_` 目录下，您可以执行以下命令查看该目录下存在的文件。示例如下：

```
$ tree _OUTPUT_
_OUTPUT_
└── 202012151049.tar.gz
0 directories, 1 file
```

5. 执行以下命令，准备还原配置文件。示例如下：

```
export TOKEN=<TOKEN>
cat > ~/.grafana-backup.json <<EOF
{
```

```
"general": {
"debug": true,
"backup_dir": "_OUTPUT_"
},
"grafana": {
"url": "http://prom-xxxxxx-grafana.ccs.tencent-cloud.com",
"token": "${TOKEN}"
}
}
EOF
```

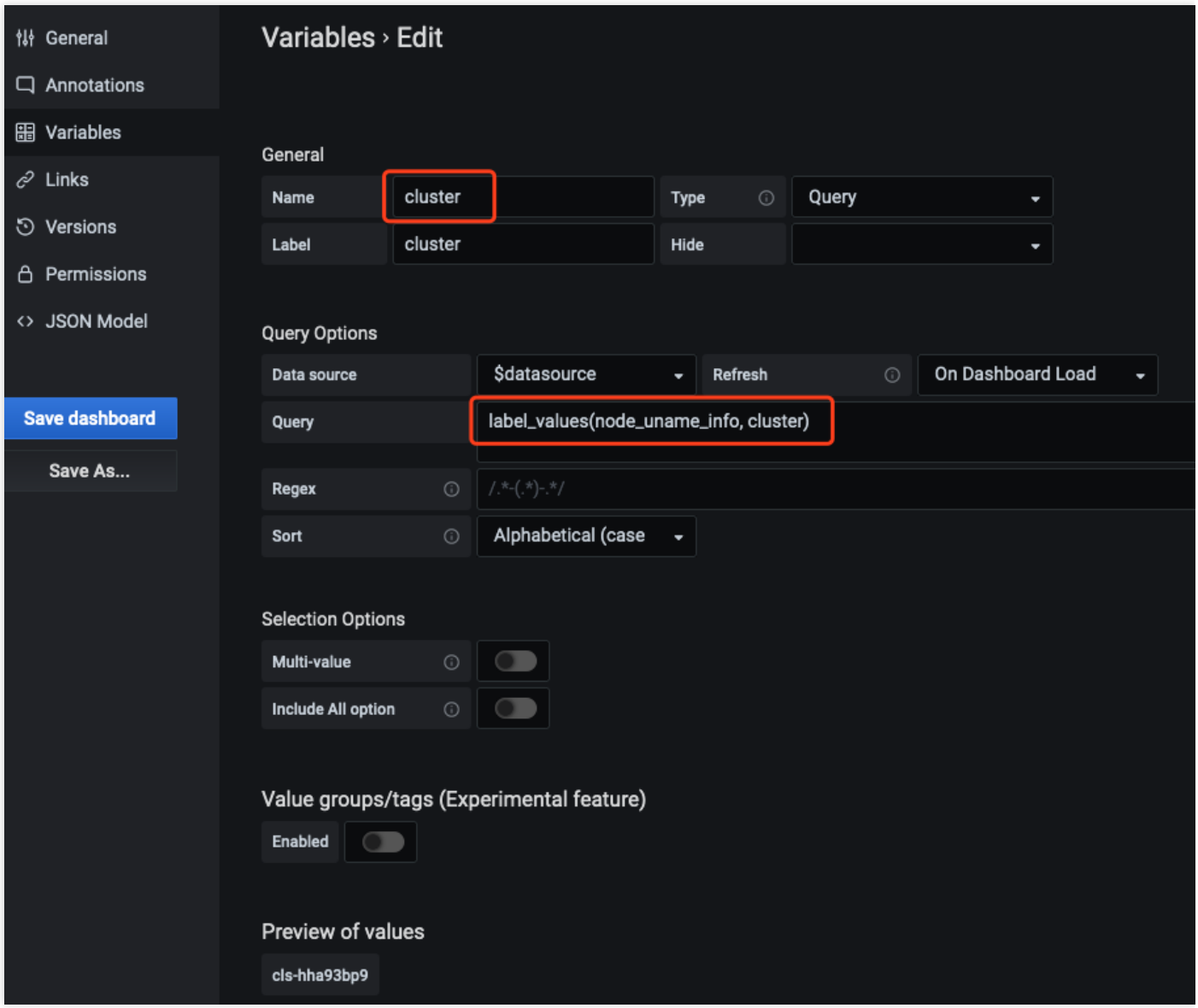
说明：
将 <TOKEN> 替换为云原生监控 Grafana 的 APIKey，url 替换为云原生监控 Grafana 的访问地址（通常用外网访问地址，需开启）。

6. 执行以下命令，将导出的面板一键导入到云原生监控 Grafana。示例如下：

```
grafana-backup restore _OUTPUT_/202012151049.tar.gz
```

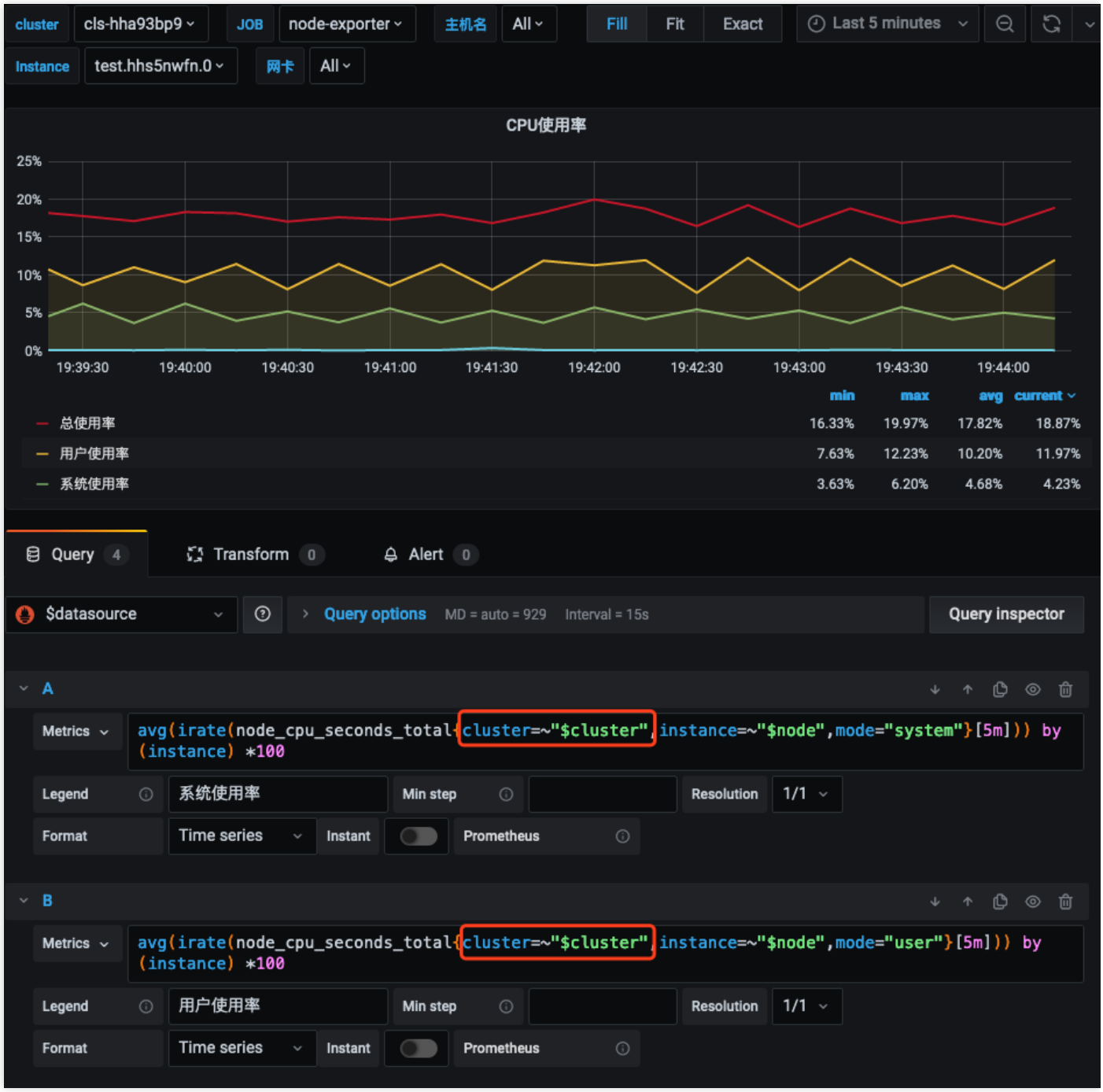
7. 在 Grafana 配置面板选择 **Dashboard settings>Variables>New**， 新建 cluster 字段。建议为所有面板都加上 cluster 的过滤字段，云原生监控支持多集群，将会给每个集群的数据打上 cluster 标签，用集群 ID 来区分不同集

群。如下图所示：



> 说明：
> label_values 中填入当前面板任意涉及到的一个指标名（示例中为 node_uname_info）。

8. 修改所有面板中的 PromQL 查询语句，加入 `cluster=~"$cluster"` 过滤条件。如下图所示：



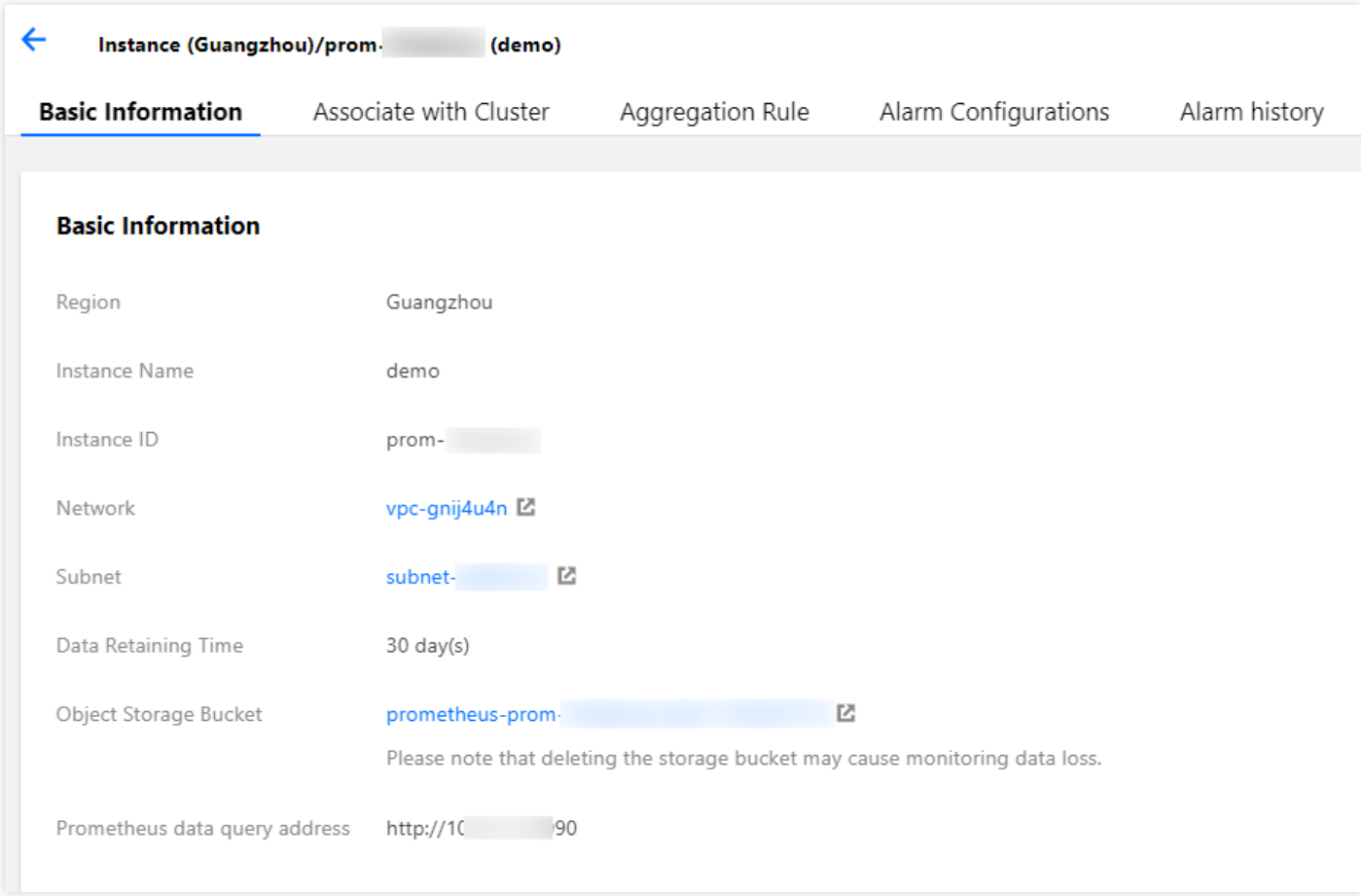
与现有系统集成

云原生监控支持接入自建 Grafana 和 AlertManager 系统：

- 接入自建 Grafana
- 接入自建 AlertManager

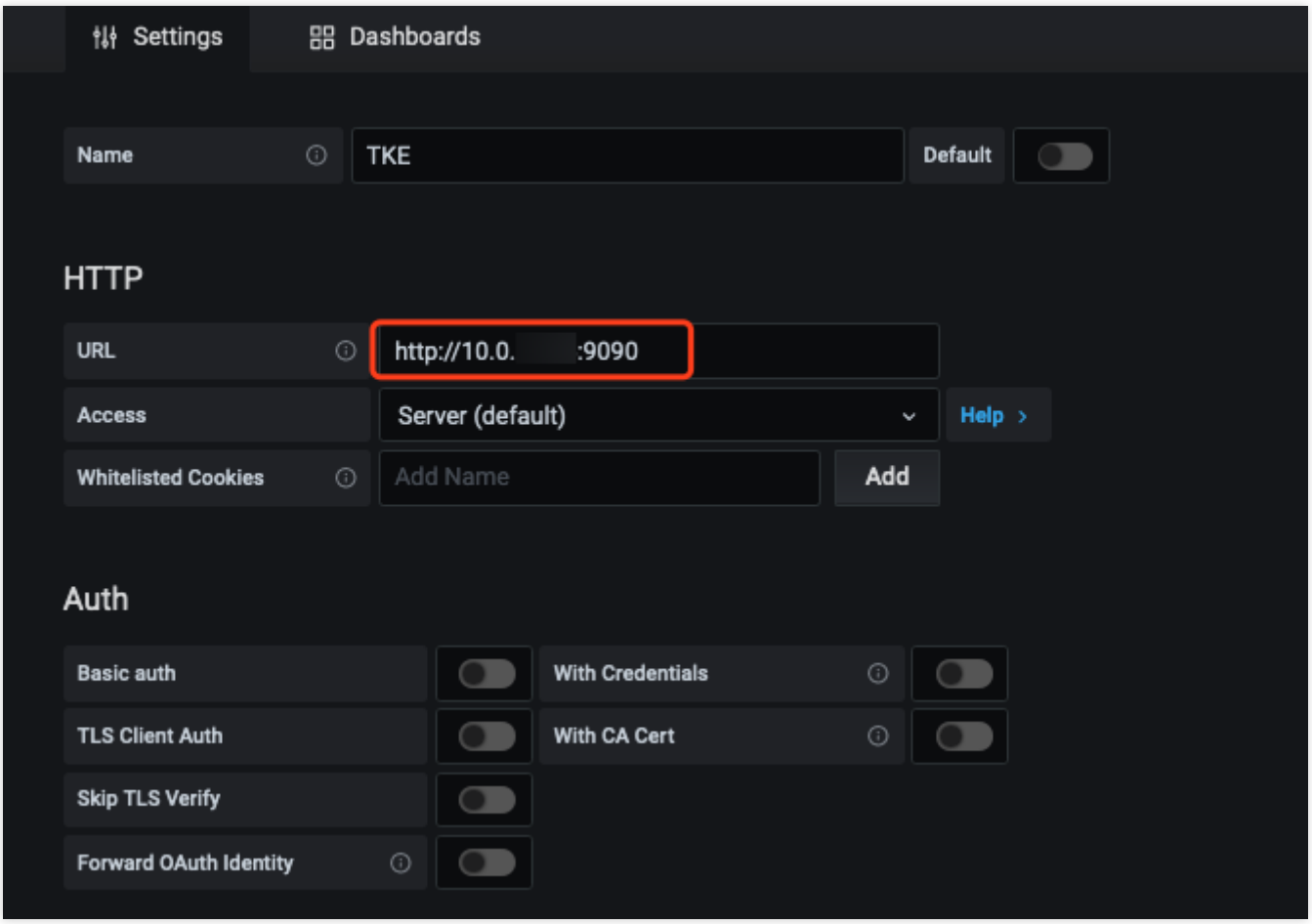
云原生监控提供 Prometheus 的 API，如需使用自建的 Grafana 来展示监控，可以将云原生监控的数据作为一个 Prometheus 数据源添加到自建 Grafana，Prometheus API 的地址可在 TKE 控制台云原生监控基本信息中查到。

1. 登录 容器服务控制台。
2. 在左侧菜单栏中单击 **云原生监控**进入云原生监控页面。
3. 单击需要配置的云原生监控 ID/名称，进入基本信息页面，获取 Prometheus API 地址。



>?确保自建的 Grafana 与云原生监控在同一私有网络 VPC 下或两者网络已打通。

4. 在 Grafana 中添加 Prometheus API 地址作为 Prometheus 数据源。如下图所示：



运维

TKE 集群中节点移出再移入操作指引

最近更新时间：2023-02-02 17:05:22

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。

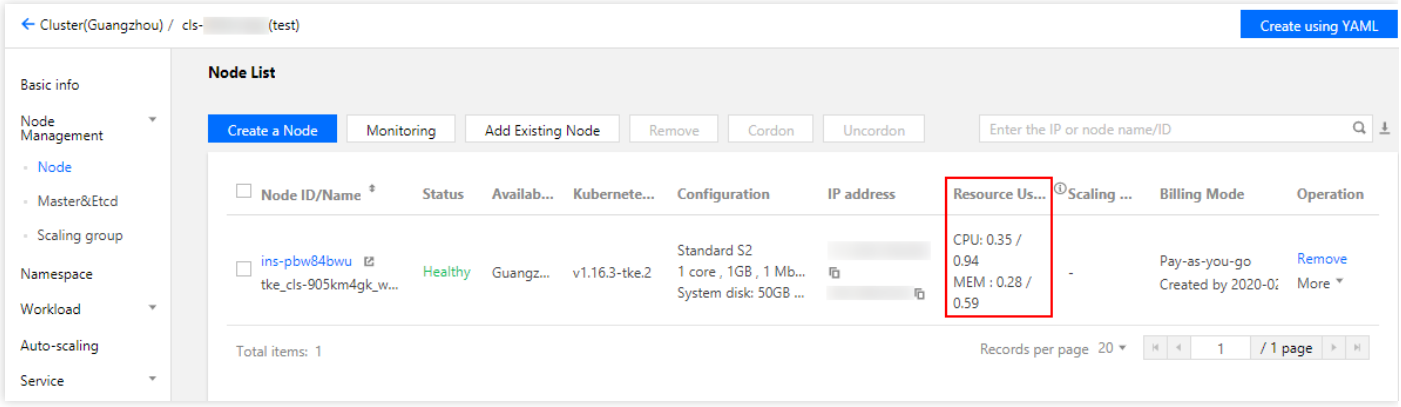
此过程是先删除，再创建，并非滚动更新。因此更新过程中，可能会导致被驱逐的服务部分请求失败，如果被驱逐的服务所有相关 Pod 都在被驱逐的节点上，则可能导致该服务完全不可用。

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。

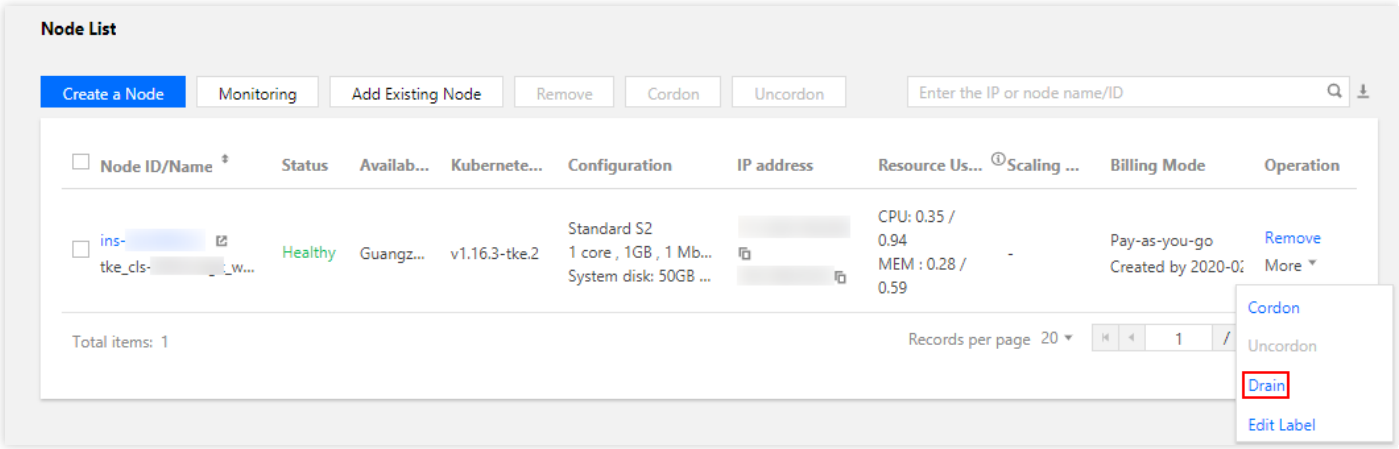
> 说明：
> 目前 kubelet 的镜像拉取策略是串行的，如果短时间内有大量的 Pod 都被调度到同一个节点上之后，Pod 的启动时间有可能会变长。

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
- 通过 kubectl drain 命令行驱逐

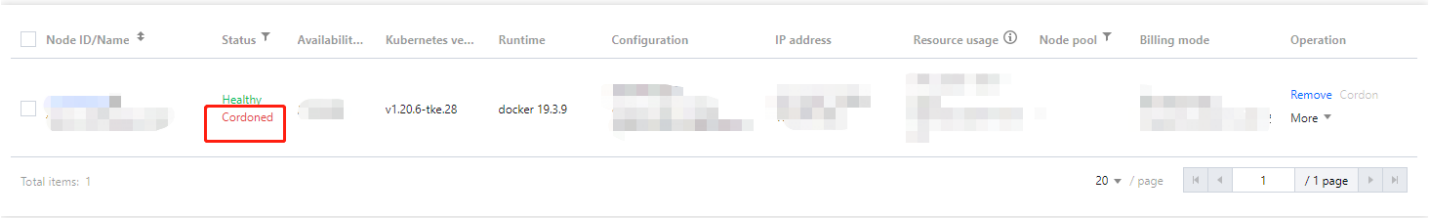
1. 在 集群列表 页面，单击目标集群 ID。
2. 在集群详情页中，选择**节点管理** > **节点**。
3. 在“节点”页面，选择目标节点所在行右侧的**驱逐**。如下图所示：



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。

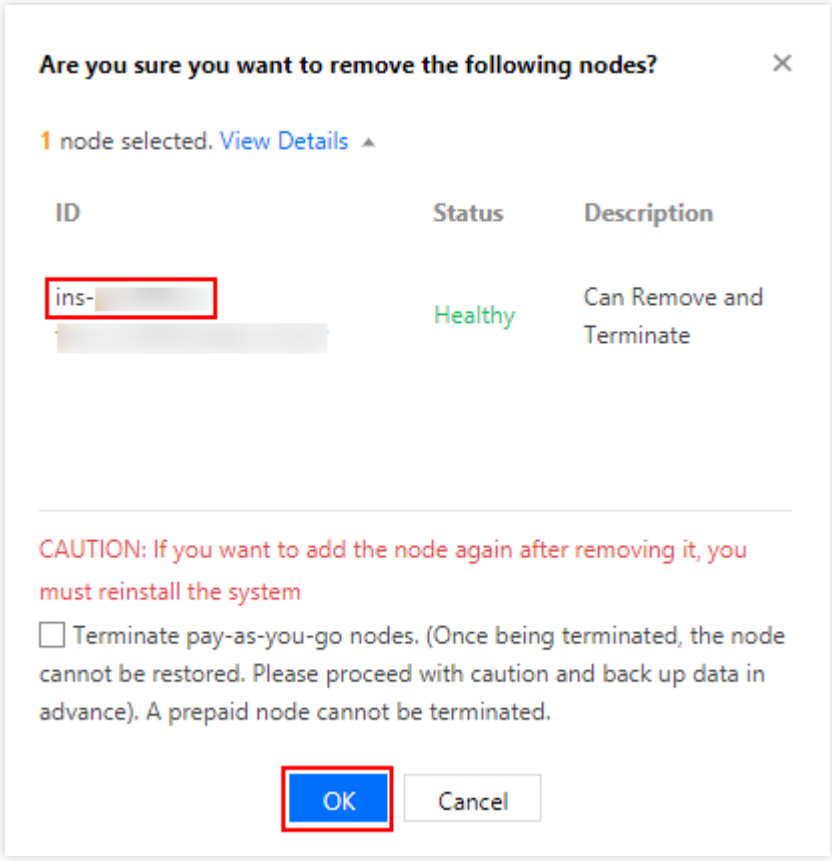
步骤2：移出节点

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：



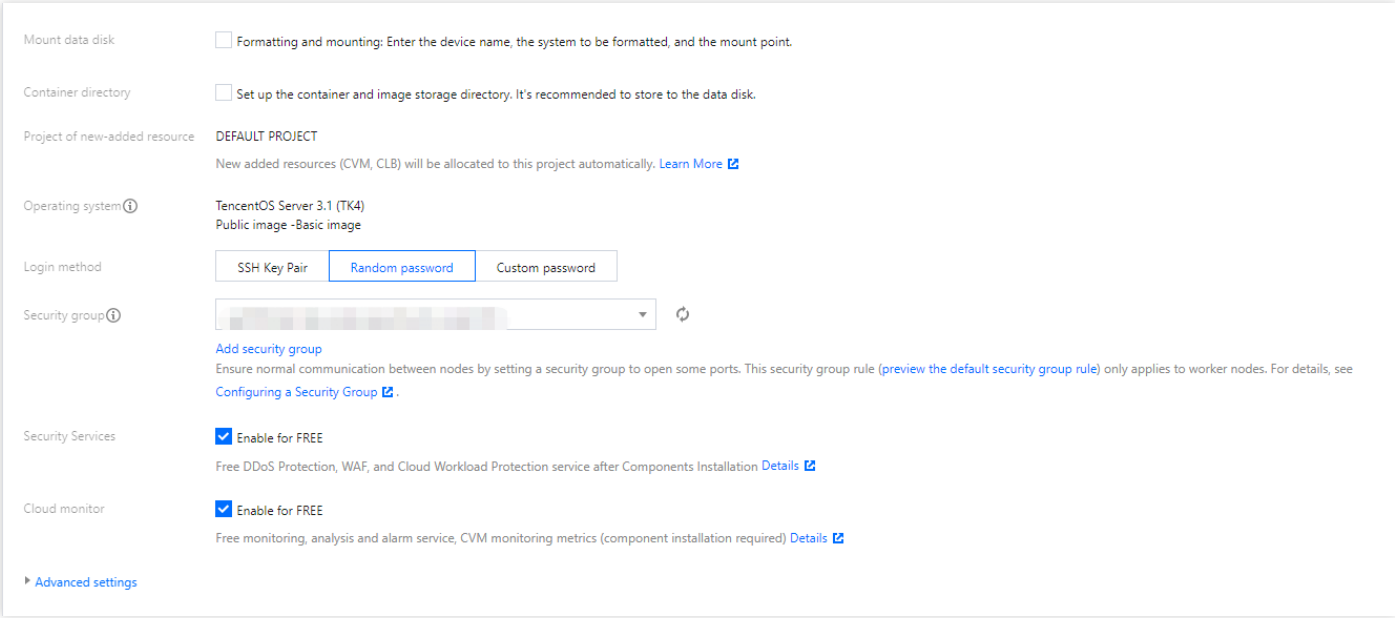
1. 在“节点列表”页面，单击目标节点所在行右侧的**移出**。
2. 在弹出窗口中，取消勾选“销毁按量计费的节点”，并单击**确定**即可将节点移除集群。如下图所示：

> 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。



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

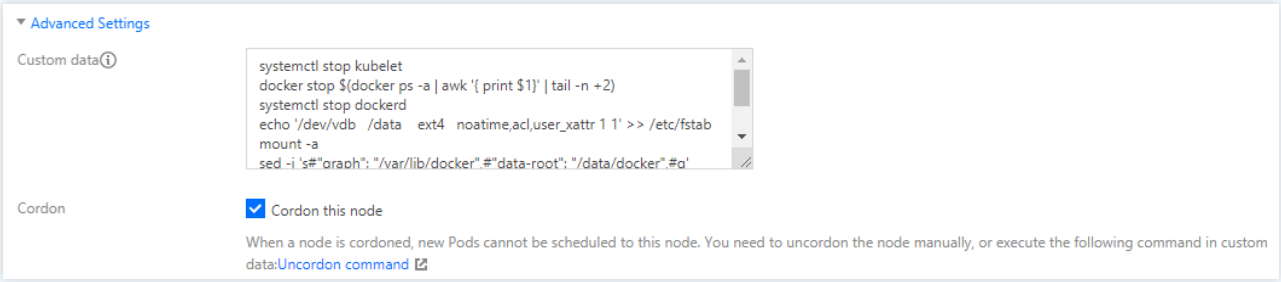
数据盘挂载与**容器目录**默认不勾选。

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 文件系统的系统盘将直接挂载，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：

i. 在“云服务器配置”页面，不勾选**数据盘挂载**。

ii. 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：2020-10-12 16:54:36

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 ansibl
     e -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 Us
erKnownHostsFile=/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 Us
erKnownHostsFile=/dev/null" --user ubuntu --become --become-user=root -e "ans
ible_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 Us
erKnownHostsFile=/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：2023-05-06 17:36:46

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。

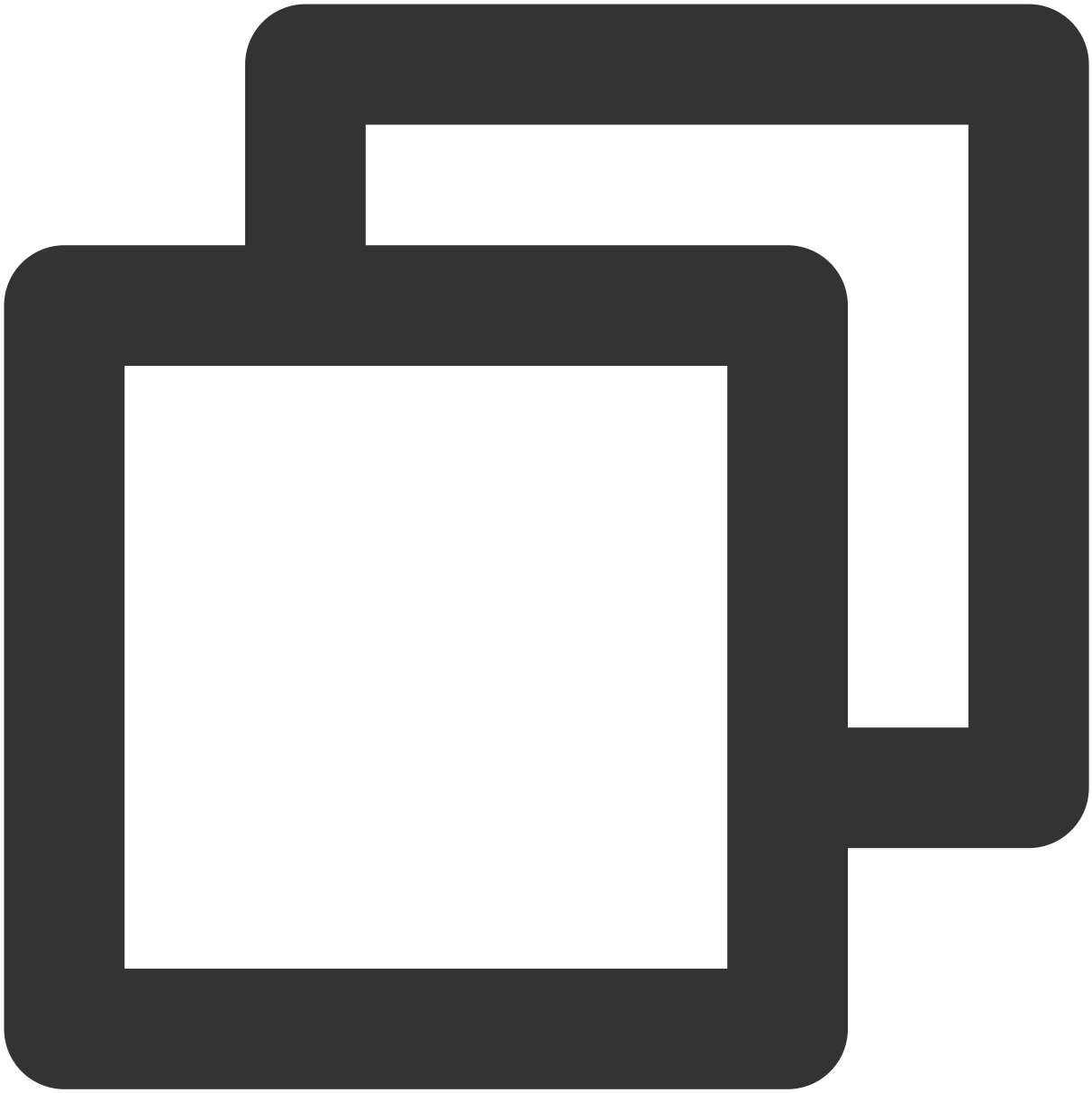
使用示例

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。

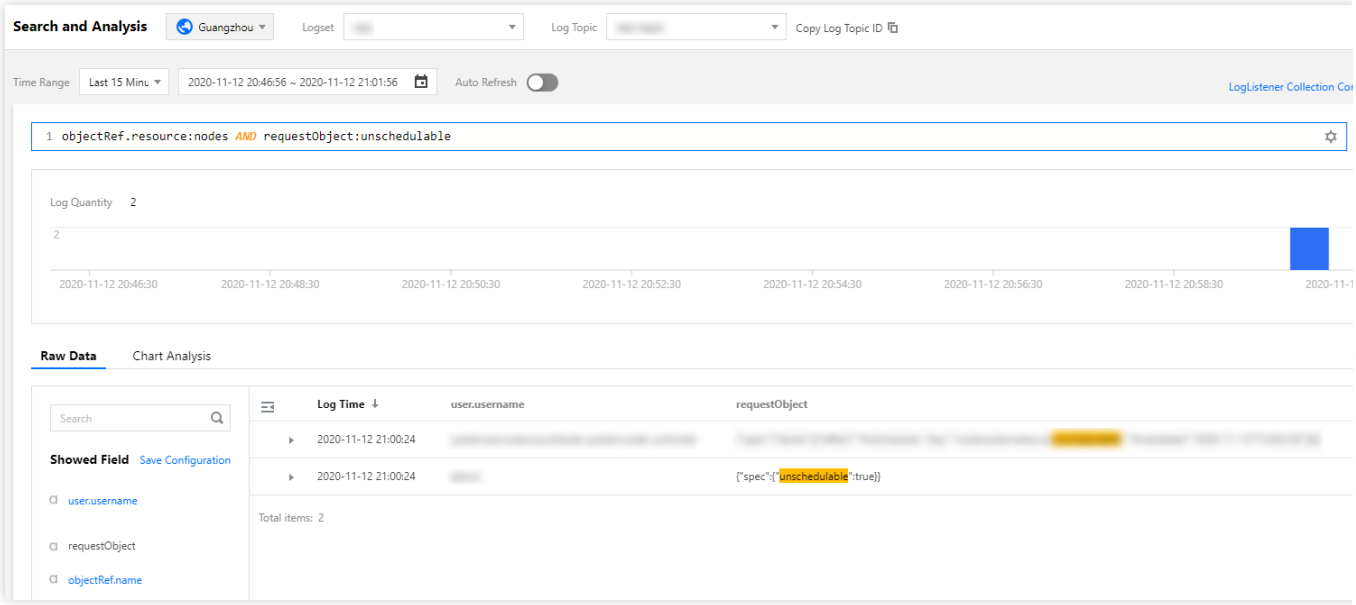
示例1：查询封锁节点的操作者

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：



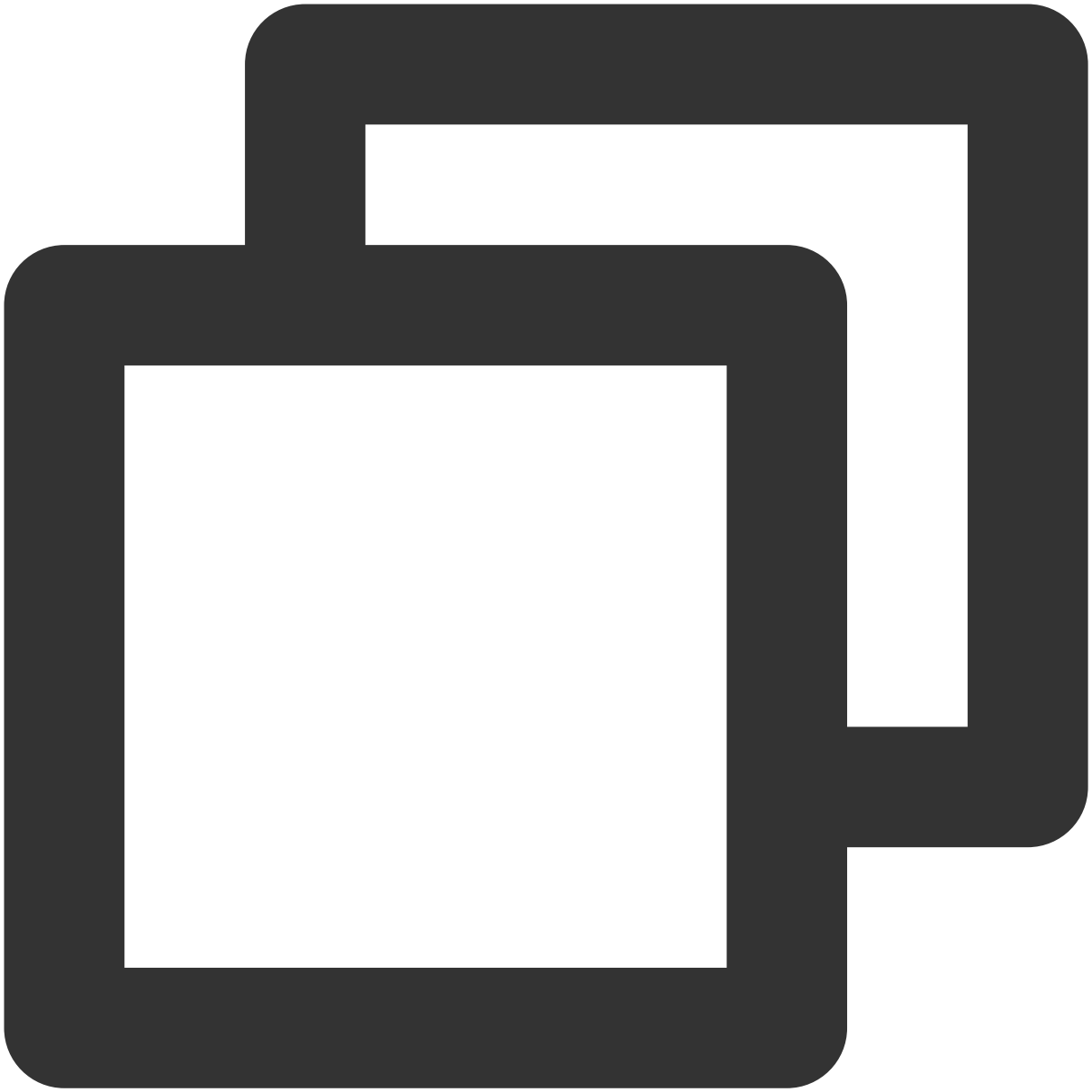
```
objectRef.resource:nodes AND requestObject:unschedulable
```

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：



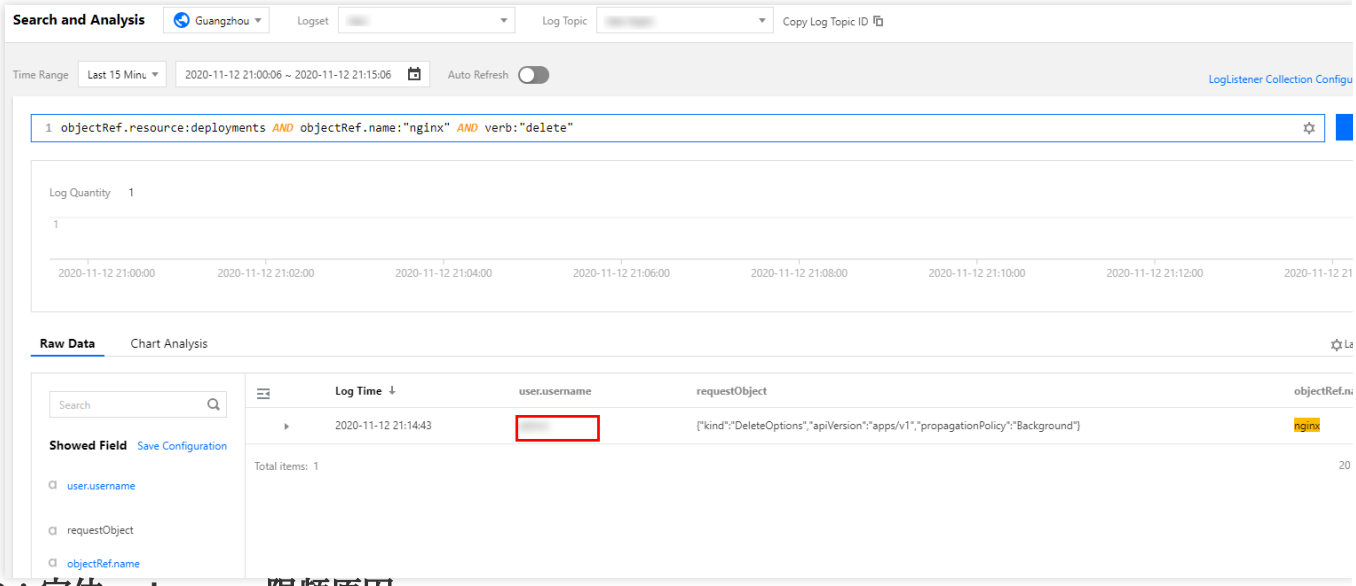
示例2：查询删除工作负载的操作者

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"
```

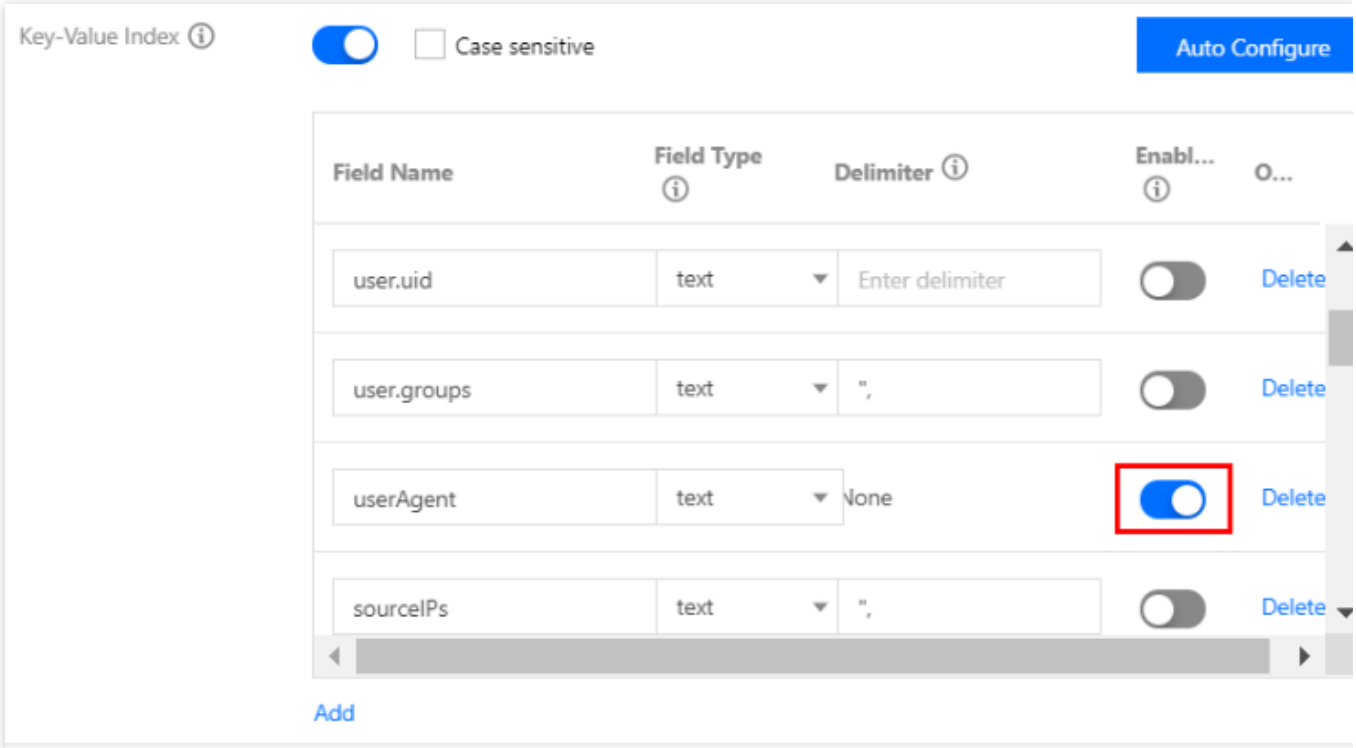
您可根据检索结果获取此子账号的详细信息。



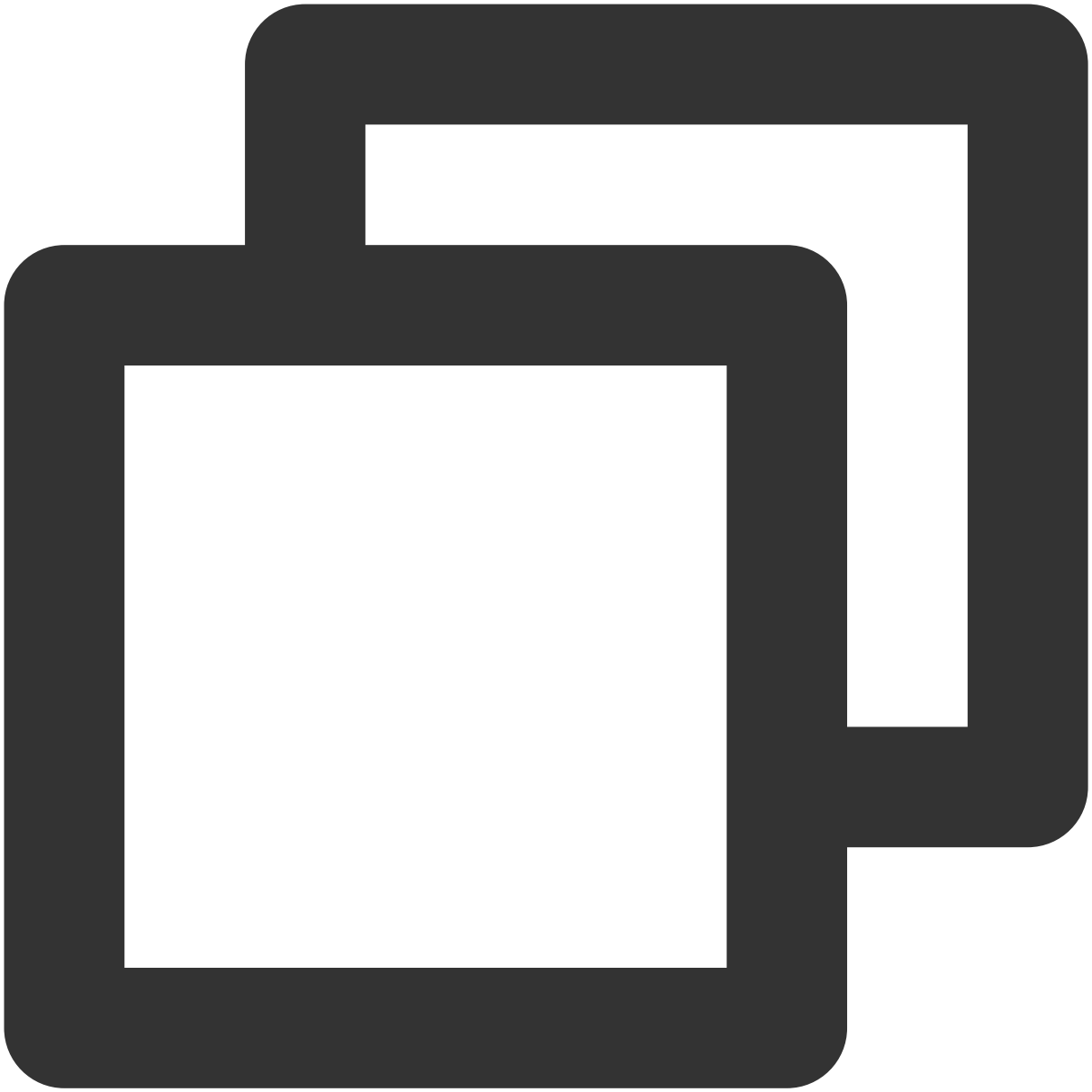
示例3：定位 apiserver 限频原因

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：

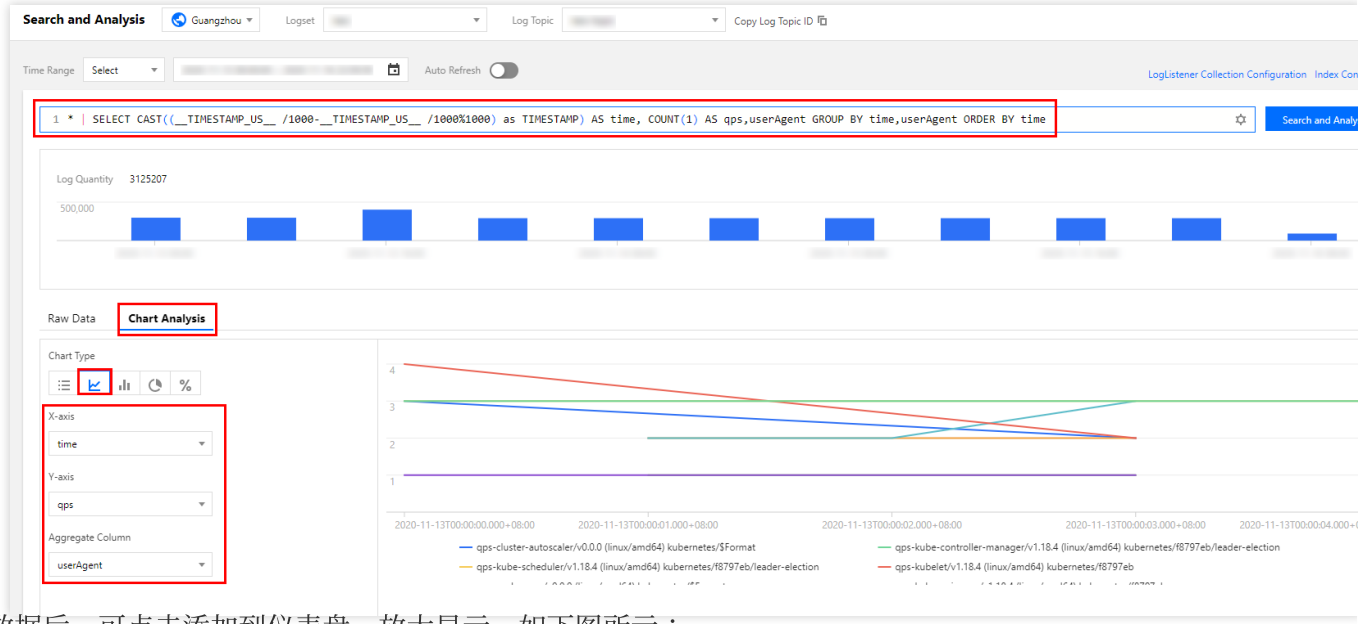


2. 执行以下命令，对每种客户端请求 apiserver 的 QPS 大小进行统计：

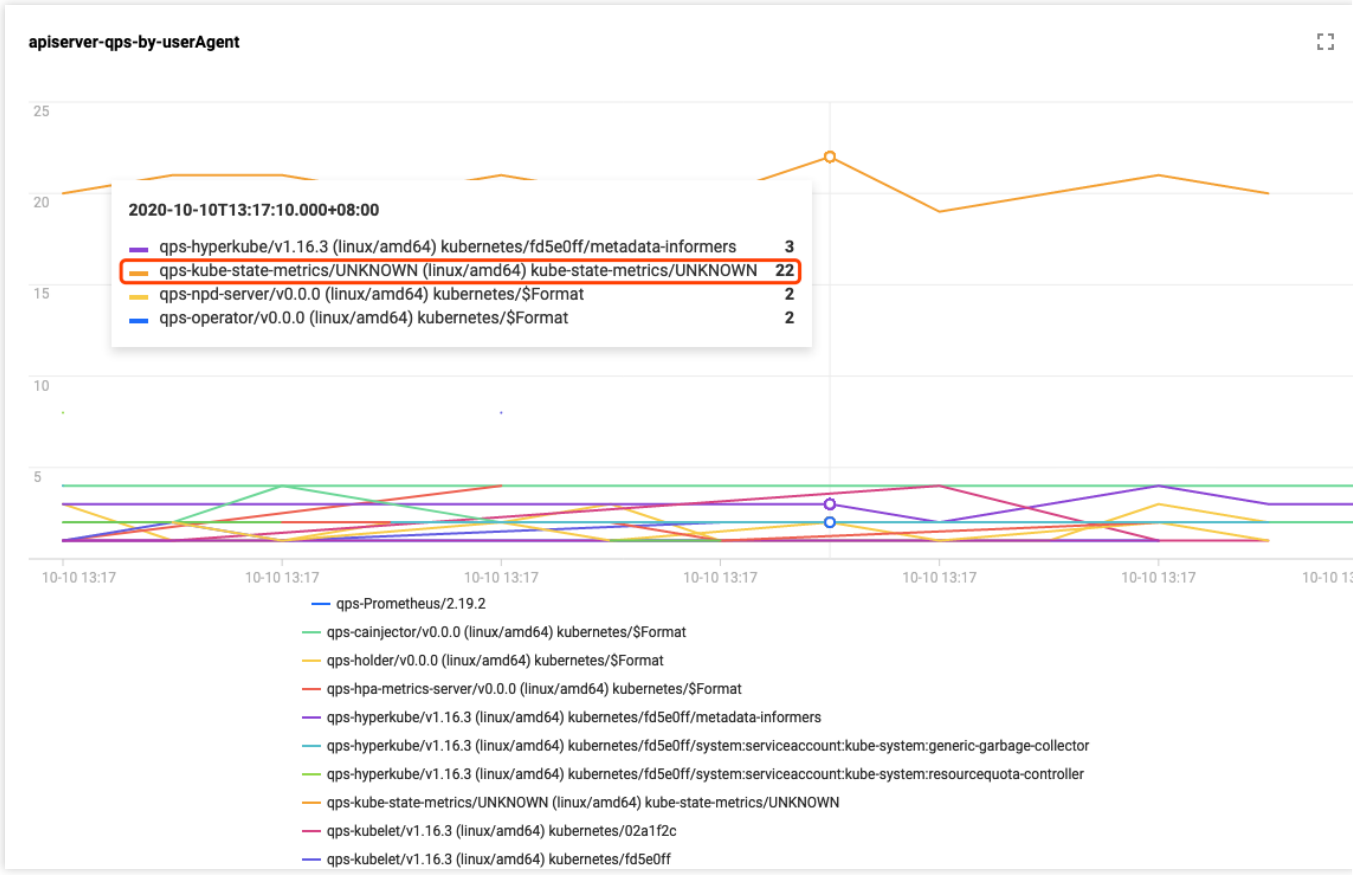


```
* | SELECT histogram( cast(__TIMESTAMP__ as timestamp),interval 1 minute) AS time,
```

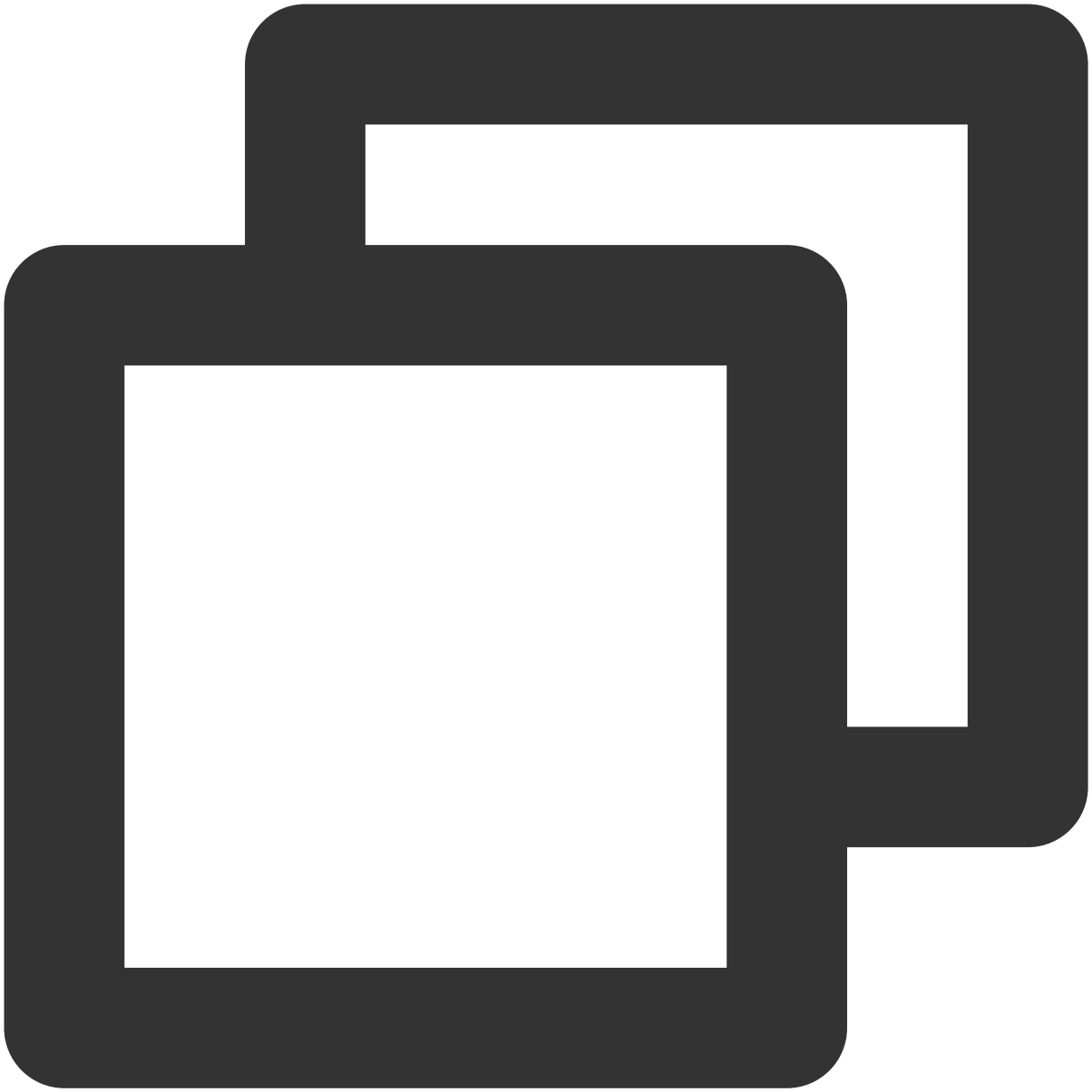
3. 切换到统计图表，选择时序图，可设置基本信息、坐标轴等，如下图所示：



获得数据后，可点击添加到仪表盘，放大显示。如下图所示：

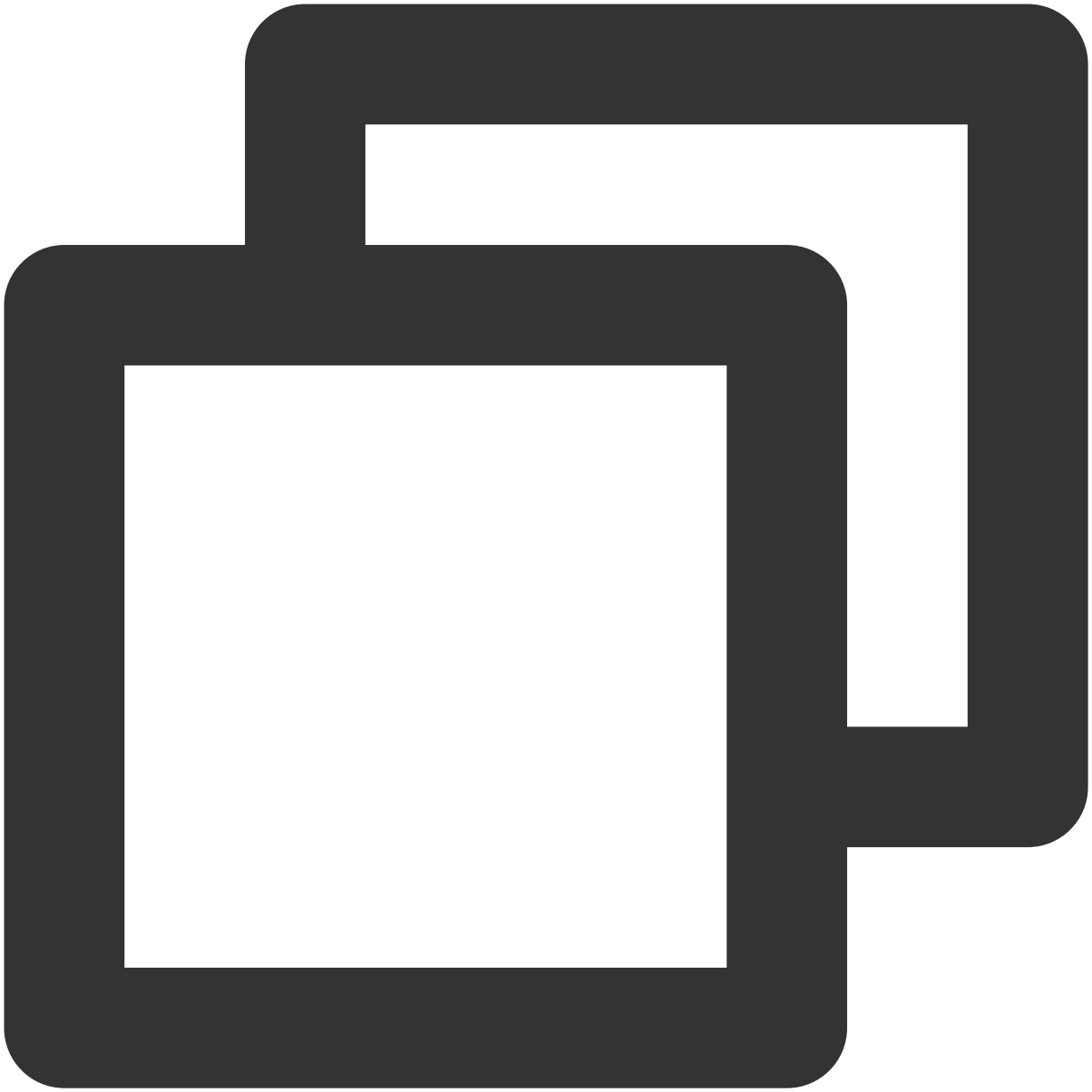


由图可见，kube-state-metrics 客户端对 apiserver 请求频率远远高于其它客户端。查看日志可得，由于 RBAC 权问题导致 kube-state-metrics 不停的请求 apiserver 重试，触发了 apiserver 的限频。日志如下所示：



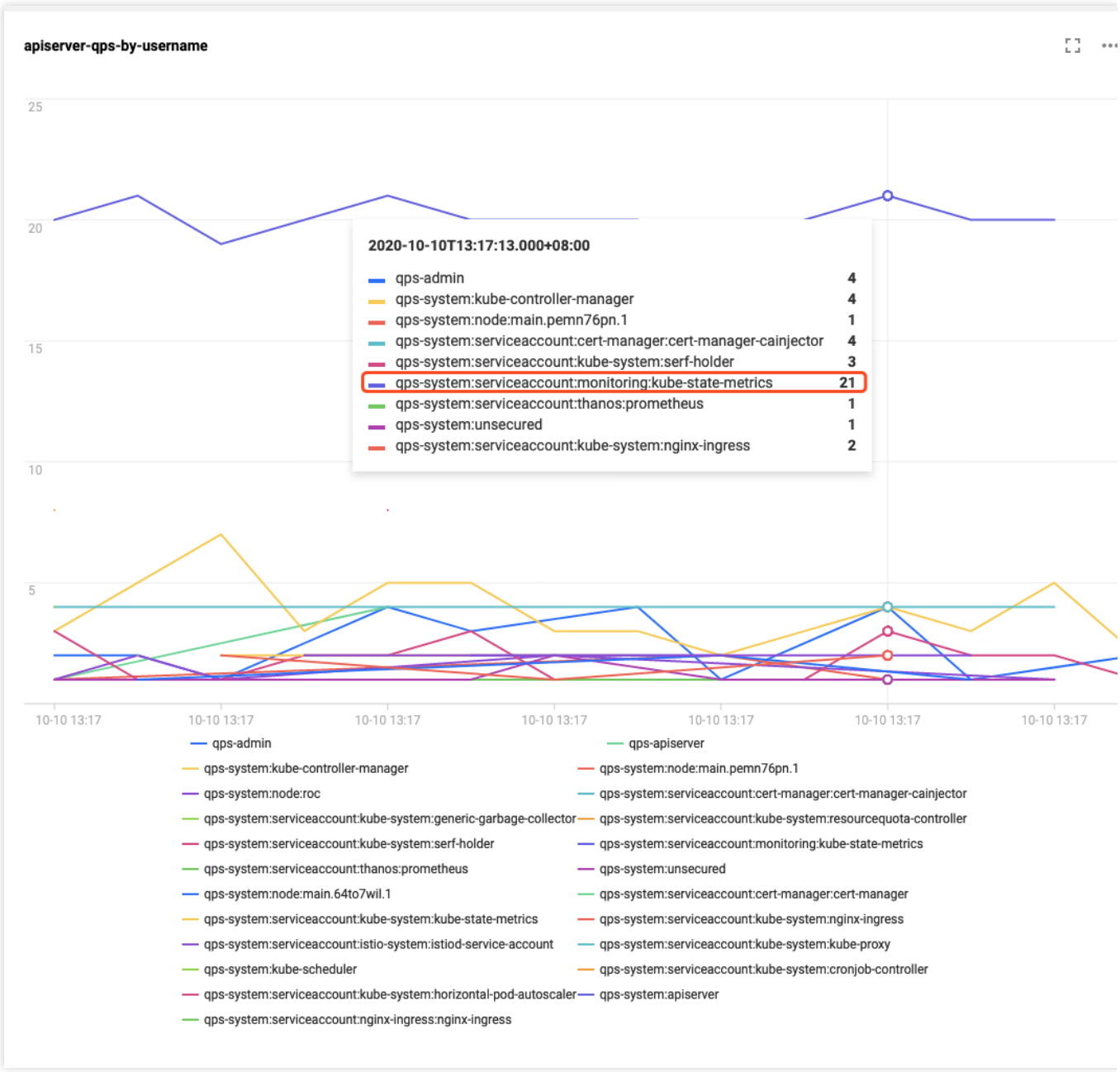
```
I1009 13:13:09.760767       1 request.go:538] Throttling request took 1.393921018s,
E1009 13:13:09.766106       1 reflector.go:156] pkg/mod/k8s.io/client-go@v0.0.0-201
```

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,
```

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 日志分析简介。

为 TKE Ingress 证书续期

最近更新时间：2022-05-31 11:53:12

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**证书管理**。
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

