微服务引擎

「DaoCloud 道客」微服务引擎是面向业界主流微服务生态的一站式微服务管理平台，主要提供微服务治理中心和微服务网关两个维度的功能， 具体包括服务注册发现、配置管理、流量治理、服务级别的链路追踪、API 管理、域名管理、监控告警等，覆盖了微服务生命周期中的各种管理场景。 微服务引擎具有很强的兼容性，不仅可以无缝对接 DCE 5.0 的其他组件，也可以完美兼容 Spring Cloud、Dubbo 等开源生态， 帮助您更便捷地使用开源微服务技术构建自己的微服务体系。

- : material-run-fast:{ .lg .middle } **快速入门**

  - 微服务引擎管理组件
  - 微服务引擎集群初始化组件
  - 选择工作空间
  - skoala-init 故障排查

- : material-connection:{ .lg .middle } **服务对接规范**

  - 接入 Sentinel
  - 接入 Nacos SDK
  - 接入 Sentinel 监控
  - 接入 Sentinel 集群流控

- : fontawesome-solid-cubes-stacked:{ .lg .middle } **最佳实践**

  - 示例应用体验微服务治理

  - 在云原生微服务中使用 JWT 插件
  - 微服务网关接入认证服务器
  - 网关 API 策略

- : material-engine:{ .lg .middle } **云原生微服务**

  - 导入/移除服务
  - 服务治理
  - Nacos 压力测试报告

- : simple-amazonapigateway:{ .lg .middle } **云原生网关**

  - 网关管理
  - API 管理
  - 服务接入
  - 监控告警
  - 域名管理
  - 网关压力测试报告

- : simple-legacygames:{ .lg .middle } **传统微服务**

  - 注册配置中心
    - 微服务列表
    - 微服务配置管理
    - 微服务监控
  - 接入注册中心
    - 微服务管理

- 链路追踪

微服务引擎架构

微服务引擎架构

微服务引擎功能特性

DCE 5.0 微服务引擎具备以下功能特性：

功能	子功能	描述
微服务注册与发现	统一纳管	统一纳管传统微服务和云原生微服务，实现从传统微服务生态向云原生微服务生态的平稳过渡，助力企业走向云原生化。
	Nacos 托管中心	支持创建 Nacos 托管中心，管理微服务命名空间、治理微服务流量、管理微服务配置、链路追踪与监控等。
	接入支持	支持接入 Eureka、Zookeeper、Nacos、

功能	子功能	描述
		Consul 四类传统的微服务注册中心。
	云原生支持	支持接入 Kubernetes 和 Service Mesh 两类云原生微服务注册中心。
微服务流量治理	线上流量治理	在流量治理层面，采用线上流量治理方案，可以快速与主流开源微服务框架集成，用 Sentinel 和 Mesh 解决不同生产情况下的痛点。
	Sentinel 支持	支持通过 Sentinel 使用流控、熔断降级、热点、系统、授权、集群流控等规则治理传统微服务的东西向流量。
	服务网格能力	支持结合服务网格的

功能	子功能	描述
		能力使用虚拟服务、目标规则、网关规则在网格中治理微服务流量。
	Istio 支持	支持通过 Istio 使用负载均衡、熔断、离群检测、重写、故障注入、重试、超时、全局限流等规则针对服务端口进行东西流量治理。
	Wasm 插件	支持云原生微服务治理通过创建配置 Wasm 插件方式扩展能力。
	流量泳道模式	支持使用流量泳道模式下的流量管理。
微服务配置中心	Nacos 托管注册中心	Nacos 托管注册中心可作为微服务的配置管理器，可以从不同项目中抽取通用配置

功能	子功能	描述
		事先统一管理，也可以为同一项目应用多个不同配置，实现差异化管理。
	配置文件隔离	基于微服务命名空间和分组 (Group) 对配置文件进行隔离。
	动态更新	结合 @RefreshScope 注解动态更新配置项。
	历史版本管理	管理配置文件的历史版本，支持版本差异对比并一键回滚到特定版本。
	监听者查询	支持查询当前配置的监听者以及 MD5 校验值。
	灰度发布	支持配置文件定向灰度发布。
	示例代码	提供示例代码，便于新手快速使用客户端

功能	子功能	描述
		编程消费该配置，降低新手使用门槛。
微服务网关	南北流量管控	微服务网关肩负管理微服务南北流量管控的重要作用，提供 API 管理、接口限流、多种策略安全认证、黑白名单、路由转发、MockAPI 等能力，同时提供企业级高性能和高扩展的云服务能力。
	多网关管理	原生支持对容器管理模块中的多集群、多命名空间的网关实例进行管理，支持网关实例的全生命周期管理。
	API 策略管理	通过图形化界面进行 API 的增删改查，配置 API 策略，例如

功能	子功能	描述
		负载均衡、路径改写、超时配置、重试机制、请求头/响应头重写、WebSocket、本地限流、健康检查等，同时保障原生 API 的能力不受影响。
	监控功能	微服务网关在部署时会自动配置监控等功能。每个网关都自带完善的资源监控和网关业务监控。
插件中心	功能丰富的插件	提供了一系列功能丰富的插件，包括安全加固、流量管理和数据缓存等，以增强您的使用体验。此外，我们还支持自定义插件，让您能够根据自身需求进行个性化配

功能	子功能	描述
		置。
	插件管理	所有插件均可通过简单的一键操作进行启用或停用，确保您的操作便捷高效。
	JWT 认证	支持 JWT 认证、安全认证、全局限等流插件，自定义多重网关 API 认证策略，一键即可快速接入到网关实例中。
	自定义 Wasm 插件	支持用户自定义创建 Wasm 插件。

适用场景

「DaoCloud 道客」微服务引擎是面向业界主流微服务生态的一站式微服务管理平台，主要提供微服务治理中心和微服务网关两个维度的功能， 具体包括服务注册发现、配置管理、流量治理、服务级别的链路追踪、API 管理、域名管理、监控告警等，覆盖了微服务生命周期中的各种管理场景。

适用场景

基于所提供的各项功能，微服务引擎可用于微服务的注册与发现、配置管理、微服务流量治理，以及微服务网关管理等场景。典型的应用场景如下：

- 一站式管理大量异构微服务

  随着应用服务的爆发式增长，微服务的数量越来越多，使用的架构也可能各不相同，传统的微服务和云原生微服务并存。 各个微服务之间互相调用，互相依赖，牵一发而动全身，容易出现级联效应，造成系统雪崩。随着微服务系统的运维难度和成本越来越高， 用户需要一个能够同时管理传统微服务和云原生微服务、监控服务信息、追踪服务级别的链路调用情况、统一管理微服务配置、提供微服务网关的一站式 产品。微服务引擎完全覆盖了这些场景需求，能够很好地满足用户需要。

- 从传统到云原生的转变和过渡

  云原生在弹性扩展、屏蔽底层差异、故障处理等方面具有显著优势，一些企业在云原生浪潮的影响下， 希望采取稳态模式逐渐从传统的微服务架构转向云原生微服务架构，有些企业希望采取敏态模式实现快速转变。 无论采取稳态还是敏态模式，微服务引擎都是很好的选择，因为其支持传统微服务和云原生微服务的统一纳管，支持 Sentinel 和 Service Mesh 两种流量治理模式，分别适用于传统微服务和云原生微服务的东西向流量治理。

- 微服务网关的可视化与高效能

  很多开源的网关产品仅支持命令行操作，使用门槛高、难度大。某些情况下，多个微服务共用一个网关，网关资源开销大， 而且可能逐渐成为整个系统的资源瓶颈。使用「DaoCloud 道客」微服务引擎在 Contour 的基础上增加界面操作能力， 大大降低了使用门槛和维护成本，可以实现多网关管理，在不同集群和不同命名空间下

轻松创建多个网关实例。 这些网关实例之间互相隔离，因而具有更高的可用性和稳定性。此外，还可以使用网关路由的级联功能，实现微服务的蓝绿部署动态切换。

产品优势

微服务引擎包含微服务治理中心和微服务网关两个模块，提供微服务的注册发现、流量治理、配置管理、网关 API 等一组简单实用且高性能的微服务治理能力，帮助从传统微服务架构稳定升级到云原生微服务架构。

相比其他同类产品，「DaoCloud 道客」提供的微服务引擎具有以下优势：

- 一站式治理

  提供服务注册发现、配置管理、流量治理、链路追踪、指标监控、网关管理、API 管理、域名管理、监控告警、网关策略等功能，满足微服务生命周期中各个阶段的管理需求，实现一站式治理。

- 无感迁移

  完全兼容 Nacos 开源注册中心和 Envoy、Contour 开源网关，支持在零代码改造的前提下完成从自建服务到「DaoCloud 道客」微服务引擎的迁移。传统微服务无需作任何改动，即可通过注册中心接入到微服务引擎，进而实现流量治理、配置管理、链路追踪、指标监控等功能。

- 平稳过渡

  统一管理传统微服务和云原生微服务，支持接入传统注册中心（Zookeeper、Eureka、Nacos、Consul）和云原生注册中心 （Kubernetes、Service Mesh）类型的注册中心，实现从传统微服务生态向云原生微服务生态的平稳过渡，助力企业走向云原生化。

- **开放兼容**

  微服务引擎支持传统和云原生的注册中心，也支持 Spring Cloud、Dubbo 等主流的开源微服务框架， 以及 Envoy、Contour 等开源网关组件。此外，还可以和 DCE 5.0 的多云编排、数据服务中间件、服务网格、应用工作台等模块自由组合，实现更多定制化、精细化的功能。

- 可视化界面与数据

  通过功能封装对外暴露简单易用的交互页面，支持通过简洁直观的 UI 界面进行所有操作，降低操作难度， 真正实现点点鼠标就可以完成微服务整个生命周期中的各类管理操作。

微服务引擎 Release Notes

本页列出微服务引擎的 Release Notes，便于您了解各版本的演进路径和特性变化。

*[skoala]: DaoCloud 微服务引擎的内部开发代号

2025-02-24

v0.45.0

- **新增** 网关自定义日志字段配置的功能
- **修复** 网关基于 Header 路由时配置中 Header Key 无法支持大写字母的问题

2025-01-24

v0.44.0

新增

- **新增** 托管 Nacos 通过 LB 暴露服务功能
- **新增** 托管 Nacos 节点亲和性配置功能
- **新增** 网关节点亲和性配置功能
- **新增** 托管 Nacos 自定义配置能力

修复

- **修复** Skoala Agent TLS 漏洞
- **修复** Skoala Agent CVE-2022-1996，CVE-2024-45337 以及 CVE-2024-24790
- **修复** 通过界面操作网关实力会导致自定义日志配置丢失的问题

2024-12-31

v0.43.4

- **优化** 界面与用户体验

2024-12-25

v0.43.3

- **修复** Nacos 负载均衡模式 API 的问题

- **修复** 全局限流服务器标签展示不正常的问题
- **修复** Nacos 列表日志不正常的问题

!!! note
为支持网关日志滚动更新与添加调试工具，升级网关运行时至私有版本 v0.32.0-dcv3（基于 Envoy v1.32.0 构建，如不需要网关日志滚动更新功能，仍然可以只用社区版 Envoy v1.32.0）

2024-11-30

v0.43.2

新增

- **新增** 创建网关时页面应该支持自定义注解
- **新增** 网关日志文件自动拆分滚动更新能力

修复

- **修复** 当用户名为中文时资源无法加载的问题
- **修复** 网关高级配置默认值不生效的问题
- **修复** 编辑网关时出现无意义的注解问题

优化

- **优化** 网关日志滚动更新逻辑，避免日志丢失情况（更新网关运行时至私有版本 v0.31.0-dcv2）
- **优化** 禁用 skoala-agent 的 /debug/pprof 接口，避免安全风险
- **优化** 更新 Nacos 2.4.3 替代 2.4.2.1 版（稳定版仍为 2.3.2）
- **优化** 网关默认网络连接数为较大值，可有效避免由于网络连接数默认值过小导致的性

能问题

- **优化** 移除 Skoala Init Helm Chart 针对 Insight CRD ServiceMonitor 的检测，避免安装功能不完整的 Skoala Init 组件
- **优化** 升级网关控制面 Contour 至 v1.30.1-0be3efa
- **优化** 为支持网关日志滚动更新升级网关运行时至私有版本 v0.31.0-dc（如不需要网关日志滚动更新功能，仍然可以只用社区版 Envoy v1.31.0）

2024-10-31

v0.42.1

新增

- **新增** MetalLB 的安装状态检测支持
- **新增** 网关支持自定义 dns-lookup-family 能力
- **新增** 全局限流规则管理
- **新增** 支持内置全局限流插件模板

修复

- **修复** 网关插件在界面中无法展示查看的问题
- **修复** 调用容器管理组件注解不正常的问题
- **修复** 网关 API 删除操作不符合预期的问题
- **修复** 插件中心主菜单跳转链接不正确的问题
- **修复** 固定 IP 设置后界面展示不正常的问题

- **修复** 网关限流策略不能关闭的问题
- **修复** 云原生微服务使用限流规则异常的问题
- **修复** 托管资源展示的小数位数过多的问题

优化

- **优化** 托管 Nacos 8848/9848 端口对应 NodePort 选择逻辑，由随机端口改为筛选可用端口
- **优化** 更新 Nacos 2.4.2.1 替代 2.4.1 版（稳定版仍为 2.3.2）
- **优化** 升级网关控制面 Contour 至 v1.30.0-54ceade
- **优化** 增强网关插件缓存读取的机制，增加等待时间

2024-09-30

v0.41.3

新增

- **新增** 域名为 HTTPS 时 API 允许配置非安全请求

修复

- **修复** 不能按照命名空间展示 skoala-init 安装情况的问题
- **修复** 网关统计在网关运行时多副本情况下不符合预期的问题
- **修复** 安装时 CRD 不存在的问题
- **修复** 离线镜像逻辑
- **修复** 审计日志部分记录不全的问题

- **修复** 当托管 Nacos 所在集群失联时 Nacos 实例状态不符合预期的问题
- **修复** 当网关所在集群失联时网关实例状态不符合预期的问题
- **修复** 网关 API 的域名为 HTTP 调试的时 HTTPS POST 请求不选择跳过证书验证时调试失败的问题
- **修复** 网关 API 策略展示逻辑不正确的问题
- **修复** 当托管 Nacos 实例的端口被占用时 Nacos 实例状态不符合预期的问题
- **修复** 网关工作节点实例列表没有 Pod IP 的问题

优化

- **优化** 证书及令牌管理逻辑
- **优化** 为云原生微服务添加自定义插件支持
- **优化** 托管 Nacos 所支持的版本中由 v2.4.0.1 升级为 v2.4.1

2024-09-02

v0.40.1

- **修复** 网关共享 LB VIP 时无法使用的问题

2024-08-26

v0.40.0

新功能

- **新增** Nacos 基于元数据标签查询服务实例

- **新增** 网关实例概览中，**展示**节点固定 IP

修复

- **修复** 网关控制面某些情况下重启不符合预期的问题
- **修复** 托管 Nacos 命名空间相关 API 无法使用的问题
- **修复** 网关 API 导入时 API 名称校验不符合预期的问题
- **修复** 管理组件 CVE-2024-41110 漏洞

优化

- **优化** 升级网关运行时 Envoy 到 v1.31.0
- **优化** 升级网关控制面 Contour 到 v1.30.0-d59d534
- **优化** 扩展网关证书有效时间配置，默认时间为 5 年，更新或新创建网关时皆可生效
- **优化** 升级 Nacos 到 v2.4.0.1，默认版本生然为稳定版 v2.3.2
- **优化** Nacos 命名空间删除时的相关资源依赖检测
- **优化** 网关删除功能，现在允许强制删除工作异常的网关实例
- **优化** 网关域名安全策略校验逻辑
- **优化** 网关 API 导入时异常信息的展示能力

!!! note
任意版本升级到 0.40.x 时，由于 Gateway API 社区版本原因需要手动处理 CRD 升级，
参考 [Skoala 0.40.x 升级注意事项](#skoala-040x)。

2024-08-16

v0.39.4

- **修复** 托管 Nacos 命名空间相关 API 无法使用的问题.

2024-08-15

v0.39.3

- **修复** 托管 Nacos 命名空间相关 API 无法使用的问题.