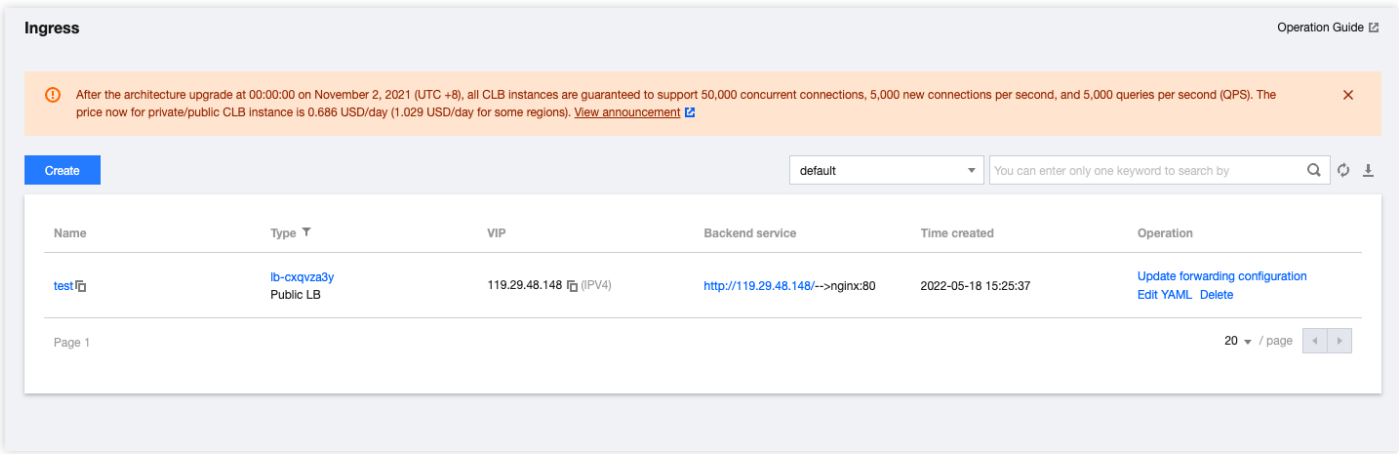
1. 登录 SSL 证书控制台，选择旧证书右侧的**关联资源**即可查看引用此证书的负载均衡器。
2. 点击负载均衡器的 ID 跳转到**负载均衡**详情页面。如果是 TKE Ingress 的负载均衡器，在标签栏会出现 `tke-clusterId` 和 `tke-lb-ingress-uuid` 的标签，分别表示集群 ID 和 Ingress 资源的 UID。
3. 在负载均衡器的“基本信息”页面，点击标签行右侧的编辑按钮，即可进入“编辑标签”页面。
4. 使用 Kubectl 可以查询集群 ID 对应集群的 Ingress，过滤 uid 为 `tke-lb-ingress.uuid` 对应值的 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
```

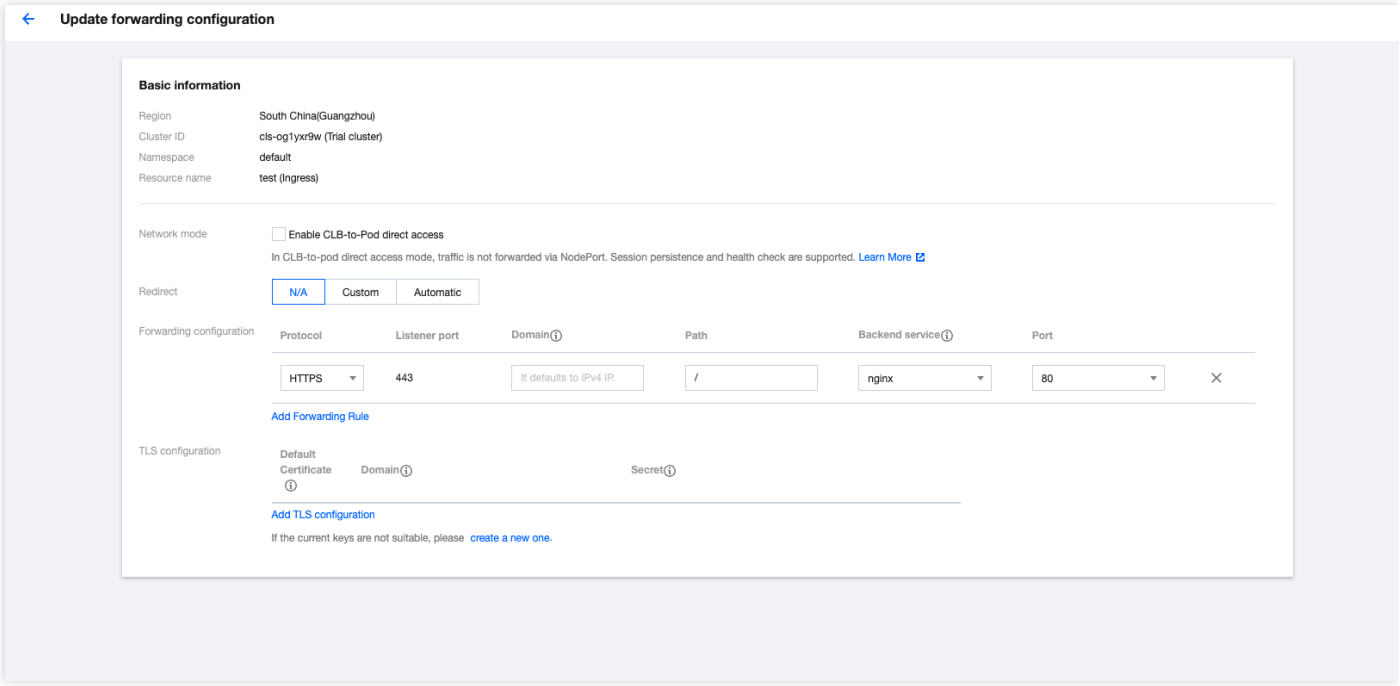
由查询结果可知，该集群中 `api-prod/gateway` 引用了此证书，因此需要更新此 Ingress。

更新 Ingress

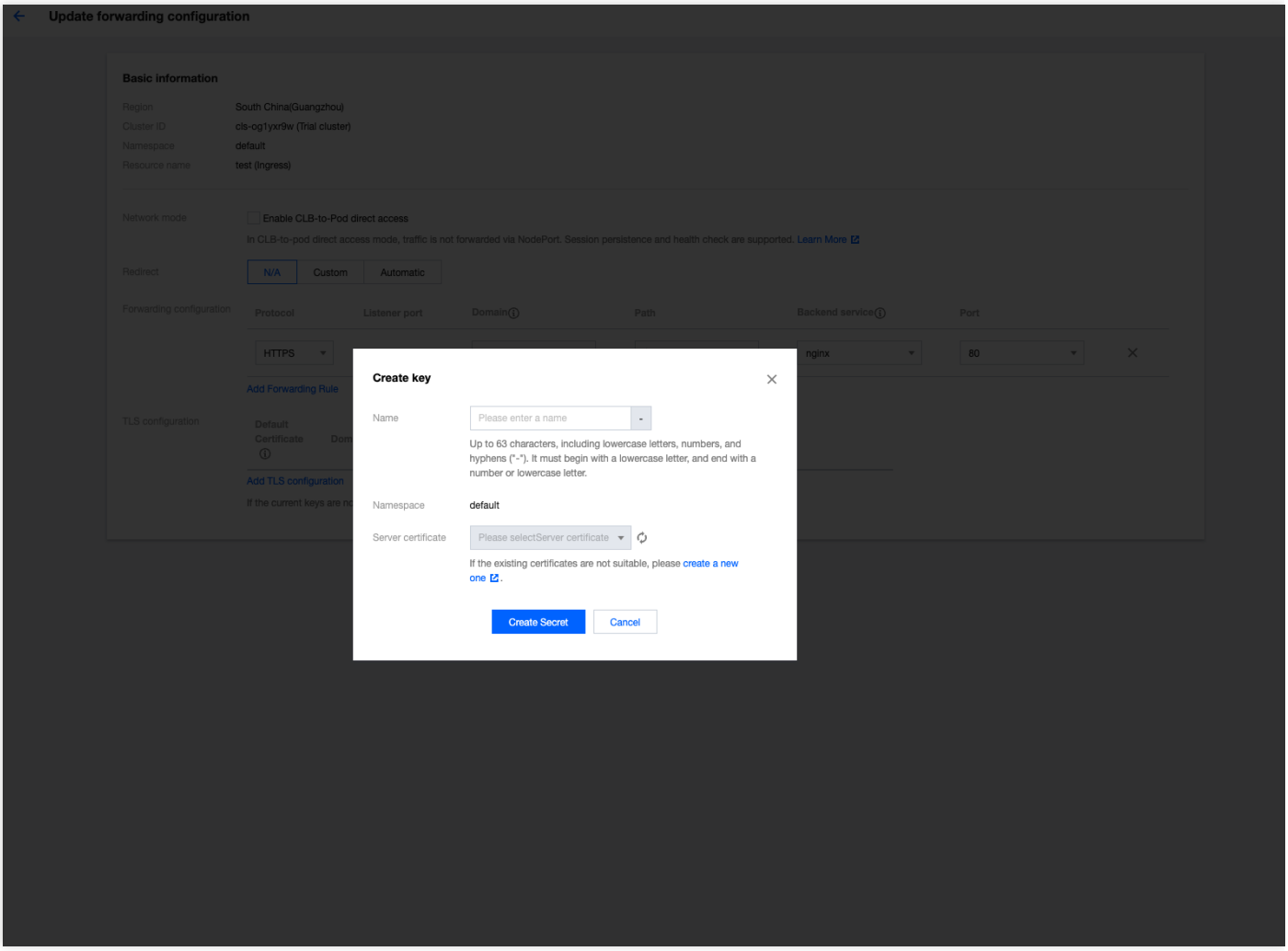
1. 在 容器服务控制台 找到 引用旧证书的 Ingress 中对应的 Ingress 资源，单击**更新转发配置**。如下图所示：



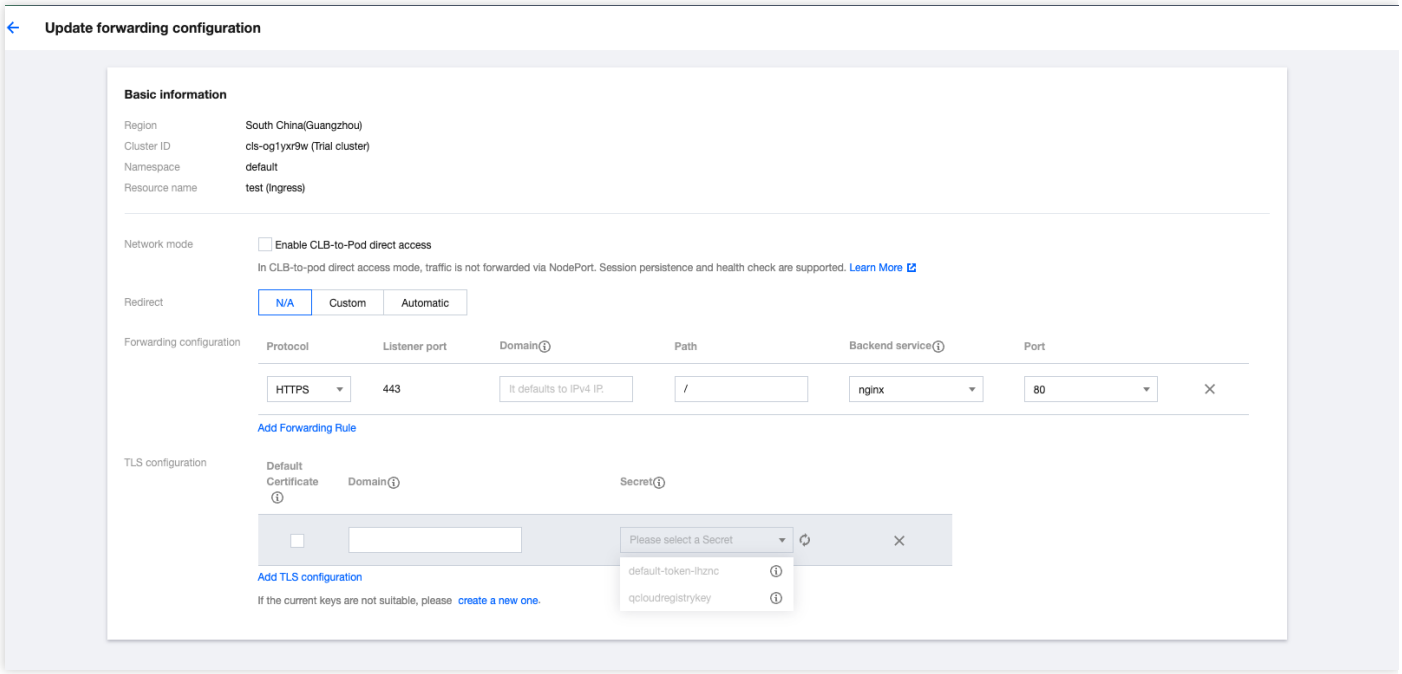
2. 在“更新转发配置”页面，为新证书**新建密钥**。如下图所示：



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：2023-05-23 17:55:11

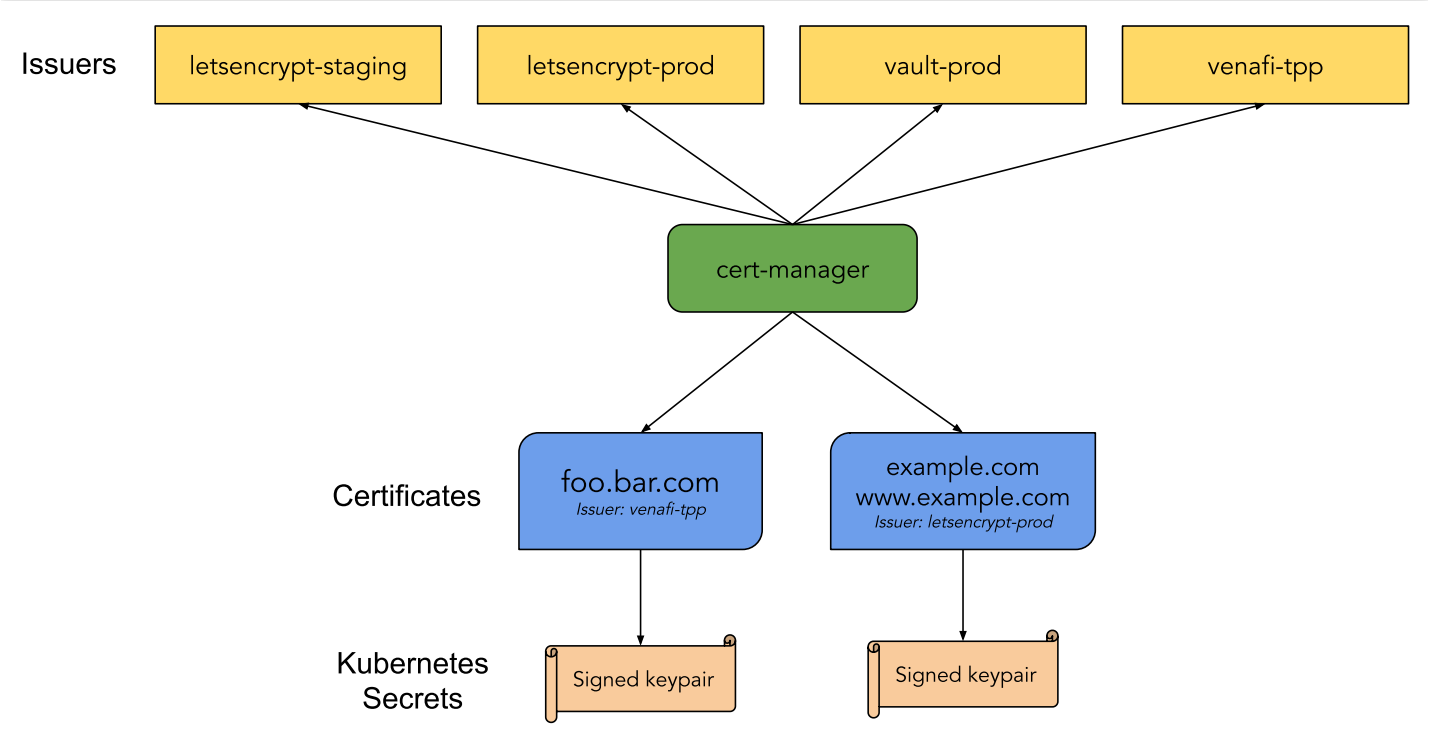
概述

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
- DNS-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。

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。

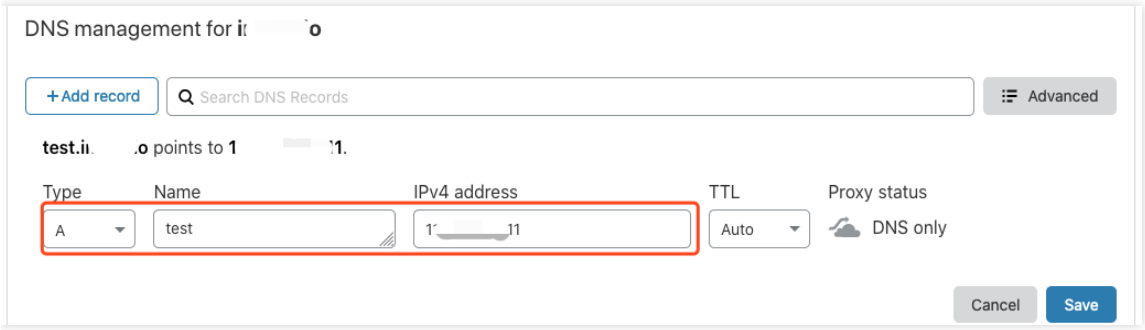
操作步骤

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。也可以执行以下命令，使用同步到国内 CCR 的镜像一键安装：

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：

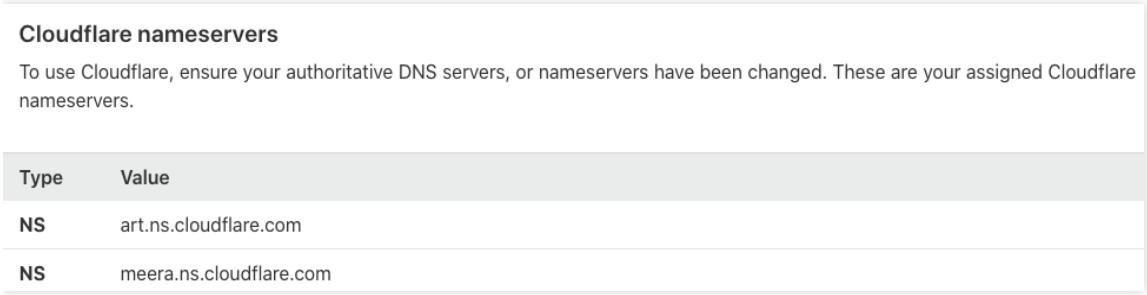
```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
name: test-mydomain-com
namespace: prod
spec:
dnsNames:
- test.mydomain.com # 要签发证书的域名
issuerRef:
kind: Issuer
name: letsencrypt-http01 # 引用 Issuer，指示采用 http01 方式进行校验
secretName: test-mydomain-com-tls # 最终签发出来的证书会保存在这个 Secret 里面
```

DNS-01 校验方式签发证书

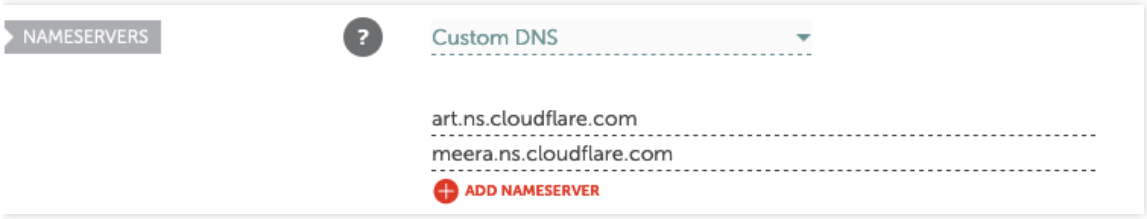
若使用 DNS-01 的校验方式，则需要选择 DNS 提供商。cert-manager 内置 DNS 提供商的支持，详细列表和用法请参见 Supported DNS01 providers。若需要使用列表外的 DNS 提供商，可参考以下两种方案：

- 方案1：设置 Custom Nameserver
- 方案2：使用 Webhook

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：



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。

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：2021-12-03 16:12:26

概述

如果您的域名使用腾讯云 DNSPod 管理，并期望在 Kubernetes 上为域名自动签发免费证书，可以使用 cert-manager 来实现。

cert-manager 支持许多 DNS provider，但不支持国内的 DNSPod，不过 cert-manager 提供了 Webhook 机制来扩展 provider，社区也有 DNSPod 的 provider 实现。本文将介绍如何结合 cert-manager 与 cert-manager-webhook-dnspod 来实现为 DNSPod 上的域名自动签发免费证书。

基础知识

推荐先阅读 使用 cert-manager 签发免费证书 。

操作步骤

1. 创建 DNSPod 密钥

登录 DNSPod 控制台，在 密钥管理 中创建密钥，复制自动生成的 `ID` 和 `Token` 并保存。

2. 安装 cert-manager

安装 cert-manager，详情可参见 使用 cert-manager 签发免费证书 。

3. 安装 cert-manager-webhook-dnspod

使用 HELM 来安装 cert-manager-webhook-dnspod，需准备 HELM 配置文件。
`dnspod-webhook-values.yaml` 示例如下：

```
groupName: example.your.domain # 写一个标识 group 的名称，可以任意写

secrets: # 将前面生成的 id 和 token 粘贴到下面
apiID: "<id>"
apiToken: "<token>"
```

```
clusterIssuer:
enabled: true # 自动创建出一个 ClusterIssuer
email: your@email.com # 填写你的邮箱地址
```

完整配置请参见 values.yaml。

使用 HELM 进行安装：

```
git clone --depth 1 https://github.com/qqshfox/cert-manager-webhook-dnspod.git
helm upgrade --install -n cert-manager -f dnspod-webhook-values.yaml cert-manage
r-webhook-dnspod ./cert-manager-webhook-dnspod/deploy/cert-manager-webhook-dnspo
d
```

4. 创建证书

使用如下所示 YAML 文件创建 `Certificate` 对象来签发免费证书：

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
name: example-com-crt
namespace: istio-system
spec:
secretName: example-com-crt-secret # 证书保存在这个 secret 中
issuerRef:
name: cert-manager-webhook-dnspod-cluster-issuer # 这里使用自动生成出来的 ClusterIss
uer
kind: ClusterIssuer
group: cert-manager.io
dnsNames: # 填入需要签发证书的域名列表，确保域名是使用 dnspod 管理的
- example.com
- test.example.com
```

等待状态变成 Ready 表示签发成功：

```
$ kubectl -n istio-system get certificates.cert-manager.io
NAME READY SECRET AGE
example-com-crt True example-com-crt-secret 25d
```

若签发失败可通过 describe 查看原因：

```
kubectl -n istio-system describe certificates.cert-manager.io example-com-crt
```

5. 使用证书

证书签发成功后会保存到指定的 Secret 中，可参考以下使用示例：

- 在 Ingress 中使用
- 在 Istio 的 ingressgateway 中使用

```
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-com-crt-secret # 引用证书 secret
```

使用 TKE NPDPlus 插件增强节点的故障自愈能力

最近更新时间：2023-05-23 10:37:12

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并且采取相应的恢复策略，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 Promethues 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

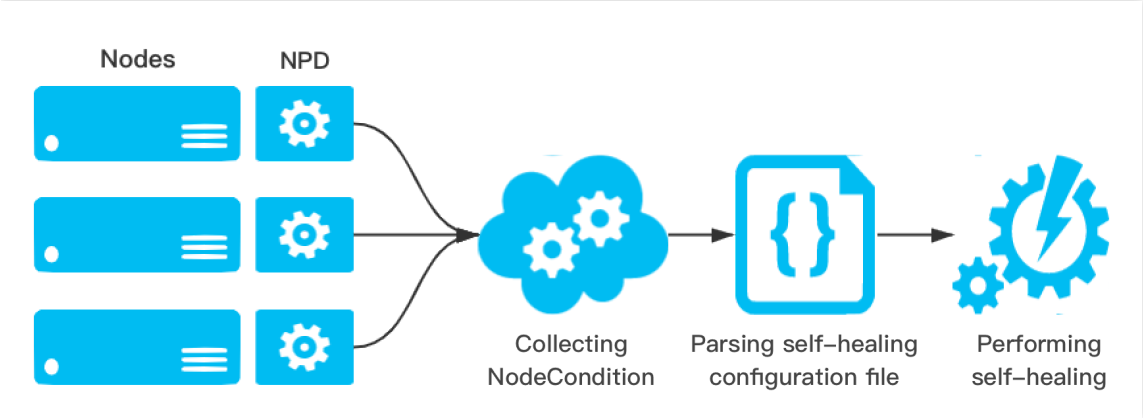
络状态，并判断节点是否能够在不依赖于 Kubernetes master 组件通信的情况下，与其他节点相互通信。TKE NPDPlus 组件使用详情请参见 NodeProblemDetectorPlus 使用方法。

节点自愈

采集节点的健康状态是为了能够在业务 Pod 不可用之前提前发现节点异常，从而运维或开发人员可以对 Docker、Kubelet 或节点进行修复。在 NPDPlus 中，为了减轻运维人员的负担，提供了根据采集到的节点状态从而进行不同自愈动作的能力。集群管理员可以根据节点不同的状态配置相应的自愈能力，如重启 Docker、重启 Kubelet 或重启 CVM 节点等。同时为了防止集群中的节点雪崩，在执行自愈动作之前做了严格的限流，防止节点大规模重启。具体策略为：

- 在同一时刻只允许集群中的一个节点进行自愈行为，并且两个自愈行为之间至少间隔1分钟。
- 当有新节点添加到集群中时，会给节点2分钟的容忍时间，防止由于节点刚添加到集群的不稳定性导致错误自愈。
- 当节点触发重启 CVM 自愈动作后还处于异常状态时，在3小时之内此节点不再执行任何自愈动作。

NPDPlus 会将执行过的所有自愈动作记录在 Node 的 Event 中，方便集群管理员了解在 Node 上发生的事件。如下图所示：



使用 kubecm 管理多集群 kubeconfig

最近更新时间：2023-02-23 18:34:01

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

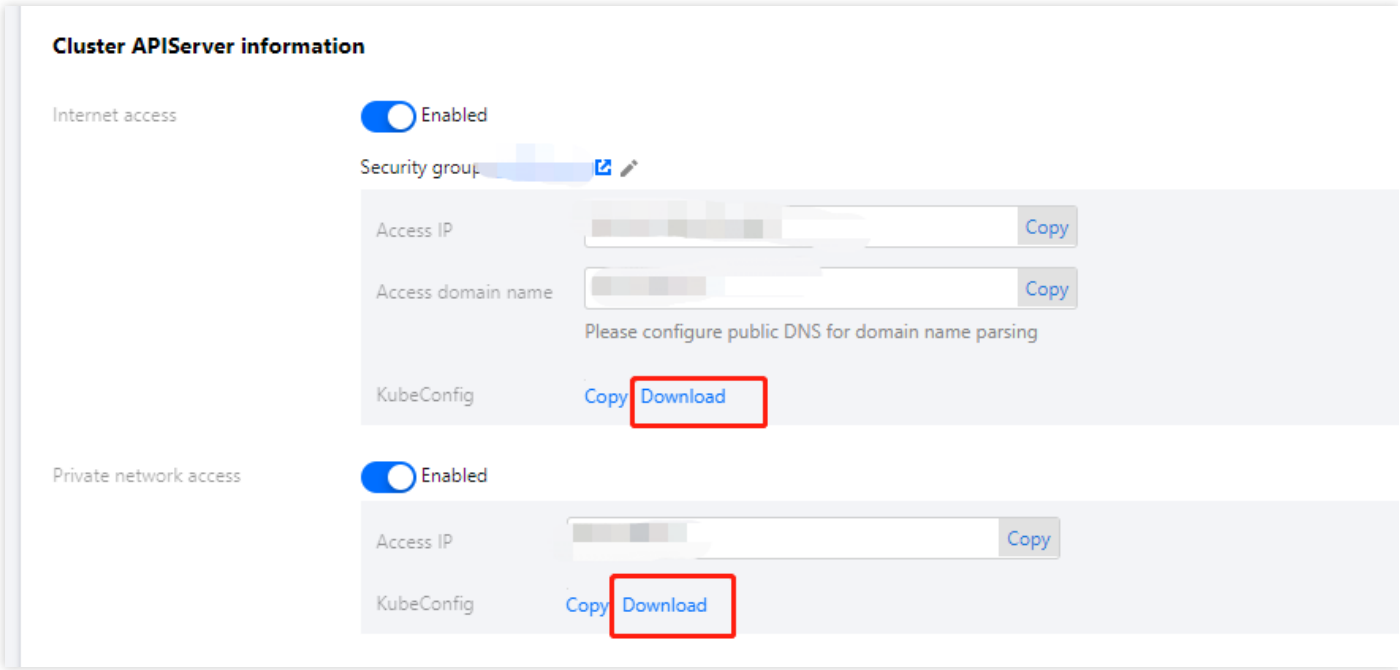
在管理多集群的设备上安装 Kubecm。

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

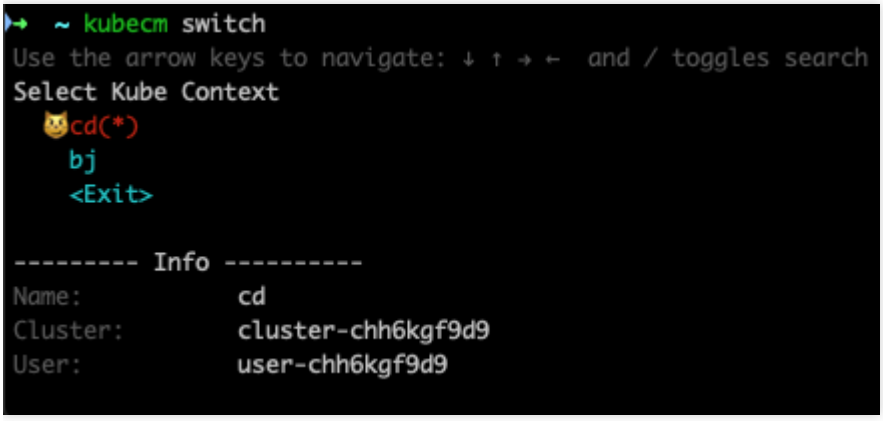
执行以下 `kubecm ls` 命令查看 kubeconfig 中的集群列表（星号标识的是当前操作的集群）。示例如下：

```
$ 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：2023-05-23 15:17:04

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。

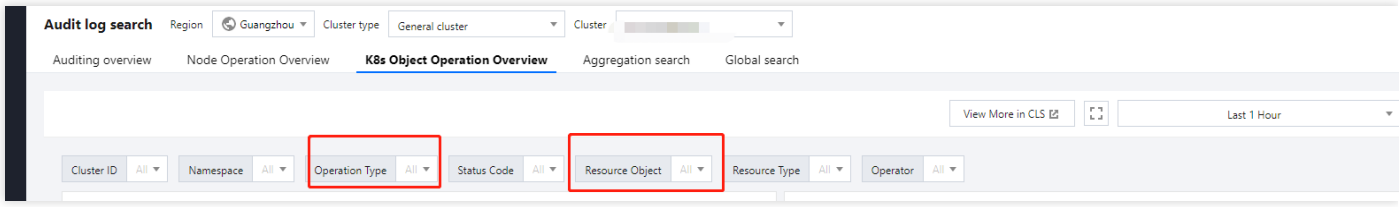
前提条件

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：



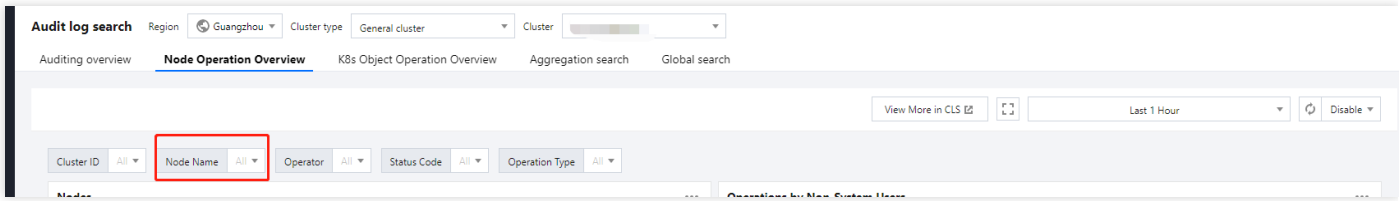
4. 查询结果如下图所示：

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。

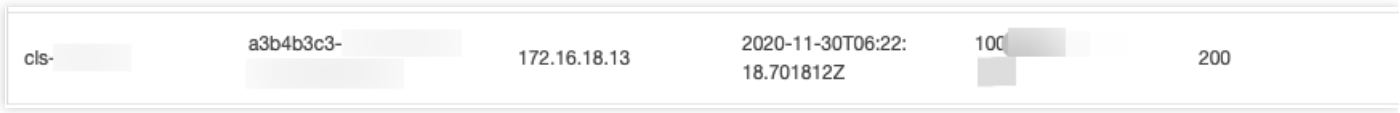
示例2：排查节点被封锁问题

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：



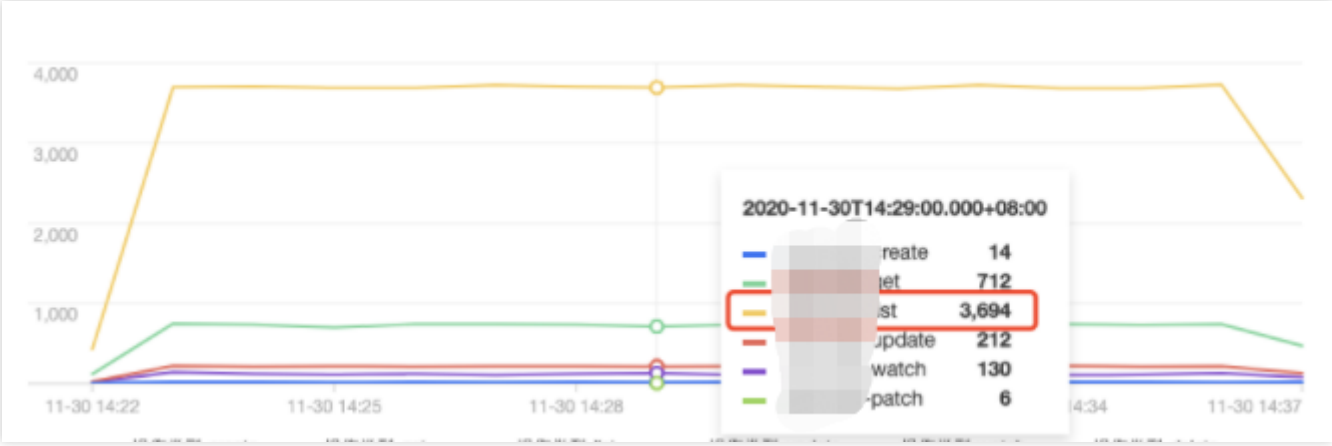
由图可见，`10001****7138` 账号在 `2020-11-30T06:22:18` 时对 `172.16.18.13` 节点进行了封锁操作。

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**：

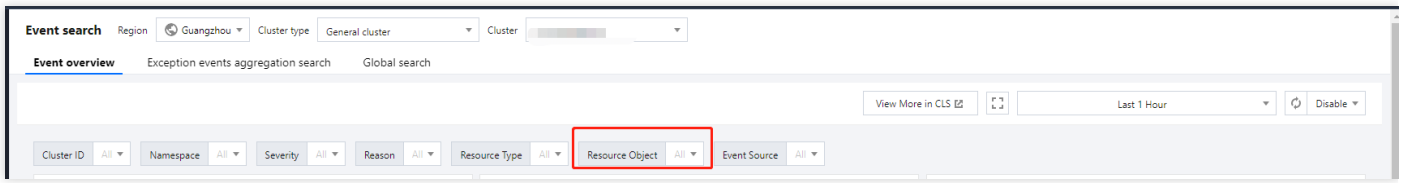
由图可见，用户 `tke-kube-state-metrics` 的访问量远高于其他用户，并且在 操作类型分布趋势 图中可以看出大多数为 list 操作，在 状态码分布趋势 图中可以看出，状态码大多数为403。结合业务日志可知，由于 RBAC 鉴权问题导致 `tke-kube-state-metrics` 组件不停的请求 apiserver 重试，导致 apiserver 访问剧增。日志示例如下：

```
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。
5. 单击该事件，进一步查看异常事件趋势。

cls-ire2oyho	2020-11-25T14:20:29+0000	Warning	Node	172.16.18.13	EvictionThresholdMet	Attempting to reclaim ephemeral-storage	56
cls-ire2oyho	2020-11-25T14:15:28+0000	Warning	Node	172.16.18.13	EvictionThresholdMet	Attempting to reclaim ephemeral-storage	26
cls-ire2oyho	2020-11-25T14:14:57+0000	Warning	Node	172.16.18.13	EvictionThresholdMet	Attempting to reclaim ephemeral-storage	23
cls-ire2oyho	2020-11-25T14:14:47+0000	Warning	Node	172.16.18.13	EvictionThresholdMet	Attempting to reclaim ephemeral-storage	22
cls-ire2oyho	2020-11-25T14:14:37+0000	Warning	Node	172.16.18.13	EvictionThresholdMet	Attempting to reclaim ephemeral-storage	21
cls-ire2oyho	2020-11-25T14:14:27+0000	Warning	Node	172.16.18.13	EvictionThresholdMet	Attempting to reclaim ephemeral-storage	20

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-autoscler）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。

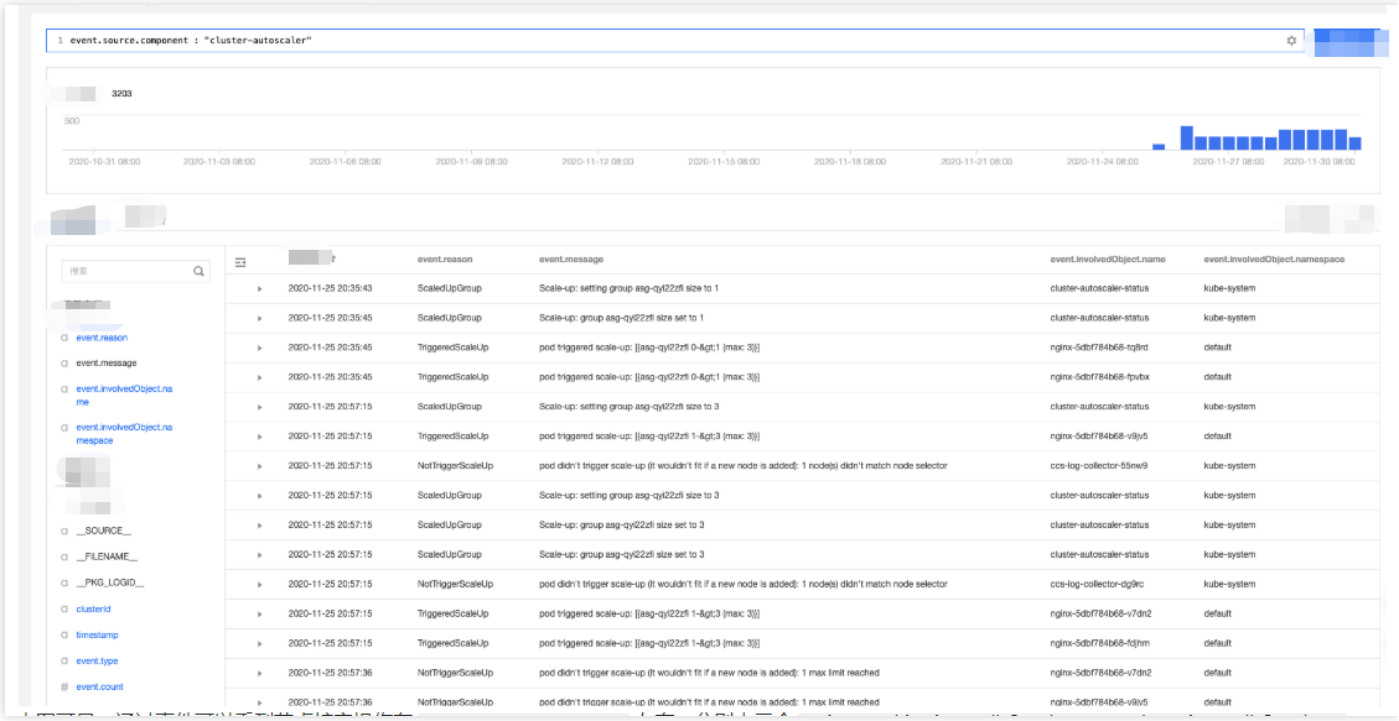
2. 在左侧导航栏中，选择**日志管理 > 事件日志**，进入“事件检索”页面。

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：2022-06-10 19:32:53

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。

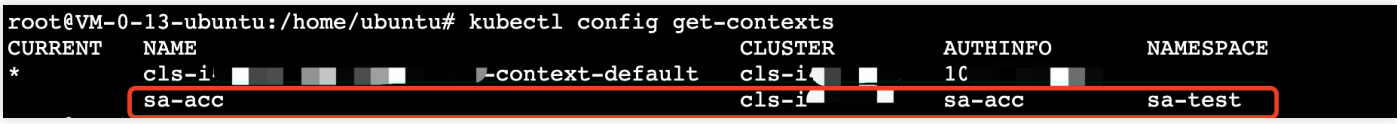
方式1：作用于单个命名空间的权限绑定

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：



3. 创建一个 Role 权限对象资源 sa-role.yaml 文件。示例如下：

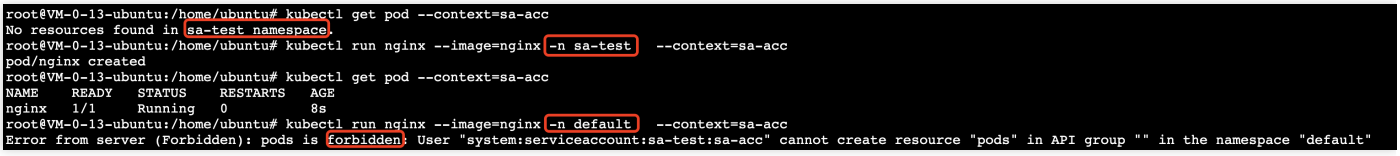
```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
namespace: sa-test # 指定 Namespace
name: sa-role-test
rules: # 设置权限规则
- apiGroups: ["", "extensions", "apps"]
```

```
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。



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 ${USERNAME}.csr -subj "/CN=${USERNAM
E}/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。

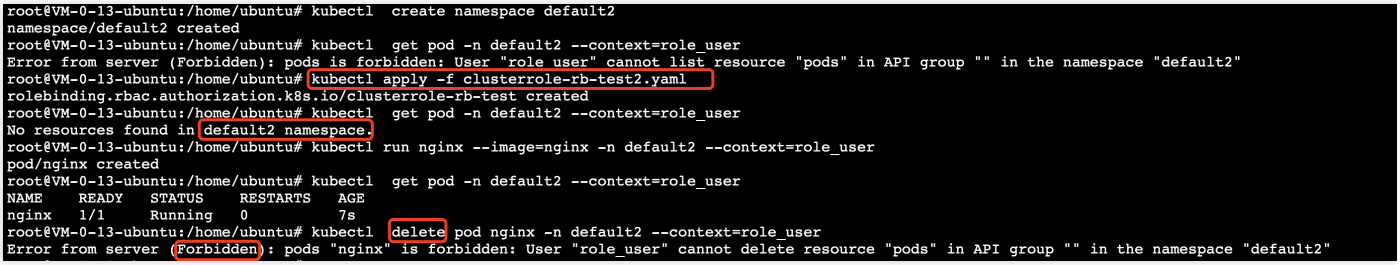
```
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod --context=role_user
No resources found in default namespace.
root@VM-0-13-ubuntu:/home/ubuntu# kubectl run nginx --image=nginx --context=role_user
pod/nginx created
root@VM-0-13-ubuntu:/home/ubuntu# kubectl  get pod --context=role_user
NAME    READY   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。



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

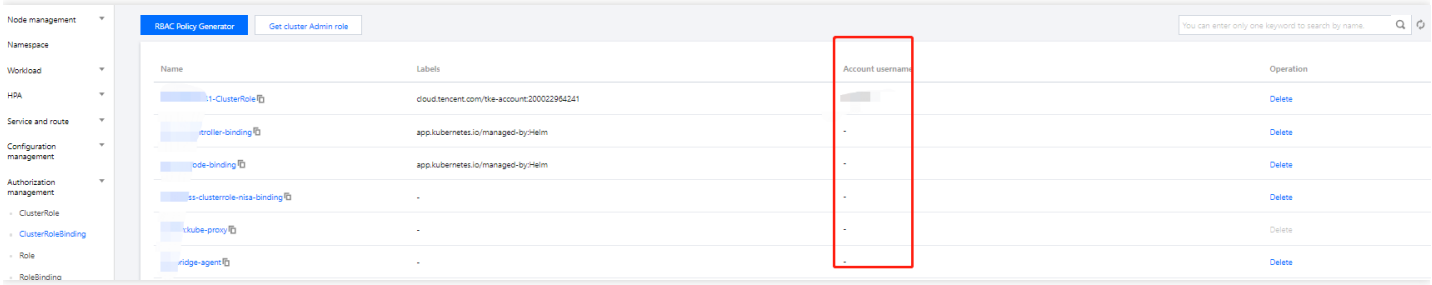
本文以集群管理员给一个用户自定义权限为例：权限包括预设的只读权限额外添加登录容器的权限。

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。




```yaml
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/v1beta1
kind: ClusterRole
metadata:
name: "700000xxxxxx-ClusterRole-ro" # ClusterRole 的名字
rules:
- apiGroups:
- ""
resources:
- pods
- pods/attach
- pods/exec # Pod 的登陆权限
- 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：2022-08-26 17:44:49

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 Kuberentes 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

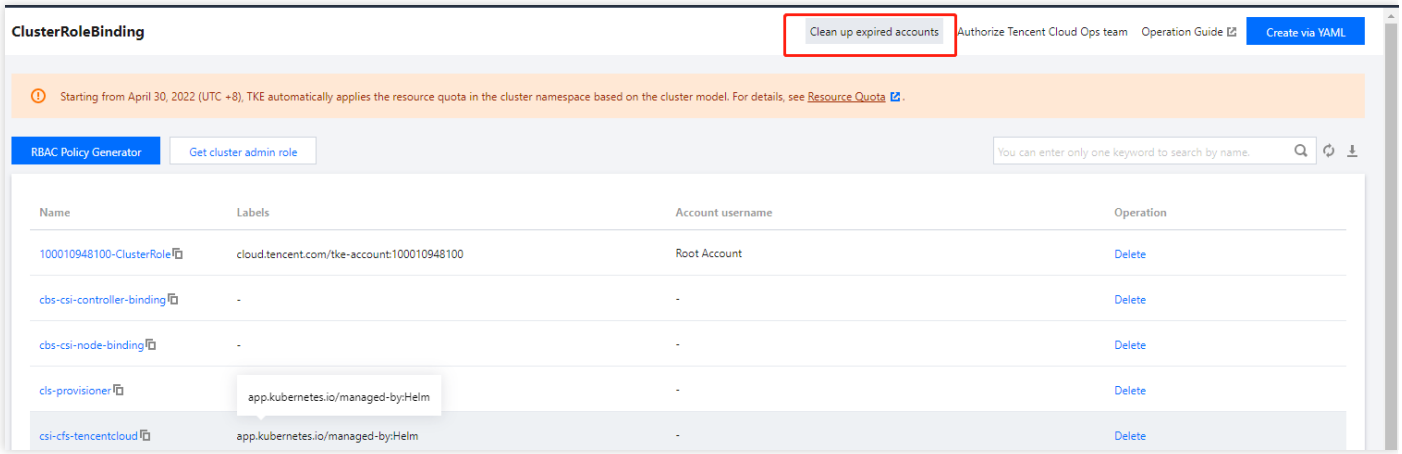
若您的集群中存在已注销的腾讯云账户，您可通过如下步骤查看：

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。

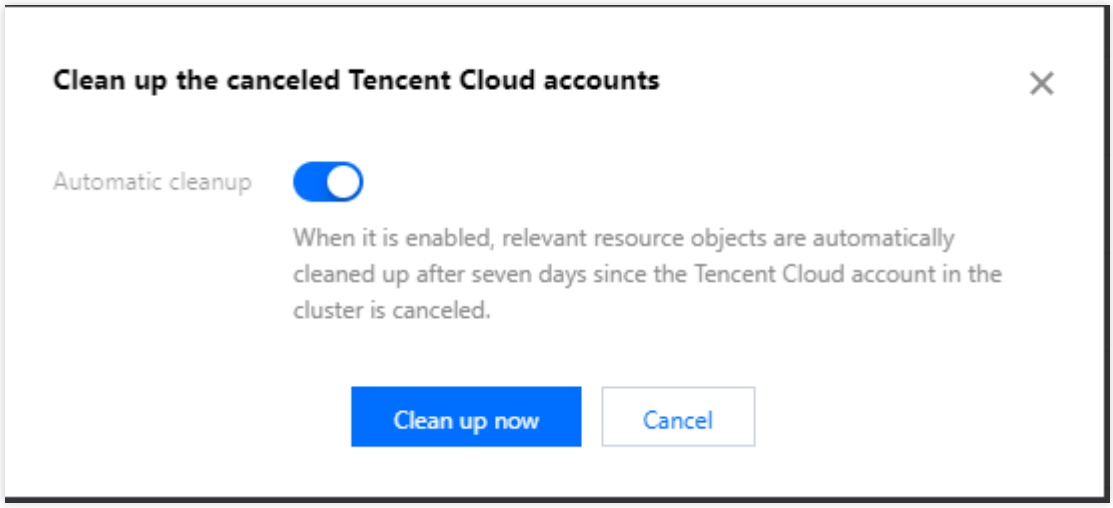
清理失效账户

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：2023-09-05 09:38:03

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

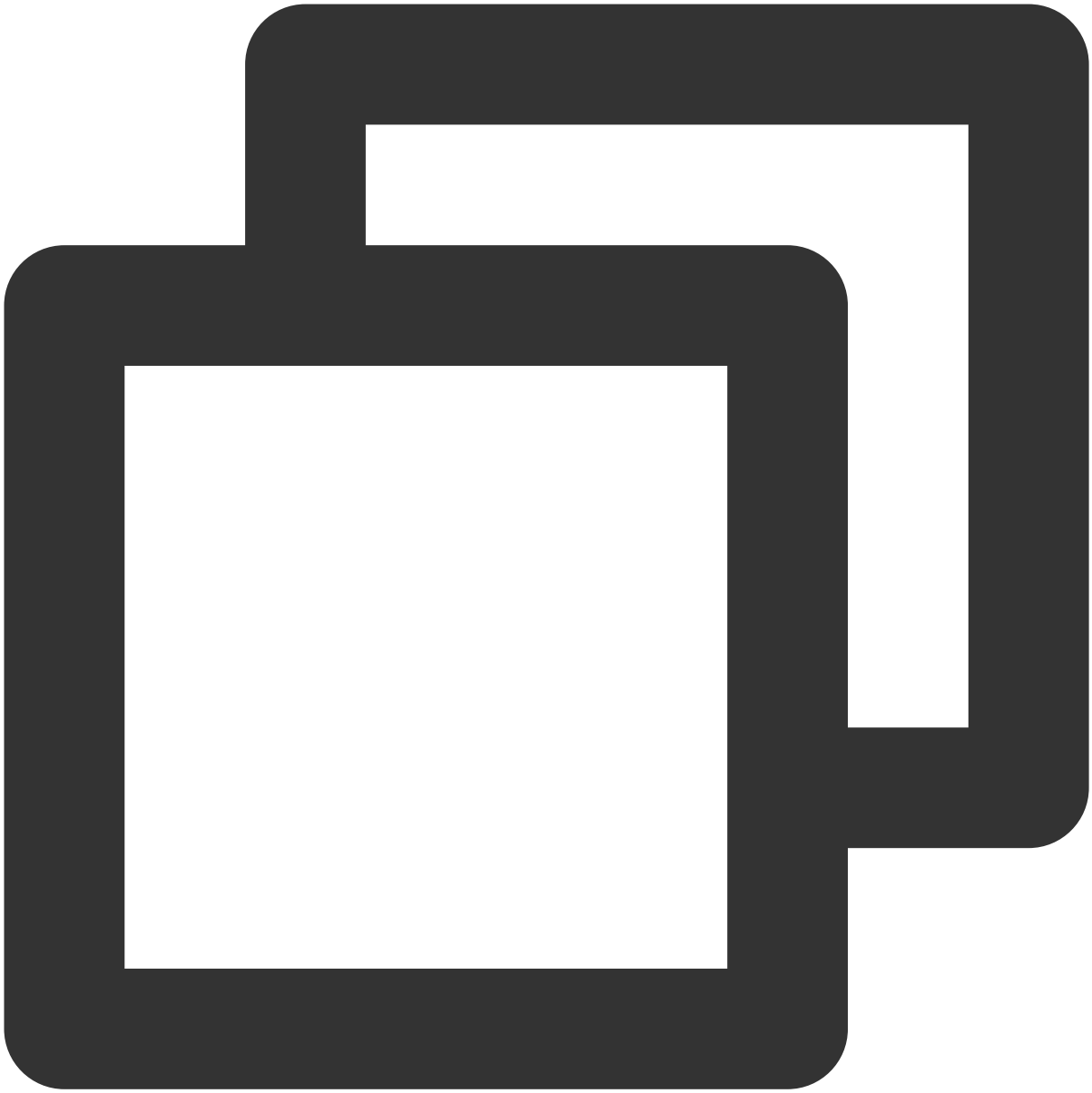
在首次使用 Terraform 之前，请前往 云 API 密钥页面 申请安全凭证 SecretId 和 SecretKey。若已有可使用的安全凭证，则跳过该步骤。

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：



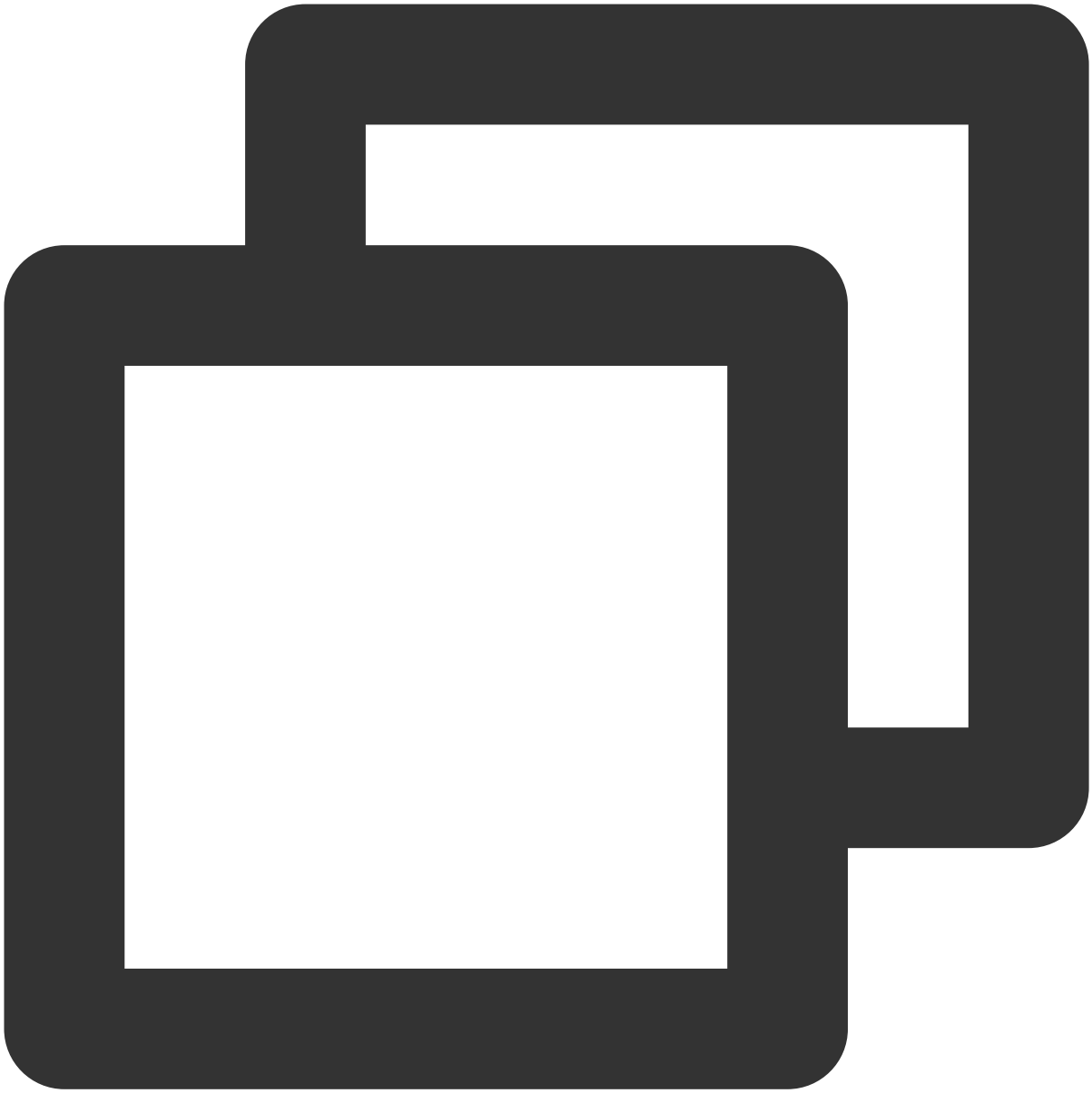
```
export TENCENTCLOUD_SECRET_ID="xxx"                    # 替换为账号访问密钥的SecretId
export TENCENTCLOUD_SECRET_KEY="xxx"                # 替换为账号访问密钥的SecretKey
```

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"
  secret_key = "xxx"
}
```

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

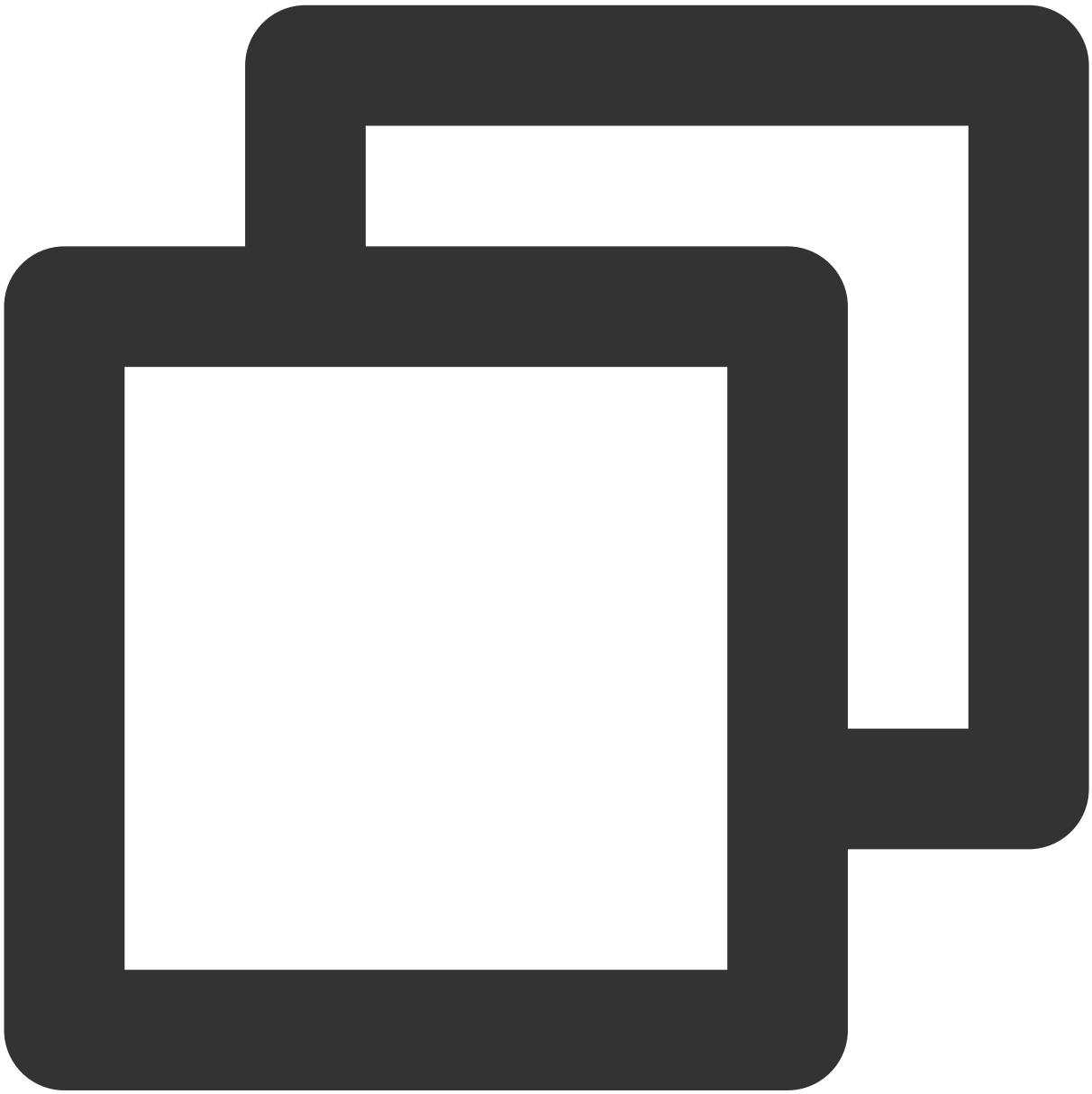
`main.tf` 文件描述的是以下 Terraform 配置：

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 {
```

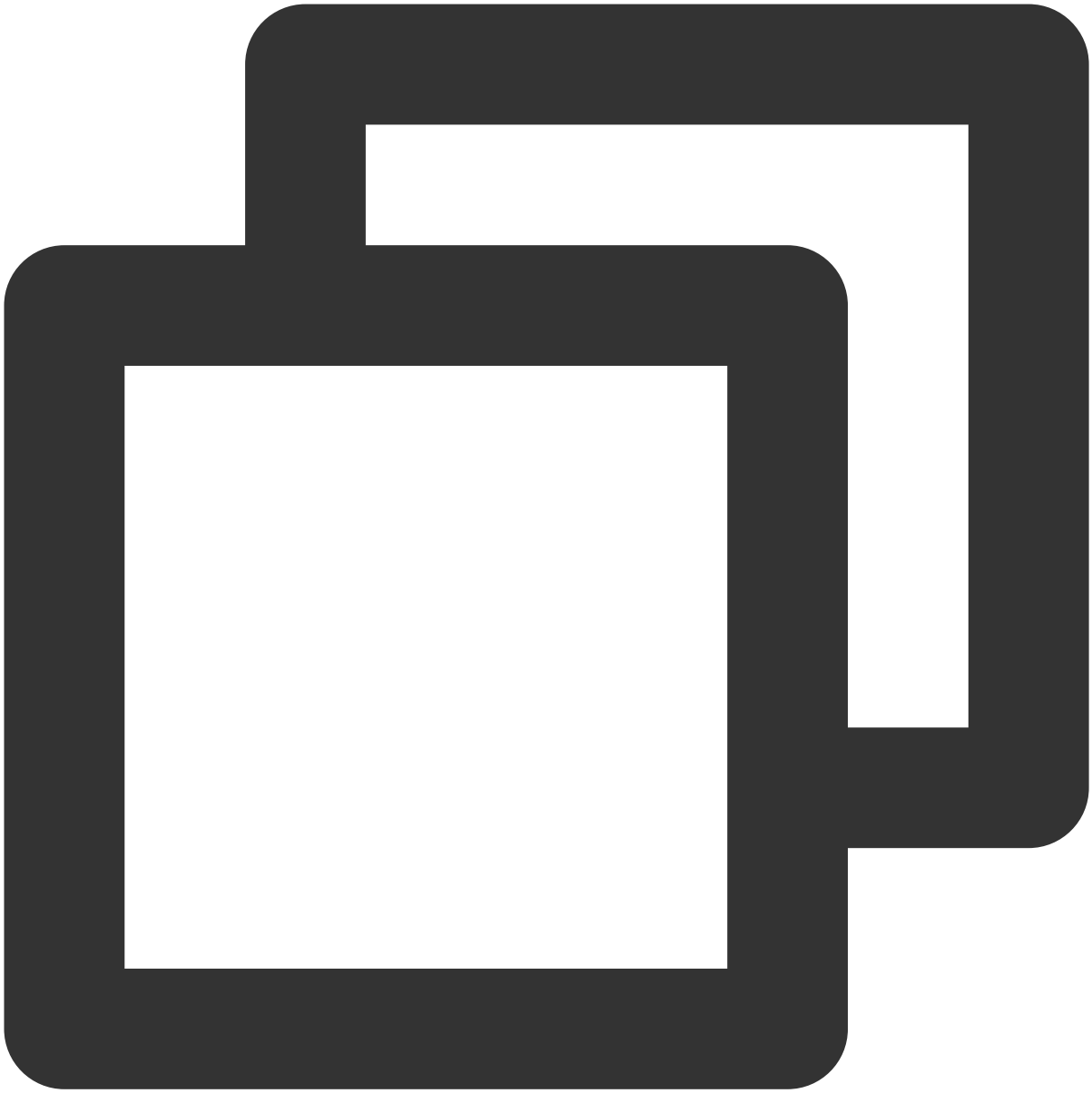
```
    tencentcloud = {
      source = "tencentcloudstack/tencentcloud"
    }
  }
}

# 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
    zone1 = "xxx"
    vpc_name = "xxx"                                                  # 设置VPC的名
    vpc_cidr_block = "xxx"                                # VPC的CIDR设置，如10.0.0.0/1
    subnet1_name = "xxx"                                              # 子网1的名字，如tke-t
    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tk
    network_type = "xxx"                                              # TKE托管集群的网络模式
    cluster_cidr = "xxx"                                              # 集群的容器网络，不能
    cluster_version = "xxx"                               # TKE集群的Kubernetes版本，如
}


# 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
    # 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
}

# 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
  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
#   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：

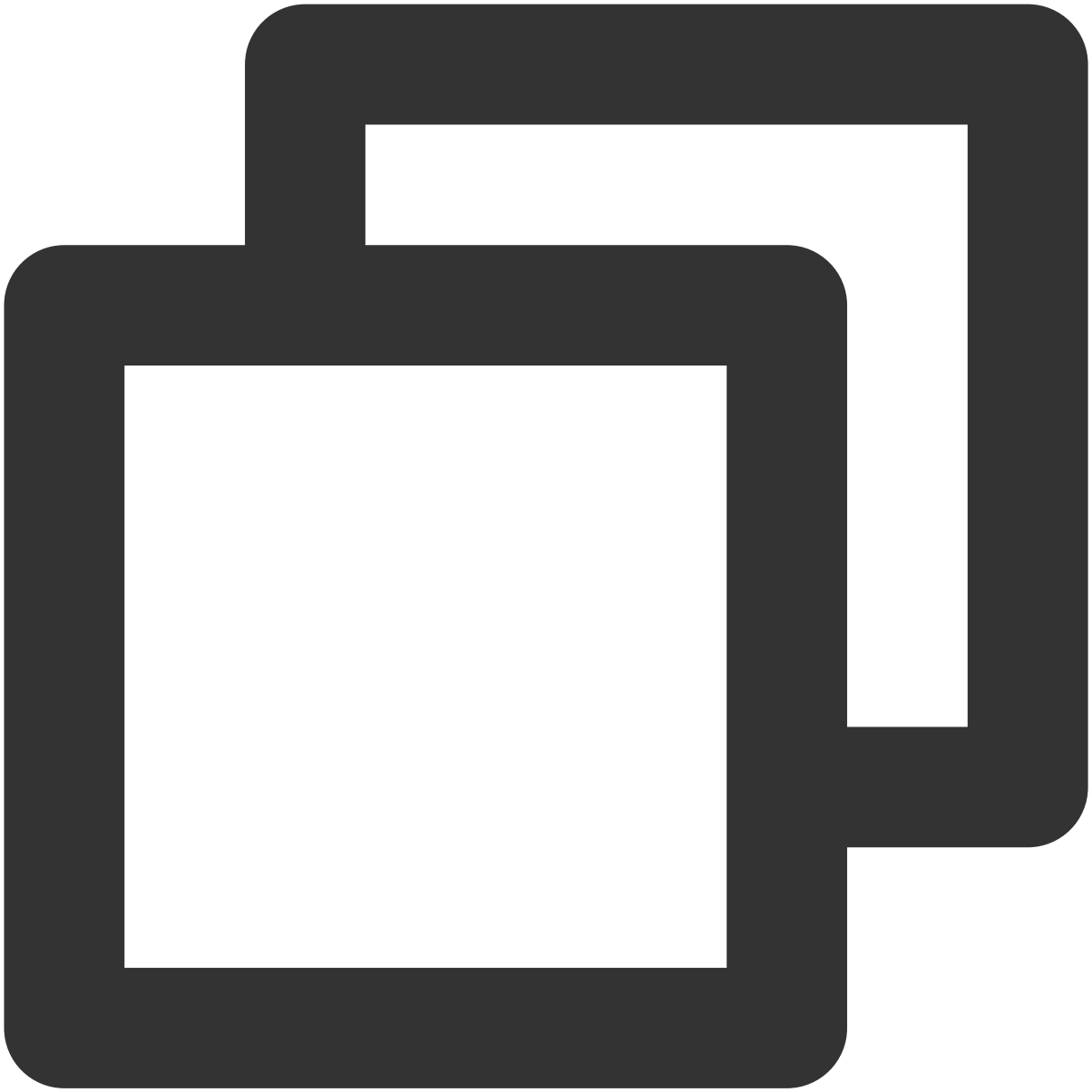


```
############################ 请按需在Terraform配置文件中添加声明配置，已在控制台完成授权的角色

# 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
}

# 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":{
```

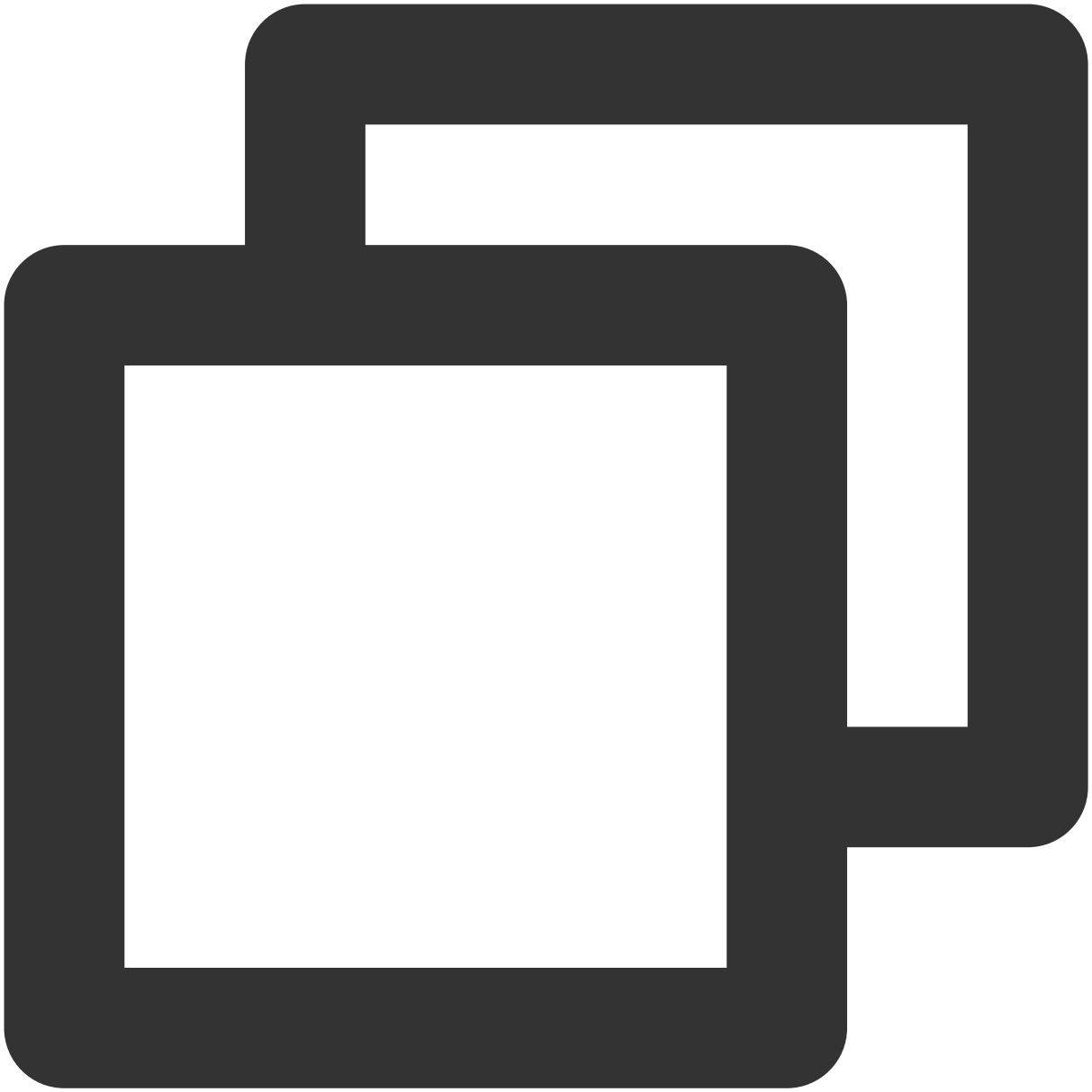
```
    "service":"ccs.qcloud.com"
  }
}
 ],
 "version":"2.0"
}
EOF
 description = "当前角色为 容器服务IPAMD支持 服务角色，该角色将在已关联策略的权限范围内访问您的
}

# 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
```

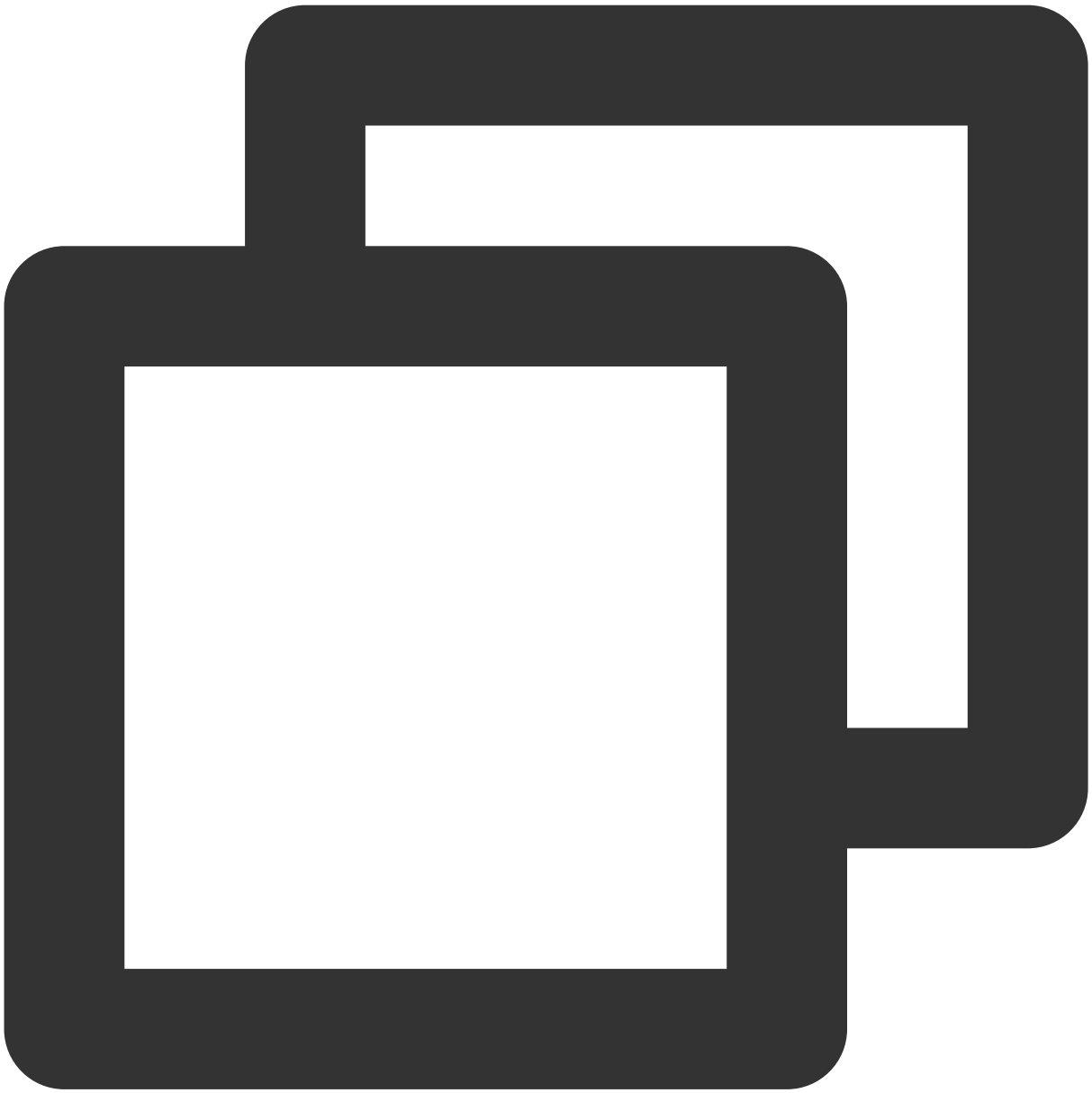
返回信息如下所示：



```
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.
```

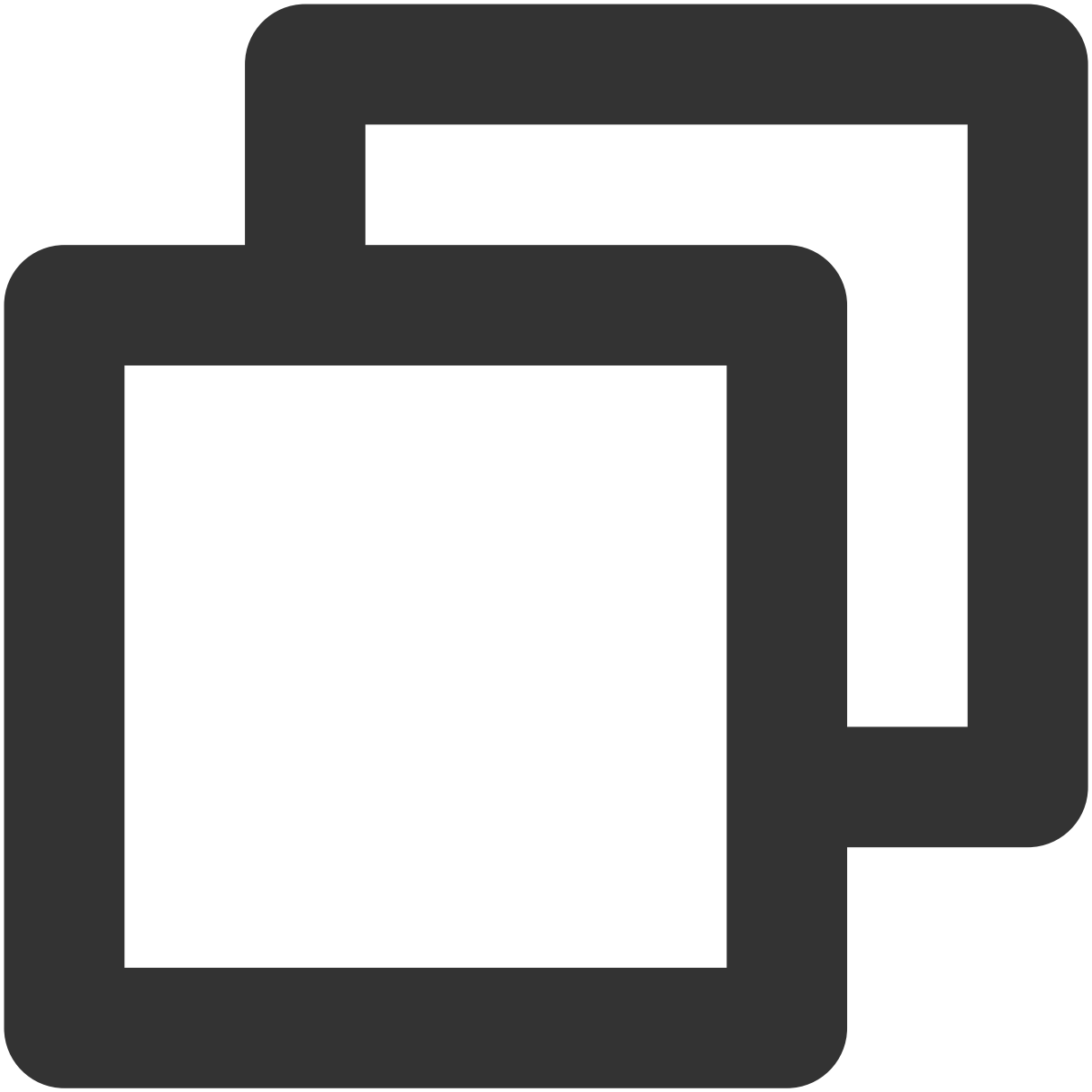
```
...
```

4. 执行以下命令，查看 Terraform 根据配置文件生成的资源规划。



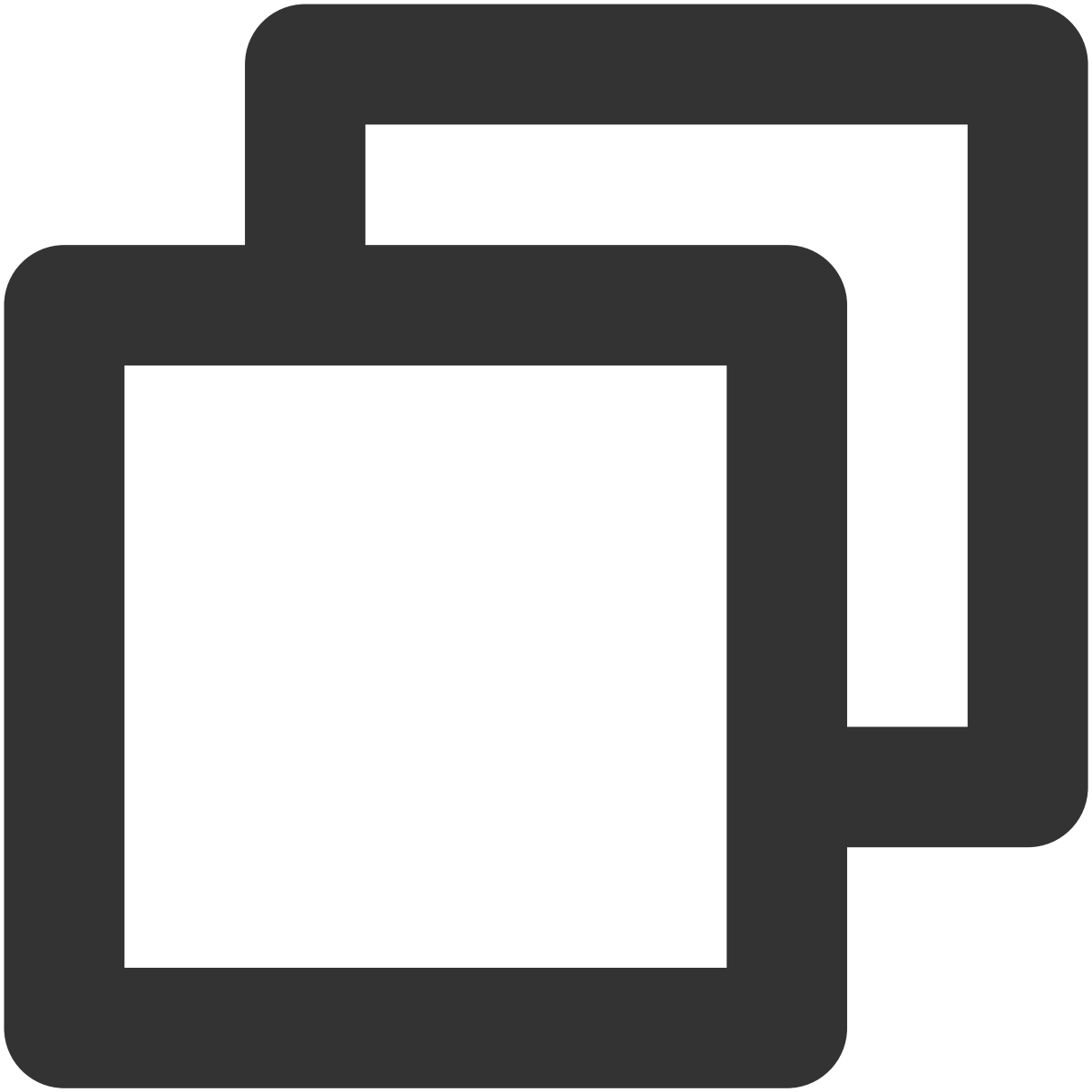
```
terraform plan
```

返回信息如下所示：



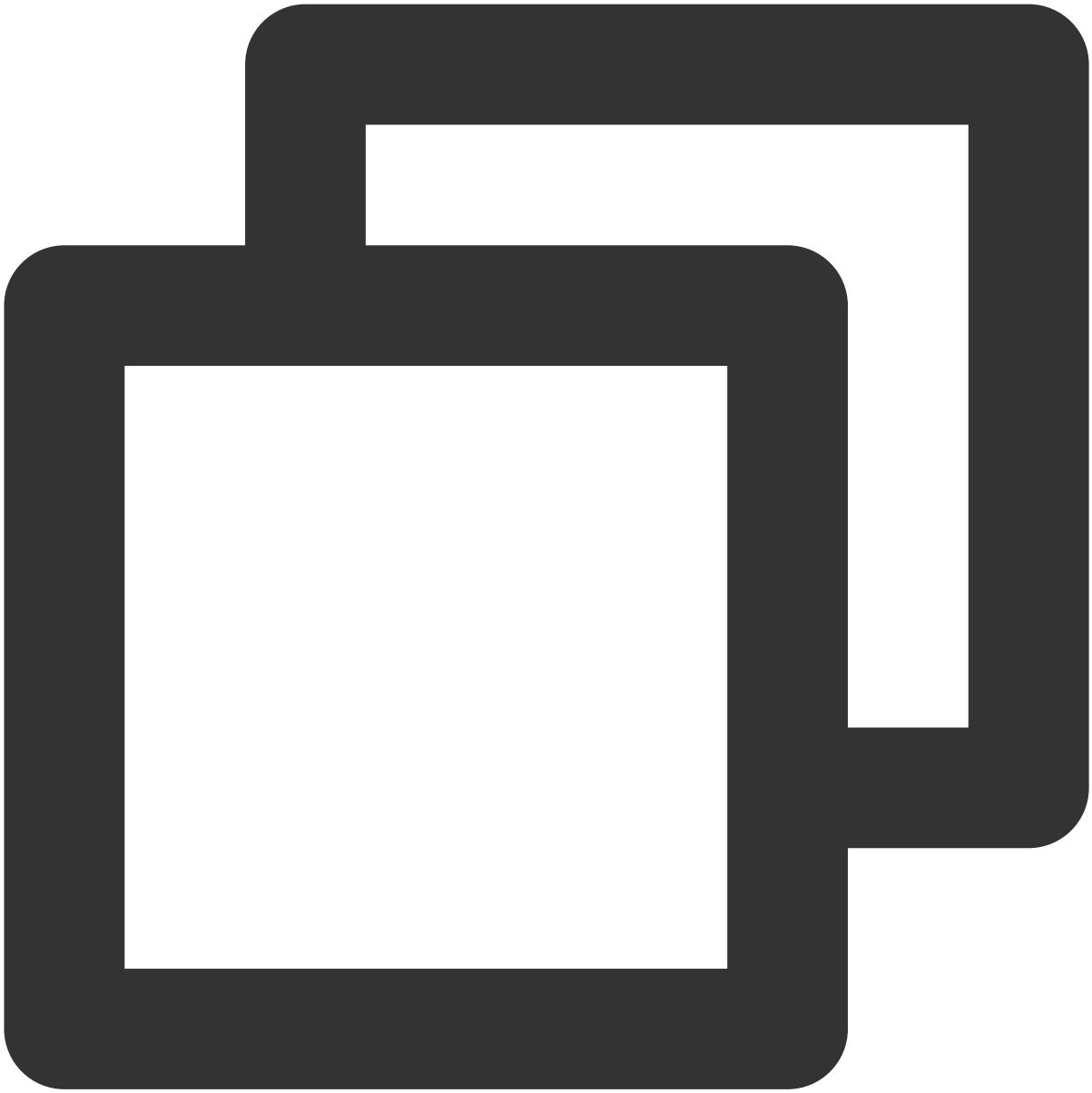
```
Terraform used the selected providers to generate the following execution plan. Res
  + create

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。



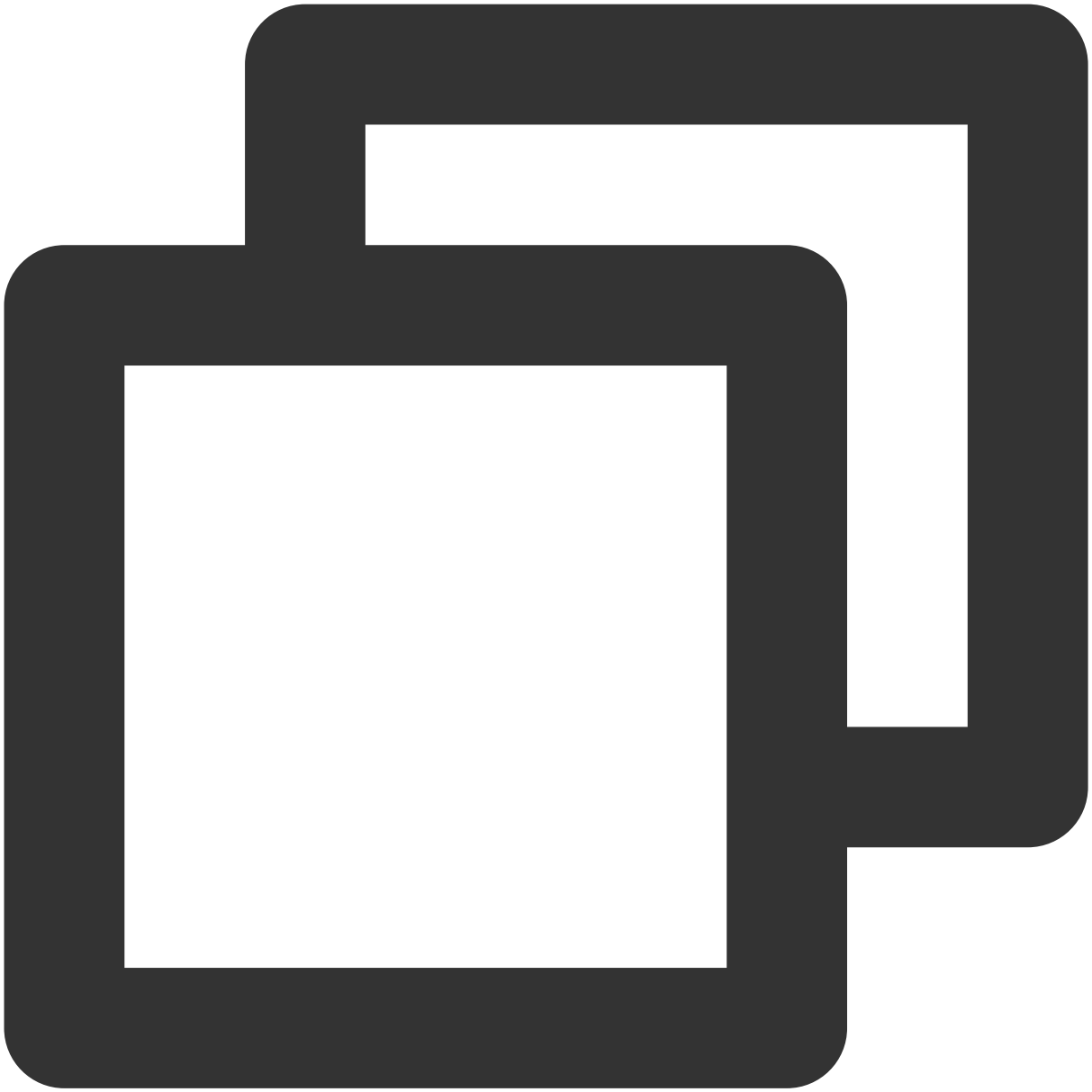
```
terraform apply
```

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：

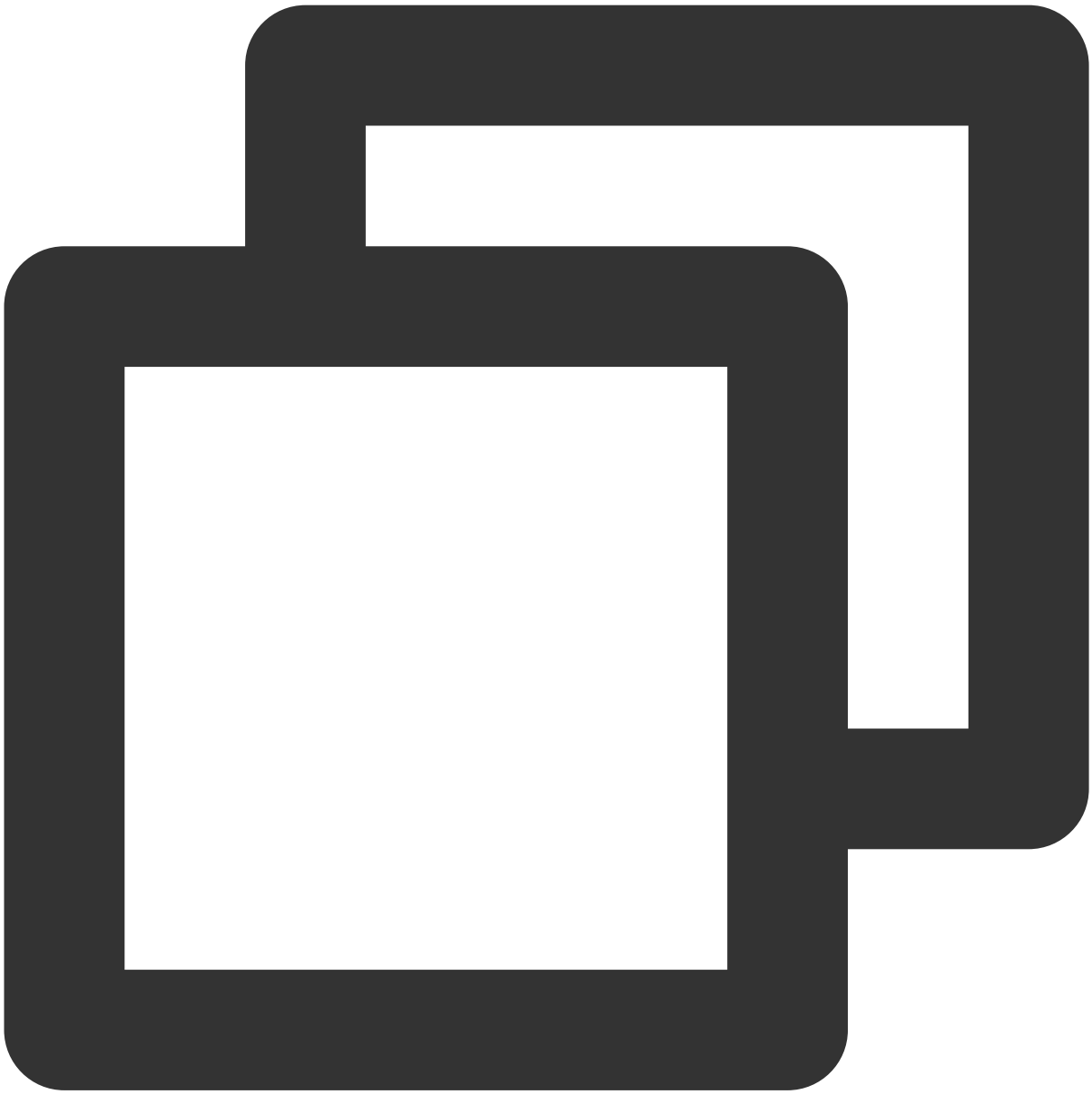