Skoala 0.40.x 升级注意事项

影响版本

从任意版本升级至 0.40.x。

影响说明

0.40.x 版本中更新了相关自定义资源（CRD），由于在 skoala-init Chart 对应组件 crds 目录中的自定义资源不会自动随安装被更新， 本次更新的 CRD 为 gateway-api CRD，BackendTLSPolicy 升级需要手动操作。 由于本次更新的内容存在社区兼容性问题，详情参考 kubernetes-sigs 相关 issue。

升级步骤

请按照如下步骤手动更新需要升级的网关 CRD 文件：

1. 删除 BackendTLSPolicy：

   ```
   kubectl delete crds `kubectl get crds | grep -E "backendtlspolicies.gateway.networking.k8s.io"`
   ```

2. 手动更新网关相关 CRD 文件

   ```
   # projectcontour 相关 crd
   kubectl apply -f skoala-init/charts/contour-provisioner/crds/contour.yaml
   # gateway-api 相关 crd
   kubectl apply -f skoala-init/charts/contour-provisioner-prereq/crds/gateway-api.yaml
   ```

2024-08-06

v0.39.2

- **修复** 网关插件名称展示错误的问题
- **修复** 分布式事务组件数据库地址错误时无法编辑的问题
- **修复** 托管 Nacos 实例在以 NodePort 方式部署时端口被占用情况下无法编辑的问题
- **修复** 网关请求 Top 10 总请求数量数据不符合预期的问题
- **修复** Sesame 管理组件注入 Istio Sidecar 时报错的问题
- **优化** 适配 Istio v1.23.0
- **优化** 解决 Hive 管理组件线程泄露的问题

2024-07-31

v0.39.1

- **修复** Nacos 镜像版本错误的问题

2024-07-25

v0.39.0

新功能

- **新增** Nacos 支持集成 LDAP 认证
- **新增** Nacos 删除前检测能力

修复

- **修复** 网关及插件相关权限问题
- **修复** Nacos 异常情况启动问题
- **修复** Nacos 删除操作接口返回不符合预期的问题
- **修复** 网关开启后默认添加标签不正确的问题

优化

- **优化** Docker 基础镜像至 3.20.1
- **优化** 权限逻辑，自动通过 API 定义生成权限点
- **优化** 弃用 ListGatewayPodsByType

2024-07-02

v0.38.2

- **修复** 网关统计 Top10 API 相关展示问题

2024-06-25

v0.38.1

- **新增** 网关接入注册中心服务支持自动填充和多实例
- **新增** 网关日志自定义数据
- **新增** 网关 API 文档中集成 API 测试能力
- **修复** 离线包公共镜像拉取问题

2024-06-25

v0.38.0

修复

- **修复** 设置 Nacos 环境变量不生效问题
- **修复** 当网关内存设置为小数时与最大堆内存的配比错误的问题
- **修复** 网关日志中涉及域名查询的权限问题
- **修复** 接入注册中心相关时间显示不正常的问题

优化

- **优化** 升级 Ghippo 集成 SDK 至 v0.28.0-dev1 版
- **优化** 配合新版本 Ghippo SDK 调整权限依赖逻辑

2024-06-04

v0.37.1

- **修复** 托管 Nacos 详情中版本展示不正确的问题
- **修复** 微服务 JVM 监控跳转链接不正确的问题
- **修复** 流量泳道引流规则接口调用不正常的问题
- **修复** 分布式事务对接 Nacos 时用户名密码配置不符合预期的问题

2024-05-27

v0.37.0

新增

- **新增** 支持分布式事务组件在概览中的异常信息
- **新增** 网关日志中支持状态码筛选
- **新增** nacos jvm 实现可配置
- **新增** 新增网关 API 测试相关的日志查询功能
- **新增** 支持网关外接服务多实例能力
- **新增** 支持 Seata 关联的 Nacos 密码自定义

修复

- **修复** 网关 API 详情页面显示异常的问题
- **修复** 网关本地限流策略不正常的问题
- **修复** 域名级别鉴权插件关闭不正常的问题
- **修复** 网关根命名空间过滤和检查的问题
- **修复** 英文版本 Sentinel 监控面板中出现中文的问题

优化

- **优化** 升级托管 Nacos 版本至 2.3.2 替代 2.3.1 版
- **优化** API 并添加按工作空间隔离能力
- **优化** 网关插件及自定义插件逻辑

- **优化** **服**务网格集成逻辑
- **优化** **网关控制面**组件版本及相关自定义资源
- **优化** **网关** API 导出功能中跳过流量泳道相关 API

!!! note
重要：0.37.x 版本中更新了相关自定义资源（CRD），由于在 Chart crds 目录中的自定义资源不会自动随安装被更新，
请升级后手动应用相关 CRD 到工作集群，该文件位于：
skoala-init/charts/contour-provisioner/crds/contour.yaml

2024-04-30

v0.36.1

- **修复** **云原生微服**务流量泳道引流规则权限点缺失的问题
- **修复** **网关注入网格**边车后链路信息上报不正常的问题

2024-04-26

v0.36.0

新增

- **新增** **支持网关自定**义插件

修复

- **修复** **分布式事**务相关 Nacos **和数据**库配置不正常的问题
- **修复** **网关内存**资源配置展示不准确的问题
- **修复** Nacos **和分布式事**务组件异常信息不准确的问题
- **修复** 带有证书的网关 API 编辑保存不正常的问题

- **修复** **网关日志**查询有时不正常的问题

优化

- **优化** Nacos **至** 2.3.1 **替代其** 2.3.0 **版**
- **优化** **网关** API 测试对于 ClusterIP **不可用**时的提示

2024-04-11

v0.35.2

- **修复** **全局管理中微服**务引擎模块权限点不准确的问题
- **修复** **网关全局限流保存不正常的**问题
- **修复** **网关**创建时内存配置不正常的问题
- **修复** **网关运行**时最大堆内存配置不正常的问题
- **修复** **接入注册中心** Eureka **相关**时间显示不正常的问题
- **修复** **自定**义密码的 Nacos 创建 Seata **不正常的**问题
- **修复** Seata **初始化数据**库异常的问题

!!! note
重要：0.35.x 版本中更新了 Gateway API 相关自定义资源（CRD），由于 Chart 更新不会自动应用 CRD 变更，
请升级后手动应用该位置的 CRD：skoala-init/charts/contour-provisioner-prereq/crds/gateway-api.yaml

2024-04-03

v0.35.1

- **修复** **网关中**创建域名时报错的问题

- **修复** 托管 Nacos 资源检查不符合预期的问题
- **修复** 网关域名开启健康检查端口并在 API 增加健康检查配置后 API 状态异常的问题
- **修复** 网关 API 统计概述不正确的问题
- **修复** 接入注册中心状态不正常的问题
- **修复** 创建流量泳道时相关资源注解不正常的问题
- **修复** 网关自定义插件功能相关问题

!!! note
重要：0.35.x 版本中更新了 Gateway API 相关自定义资源（CRD），由于 Chart 更新不会自动应用 CRD 变更，
请升级后手动应用该位置的 CRD：skoala-init/charts/contour-provisioner-prereq/crds/gateway-api.yaml

2024-03-23

v0.35.0

新增

- **新增** 云原生网关 API 可以通过请求参数进行路由匹配的能力
- **新增** 分布式事务 Seata 自定义密码的能力
- **新增** 分布式事务 Grafana 面板时间筛选能力
- **新增** 支持按命名空间创建筛选云原生微服务流量泳道的能力

修复

- **修复** 接入注册中心类型相关的问题
- **修复** 网关实例数量为零时提示信息不准确的问题
- **修复** 云原生微服务相关的逻辑

- **修复** **流量泳道列表首次加**载不刷新的问题
- **修复** Sentinel **密**码展示不正确的问题
- **修复** Envoy **作**为 xdsServer 时新建 API 导致空指针的问题
- **修复** **当网格**实例没有服务时分页异常的问题
- **修复** **由于服**务网格模块 API 变更产生的问题

优化

- **优化** **升**级适配分布式事务 Seata 2.0 **支持的能力**
- **优化** **将托管注册中心** Nacos 2.3.0 **作**为默认版本
- **优化** **网关** API 统计图表算法改为增量统计方式

2024-03-06

v0.33.4

- **修复** **原生微服**务相关的逻辑
- **修复** **流量泳道列表首次加**载不刷新的问题
- **修复** Sentinel **密**码展示不正确的问题
- **修复** **由于网格模**块 API 变更产生的集成问题

2024-03-01

v0.33.3

- **修复** **云原生微服**务相关的逻辑
- **修复** **由于网格模**块 API 变更产生的问题

2024-02-02

v0.34.0

- **新增** 根据应用工作台需求在网关查询端口中添加网关所部署的命名空间信息
- **优化** 网关日志分页查询的总数，于可观测逻辑匹配，限制为最多展示 10000 条列表日志数据
- **优化** 网关日志查询的错误提示

2024-02-01

v0.33.2

- **修复** 网关服务组件会异常重启的问题
- **修复** 网关日志导出不正常的问题
- **修复** 当网关工作负载副本为 0 时展示状态不正常的问题
- **修复** 网关注入边车时网关控制平面不重启应用配置的问题
- **修复** 网关列表中返回的边车注入状态与实际不符的问题
- **优化** 至网格模块最新 SDK 修复云原生微服务相关接口逻辑

2024-01-30

v0.33.1

- **修复** Nacos 级联删除相关逻辑

2024-01-18

v0.33.0

- **新增** 网关支持自定义逻辑插件
- **修复** 网格类型网关服务列表查询异常的问题
- **修复** 网关 API 统计中的请求完成数不准确的问题
- **修复** 托管 Nacos 下 Seata 和 Sentinel 配置级联删除的问题
- **修复** 网关调试日志中的 Header Check 错误
- **修复** 部分逻辑由于滑动 TTL 引发的不能获取最新数据问题
- **优化** API 密码返回逻辑使用 Base64 编码，增强安全性
- **优化** 云原生网关边车注入实现，遵循 Gateway API 标准

2024-01-18

v0.32.0

新功能

- **新增** 云原生微服务监控资源
- **新增** 接入注册中心功支持开启认证模式的注册中心接入
- **新增** ARM 架构离线安装包发布流程

修复

- **修复** 更新域名时会错误更新插件引用的逻辑
- **修复** 服务列表查询服务端口错误的问题

- **修复** **日志**协议字段展示错误的问题
- **修复** **流量泳道的服**务列表接口返回结果未排序的问题
- **修复** **流量泳道的服**务列表中实例数量不正确的问题
- **修复** **云原生网关已**经创建了命名空间刷新报错的问题
- **修复** **接入的** Kubernetes **集群中的**实例状态与实际不符的问题
- **修复** **接入的网格中服**务的实例列表结果有误的问题
- **修复** **网关中** Nodeport 类型的服务如果对应工作负载停止地址不正确的问题
- **修复** **接入注册中心中** Nacos **服**务在非默认 Group **中展示异常的**问题

优化

- **优化** 应应用工作台要求优化路由权重总和大于零即可
- **优化** **网关** API **中服**务的权重校验逻辑
- **优化** **集群接入地址**变化引发的资源首次操作不正常问题

2024-01-03

v0.31.2

- **修复** **网关** API **中缺少**协议字段展示的问题
- **修复** **集群重新接入后地址**变更导致的托管资源异常问题
- **修复** **网关**针对异常访问统计不正确的问题
- **修复** **集群服**务 NodePort **端口**为 0 **的**问题
- **修复** **自**动填充集群服务和网格服务的状态提示不准确的问题
- **修复** Init Chart **在** insight-system **命名空**间不存在时安装异常的问题

2023-12-26

v0.31.1

- **修复** 网关管理组件的日志文件缺失问题

2023-12-26

v0.31.0

新增

- **新增** 对托管 Nacos 2.3.0 版本的支持
- **新增** 流量入口支持服务和 API 网关服务
- **新增** 离群实例检测
- **新增** 增强流量泳道拓扑的细节（流量指向，版本等）
- **新增** 网关运行时的堆内存高级配置

修复

- **修复** 网关服务列表和服务详情中没有连接地址的问题
- **修复** 网关 API 更新后 label 缺失的问题
- **修复** 网关 API 路由规则重复的问题
- **修复** 网关运行时最大堆内存为空值的问题
- **修复** 托管 Nacos 集成网格服务时的接入服务问题
- **修复** 网关注入 Istio 边车注解不正确的问题
- **修复** 网关查询 Nacos 服务不正常的问题

- **修复** **流量泳道服**务名过长导致无法创建引流规则的问题
- **修复** **网关更新服**务端口异常的问题
- **修复** 创建托管资源时资源配额计算异常的问题
- **修复** **网关** API **列表添加根据域名**筛选的逻辑问题
- **修复** **云原生微服**务治理重复端口的问题
- **修复** **当** API **存在多路由**时，请求头始终会进行判断，**并且判断条件重复的**问题
- **修复** HTTPS **域名的** API 测试异常报错问题
- **修复** 删除 Sentinel **功能**时异常的问题
- **修复** **网关日志**为空时导出日志导致组件重启的问题
- **修复** **托管** Nacos 资源信息接口返回的控制台地址为空的问题
- **修复** **托管** Nacos **名称**为 -seata **后**缀时开启 Seata **插件失**败的问题
- **修复** **托管** Nacos **管理中**间件 MySQL 查看权限的问题
- **修复** **网关接口** v1alpha2 **版本的**标签问题

优化

- **优化** Skoala Agent 组件并添加日志功能
- **优化** **升**级网关控制面（Contour）**至** v1.27.0 **私有分支版本**
- **优化** **升**级网关运行时（Envoy）**至** v1.28.0 **官方版本**
- **优化** **升**级网关依赖 Gateway API **自定**义资源至 v1.0 **正式版**
- **优化** 调整网关日志查询索引（**兼容之前版本**）

2023-12-11

v0.30.2

- **修复** 网关 API 多个重复路由记录的问题

2023-12-01

v0.30.0

- **修复** 网关运行状态不正确的问题
- **修复** Nacos Operator 的状态检测问题
- **修复** 网格服务的边车状态检查问题
- **优化** 网关相关接口
- **优化** 网关日志的查询逻辑支持自定义字段的查询

2023-11-26

v0.29.0

新增

- **新增** API 文档管理、API 查询
- **新增** 流量泳道灰度过程可视化，通过视图直接显示集群内部灰度泳道划分，数据流转情况

修复

- **修复** 网格模式治理的网格实例列表逻辑
- **修复** 托管 Nacos 控制器的一些不同步问题

优化

- **优化** 网关服务列表接口并新增访问地址回显功能
- **优化** 修改服务后同步更新接口的逻辑
- **优化** 网关资源列表的加载速度
- **优化** 网关超时设置的配置细节

2023-11-07

v0.28.1

- **修复** 网关服务列表的排序问题
- **修复** 导入网关 API 的多个换行符问题
- **修复** Seata Operator 镜像不支持离线仓库问题
- **修复** 离线发布流程异常的问题

2023-10-26

v0.28.0

新增

- **新增** 支持网关 API 请求头正则表达式匹配、精准匹配

- **新增** 支持网关 API 批量导入导出
- **新增** 支持托管 Nacos 多版本选择（目前支持 2.0.4、2.1.1、2.2.3 三个版本）
- **新增** 支持 sentinel 控制台密码复杂化
- **新增** 支持托管 Nacos 配置文件定向灰度发布
- **新增** 支持分布式事务（包括 TCC 模式、FMT 模式、SAGA 模式、XA 模式）
- **新增** 支持以 Swagger 标准通过可视化表单界面 导入接口。

修复

- **修复** 托管 Nacos 不同版本支持的问题
- **修复** 网关接口列表请求方法重复的问题
- **修复** Zookeeper 接入注册中心无法接入 TLS 协议实例的问题
- **修复** 网关全局认证开启后域名删除和更新失败的问题
- **修复** 托管 Nacos 命名空间级别 CPU 使用率不正确的问题
- **修复** 托管 Nacos 从 NodePod 模式改为 ClusterIP 模式时 Service 变更不正常的问题
- **修复** Seata 接口验证不正常的问题
- **修复** 接入注册中心切换 Workspace 接入异常的问题
- **修复** 托管 Nacos 相关接口的权限问题
- **修复** 托管 Nacos Grafana 监控面板问题
- **修复** Sentinel Grafana 监控面板问题
- **修复** 整体权限的准确性问题

优化

- **优化** **网关更新**逻辑当网关中域名开启了 HTTPS **之后不允**许关闭网关层面 HTTPS
- **优化** 审计日志的事件准确性
- **优化** **云原生微服**务 WebAssembly **插件**逻辑
- **优化** **概**览页面的网关健康度查询逻辑（**由** Rate 变更为 Increase）
- **优化** Swagger **路径在** URL **中去掉版本信息并支持多版本接口**
- **优化** Skoala Init Helm Chart **支持分布式事**务控制器（Seata Operator）
- **优化** Skoala Agent 证书
- **优化** **所有** Swagger **中接口描述与**详情内容
- **优化** 审计日志格式及自动生成方案
- **优化** **当**发布正式版本时，**会将** Skoala Helm Chart 发布到正式版镜像仓库的 System 项目中

2023-09-01

v0.27.2

- **修复** **前端更新至** 0.18.1 **版修复界面**问题

v0.27.1

修复

- **修复** **批量**删除时返回详情重复的问题
- **修复** **云原生微服**务端口列表中插件信息不能排序的问题

- **修复** 流量泳道列表不正常展示的问题
- **修复** 前端组件的容器名称不统一的问题
- **修复** Sentinel 业务应用监控报表的一些错误

优化

- **优化** 升级 Insight 版本为 0.19.2
- **优化** 升级 Ghippo 版本为 0.20.0

!!! note
重要：微服务引擎版本大于 `v0.24.2` 版时，针对 `v0.24.2` 及之前版本存在不兼容更新，因为网关涉及开源组件仓库地址变更，
所以更新前需要手动删除旧有的 `gateway-api-admission-xxxxx` Job，然后进行正常升级更新操作。

2023-08-25

v0.27.0

新增

- **新增** 支持接入 Consul 注册中心
- **新增** 支持网关实例级别的安全认证配置
- **新增** 支持精确地控制整个集群的流量调用总量
- **新增** 流量泳道

修复

- **修复** 网关服务接入时权限与预定义权限不符的问题
- **修复** 服务治理状态表达不正确的问题

- **修复** 托管 Nacos 配置灰度发布返回异常问题
- **修复** 托管 Nacos 中服务实例列表报错问题
- **修复** 流量泳道服务版本重复的问题
- **修复** 流量泳道中服务列表异常问题
- **修复** 流量泳道服务删除的问题
- **修复** 当错误原因未空时泳道列表空指针异常的问题
- **修复** Skoala-init Chart 表单模式配置不生效的问题

优化

- **优化** 依要求将删除泳道服务从 delete 方法变为 put。
- **优化** 移除多余权限依赖并完善内部权限依赖关系
- **优化** 流量泳道适配界面展示需求
- **优化** 云原生微服务插件接口
- **优化** 添加泳道服务的方式由批量改为单个
- **优化** 前端组件 Deployment 端口由 80 修改为 8080
- **优化** 升级 Ghippo SDK 到 v0.20.0-dev2

!!! note
重要：微服务引擎版本大于 `v0.24.2` 版时，针对 `v0.24.2` 及之前版本存在不兼容更新，因为网关涉及开源组件仓库地址变更，
所以更新前需要手动删除旧有的 `gateway-api-admission-xxxxx` Job，然后进行正常升级更新操作。

2023-08-03

v0.26.2

- **修复** Consul **注册中心接入**时检测不通的问题
- **修复** **网关由于插件**应用异常重启问题
- **修复** **网关**验证插件配置异常问题
- **修复** Sentinel **集群流控**规则编辑异常的问题

!!! note
重要：微服务引擎版本大于 `v0.24.2` 版时，针对 `v0.24.2` 及之前版本存在不兼容更新，因为网关涉及开源组件仓库地址变更，
所以更新前需要手动删除旧有的 `gateway-api-admission-xxxxx` Job，然后进行正常升级更新操作。

2023-07-26

v0.26.1

- **修复** **修复** Agent 组件版本错误问题

2023-07-25

v0.26.0

新增

- **新增** **支持**给网关运行时 Envoy **注入** Istio Sidecar Envoy **代理**
- **新增** **支持** API 级别的全局限流
- **新增** **支持域名**级别的全局限流
- **新增** **支持** API **列表批量操作**（**上**线/**下**线/删除）