```
...
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

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

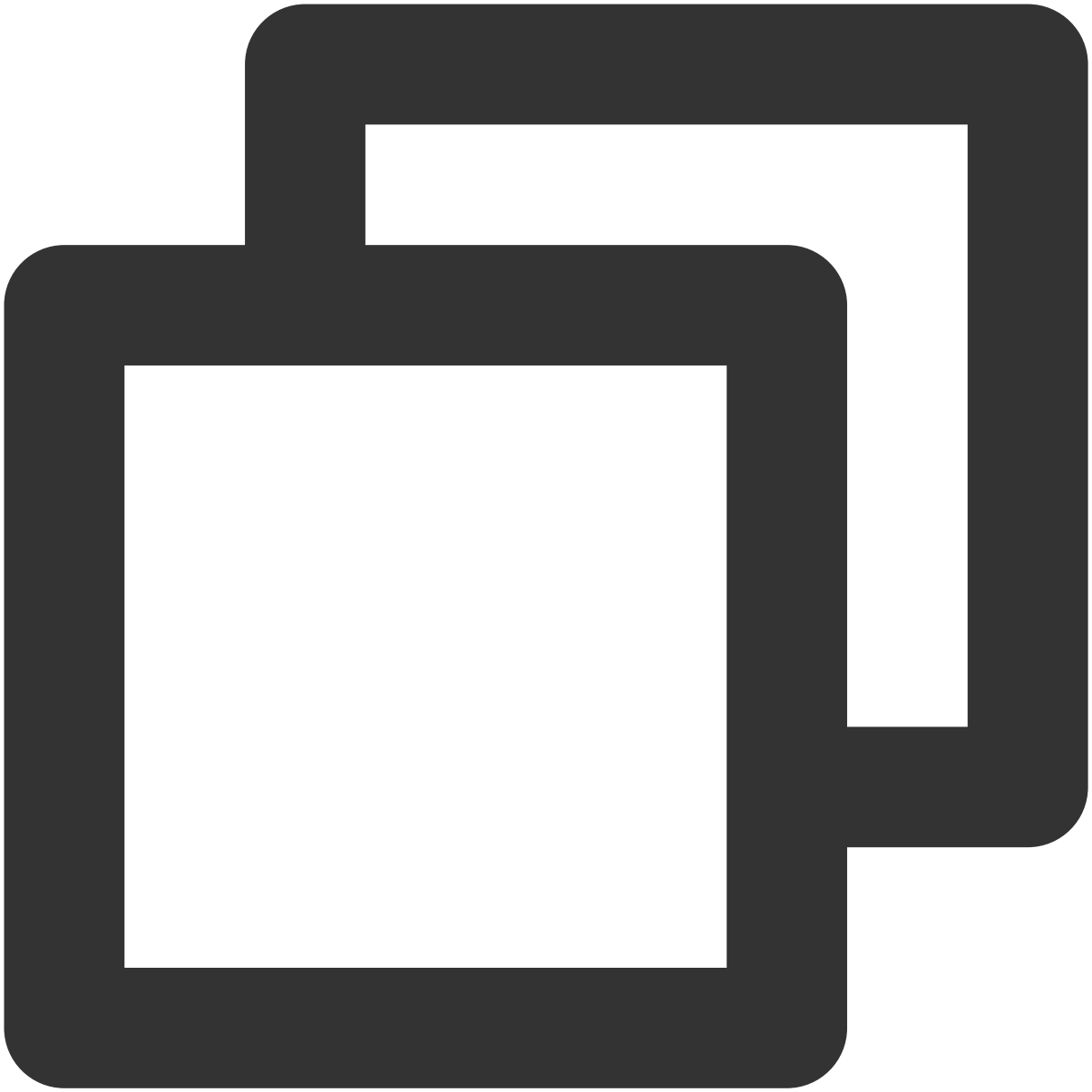
1. 创建一个工作目录，并在工作目录中创建名为 `nodepool.tf` 的 Terraform 配置文件。
`nodepool.tf` 文件内容如下：



```
# 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_clust
locals {
    node_pool_name = "xxx"                                    # 节点池名称，如tke-tf-demo-nc
    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://clou
    cvm_pass_word = "xxx"                                            # 节点池CVM机器登录密码
```

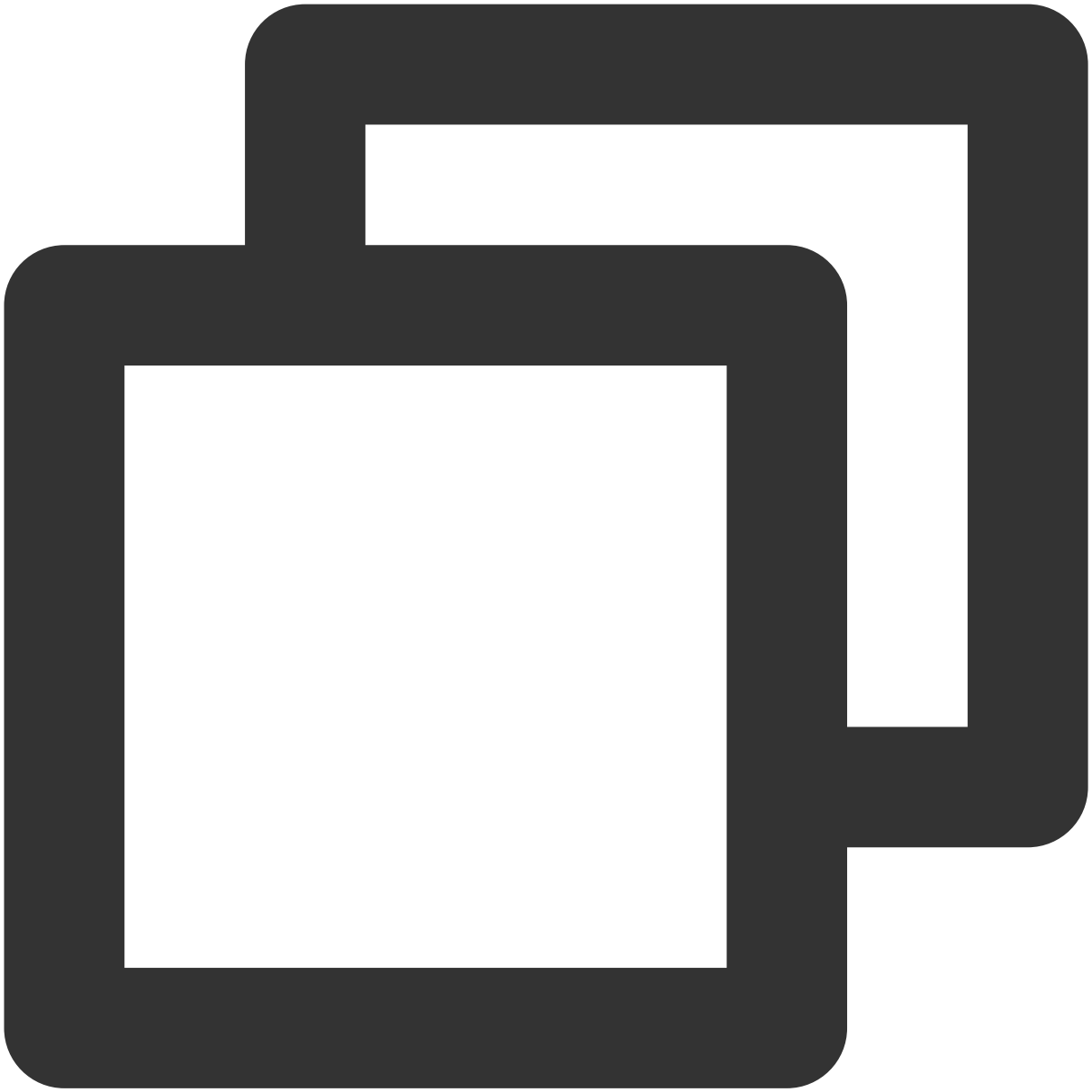
```
    security_group_ids = ["sg-xxx", "sg-xxx"]
}

# 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
  delete_keep_instance = false                    # 设置为false，表明删除节点池时删除关联
  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
    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。



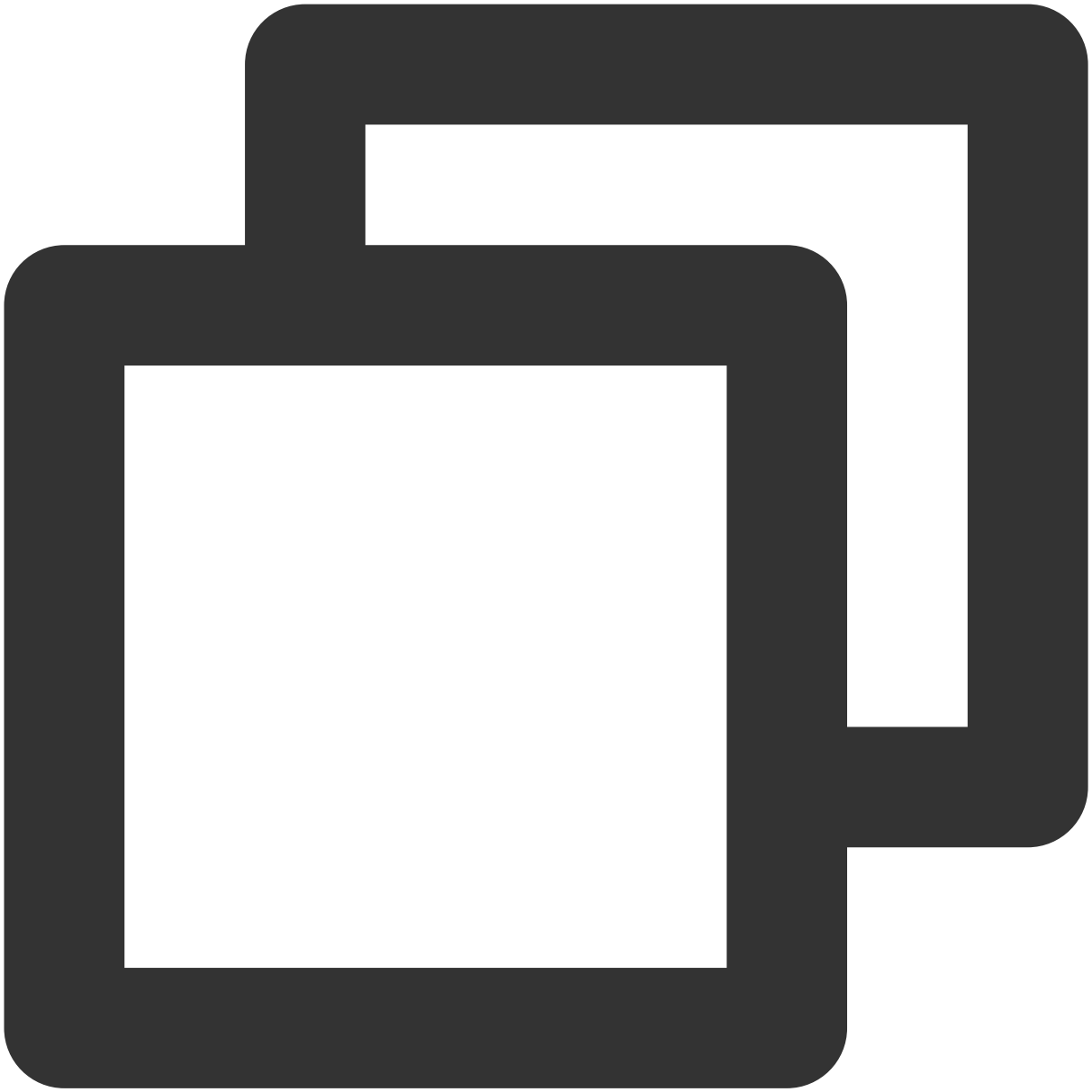
```
terraform plan
```

返回信息如下所示：



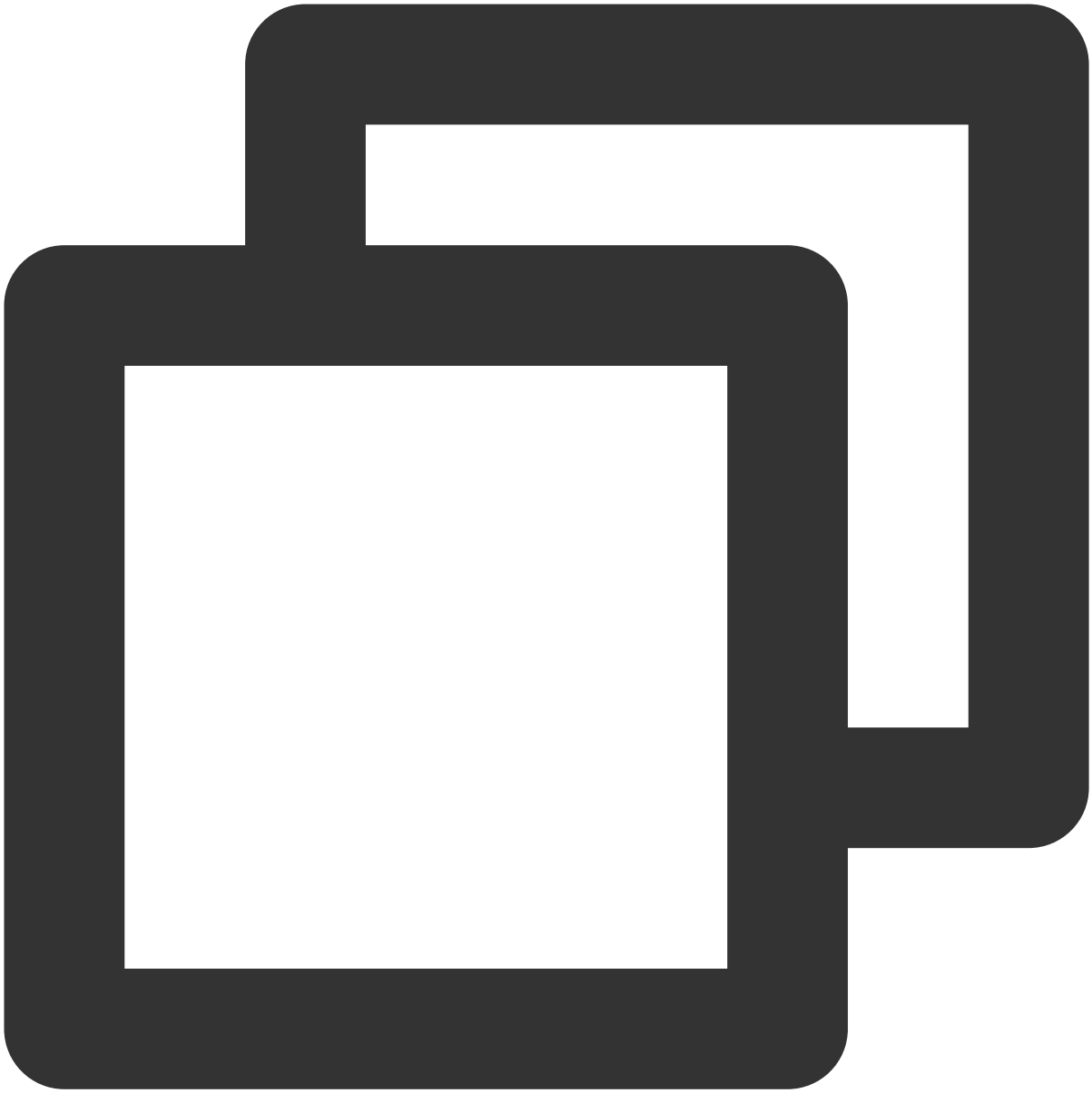
```
Terraform used the selected providers to generate the following execution plan. Res
  + create

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。



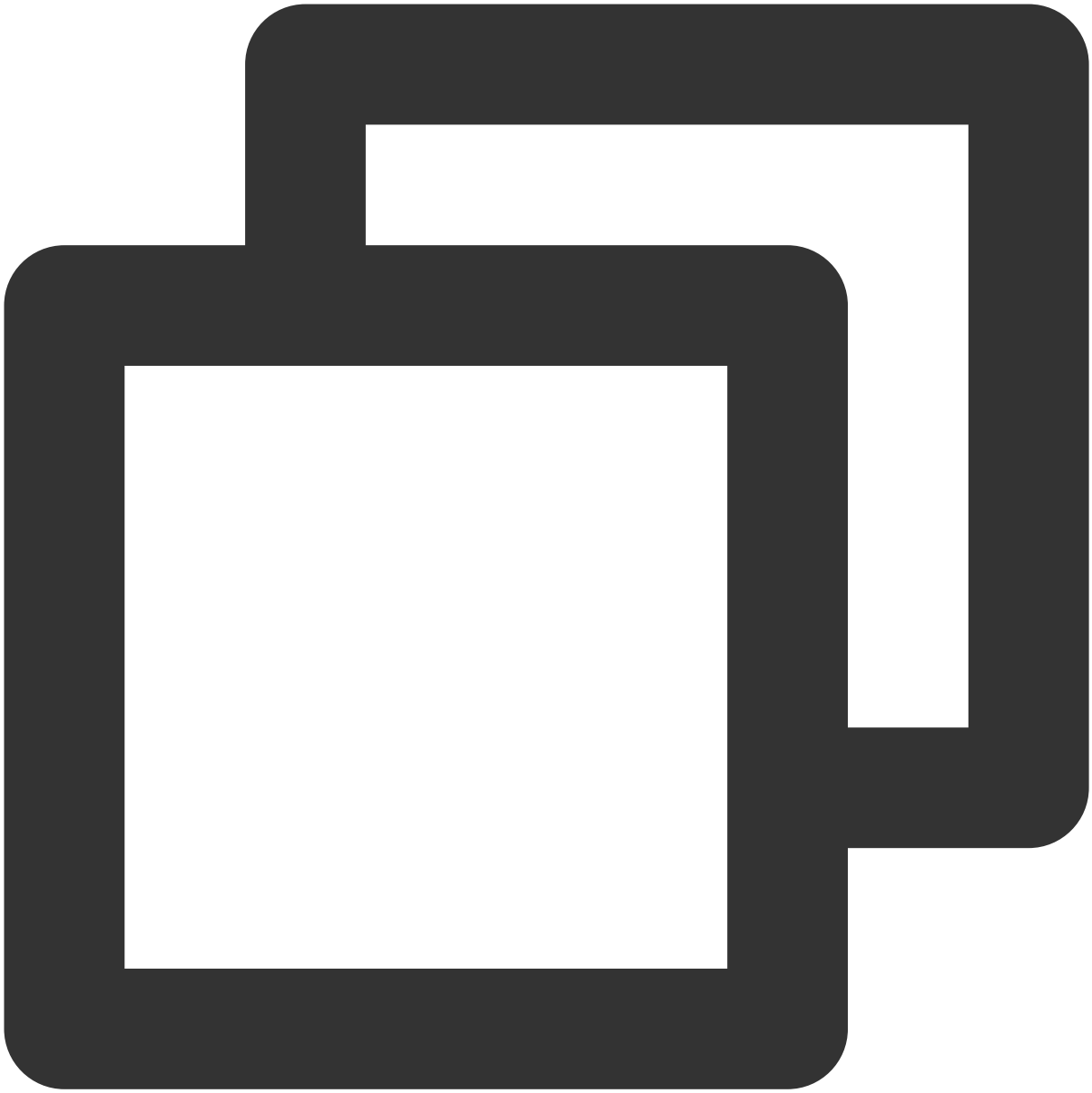
```
terraform apply
```

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：

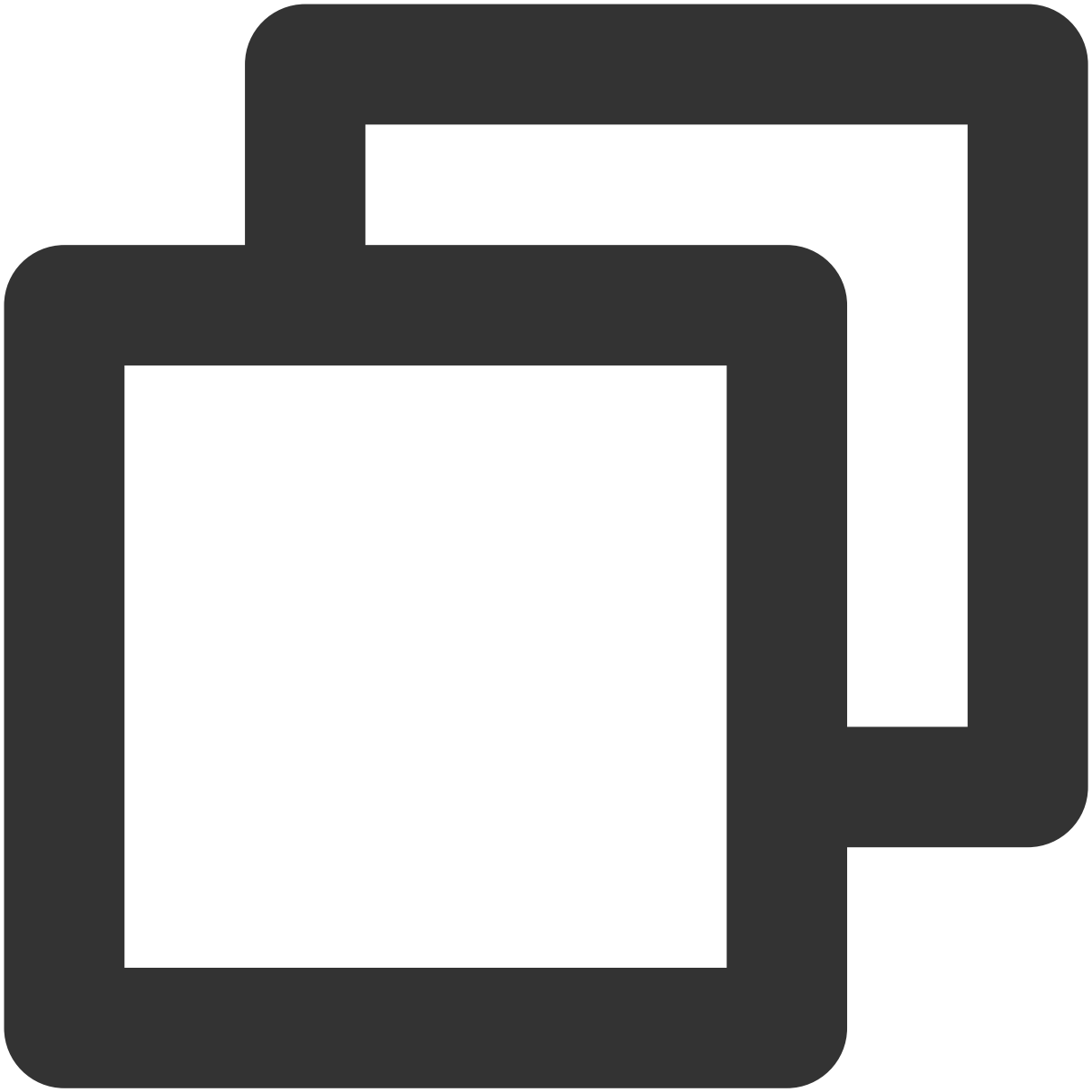


```
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

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。

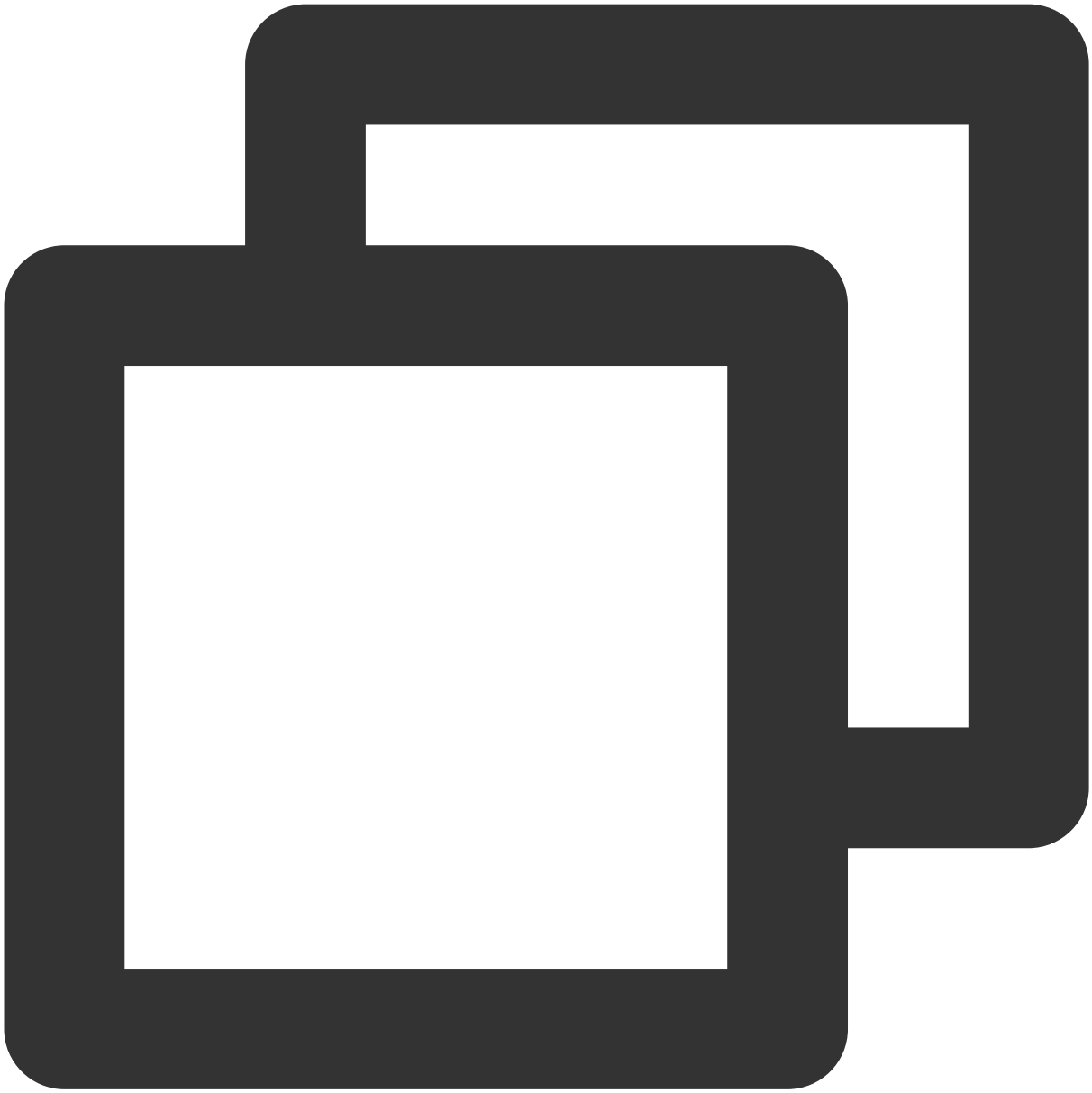
使用 Terraform 清理资源

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。



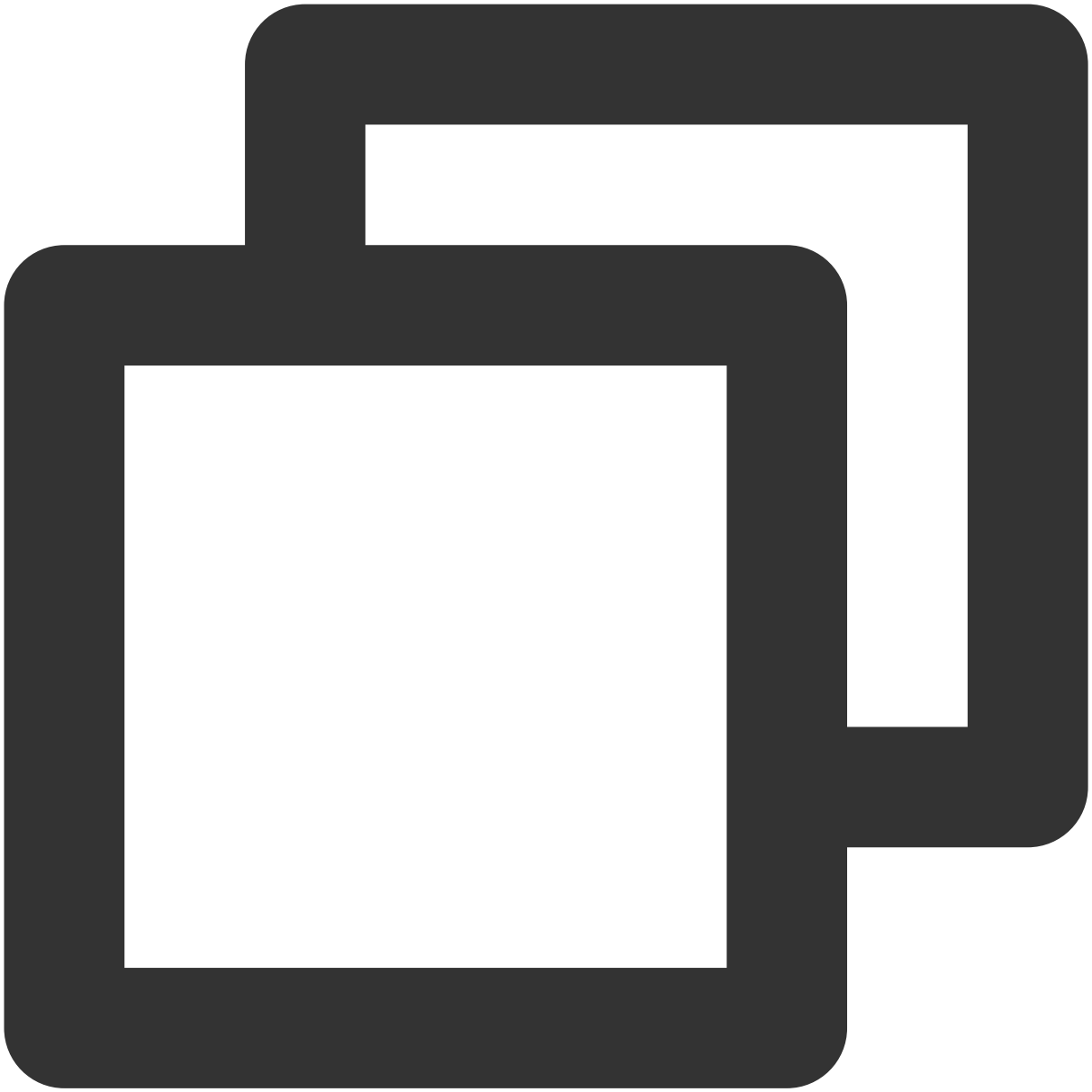
```
terraform destroy
```

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
基于 TKE 的 Jenkins 外网架构应用的构建与部署
示例说明

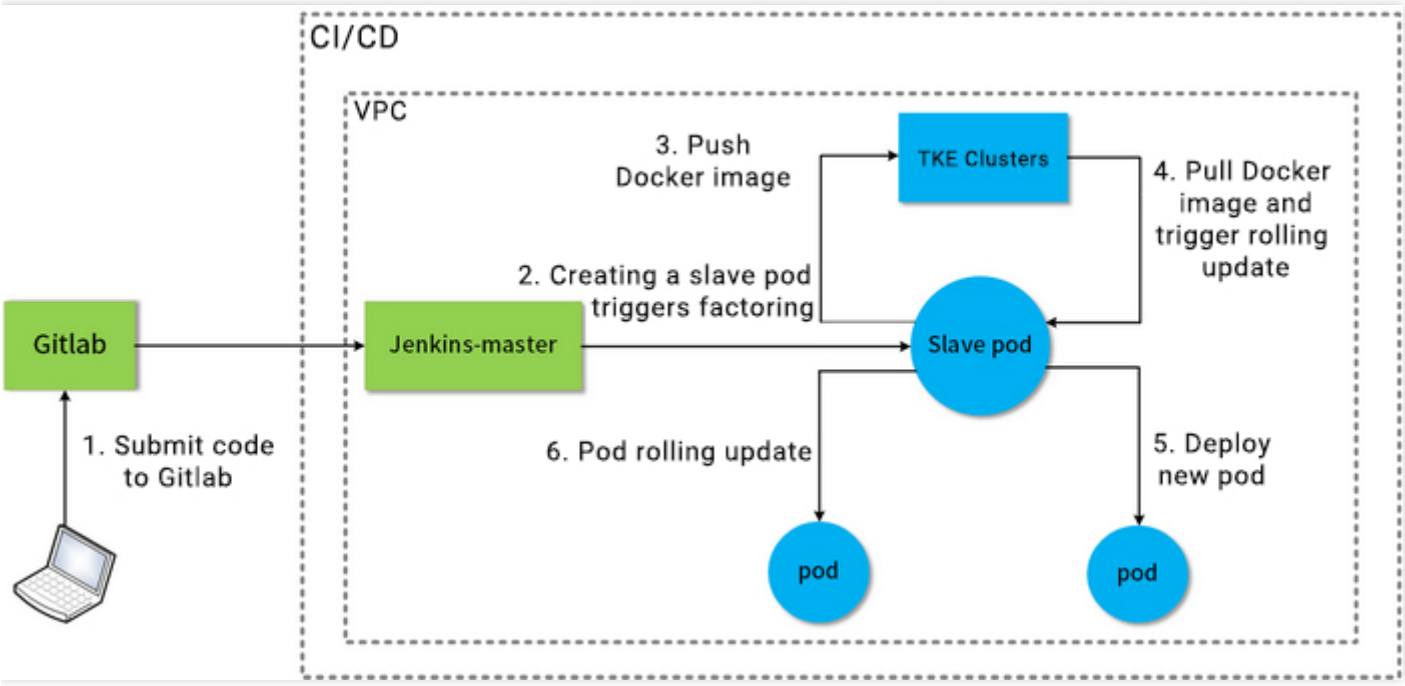
最近更新时间：2020-05-11 14:36:13

操作场景

Jenkins 是连接持续集成和持续交付的桥梁，采用 Jenkins Master/Slave pod 架构能够解决企业批量构建并发限制的痛点，实现和落地真正意义上持续集成。本文介绍了如何在腾讯云容器服务（TKE）中使用 Jenkins，以实现业务快速可持续性交付，减少资源及人力成本。

工作原理

本文采用基于 TKE 的 Jenkins 外网架构，即 Jenkins Master 在 TKE 集群外，slave pod 在集群内。该外网架构图如下所示：



- Jenkins Master 、TKE 集群位于同一 VPC 网络下。

- Jenkins Master 在 TKE 集群外，slave pod 在 TKE 集群的 node 节点上。
- 用户提交代码到 Gitlab，触发 Jenkins Master 调用 slave pod 进行构建打包并推送镜像到 TKE 镜像仓库，TKE 集群拉取镜像并触发滚动更新进行 Pod 部署。
- 多 slave pod 构建可满足批量并发构建的需求。

操作环境

本节介绍了该场景中的具体环境，如下：

TKE 集群

角色	Kubernetes 版本	操作系统
TKE 托管集群	1.16.3	CentOS 7.6.0_x64

Jenkins 配置

角色	版本
Jenins Master	2.190.3
Jenkins Kubernetes 插件	1.21.3

节点

角色	内网 IP	操作系统	CPU	内存	带宽
Jenkins Master	10.0.0.7	CentOS 7.6 64 bit	4核	8GB	3Mbps
Node	10.0.0.14	CentOS 7.6 64 bit	2核	4G	1Mbps

注意事项

- 确保与 TKE 集群同 VPC 下已具备 Jenkins Master 节点，并且该节点已安装 Git。
- 确保操作步骤中用到的 gitlab 代码仓库里面已包含 Dockerfile 文件。
- 建议设置 TKE 集群及 Jenkins Master 安全组内网访问全放通，详情请参见 容器服务安全组设置。

操作流程

按照以下步骤，先对 TKE 集群及 Jenkins 进行配置，再使用 slave pod 进行构建打包并推送镜像至 TKE 镜像仓库，最后通过 TKE 控制台使用拉取的镜像进行 Pod 部署。

1. TKE 集群侧及 Jenkins 侧配置
2. Slave pod 构建配置
3. 构建测试

步骤1：TKE 集群侧及 Jenkins 侧配置

最近更新时间：2023-05-06 17:57:00

TKE 集群侧配置

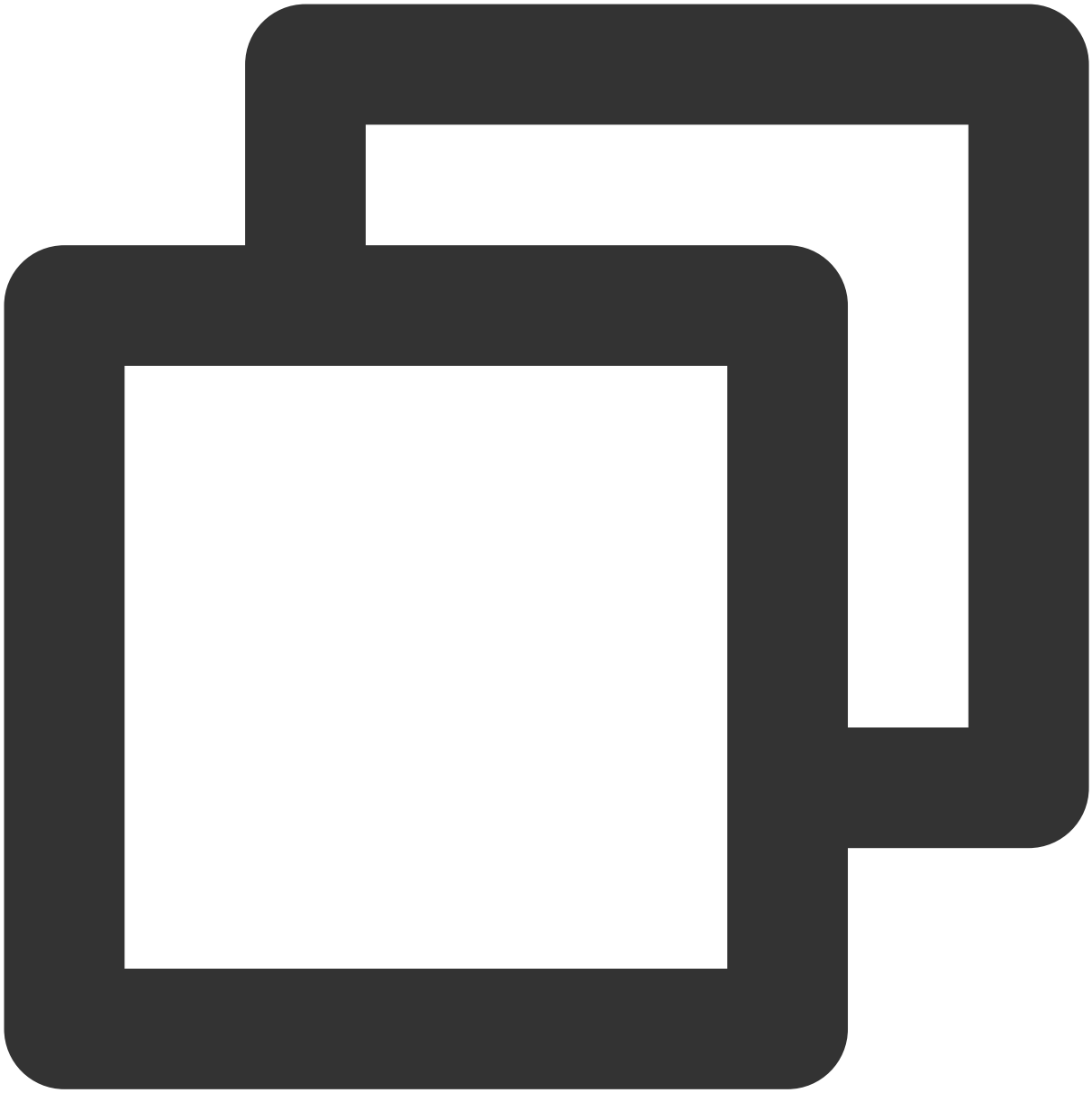
此步骤中介绍了通过在 TKE 中自定义 RBAC 授权 ServiceAccount，以及获取配置 Jenkins 时所需的集群访问地址、token 及集群 CA 证书信息。

获取集群凭证

说明

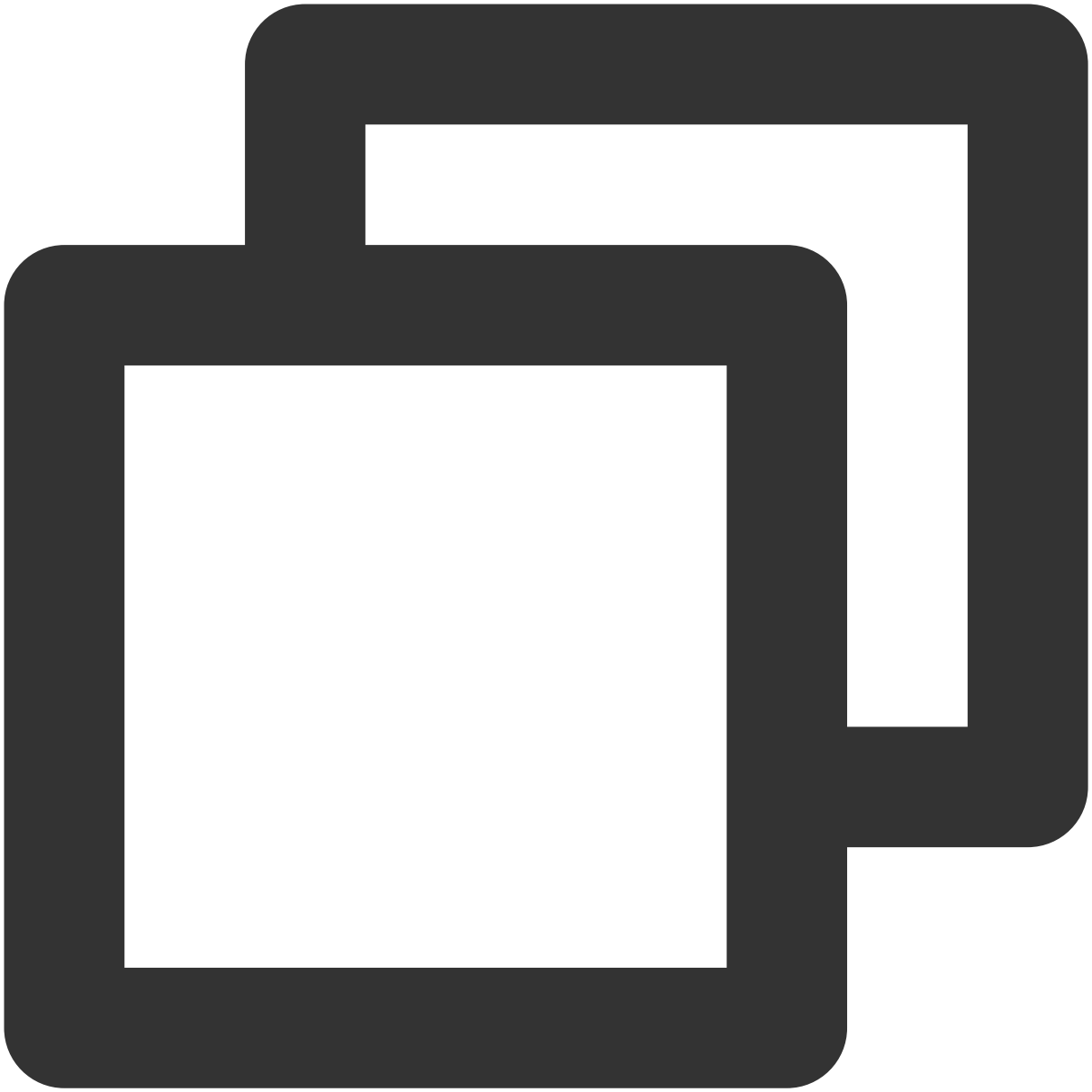
当前集群需要开启内网访问。详情见 Service 控制台操作指引。

1. 使用以下 Shell 脚本，创建测试命名空间 ci、ServiceAccount 类型的测试用户 jenkins 并获取集群访问凭证（token）认证。



```
# 创建测试命名空间ci
kubectl create namespace ci
# 创建测试 ServiceAccount 账户
kubectl create sa jenkins -n ci
# 获取 ServiceAccount 账户自动创建的 Secret token
kubectl get secret $(kubectl get sa jenkins -n ci -o jsonpath={.secrets[0].name}) -
```

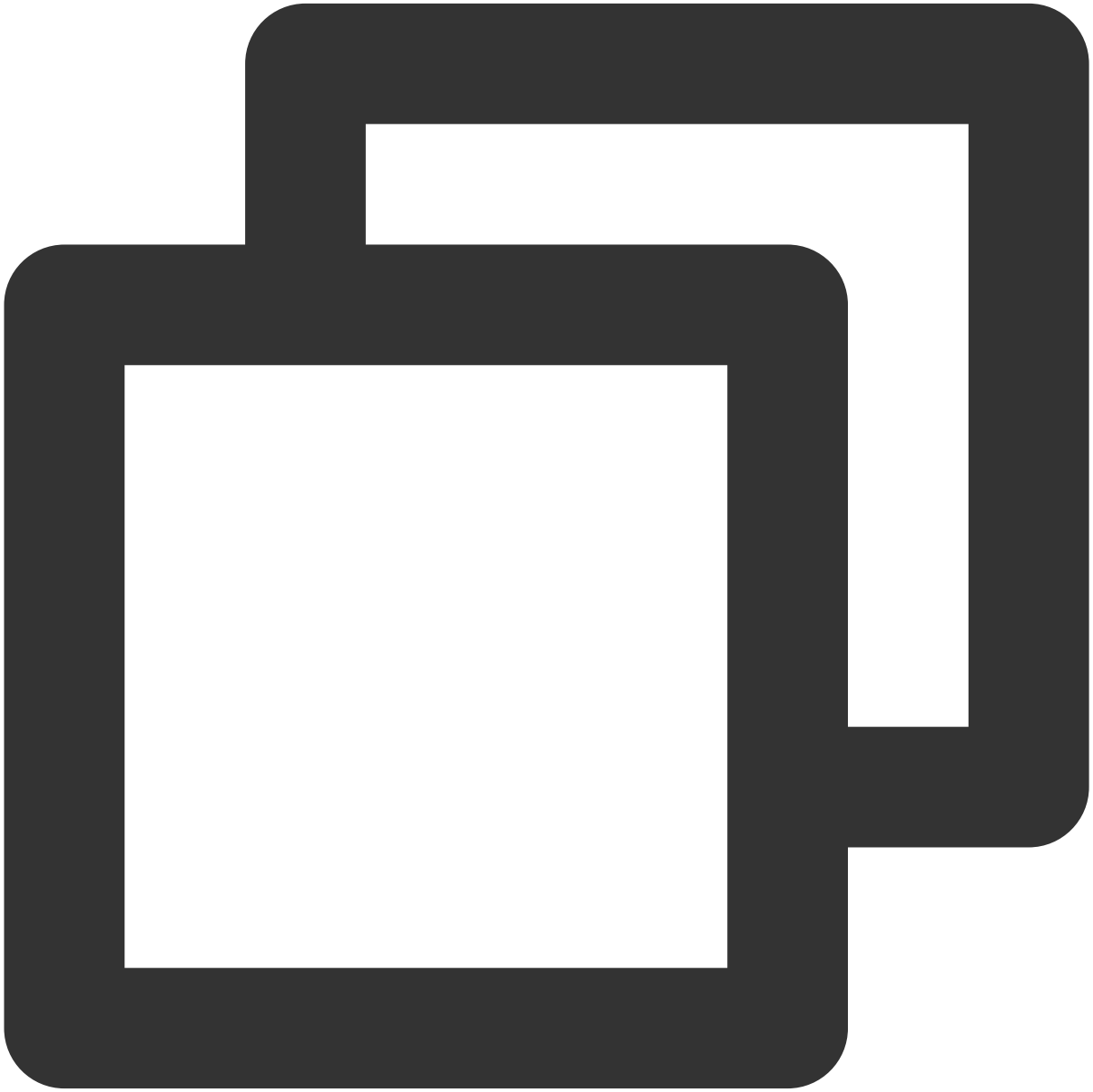
2. 在测试命名空间 ci 创建一个 Role 权限对象资源 jenkins-role.yaml 文件。示例如下：



```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: jenkins
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
```

```
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
```

3. 创建一个 RoleBinding 对象资源 jenkins-rolebinding.yaml 文件。如下权限绑定表示，添加 ServiceAccount 类型的 jenkins 用户在ci 命名空间具有 jenkins（Role 类型）的权限。示例如下：

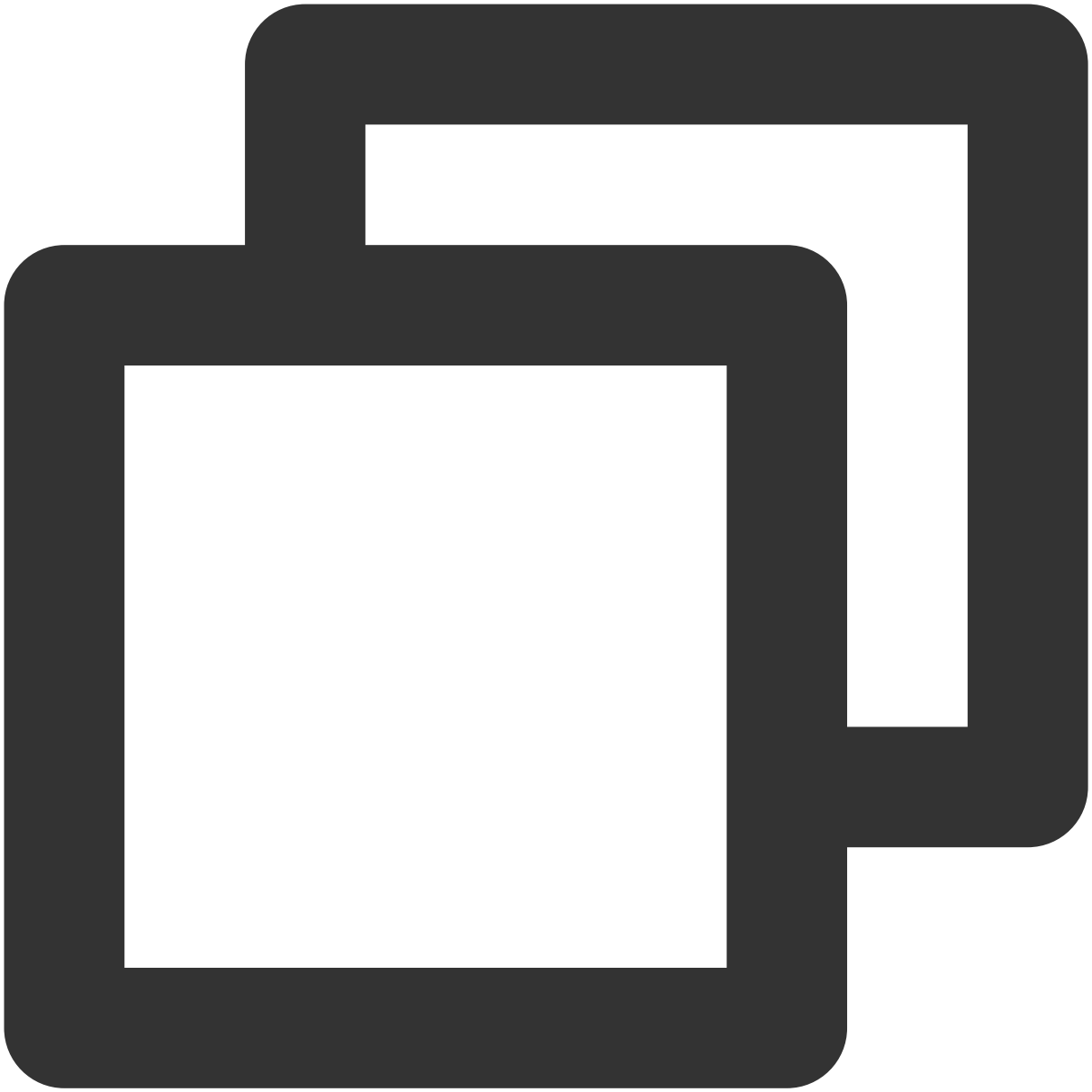


```
apiVersion: rbac.authorization.k8s.io/v1beta1
```

```
kind: RoleBinding
metadata:
  name: jenkins
  namespace: ci
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins
```

获取集群 CA 证书

1. 参考 使用标准登录方式登录 Linux 实例（推荐），登录目标集群的 node 节点。
2. 执行以下命令，查看集群 CA 证书。

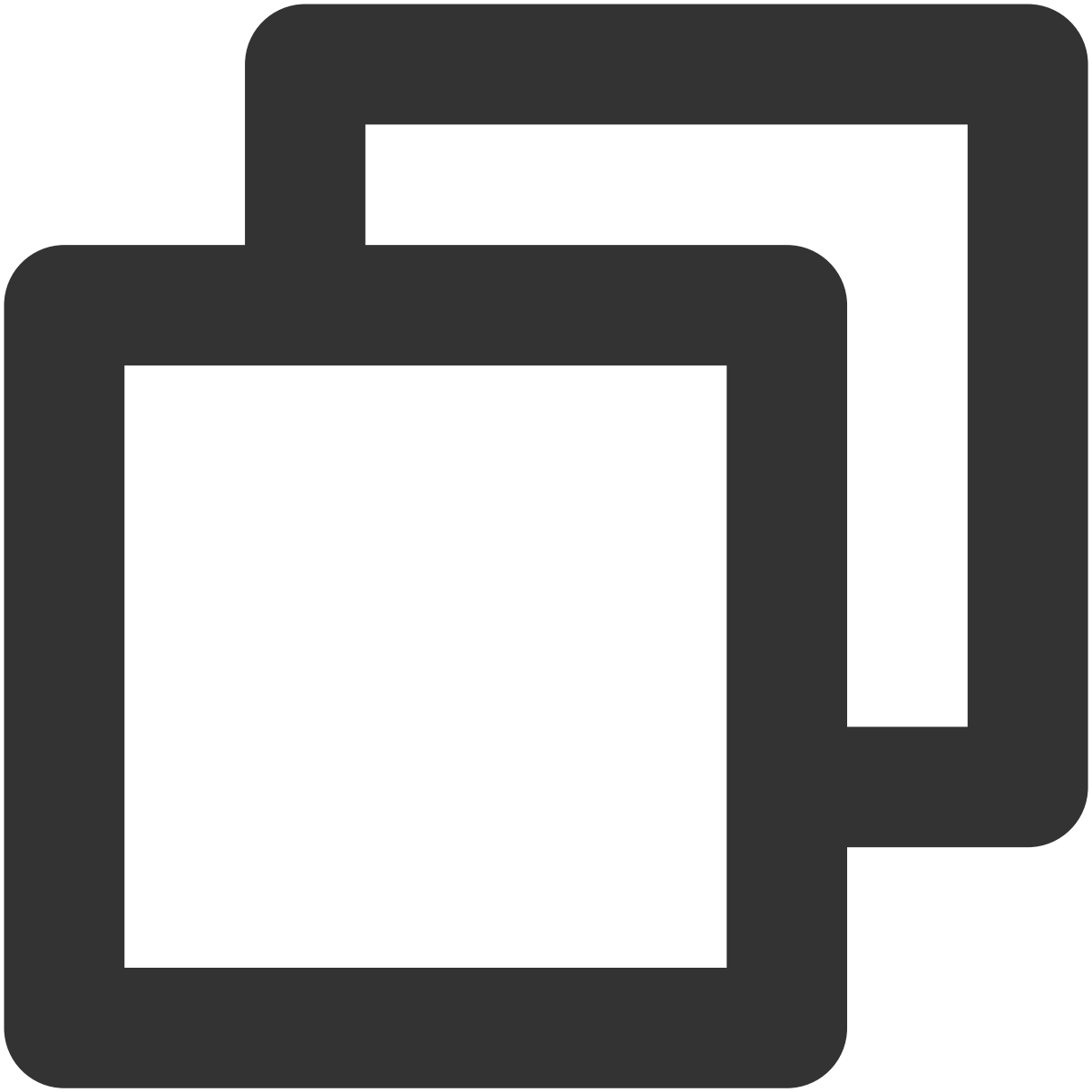


```
cat /etc/kubernetes/cluster-ca.crt
```

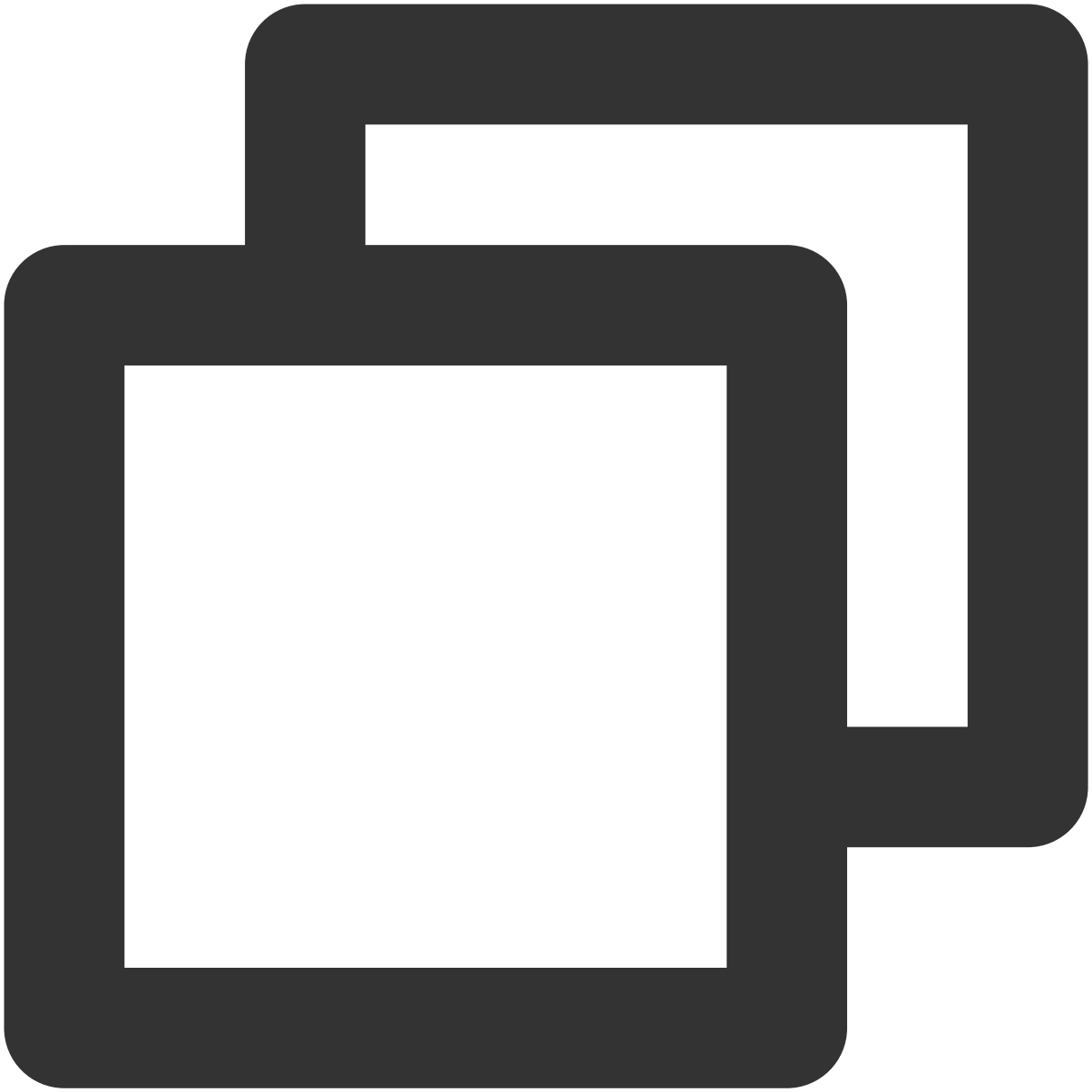
3. 请记录并保存查询所得证书信息。如下图所示：

授权 docker.sock

TKE 集群中的每个 node 节点系统里都有一个 `docker.sock` 文件，slave pod 在执行 `docker build` 时将会连接该文件。在此之前，需逐个登录到每个节点上，依次执行以下命令对 `docker build` 进行授权：



```
chmod 666 /var/run/docker.sock
```



```
ls -l /var/run/docker.sock
```

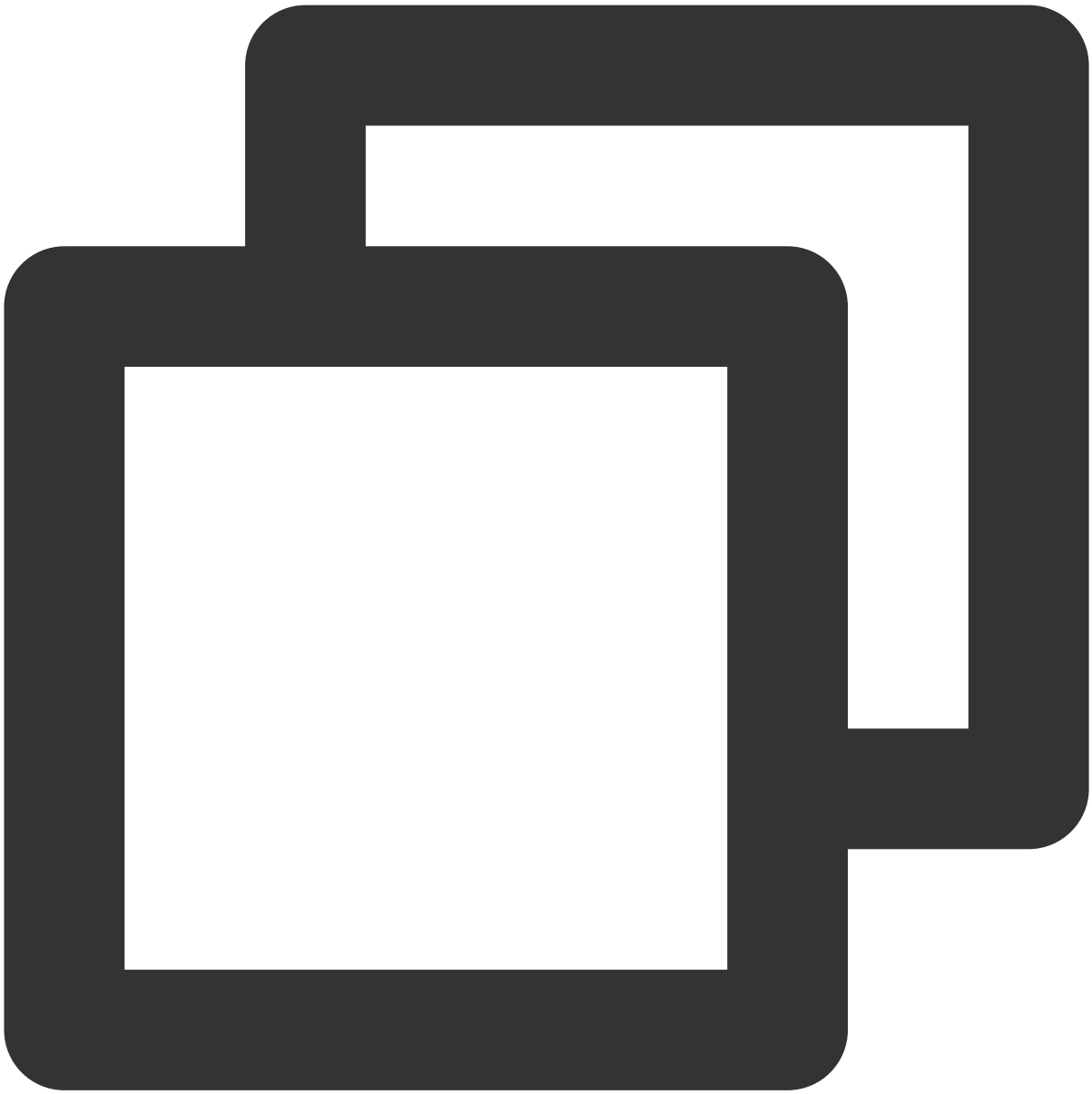
Jenkins 侧配置

说明

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。

添加 TKE 内网访问地址

1. 参考 使用标准登录方式登录 Linux 实例（推荐），登录 Jenkins Master 节点。
2. 执行以下命令，配置访问域名。

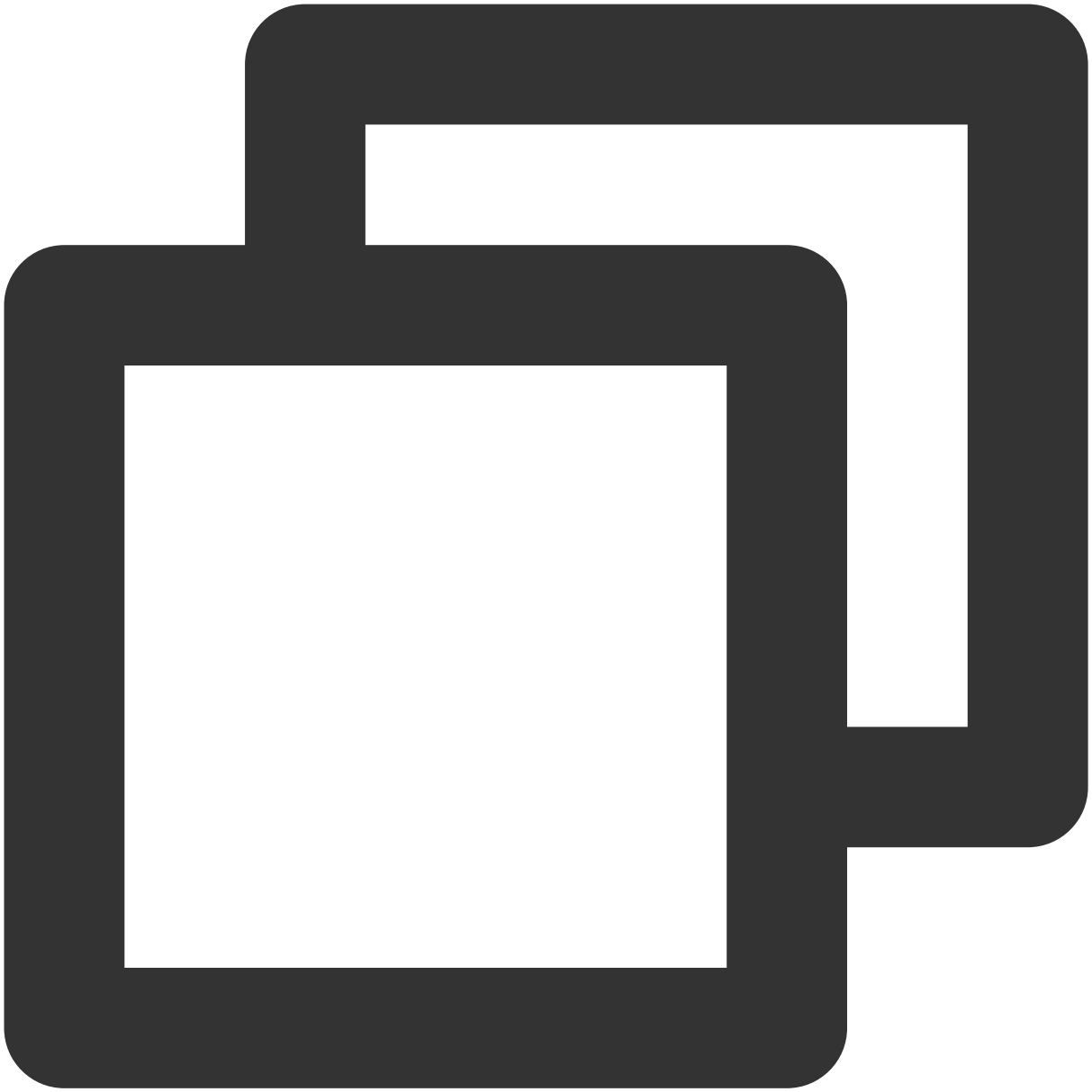


```
sudo sed -i '$a 10.x.x.x cls-ixxxelli.ccs.tencent-cloud.com' /etc/hosts
```

说明

该命令可在集群开启内网访问后，从集群基本信息页面中的“集群APIServer” 中获取，详情请参见 获取集群凭证。

3. 执行以下命令，查看是否配置成功。



```
cat /etc/hosts
```

如下图所示即为配置成功：

```
[root@VM_0_7_centos ~]# sudo sed -i '$a 10 .ccs.tencent-clou
[root@VM_0_7_centos ~]# cat /etc/hosts
127.0.0.1 VM_0_7_centos VM_0_7_centos
127.0.0.1 localhost.localdomain localhost
127.0.0.1 localhost4.localdomain4 localhost4

::1 VM_0_7_centos VM_0_7_centos
::1 localhost.localdomain localhost
::1 localhost6.localdomain6 localhost6

10. .ccs.tencent-cloud.com
```

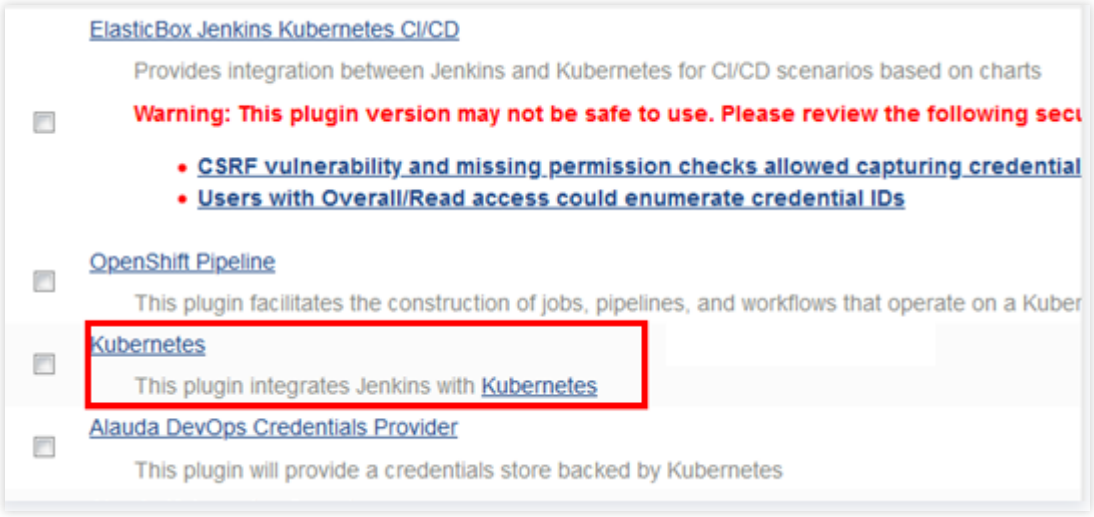
Jenkins 安装必备插件

1. 登录 Jenkins 后台，选择左侧导航栏中的**系统管理**。
2. 在打开的“管理Jenkins” 面板中，单击**插件管理**。
3. 选择插件管理页面中**可选插件**，勾选 Locale、Kubernetes、Git Parameter 和 Extended Choice Parameter。

Locale：汉化语言插件，安装该插件可使 Jenkins 界面默认设置为中文版。

Kubernetes：Kubernetes-plugin 插件。

Git Parameter 和 **Extended Choice Parameter**：用于构建打包时传参。以 Kubernetes 插件为例，如下图所示：



4. 勾选上述插件后单击**直接安装**，并重启 Jenkins 即可。

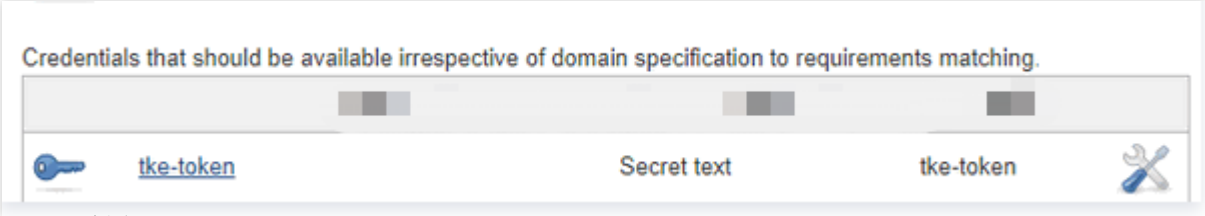
开启 jnlp 端口

1. 登录 Jenkins 后台，选择左侧导航栏中的**系统管理**。
2. 在打开的“管理Jenkins” 面板中，单击**全局安全配置**。
3. 在全局安全配置页中，设置入站代理的 TCP 端口为“指定端口 50000”。
4. 其他配置项保持默认状态，并单击页面下方的**保存**。

添加 TKE 集群 token

1. 登录 Jenkins 后台，选择左侧导航栏中的**凭据 > 系统**。

2. 在打开的“系统”面板中，选择**全局凭据 (unrestricted)** 。
3. 在“全局凭据 (unrestricted)”页中，单击左侧菜单栏中的**添加凭据**，根据以下提示设置凭据基本信息。
类型：选择**Secret text**。
范围：默认为**全局（Jenkins,nodes,items,all child items,etc）**。
Secret：填写 获取集群凭证 步骤中获取的 ServiceAccount jenkins 的 **Token**。
ID：默认不填写。
描述：填写该凭据相关信息，该内容将被显示为凭据名称及描述信息，本文以 `tke-token` 为例。
4. 单击**确定**即可添加，添加成功后该凭据将显示在凭据列表中。如下图所示：



添加 gitlab 认证

1. 在“全局凭据 (unrestricted)”页中，单击左侧菜单栏中的**添加凭据**，并根据以下提示设置凭据基本信息。
类型：选择**Username with password**。
范围：默认为**全局（Jenkins,nodes,items,all child items,etc）**。
用户名：gitlab 用户名。
密码：gitlab 登录密码。
ID：默认不填写。
描述：填写该凭据相关信息，该内容将被显示为凭据名称及描述信息，本文以 `gitlab-password` 为例。
2. 单击**确定**即可添加成功。

配置 slave pod 模板

1. 登录 Jenkins 后台，选择左侧导航栏中的**系统管理**。
2. 在打开的“管理Jenkins” 面板中，单击**系统配置**。
3. 在“系统配置”面板最下方，选择“云”模块下的**新增一个云 > Kubernetes**。
4. 单击 **Kubernetes Cloud details...**，设置 Kubernetes 以下基本信息。
主要参数配置如下，其余选项请保持默认设置：
名称：自定义，本文以 `kubernetes` 为例。
Kubernetes 地址：TKE 集群访问地址，可参考 获取集群凭证 步骤获取。
Kubernetes 服务证书 Key：集群 CA 证书，可参考 获取集群 CA 证书 步骤获取。
凭据：选择 添加 TKE 集群 token 步骤中已创建的凭据 `tke-token` ，并单击**连接测试**。若连接成功则会提示 Connection test succeessful。
Jenkins 地址：填写为 Jenkins 内网地址，例如 `http://10.x.x.x:8080` 。
5. 选择**Pod Templates > 添加 Pod 模板 > Pod Templates details...**，设置 Pod 模板基本信息。
主要参数信息如下，其余选项请保持默认设置：

名称：自定义，本文以 `jnlp-agent` 为例。

标签列表：定义标签名称，构建时可根据该标签选择 Pod，本文以 `jnlp-agent` 为例。

用法：选择**尽可能的使用这个节点**。

6. 在“容器列表”中，选择**添加容器** > **Container Template**，设置以下容器相关信息。

名称：自定义容器名称，本文以 `jnlp-agent` 为例。

Docker 镜像：输入镜像地址 `jenkins/jnlp-slave:alpine` 。

工作目录：保持默认设置，请记录工作目录，将用于 shell 脚本处构建打包。

其余选项保持默认设置即可。

7. 在“卷”中按照以下步骤添加卷，为 slave pod 配置 docker 命令。

7.1 选择**添加卷** > **Host Path Volume**，主机和挂载路径均填写 `/usr/bin/docker` 。

7.2 选择**添加卷** > **Host Path Volume**，主机和挂载路径均填写 `/var/run/docker.sock` 。

7.3 单击页面下方的**保存**，即可完成 slave pod 模板配置。

下一步操作

请前往 步骤2：Slave pod 构建配置 创建新任务及配置任务参数。

步骤2：Slave pod 构建配置

最近更新时间：2023-05-06 17:58:45

本步骤介绍了如何在 Jekins 中通过创建新任务、配置任务参数来构建 slave pod。

说明

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。

创建新任务

1. 登录 Jenkins 后台，单击**新建任务**或**创建一个新任务**。
2. 在新建任务页，设置任务的基本信息。

输入一个任务名称：自定义，本文以 `test` 为例。

类型：选择**构建一个自由风格的软件项目**。

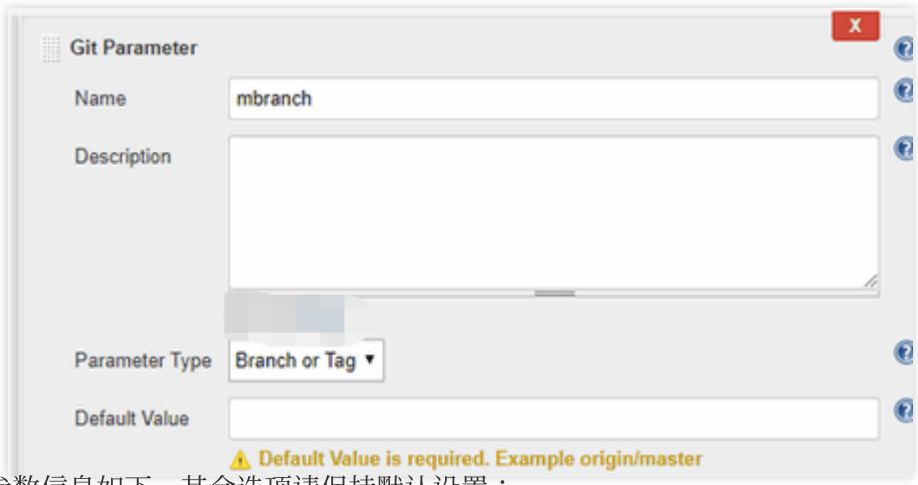
3. 单击**确定**，进入任务参数配置页。
4. 在任务参数配置页，进行基本信息配置。

描述：自定义填写任务的相关信息，本文以 slave pod test 为例。

参数化构建过程：勾选此项，并选择**添加参数** > **Git Parameter**。

任务参数配置

1. 在打开的 “Git Parameter” 面板中，依次设置以下参数。如下图所示：

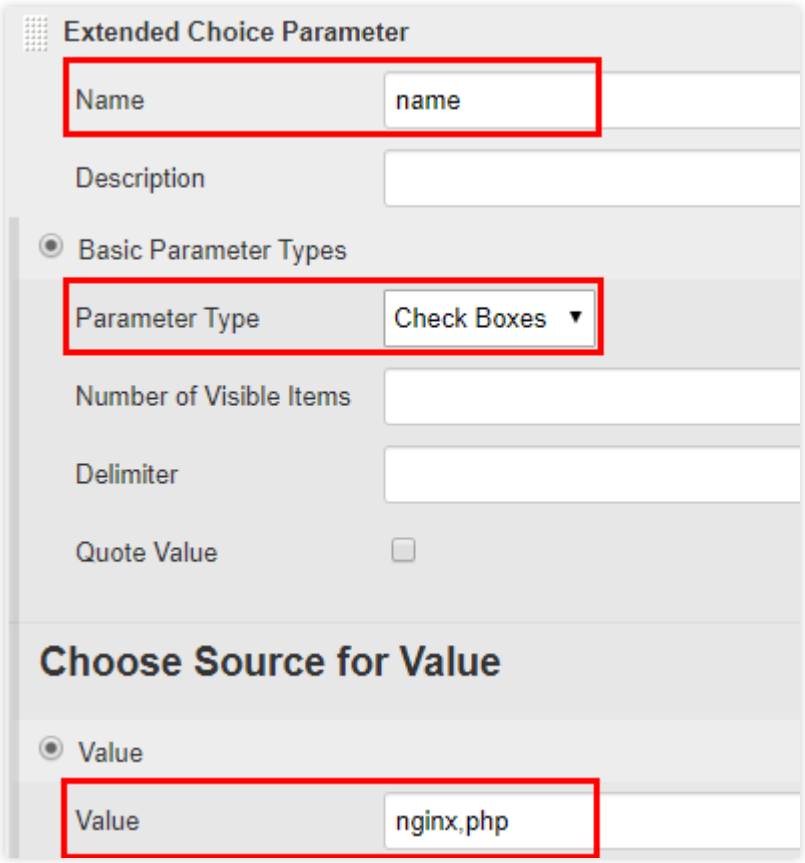


主要参数信息如下，其余选项请保持默认设置：

Name：输入 `mbranch` ，该参数可用于匹配获取分支。

Parameter Type：选择**Branch or Tag**。

2. 选择**添加参数** > **Extended Choice Parameter**，在打开的 “Extended Choice Parameter” 面板中设置以下参数。如下图所示：



主要参数信息如下，其余选项请保持默认设置：

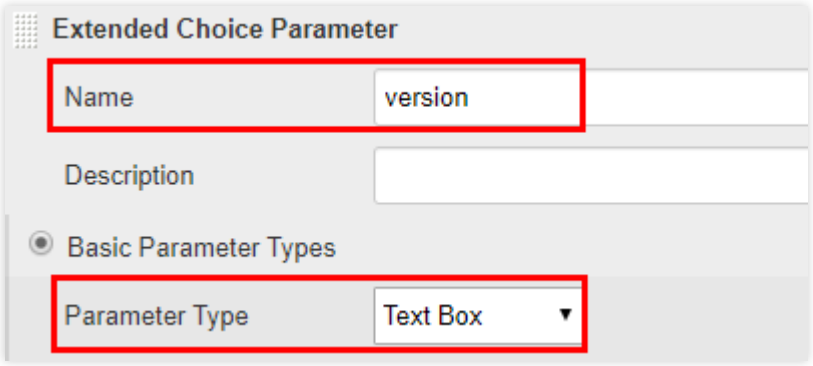
Name：输入 `name`，该参数可用于获取镜像名称。

Basic Parameter Types：选择此项。

Parameter Type：选择**Check Boxes**。

Value：选择此项，并输入自定义镜像名称，该值将传递给变量 `name`，本文以 `nginx,php` 为例。

3. 选择**添加参数** > **Extended Choice Parameter**，在打开的 “Extended Choice Parameter” 面板中设置以下参数。如下图所示：



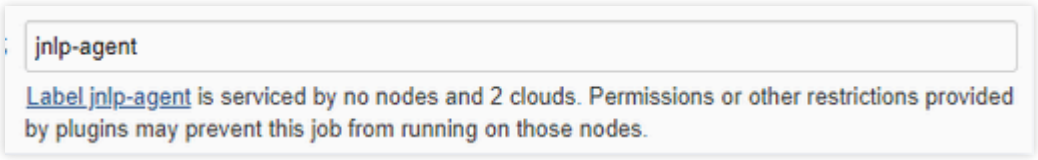
主要参数信息如下，其余选项请保持默认设置：

Name：输入 `version`，该参数用于获取镜像版本变量。

Basic Parameter Types：选择此项。

Parameter Type：选择**Text Box**，表示以文本形式获取镜像值，并传递给变量 `version` 。

4. 勾选**限制项目的运行节点**，标签表达式填写 配置 slave pod 模板 步骤中已设置的 Pod 标签 `jnlp-agent` 。如下图所示：



源码管理配置

在“源码管理”模块中，选择**Git**，并进行以下信息配置。

Repositories：

Repository URL：输入您的 gitlab 地址，例如 `https://gitlab.com/user-name/demo.git` 。

Credentials：选择已在 添加 gitlab 认证 步骤中创建的认证凭据。

Branches to build：

指定分支（为空时代表any）：输入 `$mbranch` ，用于动态获取分支，其值与 Git Parameter 参数中定义的 `mbranch` 值对应。

Shell 打包脚本配置

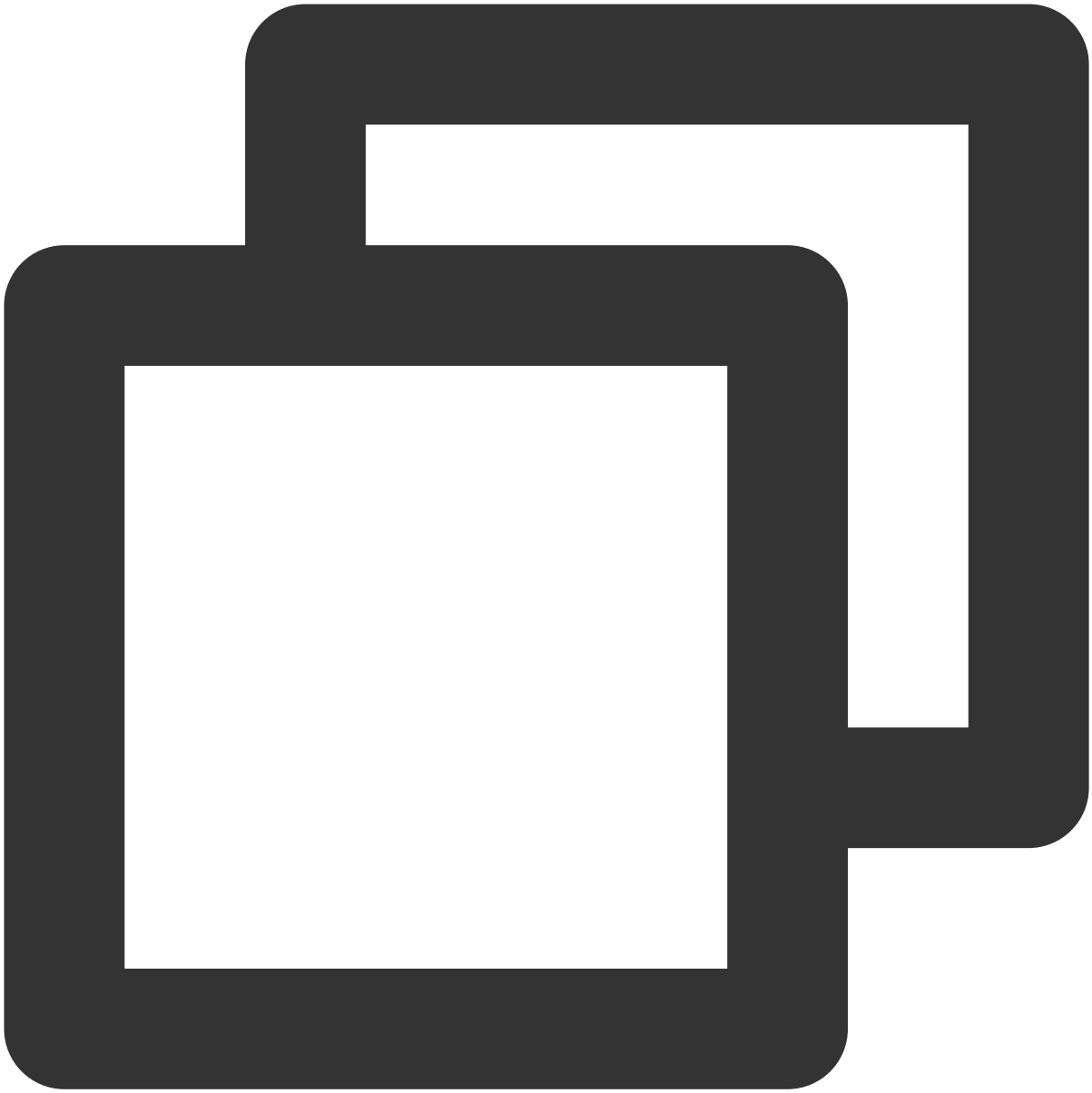
1. 在“构建”模块中，选择**增加构建步骤 > 执行 shell**。

2. 将以下脚本内容复制粘贴至“命令”输入框中，并单击**保存**。

注意

脚本中 gitlab 地址、TKE 镜像地址、镜像仓库用户名及密码等信息为示例使用，请根据实际需求进行更换。

请确保基于源代码 Docker build 构建打包处，工作目录 `/home/Jenkins/agent` 需与“容器列表”中的 Container Template 工作目录一致。



```
echo " gitlab 地址为：https://gitlab.com/[user]/[project-name]].git"
echo "选择的分支（镜像）为:"$mbranch, "设置的分支（镜像）版本为:"$version
echo " TKE 镜像地址：hkccr.ccs.tencentyun.com/[namespace]/[ImageName]"

echo "1.登录 TKE 镜像仓库"
docker login --username=[username] -p [password] hkccr.ccs.tencentyun.com

echo "2.基于源代码 Docker build 构建打包："
cd /home/Jenkins/agent/workspace/[project-name] && docker build -t $name:$version

echo "3.Docker镜像上传至TKE仓库："
```

```
docker tag $name:$version hkccr.ccs.tencentyun.com/[namespace]/[ImageName]:$name-$
docker push hkccr.ccs.tencentyun.com/[namespace]/[ImageName]:$name-$version
```

该脚本提供以下功能：

获取选择的分支、镜像名称及镜像版本。

将与代码合并构建后的 docker 镜像推送至 TKE 镜像仓库。

下一步操作

至此您已成功构建 slave pod，请前往 构建测试 进行推送镜像及验证操作。

构建测试

最近更新时间：2023-05-24 17:13:59

此步骤介绍如何推送单个或多个镜像至 TKE 镜像仓库，并通过 TKE 控制台使用该镜像创建 Deployment。

构建配置

1. 登录 Jenkins 后台，单击任务列表中已在 Slave pod 构建配置 步骤所创建的任务 test。
2. 单击左侧菜单栏中的**Build with Parameters**，打开“工程 test” 面板，进行以下参数设置。

- **mbranch**：选择构建所需分支，本文以 `origin/nginx` 为例。
- **name**：根据实际需求选择所构建镜像的名称，本文以 `nginx` 为例。
- **version**：自定义输入镜像版本号，本文以 `v1` 为例。

3. 单击**开始构建**。
   构建成功即前往容器服务控制台 >**镜像仓库**> **我的镜像** 中进行查看。

控制台发布

1. 登录 容器服务 控制台，选择左侧导航栏中的 **集群** 。
2. 选择目标集群 ID，进入待创建 “Deployment” 的集群管理页面。
3. 单击**新建**，进入 “新建Workload ” 页面，参考 创建 Deployment 进行关键参数设置。
   在“实例内容器”中，可选择**选择镜像>我的镜像**，选择上述构建过程中已成功上传的镜像。
4. 单击**保存**即可完成部署。
   在**Pod 管理页**中，nginx pod 正常运行且为 Running 状态即为部署成功。

相关操作：批量构建设置

1. 登录 Jenkins 后台，选择左侧导航栏中的**系统管理**，在打开的“管理Jenkins” 面板中单击**系统配置**。

2. 在“系统配置”页，自定义修改“执行者数量”，本文以数量10为例。

> 说明：
> 执行者数量为10 ，则表示可以同时执行10个 Job。

3. 其他配置项保持 配置 slave pod 模板 步骤中所设置的内容。

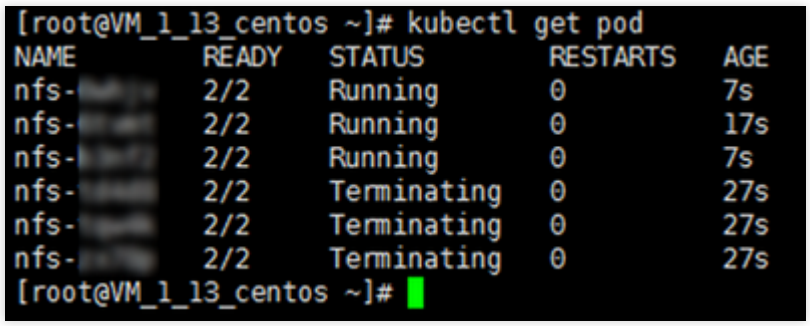
4. 参考 Slave pod 构建配置 步骤，根据实际需求依次新建10个 test。

5. 参考 构建配置 步骤依次执行多个任务构建。

6. 成功构建后，您可登录 node 节点，执行以下命令查看 job pod。

```
kubectl get pod
```

返回类似如下结果，则表示调用成功。



在 containerd 集群中使用 Docker 做镜像构建服务

最近更新时间：2020-12-24 17:56:49

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; docke
r 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; whi
le 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; doc
ker info >/dev/null 2>&1; done; docker pull library/busybox:latest; docker save
-o busybox-latest.tar library/busybox:latest; docker rmi library/busybox:lates
t; 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：2023-05-06 17:36:46

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。

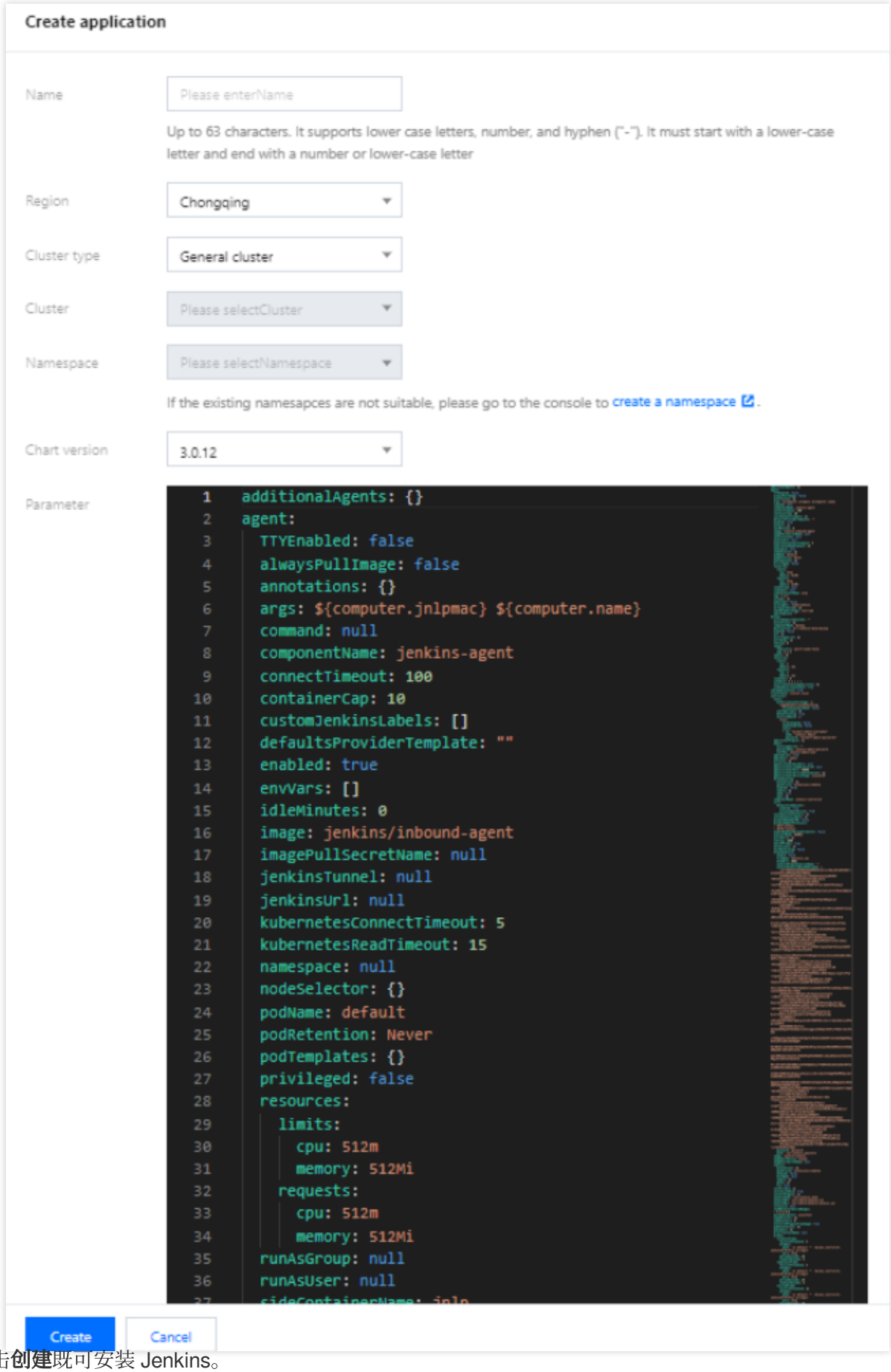
前提条件

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

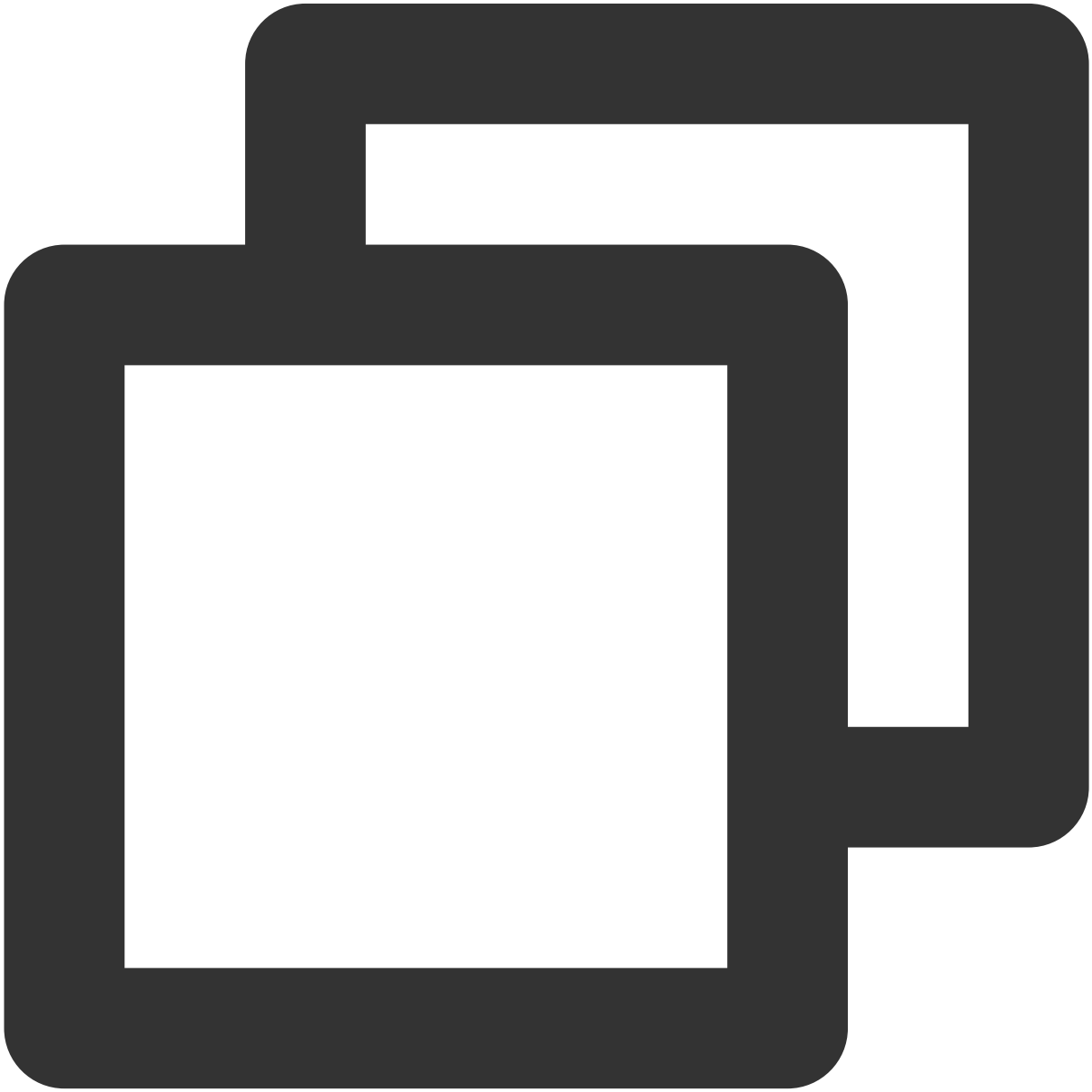
默认情况下，在集群外无法访问 Jenkins UI。如需访问 Jenkins UI，通常使用 Ingress 来暴露访问。TKE 提供 CLB 类型 Ingress 与 Nginx 类型 Ingress 两种 Ingress，您可参考文档自行选择。

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}' |
```

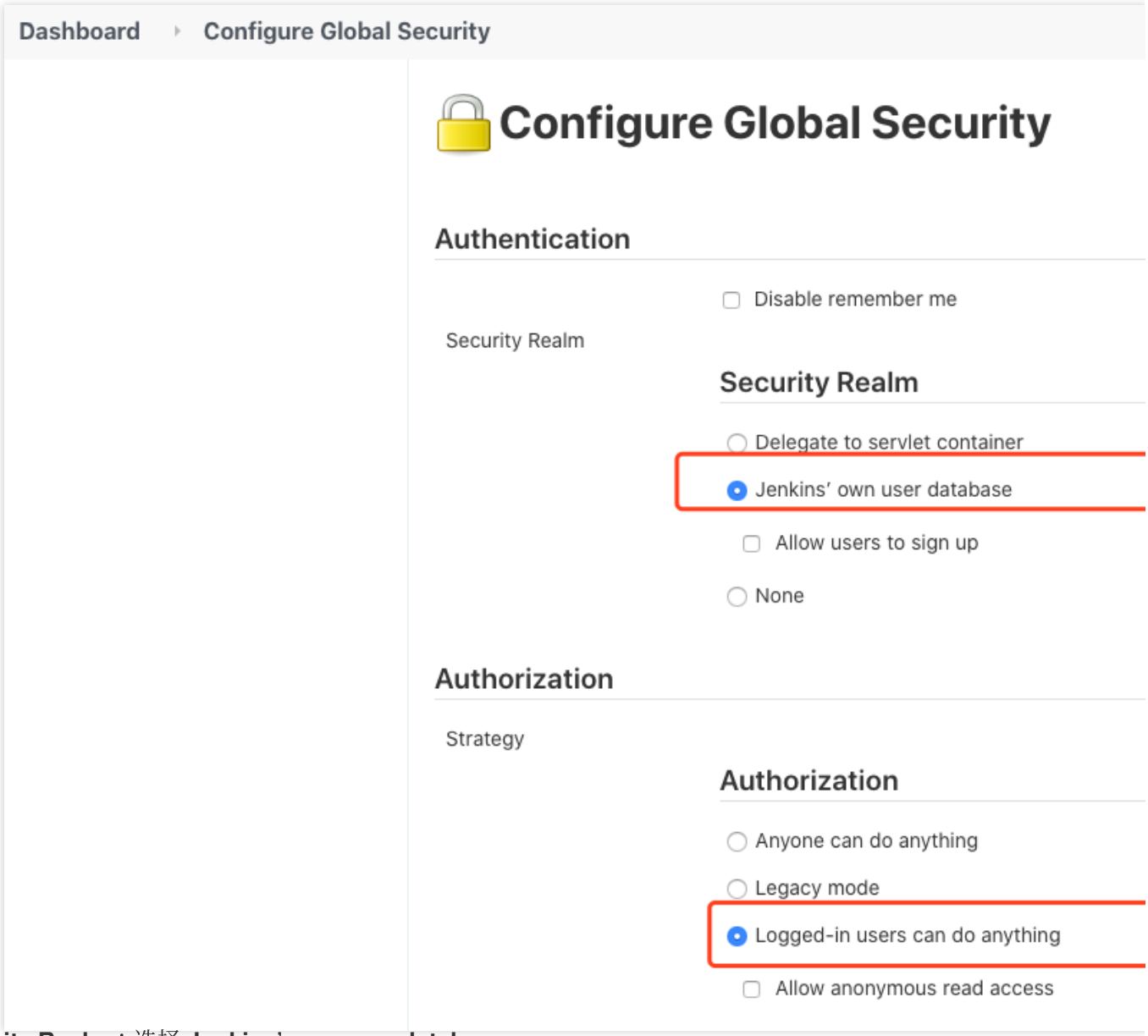
注意

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。

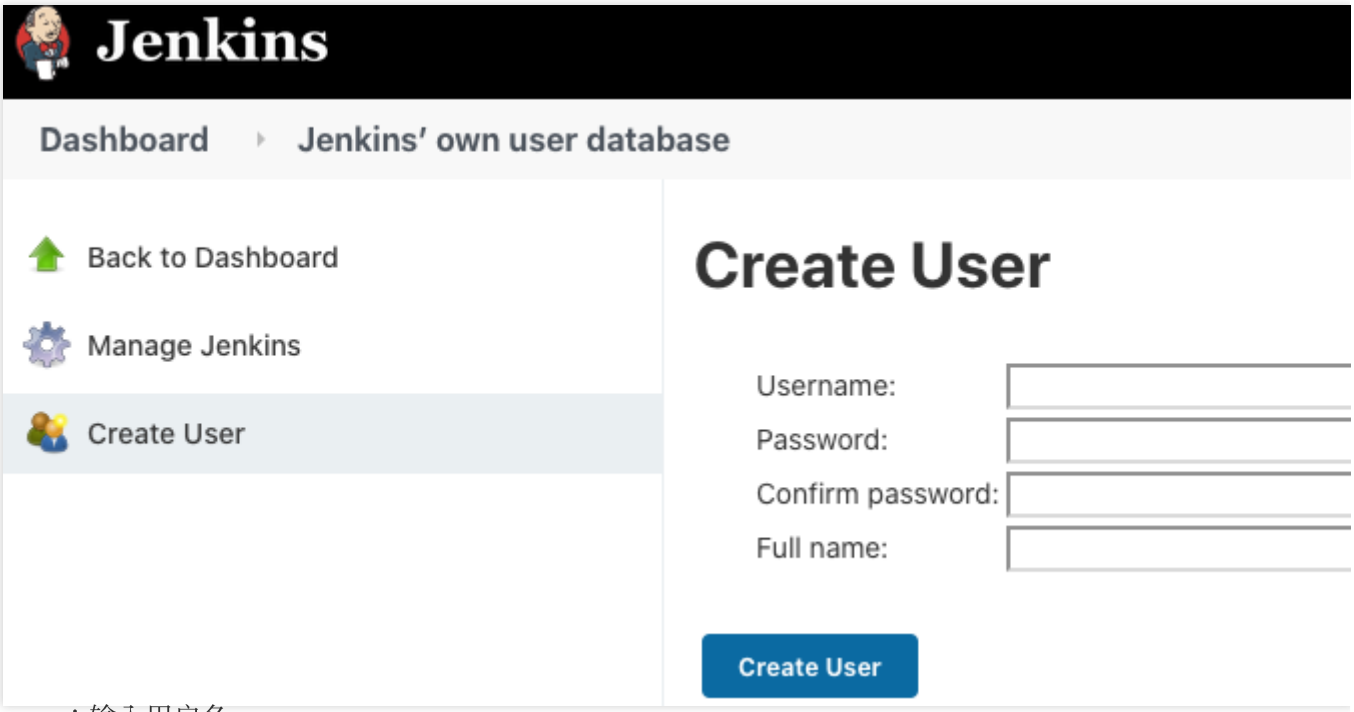
1. 登录 Jenkins 后台，选择 **Dashboard** > **Manage Jenkins** > **Security** > **Configure Global Security**，进入认证与授权策略页面。如下图所示：



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。

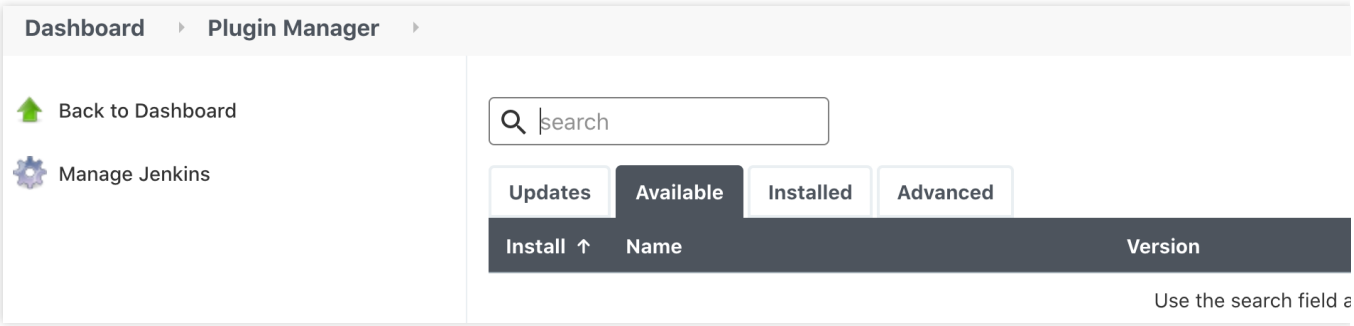
Confirm password：确认用户密码。

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

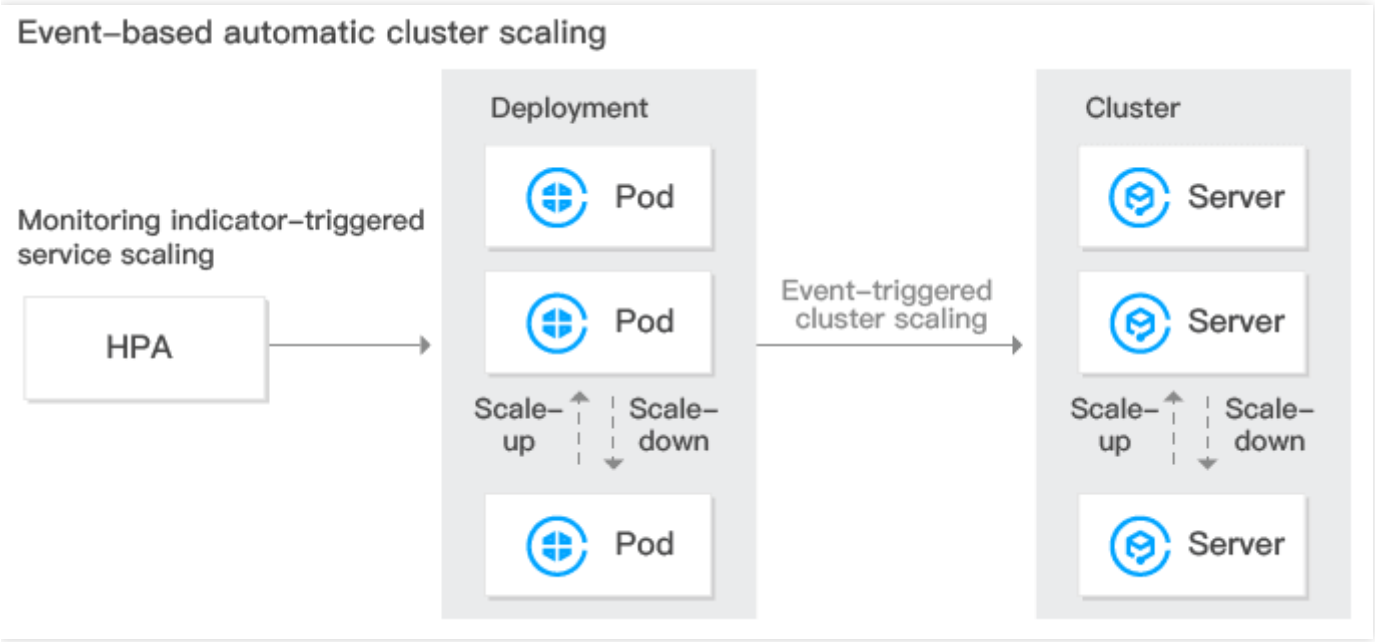
gitlab

github

弹性伸缩
集群弹性伸缩实践

最近更新时间：2022-09-22 16:55:11

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：2023-05-23 16:34:28

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 对资源进行提前占位（带 request 的 pause 容器，实际消耗资源量低），为一些可能会出现流量突高的高优先级业务预留部分资源作为缓冲。当需要扩容 Pod 时，高优先级的 Pod 就可以快速抢占低优先级 Pod 的资源进行调度，而低优先级的 `tke-autoscaling-placeholder` 的 Pod 则会被“挤走”，状态变成 Pending，如果配置了节点池并启用弹性伸缩，将会触发节点的扩容。由于通过一些资源作为缓冲，即使节点扩容慢，也能保证一些 Pod 能够快速扩容并调度上，实现秒级伸缩。调整预留的缓冲资源多少，可根据实际需求调整 `tke-autoscaling-placeholder` 的 request 或副本数。

使用限制

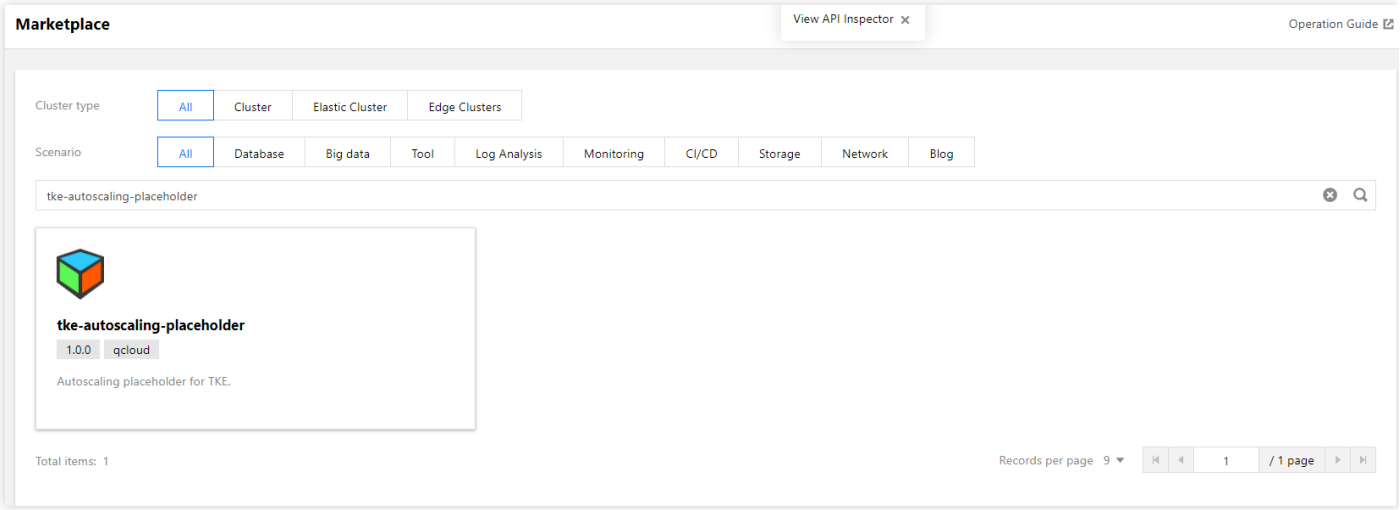
使用 `tke-autoscaling-placeholder` 应用，集群版本需要在1.18以上。

操作步骤

安装 tke-autoscaling-placeholder

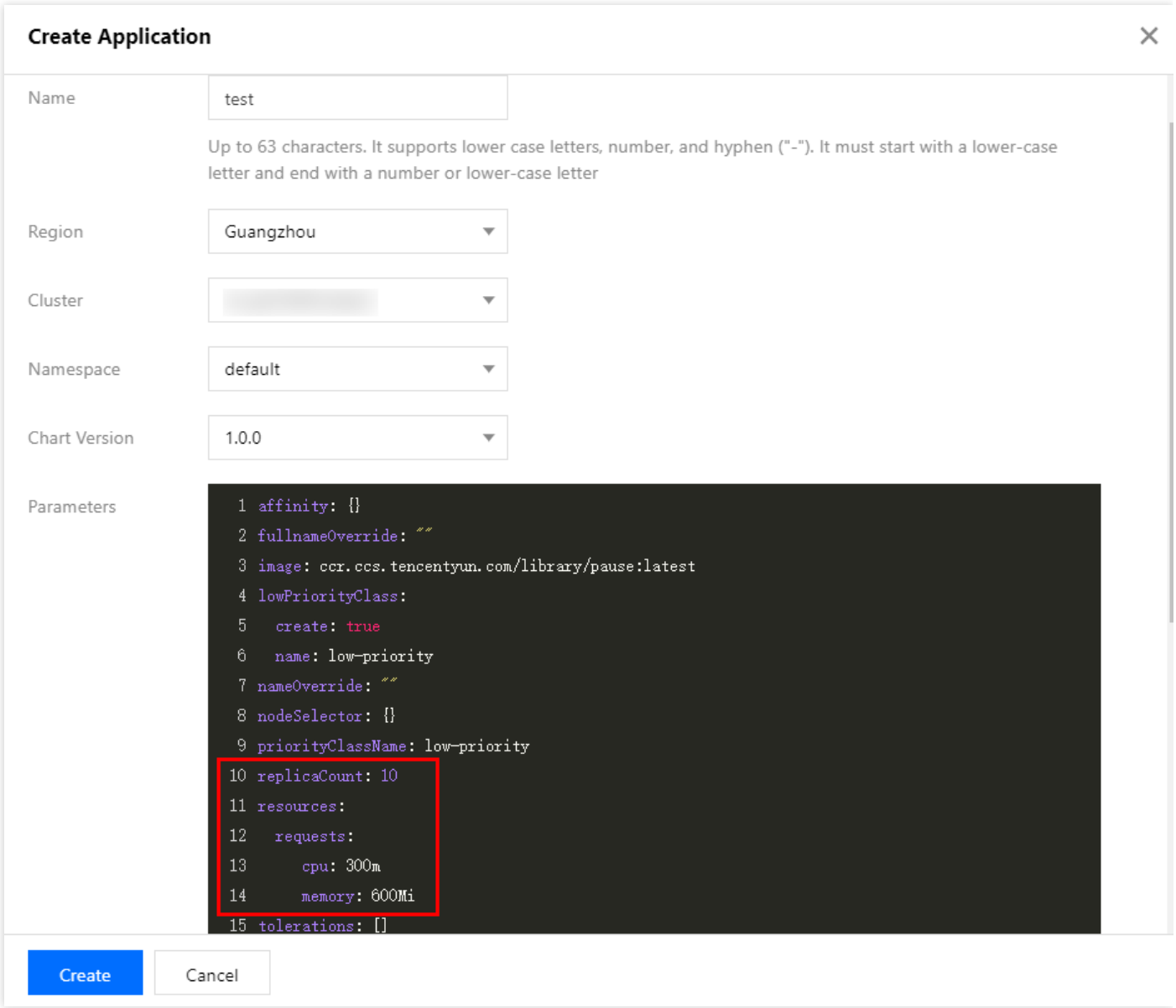
1. 登录 容器服务控制台。
2. 在左侧导航栏中，单击**应用市场**进入“应用市场”管理页面。

3. 在应用市场页面搜索框，输入 `tke-autoscaling-placeholder` 进行搜索，找到该应用。如下图所示：



4. 在“应用详情页”中，单击“基本信息”模块中的**创建应用**。

5. 在弹出的“创建应用”窗口中，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/library/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	{}

6. 单击**创建**，部署 tke-autoscaling-placeholder 应用。
7. 执行如下命令，查看进行资源占位的 Pod 是否启动成功。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：2022-04-22 10:09:58

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 hpa-metrics-server，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：2023-04-07 20:02:34

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。

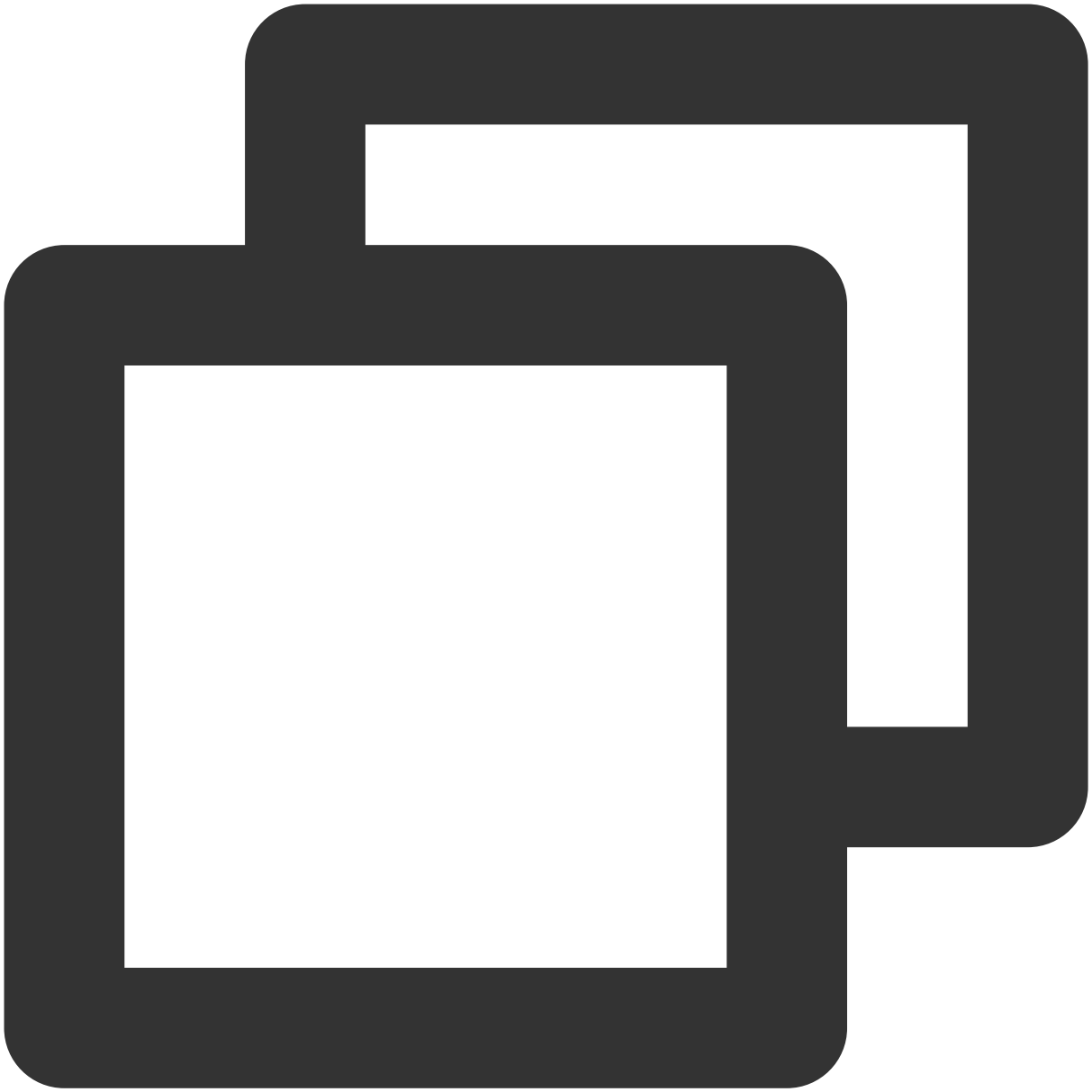
前提条件

已创建1.12或以上版本的 TKE 集群，详情请参见 创建集群。
已部署 Prometheus 并进行相应的自定义指标采集。
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：

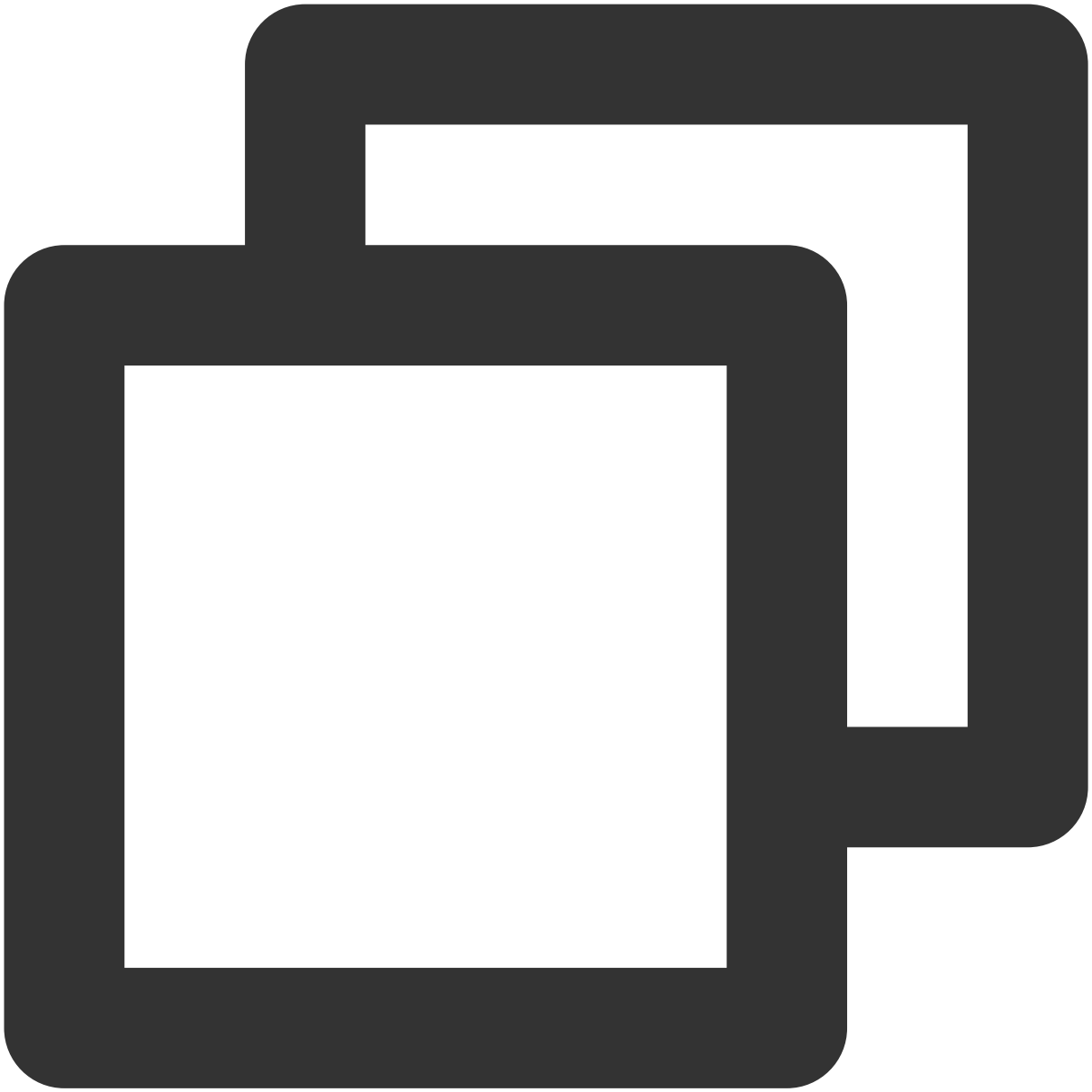


```
package main

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:
```

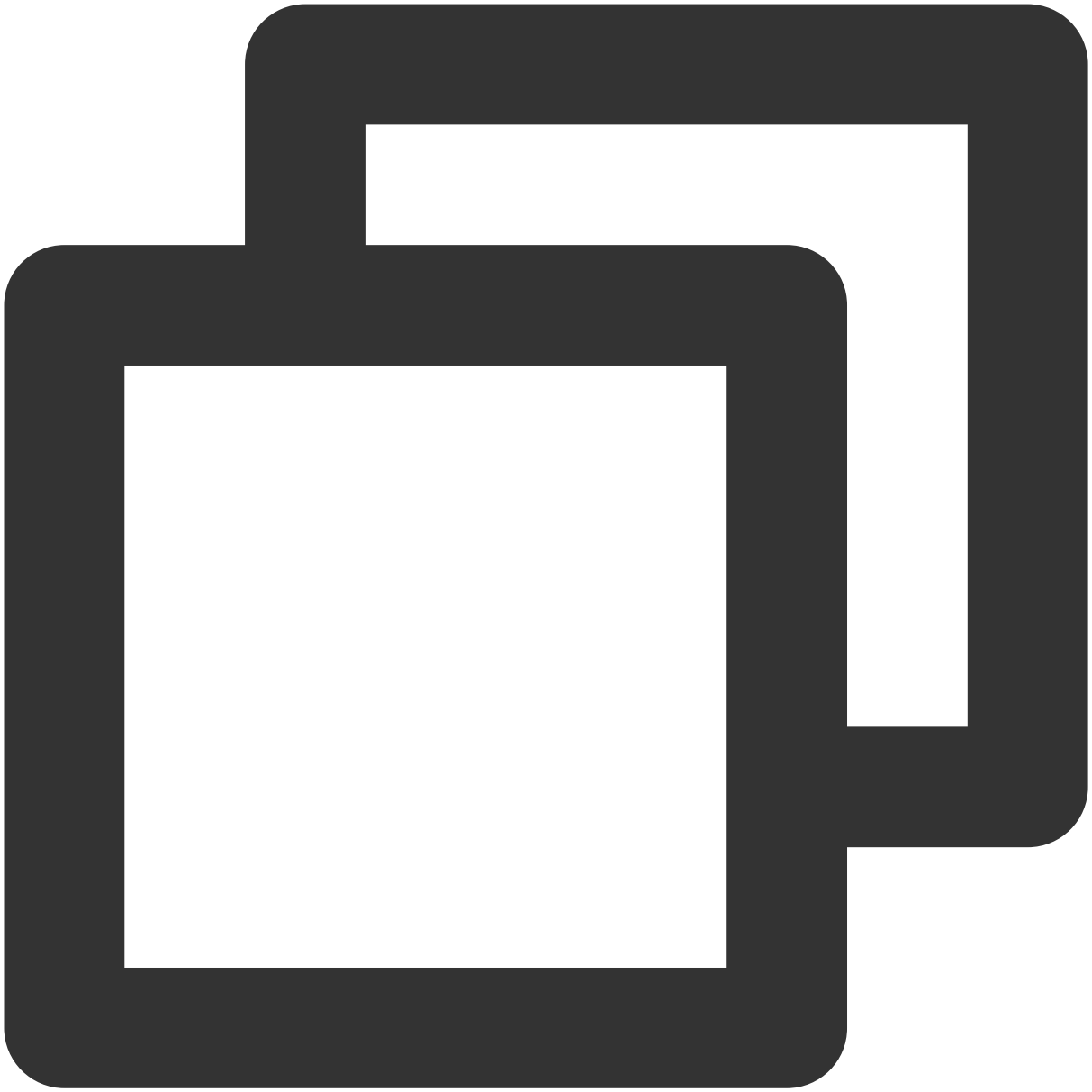
```
    metadata:
      labels:
        app: httpserver
    spec:
      containers:
      - name: httpserver
        image: registry.imroc.cc/test/httpserver:custom-metrics
        imagePullPolicy: Always

---

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

您可以通过 Promtheus 采集规则 或 ServiceMonitor 配置 Promtheus 采集业务暴露的监控指标。

方式1：配置 Promtheus 采集规则

在 Promtheus 的采集规则配置文件中添加以下采集规则。示例如下：

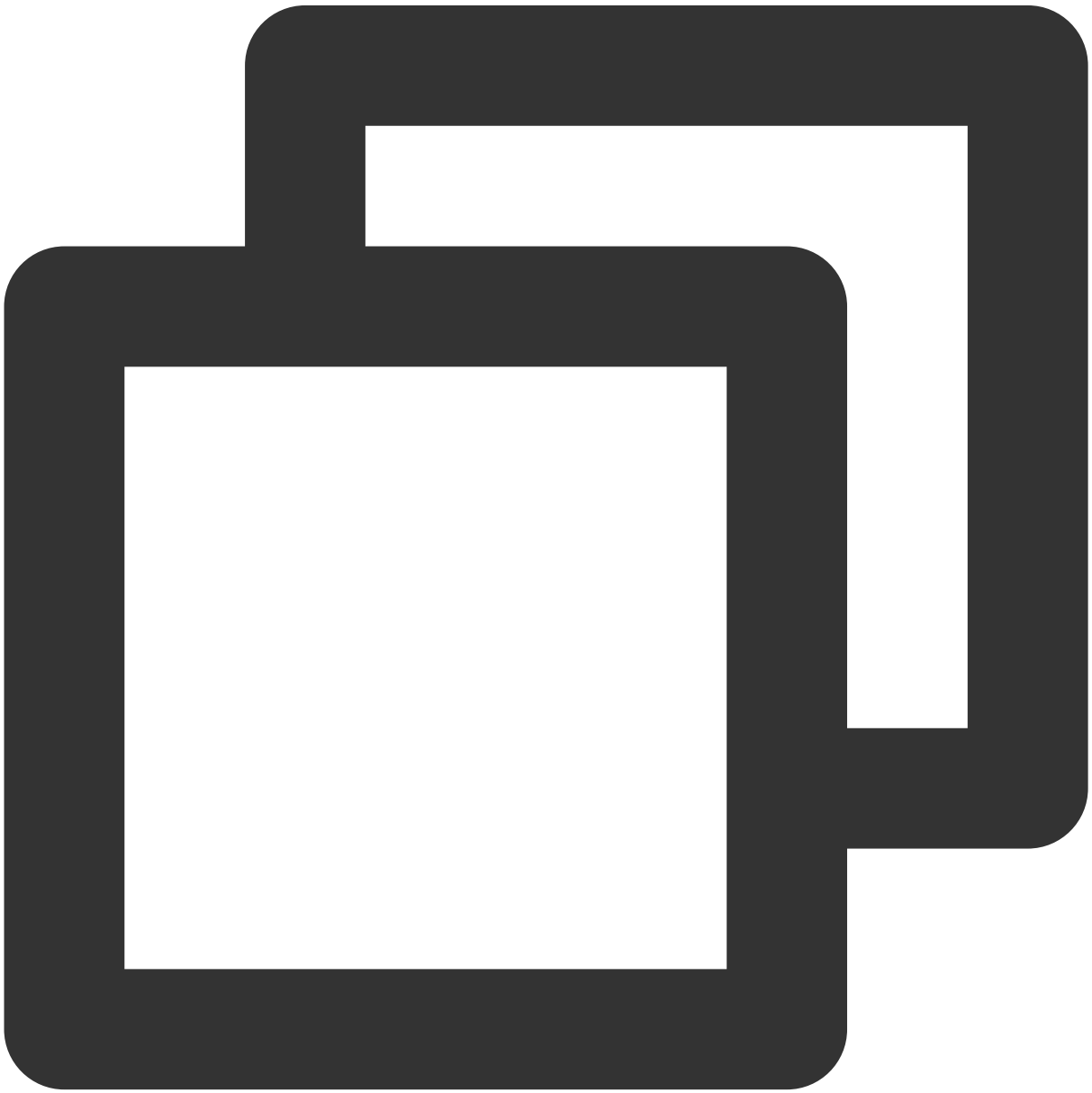


```
- job_name: httpserver
    scrape_interval: 5s
    kubernetes_sd_configs:
    - role: endpoints
        namespaces:
            names:
            - httpserver
    relabel_configs:
    - action: keep
        source_labels:
        - __meta_kubernetes_service_label_app
```

```
            regex: httpserver
        - action: keep
            source_labels:
            - __meta_kubernetes_endpoint_port_name
            regex: http
```

方式2：配置 ServiceMonitor

若已安装 prometheus-operator，可以通过创建 ServiceMonitor 的 CRD 对象配置 Prometheus。示例如下：

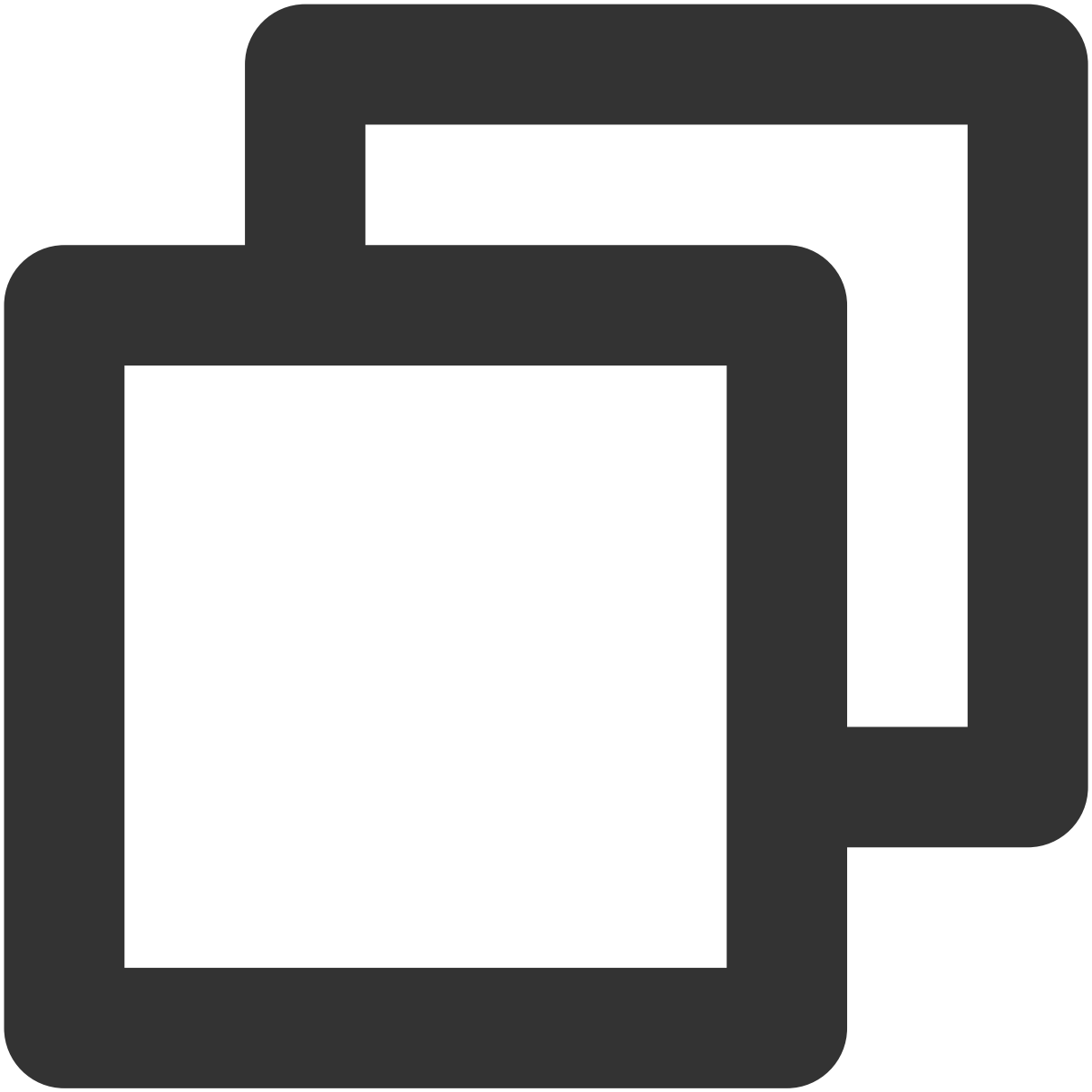


```
apiVersion: monitoring.coreos.com/v1
```

```
kind: ServiceMonitor
metadata:
  name: httpserver
spec:
  endpoints:
  - port: http
    interval: 5s
  namespaceSelector:
    matchNames:
    - httpserver
  selector:
    matchLabels:
      app: httpserver
```

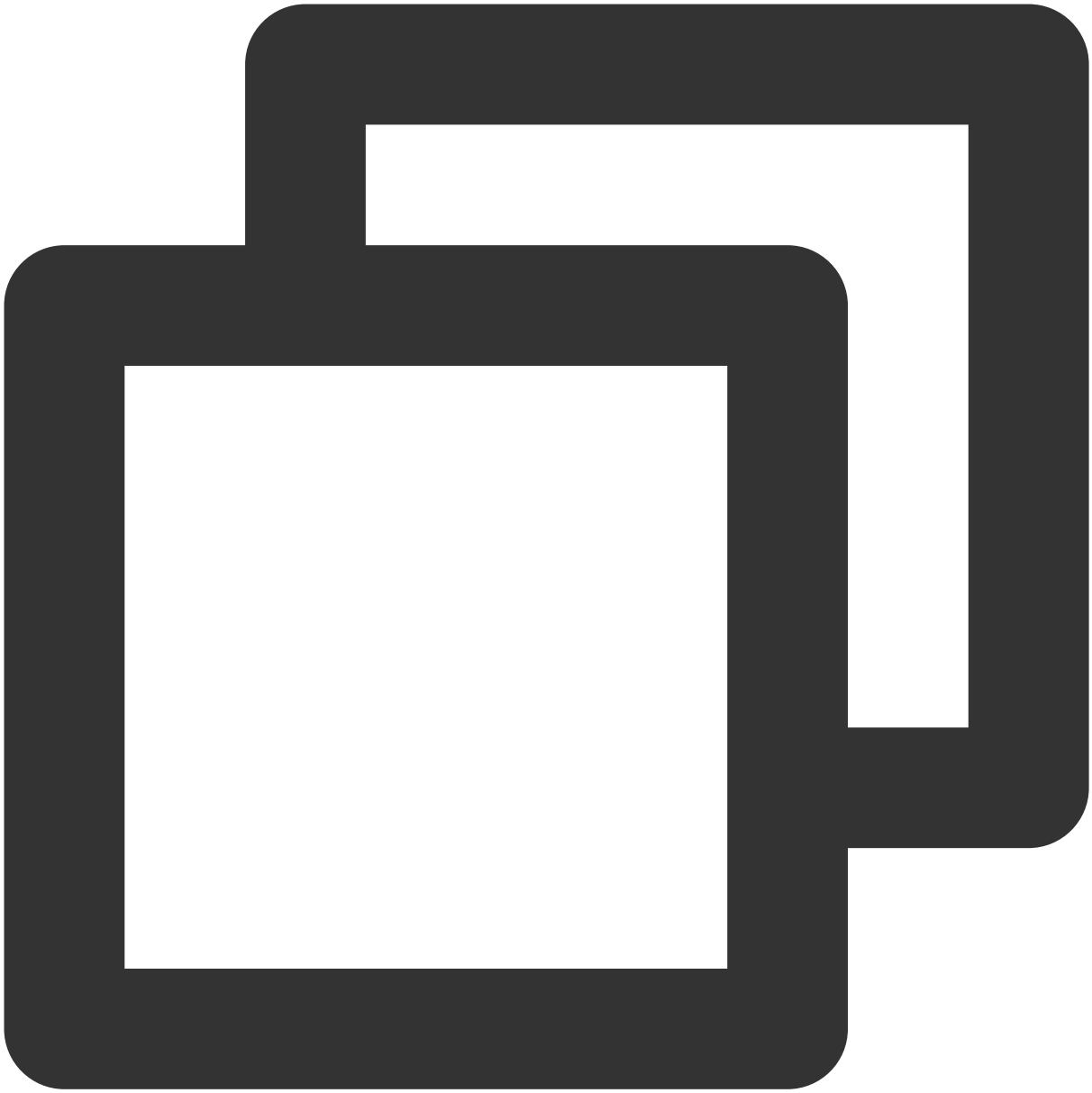
安装 prometheus-adapter

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：



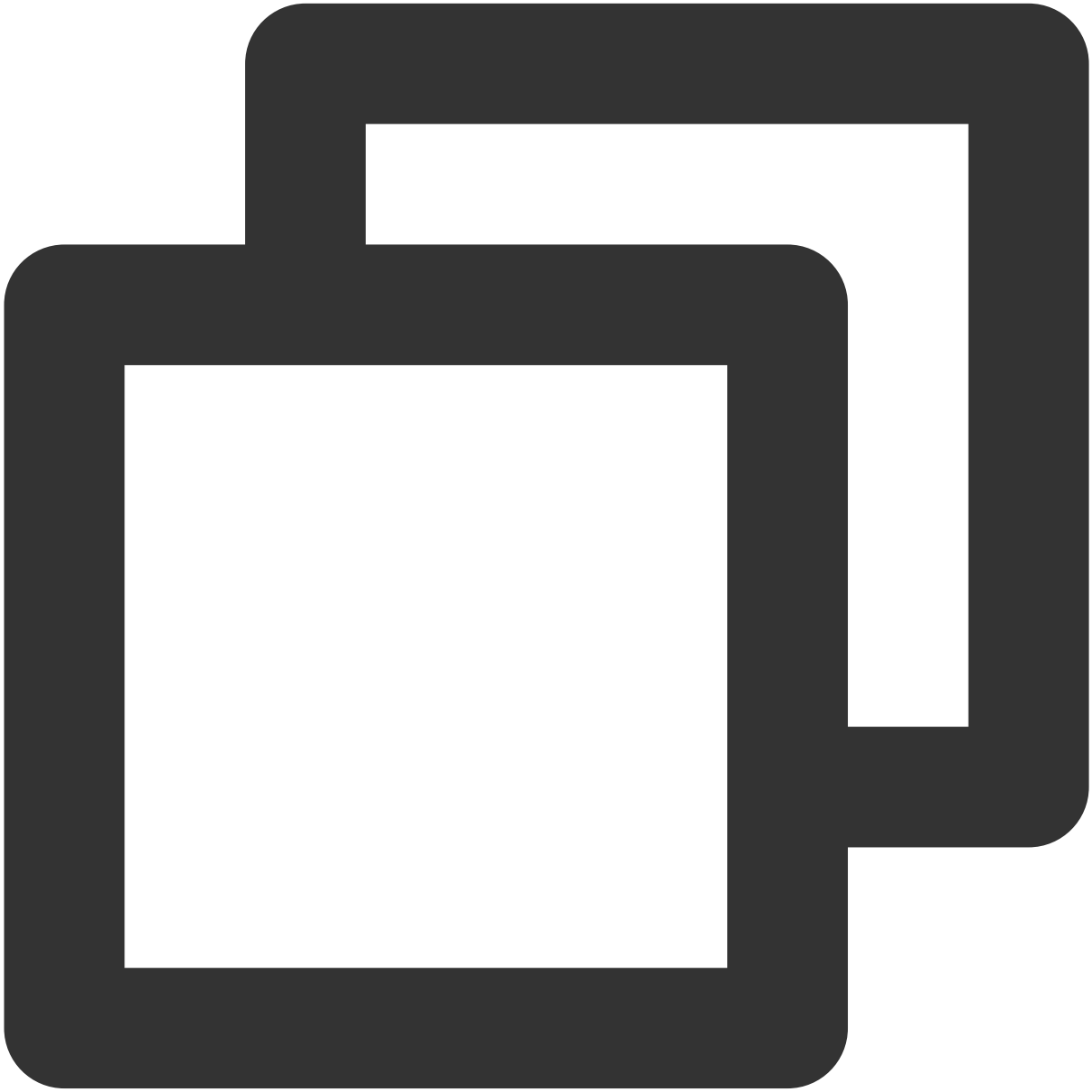
```
rules:
   default: false
   custom:
   - seriesQuery: 'httpserver_requests_total'
     resources:
       template: <<.Resource>>
     name:
       matches: "httpserver_requests_total"
       as: "httpserver_requests_qps" # PromQL 计算出来的 QPS 指标
     metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)
prometheus:
```