- **新增** 支持云原生微服务治理通过插件方式扩展能力，包括：JWT 插件、Auth 插件、全局限流插件等
- **新增** 支持云原生微服务的服务信息详情及端口列表查看

修复

- **修复** Nacos 端口修改不生效的问题
- **修复** Sentinel 集群流控规则保存不生效的问题
- **修复** 删除 Nacos 中非空服务时接口异常的问题
- **修复** 网关监控数据中重复数据的问题
- **修复** 云原生微服务使用插件相关 API 的问题
- **修复** 网关域名命名规则的相关问题
- **修复** 云原生微服务流量泳道的版本错误问题

优化

- **优化** 人大金仓数据库的驱动名由 kb_v8r6 改为 kingbase

2023-07-19

v0.25.0

新功能

- **新增** 将网关运行时的网格注入标识修改被标签方式标记
- **新增** 与中间件关联操作的权限级联选择能力
- **新增** 云原生微服务限流规则详情 API

- **新增** 云原生微服务治理相关 API
- **新增** 为安装托管资源时集群就绪检查添加对应组件版本信息
- **新增** 为各类搜索 API 添加模糊搜索能力

修复

- **修复** 流水线外部镜像扫描地址不对的问题
- **修复** 与中间件关联权限依赖未涵盖目录级别的问题

优化

- **优化** 网关及相关资源变更的连锁校验
- **优化** 网关监听端口由集群 IP 改为容器端口

2023-07-06

v0.24.2

- **修复** 页面体验优化及问题修复

2023-07-05

v0.24.1

- **修复** Skoala-init Chart 表单中版本不是最新的问题
- **修复** 页面体验优化及问题

2023-06-30

v0.24.0

- **新增** 云原生微服务插件相关 API
- **新增** 与中间件关联操作的权限级联选择能力
- **修复** Insight 集成数据异常的问题
- **修复** 网关状态筛选结果异常的问题
- **优化** 网关及相关资源变更的连锁校验

2023-06-26

v0.23.0

新增

- **新增** 网关 API 的批量上下线接口
- **新增** 为引擎组件添加增强健康检查的配置
- **新增** 为缓存逻辑添加标签缓存支持
- **新增** 网关创建和更新时网格边车强制注入选项
- **新增** 网关列表按照状态筛选支持
- **新增** 审计日志集成
- **新增** 在持续集成中添加许可检查
- **新增** 将 Chart 中的敏感信息通过 Secret 存储
- **新增** 升级托管 Nacos 版本至 2.2.3 版

- **新增** Sentinel **版本至** 0.10.5 **版**

修复

- **修复** Insight **集成数据指**标于原始数据对齐
- **修复** **与** mSpider **模**块对接时未继承权限的问题
- **修复** Nacos 实例列表治理开启状态缺失的问题
- **修复** Sentinel **集群流控名称的**问题
- **修复** **整体概**览查询数据不是按照工作空间统计的问题
- **修复** **网关未开启** HTTPS **但域名可以开启** HTTPS **的**问题
- **修复** Skoala-init Chart **表**单数据默认值问题

优化

- **优化** Sentinel Grafana **的使用体**验和问题
- **优化** **托管**资源告警逻辑只展示相关资源告警条目
- **优化** **重构全局管理模**块配置结构
- **优化** **托管** Nacos **的信息**读取通过 Clusterpedia **完成**
- **优化** Insight 对接至 0.17.3 **版**
- **优化** **数据**库中立相关配置的灵活性

2023-05-31

v0.22.2

- **修复** Sentinel **集群流控** API **的**问题

- **修复** Sentinel 规则模糊查询 API 的问题
- **优化** 连接数据库的默认值提高容错性

2023-05-29

v0.22.1

- **修复** 插件 CRD 位置不对的问题
- **修复** OpenAPI 发布流程问题
- **优化** 为 Hive 组件的数据库配置设置默认值

2023-05-26

v0.22.0

新增

- **新增** 托管 Nacos 2.2.x 版本的支持
- **新增** 网关链路功能支持
- **新增** 云原生微服务路径重写 API
- **新增** 云原生微服务超时 API
- **新增** 云原生微服务故障注入 API
- **新增** 云原生微服务重试等 API
- **新增** 云原生微服务使用 WASM 插件功能
- **新增** Skoala-init Chart 添加 JSON Schema
- **新增** OpenAPI 文档发布流程

- **新增** Hive **支持数据**库中立

修复

- **修复** **托管** Nacos **相关** API **有几率出**现空指针的问题
- **修复** **托管** Nacos **端口展示不正确的**问题
- **修复** Grafana **内的** HTTPProxy **拼写**错误
- **修复** Nacos **数据**库初始化 SQL **脚本**问题
- **修复** **网关**组件自定义权限对接问题
- **修复** 调用 Ghippo **空指**针问题
- **修复** Nacos **服**务元数据接口异常问题
- **修复** **更新域名的**错误
- **修复** 调用 Kpanda API 频繁的问题
- **修复** **接入** Insight 获取数据不准确的问题
- **修复** WASM **插件**查询出现错误
- **修复** **更新** API **路由服**务后 API **出**现错误
- **修复** Sentinel Token **服**务器资源问题
- **修复** Zookeeper 链接未关闭问题

优化

- **优化** Sentinel 监控面板结构和数据逻辑
- **优化** **网关域名管理中移除** virtualhost **自定**义资源
- **优化** Sentinel **集群流控的** API 逻辑

- **优化** Ghippo 升级至 0.17.0-rc2
- **优化** 数据库初始化组件 sweet 已于 0.22.0 版废弃，将在 0.23.0 及之后版本彻底移除，从 0.22.0 版开始数据表将自动同步更新，无需人工干预

2023-05-07

v0.21.2

- **修复** Sentinel 监控面板问题
- **修复** 网关被注入网格边车的问题
- **修复** 传统微服务开启网格治理的注册中心地址格式问题
- **修复** 托管注册中心选择中间件实例未按照集群筛选的问题
- **修复** 注册中心统计不正确的问题
- **优化** 更新网关组件到社区最新测试版

2023-04-26

v0.21.1

- **修复** 云原生微服务分页问题
- **优化** 为 Nacos 添加禁用注入 Istio 边车的配置
- **优化** 升级 Insight 版本至 0.16.0 正式版
- **优化** 组件连接数据库重试机制
- **优化** 配合 Istio 治理能力修改 Nacos 的 9848 端口名为 GRPC

2023-04-25

v0.21.0

新增

- **新增** 网关接入内部及外部地址分离展示
- **新增** 云原生微服务治理能力相关 API
- **新增** 告警消息列表 API
- **新增** 网关使用插件相关 API
- **新增** 网关各类插件的逻辑 API

修复

- **修复** 网关更新时 Envoy 配置不更新的问题
- **修复** 网关只能添加单个端口的问题
- **修复** Insight 集成 JVM 查询的问题
- **修复** 云原生微服务治理 API 的问题
- **修复** Sentinel 规则无法存取的问题
- **修复** 链接不到数据库时某些 API 调用会造成程序崩溃的问题
- **修复** 资源状态 API 的问题
- **修复** 云原生微服务治理 API 时间单位问题
- **修复** 域名相关格式校验问题
- **修复** 插件一些字段命名错误的问题

优化

- **优化** Insight **集成到** 0.16.0
- **优化** **部署模板**终端服务名称添加模块名前缀
- **优化** virtualhost crd
- **优化** httpproxy crd
- **优化** skoalaplugin crd

2023-04-21

v0.20.0

新增

- **新增** Sentinel 门户版本
- **新增** **网关域名**级别黑白名单支持
- **新增** **原生服**务治理列表 API
- **新增** **原生服**务治理编辑 API
- **新增** **可**观测 JVM 监控集成
- **新增** **网关**资源工作负载状态展示
- **新增** **网关**负载策略选择配置

修复

- **修复** Contour 镜像版本
- **修复** **自定**义角色功能点及 API **映射**

- **修复** 网关概览 API 排序和条目

优化

- **优化** 插件中心相关 API
- **优化** 配置文件结构
- **优化** 配置参数由直接读取改为配置包实现
- **优化** 管理组件整体包结构
- **优化** 管理组件
- **优化** Contour 升级到 v1.24.3-ipfilter-tracing
- **优化** Envoy 升级到 v1.25.4

2023-04-10

v0.19.4

- **修复** 托管 Nacos 的启动问题

2023-04-10

v0.19.3

- **修复** 前端问题

2023-04-04

v0.19.2

- **修复** Nacos 及 Sentinel 默认验证账号问题
- **修复** 概览网关 API 排序问题

2023-04-04

v0.19.1

- **修复** CVE-2022-31045 漏洞
- **修复** 插件中心 API 问题
- **修复** 网关重启的问题
- **修复** 插件更新时版本不能成功更新的问题
- **修复** Nacos 及 Sentinel 默认验证账号问题
- **修复** 概览内微服务网关网关 API 排序问题
- **修复** Nacos 支持版本回滚至 2.0.4

2023-03-24

v0.19.0

新增

- **新增** 自定义权限点及 API 的实现
- **新增** 注册中心概览相关 API

- **新增** 网关黑白名单相关 API
- **新增** 概览中网关健康度相关 API
- **新增** Nacos 支持版本至 2.1.2
- **新增** 获取 Nacos 及网关版本信息 API
- **新增** 概览中注册配置中心统计收集器
- **新增** 概览中注册配置中心统计 API
- **新增** 云原生微服务服务列表 API
- **新增** 云原生微服务服务导入相关 API
- **新增** 插件中心自定义资源设计
- **新增** 插件中心插件管理相关 API
- **新增** 网关前置流量拦截配置 API
- **新增** 级联资源操作添加事务 (类似) 机制处理包
- **新增** 资源重启功能

修复

- **修复** Nacos Operator 中初始化 Nacos 的数据库脚本问题
- **修复** Sentinel 相关数据概览 API 的问题
- **修复** 网关相关数据的概览 API 的问题
- **修复** 网关生命周期管理减少网关异常重启问题
- **修复** 概览 API 路径大小写问题
- **修复** Nacos 2.1.2 无法创建集群的问题
- **修复** 网关前置流量拦截修改不生效的问题

- **修复** 网关黑白名单 API 的问题
- **修复** Nacos GRPC 端口名字对集成 Istio 产生的问题
- **修复** 每日构建中的外部镜像安全扫描

优化

- **优化** CI 流程并简化不必要的任务
- **优化** 全部资源的更新操作都采用 retry 机制
- **优化** 网关相关功能重构

2023-02-25

v0.18.0

- **新增** 添加注册中心配置中心分离 API
- **新增** 添加概览相关逻辑及 API
- **修复** gateway-api 镜像版本问题
- **修复** 负载均衡模式网关的 IP 池加载问题
- **修复** 健康检查相关问题

2023-02-22

v0.17.1

新增

- **新增** 网关 NodePort 支持

- **新增** 网关 LoadBalancer 支持
- **新增** Sentinel 规则统计 API
- **新增** Sentinel 治理的服务列表 API
- **新增** 网关 API 的 Cookie 重写策略
- **新增** 概览数据定时任务
- **新增** 定时收集异常 Sentinel 任务
- **新增** Sentinel 集群流控详情 API
- **新增** 网关接入服务列表端口选择
- **新增** 网关服务健康检查策略
- **新增** 网关 API 中对健康策略的支持
- **新增** Sentinel 统计相关 API
- **新增** 支持 chart 离线化的 CI 流程
- **新增** 在每日构建中增加外部镜像安全扫描能力
- **新增** 发布自动更新 chart 中镜像版本

修复

- **修复** Nacos Namespace 创建异常的问题
- **修复** Nacos 持久化存储修改异常的问题
- **修复** Nacos 生命周期管理资源校验问题
- **修复** 网关监控面板数据展示问题
- **修复** Ghippo 链接 GRPC 地址缺失问题
- **修复** Sentinel 获取集群流控 API 的问题

- **修复** **托管** Nacos 资源状态不更新的问题
- **修复** Sentinel **适配** Nacos public **字符串**问题
- **修复** Sentinel 获取资源 API **没有聚合不同**实例问题
- **修复** Sentinel **系**统规则不生效问题
- **修复** **网关服**务注册中心类型分页错误的问题
- **修复** 创建服务端口错误的问题
- **修复** **数据**库初始化组建的问题
- **修复** **使用** Helm **命令替代** Argocd **部署** Alpha 环境
- **修复** **基**础镜像 CVE 问题并升级至 3.17.2
- **修复** 发布过程 Chart **更新**问题

优化

- **优化** **升**级 gateway-api **到** v0.6.0
- **优化** **待更新**资源获取由 clusterpedia **改**为 client-go
- **优化** Sentinel 应用监控模板
- **优化** **将离**线 chart **构建** CI **步**骤独立
- **优化** Contour **升**级到 v1.24.1
- **优化** envoy **升**级到 v1.25.1
- **优化** **通**过 Chart **能力使** Skoala Init **安装**时固定命名空间

2022-12-30

v0.16.1

- **修复** 构建镜像时重复创建 builder 的问题
- **优化** Sentinel 应用监控面板细节

2022-12-29

v0.16.0

- **修复** Sentinel 调用具备认证开启 Nacos 接口的问题
- **修复** nacos-operator 频繁修改服务资源的问题
- **优化** 添加 Sentinel 服务的 Grafana 监控面板
- **优化** 升级 Insight 为最新版本支持通过集群名查询监控数据

2022-12-28

v0.15.2

新增

- **新增** 网关 API 对于认证服务器的支持
- **新增** 托管注册中心服务接入 API
- **新增** Sentinel 集群流控相关 API

修复

- **修复** Sentinel 规则拼接错误的问题
- **修复** Sentinel 仪表盘名称的问题
- **修复** 管理组件 Chart 对于生产换环境的 Service IP 问题
- **修复** Nacos 控制器处理逻辑的问题
- **修复** 与集群管理集成的 egress 地址问题

优化

- **优化** 托管 Nacos 监控仪表盘问题
- **优化** nacos-operator 数据库初始化的文件获取地址
- **优化** 更新 Sentinel 镜像至 v0.6.0

2022-12-22

v0.14.0

新增

- **新增** Init Chart 所需镜像的离线支持
- **新增** 获取托管 Nacos 的令牌

修复

- **修复** Skoala Chart 中 Values 命名问题
- **修复** CI 流程中的镜像问题

优化

- **优化** 设置默认日志输出到控制台
- **优化** 升级 nacos-operator **到社区版本**
- **优化** **更新** Nacos **自定**义资源的认证开启支持
- **优化** 设置默认组件日志级别

2022-12-21

v0.13.0

新增

- **新增** 对接中间件 MySQL **和** Redis **的相关** API
- **新增** **网关** JWT 验证支持的 API
- **新增** **网关域名校**验逻辑
- **新增** Sentinel 资源列表 API
- **新增** **网关**查询注册中心服务的接口
- **新增** **版本**发布后推送 Init Chart **至** addon 仓库
- **新增** **版本**发布时完成 gitlab release **操作**
- **新增** 动态更改日志级别

修复

- **修复** **全局限流**规则更新时不生效的问题
- **修复** Envoy Log Level **未**设置问题

- **修复** 更新网关时异常未判断的问题
- **修复** 托管 Nacos 的数据库初始化问题

优化

- **优化** 注册中心列表按更新时间降序排列
- **优化** 统一网关 JWT 相关的字段名称
- **优化** 网关域名列表增加是否开启 JWT 的字段
- **优化** Sentinel 服务名连接符的逻辑
- **优化** 升级 Contour 到 1.23 版
- **优化** 升级 Envoy 到 1.24 版
- **优化** 升级 k8s.io/相关组件到 0.25 版
- **优化** 将 go-replayers 组件回归社区版本
- **优化** 将 go-helm-client 组件回归社区版本
- **优化** 升级 Contour 到 1.23.1 版
- **优化** 修改 Agent 组件为强制不注入网格边车
- **优化** 将 Nacos 镜像默认配置回归社区版本
- **优化** 移除 Nacos 镜像相关 CI 流程

2022-12-13

v0.12.2

- **新增** 添加 Sentinel 自身监控的 Grafana 模板支持
- **新增** 添加自定义配置网关索引的配置信息

- **修复** 微服务集成可观测组件的状态问题
- **修复** 注册中心开启网格插件能力的治理状态问题
- **修复** 网关日志索引问题
- **修复** 前置依赖检查接口的问题
- **修复** Sentinel 与 Nacos 默认命名空间匹配的逻辑问题
- **修复** 连接容器管理模块的端口异常情况的逻辑

微服务引擎管理组件

本教程旨在补充需要手工 单独在线安装 微服务引擎模块的场景。下文出现的 skoala 是微服务引擎的内部开发代号，代指微服务引擎。

微服务引擎管理组件部署结构

image

image

蓝色框内的 Chart 即 skoala 组件，需要安装在控制面集群，即 DCE 5.0 的全局服务集群 kpanda-global-clsuter， 详情可参考 DCE 5.0 的部署架构。 安装 skoala 组件之后即可以在 DCE 5.0 的一级导航栏中看到微服务引擎模块。 另外需要注意：安装 skoala 之前需要安装好其依赖的 common-mysql 组件用于存储资源。

在线安装

如需安装微服务引擎，推荐通过 DCE 5.0 商业版的安装包进行安装；通过商业版可以一次性同时安装 DCE 的所有模块。

!!! note
本文提供了在线安装的方式，如果已部署了离线商业版，建议参考[离线升级微服务引擎](#_11)离线安装或升级微服务引擎。

使用商业版安装包安装

通过商业版安装微服务引擎管理组件时，需要注意商业版的版本号（点击查看最新版本）。

需要针对不同版本执行不同操作。

商业版的 版本号 ≥ v0.3.29 时，默认会安装微服务引擎管理组件，但仍旧建议检查 manifest.yaml 文件确认 components/skoala/enable 的值是否为 true，以及是否指定了 Helm 的版本。

商业版中默认安装的是经过测试的最新版本。如无特殊情况，不建议修改默认的 Helm 版本。

??? note “如果商业版 ≤ v0.3.28，点击查看对应操作”

此注释仅适用于商业版 ≤ v0.3.28；大多数情况下您的版本都会大于此版本。

执行安装命令时，默认不会安装微服务引擎。需要对照下面的配置修改 manifest.yaml 以允许安装微服务引擎。

修改文件：

```bash
./dce5-installer install-app -m /sample/manifest.yaml
```

修改后的内容：

```yaml title="manifest.yaml"
...
components:
  skoala:
    enable: true
    helmVersion: v0.12.2 # 替换为当前最新的版本号
    variables:
...
```

检测微服务引擎是否已安装

查看 skoala-system 命名空间中是否有以下对应的资源。如果没有任何资源，说明目前尚未安装微服务引擎。

```
$ kubectl get pods -n skoala-system
NAME                                  READY   STATUS    RESTARTS   AGE
hive-8548cd9b59-948j2                 2/2     Running   0          3h48m
sesame-5955c878c6-jz8cd               2/2     Running   0          3h48m
skoala-ui-75b8f8c776-nbw9d            2/2     Running   0          3h48m

$ helm list -n skoala-system
NAME    NAMESPACE       REVISION        UPDATED                                 STATUS          CHART           APP VERSION
skoala  skoala-system   2               2023-11-03 10:23:22.373053803 +0800 CST deployed        skoala-0.28.1   0.28.1
```

检测依赖的存储组件

安装微服务引擎时需要用到 common-mysql 组件来存储配置，所以要确保该组件已经存在。

此外，还需要查看 common-mysql 命名空间中是否有名为 skoala 的数据库。

```
$ kubectl get statefulset -n mcamel-system
NAME                                READY   AGE
mcamel-common-mysql-cluster-mysql   2/2     7d23h
```

建议使用如下参数为微服务引擎配置数据库信息：

- host: mcamel-common-mysql-cluster-mysql-master.mcamel-system.svc.cluster.local
- port: 3306
- database : skoala
- user: skoala
- password:

!!! note
如果未安装 common-mysql，可用自定义的数据库，上述参数按照实际情况填写即可。

检测依赖的监控组件

微服务引擎依赖 DCE 5.0 可观测性模块的能力。 如您需要监控微服务的各项指标、追踪

链路，则需要在集群中安装对应的 insight-agent， 具体说明可参考安装 insight-agent。

image

image

手动安装过程

一切就绪之后，就可以开始正式安装微服务引擎管理组件了。具体的流程如下：

!!! note
- 如果安装的是 skoala-release/skoala v0.17.1 以下的版本，则需要手动初始化数据库表。
- 如果安装的是 skoala-release/skoala v0.17.1 或更高版本，系统会自动初始化数据库表，无需手动进行。

??? note “如果初始化失败，请检查 skoala 数据库内是否有下方 3 张数据表以及对应的 SQL 是否全部生效”

```sql
mysql> desc api;
+-----------------+----------------+------+-----+-------------------+-----------------------------------------------+
| Field           | Type           | Null | Key | Default           | Extra                                         |
+-----------------+----------------+------+-----+-------------------+-----------------------------------------------+
| id              | bigint unsigned | NO   | PRI | NULL              | auto_increment                                |
| is_hosted       | tinyint        | YES  |     | 0                 |                                               |
| registry        | varchar(50)    | NO   | MUL | NULL              |                                               |
| service_name    | varchar(200)   | NO   |     | NULL              |                                               |
| nacos_namespace | varchar(200)   | NO   |     | NULL              |                                               |
| nacos_group_name | varchar(200)  | NO   |     | NULL              |                                               |
| data_type       | varchar(100)   | NO   |     | NULL              |                                               |
| detail          | mediumtext     | NO   |     | NULL              |                                               |
| deleted_at      | timestamp      | YES  |     | NULL              |                                               |
| created_at      | timestamp      | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED                             |
| updated_at      | timestamp      | NO   |     | CURRENT_TIMESTAMP | DEFAULT_
```

```
GENERATED on update CURRENT_TIMESTAMP |
+-----------------+----------------+------+-----+-----------------+-----------------------------------------------+

mysql> desc book;
+-------------+-----------------+------+-----+-----------------+-----------------------------------------------+
| Field       | Type            | Null | Key | Default         | Extra                                         |
+-------------+-----------------+------+-----+-----------------+-----------------------------------------------+
| id          | bigint unsigned | NO   | PRI | NULL            | auto_increment                                |
| uid         | varchar(32)     | YES  | UNI | NULL            |                                               |
| name        | varchar(50)     | NO   | UNI | NULL            |                                               |
| author      | varchar(32)     | NO   |     | NULL            |                                               |
| status      | int             | YES  |     | 1               |                                               |
| isPublished | tinyint unsigned | NO  |     | 1               |                                               |
| publishedAt | timestamp       | YES  |     | NULL            |                                               |
| deleted_at  | timestamp       | YES  |     | NULL            |                                               |
| createdAt   | timestamp       | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED                           |
| updatedAt   | timestamp       | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
+-------------+-----------------+------+-----+-----------------+-----------------------------------------------+
10 rows in set (0.00 sec)

mysql> desc registry;
+--------------+-----------------+------+-----+-----------------+-----------------------------------------------+
| Field        | Type            | Null | Key | Default         | Extra                                         |
+--------------+-----------------+------+-----+-----------------+-----------------------------------------------+
| id           | bigint unsigned | NO   | PRI | NULL            | auto_increment                                |
| uid          | varchar(32)     | YES  | UNI | NULL            |                                               |
| workspace_id | varchar(50)     | NO   |     | default         |                                               |
| ext_id       | varchar(50)     | YES  |     | NULL            |                                               |
```

```
| name          | varchar(50)      | NO   | MUL | NULL                  |
                                    |
| type          | varchar(50)      | NO   |     | NULL                  |
                                  |
| addresses     | varchar(1000)    | NO   |     | NULL                  |
                                 |
| namespaces    | varchar(2000)    | NO   |     | NULL                  |
                                  |
| is_hosted     | tinyint          | NO   |     | 0                     |
                                |
| deleted_at    | timestamp        | YES  |     | NULL                  |
                                  |
| created_at    | timestamp        | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GEN
ERATED                                  |
| updated_at    | timestamp        | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GEN
ERATED on update CURRENT_TIMESTAMP |
+-------------+----------------+------+-----+-------------------+-----------------------------------------------+
12 rows in set (0.00 sec)
```

配置 skoala helm repo

配置好 Skoala 仓库，即可查看和获取到 Skoala 的应用 Chart：

```
helm repo add skoala-release https: //release.daocloud.io/chartrepo/skoala
helm repo update
```

需要事先安装 Helm

Skoala 是微服务引擎的控制端的服务：

- 安装完成后，可以在 DCE 5.0 平台看到微服务引擎的入口
- 包含 3 个组件：skoala-ui、hive、sesame
- 需要安装在全局服务集群

默认情况下，安装完成 skoala 到 kpanda-global-cluster（全局服务集群），就可以在侧边栏看到对应的微服务引擎的入口了。

查看微服务引擎管理组件最新版本

在全局服务集群中，查看 Skoala 的最新版本，直接通过 Helm 命令获取版本信息；

```
$ helm repo update skoala-release
$ helm search repo skoala-release/skoala --versions
NAME                        CHART VERSION    APP VERSION  DESCRIPTION
skoala-release/skoala       0.28.1           0.28.1       The helm chart for Skoala
skoala-release/skoala       0.28.0           0.28.0       The helm chart for Skoala
skoala-release/skoala       0.27.2           0.27.2       The helm chart for Skoala
skoala-release/skoala       0.27.1           0.27.1       The helm chart for Skoala
......
```

执行部署（同样适用于升级）

执行以下命令，注意对应的版本号：

```
helm upgrade --install skoala --create-namespace -n skoala-system --cleanup-on-fail \
    --set ui.image.tag=v0.19.0 \
    --set hive.configMap.database[0].driver="mysql" \
    --set hive.configMap.database[0].dsn="skoala:xxx@tcp(mcamel-common-mysql-cluster-mysql-master.mcamel-system.svc.cluster.local:3306)/skoala?charset=utf8&parseTime=true&loc=Local&timeout=10s" \
    skoala-release/skoala \
    --version 0.28.1
```

查看 Pod 是否启动成功：

```
$ kubectl get pods -n skoala-system
NAME                                READY   STATUS    RESTARTS        AGE
hive-8548cd9b59-948j2               2/2     Running   0               3h48m
sesame-5955c878c6-jz8cd             2/2     Running   0               3h48m
skoala-ui-7c9f5b7b67-9rpzc          2/2     Running   0               3h48m
```

在线升级

开始升级操作之前，了解一下微服务引擎的部署架构有助于更好地理解后续的升级过程。

微服务引擎由两个组件构成：

- skoala 组件安装在控制面集群，用于在 DCE 5.0 的一级导航栏中加载微服务引擎模块

- skoala-init 组件安装在工作集群，用于提供微服务引擎的核心功能，例如创建注册中心、网关实例等

!!! note
- 升级微服务引擎时需要同时升级这两个组件，否则会存在版本不兼容的情况。
- 有关微服务引擎的版本更新情况，可参考 [release notes](../intro/release-notes.md)。

由于 skoala 组件安装在控制面集群中，所以需要在控制面集群中执行下列操作。

1. 执行如下命令备份原有数据

   ```
   helm get values skoala -n skoala-system -o yaml > skoala.yaml
   ```

2. 添加微服务引擎的 Helm 仓库

   ```
   helm repo add skoala https://release.daocloud.io/chartrepo/skoala
   ```

3. 更新微服务引擎的 Helm 仓库

   ```
   helm repo update
   ```

4. 执行 helm upgrade 命令

   ```
   helm upgrade --install --create-namespace -n skoala-system skoala skoala/skoala --version=0.28.1 --set hive.image.tag=v0.28.1 --set sesame.image.tag=v0.28.1 --set ui.image.tag=v0.19.0 -f skoala.yaml
   ```

   需要将 version、hive.image.tag、sesame.image.tag、ui.image.tag 四个参数的值调整为您需要升级到的微服务引擎的版本号。

离线升级

DCE 5.0 的各个模块松散耦合，支持独立安装、升级各个模块。此文档适用于通过离线方式安装微服务引擎之后进行的升级。

下载离线包

下载微服务引擎模块离线包并解压。

```
tar -vxf skoala_x.y.z_amd64.tar
```

解压完成后会得到四个压缩包:

- skoala-x.y.z.tgz

- skoala-init-x.y.z.tgz
- skoala_x.y.z.bundle.tar
- skoala-init_x.y.z.bundle.tar

同步镜像

将镜像下载到本地节点之后，需要通过 charts-syncer 或容器运行时将最新版镜像同步到您的镜像仓库。推荐使用 charts-syncer 同步镜像，因为该方法更加高效便捷。

charts-syncer 同步镜像

1. 使用如下内容创建 load-image.yaml 作为 charts-syncer 的配置文件

   load-image.yaml 文件中的各项参数均为必填项。您需要一个私有的镜像仓库，并参考如下说明修改各项配置。 有关 charts-syncer 配置文件的详细解释，可参考其官方文档。

   === “已安装 HARBOR chart repo”

   若当前环境已安装 HARBOR chart repo，charts-syncer 也支持将 chart 导出为 tgz 文件。

   ```yaml title="load-image.yaml"
   source:
     intermediateBundlesPath: skoala-offline # (1)
   target:
     containerPrefixRegistry: 10.16.10.111 # (2)
     repo:
       kind: HARBOR # (3)
       url: http://10.16.10.111/chartrepo/release.daocloud.io # (4)
       auth:
         username: "admin" # (5)
         password: "Harbor12345" # (6)
     containers:
       auth:
         username: "admin" # (5)
         password: "Harbor12345" # (6)
   ```

1. 到执行 charts-syncer 命令的相对路径，而不是此 YAML 文件和离线包之间的相对路径
2. 需更改为你的镜像仓库 url
3. 也可以是任何其他支持的 Helm Chart 仓库类别
4. 需更改为 chart repo project url
5. 你的镜像仓库用户名
6. 你的镜像仓库密码

=== “已安装 CHARTMUSEUM chart repo”

若当前环境已安装 CHARTMUSEUM chart repo，charts-syncer 也支持将 chart 导出为 tgz 文件。

```yaml title="load-image.yaml"
source:
  intermediateBundlesPath: skoala-offline # (1)
target:
  containerPrefixRegistry: 10.16.10.111 # (2)
  repo:
    kind: CHARTMUSEUM # (3)
    url: http://10.16.10.111 # (4)
    auth:
      username: "rootuser" # (5)
      password: "rootpass123" # (6)
  containers:
    auth:
      username: "rootuser" # (7)
      password: "rootpass123" # (6)
```

1. 到执行 charts-syncer 命令的相对路径，而不是此 YAML 文件和离线包之间的相对路径
2. 需更改为你的镜像仓库 url
3. 也可以是任何其他支持的 Helm Chart 仓库类别
4. 需更改为 chart repo url
5. 你的镜像仓库用户名，如果 chartmuseum 没有开启登录验证，就不需要填写 auth
6. 你的镜像仓库密码
7. 你的镜像仓库用户名

=== “未安装 chart repo”

若当前环境未安装 chart repo，charts-syncer 也支持将 chart 导出为 `tgz` 文件并存放在指定路径。

```yaml title="load-image.yaml"
source:
```

```
    intermediateBundlesPath: skoala-offline # (1)
  target:
    containerRegistry: 10.16.23.145 # (2)
    containerRepository: release.daocloud.io/skoala # (3)
    repo:
      kind: LOCAL
      path: ./local-repo # (4)
    containers:
      auth:
        username: "admin" # (5)
        password: "Harbor12345" # (6)
```

1. 到执行 charts-syncer 命令的相对路径，而不是此 YAML 文件和离线包之间的相对路径
2. 需更改为你的镜像仓库 url
3. 需更改为你的镜像仓库
4. chart 本地路径
5. 你的镜像仓库用户名
6. 你的镜像仓库密码

2. 将 skoala_x.y.z.bundle.tar 和 skoala-init_x.y.z.bundle.tar 放到 skoala-offline 文件夹下。

3. 执行同步镜像命令。

```
charts-syncer sync --config load-image.yaml --insecure true
```

Docker 或 containerd 同步镜像

1. 解压 tar 压缩包。

```
tar -vxf skoala_x.y.z.bundle.tar
```

解压成功后会得到 3 个文件：

- hints.yaml
- images.tar
- original-chart

2. 从本地加载镜像到 Docker 或 containerd。

=== “Docker”

```shell
docker load -i images.tar
```

=== “containerd”

```shell
ctr -n k8s.io image import images.tar
```

!!! note
- 需要在每个节点上都通过 Docker 或 containerd 加载镜像。
- 加载完成后需要 tag 镜像，保持 Registry、Repository 与安装时一致。

开始升级

镜像同步完成之后，**就可以开始升**级微服务引擎了。

=== "**通**过 helm repo **升**级"

1. 检查微服务引擎 Helm 仓库是否存在。

   ```shell
   helm repo list | grep skoala
   ```

   若返回结果为空或如下提示，则进行下一步；反之则跳过下一步。

   ```none
   Error: no repositories to show
   ```

2. 添加微服务引擎的 Helm 仓库。

   ```shell
   helm repo add skoala-release http://{harbor url}/chartrepo/{project}
   ```

3. 更新微服务引擎的 Helm 仓库。

   ```shell
   helm repo update skoala-release # (1)
   ```

   1. Helm 版本过低会导致失败，若失败，请尝试执行 `helm repo update`

4. 选择您想安装的微服务引擎版本（建议安装最新版本）。

   ```shell
   helm search repo skoala-release/skoala --versions
   ```

```text
NAME                        CHART VERSION   APP VERSION   DESCRIPTION
skoala-release/skoala   0.28.1          v0.28.1       A Helm chart for Skoala
...
```

5. 备份 `--set` 参数。

   在升级微服务引擎版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

   ```shell
   helm get values skoala -n skoala-system -o yaml > bak.yaml
   ```

6. 执行 `helm upgrade`。

   升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 字段为当前使用的镜像仓库地址。

   ```shell
   export imageRegistry={你的镜像仓库}
   ```

   ```shell
   helm upgrade skoala skoala-release/skoala \
   -n skoala-system \
   -f ./bak.yaml \
   --set global.imageRegistry=$imageRegistry \
   --version 0.28.1
   ```

=== "通过 Chart 包升级"

1. 准备好 `original-chart`（解压 `skoala_x.y.z.bundle.tar` 得到）。

2. 备份 `--set` 参数。

   在升级微服务引擎版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

   ```shell
   helm get values skoala -n skoala-system -o yaml > bak.yaml
   ```

3. 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 为当前使用的镜像仓库地址。

```shell
export imageRegistry={你的镜像仓库}
```

```shell
helm upgrade skoala original-chart \
-n skoala-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry
```

卸载微服务引擎管理组件

helm uninstall skoala -n skoala-system

微服务引擎集群初始化组件

本教程旨在补充需要手工 **单独在线安装 微服**务引擎集群初始化组件的的场景。**下文出**现**的** skoala-init **是微服**务引擎集群初始化组件的内部开发代号，**代指微服**务引擎集群初始化组件。

微服务引擎集群初始化组件部署结构：

image

image

蓝色框内的 chart **即** skoala-init 组件，**需要安装在工作集群**。**安装** skoala-init 组件之后即**可以使用微服**务引擎的各项功能，**例如**创建注册中心、**网关**实**例等**。**另外需要注意**，skoala-init 组件依赖 DCE 5.0 **可**观测模块的 insight-agent 组件提供指标监控和链路追踪等**功能**。 **如您需要使用**该项功能，则需要事先安装好 insight-agent 组件， **具体步**骤可参考安装组件 insight-agent。

!!! note

- 如果安装 `skoala-init` 之前没有事先安装 `insight-agent`，则不会安装 `service-monitor`。
- 如果需要安装 `service-monitor`，请先安装 `insight-agent`，然后再安装 `skoala-init`。

在线安装

skoala-init 是微服务引擎所有的组件 Operator：

- 仅安装到工作集群即可
- 包含组件有：skoala-agent、nacos-operator、sentinel-operator、seata-operator、contour-provisioner、gateway-api-adminssion-server
- 未安装时，创建注册中心和网关时会提示缺少组件

由于 Skoala 涉及的组件较多，我们将这些组件打包到同一个 Chart 内，也就是 skoala-init，所以我们应该在用到微服务引擎的工作集群安装好 skoala-init。此安装命令也可用于更新该组件。