```
    url: http://prometheus.monitoring.svc.cluster.local # 替换 Prometheus API 的地址 (
    port: 9090
```

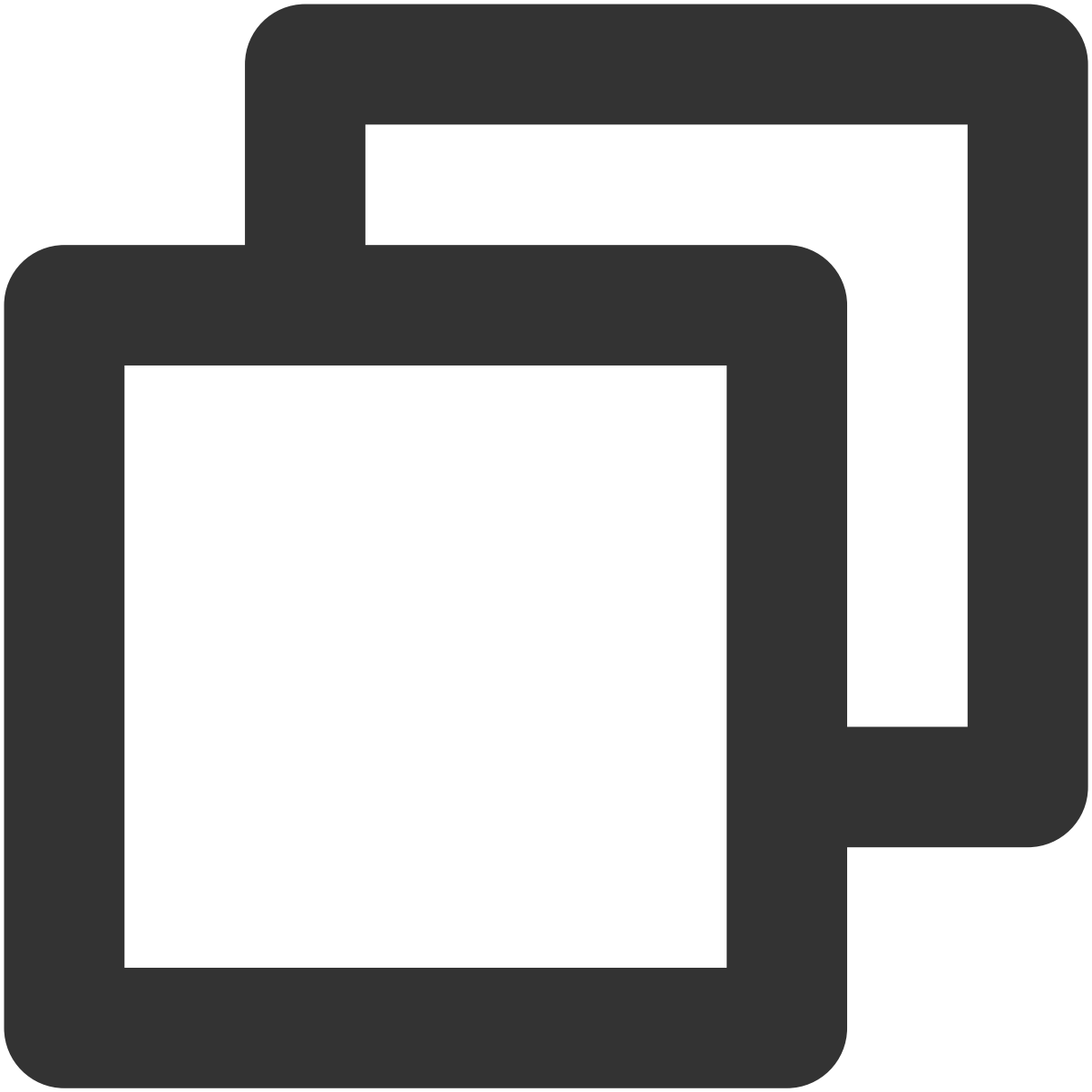
3. 执行以下 Helm 命令安装 prometheus-adapter，示例如下：

注意

安装前需要删除 TKE 已经注册的 Custom Metrics API，删除命令如下：



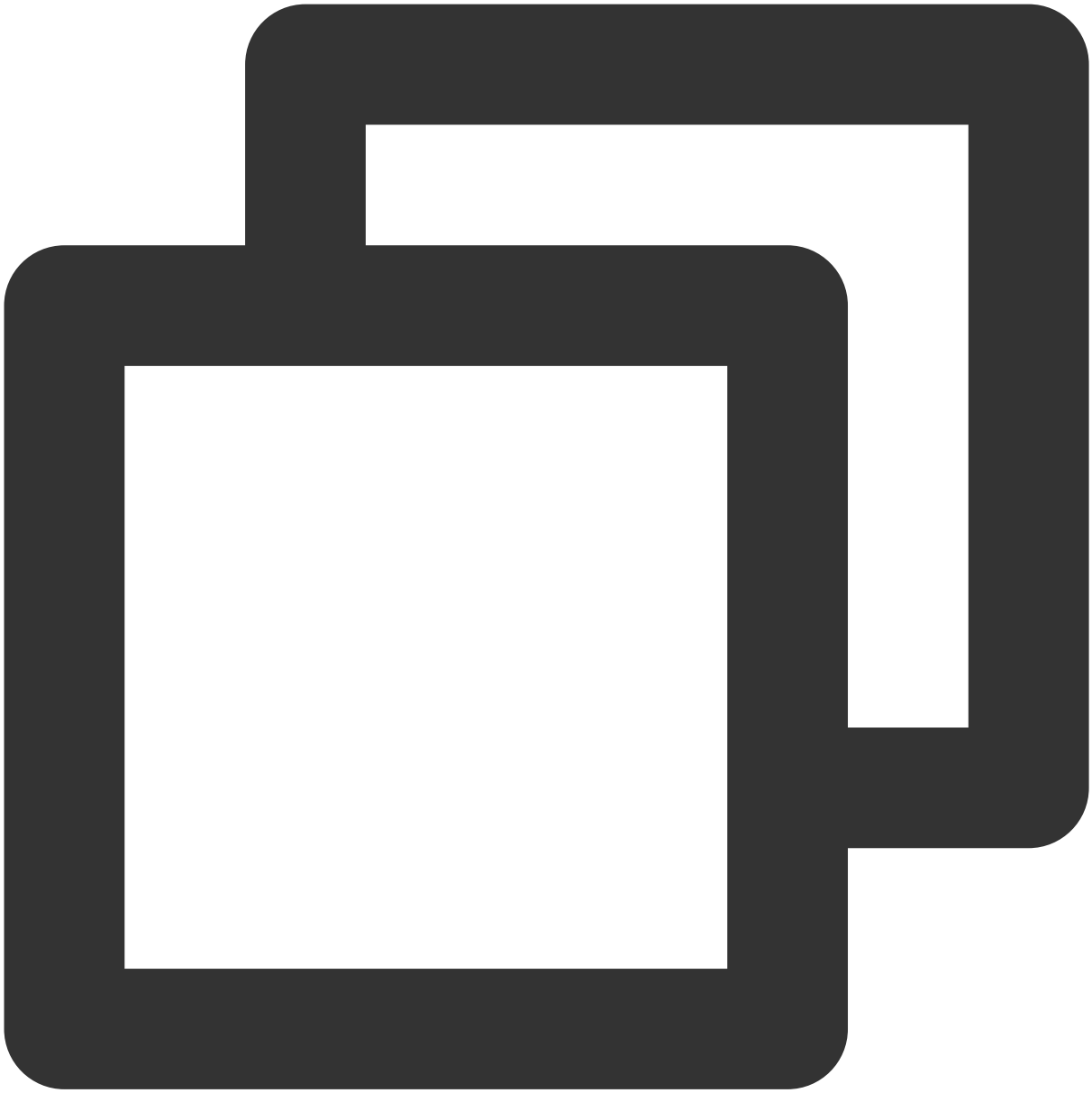
```
kubectl delete apiservice v1beta1.custom.metrics.k8s.io
```



```
helm repo add prometheus-community https://prometheus-community.github.io/helm-char
helm repo update
# Helm 3
helm install prometheus-adapter prometheus-community/prometheus-adapter -f values.y
# Helm 2
# helm install --name prometheus-adapter prometheus-community/prometheus-adapter -f
```

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：



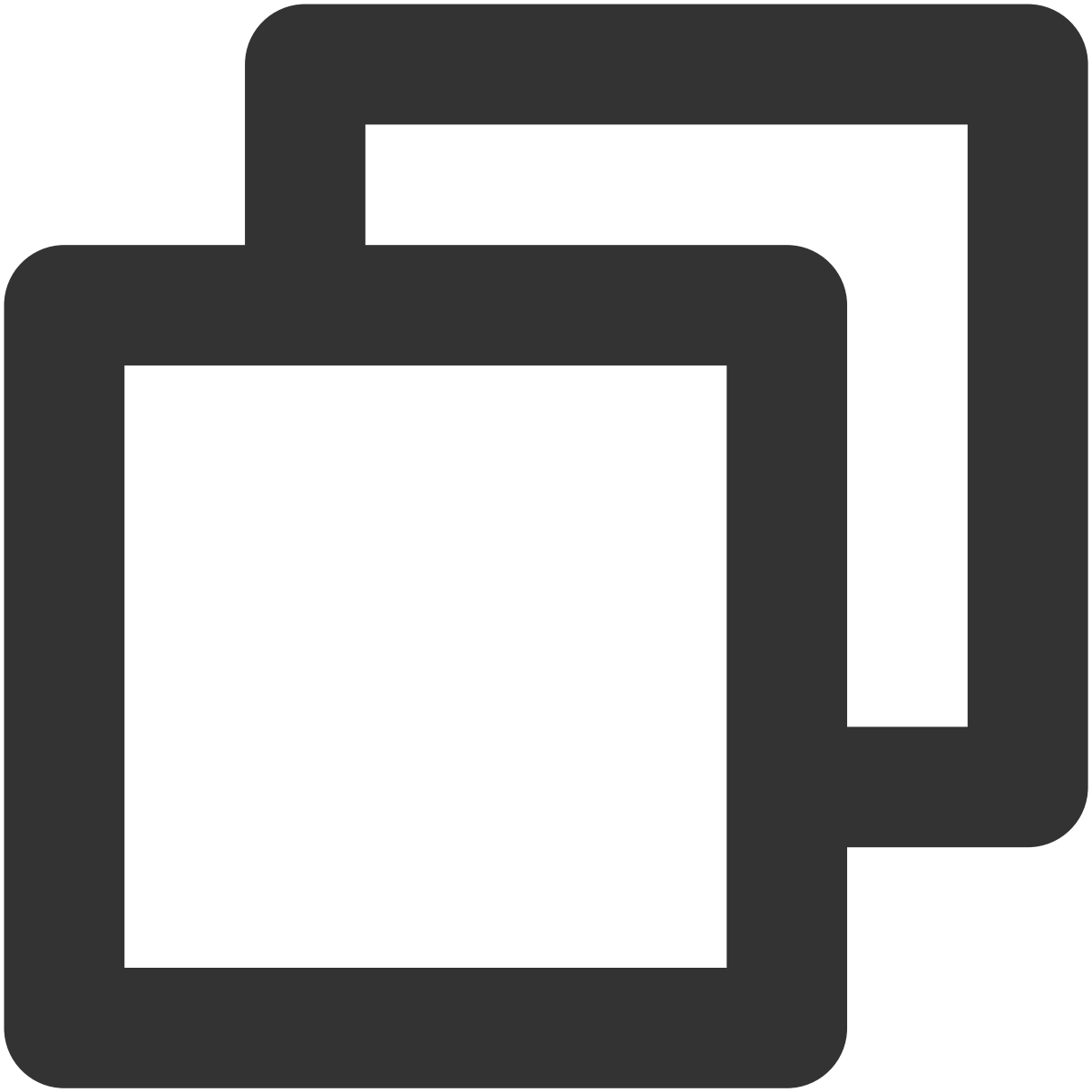
```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "jobs.batch/httpserver_requests_qps",
      "singularName": "",
      "namespaced": true,
      "kind": "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，可以查看到 Pod 的 QPS 值。示例如下：

说明

下述示例 QPS 为500m，表示 QPS 值为0.5。

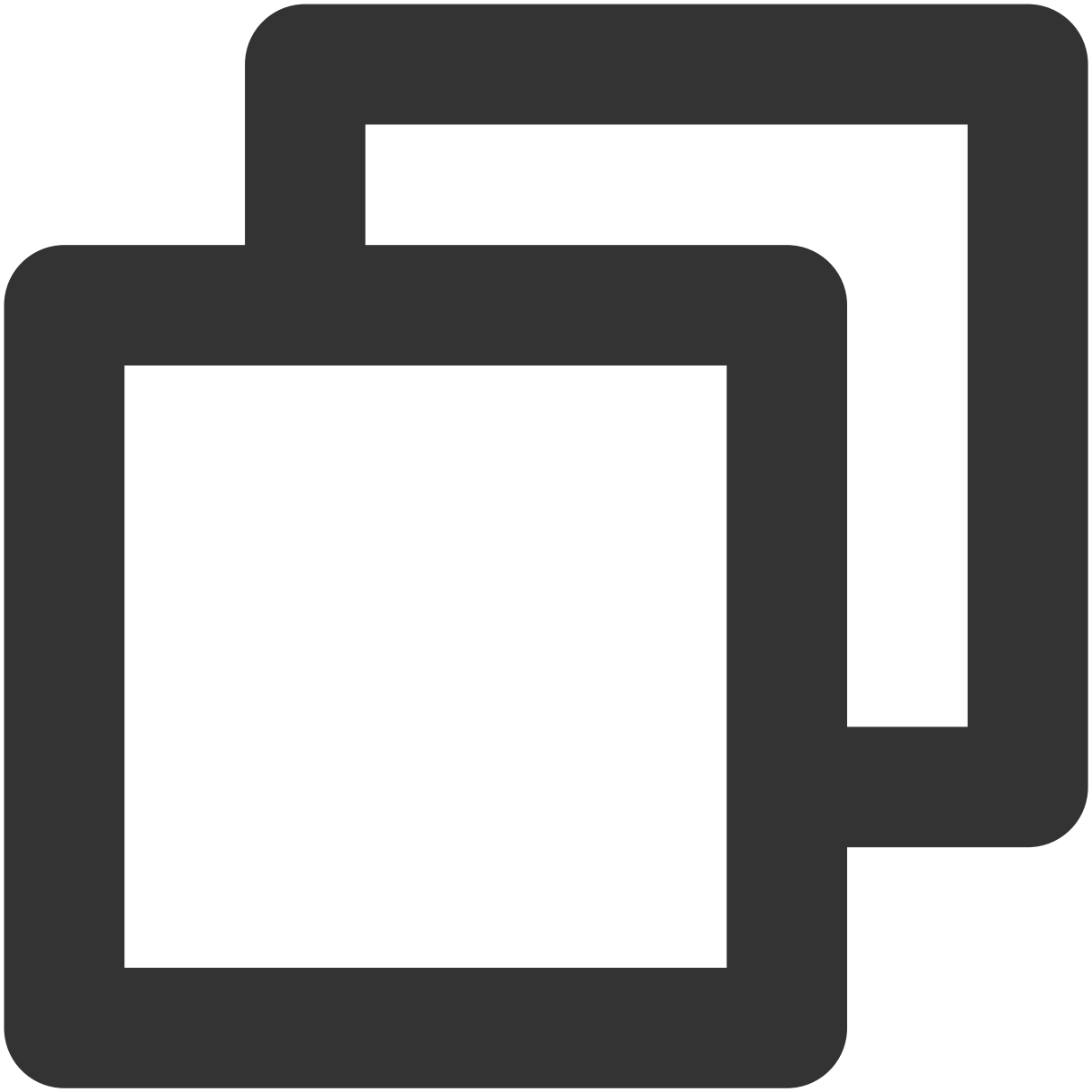


```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1/namespaces/httpserver/pods/
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/httpserver/pods/%2A
  },
  "items": [
    {
      "describedObject": {
        "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

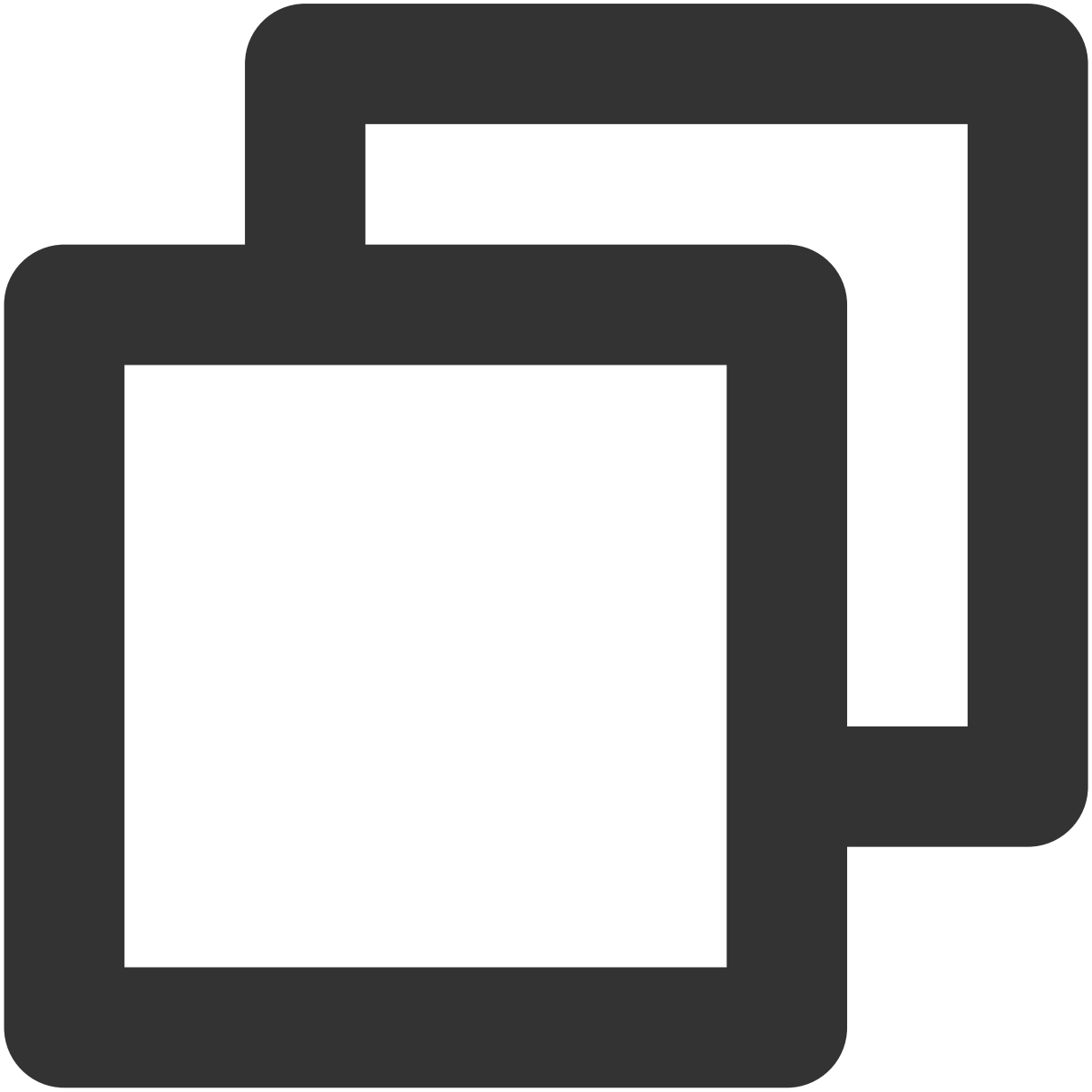
假如设置每个业务 Pod 的平均 QPS 达到50时将触发扩容，最小副本为1个，最大副本为1000个，则配置示例如下：



```
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：



```
$ kubectl get hpa
NAME         REFERENCE                TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
httpserver   Deployment/httpserver   83933m/50   1         1000      2          18h
$ kubectl get pods
NAME                          READY   STATUS              RESTARTS   AGE
httpserver-6f94475d45-47d5w   1/1     Running             0          3m41s
httpserver-6f94475d45-7rln9   1/1     Running             0          37h
httpserver-6f94475d45-6c5xm   0/1     ContainerCreating   0          1s
httpserver-6f94475d45-wl78d   0/1     ContainerCreating   0          1s
```

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：2023-05-06 17:36:46

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

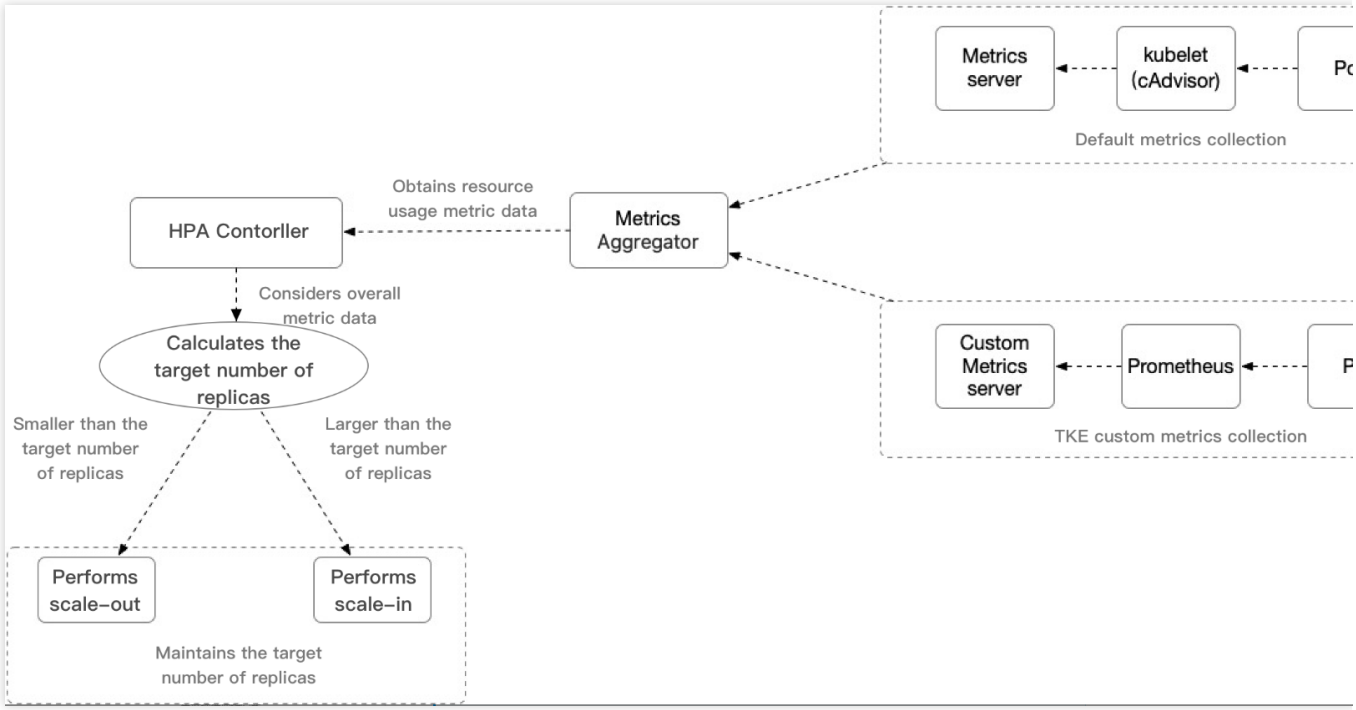
HPA 自动伸缩特性使容器服务具有非常灵活的自适应能力，能够在用户设定内快速扩容多个 Pod 副本来应对业务负载的急剧飙升，也可以在业务负载变小的情况下根据实际情况适当缩容来节省计算资源给其他的服务，整个过程自动化无须人为干预，适合服务波动较大、服务数量多且需要频繁扩缩容的业务场景，例如：电商服务、线上教育、金融服务等。

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 Aggeregator：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，了解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。

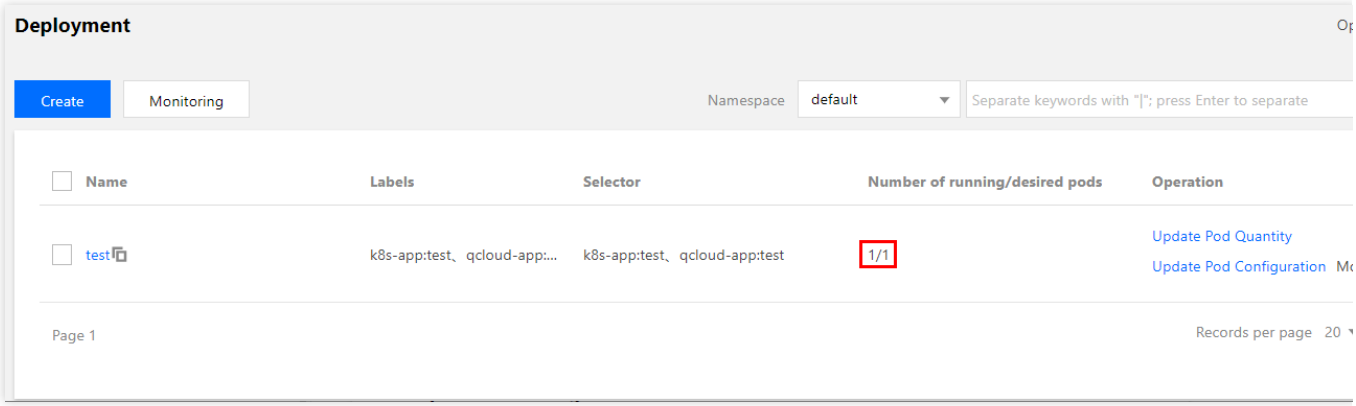
已登录 腾讯云容器服务控制台。

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

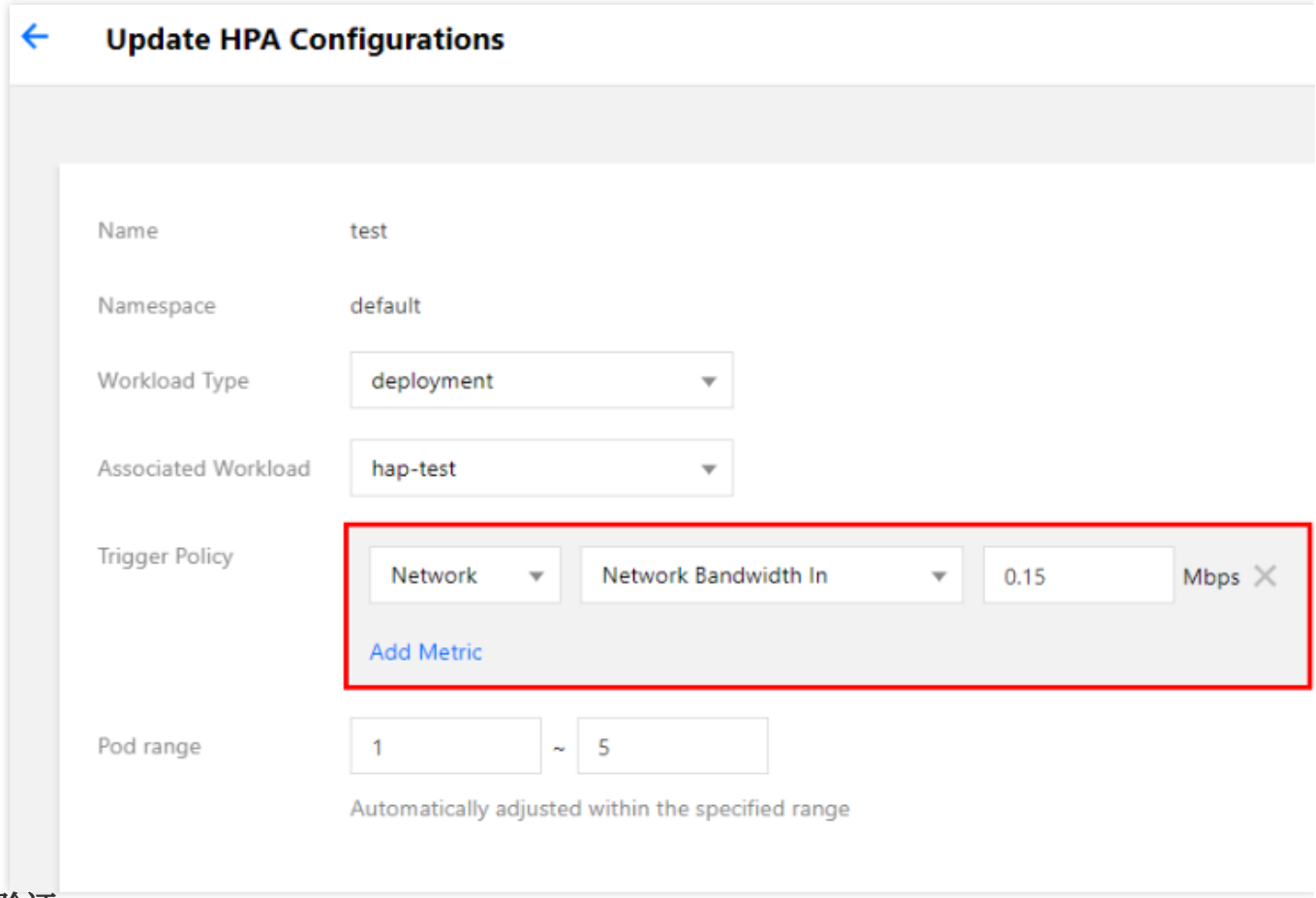
部署测试工作负载

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。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。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

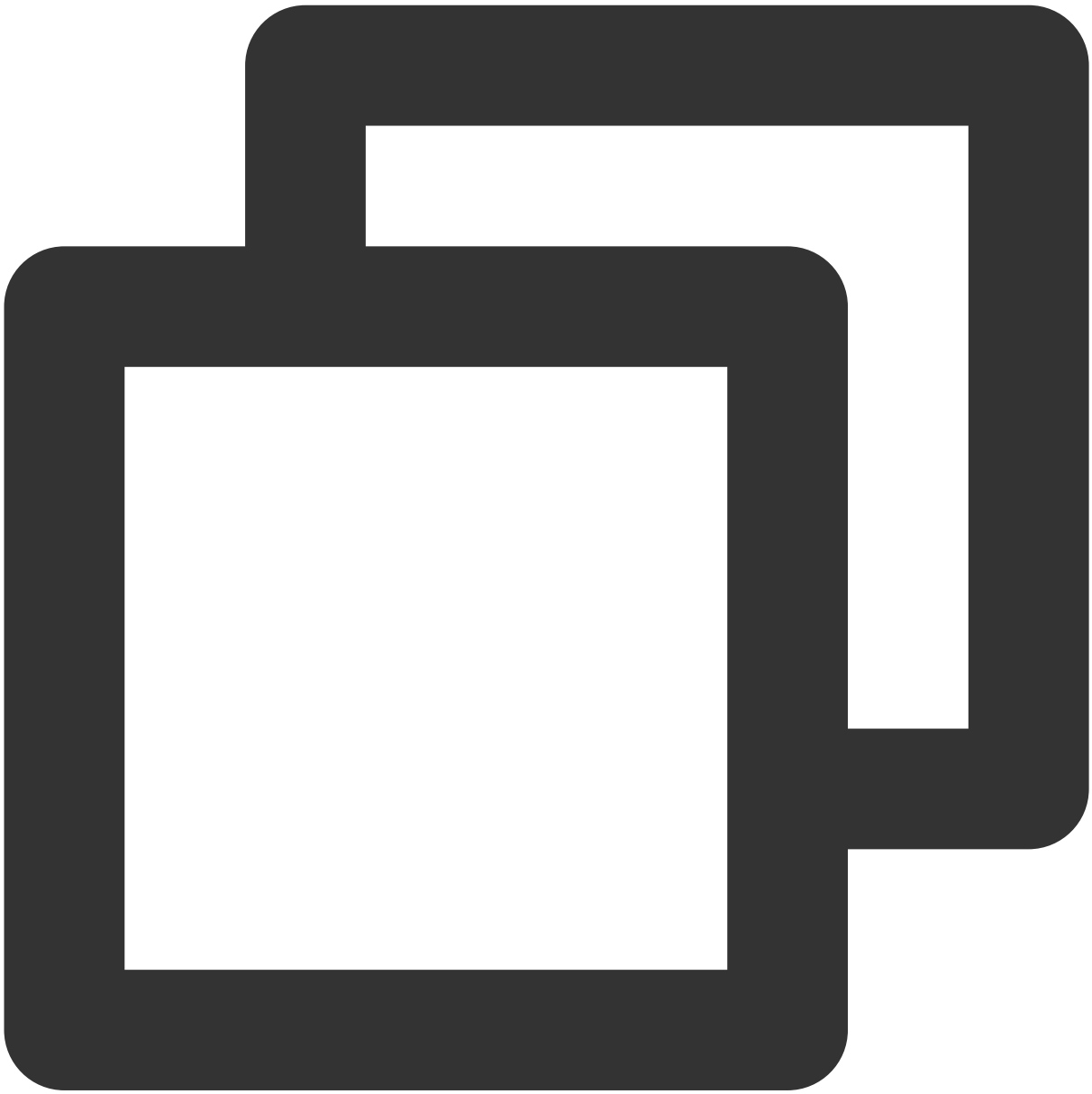
在容器服务控制台为测试工作负载绑定一个 HPA 配置，关于如何绑定配置 HPA 请参见 HPA 操作步骤，本文以配置当网络出带宽达到0.15Mbps（150Kbps）时触发扩容的策略为例。如下图所示：



功能验证

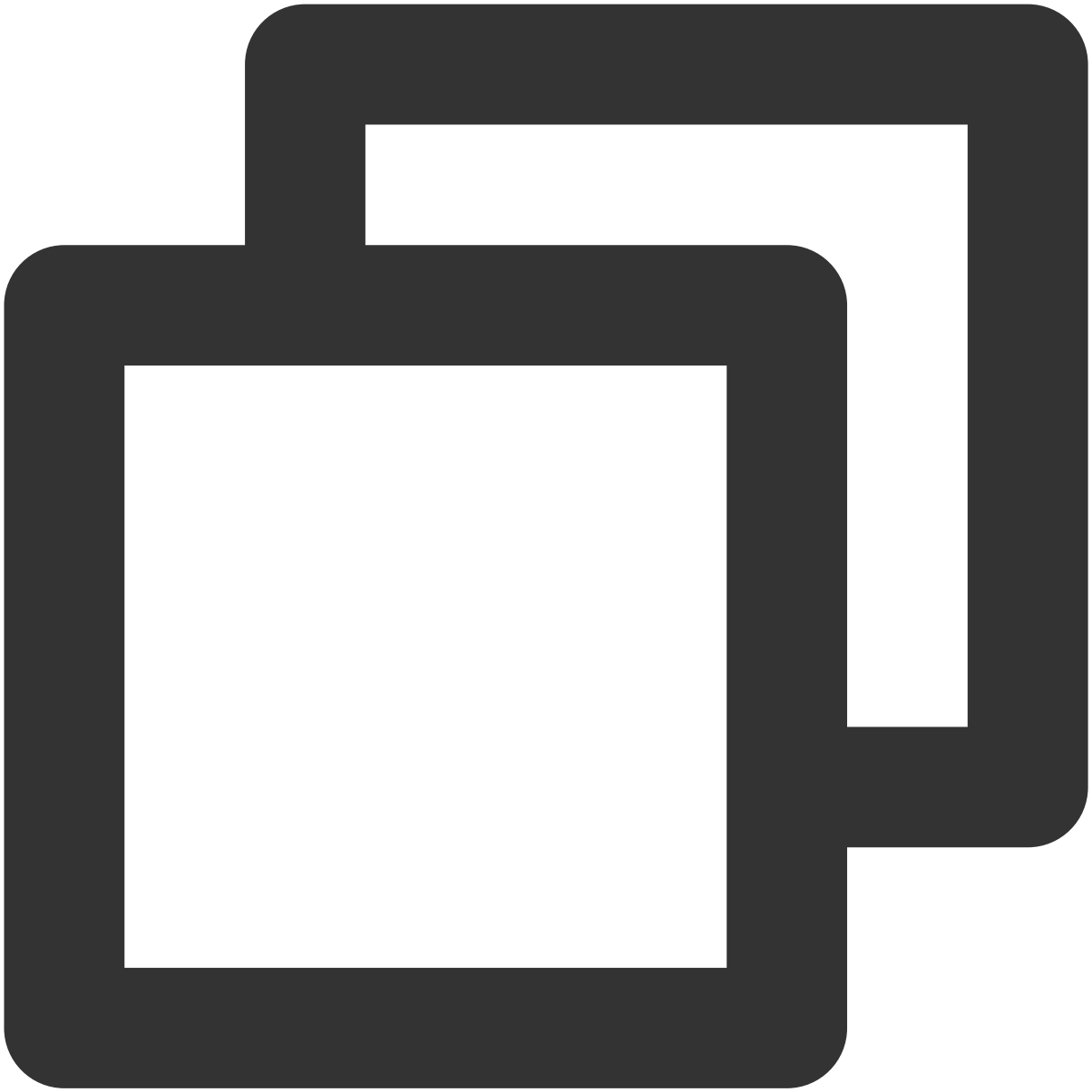
模拟扩容过程

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）：



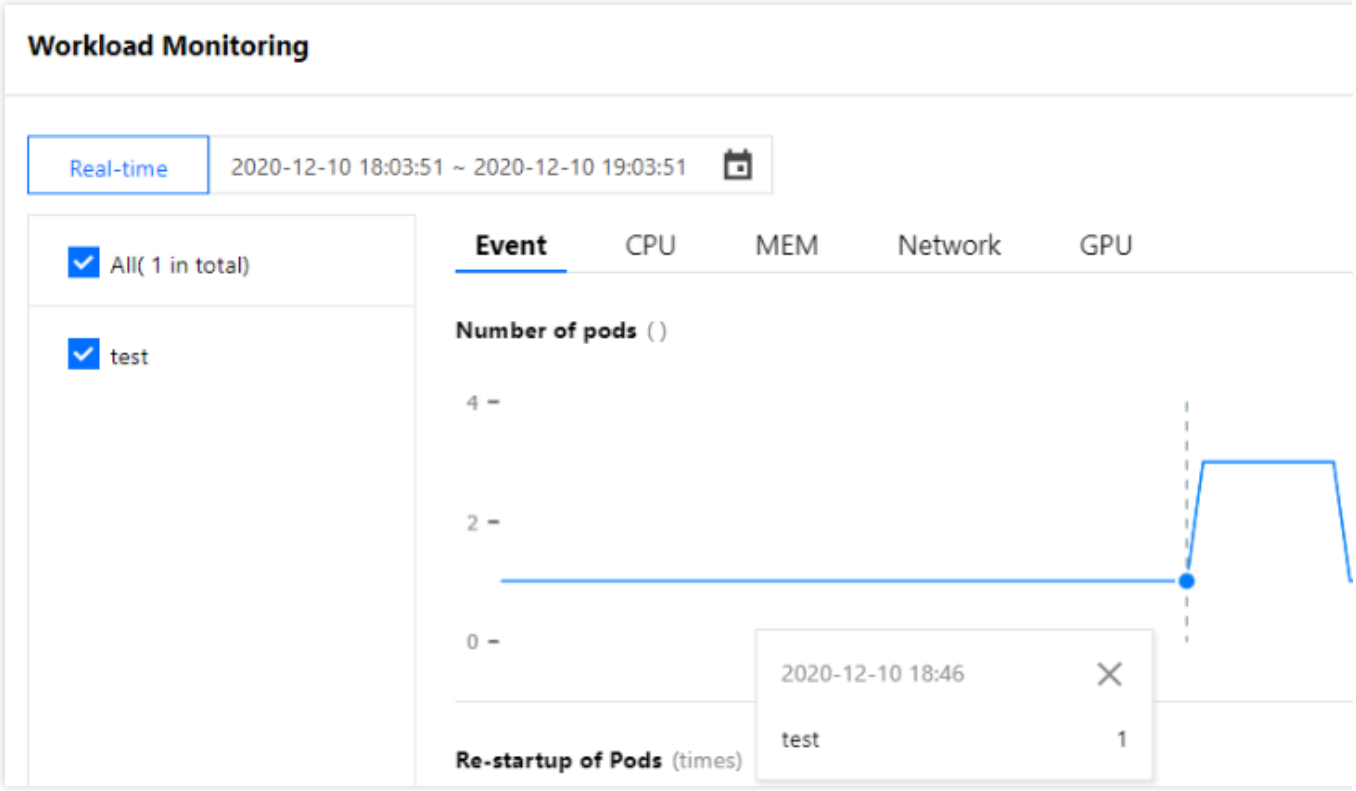
```
kubectl run -it --image alpine hpa-test --restart=Never --rm /bin/sh
```

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
```

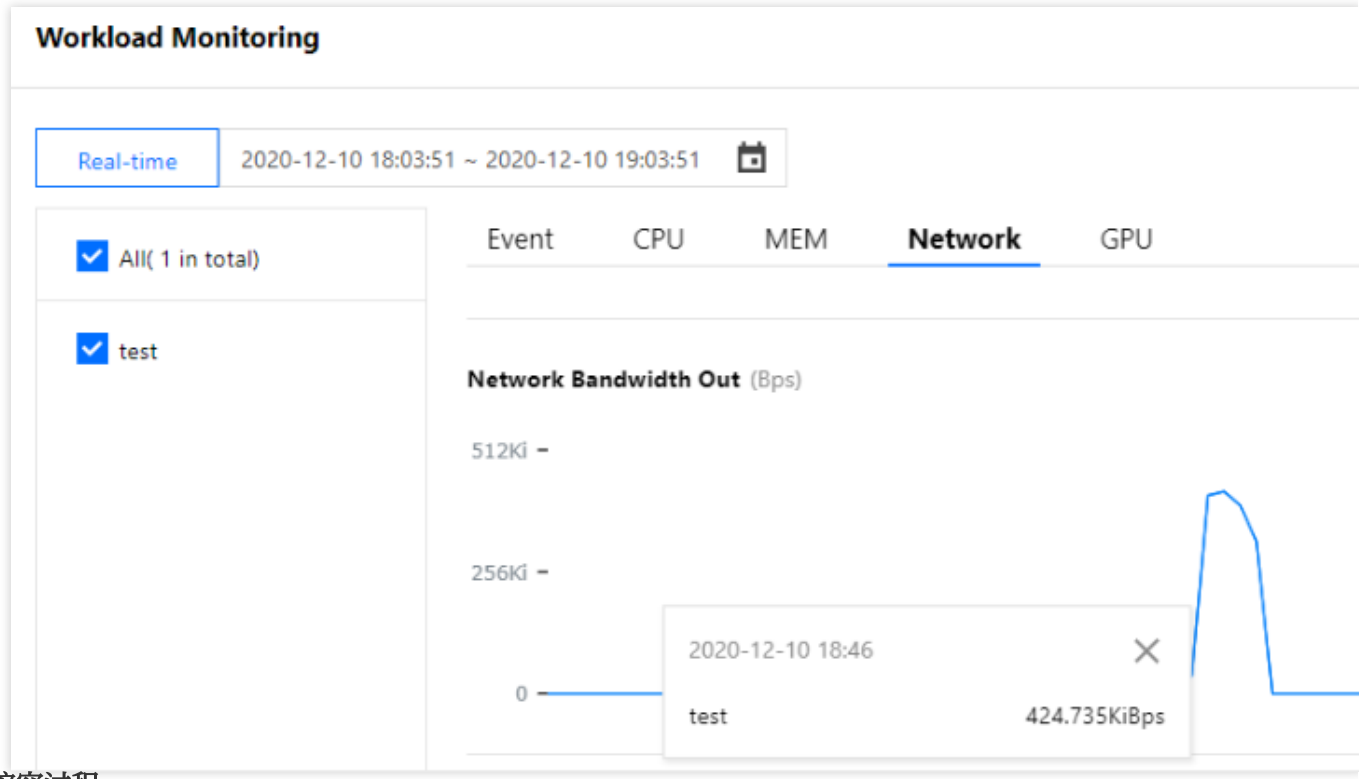
在测试 Pod 中执行模拟请求命令后，通过观察工作负载的 Pod 数量监控，发现在16:21分时工作负载扩容副本数量至2个，由此可推断出已经触发了 HPA 的扩容事件。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：



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）：

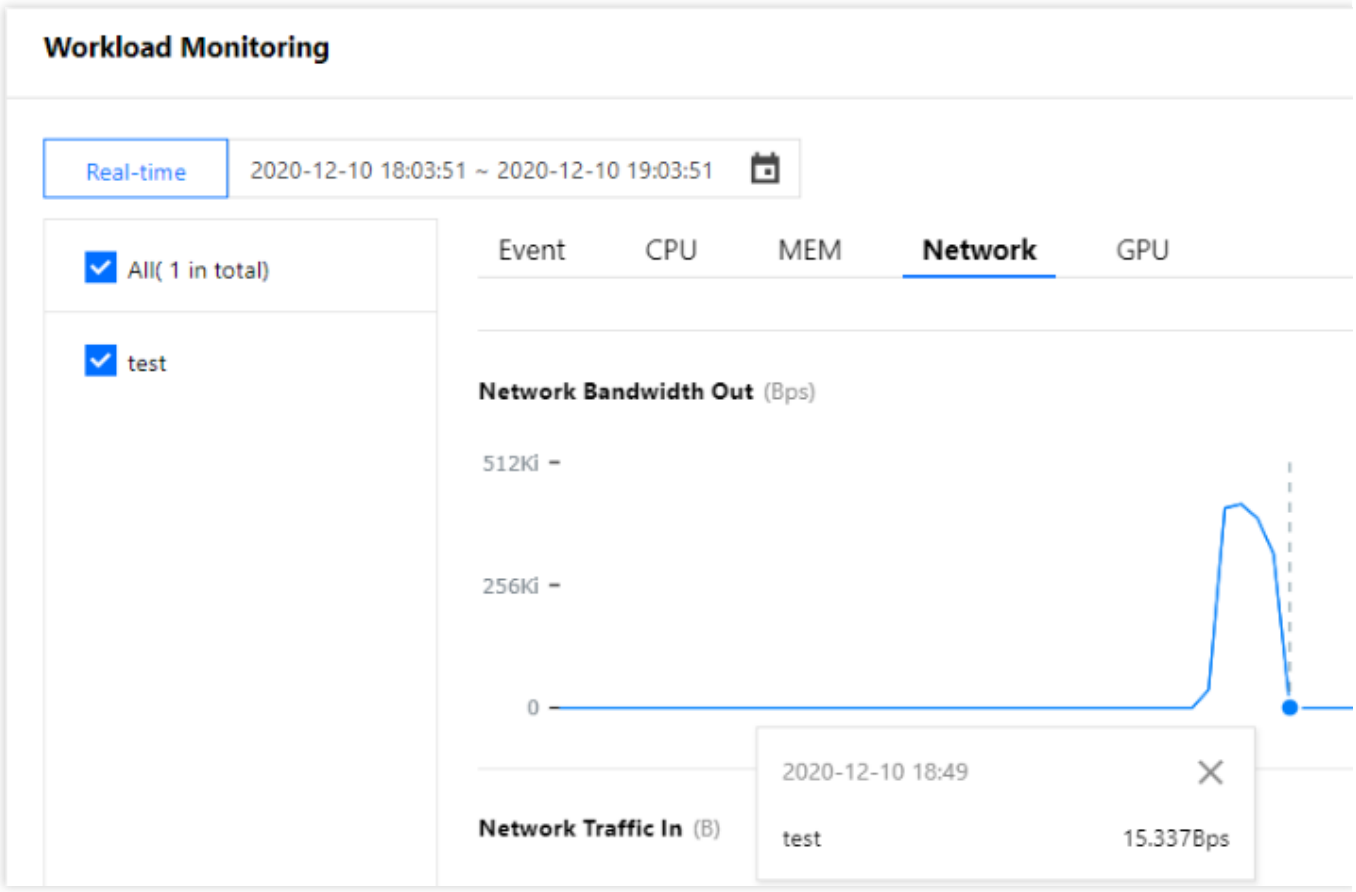
注意

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。

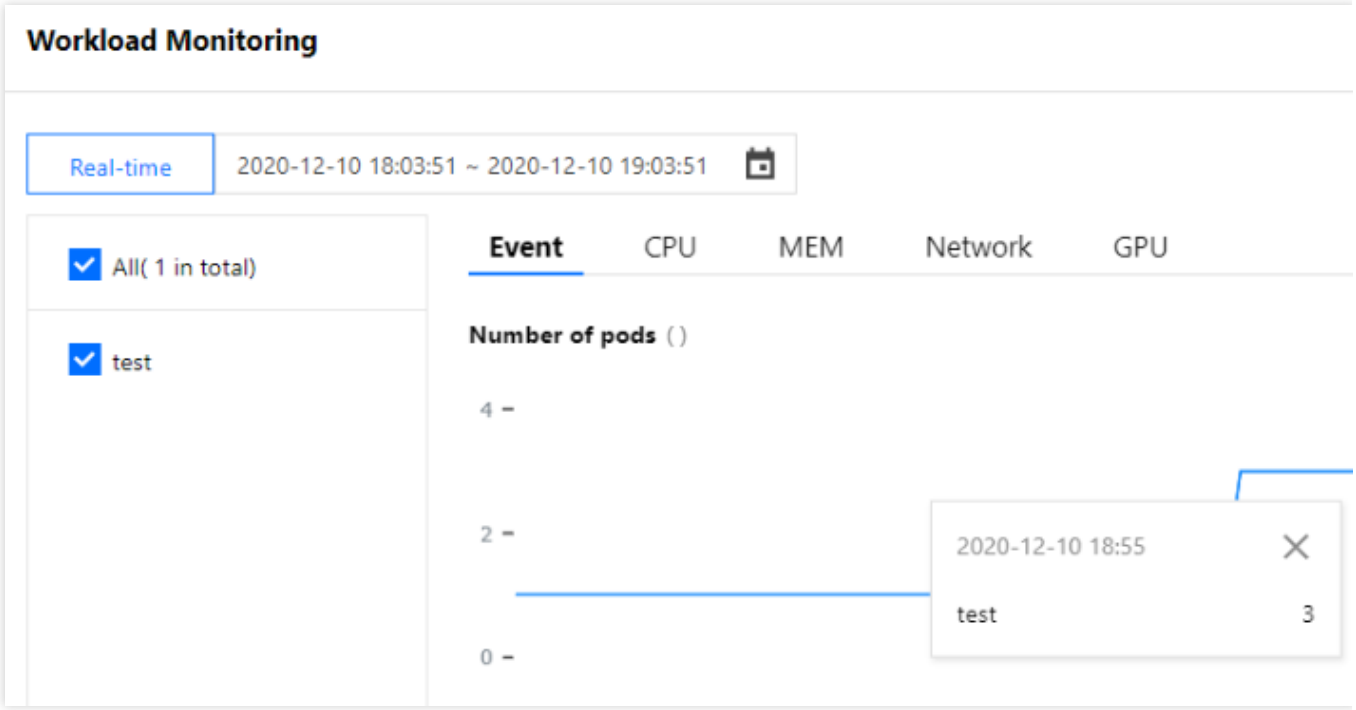


模拟缩容过程

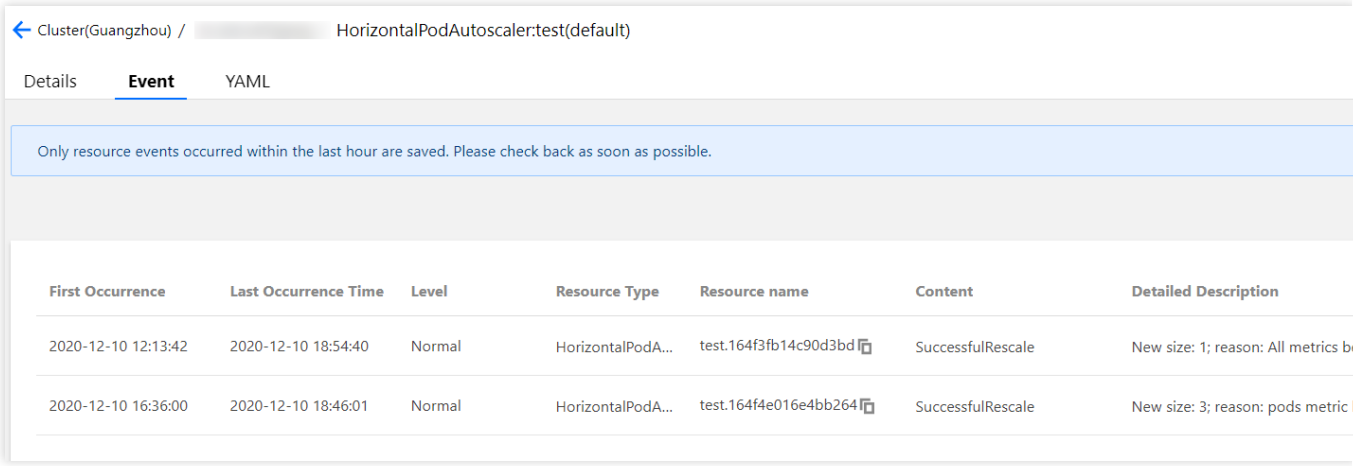
模拟缩容过程时，在16:24左右手动停止执行模拟请求的命令，从监控可以观察到此时网络出口带宽值下降到扩容前位置，按照 HPA 的逻辑，此时已经满足工作负载缩容的条件。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：



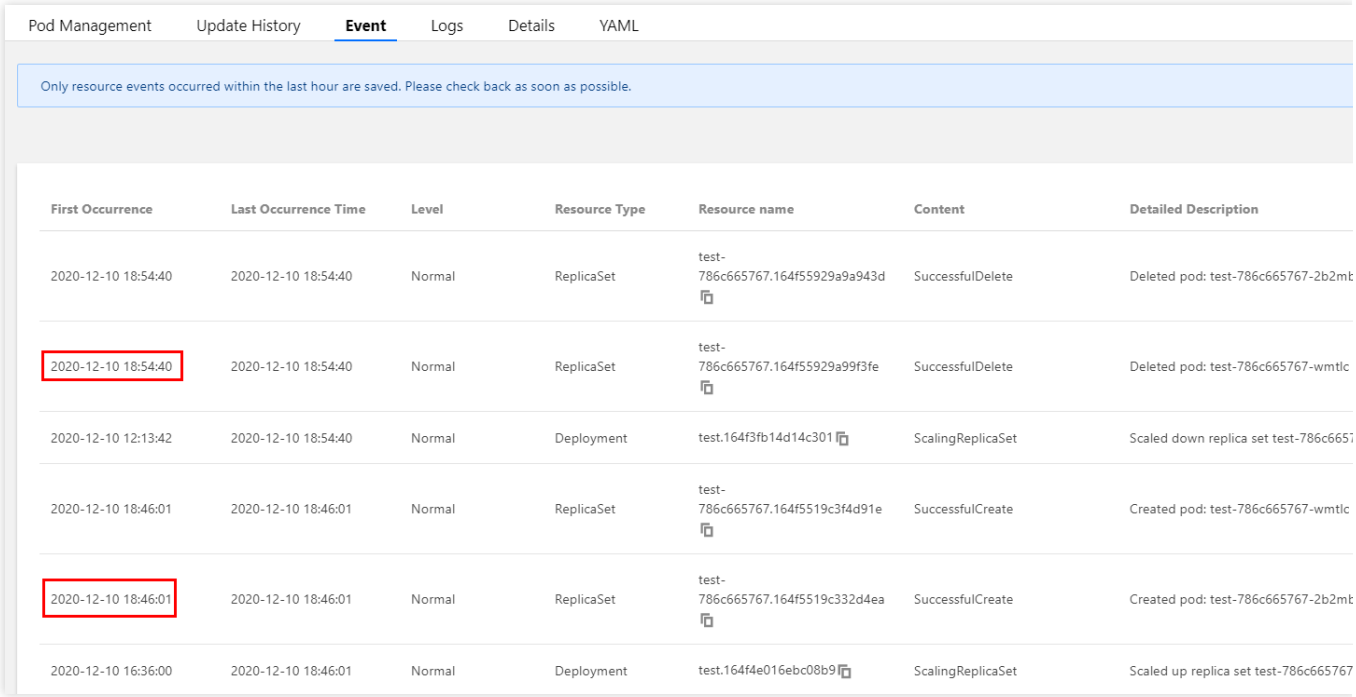
但从下图工作负载的 Pod 数量监控可以看出，工作负载在16:30分时才触发了 HPA 的缩容，原因是触发 HPA 缩容后有默认5分钟容忍的时间算法，以防止度量指标短时间波动导致的频繁的扩缩容，详情请参见 冷却 / 延迟支持。从下图可以看出工作负载副本数在停止命令5分钟后按照 HPA 扩缩容算法 缩容到了最初设定的1个副本数。如下图所示（本文的截图信息可能滞后于控制台实际界面，以控制台实际显示为准）：



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。

在 TKE 上利用 VPA 实现垂直扩缩

最近更新时间：2021-06-09 17:46:06

概述

Kubernetes Pod 垂直自动扩缩（Vertical Pod Autoscaler，以下简称 VPA）可以自动调整 Pod 的 CPU 和内存预留，帮助提高集群资源利用率并释放 CPU 和内存供其它 Pod 使用。本文介绍如何在腾讯云容器服务 TKE 上使用社区版 VPA 功能实现 Pod 垂直扩缩容。

使用场景

VPA 自动伸缩特性使容器服务具有非常灵活的自适应能力。应对业务负载急剧飙升的情况，VPA 能够在用户设定范围内快速扩大容器的 Request。在业务负载变小的情况下，VPA 可根据实际情况适当缩小 Request 节省计算资源。整个过程自动化无须人为干预，适用于需要快速扩容、有状态应用扩容等场景。此外，VPA 可用于向用户推荐更合理的 Request，在保证容器有足够使用的资源的情况下，提升容器的资源利用率。

VPA 优势

相较于 自动伸缩功能 HPA，VPA 具有以下优势：

- VPA 扩容不需要调整 Pod 副本数量，扩容速度更快。
- VPA 可为有状态应用实现扩容，HPA 则不适合有状态应用的水平扩容。
- Request 设置过大，使用 HPA 水平缩容至一个 Pod 时集群资源利用率仍然很低，此时可以通过 VPA 进行垂直缩容提高集群资源利用率。

VPA 限制

> 注意
> 社区版 VPA 功能当前处于试验阶段，请谨慎使用。推荐您将 “updateMode” 设置为 “Off”，以确保 VPA 不会自动为您更换 Request 数值。您仍然可以在 VPA 对象中查看已绑定负载的 Request 推荐值。

- 自动更新正在运行的 Pod 资源是 VPA 的一项实验功能。当 VPA 更新 Pod 资源时，会导致 Pod 的重建和重启，并且有可能被调度到其他节点上。
- VPA 不会驱逐不在控制器下运行的 Pod。对于此类 Pod，`Auto` 模式等效于 `Initial`。
- VPA 与 HPA 不可同时在 CPU 和内存预留上运行。如需同时运行 **VPA** 与 **HPA**，则 HPA 需使用除 CPU 和内存以外的指标，详情可参见 在 TKE 上使用自定义指标进行弹性伸缩。
- VPA 使用 Admission Webhook 作为其准入控制器。如果集群中存在其他的 Admission Webhook，需要确保它们不会与 VPA 的 Admission Webhook 发生冲突。准入控制器的执行顺序定义在 API Server 的配置参数中。
- VPA 会处理大多数 OOM（Out Of Memory）事件。
- VPA 性能尚未在大型群集中进行测试。
- VPA 对 Pod 资源 Request 的建议值可能会超出可用资源（例如节点资源上限、空闲资源或资源配额），并导致 Pod 处于 Pending 状态无法被调度。同时使用 VPA 与 Cluster Autoscaler 可以部分解决此问题。
- 与同一个 Pod 匹配的多个 VPA 资源具有未定义的行为。

更多 VPA 限制请参见 VPA Known limitations。

前提条件

- 已创建容器服务 TKE 集群。
- 已使用命令行工具 Kubectl 连接集群。如果您还未连接集群，请参考 连接集群。

操作步骤

部署 VPA

1. 登录集群中的云服务器。
2. 通过命令行工具 Kubectl 从本地客户端机器连接到 TKE 集群。
3. 执行以下命令，克隆 kubernetes/autoscaler GitHub Repository。

```
sh
git clone https://github.com/kubernetes/autoscaler.git
```

4. 执行以下命令，切换至 `vertical-pod-autoscaler` 目录。

```
cd autoscaler/vertical-pod-autoscaler/
```

5. （可选）如果您已经部署其他版本的 VPA，执行以下命令将其删除。否则将会产生异常影响。

```
./hack/vpa-down.sh
```

6. 执行以下命令，将 VPA 相关组件部署到您的集群。

```
./hack/vpa-up.sh
```

7. 执行以下命令，验证是否成功创建 VPA 组件。

```
kubectl get deploy -n kube-system | grep vpa
```

成功创建 VPA 组件后，您可在 kube-system 命名空间中查阅三个 Deployment，分别为 vpa-admission-controller、vpa-recommender、vpa-updater。如下图所示：

```
[root@VM-22-114-centos hack]# kubectl get deploy -n kube-system | grep vpa
vpa-admission-controller                1/1     1            1           17s
vpa-recommender                         1/1     1            1           22h
vpa-updater                             1/1     1            1           22h
```

示例1：使用 VPA 获取 Request 推荐值

说明：

- 不建议在生产环境中使用 VPA 自动更新 Request。
- 您可以利用 VPA 查看 Request 推荐值，在合适条件下手动触发更新。

在本示例中，您将创建 `updateMode` 为 `Off` 的 VPA 对象，并创建具有两个 Pod 的 Deployment，每个 Pod 各有一个容器。在创建 Pod 后，VPA 会分析容器的 CPU 和内存需求，并在 `status` 字段中记录 Request 推荐值。VPA 不会自动更新正在运行的容器的资源请求。

在终端中执行以下命令，生成一个名为 `tke-vpa` 的 VPA 对象，指向一个名为 `tke-deployment` 的 Deployment：

```
shell
cat <<EOF | kubectl apply -f -
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
name: tke-vpa
spec:
```

```
targetRef:
apiVersion: "apps/v1"
kind: Deployment
name: tke-deployment
updatePolicy:
updateMode: "Off"
EOF
```

执行以下命令，生成一个名为 `tke-deployment` 的 Deployment 对象：

```
shell
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
name: tke-deployment
spec:
replicas: 2
selector:
matchLabels:
app: tke-deployment
template:
metadata:
labels:
app: tke-deployment
spec:
containers:
- name: tke-container
image: nginx
EOF
```

生成的 Deployment 对象如下图所示：

```
[root@VM-22-114-centos ~]# kubectl get deploy,vpa | grep tke
deployment.apps/tke-deployment   2/2     2            2           7m1s
verticalpodautoscaler.autoscaling.k8s.io/tke-vpa   Off    25m    262144k   True       66s
```

注意：

上述操作创建 `tke-deployment` 时并没有设置 CPU 或内存的 Request，Pod 中的 Qos 为 BestEffort，此时 Pod 容易被驱逐。建议您在创建业务的 Deployment 时设置 Request 及 Limit。如果您通过容器服务控制台

创建工作负载，控制台将自动为每个容器的 Request 和 Limits 设置默认值。

执行以下命令，您可以查看 VPA 推荐的 CPU 和内存 Request：

```shell
kubectl get vpa tke-vpa -o yaml
```

执行结果如下所示：

```yaml
...
recommendation:
containerRecommendations:
- containerName: tke-container
lowerBound:
cpu: 25m
memory: 262144k
target: # 推荐值
cpu: 25m
memory: 262144k
uncappedTarget:
cpu: 25m
memory: 262144k
upperBound:
cpu: 1771m
memory: 1851500k
```

其中 `target` 对应的 CPU 和内存为推荐 Request。您可以选择删除之前的 Deployment，并使用推荐的 Request 值创建新的 Deployment。

字段	释义

字段	释义
lowerBound	推荐的最小值。使用小于该值的 Request 可能会对性能或可用性产生重大影响。
target	推荐值。由 VPA 计算出最合适的 Request。
uncappedTarget	最新建议值。仅基于实际资源使用情况，不考虑 `.spec.resourcePolicy.containerPolicies` 中设置的容器可以被推荐的数值范围。uncappedTarget 可能与推荐上下界限不同。该字段仅用作状态指示，不会影响实际的资源分配。
upperBound	推荐的最大值。使用高于该值的 Request 可能造成浪费。

示例2：停用特定容器

如果您的 Pod 中有多个容器，例如一个是真正的业务容器，另一个是辅助容器。为了节省集群资源，您可以选择停止为辅助容器推荐 Request。

在示例中，您将创建一个停用了特定容器的 VPA，并创建一个 Deployment。Deployment 中包含一个 Pod，该 Pod 内包含两个容器。在创建 Pod 后，VPA 仅为一个容器创建并计算推荐值，另外一个容器被停用 VPA 的推荐能力。

在终端中执行以下命令，生成一个名为 `tke-opt-vpa` 的 VPA 对象，指向一个名为 `tke-opt-deployment` 的 Deployment：

```
shell
cat <<EOF | kubectl apply -f -
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
name: tke-opt-vpa
spec:
targetRef:
apiVersion: "apps/v1"
kind: Deployment
name: tke-opt-deployment
updatePolicy:
updateMode: "Off"
resourcePolicy:
containerPolicies:
- containerName: tke-opt-sidecar
mode: "Off"
EOF
```

注意：

该 VPA 的 `.spec.resourcePolicy.containerPolicies` 中，指定了 `tke-opt-sidecar` 的 `mode` 为 “Off”，VPA 将不会为 `tke-opt-sidecar` 计算和推荐新的 Request。

执行以下命令，生成一个名为 `tke-deployment` 的 Deployment 对象：

```
sh
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
name: tke-opt-deployment
spec:
replicas: 1
selector:
matchLabels:
app: tke-opt-deployment
template:
metadata:
labels:
app: tke-opt-deployment
spec:
containers:
- name: tke-opt-container
image: nginx
- name: tke-opt-sidecar
image: busybox
command: ["sh","-c","while true; do echo TKE VPA; sleep 60; done"]
EOF
```

生成的 Deployment 对象如下图所示：

执行以下命令，您可以查看 VPA 推荐的 CPU 和内存 Request：

```
shell
kubectl get vpa tke-opt-vpa -o yaml
```

执行结果如下所示：

```
yaml
...
recommendation:
containerRecommendations:
- containerName: tke-opt-container
lowerBound:
cpu: 25m
memory: 262144k
target:
cpu: 25m
memory: 262144k
uncappedTarget:
cpu: 25m
memory: 262144k
upperBound:
cpu: 1595m
memory: 1667500k
```

在执行结果中，仅有 `tke-opt-container` 的推荐值，没有 `tke-opt-sidecar` 的推荐值。

示例3：自动更新 Request

> 注意：
> 自动更新正在运行的 Pod 资源是 VPA 的一项实验功能，建议不要在生产环境中使用该功能。

在本示例中，您将创建一个自动调整 CPU 和内存请求的 VPA，并创建具有两个 Pod 的 Deployment。每个 Pod 都会设置资源的 Request 和 Limits。

在终端中执行以下命令，生成一个名为 `tke-auto-vpa` 的 VPA 对象，指向一个名为 `tke-auto-deployment` 的 Deployment：

```
yaml
cat <<EOF | kubectl apply -f -
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
name: tke-auto-vpa
spec:
targetRef:
apiVersion: "apps/v1"
kind: Deployment
name: tke-auto-deployment
updatePolicy:
```

```
updateMode: "Auto"
EOF
```

注意：
该 VPA 的 `updateMode` 字段的值为 `Auto`，表示 VPA 可以在 Pod 的生命周期内更新 CPU 和内存请求。VPA 可以删除 Pod，调整 CPU 和内存请求，然后启动一个新 Pod。

执行以下命令，生成一个名为 `tke-auto-deployment` 的 Deployment 对象：

```
shell
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
name: tke-auto-deployment
spec:
replicas: 2
selector:
matchLabels:
app: tke-auto-deployment
template:
metadata:
labels:
app: tke-auto-deployment
spec:
containers:
- name: tke-container
image: nginx
resources:
requests:
cpu: 100m
memory: 100Mi
limits:
cpu: 200m
memory: 200Mi
EOF
```

注意：
上述操作创建 Deployment 时设置了资源的 Request 和 Limits，VPA 此时不仅会推荐 Request 值，还会按照 Request 和 Limits 的初始比例自动推荐 Limits 值。例如，YAML 中 CPU 的 Request 和 Limits 的初始比例为

100m:200m = 1:2，那么 VPA 推荐的 Limits 数值则是 VPA 对象中推荐的 Request 数值的两倍。

生成的 Deployment 对象如下图所示：

```
[root@VM-22-114-centos ~]# kubectl get deploy,vpa | grep tke-auto
deployment.apps/tke-auto-deployment   2/2     2            2           10m
verticalpodautoscaler.autoscaling.k8s.io/tke-auto-vpa   Auto   25m   262144k   True   7m26s
```

执行以下命令，获取正在运行中的 Pod 的详细信息：

```
sh
kubectl get pod pod-name -o yaml
```

执行结果如下所示。VPA 修改了原来设置的 Request 和 Limits，更新为 VPA 的推荐值，并维持了初始的 Request 和 Limits 比例。同时生成一个记录更新的 Annotation：

```
yaml
apiVersion: v1
kind: Pod
metadata:
annotations:
...
vpaObservedContainers: tke-container
vpaUpdates: Pod resources updated by tke-auto-vpa: container 0: memory request, c
pu request
...
spec:
containers:
...
resources:
limits: # 新的 Request 和 Limits 会维持初始设置的比例
cpu: 50m
memory: 500Mi
requests:
cpu: 25m
memory: 262144k
...
```

执行以下命令，获取相关 VPA 的详细信息：

```
sh
kubectl get vpa tke-auto-vpa -o yaml
```

执行结果如下所示：

```
yaml
...
recommendation:
containerRecommendations:
- containerName: tke-container
Lower Bound:
Cpu: 25m
Memory: 262144k
Target:
Cpu: 25m
Memory: 262144k
Uncapped Target:
Cpu: 25m
Memory: 262144k
Upper Bound:
Cpu: 101m
Memory: 262144k
```

其中 `target` 表示容器请求 25m CPU 和 262144k 的内存时将以最佳状态运行。

VPA 使用 `lowerBound` 和 `upperBound` 推荐值来决定是否驱逐 Pod 并将其替换为新 Pod。如果 Pod 的请求小于下限或大于上限，则 VPA 将删除 Pod 并将其替换为具有目标推荐值的 Pod。

故障处理

1. 执行 `vpa-up.sh` 脚本时报错

报错信息

```
shell
ERROR: Failed to create CA certificate for self-signing. If the error is "unknown
option -addext", update your openssl version or deploy VPA from the vpa-release-0
.8 branch.
```

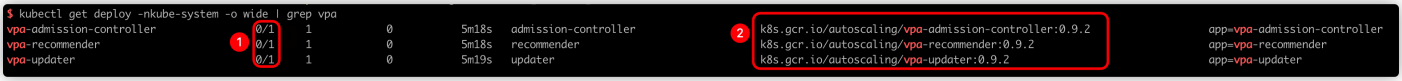
解决方案

1. 如果您没有通过集群中的云服务器执行命令，建议您在云服务器中下载 Autoscaler 项目，并执行完整的 部署 VPA 操作。如需为您的云服务器连接集群，详情可参见 连接集群。
2. 如出现继续报错的情况，请检查是否存在以下问题：
   - 检查集群 CVM 的 `openssl` 版本是否大于 1.1.1。
   - 是否使用 Autoscaler 项目的 `vpa-release-0.8` 分支。

2. VPA 相关负载无法启动

报错信息

如果您的 VPA 相关负载无法启动，并产生如下图所示信息：



信息1：表示负载中的 Pod 没有成功运行。

信息2：表示镜像的地址。

解决方案

VPA 相关负载无法启动的原因是位于 GCR 的镜像无法被下载，为解决问题您可尝试以下步骤：

1. **下载镜像**。
   访问 “k8s.gcr.io/” 镜像仓库，下载 vpa-admission-controller、vpa-recommender、vpa-updater 的镜像。
2. **更换标签及推送**。
   将 vpa-admission-controller、vpa-recommender、vpa-updater 的镜像更换标签后推送到您的镜像仓库中。上传镜像操作详情可参见 容器镜像服务个人版快速入门。
3. **更改 YAML 镜像地址**。
   在 YAML 文件中将 vpa-admission-controller、vpa-recommender、vpa-updater 的镜像地址更新为您设定的新地址。

根据不同业务场景调节 HPA 扩缩容灵敏度

最近更新时间：2023-05-23 15:19:34

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，极大降低了缩容速度，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。

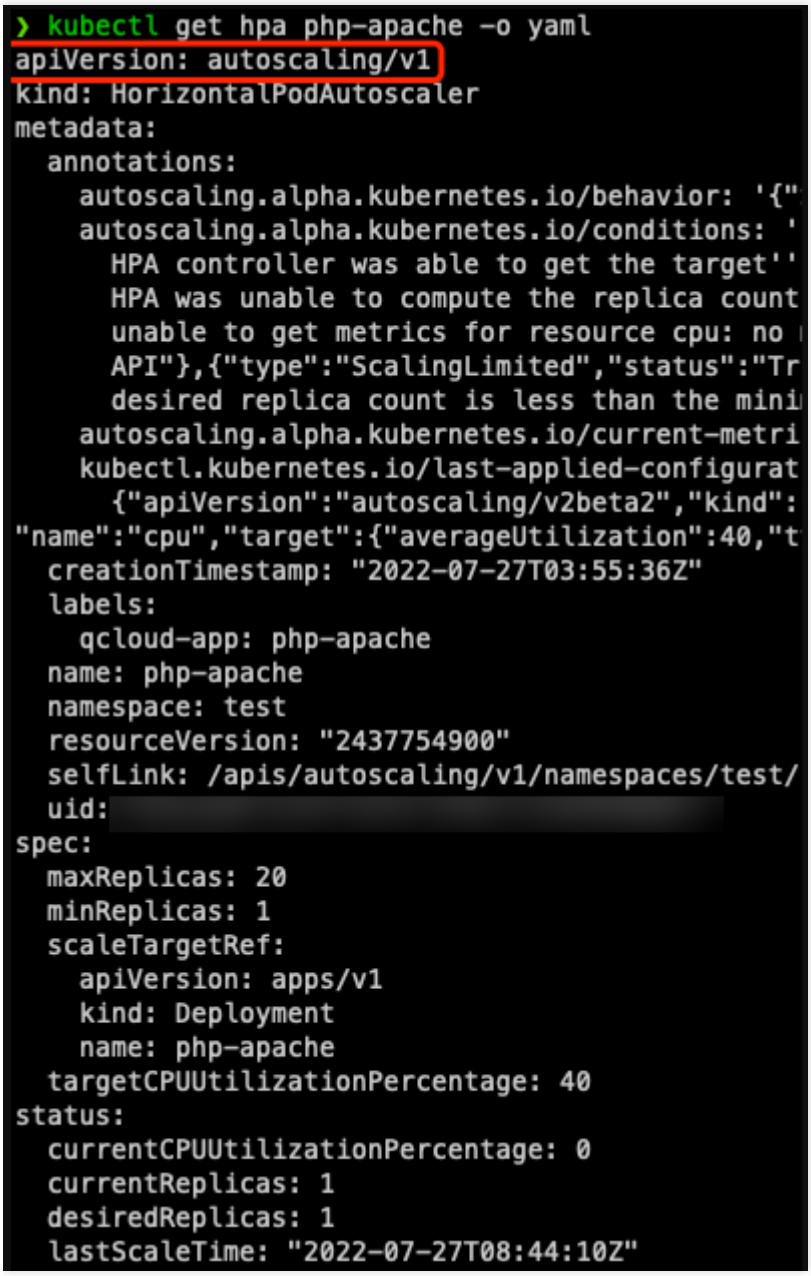
默认的扩容算法会在较短的时间内扩容，针对这种场景我们可以给扩容增加一个时间窗口以避免毛刺导致扩容带来的资源浪费，`behavior` 配置示例如下：

```
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?