配置好 Skoala 仓库，即可查看和获取到 skoala-init 的应用 chart。

```
helm repo add skoala-release https: //release.daocloud.io/chartrepo/skoala
helm repo update
$ helm search repo skoala-release/skoala-init --versions
NAME                          CHART VERSION   APP VERSION DESCRIPTION
skoala-release/skoala-init    0.28.1          0.28.1      A Helm Chart for Skoala init, it includ
es Skoal...
skoala-release/skoala-init    0.28.0          0.28.0      A Helm Chart for Skoala init, it includ
es Skoal...
skoala-release/skoala-init    0.27.2          0.27.2      A Helm Chart for Skoala init, it includ
es Skoal...
skoala-release/skoala-init    0.27.1          0.27.1      A Helm Chart for Skoala init, it includ
es Skoal...
......
```

执行以下命令，注意对应的版本号：

```
helm upgrade --install skoala-init --create-namespace -n skoala-system --cleanup-on-fail \
    skoala-release/skoala-init \
    --version 0.28.1
```

查看 Pod 是否启动成功：

```
$ kubectl get pods -n skoala-system
NAME                                        READY   STATUS      RESTARTS   A
GE
contour-provisioner-54b55958b7-5ltng        1/1     Running     0          2d6
h
gateway-api-admission-patch-bk7c8           0/1     Completed   0          2d
6h
gateway-api-admission-pwhdh                 0/1     Completed   0          2
d6h
gateway-api-admission-server-77545d74c4-v6fpr   1/1     Running     0          2d6
h
nacos-operator-6d94bdccc8-wx4w5             1/1     Running     0
2d6h
seata-operator-f556d989d-8qrf8              1/1     Running     0          2d
6h
sentinel-operator-6fb9dc98f4-d44k5          1/1     Running     0          2d6
h
skoala-agent-54d4df7897-7p4pz               1/1     Running     0          2
d6h
```

除了通过终端安装，也可以在 容器管理-> Helm 应用 内找到 skoala-init 进行安装。

image

image

在线升级

由于 skoala-init 组件安装在工作集群中，所以需要在每个工作集群中各执行一次下列操作。

1. 备份原有参数

   helm get values skoala-init -n skoala-system -o yaml > skoala-init.yaml

2. 添加微服务引擎的 Helm 仓库

   helm repo add skoala https://release.daocloud.io/chartrepo/skoala

3. 添加微服务引擎的 Helm 仓库

   helm repo update

4. 删除 gateway-api 相关 job

   kubectl delete jobs gateway-api-admission gateway-api-admission-patch -n skoala-system

5. 执行 helm upgrade 命令

```
helm upgrade --install --create-namespace -n skoala-system skoala-init skoala/skoala-init --version=0.28.1 --set nacos-operator.image.tag=v0.28.1 --set skoala-agent.image.tag=v0.28.1 --set sentinel-operator.image.tag=v0.28.1 --set seata-operator.image.tag=v0.28.1 -f skoala-init.yaml
```

!!! note
需要将 `version`、`nacos-operator.image.tag`、`skoala-agent.image.tag`、`sentinel-operator.image.tag`、`seata-operator.image.tag` 五个参数的值调整为您需要升级到的微服务引擎的版本号。

6. **根据自身需要，手**动更新需要升级的网关相关 CRD **文件**。

```
# projectcontour 相关 crd
kubectl apply -f skoala-init/charts/contour-provisioner/crds/contour.yaml
# gateway-api 相关 crd
kubectl apply -f skoala-init/charts/contour-provisioner-prereq/crds/gateway-api.yaml
```

离线升级

参考微服务引擎管理组件的离线升级**方式**

卸载微服务引擎集群初始化组件

!!! note
- nacos sentinel seata 的 crd 会随之卸载，特别注意，相关 cr 会被删除。
- 网关相关 crd 不会被随之卸载，如需清除需手动处理，特别注意，清除 crd 时相关 cr 会被删除。

??? note "网关相关 crd 清单"

contourconfigurations.projectcontour.io
contourdeployments.projectcontour.io
extensionservices.projectcontour.io
gatewayclasses.gateway.networking.k8s.io
gateways.gateway.networking.k8s.io
grpcroutes.gateway.networking.k8s.io
httpproxies.projectcontour.io
httproutes.gateway.networking.k8s.io
referencegrants.gateway.networking.k8s.io
tcproutes.gateway.networking.k8s.io
tlscertificatedelegations.projectcontour.io
tlsroutes.gateway.networking.k8s.io
udproutes.gateway.networking.k8s.io

```
helm uninstall skoala-init -n skoala-system
```

手动清理 gateway-api 相关 job

```
kubectl delete jobs gateway-api-admission gateway-api-admission-patch -n skoala-system
```

手动清理网关相关 crd

```
kubectl delete crds `kubectl get crds | grep -E "projectcontour.io|gateway.networking.k8s.io" | awk '{print $1}'`
```

选择工作空间

首次进入微服务治理模块时，首先必须选择一个工作空间。

选择工作空间

选择工作空间

!!! note
如果当前没有可选的工作空间，需要先联系超级管理员创建一个工作空间。

如需更改当前所在的工作空间，可以在左侧边栏点击更换图标重新选择工作空间。

更改工作空间

更改工作空间

服务治理

开始治理云原生微服务之前，首先需要将导入服务。 目前仅支持 DCE 5.0 服务网格模块

中网格下的服务导入。

导入服务

支持

1. 进入 **微服务引擎** 模块，在左侧点击 **云原生微服务** -> **服务治理** ，然后在右上角点击 **手工导入** 。

手工导入

手工导入

2. 筛选目标服务所在的服务网格和命名空间，勾选该服务，然后点击 **确定** 即可。

确定

确定

查看服务

查看已经导入的所有云原生微服务，点击服务名称进一步查看服务暴露的端口和协议。

查看

查看

移除服务

在右侧操作栏移除不需要的微服务。

移除

移除

服务治理指基于 Service Mesh 对 DCE 5.0 服务网格中的服务进行东西向流量治理。

将网格服务导入云原生微服务引擎之后，就可以针对服务暴露的不同端口设置不同的东西向流量策略。

1. 点击服务名称

点击某个名称

点击某个名称

2. 进入治理能力页面

治理能力

治理能力

3. 根据需要在端口上创建规则，最后点击 确定

全链路流量泳道

微服务引擎允许将应用程序的不同版本或特性隔离到各自独立的运行环境，也就是所谓的“泳道”。 随后，通过定义泳道规则，可以将符合条件的请求流量引导到目标版本或特性的应用程序上。 本文介绍流量泳道的概念、使用场景。

功能介绍

灰度发布会根据请求内容或者请求流量的比例将线上流量的一小部分转发至服务的新版本，待灰度验证通过后，逐步调大新版本的请求流量，是一种循序渐进的发布方式。

当服务之间存在调用链路时，对服务的灰度发布往往不局限于单个服务，而是需要对服务的整条请求链路进行环境隔离与流量控制， 即保证灰度流量只发往调用链路中服务的灰度版本，实现调用链路之间相互隔离的隔离环境。

具有相同版本（或其他特征）的不同服务组成的一个调用链路隔离环境称为泳道。 通过使用流量泳道功能，您仅需制定少量的治理规则，便可构建从网关到整个后端服务的多个流量隔离环境， 有效保障多个服务顺利安全发布以及服务多版本并行开发，进一步促进业务的快速发展。

微服务网关

微服务网关支持多租户实例的高可用架构，兼容多种模式微服务的统一网关接入能力。本页介绍如何管理微服务网关实例。

创建微服务网关

创建微服务网关的步骤如下：

1. 在左侧导航栏点击 **云原生网关** ，然后在右上角点击 **创建网关** 。

   进入创建页面

进入创建页面

2. 参考以下说明填写基本信息

   - **网关名称**：长度不超过 63 个字符，支持字母、数字、连字符并且必须以字母或数字字符开头及结尾。名称在网关创建完成后不可更改。
   - **部署集群**：选择将网关部署在哪个集群。

     如果可选列表中没有出现目标集群，可以去容器管理模块中接入或创建集群并通过全局管理模块将该集群或其中的某个命名空间绑定到当前工作空间。
   - **命名空间**：选择将网关部署在哪个命名空间。一个命名空间中只能部署一个网关。
   - **安装**环境检测：选择集群和命名空间后，系统会自动检测安装环境。未通过检测时，页面会给出原因和操作建议，根据页面提示操作即可。
   - **管辖命名空间**：设置新建的网关可以管辖哪些命名空间。默认管辖网关所在

的命名空间。支持同时管辖多个命名空间。同一个命名空间不能被两个网关同时管辖。

填写基本配置

填写基本配置

3. 参考以下说明填写配置信息

=== “服务配置”

- 集群内部访问：只能在同一集群的内部访问服务。

  ![填写基本配置](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/images/gw-create03.png)

- 节点访问：通过节点的 IP 和静态端口访问服务，支持从集群外部访问服务。

  外部流量策略：`Cluster` 指流量可以转发到集群中其他节点上的 Pod；`Local`指流量只能转发到本节点上的 Pod。

  ![填写基本配置](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/images/gw-create04.png)

- 负载均衡器：使用云服务提供商的负载均衡器使得服务可以公开访问

  - 外部流量策略：`Cluster` 指流量可以转发到集群中其他节点上的 Pod；`Local` 指流量只能转发到本节点上的 Pod
  - 负载均衡类型：MetalLB 或其他
  - MetalLB IP 池：支持自动选择或指定 IP 池子
  - 负载均衡器 IP 地址：支持自动选择或指定 IP

  ![填写基本配置](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/images/gw-create05.png)

=== “资源配置”

为当前网关配置多少控制节点和工作节点。单副本存在不稳定性，需谨慎选择。

![填写基本配置](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/images/gw-create06.png)

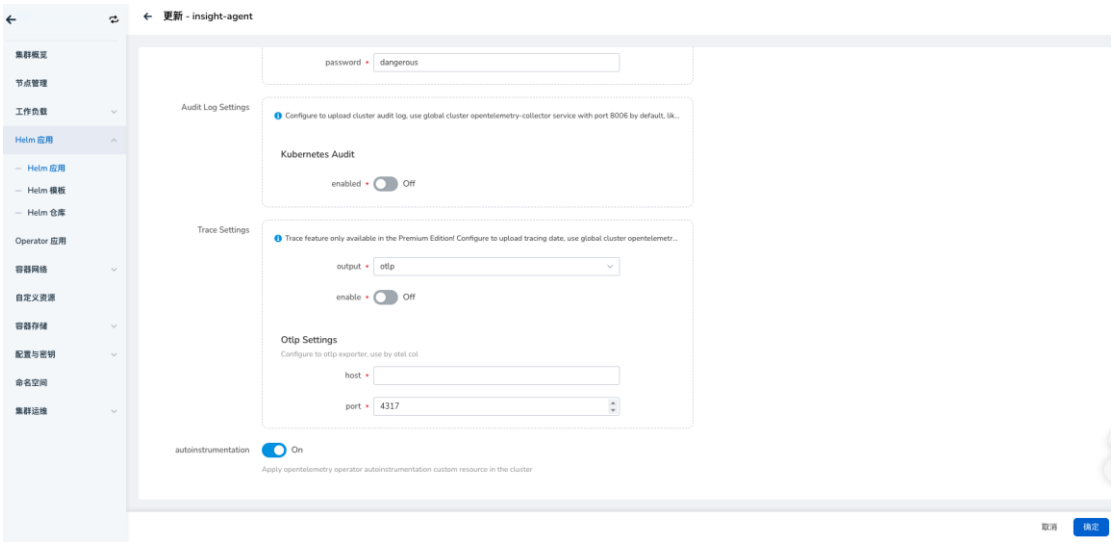
=== “高级配置”

- 控制节点：设置为控制节点（contour）配置多少 CPU 和内存资源
- 工作节点：设置为工作节点（envoy）配置多少 CPU 和内存资源

![填写高级配置](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/images/gw-create07.png)

4. 参考以下信息填写高级配置

   - **日志配置**：设置工作节点 （envoy）的日志级别和 Pod 的日志级别
   - **更新机制**：重新创建指删除原来的网关并新建一个网关，滚动更新指不删除网关，而是滚动式地更新网关相关的 Pod
   - **网关前置代理层数**：请求从客户端到网关中途需要经过几个代理端点。需要根据实际情况填写。例如客户端-Nginx-网关的代理层数为 1，因为中间只经过 1 个 Nginx 代理端点。
   - **网关链路**：需在 insight-agent 中启用 autoinstrumentation，同时在网关实例中开启网关链路配置，可以根据通过网关进行的请求生成链路信息并发送给可观测模块进行数据采集。



insight-agent

填写高级配置

填写高级配置

5. 参考以下信息填写插件配置（选填），最后在页面右下角点击 **确定** 即可。

选择是否启用全局限流插件。

需要事先在插件中心接入插件，或在当前页面点击蓝色文字跳转到相应页面接入创建。

填写高级配置

填写高级配置

!!! note
系统会自动返回云原生网关列表页面，在右侧可以执行[更新网关](../gateway/index.md#_7)或[删除网关](../gateway/index.md#_8)的操作。

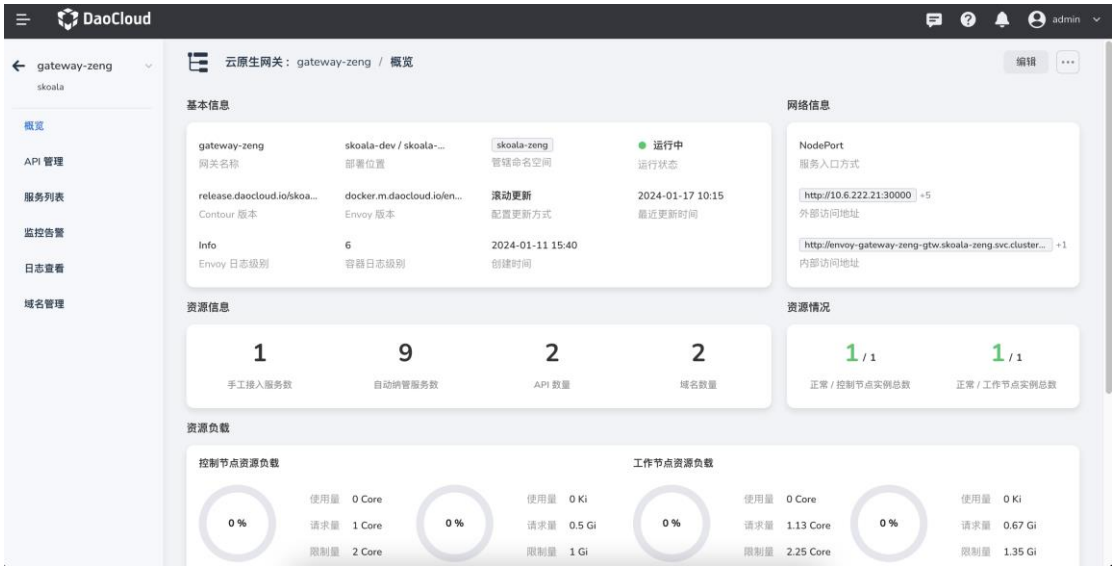
![确认所填信息](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/images/gw-create10.png)

查看微服务网关

可以在 **概览** 页面查看网关的详细信息，包括名称、部署位置、网关运行状态、服务入口方式、控制节点/工作节点的健康状态、API、插件等信息。

进入网关详情页面

在 **网关列表** 页面，选择目标网关的名称，即可进入网关概览页面。



概览页面

网关详情

网关详情分为：基本信息、网络信息、TOP 10 热门 API 情况、资源情况、资源负载、插件信息等部分。

部分数据的说明如下：

更新微服务网关

微服务网关支持多租户实例的高可用架构，兼容多种模式微服务的统一网关接入能力。

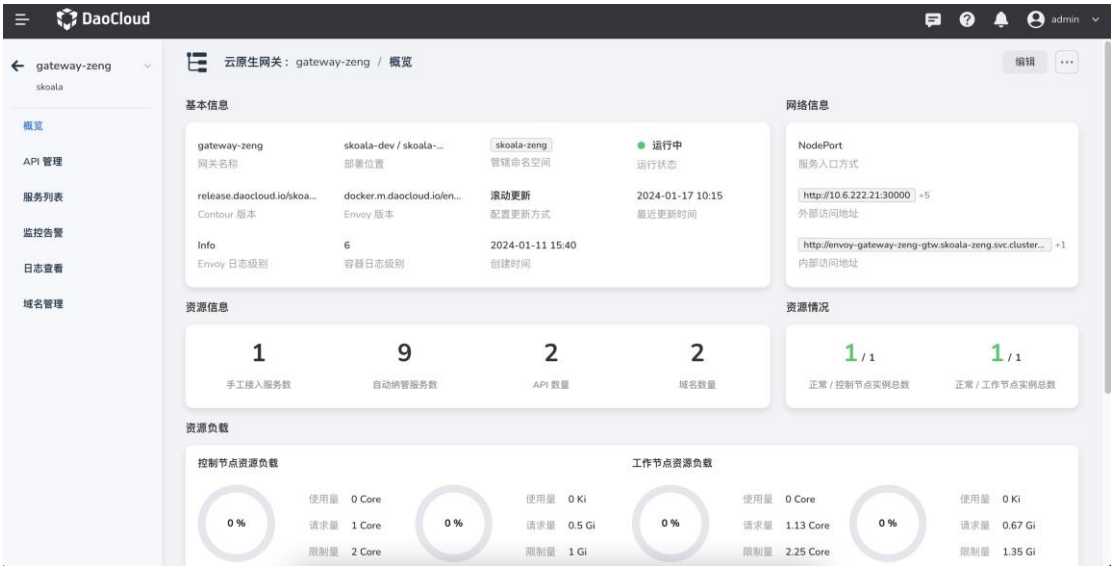
有两种方式可以更新网关配置。

- 在 微服务网关列表 页选择需要更新的网关实例，在实例右侧点击 ... 并选择 编辑 。

更新网关

更新网关

- 点击网关名称进入进入概览页面后，在右上角点击 编辑 。



更新网关

删除微服务网关

微服务网关支持多租户实例的高可用架构，兼容多种模式微服务的统一网关接入能力。

删除网关同样也有两种方式。为保证服务不受影响，删除网关之前需要释放到网关中全部路由的 API。

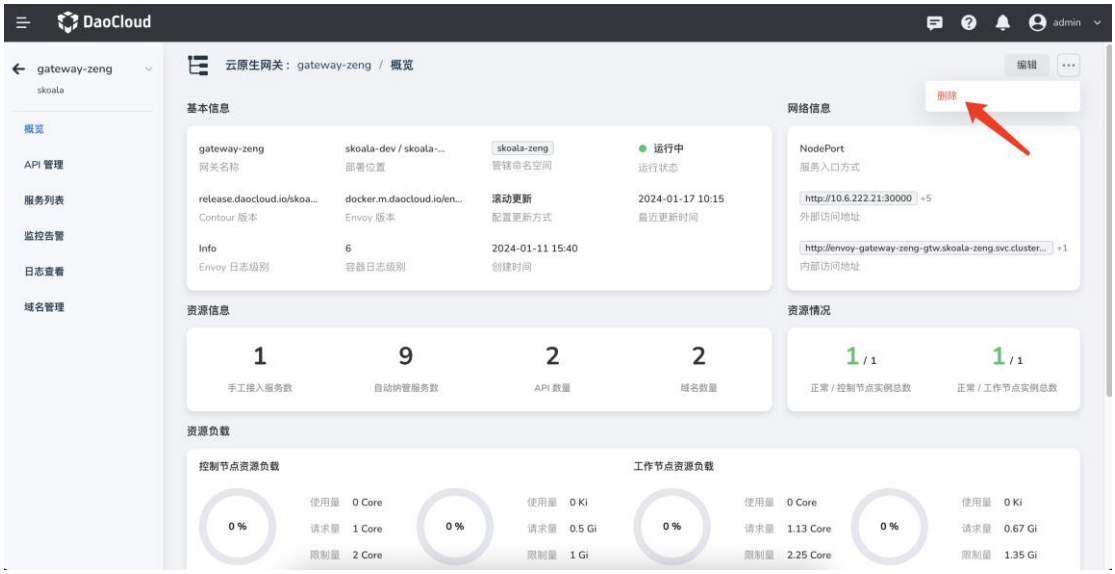
!!! danger
网关删除后不可恢复，请谨慎操作。

- 在 微服务网关列表 页选择需要移除的网关实例，在实例右侧点击 … 并选择 删除 。

移除网关

移除网关

- 点击网关名称进入进入概览页面后，在右上角 … 并选择 删除 。



移除网关

域名管理

微服务网关支持对统一托管的域名进行全生命周期管理，包括添加、更新、删除域名域名。通过域名管理，可以将一个域名应用到网关内的多个 API，并且可以配置域名层级的网关策略。

添加域名

具体操作步骤如下：

1. 点击目标网关的名称进入网关概览页面，然后在左侧导航栏点击 **域名管理** ，在页面右上角点击 **添加域名** 。

   添加域名

   添加域名

2. 填写配置信息

   域名的配置信息分为基础信息（必填）和策略配置（可选）和安全配置（可选）三部分。

   - 域名：域名创建后不可以修改。
   - 协议：默认选择 HTTP。如果选择 HTTPS，需要提供对应的 HTTPS 证书。

     目前仅支持选取已经存在的证书，自动签发证书和手动上传证书功能正在开发。

     https

     https
   - 本地限流：参考本地限流
   - 跨域：参考跨域

填写配置

填写配置

3. 在页面右下角点击 **确定**

   点击 **确定** 后，将自动跳转到 **域名管理** 页面，可以在域名列表中看到刚才新建的域名。

添加成功

添加成功

更新域名

可以通过两种方式修改域名的基础信息和策略配置。

- 在 **域名管理** 页面找到需要更新的域名，在右侧点击 □ 选择 **修改基础信息** 或 **修改策略配置** 或 **修改安全配置** 。

在列表页更新基础信息

在列表页更新基础信息

- 点击域名名称进入域名详情页，在页面右上角点击 **修改基础配置** 更新基本信息，点击 **修改策略配置** 更新策略，点击 **修改安全配置** 更新安全配置。

在详情页更新

在详情页更新

删除域名

!!! danger
- 正在被 API 使用的域名无法删除，需要先删除相关的 API 然后才能删除域名。
- 域名删除后无法恢复。

可以通过两种方式删除域名。

- 在 域名管理 页面找到需要删除的域名，在点击 □ 并选择 删除 。

  在列表页删除

  在列表页删除

- 点击域名名称进入域名的详情页，在页面右上角点击 □ 操并选择 删除 。

  在详情页删除

  在详情页删除

  如果域名正在被某个 API 使用，需要页面提示点击 查看服务详情 去删除对应的 API。

  在详情页删除

  在详情页删除

配置域名策略

微服务网关提供了域名级的策略配置能力。配置域名级的策略之后，无需对同一域名下的多个 API 重复配置策略。 目前支持四种域名级的策略：跨域、本地限流、全局限流、访问黑白名单。

有两种方式可以配置域名策略：

- 在创建域名的过程中设置策略，参考添加域名。
- 在域名创建完成后通过修改域名进行调整。

对跨域和本地限流策略的详细说明如下：

本地限流

为域名配置本地限流之后，该配置会自动应用到使用此域名的所有 API。

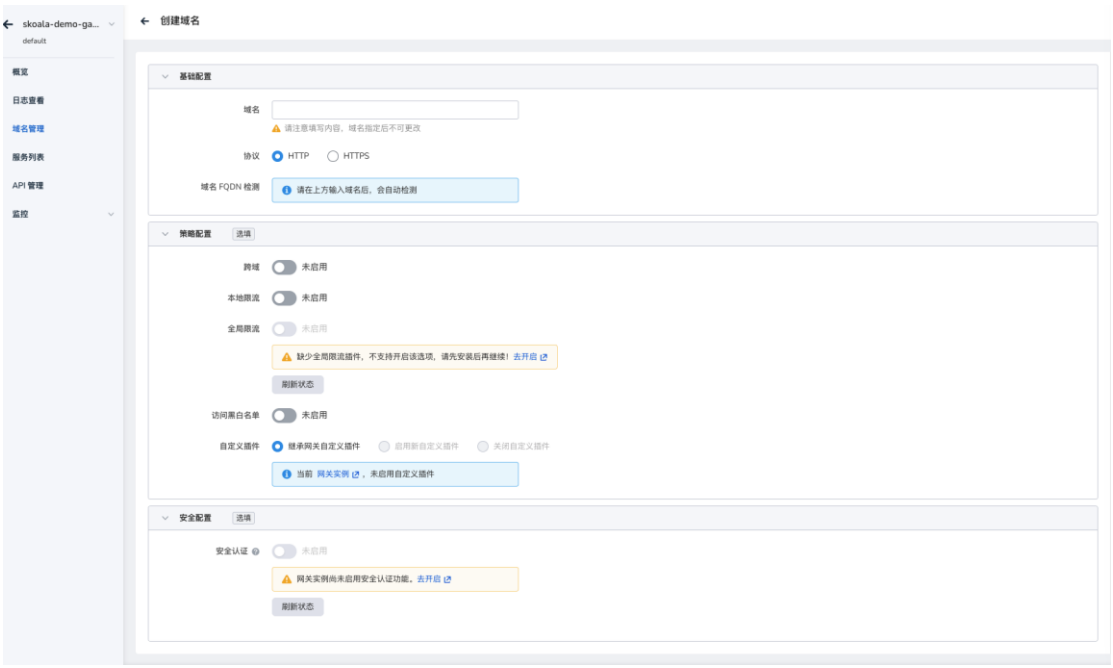
有关本地限流的详细配置说明，可参考本地限流。

!!! note
如果 API 层级与域名层级的限流策略冲突时，以 API 层级的限流策略为准。

跨域

填写配置时需要注意：

- 启用凭证：开启后，需要对跨域请求进行凭证检查。检查通过之后，才能开始处理跨域请求。
- 允许的请求方法：选择 HTTP 协议的请求方式。有关各种请求方式的详细说明，可参考 W3C 的官方文档方式定义。
- 允许的请求来源：限定多个特定的请求来源，通常使用 IP。
- 预检时长：在处理跨域请求之前检查凭证、请求方法等事项所用的时间，时间单位为秒、分、时。
- 允许的请求头：限定特定的 HTTP 请求头关键字。添加关键字后，需在请求头中加上对应的关键字才能正常访问目标服务。
- 暴露的请求头：控制暴露的请求头关键字，可以配置多项。



跨域

API 管理

微服务网关支持对网关实例的 API 进行全生命周期管理，包括 API 的添加、更新和删除。

添加 API

前提条件

- 有可选的域名，可参考域名管理创建域名。
- 如果 API 的目标服务为后端服务，则需要确保有可选的后端服务，可参考通过手动或自动方式接入服务。

创建 API 的步骤如下：

1. 点击网关名称进入网关概览页面，然后在左侧导航栏点击 **API 管理** ，在页面右上角点击 **添加 API** 。

进入添加页面

进入添加页面

2. **参考下方**说明填写基本配置。

   配置分为基本配置和策略配置和安全配置三部分。**填写基本配置信息**时需要注意：

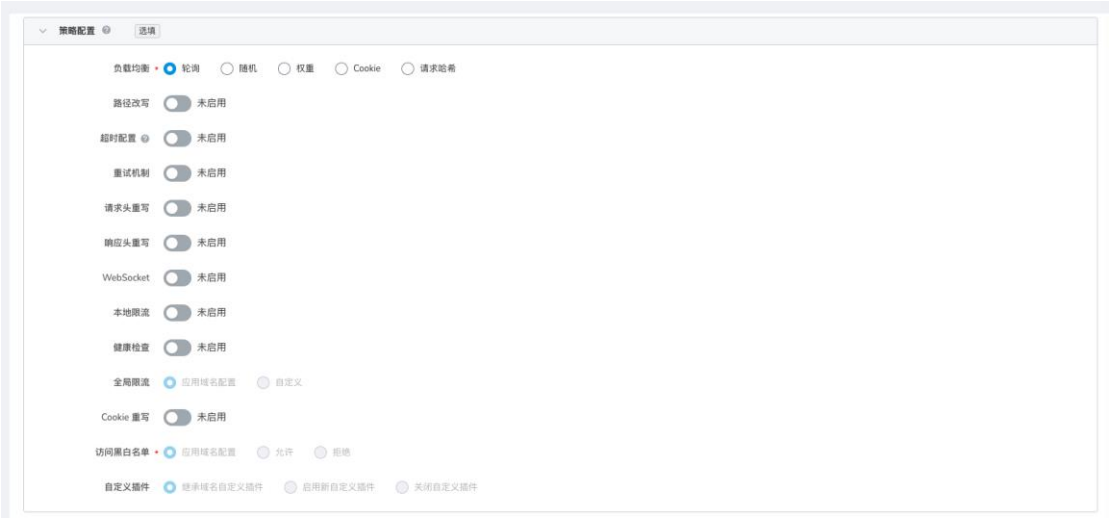
   - API **名称**：**包含小写字母、数字和以及特殊字符(- .)，不能以特殊字符开**头和结尾。
   - API **分**组：选择 API **所属的分**组组名。**如果**输入不存在的分组名称，则自动创建一个新的分组。
   - **关**联域名：**填写关**联域名后，**可以通**过域名 + **端口号的方式**访问 API。**找不到域名**时可以添加新域名，**可参考**添加域名。
   - **匹配**规则：**只允**许符合该规则的请求通过。**如果**设置了多条规则，**需要同**时满足所有规则才能放行。**如果添加了**请求头，则需要在访问该 API 时添加**相**应的请求头。
   - 请求方法：选择 HTTP 协议的请求方式。**有关各种**请求方式的详细说明，**可参考** W3C **的官方文档**方式定义。
   - **目**标服务：选择将请求直接发送到后端服务，**或者重定向到其他服**务，**或者直接返回** HTTP **状**态码。
   - **如果**选择后端服务，则需要配置权重。权重越大，**网关向其分**发的流量就越**多**。

配置信息

配置信息

3. **参考下方**说明填写策略配置（选填）。

支持 12 种 API 策略：负载均衡、路径改写、超时配置、重试机制、请求头重写、响应头重写、Websocket、本地限流、健康检查、cookie 重写、全局限流、访问黑白名单。有关各项策略的配置说明，可参考 API 策略配置。



配置策略

4. 参考下方说明填写安全配置（选填）。

   - JWT 认证：应用域名配置或不启用
   - 安全认证：应用域名配置或自定义

安全配置

安全配置

5. 在页面右下角点击 保存 （不上线）。如果点击 保存并上线 则可以直接上线 API。

   点击 确定 后，如果所有配置都正常，右上角会弹出 创建网关 API 成功 的提示信息。

   可以在 API 管理 页面查看新建的 API。

创建成功

创建成功

6. API 上线

   API 创建成功后，默认处于下线状态，此时无法访问。需要将 API 调整为 上线 ，才

能正常访问。API 上线有两种方式。

- 在 API 在 **API 管理** 页面找到需要更新的 API，在该 API 的右侧点击 □ 选择 **API 上线** 。

API 上线

API 上线

- 点击 API 名称进入 API 详情页，在页面右上角点击 □ 并选择 **API 上线** 。

API 上线

API 上线

!!! info
点击 API 名称进入 API 详情，可查看 API 的详细配置信息，例如上下线状态、域名、匹配规则、目标服务、策略配置等。

![API 上线](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/gateway/api/images/online1.png)

更新 API

可以通过两种方式更新 API 的基础配置、策略配置和安全配置。

- 在 **API 管理** 页面找到需要更新的 API，在该 API 的右侧点击 □ 选择 **修改基础配置** 、 **修改策略配置** 或 **修改安全配置** 。

在列表页更新基础信息

在列表页更新基础信息

- 点击 API 名称进入 API 详情页，在页面右上角 **修改基础配置** 、 **修改策略配置** 或 **修改安全配置** 。

在详情页更新

在详情页更新

删除 API

微服务网关支持对网关实例的 API 进行全生命周期管理，包括 API 的添加、更新和移除。

可以通过两种方式移除 API。

!!! danger
删除操作是不可逆的。无论 API 是否处于在线状态，删除后均立即失效并且不可恢复。

- 在 API 管理 页面找到需要删除的 API，在该 API 的右侧点击 □ 并选择 移除 。

  在列表页删除

  在列表页删除

- 点击 API 名称进入 API 详情页，在页面右上角点击 □ 操并选择 移除 。

  在详情页删除

  在详情页删除

配置 API 策略

DCE 5.0 微服务网关支持十二种 API 策略：负载均衡、路径改写、超时配置、重试机制、请求头重写、响应头重写、Websocket、本地限流、健康检查、全局限流、Cookie 重写、访问黑白名单。 可以单独使用某一种策略，也可以组合使用多种策略达到最佳实践。 有关 API 策略的组合配置，参考 API 策略配置最佳实践。

有两种方式可以配置 API 策略：

- 在创建 API 的过程中设置策略，参考添加 API。
- 在 API 创建完成后通过更新 API 策略配置进行调整。

视频教程：

- API 策略的高级配置（1）
- API 策略的高级配置（2）

每一项策略的配置说明如下:

负载均衡

当 API **的目**标后端服务为多实例服务时，**可以通**过负载均衡策略控制流量分发，**根据**业务场景调整不同服务的实例接收到的流量。

- **随机**

  默认的负载均衡策略。选择随机规则时，**网关会将**请求随机分发给后端服务的任意实例。**在流量**较小时，**部分后端服**务可能会负载较多。**效果参考下**图：

  负载均衡

  负载均衡

- 轮询

  向后端服务的所有实例轮流分发请求，**各个服**务实例接收到的请求数基本相近。**此**规则可以在流量较小时保障流量的平均分配。**效果参考下**图：

  负载均衡

  负载均衡

- 权重

  根据 API **目**标后端服务的权重分发流量，权重数值越大，优先级越高，**承担的流量也相**对较多。**服**务权重的配置入口见下图：

  负载均衡

  负载均衡

- Cookie

将来源请求头中属于相同 Cookie 的流量分发到固定的后端服务实例，前提是后端服务能够根据 Cookie 做出不同的响应处理。

- 请求 Hash

  选择请求 Hash 时，可以通过一些高级策略来进负载均衡分配。当前支持的 Hash 策略为：IP、请求参数。

负载均衡

负载均衡

路径改写

如果对外暴露的 API 路径与后端服务提供的路径不一致，可以改写 API 路径使其与后端服务的路径一致，确保服务的正常访问。 启用路径改写后，网关会将外部请求流量转发到重写后的路径。

!!! note
需要确保重写的路径是真实存在的，并且路径正确，以 “/” 开头。

路径改写

路径改写

超时配置

设置请求响应的最大时长，如果超出所设置的最大时长，则直接请求失败。超时时长支持数值类型为 >=1 的整数值，时间单位为“秒（s）”。

超时配置默认处于关闭状态，开启后必须配置超时时长。开启超时配置有助于减少异常处理导致的阻塞问题。

超时

超时

超时

重试机制

微服务网关的 API 支持配置非常丰富的重试机制。启用重试机制后，网关会在请求失败时自动重新尝试访问。 达到重试超时时间之后自动触发再次重试，当重试次数达到配置的最大重试次数时停止重试。重试机制默认处于关闭状态，开启后必须配置重试次数和重试超时时长。

支持通过自定义配置选择不同的重试条件，自定义重试状态码等。

HTTP 重试

- 5XX 响应错误：后端服务响应 HTTP status_code 大于 500 时进行重试。
- 网关错误：当响应结果为网关错误提示时，自动进行重试。
- 请求重置：当响应结果为请求重置消息时，自动进行重试。
- 连接失败：当响应结果为网络连接失败的返回时，自动进行重试。
- 拒绝流：当响应结果为后端服务将请求标记为拒绝处理时，自动进行重试。
- 指定状态码：当后端服务响应 HTTP status_code 为特定状态码时自动进行重试，支持配置特定的状态码。

GRPC 重试

- 请求被取消：当响应结果为 GRPC 请求被后端服务取消时自动进行重试。
- 响应超时： 当后端服务响应超时，自动进行重试。
- 服务内部错误：当响应结果为服务内部错误时，自动进行重试。
- 资源不足：当响应结果为资源不足时，自动进行重试。