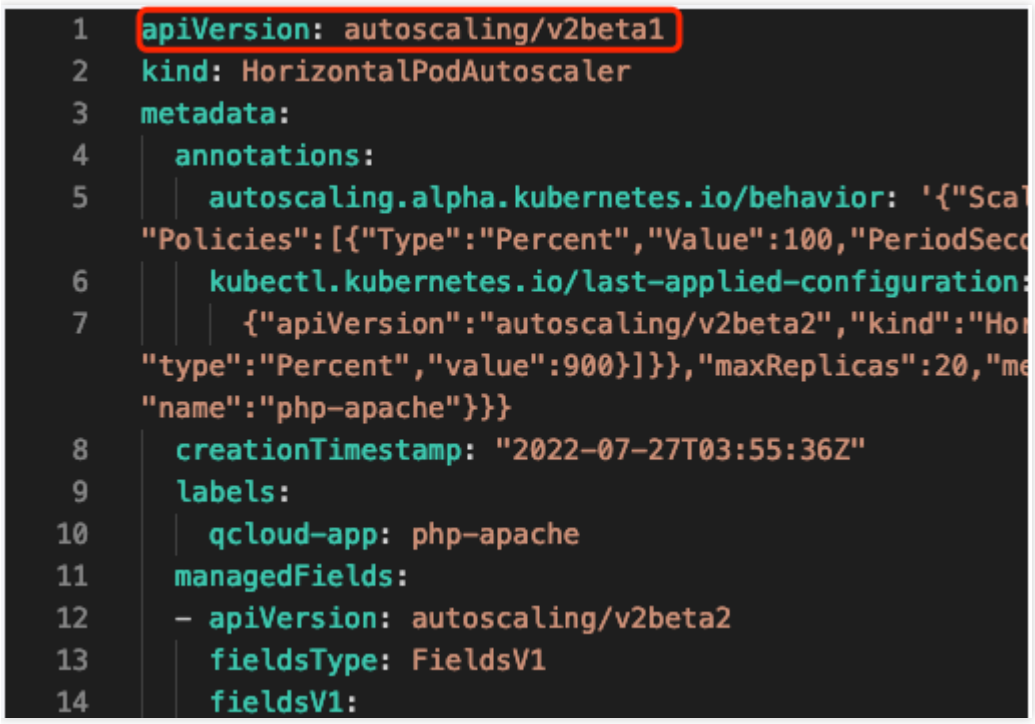


这是因为 HPA 有多个 apiVersion 版本:

```
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

使用 EHPA 实现基于流量预测的弹性

最近更新时间：2024-05-24 15:26:59

EHPA 简介

EHPA（EffectiveHorizontalPodAutoscaler）是 Crane 开源项目 提供的弹性伸缩产品。它基于社区 HPA 做底层的弹性控制，支持更丰富的弹性触发策略（预测、观测、周期），以提高弹性控制的效率，并确保服务质量。
EHPA 的主要特点包括：
提前扩容，保证服务质量：通过算法预测未来的流量洪峰，提前扩容以避免扩容不及时导致的雪崩和服务稳定性故障。
减少无效缩容：通过预测未来的需求，减少不必要的缩容，稳定工作负载的资源使用率，消除突刺误判。
支持 Cron 配置：支持基于 Cron 的弹性配置，以应对大促等异常流量洪峰。
兼容社区：使用社区 HPA 作为弹性控制的执行层，能力完全兼容社区。

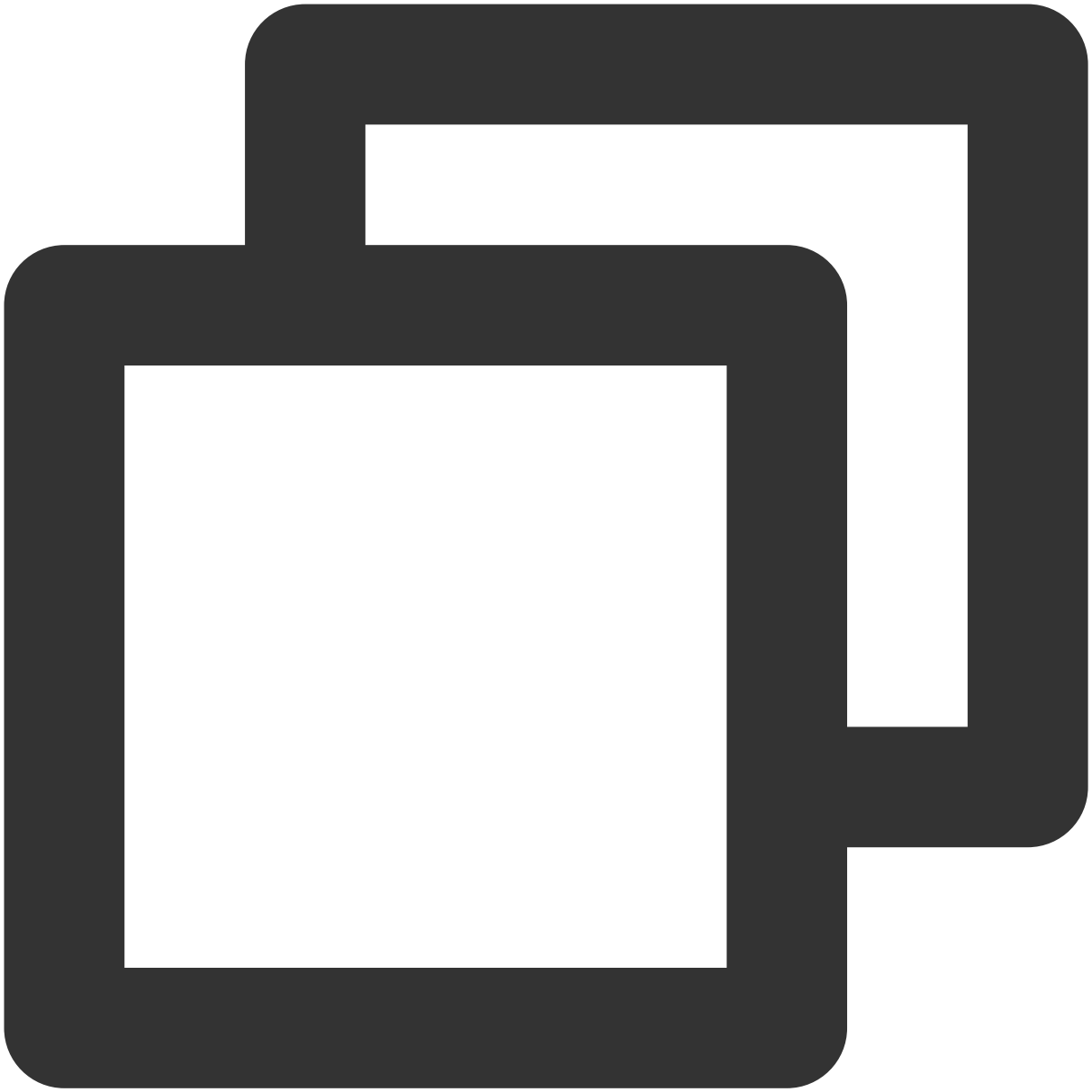
安装

安装 EHPA 可以参考 Crane 开源项目的文档，具体步骤请参见 Intelligent Autoscaling Practices Based on Effective HPA for Custom Metrics。

产品功能

基于预测的弹性

大多数在线应用的负载都具有周期性的特征。用户可以根据按天或者按周的趋势预测未来的负载。EHPA 使用 DSP 算法来预测应用未来的时间序列数据。
以下代码为一个开启了预测能力的 EHPA 模版示例：

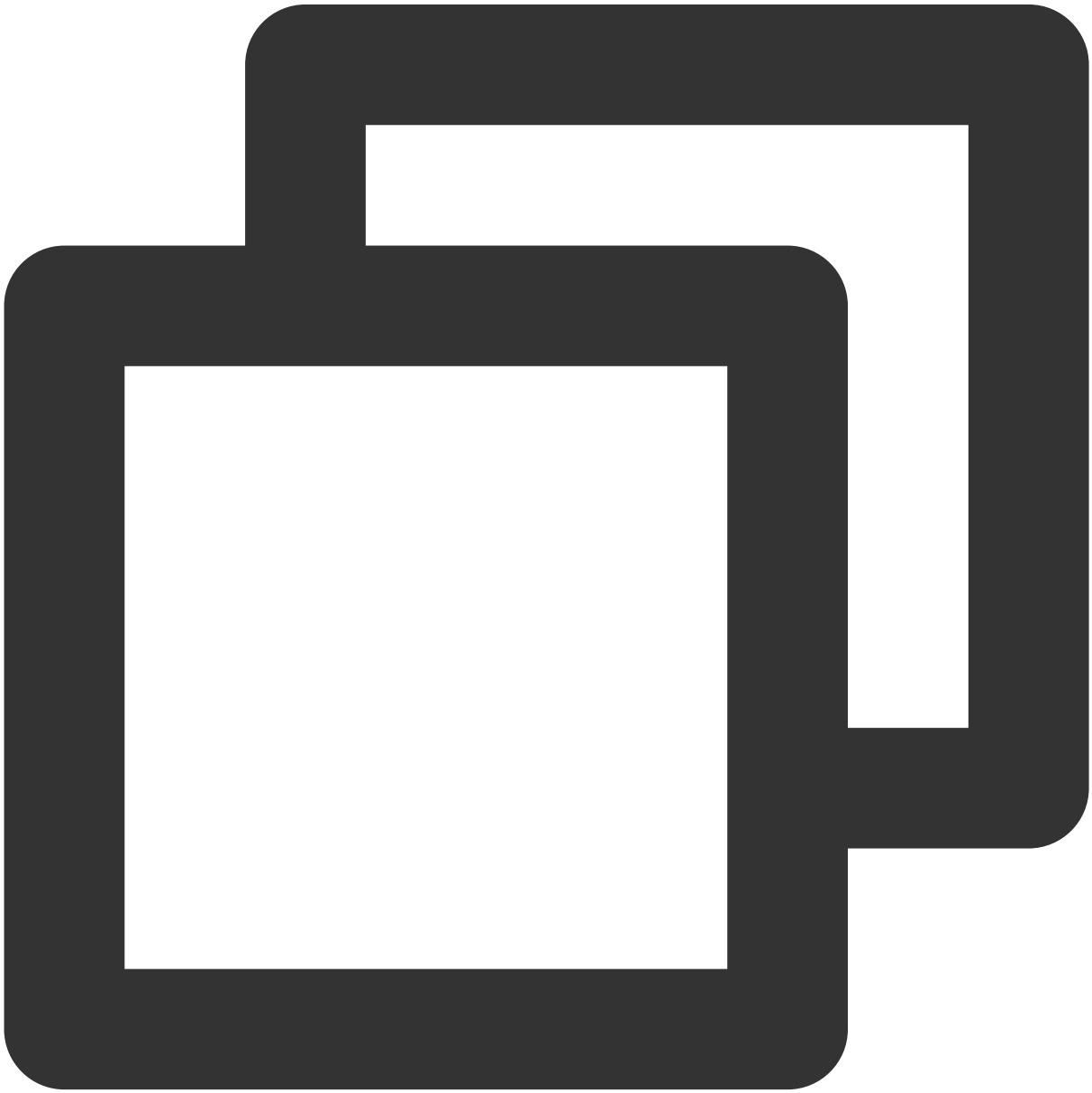


```
apiVersion: autoscaling.crane.io/v1alpha1
kind: EffectiveHorizontalPodAutoscaler
spec:
    prediction:
      predictionWindowSeconds: 3600
      predictionAlgorithm:
        algorithmType: dsp
        dsp:
        sampleInterval: "60s"
        historyLength: "3d"
```

监控数据兜底

在使用预测算法进行预测时，您可能会担心预测数据的准确性，因此在计算副本数时，EHPA 不仅会按预测数据计算，同时也会考虑实际监控数据来兜底，以提升弹性的安全性。具体实现原理是，当您在 EHPA 中定义 spec.metrics 并且开启弹性预测时，EffectiveHPAController 会根据策略自动生成多个 Metric Spec 来创建底层管理的 HPA。

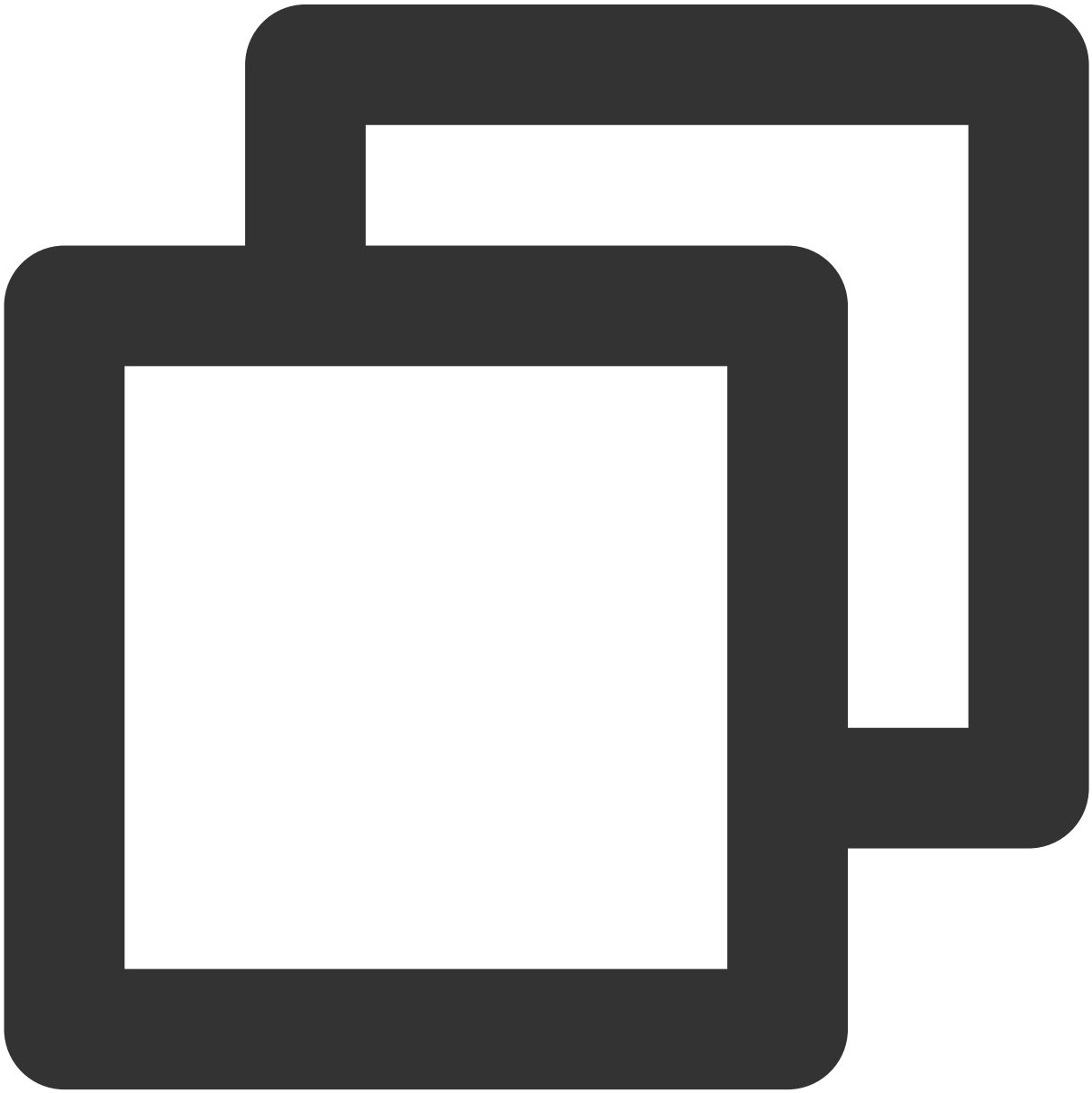
例如，当用户在 EHPA 的 yaml 文件中定义如下 Metric Spec：



```
apiVersion: autoscaling.crane.io/v1alpha1
kind: EffectiveHorizontalPodAutoscaler
spec:
```

```
    metrics:
    - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

将会自动转换为两条 HPA 的阈值配置：



```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
```

```
spec:
  metrics:
    - pods:
        metric:
          name: crane_pod_cpu_usage
          selector:
              matchLabels:
                autoscaling.crane.io/effective-hpa-uid: f9b92249-eab9-4671-afe0-179
        target:
          type: AverageValue
          averageValue: 100m
      type: Pods
    - resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
      type: Resource
```

在上述示例中，用户在 EHPA 创建的 Metric 阈值配置将自动转换为底层 HPA 上的两条 Metric 阈值配置：预测 Metric 阈值和实际监控 Metric 阈值。

预测 Metric 阈值是一个 custom metric。其值由 Crane 的 MetricAdapter 提供。

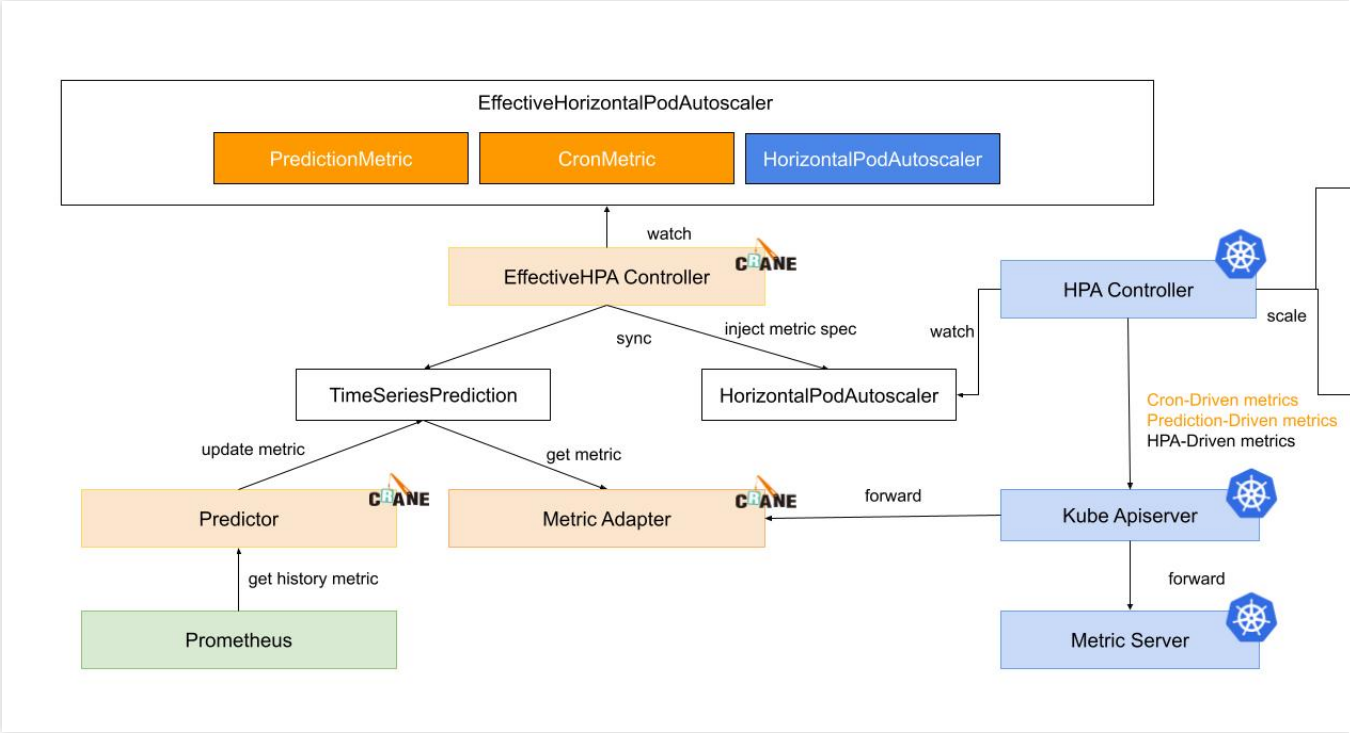
实际监控 Metric 阈值是一个 resource metric，与用户在 EHPA 中定义的相同。这样 HPA 将根据应用程序实际监控的 Metric 计算副本数。

当配置了多个弹性 Metric 阈值时，HPA 将分别计算每个 Metric 对应的副本数，并选择最大的副本数作为最终的推荐弹性结果。

水平弹性的执行流程

1. EffectiveHPAController 创建 HorizontalPodAutoscaler 和 TimeSeriesPrediction 对象。
2. PredictionCore 从 Prometheus 获取历史 metric，通过预测算法计算，将结果记录到 TimeSeriesPrediction。
3. HPAController 通过 metric client 从 KubeApiServer 读取 metric 数据。
4. KubeApiServer 将请求路由到 Crane 的 MetricAdapter。
5. HPAController 计算所有的 Metric 返回的结果，得到最终的弹性副本推荐。
6. HPAController 调用 scale API 对目标应用进行扩/缩容。

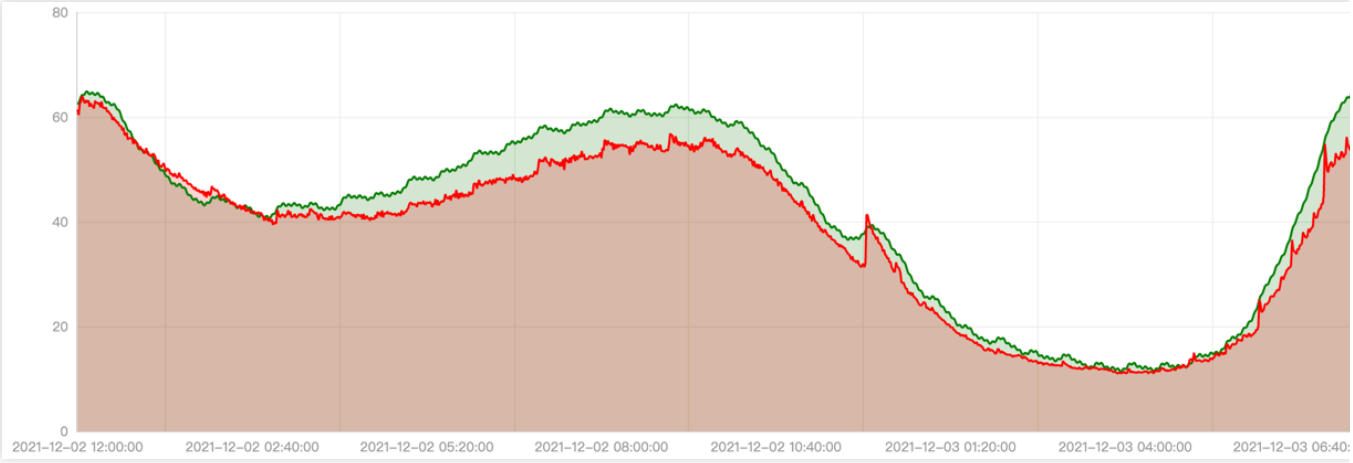
整体流程图如下所示：



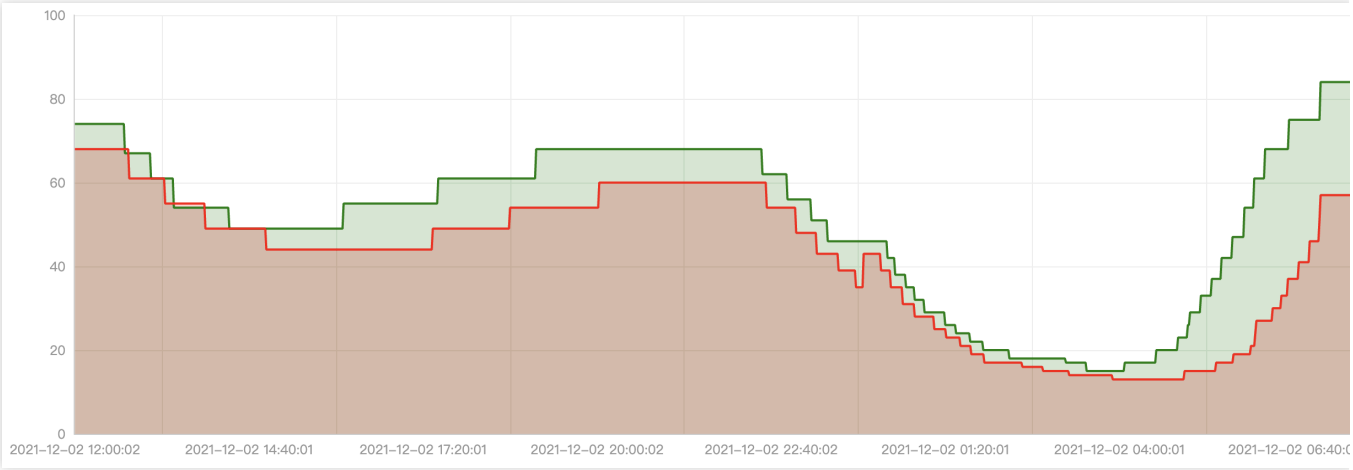
用户案例

通过一个生产环境的客户案例，向您展示 EHPA 的实际效果。本案例在预发环境中重放了生产数据，并对比了使用 EHPA 和社区的 HPA 的弹性效果。

下图中的红线代表应用在一天内的实际 CPU 使用量曲线，您可以看到在早上8点、中午12点和晚上8点时出现了使用高峰。绿线代表 EHPA 预测的 CPU 使用量。



下图是对应的自动弹性的副本数曲线，红线表示社区 HPA 的副本数曲线，绿线表示 EHPA 的副本数曲线。



可以看出 EHPA 具有以下优势：
在流量洪峰到来之前进行扩容。
当流量先下降后立即上升时，不做无效的缩容。
相比 HPA，EHPA 具有更少的弹性次数却更高效。

ScaleStrategy 弹性策略

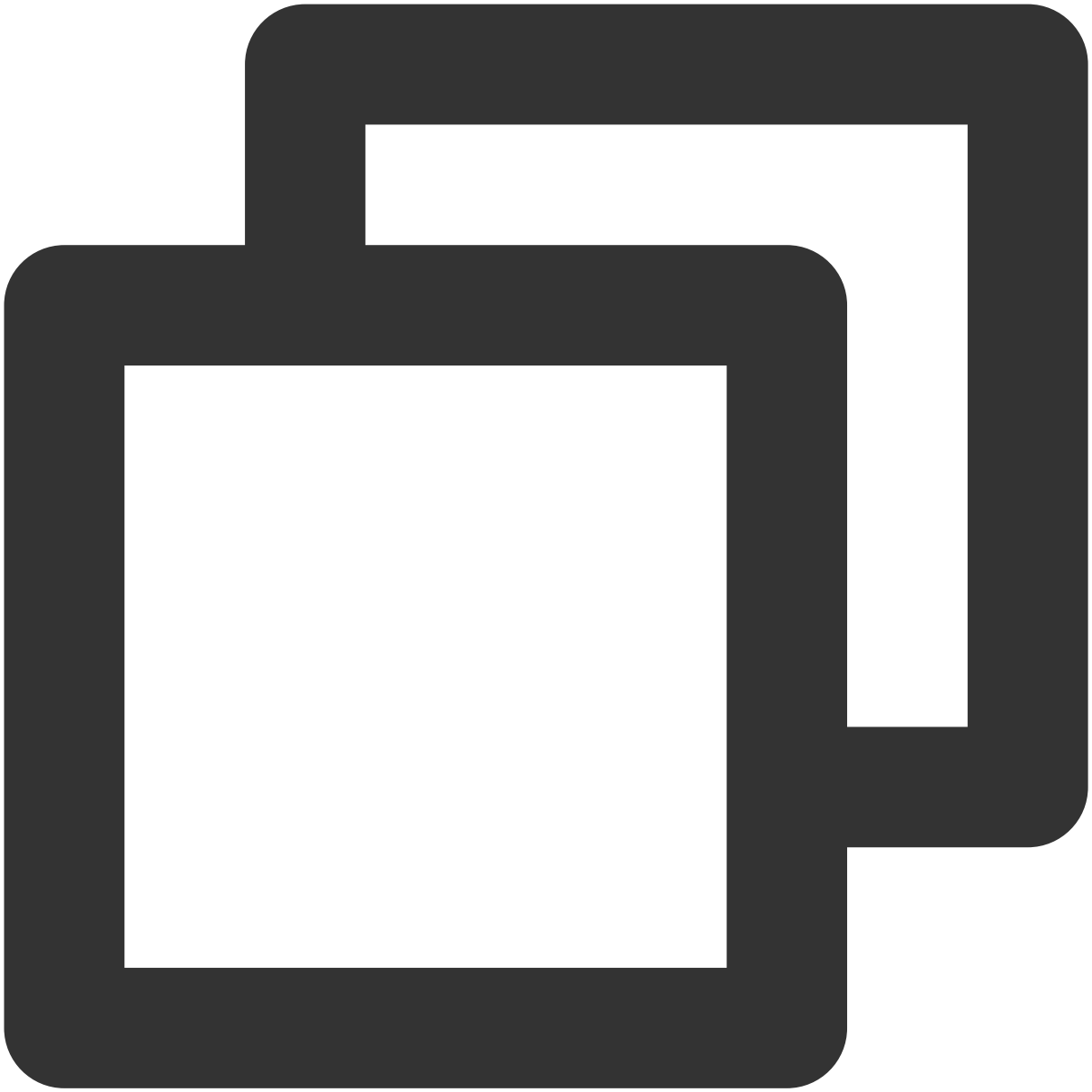
EHPA 提供了两种弹性策略：Auto 和 Preview。您可以随时切换它并立即生效。

Auto

Auto 策略下 EHPA 会自动执行弹性行为。默认 EHPA 的策略是 Auto。在这个模式下 EHPA 会创建一个社区的 HPA 对象并自动接管它的生命周期。不建议您修改或者控制这个底层的 HPA 对象，当 EHPA 被删除时，底层的 HPA 对象也会一并删除。

Preview

Preview 策略提供了一种让 EHPA 不自动执行弹性的能力。因此，您可以通过 EHPA 的 desiredReplicas 字段观测 EHPA 计算出的副本数。您可以随时在两个模式之间切换，当您切换到 Preview 模式时，您可以通过 spec.specificReplicas 调整应用的副本数，如果 spec.specificReplicas 为空，则不会对应用执行弹性，但仍会执行副本数的计算。
以下是一个配置为 Preview 模式的 EHPA 模板示例：



```
apiVersion: autoscaling.crane.io/v1alpha1
kind: EffectiveHorizontalPodAutoscaler
spec:
    scaleStrategy: Preview    # ScaleStrategy indicate the strategy to scaling targe
    pecificReplicas: 5       # SpecificReplicas specify the target replicas.
status:
    expectReplicas: 4         # expectReplicas is the calculated replicas that based
    currentReplicas: 4        # currentReplicas is actual replicas from target
```

在 TKE 使用 KEDA 实现基于 CLB 监控指标的水平伸缩

最近更新时间：2024-05-31 09:43:29

业务场景

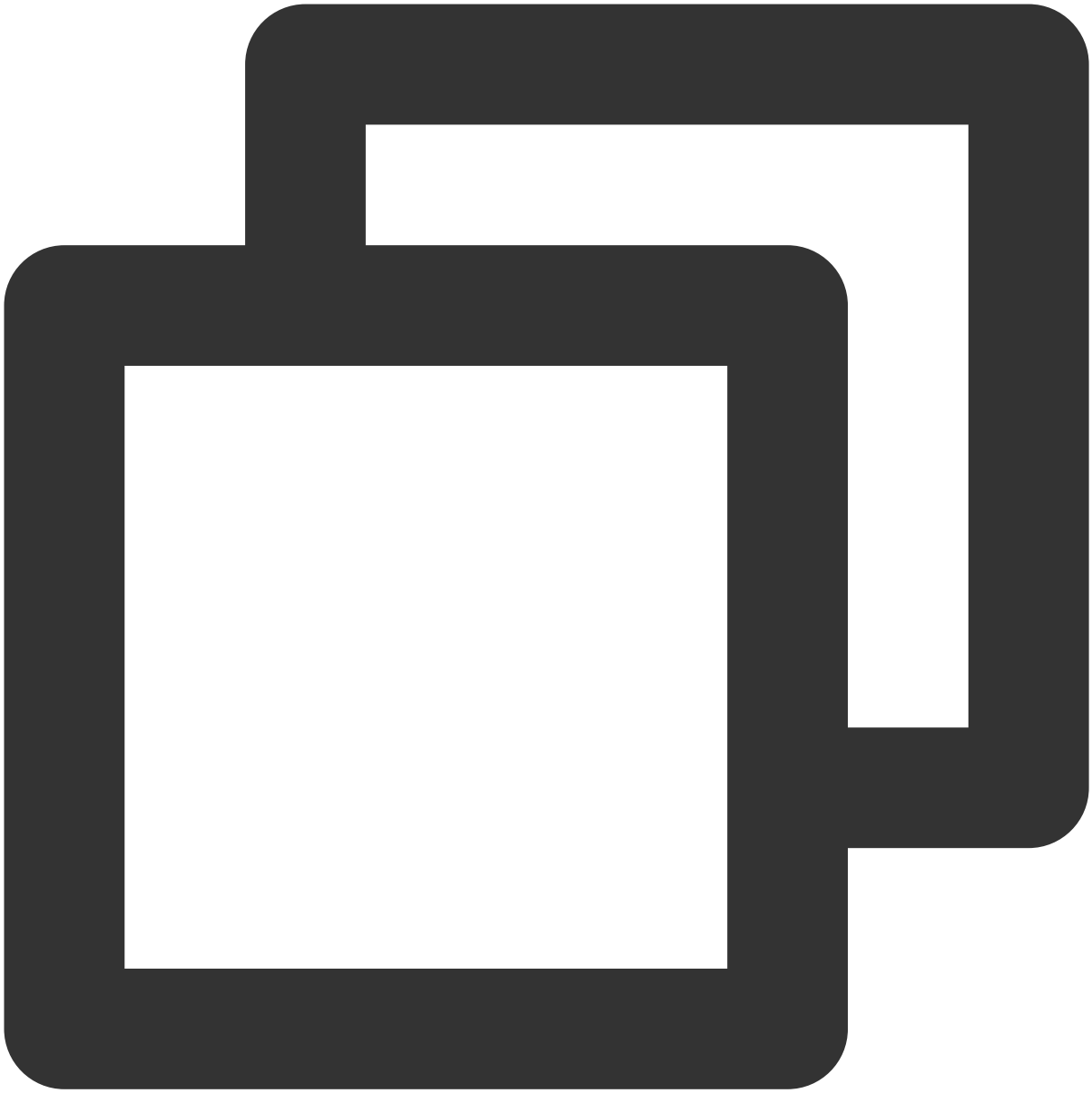
TKE 上的业务流量通常通过 CLB（腾讯云负载均衡器）进行接入。有时候，您希望工作负载能够根据 CLB 的监控指标进行伸缩，例如：
1. 长连接场景（如游戏房间、在线会议）：每个用户对应一条连接，工作负载里的每个 Pod 处理的连接数上限相对固定。这时可以根据 CLB 连接数指标进行伸缩。
2. HTTP 协议的在线业务：工作负载里的单个 Pod 所能支撑的 QPS 相对固定。这时可以根据 CLB 的 QPS（每秒请求数） 指标进行伸缩。

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

安装 keda-tencentcloud-clb-scaler

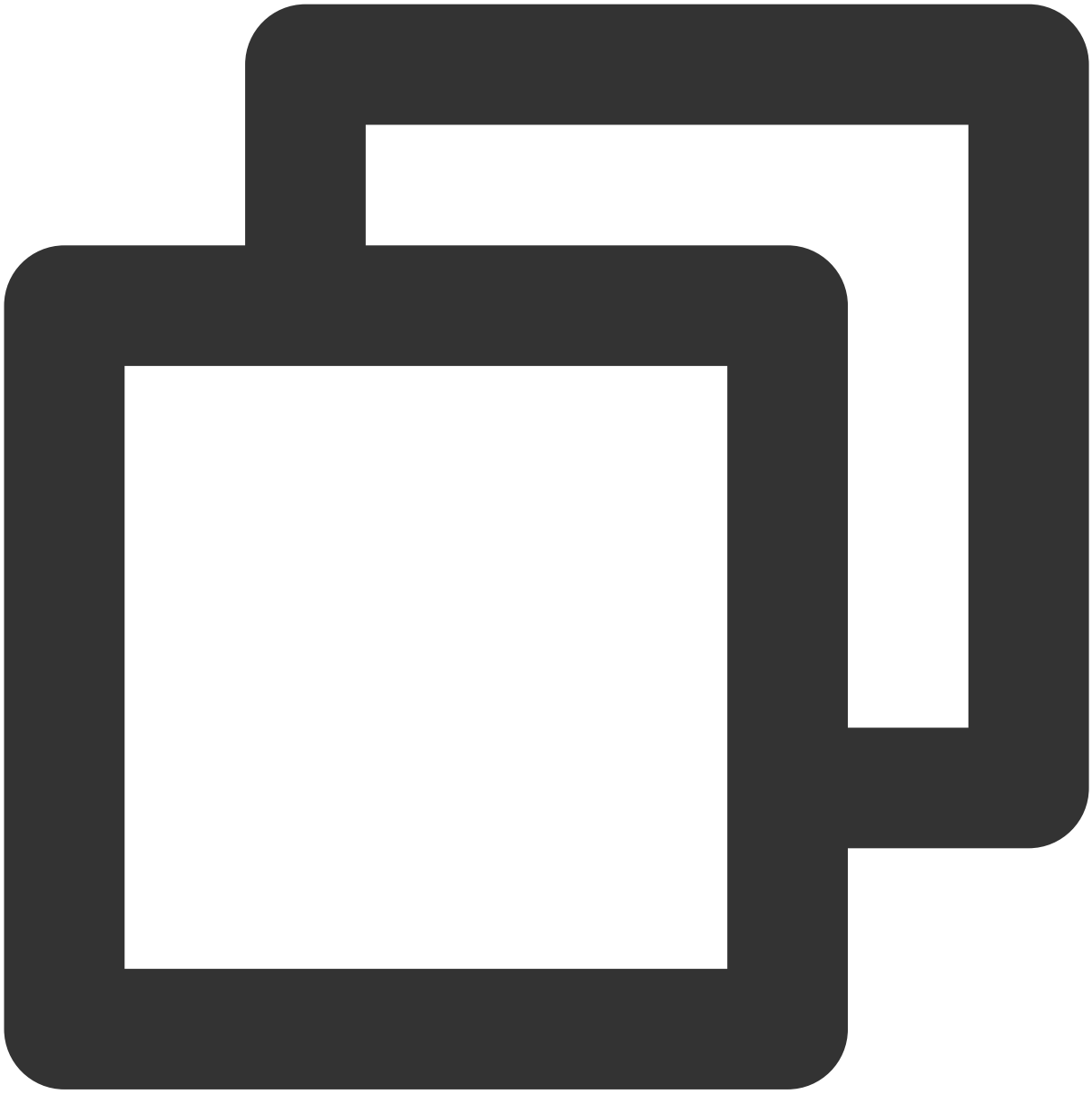


```
helm repo add clb-scaler https://imroc.github.io/keda-tencentcloud-clb-scaler
helm upgrade --install clb-scaler clb-scaler/clb-scaler -n keda \\
  --set region="ap-chengdu" \\
  --set credentials.secretId="xxx" \\
  --set credentials.secretKey="xxx"
```

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

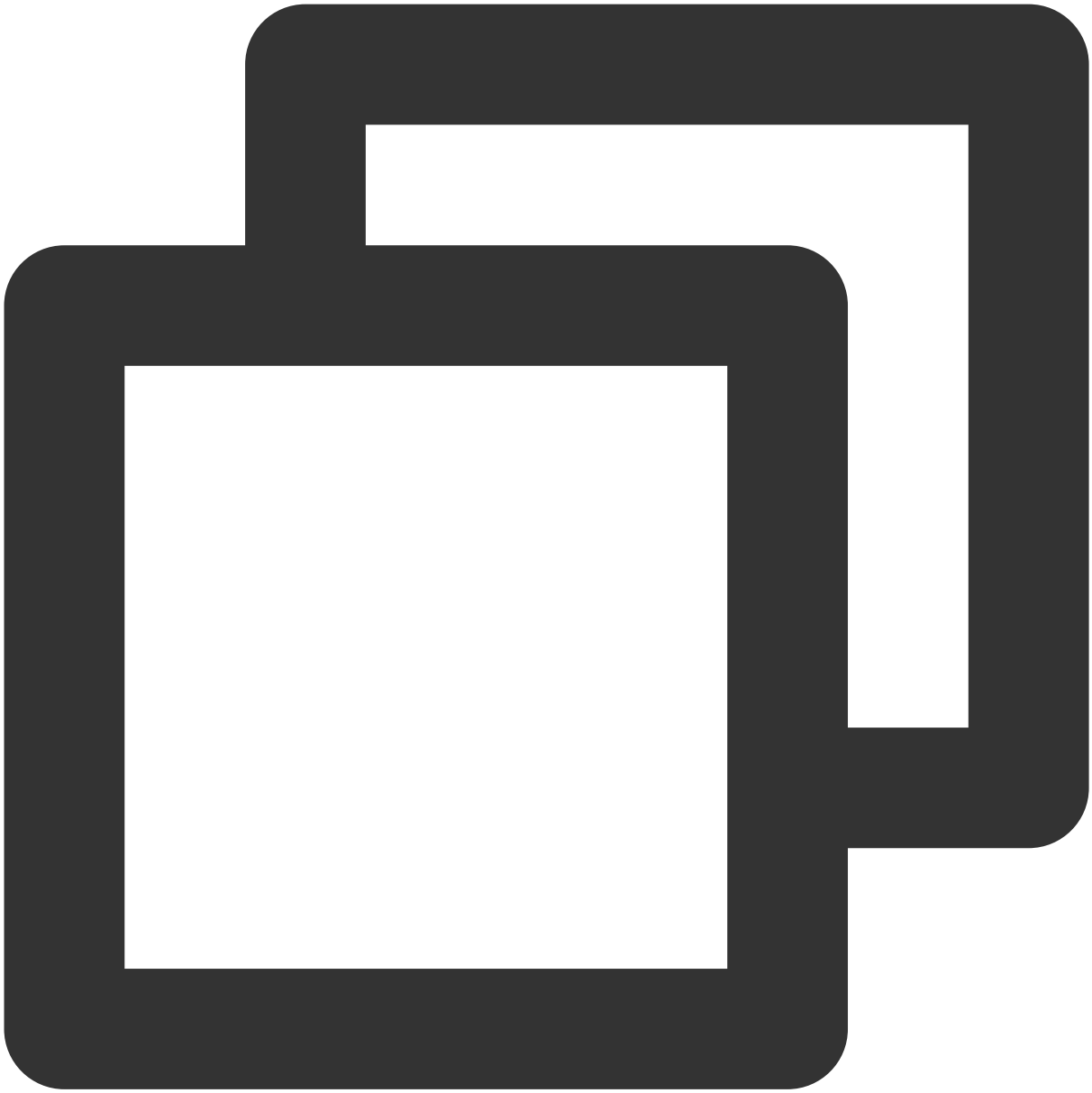
您可以使用以下用于测试的工作负载 YAML 样例：



```
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 svc httpbin -o jsonpath='{.metadata.annotations.service\\.kubernetes\\.io
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。

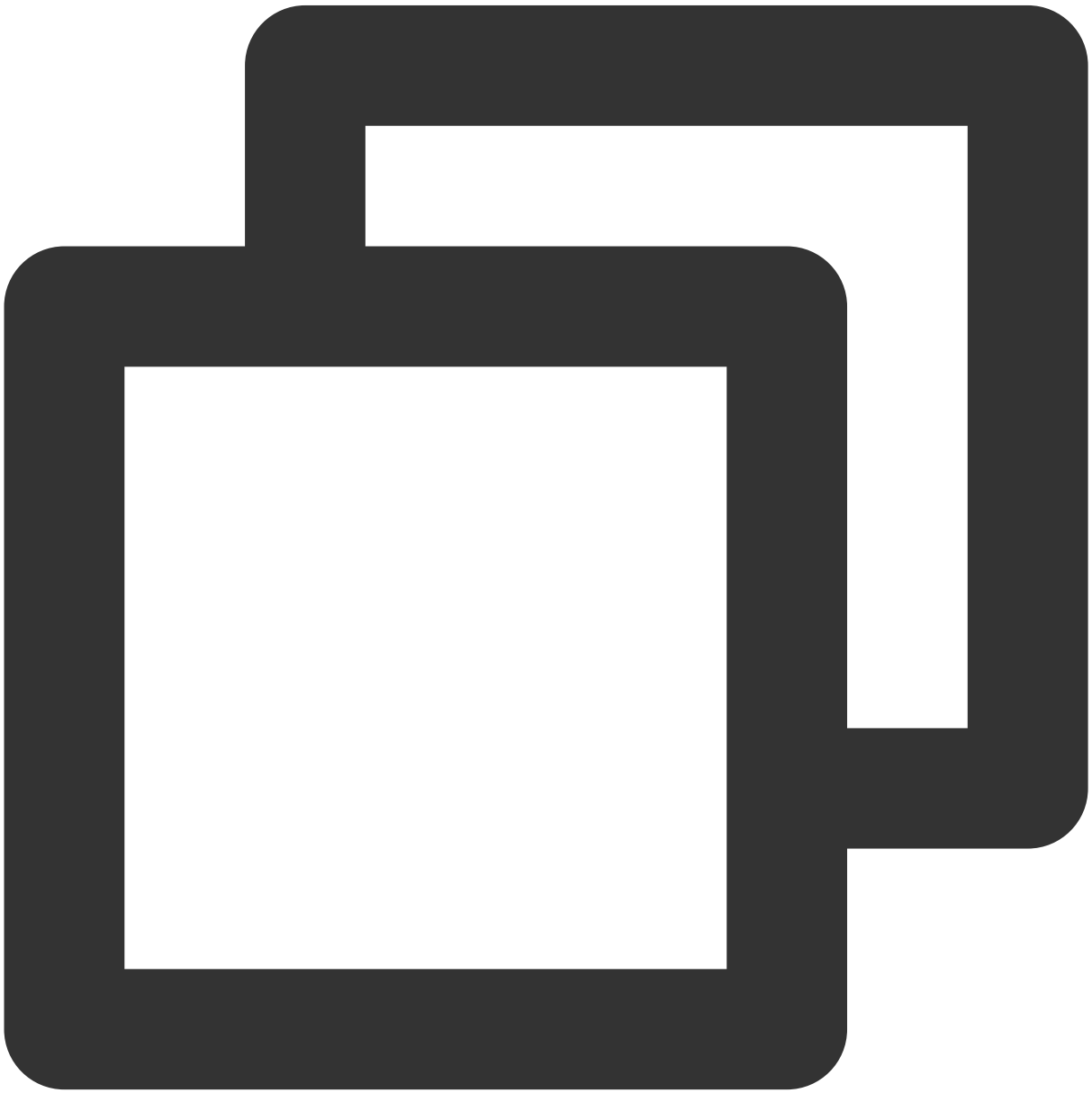
loadBalancerId 是 CLB 的实例 ID。

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。

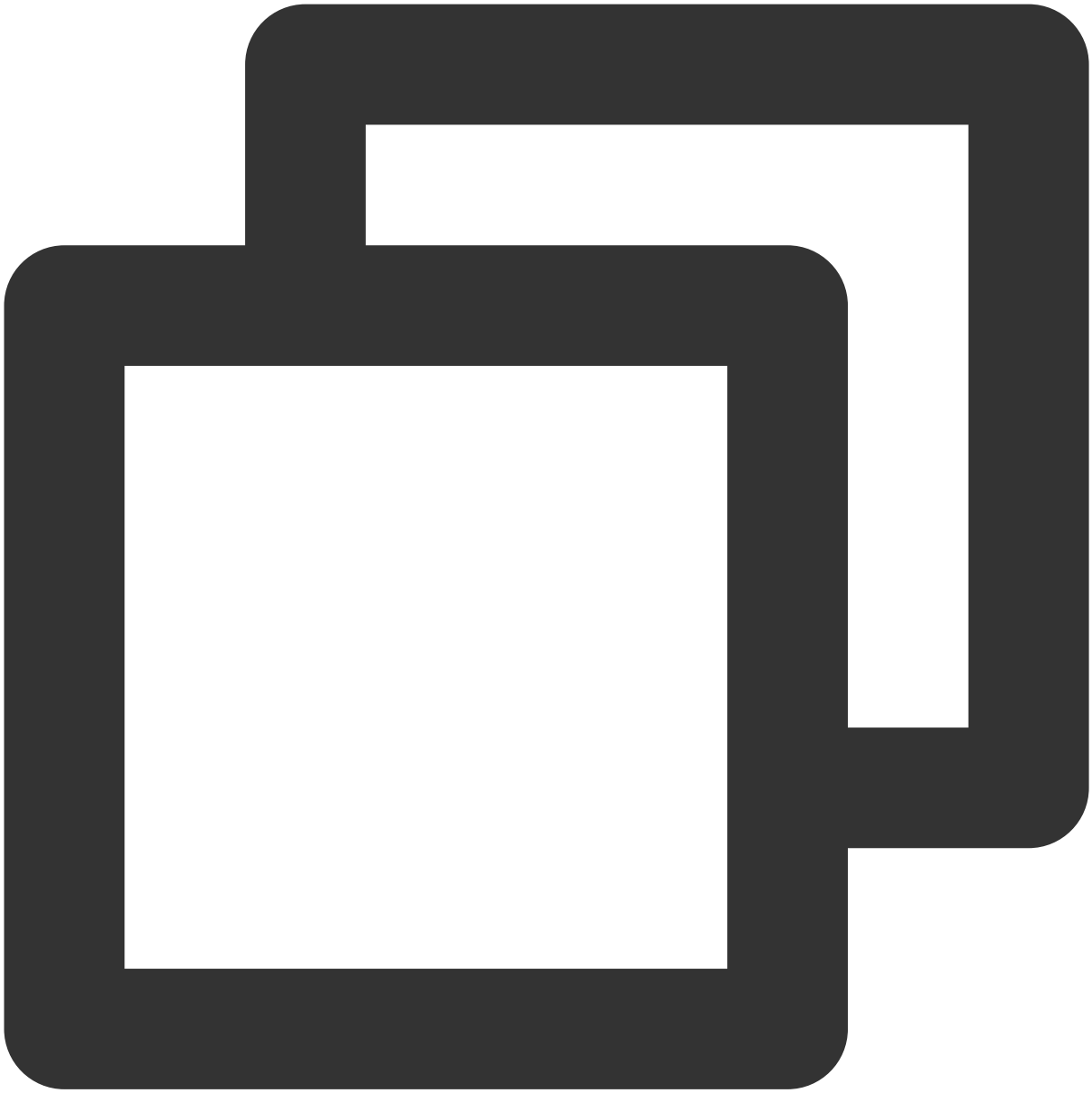
配置示例一：基于 CLB 连接数指标的弹性伸缩



```
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
```

存储
使用 CBS CSI 插件对 PVC 进行备份与恢复

最近更新时间：2023-05-23 16:58:46

操作场景

如需为 PVC 数据盘创建快照来备份数据，或者将备份的快照数据恢复到新的 PVC 中，可以通过 CBS-CSI 插件来实现，本文将介绍如何利用 CBS-CSI 插件实现 PVC 的数据备份与恢复。

前提条件

- 已创建 TKE 集群 或已在腾讯云自建 Kubernetes 集群，集群版本 >= 1.18。
- 已安装 CBS-CSI 插件。
- 在 访问管理 控制台完成对 `TKE_QCSRole` 角色授予 CBS 快照操作的相关权限，详情请参考 快照授权。

操作步骤

备份 PVC

创建 VolumeSnapshotClass

1. 使用以下 YAML，创建 VolumeSnapshotClass 对象。示例如下：

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
name: cbs-snapclass
driver: com.tencent.cloud.csi.cbs
deletionPolicy: Delete
```

2. 执行以下命令，检查 VolumeSnapshotClass 是否创建成功。示例如下：

```
$ kubectl get volumesnapshotclass
NAME DRIVER DELETIONPOLICY AGE
cbs-snapclass com.tencent.cloud.csi.cbs Delete 17m
```

创建 PVC 快照 VolumeSnapshot

1. 本文以 `new-snapshot-demo` 快照名为例创建 VolumeSnapshot。使用以下 YAML，创建 VolumeSnapshot 对象。示例如下：

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshot
metadata:
name: new-snapshot-demo
spec:
volumeSnapshotClassName: cbs-snapclass # 引用前面创建的 VolumeSnapshotClass
source:
persistentVolumeClaimName: ssd-pvc # 替换成要备份的 pvc 名称
```

2. 执行以下命令，查看 Volumesnapshot 和 Volumesnapshotcontent 对象是否创建成功，若 `READYTOUSE` 为 true，则创建成功。示例如下：

```
$ kubectl get volumesnapshot
NAME READYTOUSE SOURCEPVC SOURCESNAPSHOTCONTENT RESTORESIZE SNAPSHOTCLASS SNAPS
HOTCONTENT CREATIONTIME AGE
new-snapshot-demo true ssd-pvc 20Gi cbs-snapclass snapcontent-170b2161-f158-4c9
e-a090-a38fdfd84a3e 2m36s 2m50s
$ kubectl get volumesnapshotcontent
NAME READYTOUSE RESTORESIZE DELETIONPOLICY DRIVER VOLUMESNAPSHOTCLASS VOLUMESNA
PSHOT AGE
snapcontent-170b2161-f158-4c9e-a090-a38fdfd84a3e true 21474836480 Delete com.te
ncent.cloud.csi.cbs cbs-snapclass new-snapshot-demo 3m3s
```

3. 执行以下命令，可以获取 Volumesnapshotcontent 对象的快照 ID，字段是 `status.snapshotHandle`（如下为 snap-rsk8v75j），可以根据快照 ID 在 容器服务控制台 确认快照是否存在。示例如下：

```
$ kubectl get volumesnapshotcontent -o yaml snapcontent-170b2161-f158-4c9e-a090
-a38fdfd84a3e
...
status:
creationTime: 1607331318000000000
readyToUse: true
restoreSize: 21474836480
snapshotHandle: snap-rsk8v75j
```

从快照恢复数据到新 PVC

1. 本文以上述 步骤 创建的 VolumeSnapshot 对象名称 `new-snapshot-demo` 为例，使用以下 YAML 从快照恢复数据到新的 PVC 中。示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: restore-test
spec:
storageClassName: ssd-csi # storage class 根据自身需求自定义
dataSource:
name: new-snapshot-demo # 引用前面创建的 VolumeSnapshot
kind: VolumeSnapshot
apiGroup: snapshot.storage.k8s.io
accessModes:
- ReadWriteOnce # CBS 是块存储，只支持单机读写
resources:
requests:
storage: 50Gi # 建议大小与被恢复的 PVC 写成一致
```

2. 执行以下命令，可以查看 PVC 已经创建并绑定 PV，从 PV 中也可以查看到对应的 diskid（如下为 disk-ju0hw7no）。示例如下：

```
$ kubectl get pvc restore-test
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
restore-test Bound pvc-940edf09-d622-4126-992b-0a209f048c7d 60Gi RWO ssd-topolo
gy 6m8s
$ kubectl get pv pvc-940edf09-d622-4126-992b-0a209f048c7d -o yaml
...
spec:
...
volumeHandle: disk-ju0hw7no
...
```

说明：
如果 StorageClass 使用了拓扑感知（先调度 Pod 再创建 PV），即指定 `volumeBindingMode: WaitForFirstConsumer`，则需要先部署 Pod（需挂载 PVC）才会触发创建 PV（从快照创建新的 CBS 并与 PV 绑定）。

静态挂载 CFS-Turbo 类文件系统
TKE 挂载 CFS-Turbo

最近更新时间：2023-08-03 16:21:58

操作场景

为 TKE 集群挂载 CFS Turbo 类型存储，可以通过安装 `kubernetes-csi-tencentloud` 组件来实现。该组件基于私有协议将腾讯云 CFS Turbo 文件系统挂载到工作负载，目前仅支持静态配置。CFS 存储类型请参考 文件存储类型及性能规格。

前提条件

已创建 TKE 集群或已在腾讯云自建 Kubernetes 集群，集群版本 >=1.14。

操作步骤

创建文件系统

创建 CFS Turbo 文件系统，具体操作请参见 创建文件系统。

> 注意：
> 文件系统创建后，需将集群网络（vpc-xx）关联到文件系统的 云联网（可在文件系统挂载点信息中查看）。

部署 RBAC 策略

如果您需要挂载 CFS Turbo 存储卷，需执行 `kubectl apply -f csi-node-rbac.yaml` 命令在集群中先部署 RBAC 策略，csi-node-rbac.yaml 代码参考如下：

```
apiVersion: v1
kind: ServiceAccount
metadata:
name: cfsturbo-csi-node-sa
namespace: kube-system
---
kind: ClusterRole
```

```
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: cfsturbo-csi-node-role
rules:
- apiGroups: [""]
resources: ["persistentvolumes", "endpoints", "configmaps"]
verbs: ["get", "list", "watch", "create", "delete", "update"]
- apiGroups: [""]
resources: ["persistentvolumeclaims", "nodes"]
verbs: ["get", "list", "watch", "update"]
- apiGroups: [""]
resources: ["events"]
verbs: ["get", "list", "watch", "create", "update", "patch"]
- apiGroups: [""]
resources: ["secrets", "namespaces"]
verbs: ["get", "list"]
- apiGroups: [""]
resources: ["nodes", "pods"]
verbs: ["get", "list", "watch", "update"]
- apiGroups: ["storage.k8s.io"]
resources: ["volumeattachments", "volumeattachments"]
verbs: ["get", "list", "watch", "update", "patch"]
- apiGroups: ["storage.k8s.io"]
resources: ["storageclasses"]
verbs: ["get", "list", "watch"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: cfsturbo-csi-node-rolebinding
subjects:
- kind: ServiceAccount
name: cfsturbo-csi-node-sa
namespace: kube-system
roleRef:
kind: ClusterRole
name: cfsturbo-csi-node-role
apiGroup: rbac.authorization.k8s.io
```

部署 Node Plugin

1. 执行 `kubectl apply -f csidriver.yaml` 命令，csidriver.yaml 代码参考如下：

```
apiVersion: storage.k8s.io/v1beta1
kind: CSIDriver
metadata:
name: com.tencent.cloud.csi.cfsturbo
```

```
spec:
attachRequired: false
podInfoOnMount: false
```

2. 执行 `kubectl apply -f csi-node.yaml` 命令，csi-node.yaml 代码参考如下：

```
# This YAML file contains driver-registrar & csi driver nodeplugin API objects
# that are necessary to run CSI nodeplugin for cfsturbo
kind: DaemonSet
apiVersion: apps/v1
metadata:
name: cfsturbo-csi-node
namespace: kube-system
spec:
selector:
matchLabels:
app: cfsturbo-csi-node
template:
metadata:
labels:
app: cfsturbo-csi-node
spec:
serviceAccount: cfsturbo-csi-node-sa
hostNetwork: true
containers:
- name: driver-registrar
image: ccr.ccs.tencentyun.com/tkeimages/csi-node-driver-registrar:v1.2.0
lifecycle:
preStop:
exec:
command: ["/bin/sh", "-c", "rm -rf /registration/com.tencent.cloud.csi.cfsturbo
/registration/com.tencent.cloud.csi.cfsturbo-reg.sock"]
args:
- "--v=5"
- "--csi-address=/plugin/csi.sock"
- "--kubelet-registration-path=/var/lib/kubelet/plugins/com.tencent.cloud.csi.c
fsturbo/csi.sock"
env:
- name: KUBE_NODE_NAME
valueFrom:
fieldRef:
fieldPath: spec.nodeName
volumeMounts:
- name: plugin-dir
mountPath: /plugin
- name: registration-dir
```

```
mountPath: /registration
- name: cfsturbo
securityContext:
privileged: true
capabilities:
add: ["SYS_ADMIN"]
allowPrivilegeEscalation: true
image: ccr.ccs.tencentyun.com/tkeimages/csi-tencentcloud-cfsturbo:v1.2.2
args :
- "--nodeID=$(NODE_ID)"
- "--endpoint=$(CSI_ENDPOINT)"
env:
- name: NODE_ID
valueFrom:
fieldRef:
fieldPath: spec.nodeName
- name: CSI_ENDPOINT
value: unix://plugin/csi.sock
imagePullPolicy: "IfNotPresent"
volumeMounts:
- name: plugin-dir
mountPath: /plugin
- name: pods-mount-dir
mountPath: /var/lib/kubelet/pods
mountPropagation: "Bidirectional"
- name: global-mount-dir
mountPath: /etc/cfsturbo/global
mountPropagation: "Bidirectional"
volumes:
- name: plugin-dir
hostPath:
path: /var/lib/kubelet/plugins/com.tencent.cloud.csi.cfsturbo
type: DirectoryOrCreate
- name: pods-mount-dir
hostPath:
path: /var/lib/kubelet/pods
type: Directory
- name: registration-dir
hostPath:
path: /var/lib/kubelet/plugins_registry
type: Directory
- name: global-mount-dir
hostPath:
path: /etc/cfsturbo/global
type: DirectoryOrCreate
```

使用 CFS Turbo 存储卷

1. 创建 CFS Turbo 文件系统，具体操作请参见 创建文件系统。
2. 使用以下模板创建 CFS Turbo 类型的 PV。

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: pv-cfsturbo
spec:
accessModes:
- ReadWriteMany
capacity:
storage: 10Gi
csi:
driver: com.tencent.cloud.csi.cfsturbo
# volumeHandle in PV must be unique, use pv name is better
volumeHandle: pv-cfsturbo
volumeAttributes:
# cfs turbo server ip
host: 10.0.0.116
# cfs turbo fsid (not cfs id)
fsid: xxxxxxxx
# cfs turbo rootdir
rootdir: /cfs
# cfs turbo subPath
path: /
proto: lustre
storageClassName: ""
```

参数说明：

- **metadata.name**: 创建 PV 名称。
- **spec.csi.volumeHandle**: 与 PV 名称保持一致。
- **spec.csi.volumeAttributes.host**: 文件系统 ip 地址，可在文件系统挂载点信息中查看。
- **spec.csi.volumeAttributes.fsid**: 文件系统 fsid（非文件系统 id），可在文件系统挂载点信息中查看（挂载命令中 “tcp0:/” 之后 “/cfs” 之前的那一段字符串，如下图）。
- **spec.csi.volumeAttributes.rootdir**: 文件系统根目录，不填写默认为 “/cfs”（挂载到 “/cfs” 目录可相对提高整体挂载性能）。如需指定根目录挂载，须确保该根目录在文件系统中存在。
- **spec.csi.volumeAttributes.path**: 文件系统子目录，不填写默认为 “/”。如需指定子目录挂载，须确保该子目录在文件系统 rootdir 中存在。容器最终访问到的是文件系统中 rootdir+path 目录（默认为 “/cfs/” 目录）。

- **spec.csi.volumeAttributes.proto**：文件系统默认挂载协议。

> 注意：
> 使用 `lustre` 协议挂载 CFS Turbo 卷需预先在集群节点内根据操作系统内核版本安装对应客户端，详情请参考 在 Linux 客户端上使用 CFS Turbo 文件系统；

3. 使用以下模板创建 PVC 绑定 PV。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: pvc-cfsturbo
spec:
storageClassName: ""
volumeName: pv-cfsturbo
accessModes:
- ReadWriteMany
resources:
requests:
storage: 10Gi
```

参数说明：

- **metadata.name**: 创建 PVC 名称。
- **spec.volumeName**: 与上一步中创建 PV 名称保持一致。

4. 使用以下模板创建 Pod 挂载 PVC。

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
containers:
- image: ccr.ccs.tencentyun.com/qcloud/nginx:1.9
```

```
imagePullPolicy: Always
name: nginx
ports:
- containerPort: 80
protocol: TCP
volumeMounts:
- mountPath: /var/www
name: data
volumes:
- name: data
persistentVolumeClaim:
claimName: pvc-cfsturbo
```

TKE Serverless 静态挂载 CFS-Turbo

最近更新时间：2023-05-06 17:15:18

使用场景

为 TKE Serverless 集群挂载文件存储（Cloud File Storage，CFS）Turbo 类型存储，该组件基于私有协议将腾讯云 CFS Turbo 文件系统挂载到工作负载，目前仅支持静态配置。CFS 存储类型详情见 文件存储类型及性能规格。

前提条件

已创建 TKE Serverless 集群且集群版本 >=1.14。

使用步骤

创建文件系统

创建 CFS Turbo 文件系统，具体操作请参见 创建文件系统。

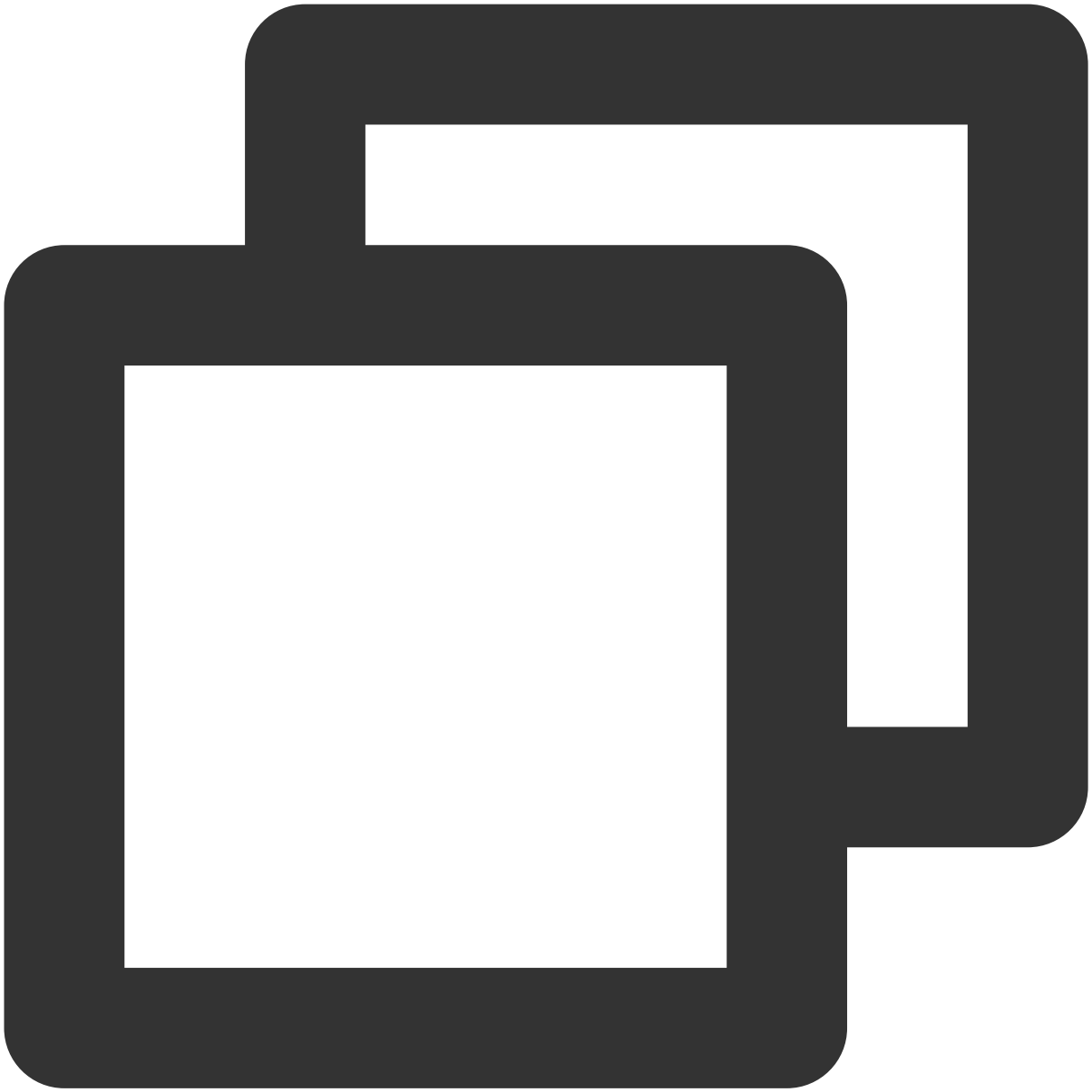
注意

文件系统创建后，需将集群网络（vpc-xx）关联到文件系统的 云联网（可在文件系统挂载点信息中查看）。

部署 Node Plugin

步骤1：新建 csidriver.yaml 文件

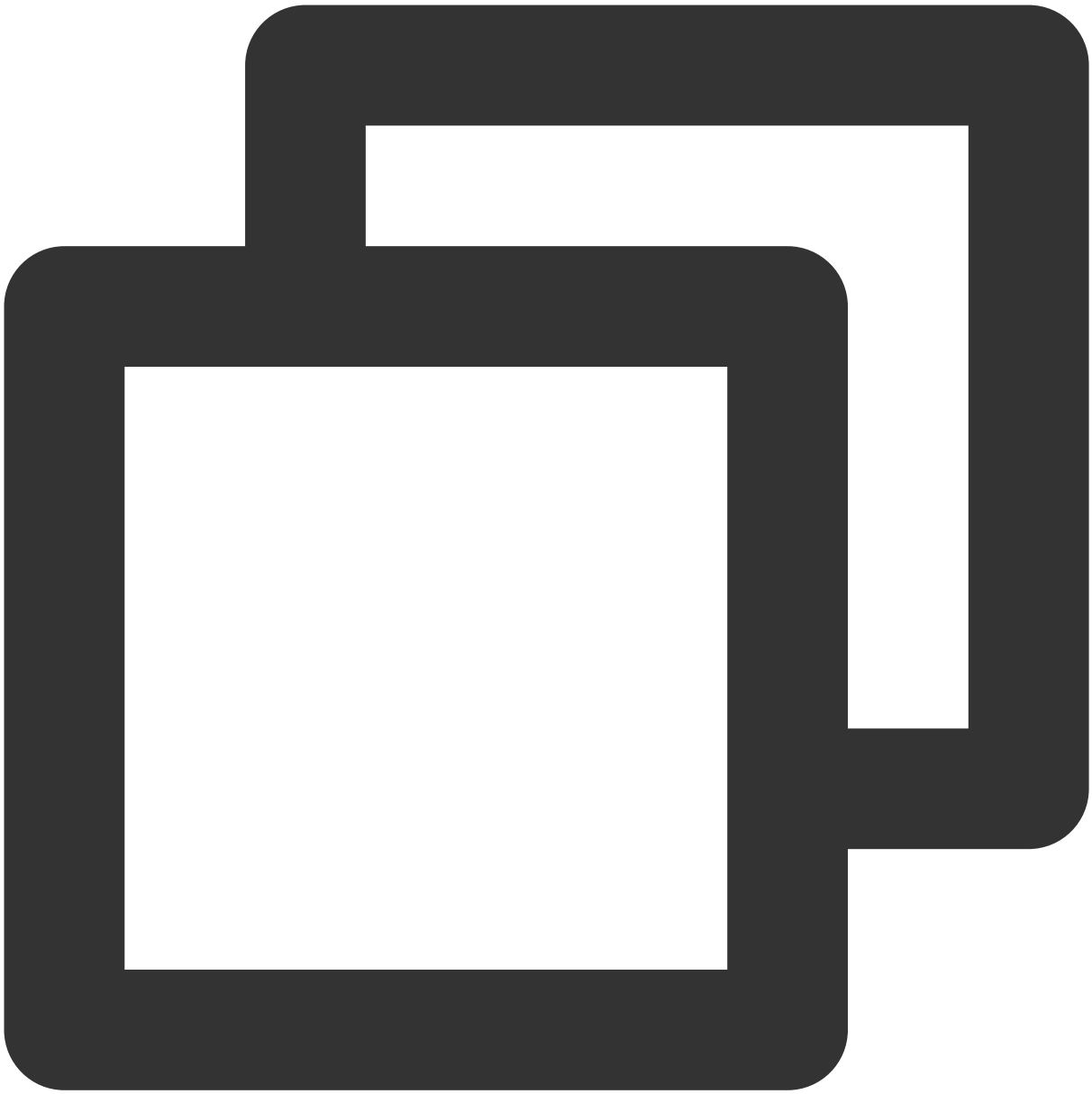
csidriver.yaml 文件示例如下：



```
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: com.tencent.cloud.csi.cfsturbo
spec:
  attachRequired: false
  podInfoOnMount: false
```

步骤2：创建 csidriver

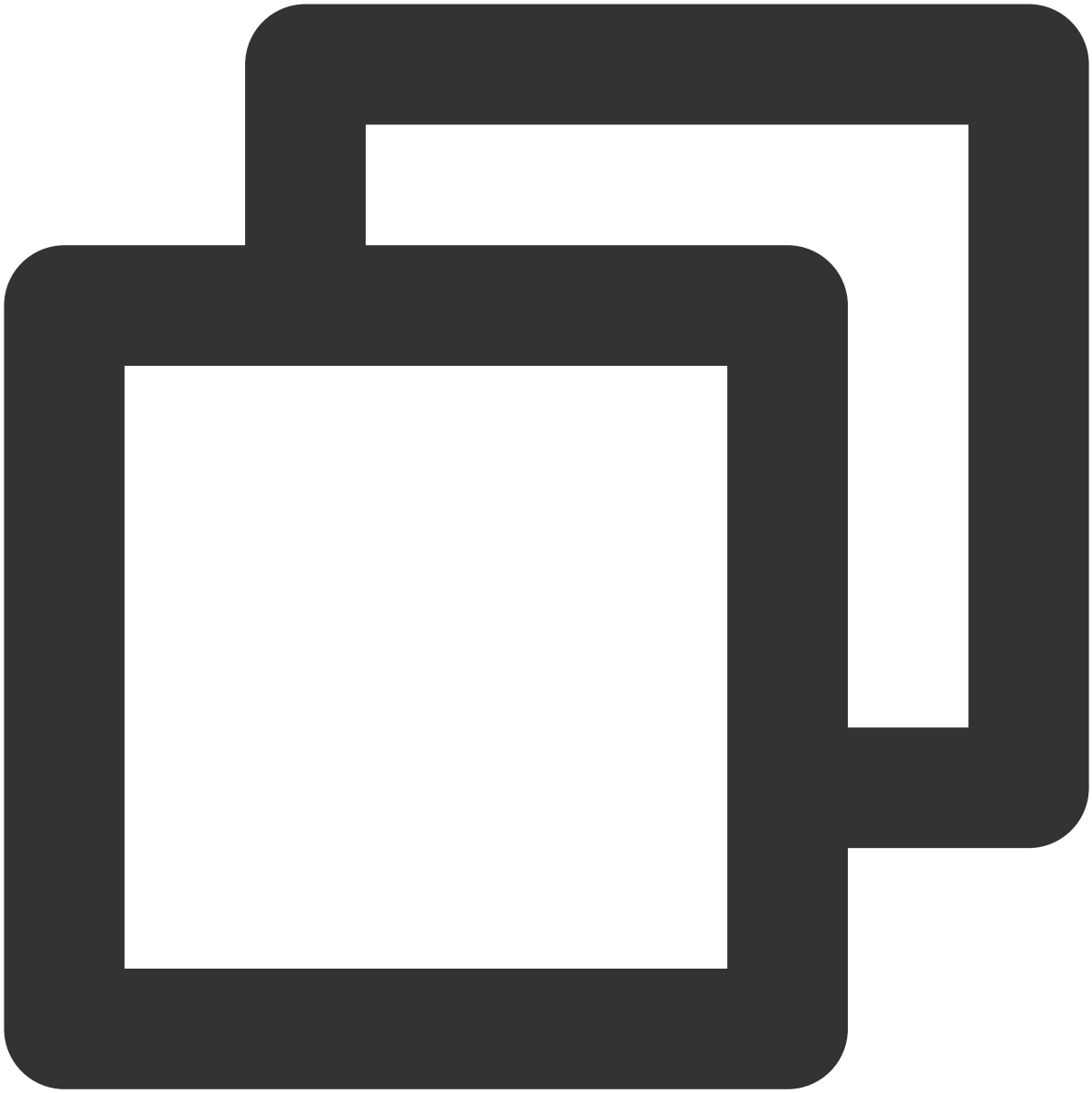
执行以下命令创建 csidriver：



```
kubectl apply -f csidriver.yaml
```

创建 CFS Turbo 存储卷

步骤1：使用以下模板创建 CFS Turbo 类型 PV



```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-cfsturbo
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: com.tencent.cloud.csi.cfsturbo
```

```
    volumeHandle: pv-cfsturbo
    volumeAttributes:
      host: *.*.*.*
      fsid: ********
      # cfs turbo subPath
      path: /
  storageClassName: ""
```

参数说明：

metadata.name：创建 PV 名称。

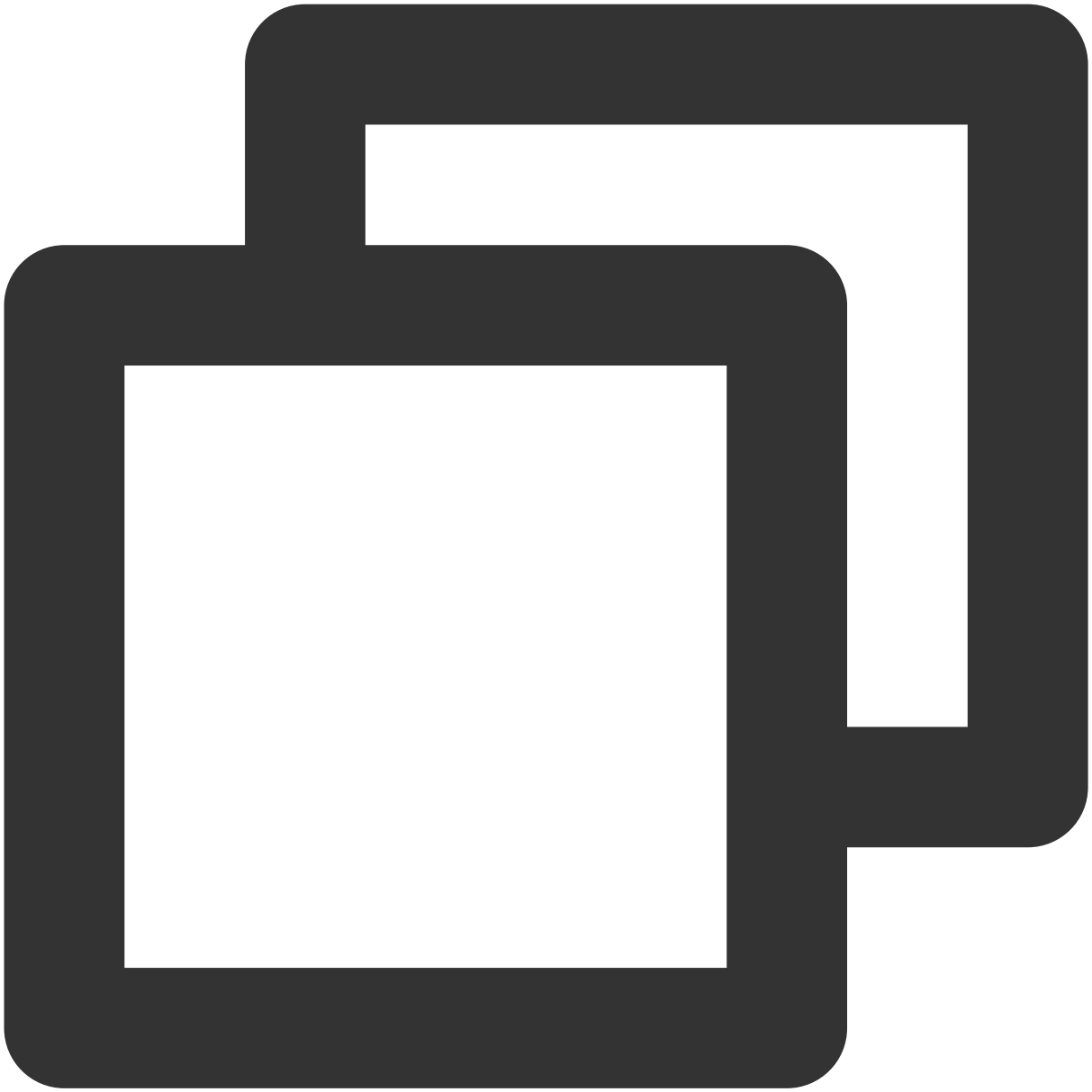
spec.csi.volumeHandle：与 PV 名称保持一致。

spec.csi.volumeAttributes.host：文件系统 ip 地址，可在文件系统挂载点信息中查看。

spec.csi.volumeAttributes.fsid：文件系统 fsid（非文件系统 id），可在文件系统挂载点信息中查看（挂载命令中 `tcp0:/` 与 `/cfs` 之间的字符串，如下图所示）。

spec.csi.volumeAttributes.path: 文件系统子目录，不填写默认为 `/`（为提高挂载性能，插件后端将 `/` 目录实际定位到 `/cfs` 目录下）。如需指定子目录挂载，须确保该子目录在文件系统 `/cfs` 中存在，挂载后 workload 将无法访问到该子目录的上层目录。例如：`path: /test`，需在文件系统中保证 `/cfs/test` 目录存在。

步骤2：使用以下模板创建 PVC 绑定 PV



```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-cfsturbo
spec:
  storageClassName: ""
  volumeName: pv-cfsturbo
  accessModes:
  - ReadWriteMany
  resources:
    requests:
```

```
      storage: 10Gi
```

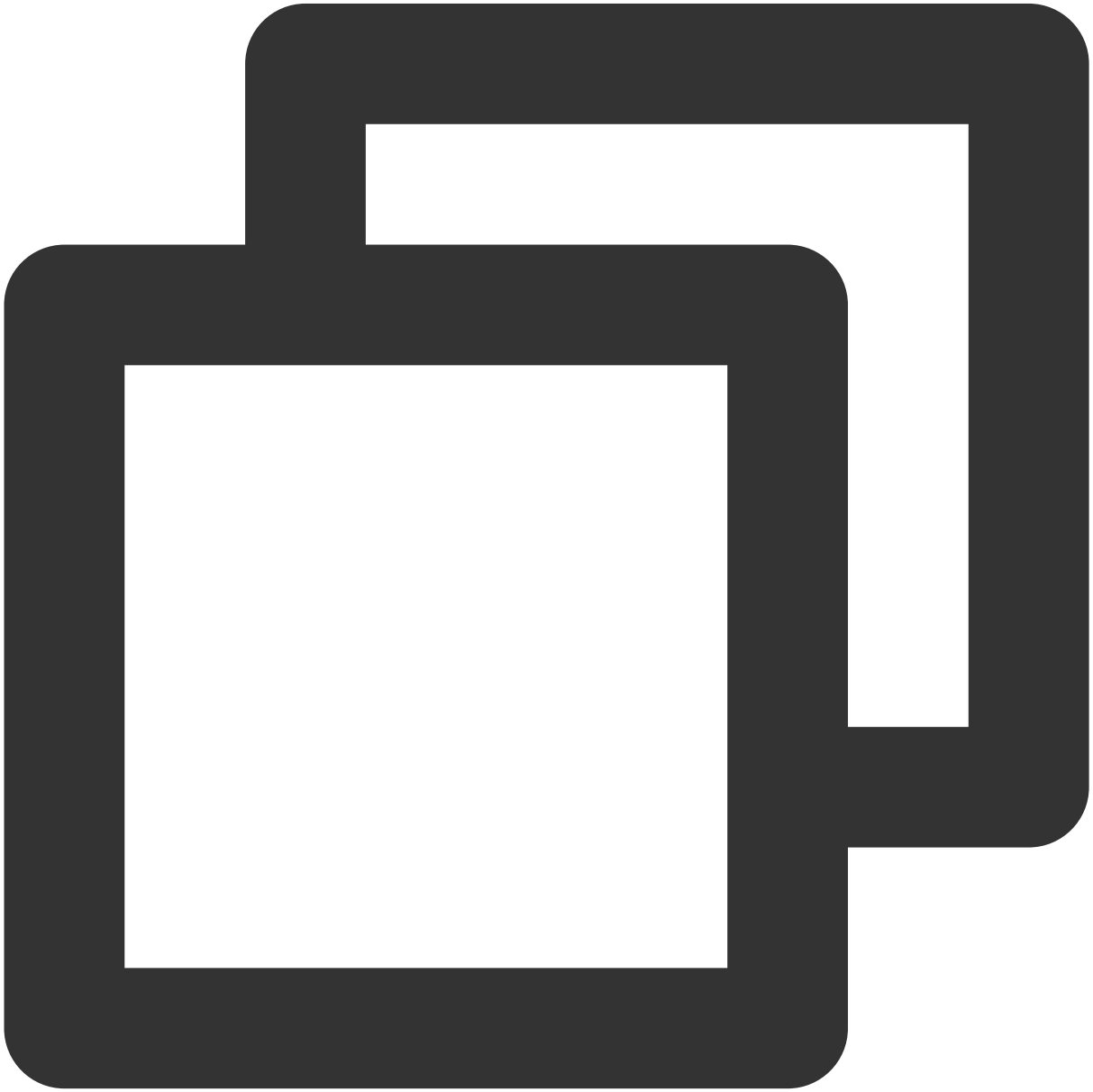
参数说明：

metadata.name：创建 PVC 名称。

spec.volumeName：与 步骤1 中创建 PV 名称保持一致。

使用 CFS Turbo 存储卷

使用以下模板创建 Pod 挂载 PVC。