- **服**务不可用时：**当响**应结果为后端不可用时，**自**动进行重试。

重试

重试

请求头/响应头改写

支持添加、**修改**、删除请求头和响应头及其对应的值。

- **增加**请求头/**响**应头：**使用**设置动作，**填写新的关**键字和新值。
- **修改**请求头/**响**应头：**使用**设置动作，**填写已有的关**键字并赋予新值。
- **移除**请求头/**响**应头，**使用移除**动作，**只填写需要移除的关**键字即可，**无需填写**对应的值。

header 改写

header 改写

Websocket

WebSocket **是一种在**单个 TCP 连接上进行全双工通信的协议。WebSocket **使得客**户端和**服**务器之间的数据交换变得更加简单，**允**许服务端主动向客户端推送数据。**在** WebSocket API **中**，浏览器和服务器只需要完成一次握手，**两者之**间就直接可以创建持久性的连接，**并**进行双向数据传输。

启用 Websocket **之后支持通**过 Websocket 协议访问 API **的后端服**务。

websocket

websocket

本地限流

微服务网关支持丰富的限流能力，**支持在** API 层级启用本地限流能力。

- 请求速率：时间窗口（秒/分/时）内允许的最大请求速率，例如每分钟最多允许 3 次请求。支持输入 >=1 的整数。
- 允许溢出速率：达到预设的请求速率时，仍旧允许额外处理一部分请求，适用于业务突增的流量高峰时段。支持输入 >=1 的整数。
- 限制返回码：默认返回码为 429，表示请求次数过多。可参考 envoy 官方文档了解本地限流支持的状态码。
- Header 关键字：默认为空，可根据需求自行设置。

下图中的配置表示：每分钟最多允许请求 8 次 (3+5)，第 9 次访问时会返回 429 状态码，提示访问次数过多。 每次请求成功后返回的响应内容都会带上 ratelimit：8 响应头。

本地限流

本地限流

!!! info
除了在 API 层级的本地限流能力之外，还可以通过[配置域名策略](../domain/domain-policy.md)针对整个域名进行限流处理。
当 API 与域名同时配置限流策略时，以 API 层级的限流策略为准。

健康检查

通过设置健康检查地址，可以有效保证当后端服务异常时，网关自动进行负载均衡调整。对不健康的后端服务进行标记，停止向该服务分发流量。当后端服务恢复并通过设定的健康检查条件后，自动恢复流量分发。

- 健康检查路径：以 “/” 开头，并且全部后端服务的所有实例都应提供相同的健康检查接口。
- 特定健康检查主机：配置主机地址后，仅对该主机进行健康检查。
- 检查时间间隔：每次健康检查的时间间隔，时间单位为“秒”，例如每隔 10 秒进行一

次健康检查。

- 检查超时时间：健康检查的最大超时时长，当健康检查超过配置的时长时，直接标记健康检查失败。
- 标记健康检查次数：连续检查 N 次并且每次结果都是健康时，才将服务实例标记为健康状态； 当服务实例被标记为健康状态后，请求流量将会自动分发到该服务实例。
- 标记不健康检查次数：连续检查 N 次并且每次结果都是不健康时，就将服务实例标记为不健康状态， 当服务实例被标记为不健康时，停止向该实例分发请求流量。

健康检查

健康检查

Cookie 重写

参考下方说明配置 cookie 重写策略：

- 名称：必须填写当前已经存在的 cookie 名称
- 域名：重新定义 cookie 的域名
- 路径：重新定义 cookie 的路径
- Secure：保持指启用安全模式，禁用指禁用安全模式。在安全模式下，请求必须为安全连接（HTTPS）， cookie 才会被保存下来。如果使用 HTTP 协议下，cookie 将无效
- Samesite：是否在跨域时发送 cookie
  - Strict：跨域请求严禁携带本站 cookie
  - Lax：大多数情况禁止，但是导航到目标网址的 Get 请求除外。

- None：**跨域**请求允许携带本站 cookie，**前提是** Secure **必**须设置为保持，**即只能在** HTTPS 协议下使用

cookie **重写**

cookie **重写**

访问黑白名单

启用访问黑白名单后，**可以**仅允许白名单上的 IP 请求通过网关，**拒**绝其他所有来源的请求；**或者拒**绝黑名单上的 IP 请求通过网关，**允**许其他所有来源的请求。

- **网关前置代理**层数：请求从客户端到网关中途需要经过几个代理端点。**例如 客**户端-Nginx-**网关 的代理**层数为 1，**因**为中间只经过 1 **个** Nginx **代理端点**。

  创建/**更新网关**时，**可以在网关的高**级配置部分设置代理层数，**需要按照**实际情况填写。

- Remote：IP **来源**为 Remote 时，**黑白名**单是否生效取决于网关前置代理层数。**当代理**层数为 n 时，**生效的是从网关开始向前第 n+1 个端点的** IP。 **例如 客**户端-Nginx-**网关 前置代理**层数为 1，则仅对网关向前第 2 **个端点的** IP **生效**，**即客**户端**的** IP。**如果填写** Nginx **的** IP，**黑白名**单不会生效。

  黑白名单

  黑白名单

- Peer：IP **来源**为 Peer 时，**无**论网关前置代理层数是多少，**黑白名**单都仅对网关的 **直接** 对端 IP **生效**。**例如客**户端-...-Nginx-**网关**，**无**论客户端和 Nginx **中**间有多少个**代理端点**，**黑白名**单都仅对最后一个 Nginx **的** IP **生效**。

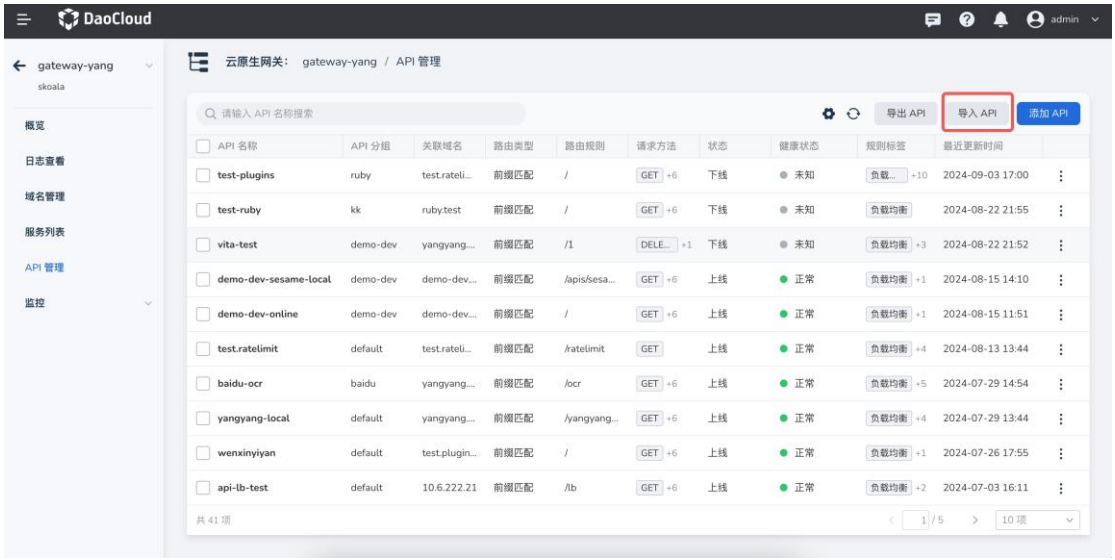
  黑白名单

黑白名单

API 导入

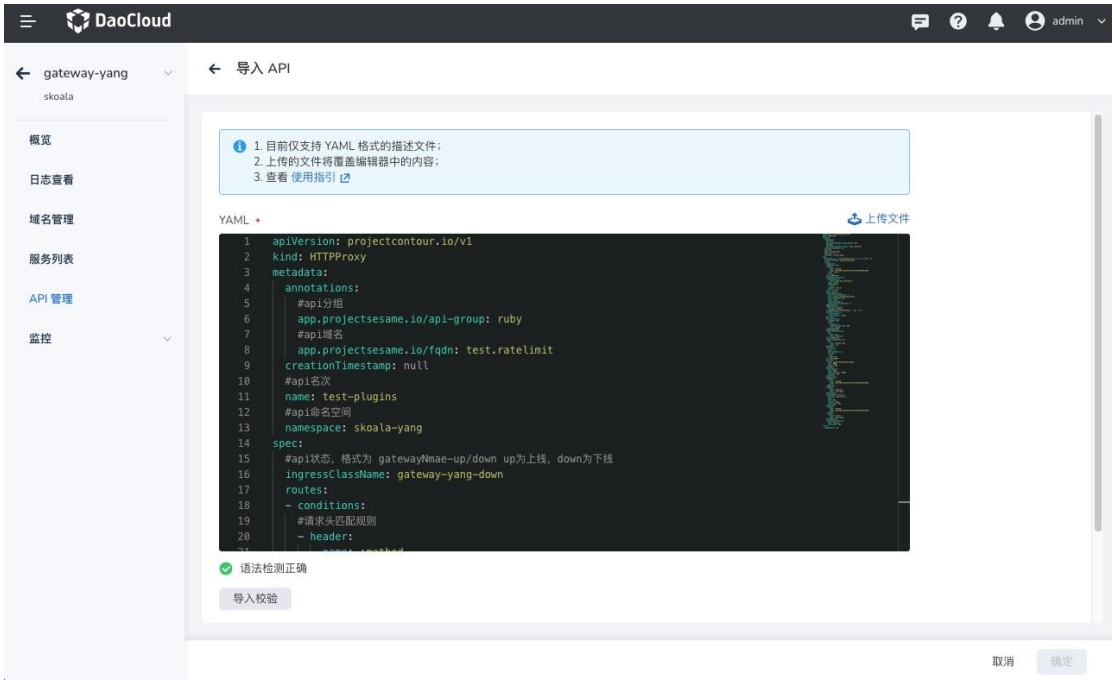
API 导入功能通常用于从外部系统或服务中获取数据并将其导入到当前应用中。 以下是一个简单的 API 导入功能的样例说明，包括请求格式和处理逻辑等。

依次选择 微服务引擎 -> 云原生网关 ，点击某个正常运行的网关名称，在左侧导航栏选择 API 管理 ， 点击右上角的 导入 API 按钮，您可以上传或填写新的 API 信息并将其录入到 DCE 5.0 微服务引擎中。



API 导入

通过 YAML 格式的文件导入 API 配置。



导入 yaml

YAML 样例如下：

```yaml
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  annotations:
    app.projectsesame.io/api-group: ruby # api 分组
    app.projectsesame.io/fqdn: test.ratelimit # api 域名
  creationTimestamp: null
  name: test-plugins-1 # api 名称
  namespace: skoala-yang # api 命名空间
spec:
  ingressClassName: gateway-yang-down # api 状态，格式为 gatewayNmae-up/down，up 为上线，down 为下线
  routes:
    - conditions:
        - header: # 请求头匹配规则
            name: :method
            regex: GET|POST|DELETE|PUT|PATCH|OPTIONS|HEAD
          prefix: / # 路径开头匹配
      cookieRewritePolicies: # cookie 重写规则
        - domainRewrite:
            value: demo-dev.daocloud.io
          name: cookie
          pathRewrite:
```

```
            value: /
          sameSite: Strict
          secure: true
      healthCheckPolicy: # 健康检查规则
        healthyThresholdCount: 1
        host: contour-envoy-healthcheck
        intervalSeconds: 60
        path: /test
        timeoutSeconds: 2
        unhealthyThresholdCount: 3
      loadBalancerPolicy: # 负载均衡策略
        strategy: Random
      pathRewritePolicy: # 路径重写策略，路径重写仅适用于路径开头匹配
        replacePrefix:
          - replacement: /test2
      rateLimitPolicy: # 限流策略
        local: # 本地限流策略
          requests: 2
          responseStatusCode: 429
          unit: second
      requestHeadersPolicy: # 请求头重写策略
        set:
          - name: host
            value: testhost
      responseHeadersPolicy: # 响应头重写策略
        set:
          - name: content-type
            value: xxx
      retryPolicy: # 重试策略
        count: 1
        perTryTimeout: 1s
        retryOn:
          - 5xx
          - cancelled
      services: # 路由的后端服务
        - name: sesame-ba73aa79cd-sesame
          port: 8088
          weight: 100
      timeoutPolicy: # 超时策略
        idle: 300s
        idleConnection: 3600s
        response: 15s
    - conditions:
        - header: # 请求方法匹配
```

```
            name: :method
            regex: GET|POST|DELETE|PUT|PATCH|OPTIONS|HEAD
          prefix: /
        - header: # 请求头匹配
            exact: redirect
            name: route-type
      requestRedirectPolicy: # 重定向策略
        hostname: daocloud.io
        path: /
        port: 443
        scheme: https
        statusCode: 301
    - conditions:
        - header:
            name: :method
            regex: GET|POST|DELETE|PUT|PATCH|OPTIONS|HEAD
          prefix: /
        - header:
            exact: direct
            name: route-type
      directResponsePolicy: # 直接响应策略
        body: success
        statusCode: 200
status:
  loadBalancer: {}
```

手动管理服务

添加成功的服务会出现在服务列表页面，添加 API 时也可以选择列表中的服务作为目标后端服务。 微服务网关支持通过手动接入和自动发现两种方式添加服务。本页介绍如何手动接入服务。

接入服务

1. 在 **云原生网关列表** 页面点击目标网关的名称，然后在左侧导航栏点击 **服务列表** ，接着在右上角点击 **添加服务** 。

服务列表

服务列表

2. 选择服务来源，配置服务连接信息，点击 **确定** 。

=== “集群服务”

选择目标服务所在的集群和命名空间，填写地址以及端口。

![添加集群服务](docs/zh/docs/skoala/gateway/service/images/cluster-svc.png)

对于集群服务的访问方式，可在 __容器管理__ -> __容器网络__ -> __服务__ 中点击服务名称进行查看：

![获取服务访问地址](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/skoala/gateway/service/images/service-access.png)

=== “网格服务”

选择目标服务所在的集群和命名空间，填写地址以及端口。

![添加网格服务](docs/zh/docs/skoala/gateway/service/images/mesh.png)

=== “接入注册中心服务”

选择目标服务所在的注册中心，填写访问协议、地址和端口。

![添加注册中心服务](docs/zh/docs/skoala/gateway/service/images/jieru.png)

=== “注册配置中心服务”

选择目标服务所在的注册中心，填写访问协议、地址和端口。

![添加注册中心服务](docs/zh/docs/skoala/gateway/service/images/zhuce.png)

=== “外部服务”

填写服务名称、访问协议、地址、端口。

![添加外部服务](docs/zh/docs/skoala/gateway/service/images/waibu.png)