```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: ccr.ccs.tencentyun.com/qcloud/nginx:1.9
    imagePullPolicy: Always
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    volumeMounts:
      - mountPath: /var/www
        name: data
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: pvc-cfsturbo
```

容器化
境外镜像拉取加速

最近更新时间：2022-11-01 10:01:52

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
- 集群运行时为 Containerd 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
```

镜像分层最佳实践

最近更新时间：2023-05-19 16:05:47

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云审计，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。

> PS: 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 Docker
file .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/alpine:v3.13
```

2.1 Dcokerfile-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-inst
ance-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/rep
o/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/do
wnload/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.co
m/f3s-tcr/centos:v7.8 uname -a
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.co
m/f3s-tcr/centos:v7.8
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencen
tcloudcr.com/f3s-tcr/centos:v7.8 sh
```

2.2 Dcokerfile-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-kon
a: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.co
m/f3s-tcr/alpine-kona:latest sh $(java -version)
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.co
m/f3s-tcr/alpine-kona:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencen
tcloudcr.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/d
ist/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:lat
```

```
est
# To test run: docker run --name test -it --rm f3s-docker-file.tencentcloudcr.co
m/f3s-tcr/alpine-kona-skywalking:latest sh $(java -version)
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.co
m/f3s-tcr/alpine-kona-skywalking:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencen
tcloudcr.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-$JMETER_VERSIO
N.tgz && \
tar -xzf apache-jmeter-$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/downl
oads/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-$JMETER_VERSION/lib/ext/pepper-box-1.0.jar http
s://github.com/raladev/load/blob/master/JARs/pepper-box-1.0.jar?raw=true && \
# 下载bzm-parallel && \
cd /jmeter/apache-jmeter-$JMETER_VERSION/ && \
wget -q -O /tmp/bzm-parallel-0.7.zip https://jmeter-plugins.org/files/packages/bz
m-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-bas
e: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-mast
er: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-slav
e: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/${owner:-github.com/8710925}/reporter/util/texlive.prof
ile /
COPY --from=build /go/src/${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://github.com/yihui/tinytex/raw/master/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/syst
ems/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-graf
ana-reporter:latest -f ./3.app/jmeter/Dockerfile-jmeter-grafana-reporter .
docker push f3s-docker-file.tencentcloudcr.com/f3s-tcr/jmeter-grafana-reporter:la
test
```

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-ngin
x: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.tencent
cloudcr.com/f3s-tcr/alpine-nginx:latest nginx -v
```

```
# docker export <container-id> | docker import f3s-docker-file.tencentcloudcr.co
m/f3s-tcr/alpine-nginx:latest
# quick interative termnal: docker run -it --entrypoint=sh f3s-docker-file.tencen
tcloudcr.com/f3s-tcr/alpine-nginx:latest sh
```

微服务
Dubbo 应用托管到 TKE

最近更新时间：2023-02-15 10:50:09

操作场景

本文介绍了 Dubbo 应用托管到腾讯云容器服务 TKE 的最佳实践。

Dubbo 应用托管到 TKE 的优势

- 提升资源利用率。
- Kubernetes 天然适合微服务架构。
- 提升运维效率，便于 Devops 落地实施。
- Kubernetes 的高弹性，可轻松实现应用的动态扩缩容。
- 容器服务 TKE 提供 Kubernetes Master 托管功能，可减少 Kubernetes 集群运维和管理的负担。
- 容器服务 TKE 和腾讯云的其他云原生产品进行了整合和优化，帮助用户更好的使用腾讯云上产品。

最佳实践实例介绍

本文以 Q 云书城（Q Cloud Book Mall，QCBM）项目为最佳实践实例，详细介绍 Dubbo 应用托管到 TKE 的过程。

QCBM 概述

QCBM 是采用微服务架构，并使用 dubbo-2.7.8 框架开发的一个网上书城 Demo 项目。QCBM 的部署和代码托管在 Coding，详情可参见 QCBM 项目。QCBM 包含以下微服务：

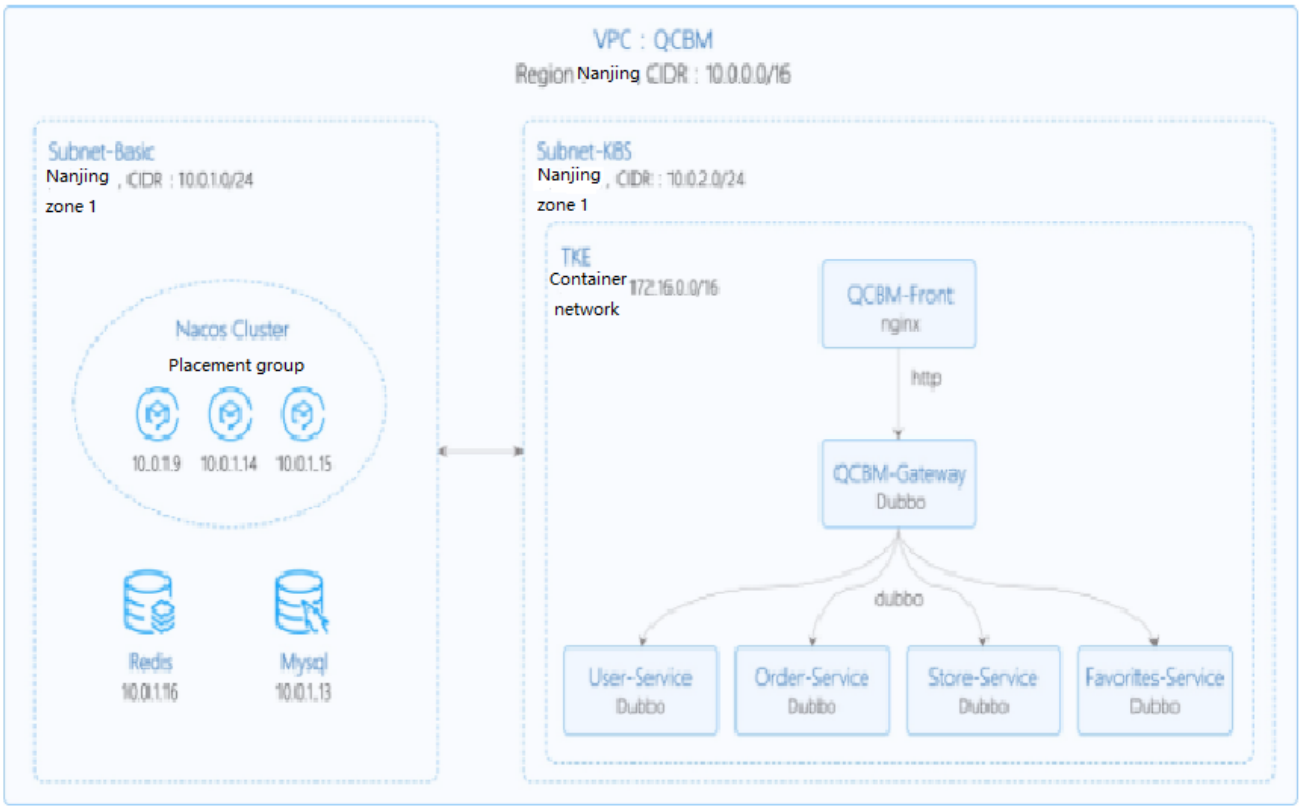
微服务	说明
QCBM-Front	使用 React 开发的前端项目，基于 Nginx 官方提供的 1.19.8 Docker 镜像 构建和部署。
QCBM-Gateway	API 网关，接受前端的 HTTP 请求，并将其转化为后台的 Dubbo 请求。
User-Service	基于 Dubbo 的微服务，提供用户注册、登录、鉴权等功能。
Favorites-Service	基于 Dubbo 的微服务，提供用户图书收藏功能。
Order-Service	基于 Dubbo 的微服务，提供用户订单生成和查询等功能。
Store-Service	基于 Dubbo 的微服务，提供图书信息的存储等功能。

QCBM 架构和组件

本文最佳实践实例模拟将原先部署在云服务器 CVM 的应用进行容器化，并托管到容器服务 TKE 的场景。在该场景中需要采用一个 VPC，并划分为以下两个子网：

- **Subnet-Basic**：部署有状态的基础服务，包括 Dubbo 的服务注册中心 Nacos、MySQL 和 Redis 等。
- **Subnet-K8S**：部署 QCBM 的应用服务，所有服务都进行容器化，并运行在容器服务 TKE 上。

子网划分如下图所示：



QCBM 实例的网络规划如下表所示：

网络规划	说明
Region/AZ	南京/南京一区
VPC	CIDR：10.0.0.0/16

网络规划	说明
子网 Subnet-Basic	南京一区，CIDR：10.0.1.0/24
子网 Subnet-K8S	南京一区，CIDR：10.0.2.0/24
Nacos 集群	采用3台 “标准型SA2” 1C2G 机型的 CVM 构建 Nacos 集群，对应的 IP 为：10.0.1.9，10.0.1.14，10.0.1.15

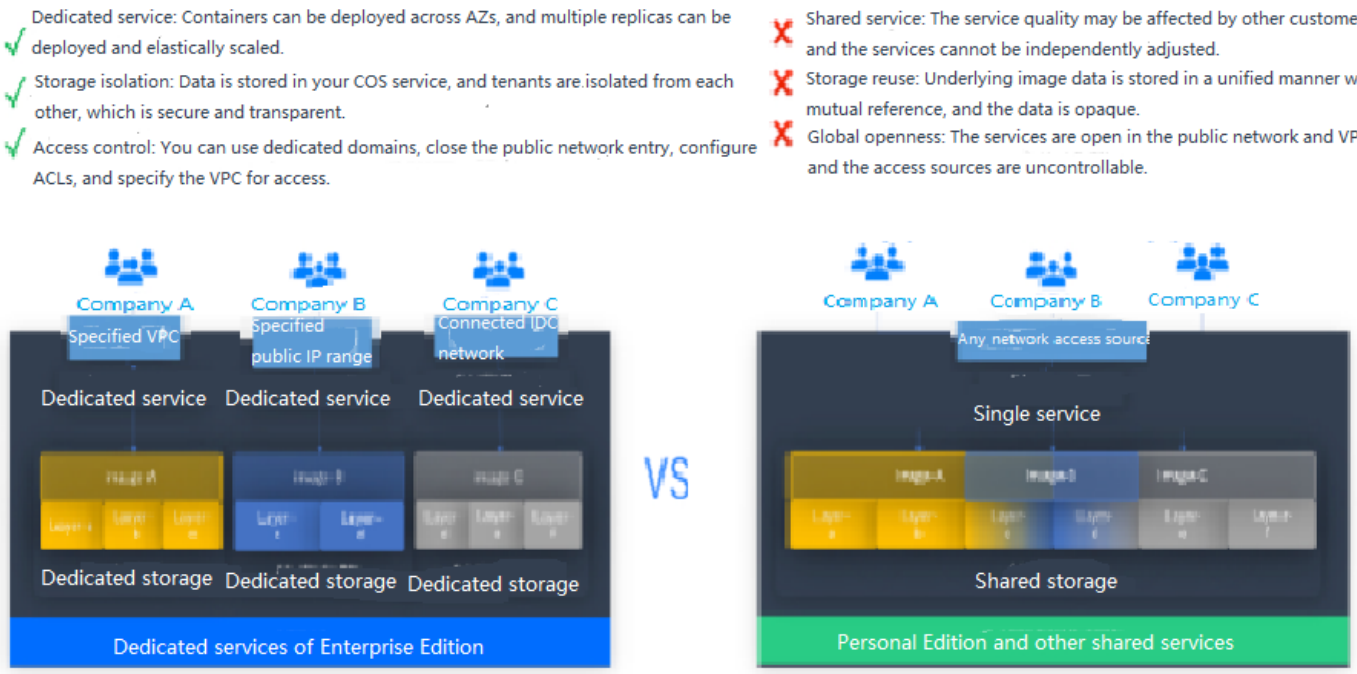
QCBM 实例中用到的组件如下表所示：

组件	版本	来源	备注
k8s	1.8.4	腾讯云	TKE 托管模式
MySQL	5.7	腾讯云	TencentDB for MySQL 双节点
Redis	5.0	腾讯云	TencentDB for Redis 标准型
CLS	N/A	腾讯云	日志服务
TSW	N/A	腾讯云	采用 Skywalking 8.4.0 版的 Agent 接入，点此 下载
Java	1.8	开源社区	Docker 镜像为 java:8-jre
Nacos	2.0.0	开源社区	点此 下载
Dubbo	2.7.8	开源社区	Github 地址

服务介绍

TCR 介绍

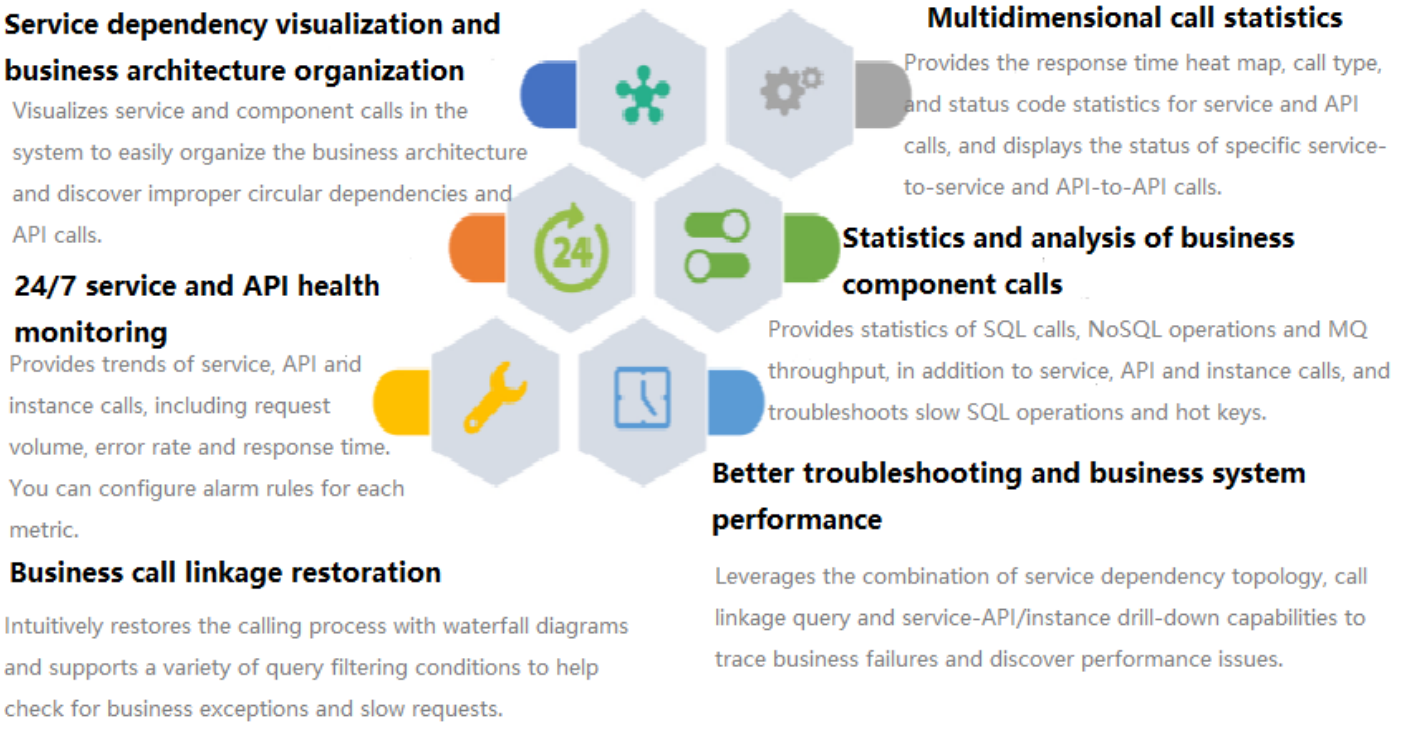
腾讯云 容器镜像服务 TCR 提供个人版和企业版两种镜像仓库。两者区别如下图所示：



QCBM 是一个 Dubbo 容器化的 Demo 项目，因此容器镜像服务个人版完全满足需求。但对于企业用户，推荐使用容器镜像服务企业版。如需使用镜像仓库，请参见 镜像仓库基本操作。

TSW 介绍

腾讯微服务观测平台 TSW（Tencent Service Watcher）提供云原生服务可观察性解决方案，能够追踪到分布式架构中的上下游依赖关系，绘制拓扑图，提供服务、接口、实例、中间件等多维度调用观测。详细介绍如下图所示：



TSW 在架构上分为以下四大模块：

展开全部

数据采集（Client）

展开&收起

使用开源探针或 SDK 用于采集数据。对于迁移上云的用户，可保留 Client 端的大部分配置，仅更改上报地址和鉴权信息即可。

数据处理（Server）

展开&收起

数据经由 Pulsar 消息队列上报到 Server，同时 Adapter 会将数据转换为统一的 Opentracing 兼容格式。根据数据的使用场景，分配给实时计算与离线计算：

- 实时计算提供实时监控、统计数据展示，并对接告警平台快速响应。
- 离线计算处理长时段大量数据的统计汇聚，利用大数据分析能力提供业务价值。

存储（Storage）

展开&收起

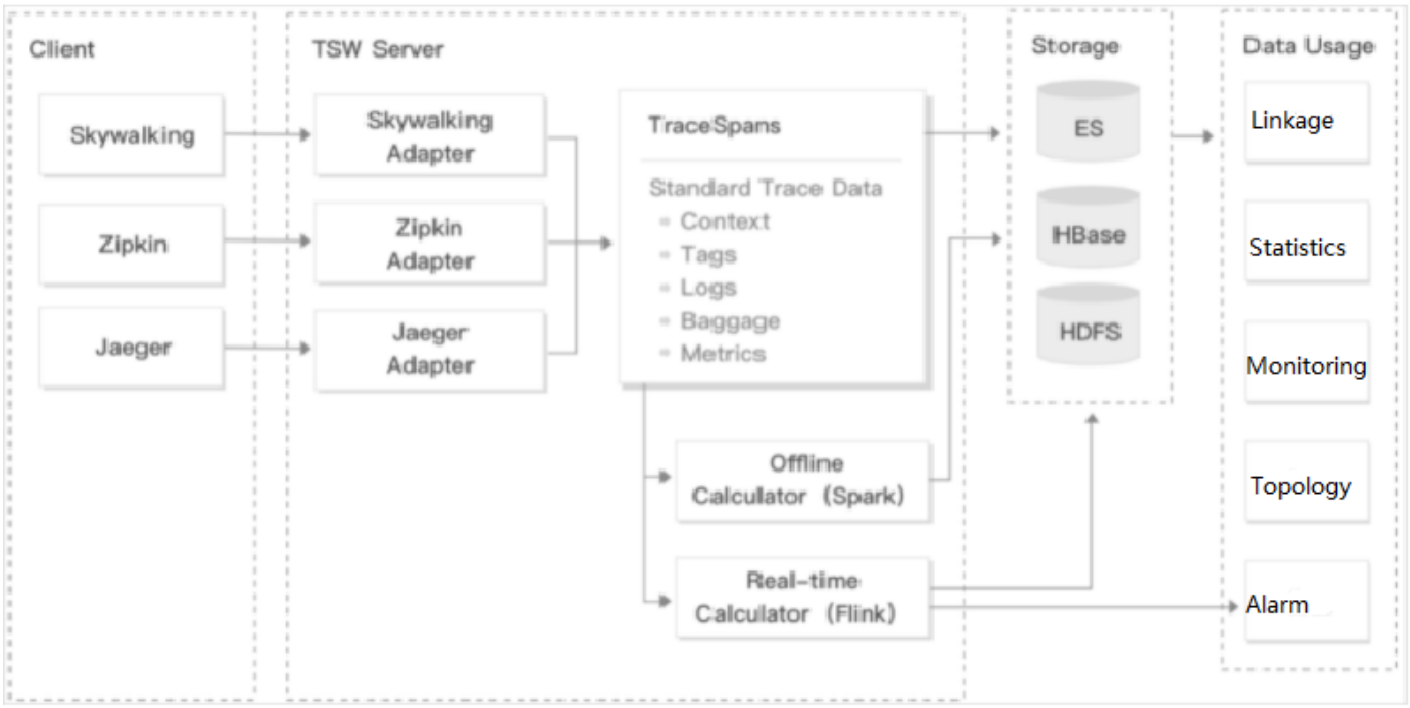
存储层可满足不同数据类型的使用场景，适配 Server 层的写入与 Data Usage 层的查询与读取请求。

数据使用（Data Usage）

展开&收起

为控制台操作、数据展示、告警提供底层支持。

架构图如下所示：



操作步骤

搭建基础服务集群

- 在 Mysql 控制台 创建实例，并使用 qcbm-ddl.sql 初始化。详情请参见 创建 MySQL 实例。
- 在 Redis 控制台 创建实例并初始化。详情请参见 创建 Redis 实例。

- 在 负载均衡控制台 为子网 Subnet-K8S 新建一个**内网型**的负载均衡（后续实践中会使用到该 CLB 实例 ID）。详情请参见 创建负载均衡实例。
- 申请通过 TSW 内测。TSW 目前处于内测阶段，支持 Java 和 Golang 两种语言接入。
- 部署 Nacos 集群：
  i. 在 云服务器控制台 购买3台 “标准型SA2” 1核2G的云服务器，详情请参见 通过购买页创建实例。
  ii. 登录实例，执行以下命令安装 Java。

```
yum install java-1.8.0-openjdk.x86_64
```

  执行以下命令，如有输出 java 版本信息，则说明 java 安装成功。

```
java - version
```

  iii. 部署 Nacos 集群，详情请参见 Nacos 官方文档 集群部署说明 。

构建 Docker 镜像

编写 Dockerfile

下文以 user-service 为例为您简单介绍如何编写 Dockerfile。示例展示的是 user-service 的工程目录结构，**Dockerfile** 位于工程的根目录下，**user-service-1.0.0.zip** 是打包后的文件，需要添加到镜像中。

```
➜ user-service tree
├── Dockerfile
├── assembly
│ ....
├── bin
│ ....
├── pom.xml
├── src
│ ....
├── target
│ .....
│ └── user-service-1.0.0.zip
└── user-service.iml
```

user-service 的 Dockerfile 如下所示：

```
FROM java:8-jre

ARG APP_NAME=user-service
ARG APP_VERSION=1.0.0
ARG FULL_APP_NAME=${APP_NAME}-${APP_VERSION}

# 容器中的工作目录为 /app
```

```
WORKDIR /app

# 将本地打包出来的应用添加到镜像中
COPY ./target/${FULL_APP_NAME}.zip .

# 创建日志目录 logs，解压并删除原始文件和解压后的目录
RUN mkdir logs \
&& unzip ${FULL_APP_NAME}.zip \
&& mv ${FULL_APP_NAME}/** . \
&& rm -rf ${FULL_APP_NAME}*

# user-service 的启动脚本和参数
ENTRYPOINT ["/app/bin/user-service.sh"] CMD ["start", "-t"]

# dubbo 端口号
EXPOSE 20880
```

注意：

- 生产中的 Java 应用有很多配置参数，导致启动脚本很复杂。将启动脚本里的内容全部写到 dockerfile 中工作量很大，其次 dockerfile 远没有 Shell 脚本灵活，若出现问题也无法快速定位，因此不建议弃用启动脚本。
- 通常在启动脚本最后使用 **nohup** 启动 Java 应用，但该方式启动的 deamon 进程会导致容器运行后直接退出。因此 `nohup java ${OPTIONS} -jar user-service.jar > ${LOG_PATH} 2>&1 &` 需改成 `java ${OPTIONS} -jar user-service.jar > ${LOG_PATH} 2>&1` 。
- Dockerfile 中每多一个 RUN 命令，生成的镜像就多一层，推荐将这些 RUN 命令合成一条。

构建镜像

容器镜像服务 TCR 提供了自动和手工构建镜像方式。为展示具体的构建过程，本文采用手工构建方式。

镜像名称需要符合规范 `ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]` ：

- 其中命名 namespace 为方便镜像管理使用，可以按项目取名。本文采用 QCBM 表示 Q 云书城项目下的所有镜像。
- ImageName 可以包含 subpath，一般用于企业用户多项目场景。此外，如果本地已构建好镜像，可使用 `docker tag` 命令，按命名规范对镜像重命名。

1. 执行以下命令构建镜像。示例如下：

```
# 推荐的构建方式，可省去二次打 tag 操作
sudo docker build -t ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

```
# 本地构建 user-service 镜像，最后一个 . 表示 Dockerfile 存放在当前目录（user-service）下
➜ user-service docker build -t ccr.ccs.tencentyun.com/qcbm/user-service:1.0.0 .
# 将已存在镜像按命名规范对镜像重命名
sudo docker tag [ImageId] ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

2. 构建完成后，可执行以下命令查看本地仓库中的所有镜像。

```
docker images
```

示例如下图所示：

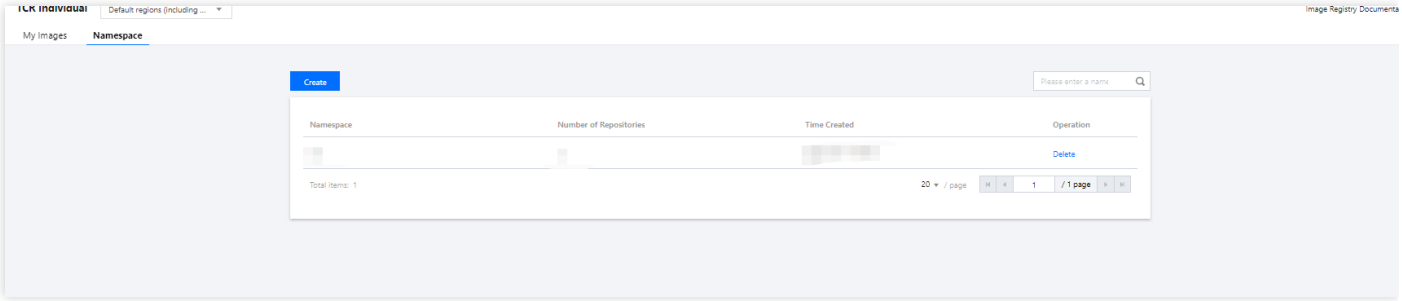
```
➜ qcbm docker images
REPOSITORY                                      TAG                 IMAGE ID            CREATED             SIZE
ccr.ccs.tencentyun.com/qcbm/qcbm-gateway        1.0.0               b9516e1a0717        About an hour ago   558MB
ccr.ccs.tencentyun.com/qcbm/favorites-service   1.0.0               157465cc30f2        About an hour ago   512MB
ccr.ccs.tencentyun.com/qcbm/order-service       1.0.0               aad52ddfc3d7        About an hour ago   512MB
ccr.ccs.tencentyun.com/qcbm/store-service       1.0.0               a7fcc435820f        About an hour ago   509MB
ccr.ccs.tencentyun.com/qcbm/user-service        1.0.0               cdc6910691ef        About an hour ago   512MB
java                                            8-jre               e44d62cf8862        4 years ago         311MB
➜ qcbm
```

上传镜像到 TCR

创建命名空间

QCBM 项目采用个人版镜像仓库（建议企业客户使用企业版镜像仓库）。

1. 登录 容器服务控制台 。
2. 选择**镜像仓库** > **个人版** > **命名空间**进入“命名空间”页面。
3. 单击**新建**，在弹出的新建命名窗口中新建命名空间 qcbm。QCBM 项目所有的镜像都存放于该命名空间下。如下图所示：



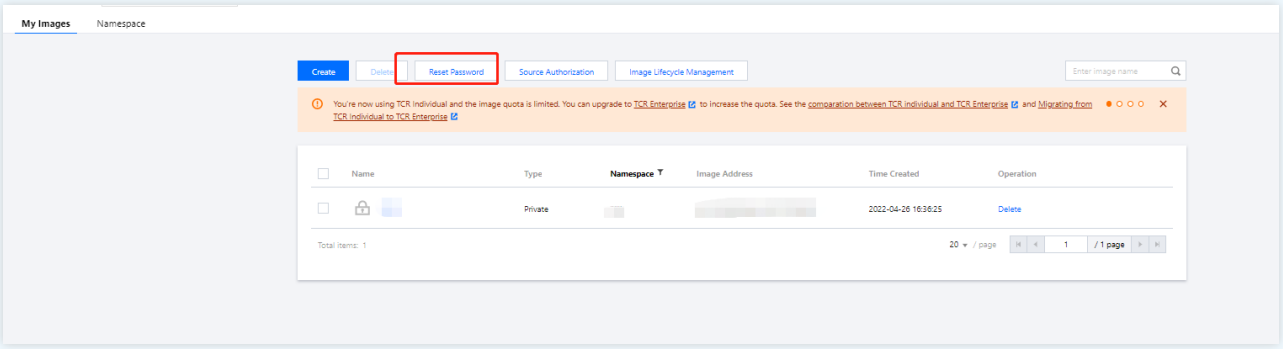
上传镜像

上传镜像需要完成以下步骤：登录腾讯云 registry 和上传镜像。

1. 执行以下命令登录腾讯云 registry。

```
docker login --username=[腾讯云账号 ID] ccr.ccs.tencentyun.com
```

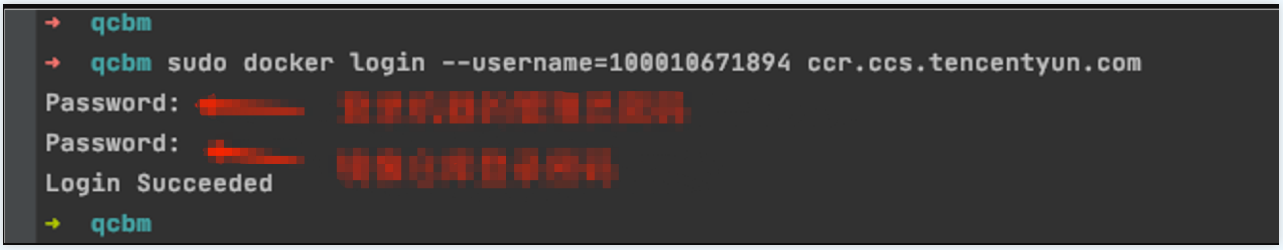
- 腾讯云账号 ID 可在 账号信息 页面获取。
- 若忘记**镜像仓库登录密码**，可前往容器服务镜像仓库个人版 我的镜像 中进行重置。



- 若执行命令提示无权限，请在上述命令前加上 sudo 再执行，如下所示。此时需要输入两个密码，第一个为 sudo 所需的主机管理员密码，第二个为**镜像仓库登录密码**。

```
sudo docker login --username=[腾讯云账号 ID] ccr.ccs.tencentyun.com
```

如下图所示：



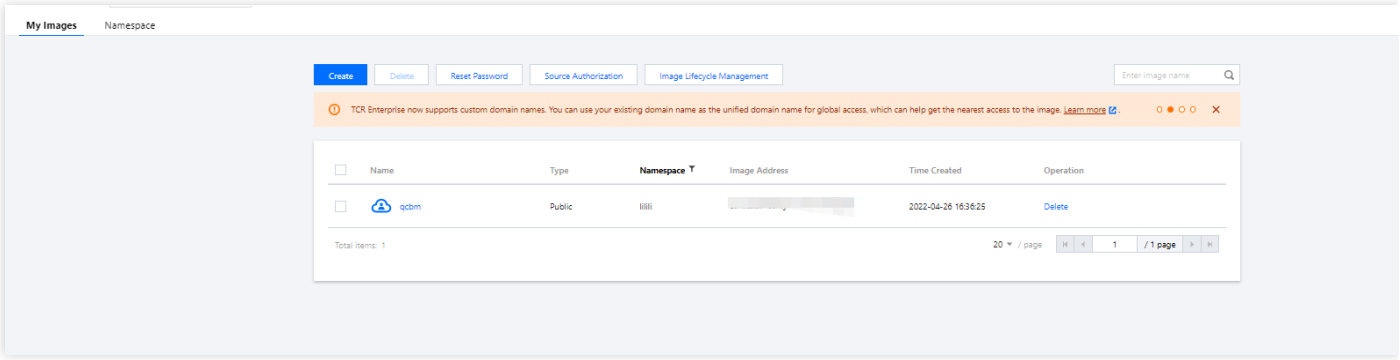
2. 执行以下命令将本地生成的镜像推送至 TKE 的镜像仓库中。

```
docker push ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

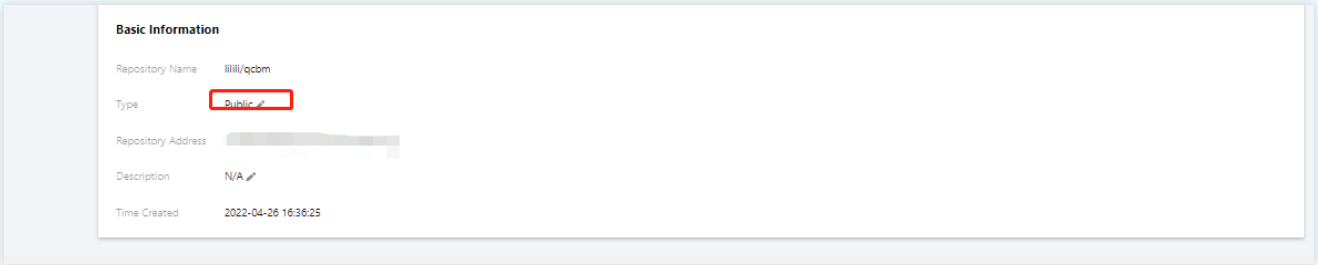
如下图所示：

```
➜  user-service docker push ccr.ccs.tencentyun.com/qcbm/user-service:1.0.1
The push refers to repository [ccr.ccs.tencentyun.com/qcbm/user-service]
bebcf5e72f77: Pushed
958f8e83f873: Pushed
a177e9d322e4: Pushed
73ad47d4bc12: Layer already exists
c22c27816361: Layer already exists
04dba64afa87: Layer already exists
500ca2ff7d52: Layer already exists
782d5215f910: Layer already exists
0eb22bfb707d: Layer already exists
a2ae92ffcd29: Layer already exists
1.0.1: digest: sha256:4af3e7ed8203a1bc92baf108ac8f65b8b00de750367e680dde4c1673bf90dd29 size: 2418
```

3. 在 我的镜像 中可以查看上传的所有镜像，下图展示的是上传到腾讯云镜像仓库中 QCBM 的5个镜像。



默认镜像类型为“私有”，如需提供镜像给他人使用，可在**镜像信息**中将镜像类型设置为公有。如下图所示：



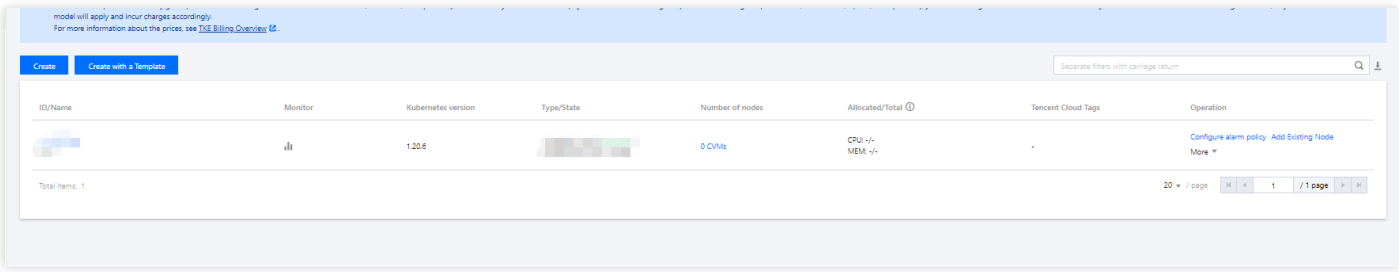
在 TKE 上部署服务

创建 k8s 集群 QCBM

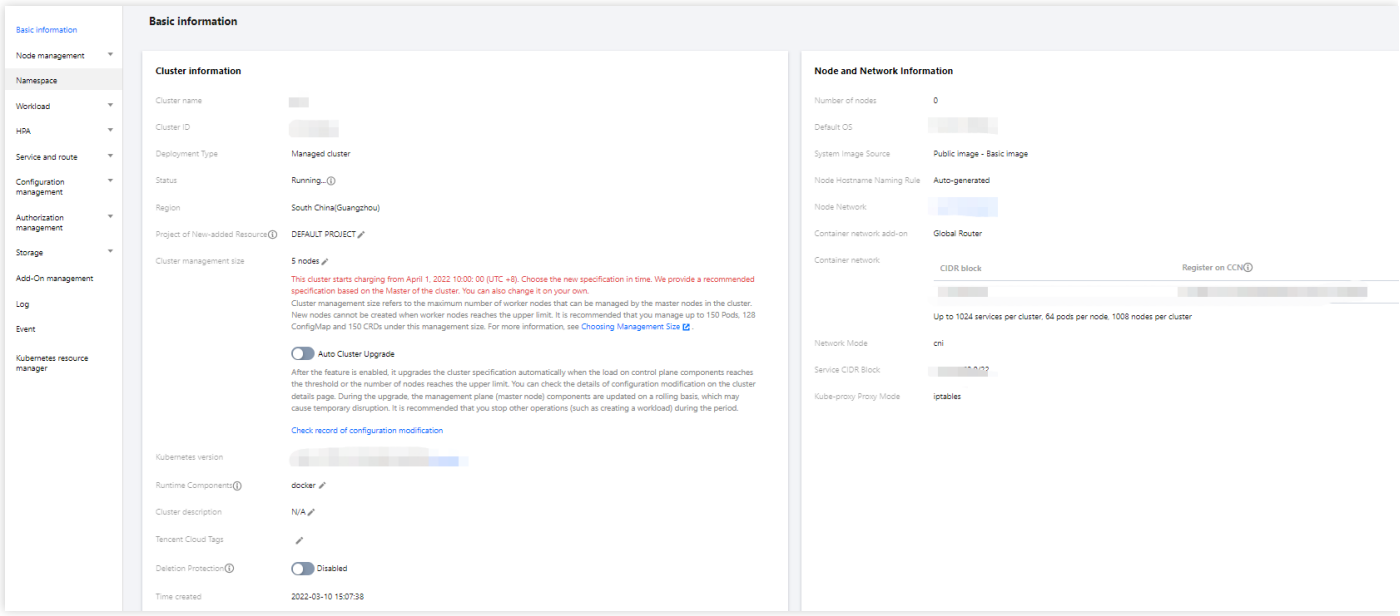
1. 实际部署前，需要新建一个 k8s 集群。有关集群的创建，请参见 创建集群 文档。

> 注意：
> 创建集群时，在“选择机型”页面建议开启“置放群组功能”，该功能可将 CVM 打散到不同母机上，增加系统可靠性。

2. 集群创建完成后，在容器服务控制台 的 集群管理 页面可以查看新建的集群信息。本文新建的集群名称为 qcbm-k8s-demo。如下图所示：

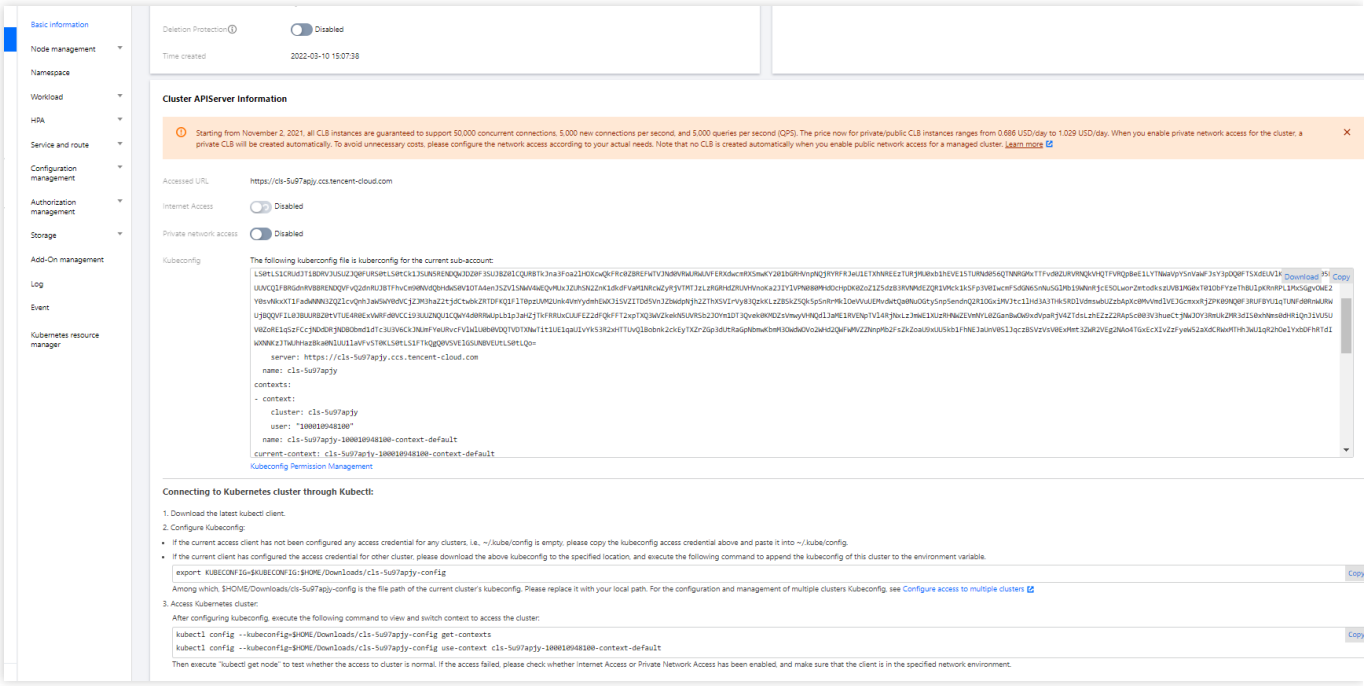


3. 单击集群名称进入“基本信息”页面，查看集群的配置信息。如下图所示：



4. （可选）如需使用 Kubectl 和 lens 等 k8s 管理工具，还需进行以下两步操作：
   i. 开启外网访问。
   ii. 将 API 认证 Token 保存在本地 `用户 home/.kube` 下的 config 文件中（若 config 文件已有内容，则需要替换），以确保每次访问都能进入默认集群中。如果选择不将 API 认证 Token 保存在 `.kube` 下的 config 文件

中，则可参考控制台**集群APIServer信息**下的 **通过Kubectl连接Kubernetes集群操作说明**。如下图所示：



创建 Namespace

Namespaces 是 Kubernetes 在同一个集群中进行逻辑环境划分的对象，通过 Namespaces 可以进行多个团队多个项目的划分。您可以通过以下三种方式创建 Namespace，推荐使用方式1命令行方式创建。

- 方式1：使用命令行
- 方式2：使用控制台
- 方式3：使用 YAML 部署

执行以下命令即可创建 Namespace：

```
kubectl create namespace qcbm
```

ConfigMap 存放配置信息

通过 ConfigMap 可以将配置和运行的镜像进行解耦，使应用程序有更强的移植性。QCBM 后端服务需要从环境变量中获取 Nacos、MySQL、Redis 主机和端口信息，并将其使用 ConfigMap 进行保存。
您可通过以下两种方式使用 ConfigMap 存放配置信息：

- 方式1：使用 YAML
- 方式2：使用控制台

下文为 QCBM 的 ConfigMap YAML，其中**纯数字类型的 value 需要使用双引号**。例如，下文示例 YAML 中的 MYSQL_PORT：

```
# 创建 ConfigMap

apiVersion: v1
kind: ConfigMap
metadata:
name: qcbm-env
namespace: qcbm
data:
NACOS_HOST: 10.0.1.9
MYSQL_HOST: 10.0.1.13
REDIS_HOST: 10.0.1.16
NACOS_PORT: "8848"
MYSQL_PORT: "3306"
REDIS_PORT: "6379"
SW_AGENT_COLLECTOR_BACKEND_SERVICES: xxx # TSW 接入地址，后文介绍
```

使用 Secret 存放敏感信息

Secret 可用于存储密码、令牌、密钥等敏感信息，降低直接对外暴露的风险。QCBM 使用 Secret 来保存相关的账号和密码信息。
您可通过以下两种方式使用 Secret 存放敏感信息：

- 方式1：使用 YAML
- 方式2：使用控制台

下文为 QCBM 创建 Secret 的 YAML。其中 Secret 的 value 需要是 base64 编码后的字符串。

```
# 创建 Secret
apiVersion: v1
kind: Secret
metadata:
name: qcbm-keys
namespace: qcbm
labels:
qcloud-app: qcbm-keys
data:
# xxx 为base64 编码后的字符串，可使用 shell 命令 “echo -n 原始字符串 | base64” 生成
MYSQL_ACCOUNT: xxx
MYSQL_PASSWORD: xxx
REDIS_PASSWORD: xxx
SW_AGENT_AUTHENTICATION: xxx # TSW 接入 token，后文介绍
type: Opaque
```

部署工作负载 Deployment

Deployment 声明了 Pod 的模板和控制 Pod 的运行策略，适用于部署无状态的应用程序。QCBM 的 front 和 Dubbo 服务都属于无状态应用，适合使用 Deployment。

以下是 user-service Deployment 的 YAML 参数说明：

参数	说明
replicas	表示需要创建的 pod 数量
image	镜像的地址
imagePullSecrets	拉取镜像时需要使用的 key，可在 **集群**>**配置管理** > **Secret**中获取。使用公共镜像时可省略
env	• 定义了 pod 的环境变量和取值 • ConfigMap 中定义的 key-value 可使用 configMapKeyRef 引用 • Secret 中定义的 key-value 可使用 secretKeyRef 引用
ports	指定容器的端口号，由于是 Dubbo 应用，所以端口号为20880

user-service Deployment 的 完整 YAML 文件示例如下：

```
# user-service Deployment

apiVersion: apps/v1
kind: Deployment
metadata:
name: user-service
namespace: qcbm
labels:
app: user-service
version: v1
spec:
replicas: 1
selector:
matchLabels:
app: user-service
version: v1
template:
metadata:
labels:
app: user-service
version: v1
```

```
spec:
containers:
- name: user-service
image: ccr.ccs.tencentyun.com/qcbm/user-service:1.1.4
env:
- name: NACOS_HOST # dubbo服务注册中心nacos的IP地址
valueFrom:
configMapKeyRef:
key: NACOS_HOST
name: qcbm-env
optional: false
- name: MYSQL_HOST # Mysql 地址
valueFrom:
configMapKeyRef:
key: MYSQL_HOST
name: qcbm-env
optional: false
- name: REDIS_HOST # Redis的IP地址
valueFrom:
configMapKeyRef:
key: REDIS_HOST
name: qcbm-env
optional: false
- name: MYSQL_ACCOUNT # Mysql 账号
valueFrom:
secretKeyRef:
key: MYSQL_ACCOUNT
name: qcbm-keys
optional: false
- name: MYSQL_PASSWORD # Mysql 密码
valueFrom:
secretKeyRef:
key: MYSQL_PASSWORD
name: qcbm-keys
optional: false
- name: REDIS_PASSWORD # Redis 密码
valueFrom:
secretKeyRef:
key: REDIS_PASSWORD
name: qcbm-keys
optional: false
- name: SW_AGENT_COLLECTOR_BACKEND_SERVICES # Skywalking 后端服务地址
valueFrom:
configMapKeyRef:
key: SW_AGENT_COLLECTOR_BACKEND_SERVICES
name: qcbm-env
optional: false
```

```
- name: SW_AGENT_AUTHENTICATION # Skywalking agent 连接后端服务的认证 token
valueFrom:
secretKeyRef:
key: SW_AGENT_AUTHENTICATION
name: qcbm-keys
optional: false
ports:
- containerPort: 20880 # dubbo 端口号
protocol: TCP
imagePullSecrets: # 拉取镜像时需要使用的 key，QCBM 所有服务的镜像已开放为公共镜像，故此处可省略
- name: qcloudregistrykey
```

部署服务 Service

Kubernetes 的 ServiceTypes 允许指定 Service 类型，默认为 ClusterIP 类型。ServiceTypes 可取如下值：

- LoadBalancer：提供公网、VPC、内网访问。
- NodePort：可通过“云服务器 IP + 主机端口”访问服务。
- ClusterIP：可通过“服务名 + 服务端口”访问服务。

对于实际生产系统来说，gateway 需要能在 VPC 或内网范围内进行访问， front 前端需要能对内/外网提供访问。因此，QCBM 的 gateway 和 front 需要制定 LoadBalancer 类型的 ServiceType。
TKE 对 LoadBalancer 模式进行了扩展，通过 Annotation 注解配置 Service，可实现更丰富的负载均衡能力。

若使用 `service.kubernetes.io/qcloud-loadbalancer-internal-subnetid` 注解，在 service 部署时，会创建内网类型 CLB。一般建议事先创建好 CLB，service 的部署 YAML 中使用注解 `service.kubernetes.io/loadbalance-id` 直接指定，可提升部署效率。

以下为 qcbm-front service 部署 YAML：

```
# 部署 qcbm-front service
apiVersion: v1
kind: Service
metadata:
name: qcbm-front
namespace: qcbm
annotations:
# Subnet-K8S 子网的 CLB 实例 ID
service.kubernetes.io/loadbalance-id: lb-66pq34pk
spec:
externalTrafficPolicy: Cluster
ports:
- name: http
port: 80
```

```
targetPort: 80
protocol: TCP
selector: # 将后端服务 qcbm-gateway 和该 Service 进行映射
app: qcbm-front
version: v1
type: LoadBalancer
```

部署 Ingress

Ingress 是允许访问到集群内 Service 规则的集合。一般使用 Ingress 提供对外访问，而不直接暴露 Service 。QCBM 项目需要为 qcbm-front 创建 Ingress，对应的 YAML 如下：

```
# 部署 qcbm-front ingress

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
name: front
namespace: qcbm
annotations:
ingress.cloud.tencent.com/direct-access: "false"
kubernetes.io/ingress.class: qcloud
kubernetes.io/ingress.extensiveParameters: '{"AddressIPVersion":"IPV4"}'
kubernetes.io/ingress.http-rules: '[{"host":"qcbm.com","path":"/","backend":{"se
rviceName":"qcbm-front","servicePort":"80"}}]'
spec:
rules:
- host: qcbm.com
http:
paths:
- path: /
backend: # 关联到后端服务
serviceName: qcbm-front
servicePort: 80
```

查看部署结果

至此，您已完成 QCBM 在容器服务 TKE 上的部署，可通过以下步骤查看部署结果：

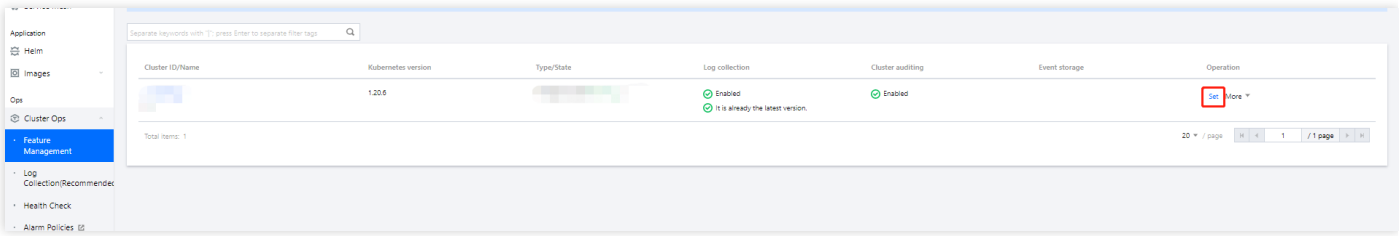
1. 登录 容器服务控制台 ，单击集群 ID/名称进入集群详情页面。
2. 单击**服务与路由** > **Ingress**进入 Ingress 页面，可查看到创建的 Ingress。通过 Ingress 的 VIP 即可访问 Q 云书城页面。

集成 CLS 日志服务

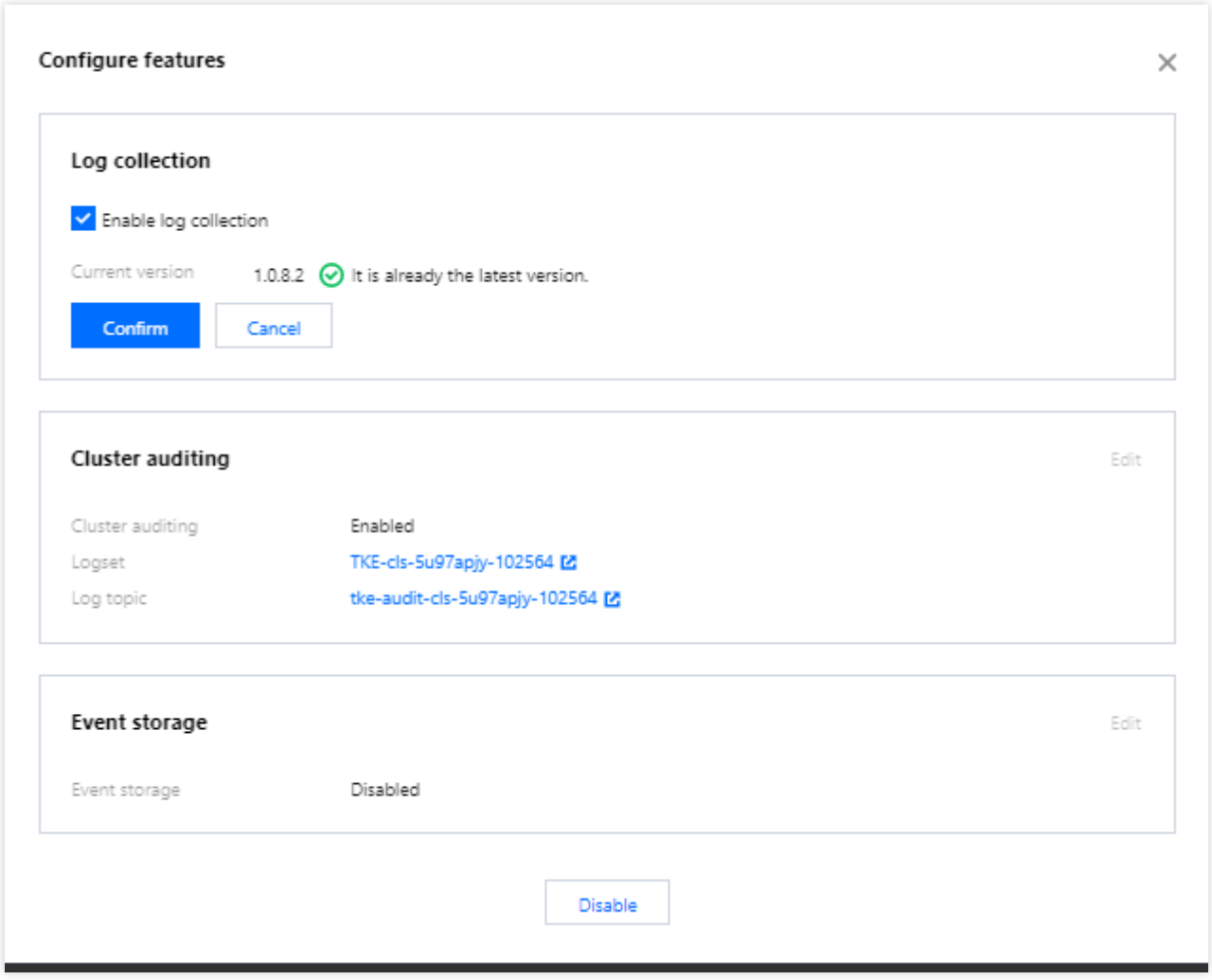
开启容器日志采集功能

容器日志采集功能默认关闭，使用前需要开启，步骤如下：

1. 登录容器服务控制台，选择左侧导航栏中的**集群运维** > **功能管理**。
2. 在“功能管理”页面上方选择地域，单击需要开启日志采集的集群右侧的**设置**。



3. 在“设置功能”页面，单击日志采集**编辑**并勾选**开启日志采集**。如下图所示：

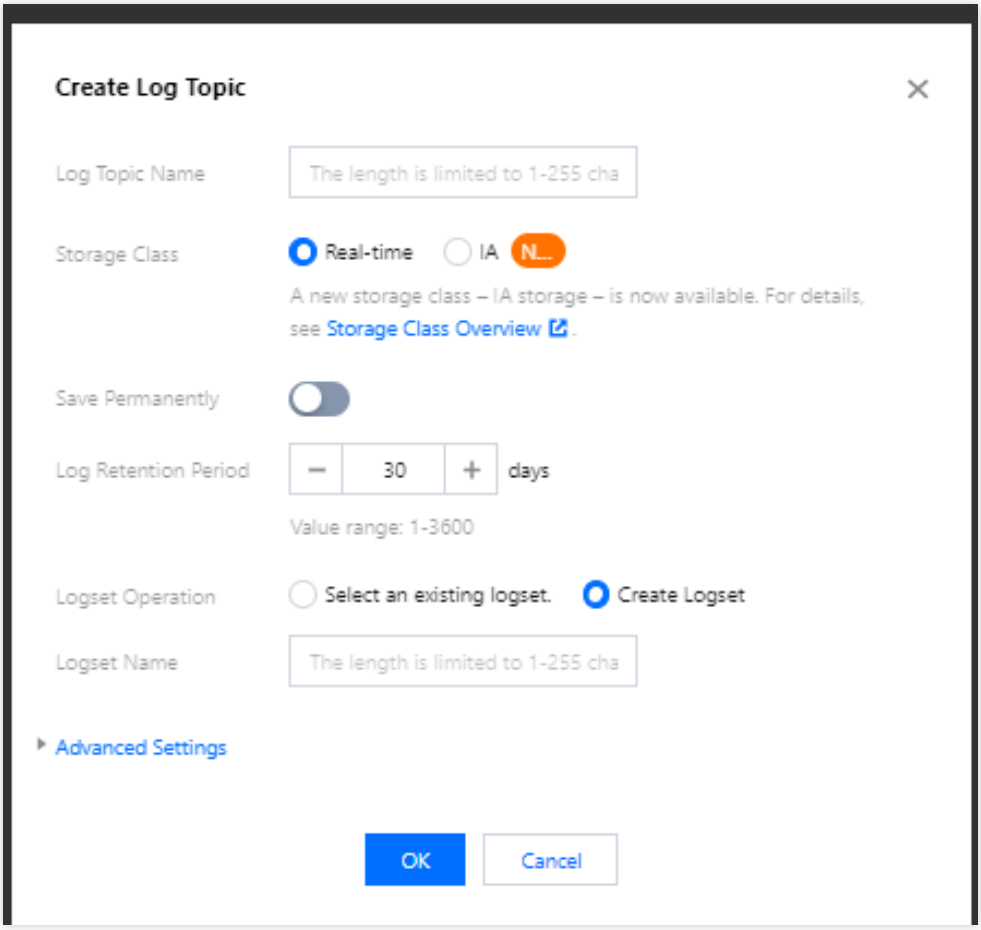


4. 单击**确定**即可开启容器日志采集功能。

创建日志主题和日志集

QCBM 部署在南京地域，因此在创建日志集时应当选择南京地域：

1. 登录 日志服务控制台，在“日志主题”页面选择南京地域。
2. 单击**创建日志主题**，在弹出的窗口中根据页面提示填写相关信息，如下图所示：



- **日志主题名称**：输入 qcbm。
- **日志集操作**：选择**创建日志集**。
- **日志集名称**：输入 qcbm-logs。

3. 单击**确定**即可创建日志主题和日志集。

> 说明：
> QCBM 有多个后端微服务，为每个微服务建个日志主题便于日志归类。
>
> - QCBM 每个服务都建立了一个日志主题。
> - 日志主题 ID，为容器创建日志规则时需要用到。

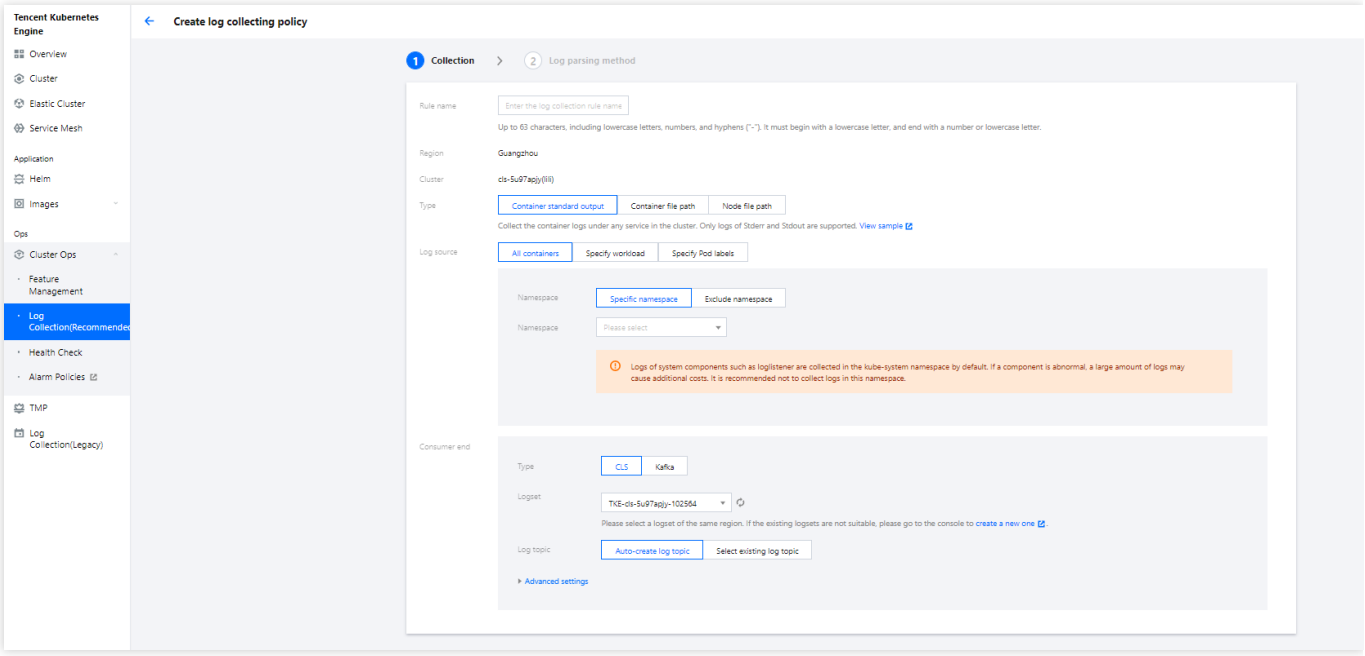
配置日志采集规则

您可通过控制台或 CRD 两种方式配置容器日志采集规则。

- 方式1：使用控制台
- 方式2：使用 CRD

日志规则指定了日志在容器内的位置：

1. 登录 容器服务控制台 ，选择左侧导航栏中的**集群运维 > 日志规则**。
2. 在“日志规则”页面，单击**新建**新建日志规则：
   - **日志源**：指定容器日志位置，QCBM 的日志都统一输出到 /app/logs 目录下，因而使用容器文件路径并指定具体的工作负载和日志位置。
   - **消费端**：选择之前创建的日志集和主题。

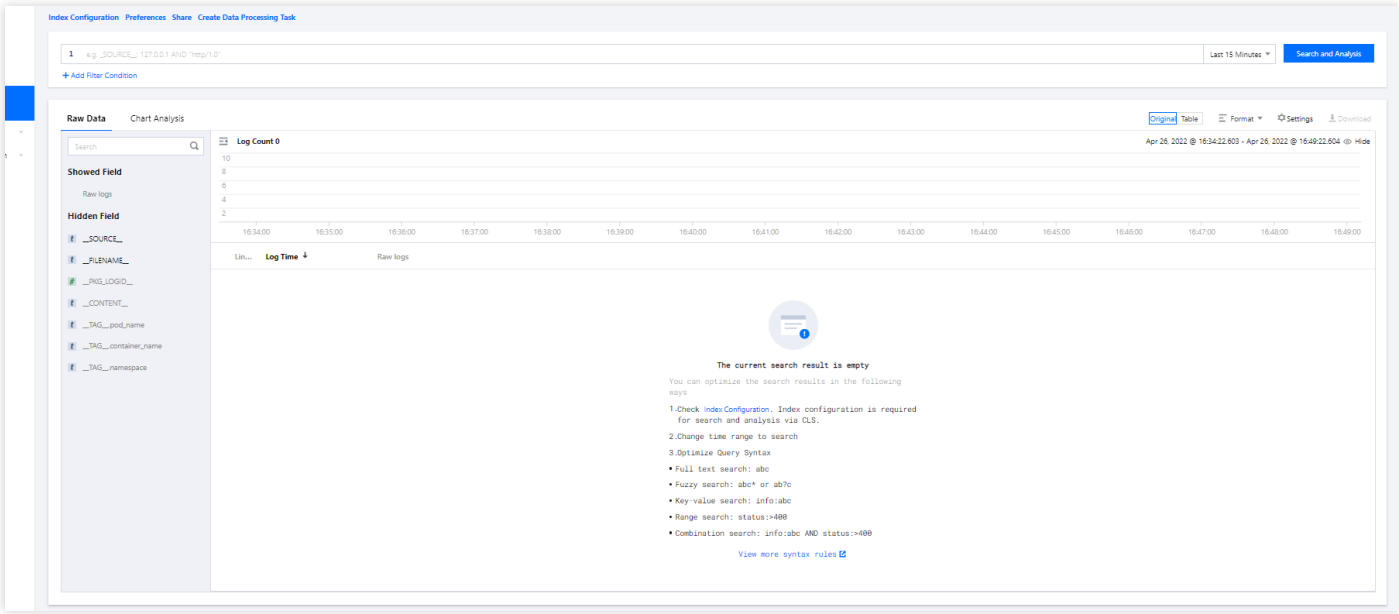


3. 单击**下一步**进入 “日志解析方式”， 其中本文示例 QCBM 使用单行文本方式。 了解更多 CLS 支持的日志格式，请参见 采集文本日志 文档。

查看日志

1. 登录 日志服务控制台，进入“检索分析”页面。

2. 检索分析中可先为日志**新建索引**，索引完毕之后再单击**检索分析**即可查看日志。



注意：
若未新建索引，则检索不到日志。

集成 TSW 观测服务

TSW 目前处于内测阶段，可在广州和上海进行部署，本文选择上海接入（QCBM 部署在南京）。

接入 TSW — 获取接入点信息

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏种的**服务观测 > 服务列表**。
2. 单击**接入服务**，选择 Java 语言与 SkyWalking 的数据采集方式。接入方式下提供了如下接入信息：**接入点**和 **Token**。

接入 TSW — 应用和容器配置

将上一步骤中获取的 TSW 的**接入点**和 **Token** 分别填写到 skywalking 的 agent.config 配置项中的 collector.backend_service 和 agent.authentication。“agent.service_name” 配置对应的服务名称，可使用

“agent.namespace” 对同一领域下的微服务归类。如下图为 user-service 配置：

```
# The agent namespace
agent.namespace=${SW_AGENT_NAMESPACE:QCBM}

# The service name in UI
agent.service_name=${SW_AGENT_NAME:user-service}

# The number of sampled traces per 3 seconds
# Negative or zero means off, by default
agent.sample_n_per_3_secs=${SW_AGENT_SAMPLE:-1}

# Authentication active is based on backend setting, see application.yml for more details.
agent.authentication = ${SW_AGENT_AUTHENTICATION:[redacted]}

# Backend service addresses.
collector.backend_service=${SW_AGENT_COLLECTOR_BACKEND_SERVICES:ap-shanghai.tencentservicewatcher.com:11800}
```

Skywalking agent 也支持使用环境变量方式进行配置，QCBM 使用 ConfigMap 和 Secret 配置对应的环境变量：

- 使用 ConfigMap 配置 SW_AGENT_COLLECTOR_BACKEND_SERVICES
- 使用 Secret 配置 SW_AGENT_AUTHENTICATION

如下图所示：

```
---
# 创建 ConfigMap
apiVersion: v1
kind: ConfigMap
metadata: meta.v1.ObjectMeta
data:
  NACOS_HOST: 10.0.1.9
  MYSQL_HOST: 10.0.1.13
  REDIS_HOST: 10.0.1.16
  SW_AGENT_COLLECTOR_BACKEND_SERVICES: ap-shanghai.tencentservicewatcher.com:11800

---
# 创建 Secret
apiVersion: v1
kind: Secret
metadata: meta.v1.ObjectMeta
data:
  MYSQL_ACCOUNT: [redacted]
  MYSQL_PASSWORD: [redacted]
  REDIS_PASSWORD: [redacted]
  SW_AGENT_AUTHENTICATION: [redacted]
type: Opaque
```

至此 TSW 接入工作已完成，启动容器服务后，在 TSW 控制台即可查看调用链、服务拓扑、SQL 分析等功能。

使用 TSW 观测服务

通过服务接口和调用链查看调用异常

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏中的**服务观测 > 接口观测**。
2. 在接口观测页面可查看一个服务下所有接口的调用情况，包括请求量、成功率、错误率、响应时间等指标。

使用 TSW 分析 SQL 和缓存等组件调用情况

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏中的**组件调用观测 > SQL 调用**。
2. 在“SQL 调用”页面可查看 SQL、NOSQL、MQ 及其它组件的调用情况。例如，通过 SQL 的请求量及耗时，可以快速定位应用中的高频 SQL 和慢查询。

查看服务拓扑

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏中的**链路追踪 > 分布式依赖拓扑**。
2. 在“分布式依赖拓扑”页面可查看完成的服务依赖情况，以及调用次数和平均延迟等信息。

SpringCloud 应用托管到 TKE

最近更新时间：2023-02-14 16:36:43

操作场景

本文章介绍了 SpringCloud 应用托管到腾讯云容器服务 TKE 的最佳实践。

SpringCloud 应用托管到 TKE 具有以下优势：

- 提升资源利用率。
- Kubernetes 天然适合微服务架构。
- 提升运维效率，便于 Devops 落地实施。
- Kubernetes 的高弹性，可轻松实现应用的动态扩缩容。
- 容器服务 TKE 提供 Kubernetes Master 托管功能，可减少 Kubernetes 集群运维和管理的负担。
- 容器服务 TKE 和腾讯云的其他云原生产品进行了整合和优化，帮助用户更好的使用腾讯云上产品。

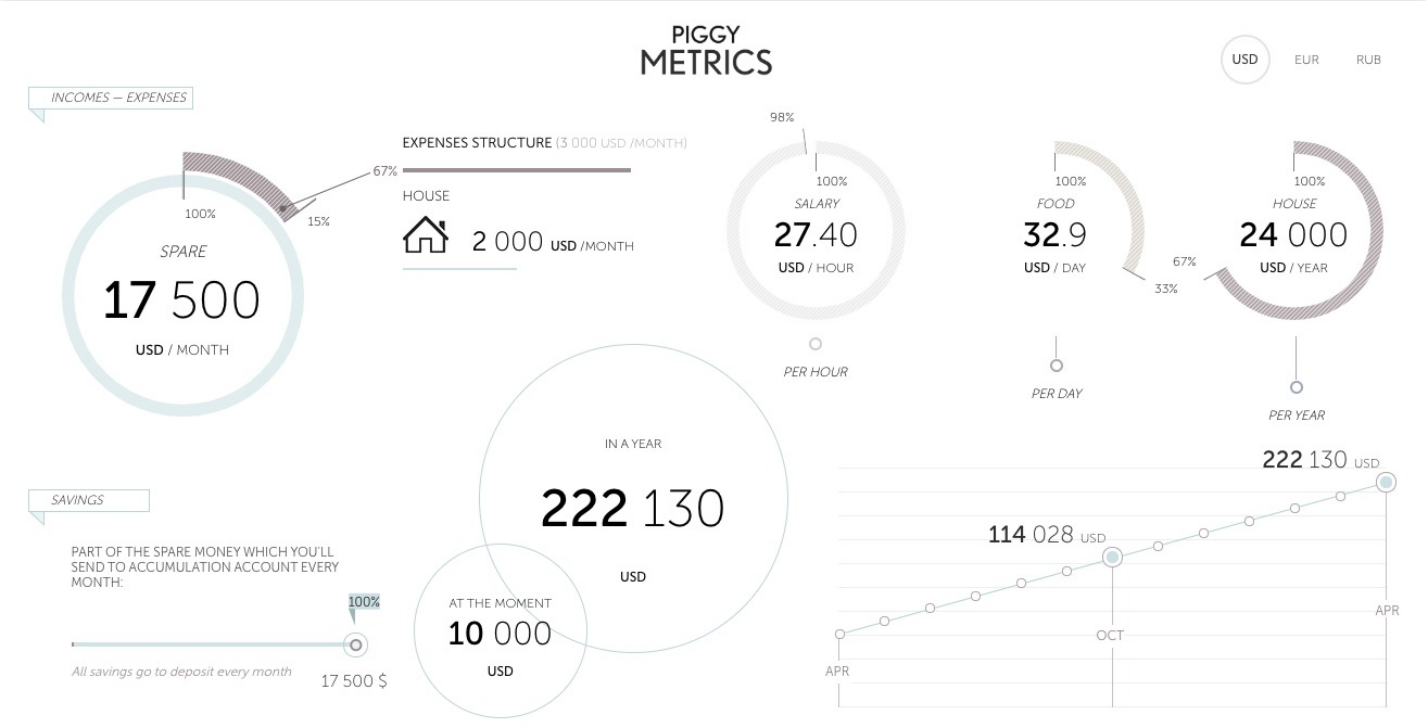
最佳实践实例介绍

PiggyMetrics 概述

本文最佳实践通过 fork GitHub 上的开源项目 PiggyMetrics，对其进行修改以适应腾讯云产品，并以最终修改后的版本为例，详细介绍 SpringCloud 应用托管到 TKE 的整个过程。

> 说明：
> 修改后的 PiggyMetrics 部署项目托管在 GitHub 上。在 搭建基础服务集群 后，可直接下载部署工程并在 TKE 上进行部署。

PiggyMetrics 首页如下图所示：



PiggyMetrics 是一个采用微服务架构，并使用 SpringCloud 框架开发的个人记账理财应用。

PiggyMetrics 微服务组成如下：

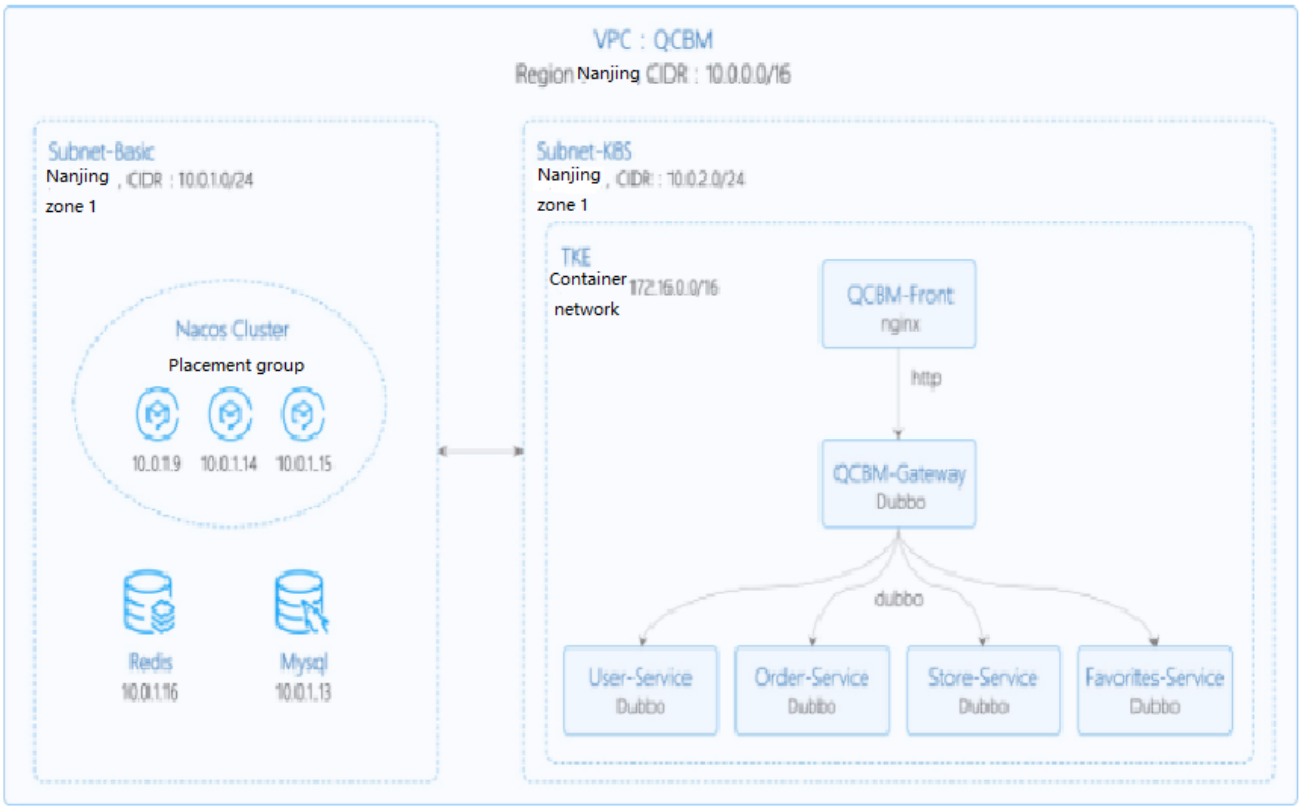
微服务	说明
API 网关	基于 Spring Cloud Zuul 的网关，是调用后台 API 的聚合入口，提供反向路由和负载均衡（Eureka+Ribbon）、限流熔断（Hystrix）等功能。CLIENT 单页应用和 ZUUL 网关暂住在一起，简化部署。
服务注册和发现	基于 Spring Cloud Eureka 的服务注册中心。业务服务启动时通过 Eureka 注册，服务之间调用也通过 Eureka 进行服务发现。
授权认证服务	基于 Spring Security OAuth2 的授权认证中心。客户端登录时通过 AUTHSERVICE 获取访问令牌。服务之间调用也通过 AUTHSERVICE 获取访问令牌（走客户端模式）。令牌校验方式，各资源服务器通过 AUTHSERVICE 集中校验令牌。
配置服务	基于 Spring Cloud Config 的配置中心，集中管理所有 Spring 服务的配置文件。
软负载和限流熔断	基于 Spring Cloud Ribbon&Hystrix，Zuul 调用后台服务，服务之间相互调用都通过 Ribbon 实现软负载，也通过 Hystrix 实现熔断限流保护。
METRICS & DASHBOARD	基于 Spring Cloud Turbine + Hystrix Dashboard，对所有 Hystrix 产生的 Metrics 流进行聚合，并展示在 Hystrix Dashboard 上。

PiggyMetrics 部署架构和组件

本文最佳实践实例模拟将原先部署在云服务器 CVM 的应用进行容器化，并托管到容器服务 TKE 的场景。在该场景中需要采用一个 VPC，并划分为以下两个子网：

- **Subnet-Basic** 中部署有状态的基础服务，包括 Dubbo 的服务注册中心 Nacos，MySQL 和 Redis 等。
- **Subnet-K8S** 中部署 PiggyMetrics 的应用服务，所有服务都进行了容器化，运行在容器服务 TKE 上。

子网划分如下图所示：



PiggyMetrics 实例的网络规划如下表所示：

网络规划	说明
Region / AZ	南京 / 南京一区
VPC	CIDR：10.0.0.0/16

网络规划	说明
子网 Subnet-Basic	南京一区，CIDR：10.0.1.0/24
子网 Subnet-K8S	南京一区，CIDR：10.0.2.0/24
Nacos 集群	采用 3 台 “标准型SA2” 1C2G 机型的 CVM 构建 Nacos 集群，对应的 IP 为：10.0.1.9，10.0.1.14，10.0.1.15

PiggyMetrics 实例中用到的组件如下表所示：

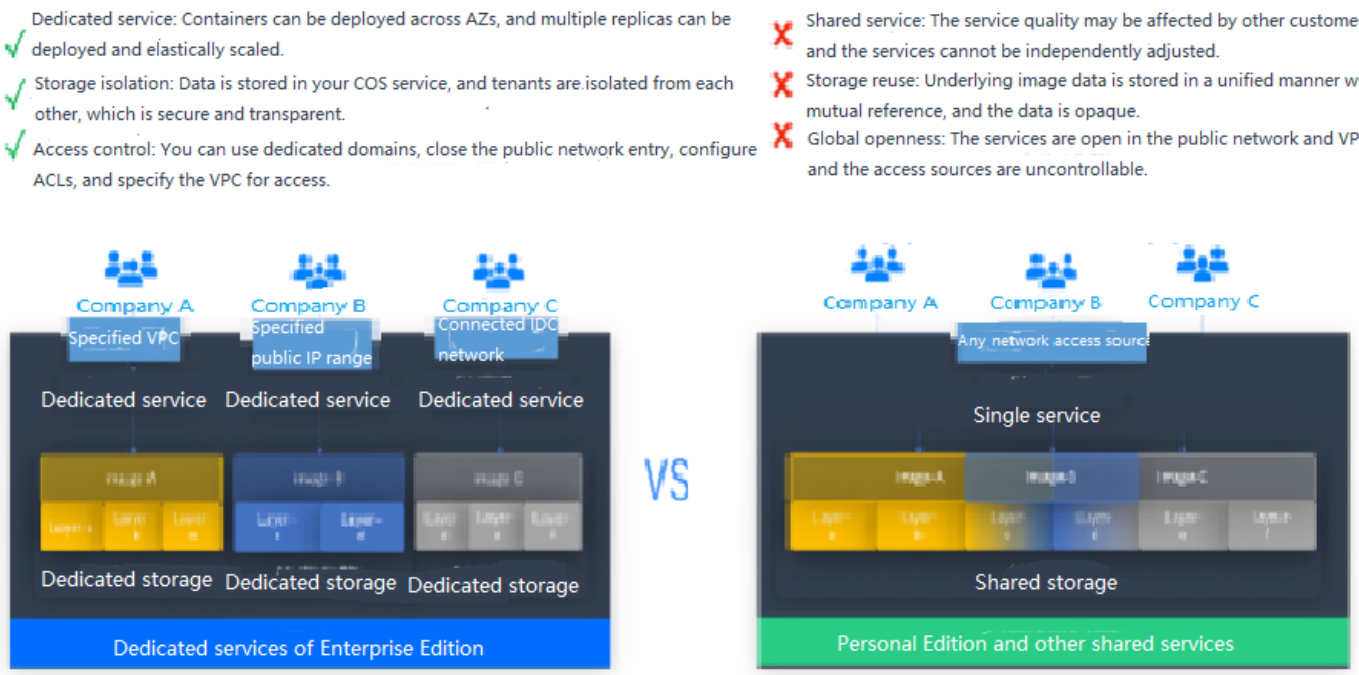
组件	版本	来源	备注
K8S	1.8.4	腾讯云	TKE 托管模式
MongoDB	4.0	腾讯云	TencentDB for MongoDB WiredTiger 引擎版
CLS	N/A	腾讯云	日志服务
TSW	N/A	腾讯云	采用 Skywalking 8.4.0 版的 agent 接入，点此 下载
Java	1.8	开源社区	Docker 镜像：java:8-jre
SrpingCloud	Finchley.RELEASE	开源社区	Spring Cloud 官网

服务介绍

TCR 介绍

腾讯云 容器镜像服务 TCR（Tencent Container Registry，TCR），提供了个人版和企业版两种镜像仓库。两者区别如下：

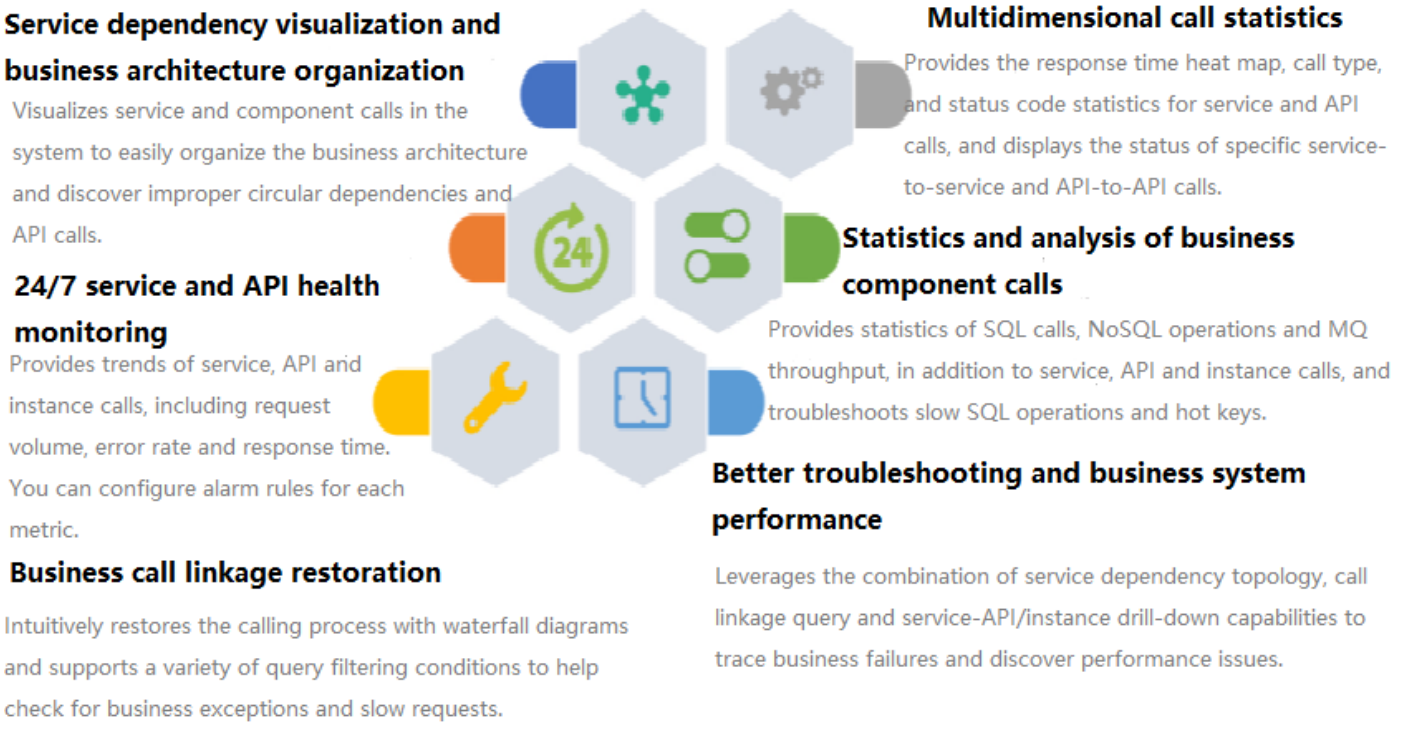
- 个人版镜像仓库仅部署在腾讯云广州，企业版在每个地域都有部署。
- 个人版未提供服务 SLA 保证。



PiggyMetrics 是一个 Dubbo 容器化的 Demo 项目，因此容器镜像服务个人版完全满足需求。但对于企业用户，推荐使用 企业版 TCR。如需使用镜像仓库，请见 镜像仓库基本操作。

TSW 介绍

腾讯云微服务观测平台 TSW（Tencent Service Watcher，TSW）提供云原生服务可观察性解决方案，能够追踪到分布式架构中的上下游依赖关系，绘制拓扑图，提供服务、接口、实例、中间件等多维度调用观测。



TSW 在架构上分为以下四大模块：

展开全部

数据采集（Client）

展开&收起

使用开源探针或 SDK 用于采集数据。对于迁移上云的用户，可保留 Client 端的大部分配置，仅更改上报地址和鉴权信息即可。

数据处理（Server）

展开&收起

数据经由 Pulsar 消息队列上报到 Server，同时 Adapter 会将数据转换为统一的 Opentracing 兼容格式。根据数据的使用场景，分配给实时计算与离线计算：

- 实时计算提供实时监控、统计数据展示，并对接告警平台快速响应。
- 离线计算处理长时段大量数据的统计汇聚，利用大数据分析能力提供业务价值。

存储（Storage）

展开&收起

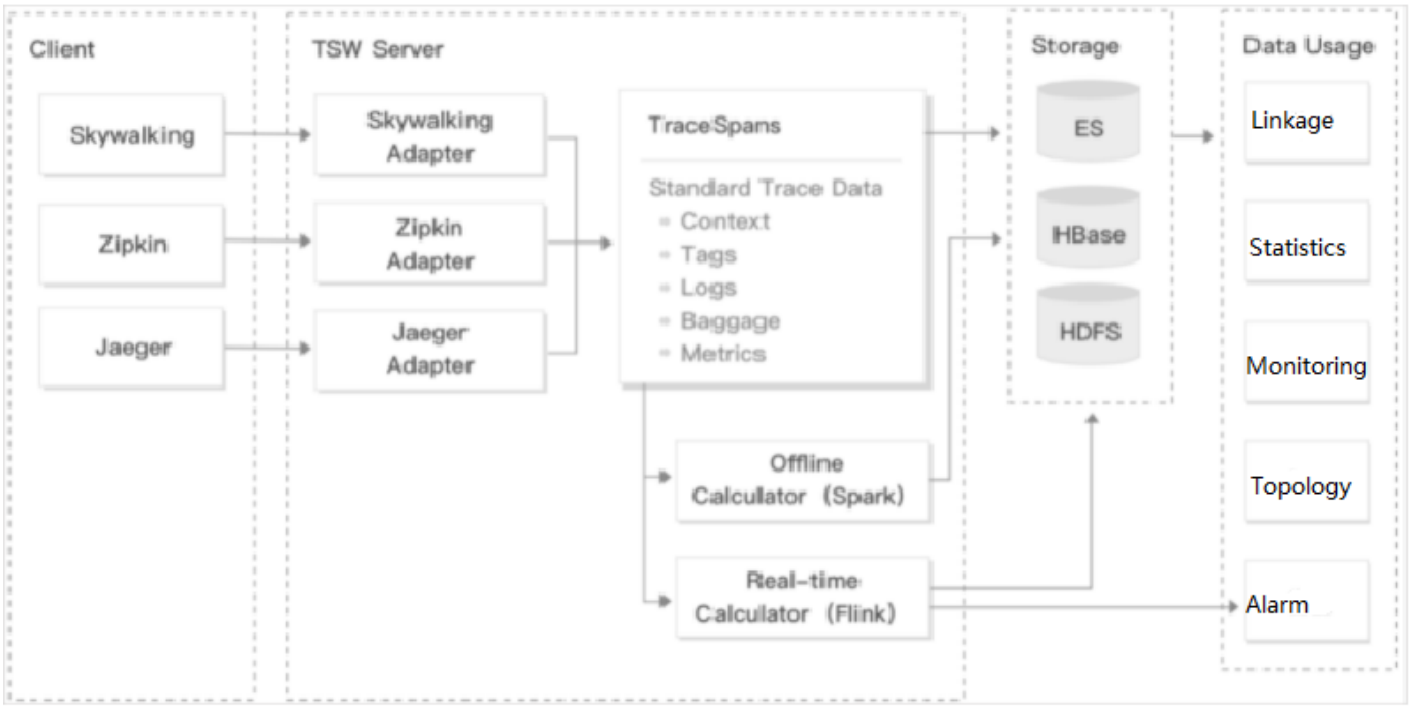
存储层可满足不同数据类型的使用场景，适配 Server 层的写入与 Data Usage 层的查询与读取请求。

数据使用（Data Usage）

展开&收起

为控制台操作、数据展示、告警提供底层支持。

架构图如下所示：



操作步骤

基础服务集群搭建

- 在 Mongodb 控制台 创建实例，并执行以下命令进行初始化：

```
# 下载 mongo client，解压，进入 bin 目录

wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-3.6.18.tgz
tar -zxvf mongodb-linux-x86_64-3.6.18.tgz
cd mongodb-linux-x86_64-3.6.18/bin

# 使用下面命令初始化 mongodb，其中 mongouser 为创建 mongodb 实例时创建的管理员账号

./mongo -u mongouser -p --authenticationDatabase "admin" [mongodb的IP]/piggymetrics mongo-init.js
```

> mongodb 初始化脚本 **mongo-init.js** 中默认创建了一个 piggymetrics 库的用户 **guest**，可按您的需求进行修改。

- 在 CLB 控制台 为子网 Subnet-K8S 新建一个内网型的 CLB（后续实践中会使用到该 CLB 实例 ID）。
- TSW 目前处于内测阶段，支持 Java 和 Golang 两种语言接入。

构建 Docker 镜像

编写 Dockerfile

下文以 account-service 为例为您简单介绍如何编写 Dockerfile。示例展示的是 account-service 的工程目录结构，**Dockerfile** 位于工程的根目录下，**account-service.jar** 是打包后的文件，需要添加到镜像中。

```
➜ account-service tree
├── Dockerfile
├── skywalking
│ ├── account.config
│ └── skywalking-agent.zip
├── pom.xml
├── src
│ ....
├── target
│ .....
│ └── account-service.jar
└── account-service.iml
```

> 说明：

此处使用 skywalking-agent 作为 TSW 接入客户端，向 TSW 后台上报调用链信息。下载 Skywalking-agent 详情可参见 PiggyMetrics 部署架构和组件。

account-service 的 Dockerfile 如下所示：

```
FROM java:8-jre

# 容器中的工作目录为

/appWORKDIR /app

# 将本地打包出来的应用添加到镜像中

ADD ./target/account-service.jar

# 将 skywalking agent 拷贝到镜像中

COPY ./skywalking/skywalking-agent.zip

# 解压 skywalking agent 并删除原始压缩文件

RUN unzip skywalking-agent.zip && rm -f skywalking-agent.zip

# 添加 skywalking 的配置文件

COPY ./skywalking/account.config ./skywalking-agent/config/agent.config

# 启动应用

CMD ["java", "-Xmx256m", "-javaagent:/app/skywalking-agent/skywalking-agent.jar"
, "-jar", "/app/account-service.jar"]

# 应用的端口说明

EXPOSE 6000
```

注意：
Dockerfile 中每多一个 RUN 命令，生成的镜像就多一层，推荐将这些 RUN 命令合成一条。

镜像构建

容器镜像服务 TCR 提供自动和手工两种构建镜像方式。为展示具体的构建过程，本文采用手工构建方式。

镜像名称需要符合规范 `ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]` ：

- 其中命名 namespace 为方便镜像管理使用，可以按项目取名。本文采用 piggymetrics 表示 PiggyMetrics 项目下的所有镜像。
- ImageName 可以包含 subpath，一般用于企业用户多项目场景。此外，如果本地已构建好镜像，可使用 `docker tag` 命令，按命名规范对镜像重命名。

1. 执行以下命令构建镜像。示例如下：

```
# 推荐的构建方式，可省去二次打 tag 操作

sudo docker build -t ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]

# 本地构建 account-service 镜像，最后一个 . 表示 Dockerfile 存放在当前目录（user-service）下

➜ account-service docker build -t ccr.ccs.tencentyun.com/piggymetrics/account-service:1.0.0 .

# 将已存在镜像按命名规范对镜像重命名

sudo docker tag [ImageId] ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

2. 构建完成后，可执行以下命令查看本地仓库中的所有镜像。

```
docker images | grep piggymetrics
```

示例如下图所示：

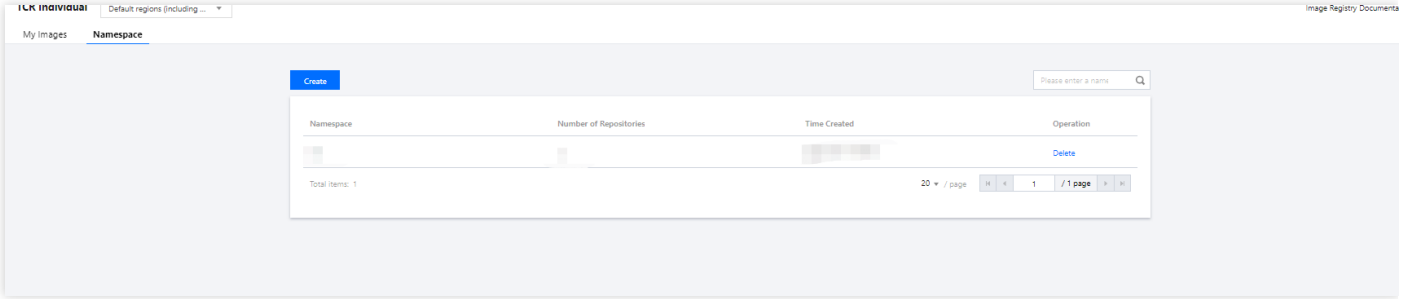
```
➜  account-service docker images | grep "piggymetrics"
ccr.ccs.tencentyun.com/piggymetrics/account-service          1.0.2               134cf538f5f7        10 minutes ago      405MB
ccr.ccs.tencentyun.com/piggymetrics/turbine-stream-service   1.0.1               9faaae7517d4        24 hours ago        356MB
ccr.ccs.tencentyun.com/piggymetrics/gateway                  1.0.3               0351544fb1c9        3 days ago          393MB
ccr.ccs.tencentyun.com/piggymetrics/config-server            1.0.5               cbb1216e4d04        3 days ago          340MB
ccr.ccs.tencentyun.com/piggymetrics/notification-service     1.0.1               5f34870d1d7c        3 days ago          404MB
ccr.ccs.tencentyun.com/piggymetrics/statistics-service       1.0.1               034f5239967a        4 days ago          404MB
ccr.ccs.tencentyun.com/piggymetrics/auth-service             1.0.1               b3aadfa22c0d        4 days ago          398MB
ccr.ccs.tencentyun.com/piggymetrics/monitoring               1.0.0               2ed5e7c9e133        4 days ago          343MB
ccr.ccs.tencentyun.com/piggymetrics/registry                 1.0.0               e946d0ed8c34        4 days ago          356MB
```

上传镜像到 TCR

创建命名空间

PiggyMetrics 项目采用个人版镜像仓库（建议企业客户使用企业版镜像仓库）。

1. 登录 容器服务控制台 。
2. 选择**镜像仓库 > 个人版 > 命名空间**进入“命名空间”页面。
3. 单击**新建**，在弹出的新建命名窗口中新建命名空间 piggymetrics。PiggyMetrics 项目所有的镜像都存放于该命名空间下。如下图所示：



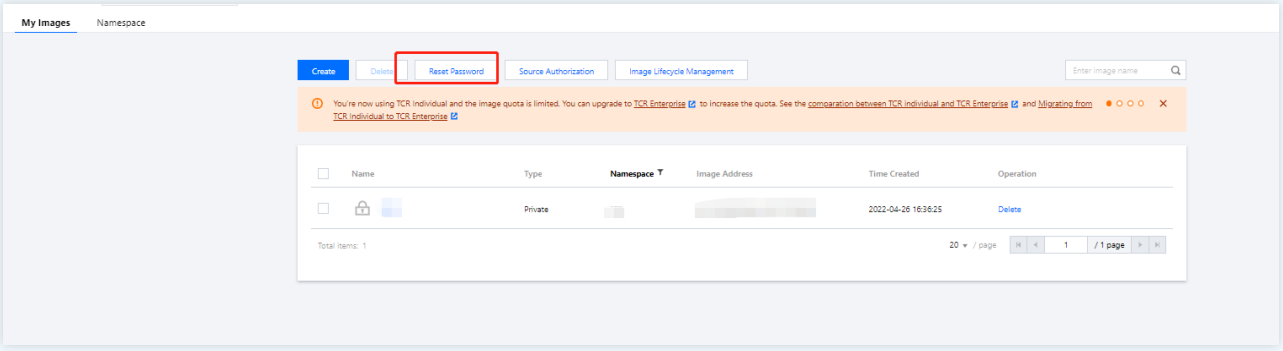
上传镜像

上传镜像需要完成以下步骤：登录腾讯云 registry 和上传镜像。

1. 执行以下命令登录腾讯云 registry。

```
docker login --username=[腾讯云账号 ID] ccr.ccs.tencentyun.com
```

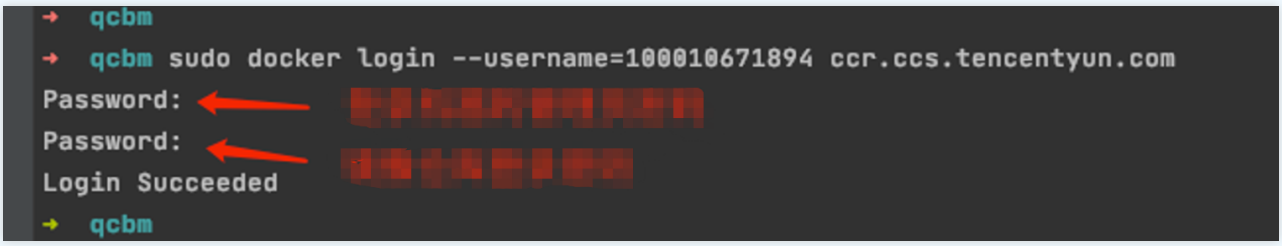
- 腾讯云账号 ID 可在 账号信息 页面获取。
- 若忘记**镜像仓库登录密码**，可前往容器服务镜像仓库个人版 我的镜像 中进行重置。



- 若执行命令提示无权限，请在上述命令前加上 sudo 再执行，如下所示。此时需要输入两个密码，第一个为 sudo 所需的主机管理员密码，第二个为**镜像仓库登录密码**。

```
sudo docker login --username=[腾讯云账号 ID] ccr.ccs.tencentyun.com
```

如下图所示：



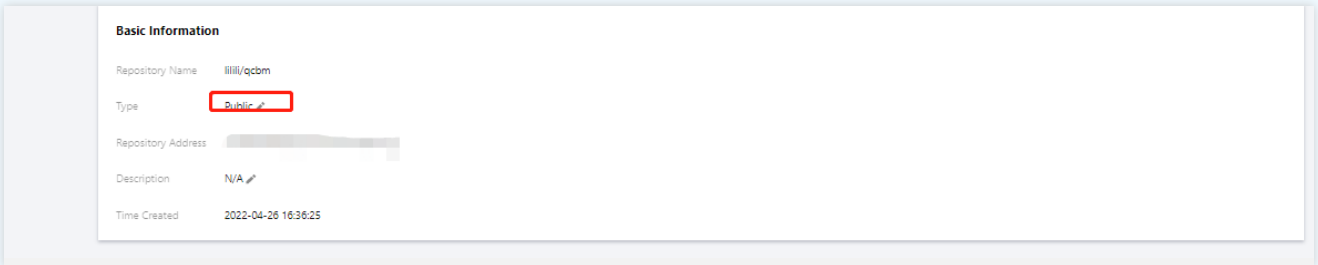
2. 执行以下命令将本地生成的镜像推送至 TKE 的镜像仓库中。

```
docker push ccr.ccs.tencentyun.com/[namespace]/[ImageName]:[镜像版本号]
```

如下图所示：

3. 在 我的镜像 中可以查看上传的所有镜像。

默认镜像类型为“私有”，如需提供镜像给他人使用，可在**镜像信息**中将镜像类型设置为公有。如下图所示：



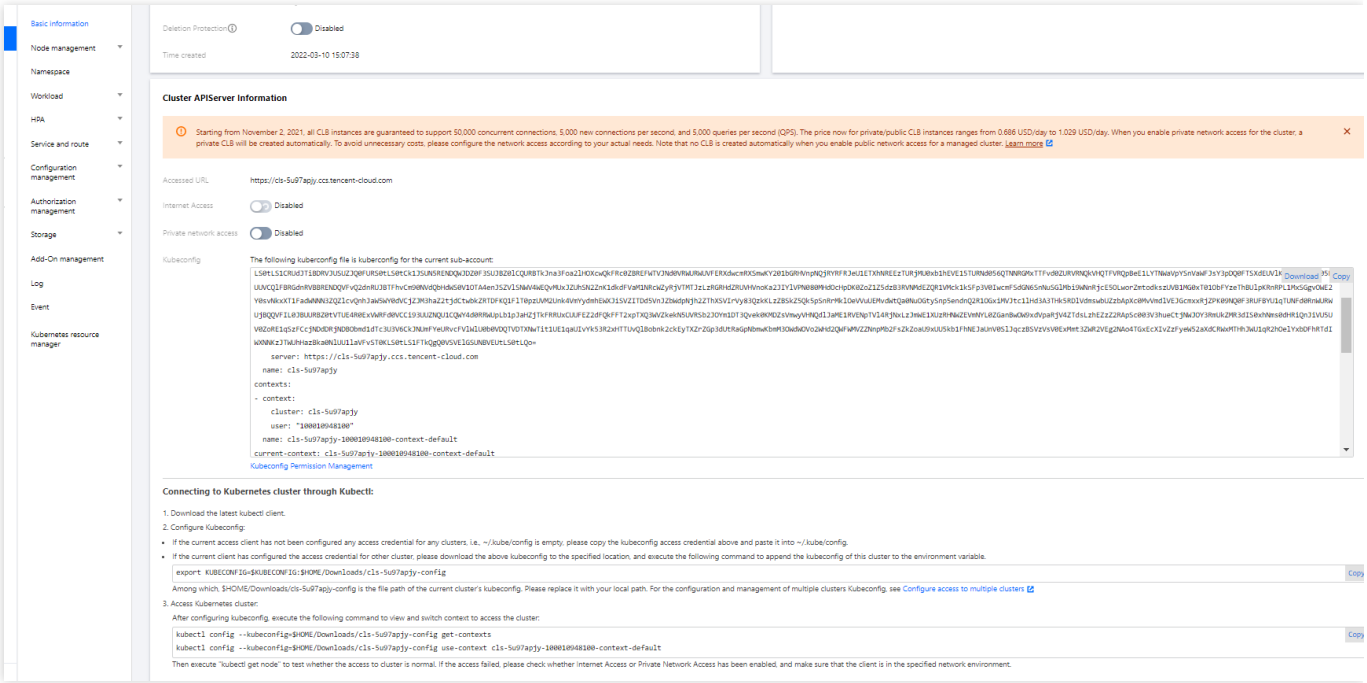
在 TKE 上部署服务

创建 K8S 集群 PiggyMetrics

1. 实际部署前，需要新建一个 K8S 集群。有关集群的创建，请参见 创建集群 文档。

> 注意：
> 在创建集群时，在“选择机型” 页面建议开启“置放群组功能”，该功能可将 CVM 打散到不同母机上，增加系统可靠性。

2. 创建完成后，在容器服务控制台 的 集群管理 页面可以看到新建的集群信息。本文新建的集群名称为 piggyMetrics。
3. 单击集群 PiggyMetrics-k8s-demo 进入“基本信息”页面，可以查看整个集群的配置信息。
4. （可选）如需使用 kubectl 和 lens 等 K8S 管理工具，还需进行以下两步操作：
   i. 开启外网访问。
   ii. 将 API 认证 Token 保存为本地 用户 `home/.kube` 下的 config 文件中（若 config 文件已有内容，需要替换），以确保每次访问都能进入默认集群中。如果选择不保存为 `.kube` 下的 config 文件中，则可参考控制台**集群APIServer信息**下的 **通过Kubectl连接Kubernetes集群操作说明**。如下图所示：



创建 Namespace

Namespaces 是 Kubernetes 在同一个集群中进行逻辑环境划分的对象， 通过 Namespaces 可以进行多个团队多个项目的划分。您可以通过以下三种方式创建 Namespace，推荐使用方式1命令行方式创建。

- 方式1：使用命令行
- 方式2：使用控制台
- 方式3：使用 YAML 部署

执行以下命令即可创建 Namespace：

```
kubectl create namespace piggymetrics
```

使用 ConfigMap 存放配置信息

通过 ConfigMap 可以将配置和运行的镜像进行解耦，使应用程序有更强的移植性。PiggyMetrics 后端服务需要从环境变量中获取 MongoDB 的主机和端口信息，并使用 ConfigMap 来保存。
您可通过以下两种方式使用 ConfigMap 存放配置信息：

- 方式1：使用 YAML
- 方式2：使用控制台

下文为 PiggyMetrics 的 ConfigMap YAML，其中**纯数字类型的 value 需要使用双引号**。

```
# 创建 ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
name: piggymetrics-env
namespace: piggymetrics
data:
# MongDB 的 IP 地址
MONGODB_HOST: 10.0.1.13
# TSW 接入地址，后文介绍
SW_AGENT_COLLECTOR_BACKEND_SERVICES: ap-shanghai.tencentservicewatcher.com:11800
```

使用 Secret 存放敏感信息

Secret 可用于存储密码、令牌、密钥等敏感信息，降低直接对外暴露的风险。PiggyMetrics 使用 Secret 来保存相关的账号和密码信息。
您可通过以下两种方式使用 Secret 存放敏感信息：

- 方式1：使用 YAML
- 方式2：使用控制台

下文为 PiggyMetrics 创建 Secret 的 YAML。其中 Secret 的 value 需要是 base64 编码后的字符串。

```
# 创建 Secret
apiVersion: v1
kind: Secret
metadata:
name: piggymetrics-keys
namespace: piggymetrics
labels:
qcloud-app: piggymetrics-keys
data:
# 请将下面的 xxx 替换为实际值
MONGODB_USER: XXX
MONGODB_PASSWORD: XXX
SW_AGENT_AUTHENTICATION: XXX
type: Opaque
```

使用 StatefulSet 部署有状态服务

StatefulSet 主要用于管理有状态的应用，创建的 Pod 拥有根据规范创建的持久型标识符。Pod 迁移或销毁重启后，标识符仍会保留。在需要持久化存储时，您可以通过标识符对存储卷进行一一对应。
PiggyMetrics 项目下的配置服务、注册中心、rabbitmq 等基础组件和服务，本身保存有数据，因此适合使用 StatefulSet 进行部署。

以下是 config-server 对应的部署 YAML 示例：

```
---
kind: Service
apiVersion: v1
metadata:
name: config-server
namespace: piggymetrics
spec:
clusterIP: None
ports:
- name: http
port: 8888
targetPort: 8888
protocol: TCP
selector:
app: config
version: v1

---
apiVersion: apps/v1
```

```
kind: StatefulSet
metadata:
name: config
namespace: piggymetrics
labels:
app: config
version: v1
spec:
serviceName: "config-server"
replicas: 1
selector:
matchLabels:
app: config
version: v1
template:
metadata:
labels:
app: config
version: v1
spec:
terminationGracePeriodSeconds: 10
containers:
- name: config
image: ccr.ccs.tencentyun.com/piggymetrics/config-server:2.0.03
ports:
- containerPort: 8888
protocol: TCP
```

部署工作负载 Deployment

Deployment 声明了 Pod 的模板和控制 Pod 的运行策略，适用于部署无状态的应用程序。PiggyMetrics 的 account 等后台服务都属于无状态应用，适合使用 Deployment。

以下是 account-service Deployment 的 YAML 参数说明：

参数	说明
replicas	表示需要创建的 Pod 数量
image	镜像的地址
imagePullSecrets	拉取镜像时需要使用的 key，可在 集群>**配置管理** > **Secret**中获取。使用公共镜像时可省略
env	• 定义了 pod 的环境变量和取值 • ConfigMap 中定义的 key-value 可使用 configMapKeyRef 引用 • Secret 中定义的 key-value 可使用 secretKeyRef 引用

参数	说明
ports	指定容器的端口号，account-service 的端口号为6000

account-service Deployment 的 完整 YAML 文件示例如下：

```
# account-service Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
name: account-service
namespace: piggymetrics
labels:
app: account-service
version: v1
spec:
replicas: 1
selector:
matchLabels:
app: account-service
version: v1
template:
metadata:
labels:
app: account-service
version: v1
spec:
containers:
- name: account-service
image: ccr.ccs.tencentyun.com/piggymetrics/account-service:1.0.1
env:
# mongodb 的IP地址
- name: MONGODB_HOST
valueFrom:
configMapKeyRef:
key: MONGODB_HOST
name: piggymetrics-env
optional: false
# mongodb 用户名
- name: MONGODB_USER
valueFrom:
secretKeyRef:
key: MONGODB_USER
name: piggymetrics-keys
optional: false
```

```
# mongodb 密码
- name: MONGODB_PASSWORD
valueFrom:
secretKeyRef:
key: MONGODB_PASSWORD
name: piggymetrics-keys
optional: false
# TSW 接入点
- name: SW_AGENT_COLLECTOR_BACKEND_SERVICES
valueFrom:
configMapKeyRef:
key: SW_AGENT_COLLECTOR_BACKEND_SERVICES
name: piggymetrics-env
optional: false
# TSW 接入 token
- name: SW_AGENT_AUTHENTICATION
valueFrom:
secretKeyRef:
key: SW_AGENT_AUTHENTICATION
name: piggymetrics-keys
optional: false
ports:
# 容器端口
- containerPort: 6000
protocol: TCP
imagePullSecrets: # 拉取镜像的 token
- name: qcloudregistrykey
```

部署服务 Service

Kubernetes 的 ServiceTypes 允许指定 Service 类型，默认为 ClusterIP 类型。ServiceTypes 可取如下值：

- LoadBalancer：提供公网、VPC、内网访问。
- NodePort：可通过“云服务器 IP + 主机端口”访问服务。
- ClusterIP：可通过“服务名 + 服务端口”访问服务。

PiggyMetrics 的前端页面和 gateway 打包在一块，需要对外提供服务，因此指定 LoadBalancer 类型的 ServiceType。TKE 对 LoadBalancer 模式进行了扩展，通过 Annotation 注解配置 Service，可实现更丰富的负载均衡能力。

若使用 `service.kubernetes.io/qcloud-loadbalancer-internal-subnetid` 注解，在 service 部署时，会创建内网类型 CLB。一般建议事先创建好 CLB，service 的部署 YAML 中使用注解 `service.kubernetes.io/loadbalance-id` 直接指定，可提升部署效率。

以下是 gateway service 部署 YAML：

```
# 部署 gateway service
apiVersion: v1
kind: Service
metadata:
name: gateway
namespace: piggymetrics
annotations:
# 请替换成 Subnet-K8S 子网的 CLB 实例 ID
service.kubernetes.io/loadbalance-id: lb-hfyt76co
spec:
externalTrafficPolicy: Cluster
ports:
- name: http
port: 80
targetPort: 4000
protocol: TCP
selector: # 将后端服务 gateway 和该 Service 进行映射
app: gateway
version: v1
type: LoadBalancer
```

查看部署结果

至此，您已完成 PiggyMetrics 在容器服务 TKE 上的部署，可通过以下步骤查看部署结果：

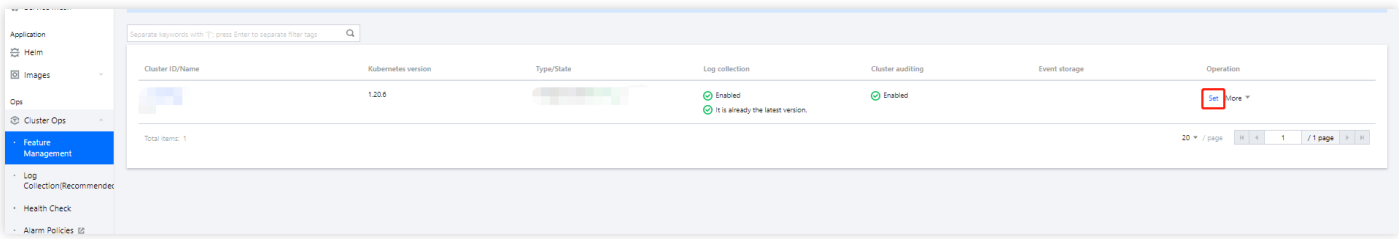
1. 登录 容器服务控制台 ，单击集群 ID/名称进入集群详情页面。
2. 单击**服务与路由** > **Service**进入 “Service 页面”，可查看到创建的 Service。通过 gateway service 的 VIP 即可访问 PiggyMetrics 页面。

集成 CLS 日志服务

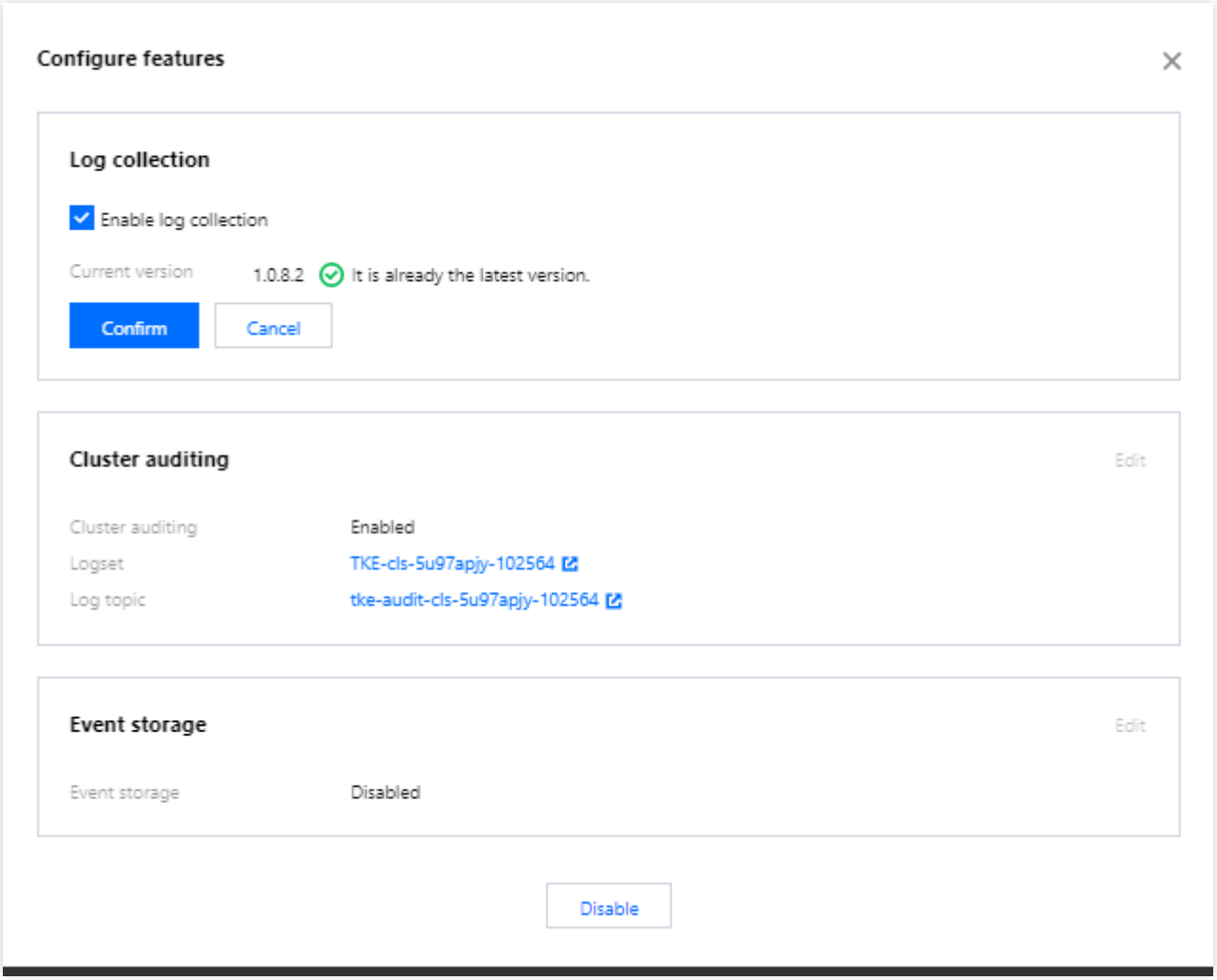
开启容器日志采集功能

容器日志采集功能默认关闭，使用前需要开启，步骤如下：

1. 登录容器服务控制台 ，选择左侧导航栏中的**集群运维** > **功能管理**。
2. 在“功能管理”页面上方选择地域，单击需要开启日志采集的集群右侧的**设置**。



3. 在“设置功能”页面，单击日志采集**编辑**，开启日志采集后确认。如下图所示：



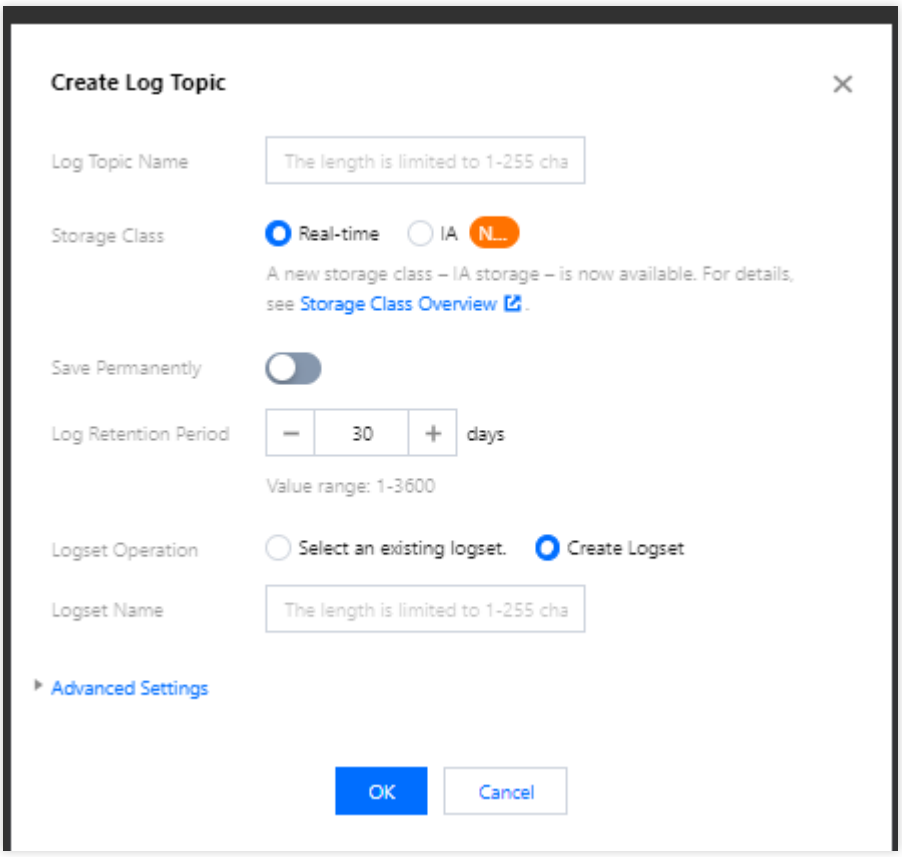
创建日志主题和日志集

日志服务区分地域，为了降低网络延迟，尽可能选择与服务邻近的服务地域创建日志资源。日志资源管理主要分为日志集和日志主题，一个日志集表示一个项目，一个日志主题表示一类服务，单个日志集可以包含多个日志主题。

PiggyMetrics 部署在南京，在“日志主题”页面选择南京地域。因此在创建日志集时应当选择南京地域：

1. 登录 日志服务控制台，在“日志主题”页面选择南京地域。

2. 单击**创建日志主题**，在弹出的窗口中根据页面提示填写相关信息，如下图所示：



- **日志主题名称**：输入 piggymetrics。
- **日志集操作**：选择**创建日志集**。
- **日志集名称**：输入 piggymetrics-logs。

3. 单击**确定**即可创建日志主题和日志集。

> 说明：
> PiggyMetrics 有多个后端微服务，为每个微服务建个日志主题便于日志归类。
>
> - PiggyMetrics 每个服务都建立了一个日志主题。
> - 日志主题 ID，为容器创建日志规则时需要用到。

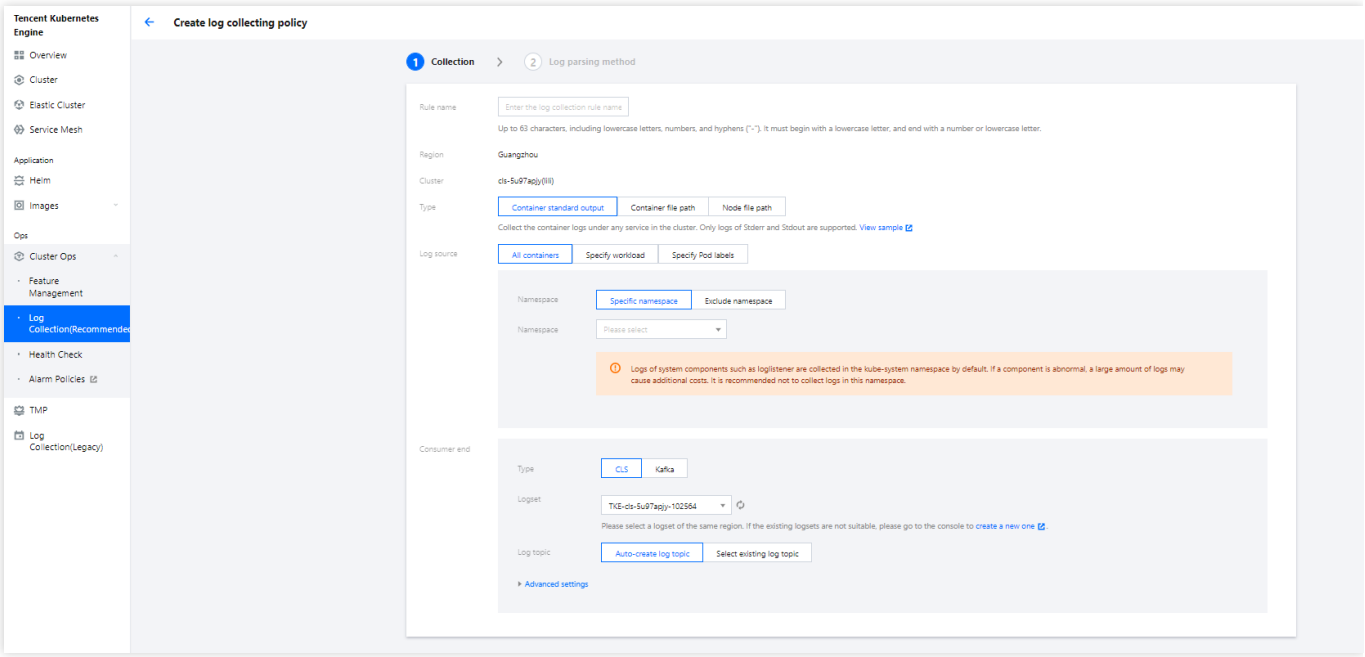
配置日志采集规则

您可通过控制台或 CRD 两种方式配置容器日志采集规则。

- 方式1：使用控制台
- 方式2：使用 CRD

日志规则指定了日志在容器内的位置：

1. 登录 容器服务控制台 ，选择左侧导航栏中的**集群运维** > **日志规则**。
2. 在“日志规则”页面，单击**新建**新建日志规则：
   - **日志源**：指定容器日志位置，PiggyMetrics 采用 SpringCloud 的默认配置，所有日志都打印到标准输出中，因而使用容器标准输出，并指定具体的 Pod Label。
   - **消费端**：选择之前创建的日志集和主题。

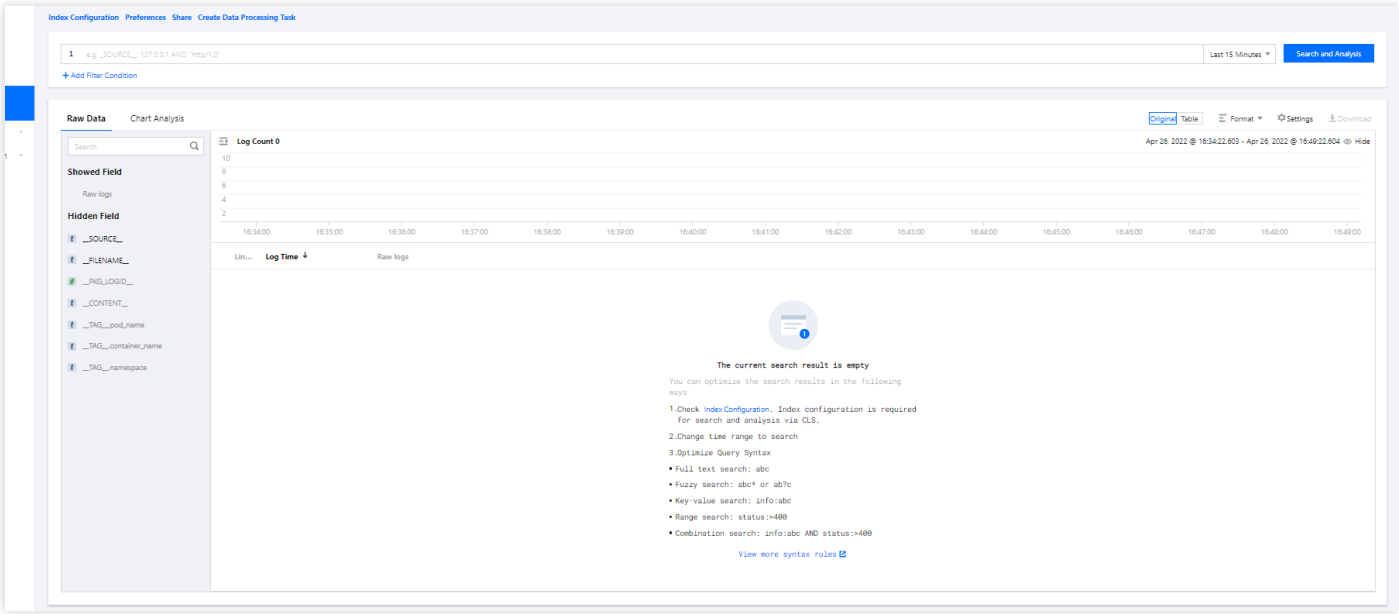


3. 单击**下一步**，进入“日志解析方式”，其中本文示例 PiggyMetrics 使用单行文本方式。了解更多 CLS 支持的日志格式，请参见 采集文本日志 文档。

查看日志

1. 登录 日志服务控制台，进入“检索分析”页面。

2. 检索分析中可先为日志**新建索引**，索引完毕之后再单击**检索分析**即可查看日志。



> 注意：
> 若未新建索引，则检索不到日志。

集成 TSW 观测服务

微服务观测平台 TSW 目前处于内测阶段，可在广州和上海进行了部署，本文选择上海接入（PiggyMetrics 部署在南京）。

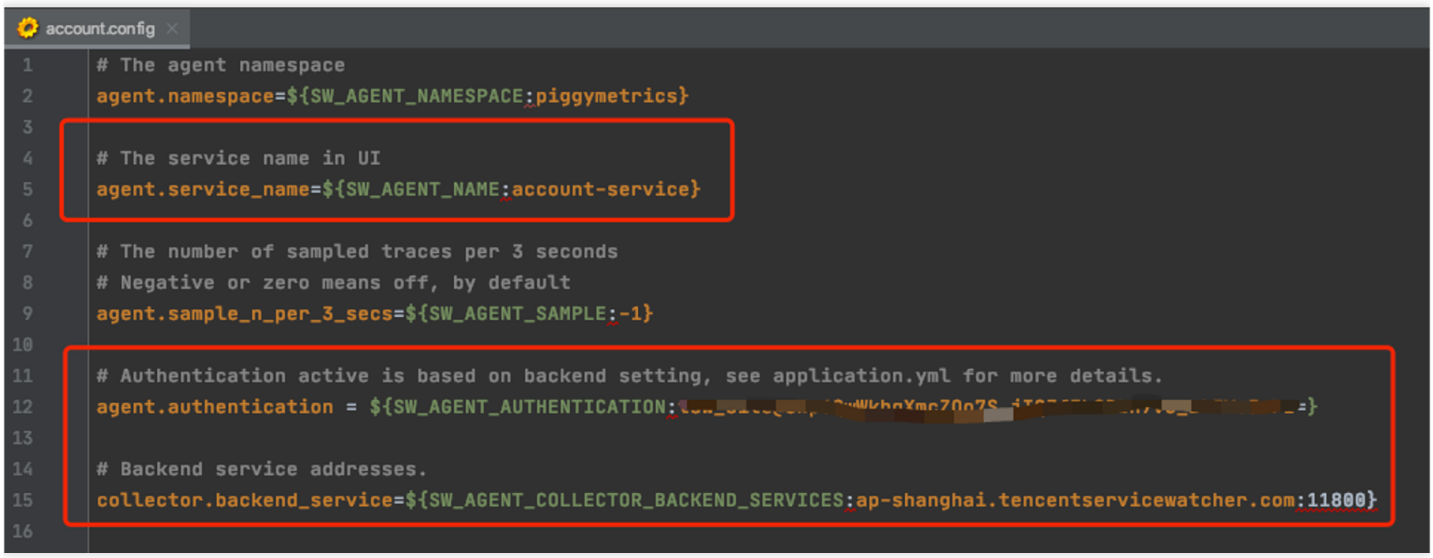
接入 TSW — 获取接入点信息

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏中的**服务观测 > 服务列表**。
2. 单击**接入服务**，选择 Java 语言与 SkyWalking 的数据采集方式。接入方式下提供了如下接入信息：**接入点**和 **Token**。

接入 TSW — 应用和容器配置

将上一步骤中获取的 TSW 的**接入点**和 **Token** 分别填写到 skywalking 的 agent.config 里的配置项 collector.backend_service 和 agent.authentication。“agent.service_name” 配置对应的服务名称，可使用

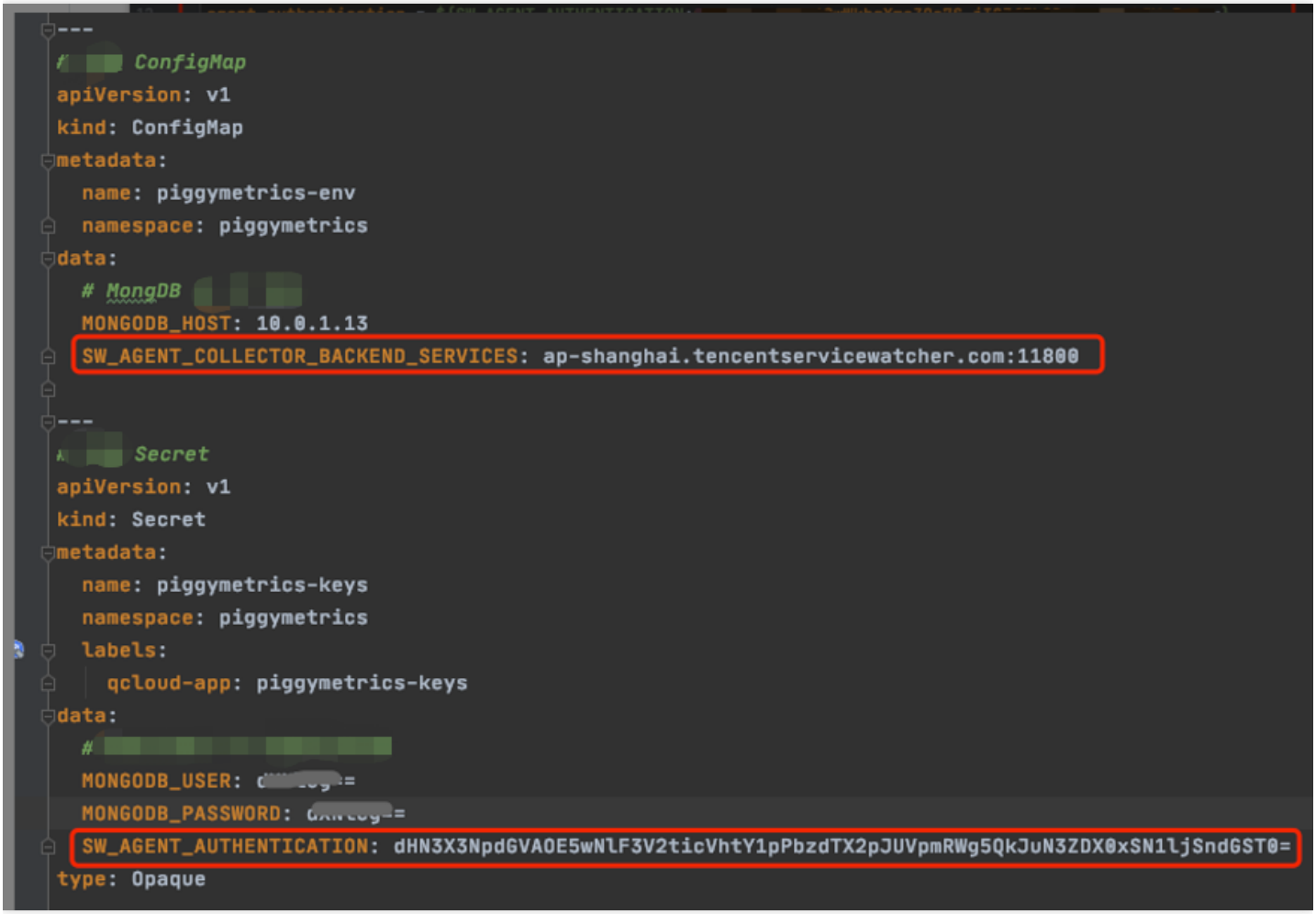
“agent.namespace” 对同一领域下的微服务归类。如下图为 user-service 配置：



Skywalking agent 也支持使用环境变量方式进行配置，PiggyMetrics 使用 ConfigMap 和 Secret 配置对应的环境变量：

- 使用 ConfigMap 配置 `SW_AGENT_COLLECTOR_BACKEND_SERVICES`
- 使用 Secret 配置 `SW_AGENT_AUTHENTICATION`

如下图所示：



至此 TSW 接入工作已完成，启动容器服务后，在 腾讯微服务观测平台控制台 即可查看调用链、服务拓扑、SQL 分析等功能。

使用 TSW 观测服务

通过服务接口和调用链查看调用异常

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏中的**服务观测 > 接口观测**。
2. 在接口观测页面可查看一个服务下所有接口的调用情况，包括请求量、成功率、错误率、响应时间等指标。
   图中展示的是最近1小时内 gateway 和 account-service 响应时间过大，statistic-service 所有请求全部失败。
3. 单击服务名称 statistics-service 进入该服务的信息页，单击**接口观测**可以查看到接口 `{PUT}/{accountName}` 抛出了 NestedServletException 异常，从而导致该接口不可用。
4. 单击 Trace ID 后可以查看完整的调用链详情。

查看服务拓扑

1. 登录 腾讯微服务观测平台控制台，选择左侧导航栏中的**链路追踪 > 分布式依赖拓扑**。

2. 在“分布式依赖拓扑”页面可查看完成的服务依赖情况，以及调用次数和平均延迟等信息。

成本管理
资源利用率提升工具大全

最近更新时间：2023-02-09 10:31:05

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：

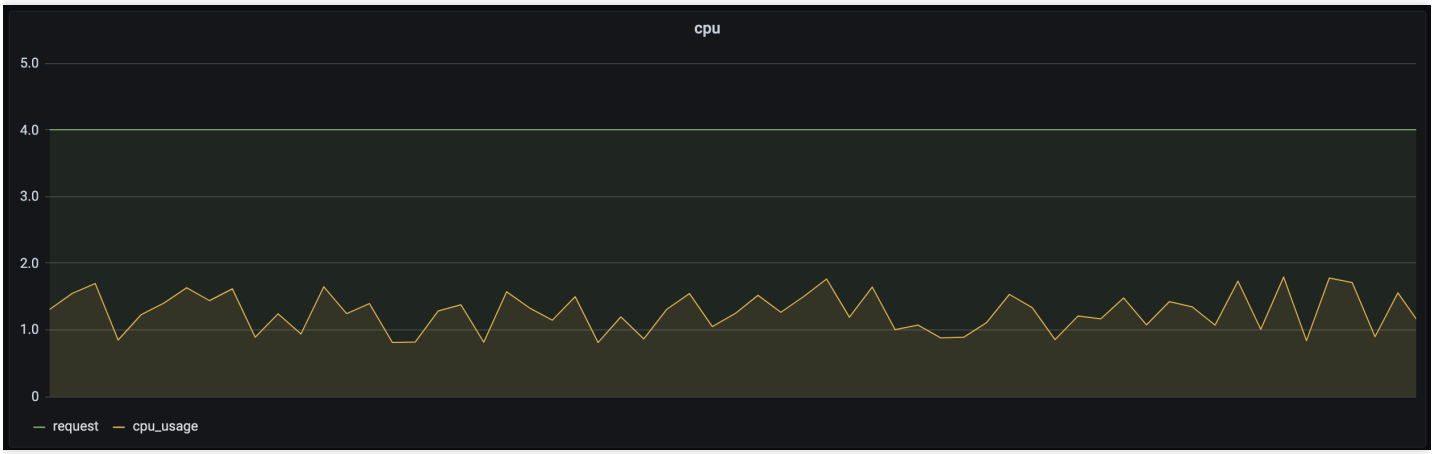
1. 为什么 Kubernetes 集群中的 **CPU 和内存资源利用率**通常都如此之低？
2. 现阶段在 TKE 上面有哪些产品化的方法可以轻松提升资源利用率？

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

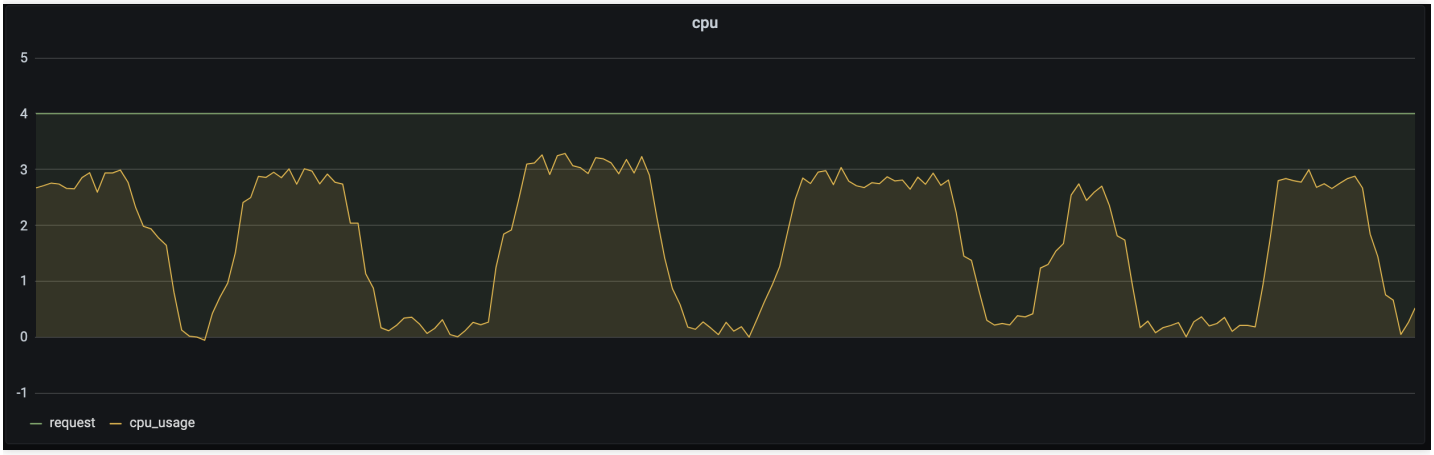
Kubernetes 中的 Request（请求）字段用于管理容器对 CPU 和内存资源预留的机制，保证容器至少可以达到的资源量，该部分资源不能被其他容器抢占，详情见 Kubernetes 官方文档。当 Request 设置过小，无法保证业务的资源量，当业务的负载变高时无力承载，因此用户通常习惯将 Request 设置得很高，以保证服务的可靠性。但实际上，业务在大多数时段时负载不会很高。以 CPU 为例，下图是某个实际业务场景下容器的资源预留（Request）和实际使用量（CPU_Usage）关系图：



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

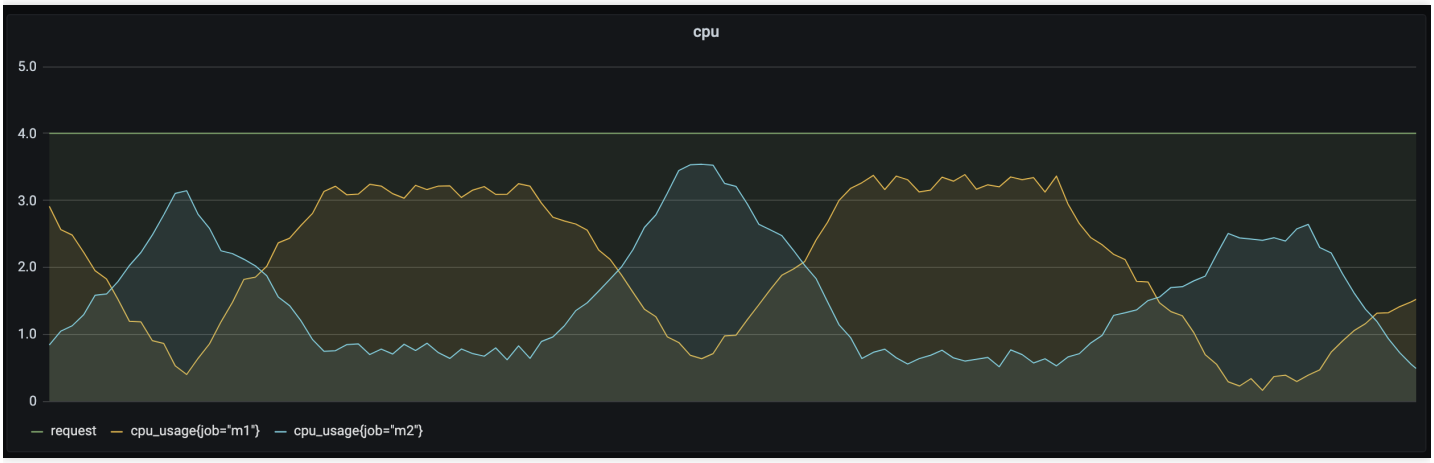
大多数业务存在波峰波谷，例如公交系统通常在白天负载增加，夜晚负载减少；游戏业务通常在周五晚上开始出现波峰，在周日晚开始出现波谷。如下图所示：



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

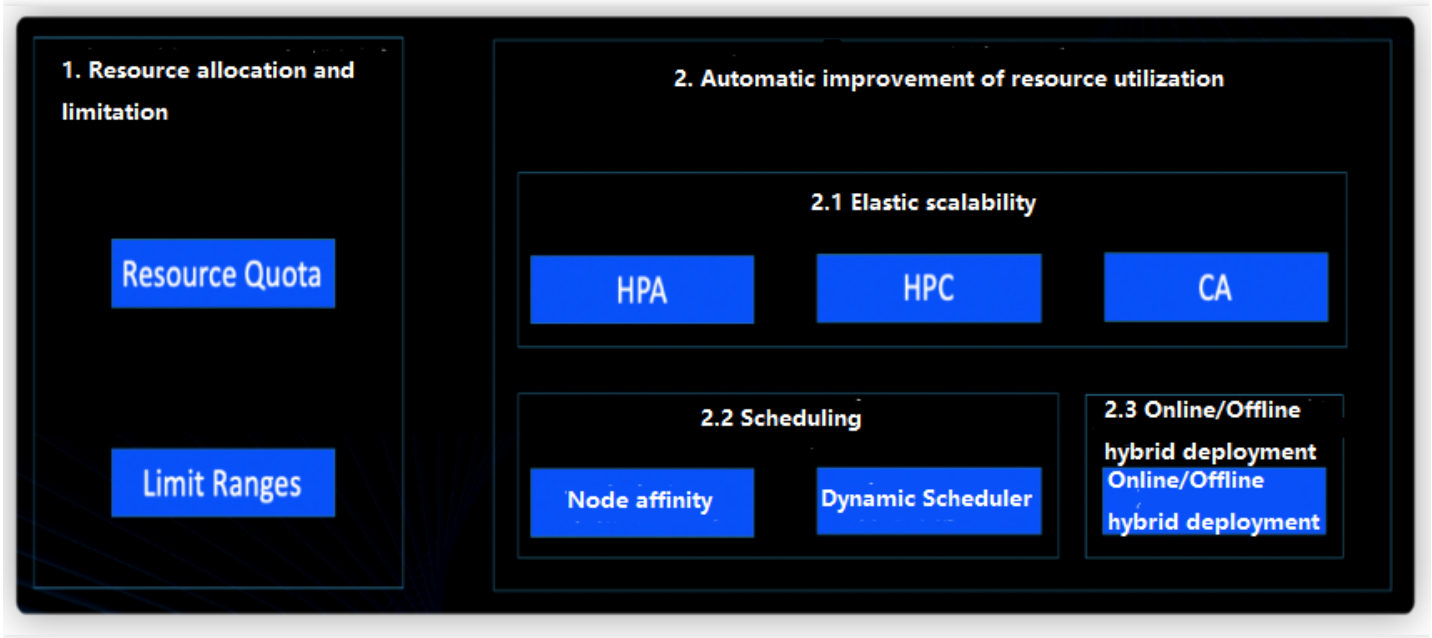
在线业务通常白天负载较高，对时延要求较高，必须优先调度和运行；而离线的计算型业务通常对运行时段和时延要求相对较低，理论上可以在在线业务波谷时运行。此外，有些业务属于计算密集型，对 CPU 资源消耗较多，而有些业务属于内存密集型，对内存消耗较多。



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
- 使用 Limit Ranges 限制资源

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。

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
- 通过 HPC 定时扩缩容
- 通过 CA 自动调整节点数量

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。

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
- 动态调度器

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。

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。

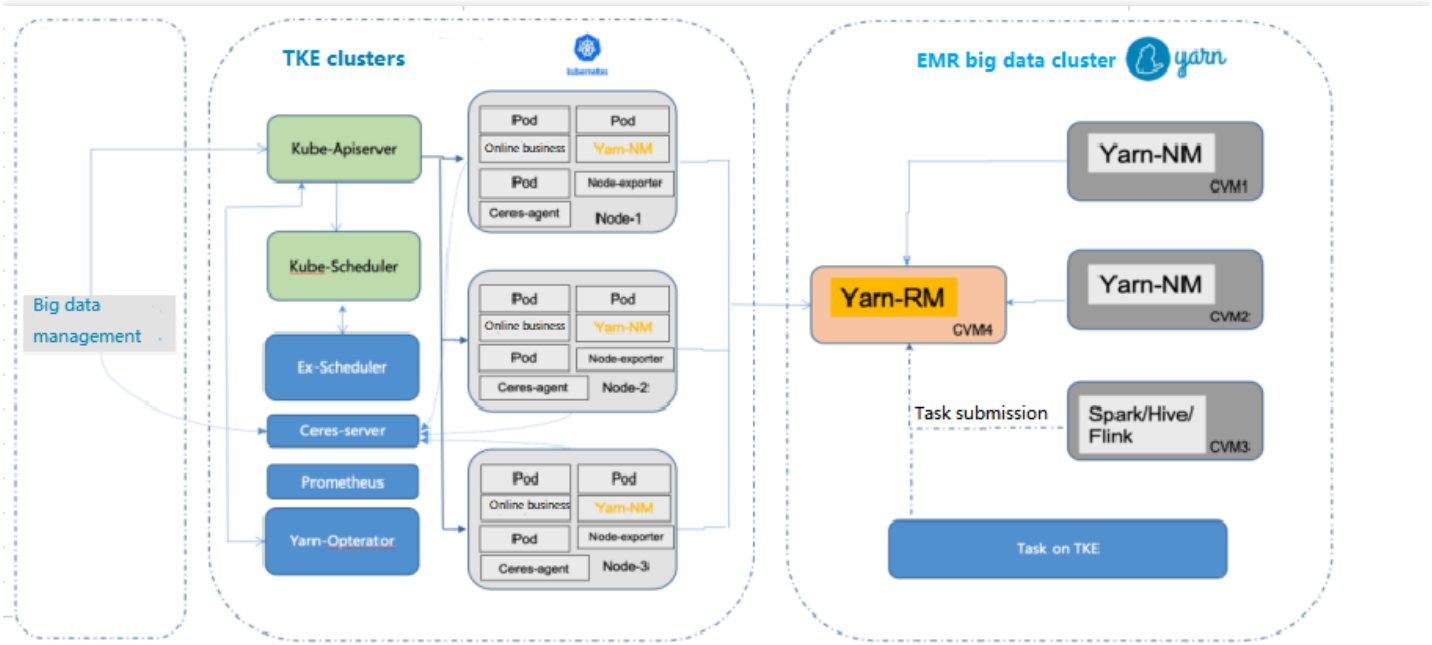
离在线业务混部使用场景

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。

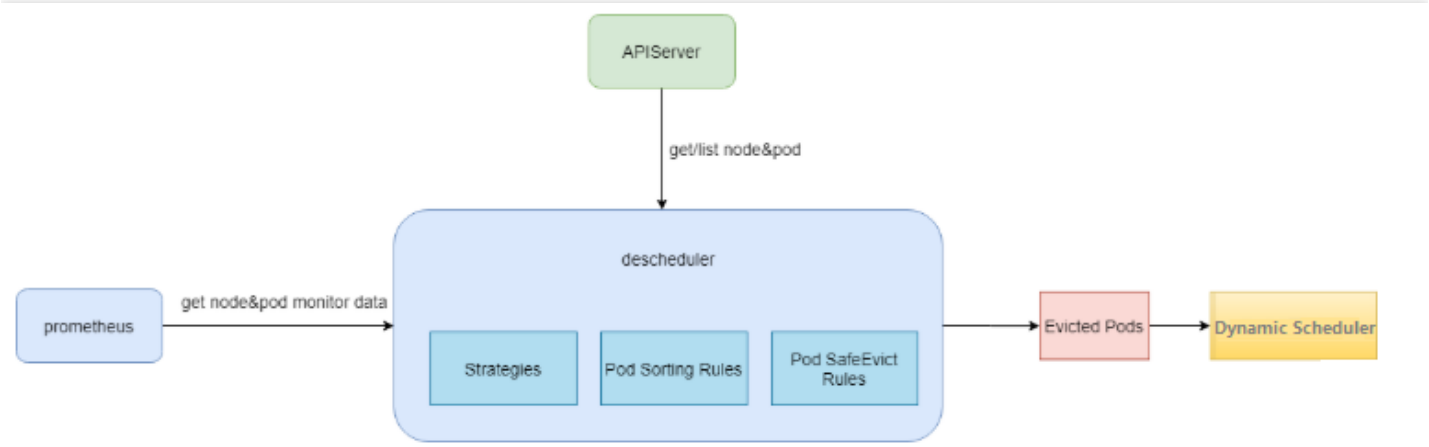


如何权衡资源利用率与稳定性

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。

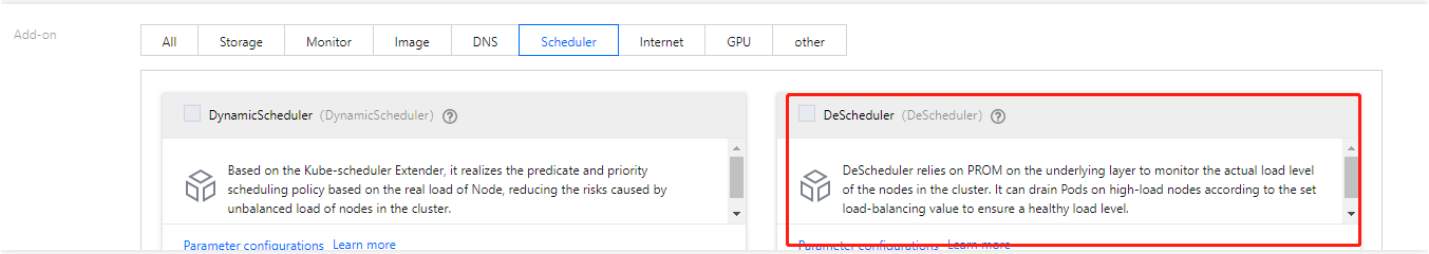
为了减小企业成本控制之路上的顾虑，TKE 还提供了**重调度器**来保障集群负载水位在可控范围内。重调度器与动态调度器的关系可以参考下图，重调度器主要负责“保护”节点中已经负载比较“危险”的节点，优雅驱逐这些节点上的业

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》。