查看服务详情

1. 在服务列表页面点击目标服务的名称，进入服务详情页面。

服务详情

服务详情

2. 查看服务来源、连接信息、关联 API 等信息。

服务详情

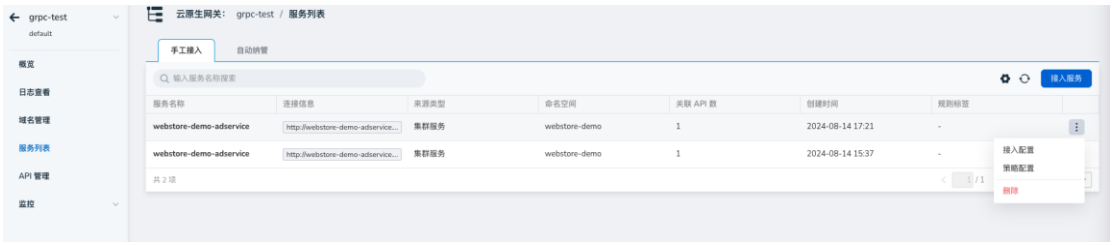
服务详情

更新服务

更新基础配置

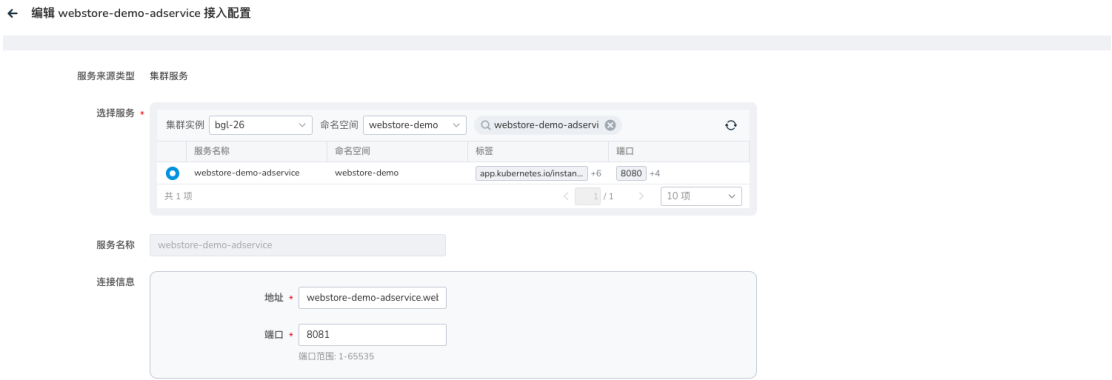
更新基础配置指修改服务的名称、协议、地址和端口等连接信息。

1. 在 **服务列表** 页面找到需要更新的服务，在服务右侧点击 □ ，选择 **接入配置** 。



更新服务

2. 更新基本信息，点击 **确定** 。



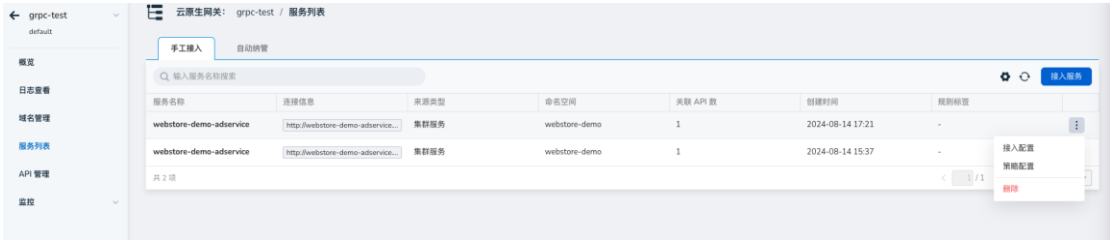
更新服务

!!! danger

如果更新基础配置时选择了其他服务或修改了外部服务的连接信息，那么原来的服务会被删除，相当于添加了一个新的服务。
但原服务关联的 API 会被自动关联到新的服务。

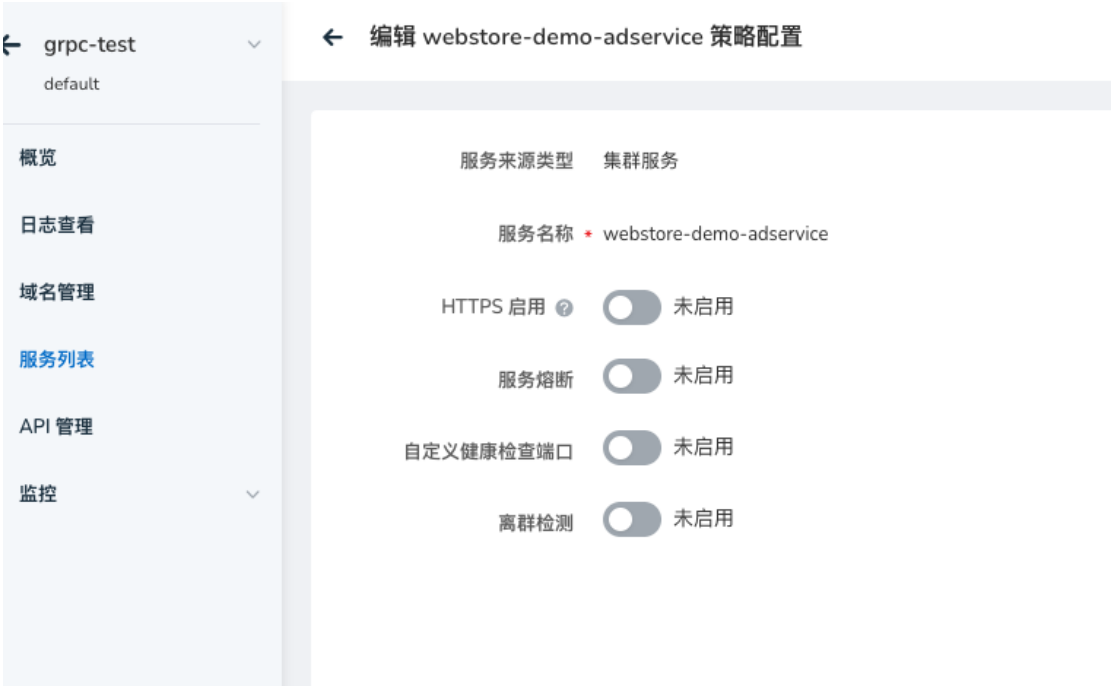
更新策略配置

1. 在 **服务列表** 页面找到需要更新的服务，在服务右侧点击 □ ，选择 **修改策略配置** 。



更新服务

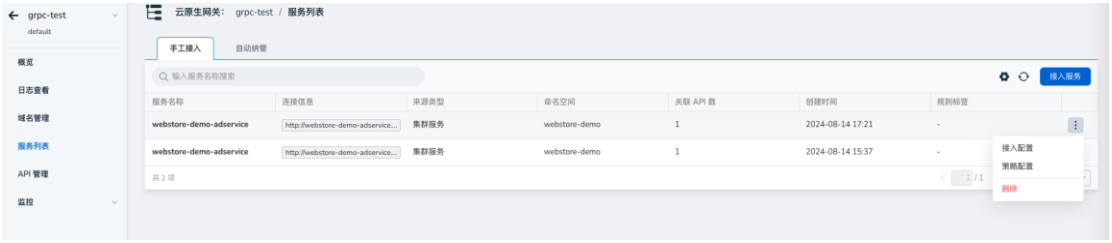
2. 更新策略配置，点击 **确定** 。



更新服务

删除服务

在 **服务列表** 页面找到需要删除的服务，在服务右侧点击 □ ，选择 **删除** 。



删除服务

删除服务之前，需要确保没有 API 正在使用该服务。如果该服务正在被某个 API 使用，需要先根据页面提示，点击 **API 管理** 删除关联的 API 之后才能删除该服务。

删除服务

删除服务

自动纳管服务

添加成功的服务会出现在服务列表页面，添加 API 时也可以选择列表中的服务作为目标后端服务。 微服务网关支持通过手动接入和自动纳管两种方式添加服务。本页介绍如何自动纳管服务。

网关实例创建成功之后，服务来源中的服务会被自动添加到该网关实例的服务列表中，无需手动添加。

查看自动纳管的服务

1. 在 **微服务网关列表** 页面点击目标网关的名称，进入网关概览页后，在左侧导航栏点击 **服务接入 -> 服务列表** 。

服务列表

服务列表

2. 在 **服务列表** 页面点击 **自动纳管** 。

自动发现服务

自动发现服务

配置服务策略

1. 在 **服务列表** -> **自动纳管** 页面找到目标服务，在右侧点击 □ 选择 **修改策略配置** 。

策略配置

策略配置

2. 按需调整服务策略配置，在弹框右下角点击 **确定** 。
   - HTTPS 证书验证：**开启后**，必须通过证书校验才能成功访问该服务。
   - 服务熔断：**当最大**连接数、**最大**处理连接数、**最大并行**请求数、**最大并行重**试数 **任何一个** 指标达到设定的阈值时，自动切断对该服务的调用，保护系统整体的可用性。**当指**标降到设定的阈值之后，自动恢复对该服务的调用。

策略配置

策略配置

查看服务详情

1. 在 **服务列表** -> **自动纳管** 页面找到目标服务，点击服务名称。

服务详情

服务详情

2. 查看服务名称、来源、关联的 API 等信息。支持安装最近更新时间进行排序。

服务详情

服务详情

!!! info
对于自动纳管的服务，仅支持上述操作，不支持更新、删除服务的操作。

监控告警

微服务网关通过内置 Grafana 看板监控网关的资源使用情况、Envoy 的监控详情、服务监控详情、APIServer、HTTPProxy 等资源的运行信息。

进入微服务引擎模块，在左侧导航栏点击微服务网关，点击目标网关的名称。

点击名称

点击名称

业务监控看板

在左侧导航栏点击 监控告警 ，点击 业务监控看板 页签可查看业务监控信息。

点击名称

点击名称

指标 | 含义 |

: — | :— |

CPU 利用率 | 用于监测 Kubernetes 集群中的 Pod 的 CPU 利用率 |

内存利用率 | 用于监测 Kubernetes 集群中的 Pod 的 Memory 利用率 |

数据接收 | 用于监测 Kubernetes 集群中的 Pod 接收的网络字节数 |

Envoy 连接（Open） | 用于记录与 Envoy 代理相连接的 HTTP/HTTPS 下游（即客户端）的活动连接数，这包括正在处理请求或保持打开状态的连接数 |

Envoy（堆内存） | 用于记录 Envoy 代理服务器使用的内存堆的大小，内存堆是用于动态分配内存的一部分，用于存储对象、数据结构和其他运行时数据 |

Envoy（已分配内存） | 用于记录 Envoy 代理服务器当前分配的内存总量，它表示 Envoy 在运行过程中使用的实际物理内存大小 |

监听器（Active） | 用于记录 Envoy 代理中当前处于活动状态的监听器（Listener）的数量，监听器用于接收和处理传入的网络连接请求，它们定义了代理的网络入口点和配置 |

监听器（Warming） | 用于记录 Envoy 代理中当前正在启动或预热中的监听器（Listener）的数量，监听器的启动和预热过程是指在监听器可以接受和处理传入的网络连接之前所需的初始化和准备阶段 |

监听器（Draining） | 用于记录 Envoy 代理中当前正在排空中的监听器（Listener）的数量，监听器的排空是指在停止接受新连接之前，等待所有现有连接关闭的过程。这通常是进行平滑升级、维护或停止代理时的一种策略，以确保已建立的连接正常结束。 |

下行连接（总计） | 用于记录与 Envoy 代理相连接的 HTTP/HTTPS 下游（即客户端）的活动连接数，这包括正在处理请求或保持打开状态的连接数 |

下行延迟 | 用于记录 HTTP 下游请求的处理时间分布情况，它将请求根据其处理时间分为不同的时间桶（buckets），每个时间桶表示一个时间范围的请求数量 |

RPS（下行） | 用于记录 Envoy 代理接收到的 HTTP 下游请求的总数 |

CPS（下行） | 用于记录与 Envoy 代理相连的 HTTP 下游（即客户端）的连接总数 |

端点（健康百分比） | 用于记录集群成员的健康状态百分比，集群成员通常是指代理所连

接的上游服务或后端服务的实例或节点 |

端点（健康） | 用于记录集群成员的健康状态，集群成员通常是指代理所连接的上游服务或后端服务的实例或节点 |

端点（不健康） | 用于记录集群成员的不健康状态，集群成员通常是指代理所连接的上游服务或后端服务的实例或节点 |

端点（总计） | 用于记录集群成员的数量，集群成员通常是指代理所连接的上游服务或后端服务的实例或节点 |

上行应答（2xx） | 表示从 Envoy 代理发送到上游集群状态码为 2xx 的请求数量 |

上行应答（3xx） | 表示从 Envoy 代理发送到上游集群状态码为 3xx 的请求数量 |

上行应答（4xx） | 表示从 Envoy 代理发送到上游集群状态码为 4xx 的请求数量 |

上行应答（5xx） | 表示从 Envoy 代理发送到上游集群状态码为 5xx 的请求数量 |

上行连接（总计） | 用于记录与上游集群的当前活动连接数，活动连接是指与上游集群建立的有效连接，表示正在使用的连接数 |

上行延迟 | 用于将上游集群请求的响应时间按照预定义的时间区间（或桶）进行分组，并记录每个时间区间内的请求数量 |

CPS（上行） | 用于记录与上游集群建立的总连接数，它统计了与上游集群建立的所有连接，无论这些连接是活动的还是已经关闭的 |

RPS（上行） | 记录了代理向上游集群发送的所有请求的数量，该指标可以帮助监控和度量与上游集群之间的请求流量 |

资源看板

点击 资源看板 页签可查看资源监控信息。

点击名称

点击名称

指标 | 含义 |

: – | :– |

CPU 利用率 | 用于监测 Kubernetes 集群中的 Pod 的 CPU 利用率 |

内存利用率 | 用于监测 Kubernetes 集群中的 Pod 的 Memory 利用率 |

数据接收 | 用于监测 Kubernetes 集群中的 Pod 接收的网络字节数 |

已接收的 Kubernetes 对象更新总计（按操作与对象 Kind 计） | 按操作和对象类型分类的，由 Kubernetes 事件触发 Contour 产生变化的总数在时间范围内的平均值 |

已接收的 Kubernetes 对象更新总计（按对象 Kind 计） | 按对象类型分类的，由 Kubernetes 事件触发 Contour 产生变化的总数在时间范围内的平均值 |

HTTPProxy（总计） | 用于监测 HTTPProxy 资源的数量（网关 API 数量 + 网关域名数量） |

HTTPProxy（孤儿） | 用于监测是否存在孤立的 HTTPProxy 资源（状态为 orphaned 的 HTTPProxy） |

HTTPProxy（有效） | 用于监测是否存在有效的 HTTPProxy 资源（状态为 valid 的 HTTPProxy） |

HTTPProxy（无效） | 用于监测是否存在无效的 HTTPProxy 资源（状态为 invalid 的 HTTPProxy） |

请求日志

微服务网关支持查看请求日志和实例日志。本页介绍如何查看实例日志以及查看日志时的相关操作。

查看方式

点击目标网关的名称，进入网关概览页面，然后在左侧导航栏点击 日志查看 -> 请求日志 。

查看请求日志的路径

查看请求日志的路径

相关操作

- 筛选日志：支持通过 Request ID、请求路径、域名、请求方法、HTTP、GRPC 等条件筛选日志， 支持按照请求开始时间、请求耗时、请求服务耗时对日志进行排序。

  筛选日志

  筛选日志

- 限定时间范围：可选择近 5 分钟、15 分钟、30 分钟的日志，或者自定义时间范围。

  限定时间

  限定时间

- 导出日志：支持将日志文件导出到本地。

  导出日志

  导出日志

实例日志

微服务网关支持查看请求日志和实例日志。本页介绍如何查看实例日志以及查看日志时的相关操作。

查看方式

点击目标网关的名称，进入网关概览页面，然后在左侧导航栏点击 日志查看 -> 实例日志 。

查看实例日志的路径

相关操作

- 筛选日志：支持筛选实例仅查看某个容器组的日志，也可以参考跳转到可观测性模块的日志查询去查找特定内容。
- 限定时间范围：日志时间范围可以选择：近 5 分钟的日志、近 1 小时、近 12 小时、近 7 天的日志。
- 自动刷新：在查看日志时，可以查看实时日志。

!!! info
如需查看更多日志或下载日志，可前往[可观测性模块](../../../insight/intro/index.md)通过[日志查询](../../../insight/user-guide/data-query/log.md)功能查询或下载特定集群、命名空间、工作负载、容器组的日志。

托管注册中心

微服务引擎支持托管 Nacos 注册中心，即在微服务治理中心从零创建一个 Nacos 类型的全新注册中心，并且可以通过微服务治理中心全面管理该注册中心。 相对于接入型注册中心而言，托管型注册中心支持更多操作，包括查看注册中心实例的基础信息、 微服务命名空间管理、微服务列表、微服务配置列表、监控告警、日志查看、插件中心等。

创建托管注册中心

!!! note

- 需要事先在目标集群的 **skoala-system** 命名空间中安装 skoala-init 组件，
  具体步骤可参考[管理 Helm 应用](../../../kpanda/user-guide/helm/helm-app.md)。
- 如果所选集群中没有 **skoala-system** 命名空间，可参考[创建命名空间](../../../kpanda/user-guide/namespaces/createns.md)
  创建一个名为 **skoala-system** 的命名空间。

创建托管注册中心的步骤如下。

1. 在左侧导航栏点击 **传统微服务 -> 注册配置中心** ，然后在页面右上角点击 **创建注册配置中心** ，进入托管注册中心实例的创建页面。

   进入创建注册中心页面

   进入创建注册中心页面

2. 填写配置信息。

   需要注意的是：

   - **注册中心名称**：支持输入字母、数字和分隔符（-），注册中心创建之后不可更改名称。
   - **部署位置**：系统会自动校验所选集群下的 **skoala-system 命名空**间中是否安装了 skoala-init 组件。
     - 如未安装，则无法创建注册中心。可以根据页面提示去安装该组件。
     - 注册中心创建之后不可更改部署位置。
   - 资源配置：可直接选择 **1 核 2G** 、2 核 4G 等配置，也可以自定义配置资源限额。
     - 1 核 2G 指 CPU 的请求值和限制值分别为 2 核，内存的请求值和限制值分别为 2 G，以此类推。
     - 点击《实例能力评估》可以查看在 2 Core 4 GiB、4 Core 8 GiB 和 8 Core 16 GiB 等主流规格下的吞吐量 (TPS)。

进入创建注册中心页面

进入创建注册中心页面

- 访问方式：选择 **节点访问** **可通**过 **服务端口+目标端口** **的方式从外部**访问注册中心，选择 **内部访问** 则只能在所在的集群范围内通过服务端口访问注册中心。**默**认的服务端口为 8848。
- **服**务认证：**开启服**务认证时，**注册至**该注册中心的服务需要认证信息。**默**认用户名/**密**码为 nacos/nacos。**当数据持久化**为内置数据库模式时，**不支持开启服**务认证。
- **部署模式**：选择高可用模式时，节点数量不少于 **3 个**。**生**产环境下建议使用**高可用模式**。
- 镜像版本：**目前支持** v2.0.4-slim 、v2.1.1-slim **和** v2.2.3-slim **三种版本**。

进入创建注册中心页面

进入创建注册中心页面

- **数据持久化**：**建**议使用外置存储化。
  - **不使用外置存**储化：**数据存**储在 Nacos **所在的** Pod **的文件系**统里。Pod **重启之后数据会**遗失，**因此建**议使用外部存储。
  - **使用数据**库：**填写数据**库的名称、**地址**、**端口**、**用**户名和密码

进入创建注册中心页面

进入创建注册中心页面

3. **在**页面底部点击 **确定** 。

如果操作正确，页面右上角会弹出创建成功的消息，**托管注册中心列表**页会展示新建**的注册中心**实例。

创建成功

创建成功

!!! info
- 新建注册中心需要一段时间进行初始化，其间处于“启动中”状态。初始化完成后进入“运行中”状态。
- 点击注册中心的名称可以查看所在集群/命名空间、运行状态、资源配额、服务端口、存储配置、节点列表等基础信息。

更新托管注册中心

1. 在 **注册配置中心** **列表**页选择需要更新的注册中心，**在右**侧点击 ⍰ **并**选择 **编辑** 。

进入更新页面

进入更新页面

2. **更新注册中心的配置**，**然后在**页面底部点击 **确定** 。

!!! warning
- 修改存储池/数据库，变动存储位置后，之前的数据不会随之迁移！
- 托管注册中心名称和部署位置不可编辑。

更新配置

更新配置

查看注册中心详情

在注册配置中心列表页找到需要查看详情的注册中心实例，**点**击实例名称进入基础信息页面。

在基础信息页面可以查看注册中心的基础信息、**服**务治理信息（**需要开启服**务治理功能）、节点列表、**数据持久化信息等**。 “**运行状**态”**是注册中心**实例信息的一部分，**用于反映注册中心**实例的状态。

相关操作：

- **重启注册中心**实例：**在**页面右上角点击 **并**选择 **重启** **可以**对整个托管注册中心实例进行重启。

全量重启

全量重启

- 查看节点元数据

查看元数据

查看元数据

删除注册中心

1. **在** **托管注册中心列表** 页选择需要删除的注册中心，**在右**侧点击 **并**选择 **删除** 。

进入删除页面

进入删除页面

2. 输入注册中心的名称，**点**击 **移除** 。

确认名称

确认名称

!!! note
接入型的注册中心仅支持`移除`操作，而托管型的注册中心仅支持`删除`操作。二者的区别在于：

- 移除：只是将注册中心从 DCE 5.0 的微服务引擎中移除，不会删除原有的注册中心和数据，后续还可以再次接入该注册中心。
- 删除：删除注册中心及其中的所有数据，后续无法再次使用该注册中心，需要重新创建新的注册中心。

服务接入 Sentinel 规范

为了正常使用 DCE 5.0 **微服**务引擎提供的 Sentinel 流量治理**和**查看 Sentinel 数据监控，

需要将应用接入 Sentinel 控制台，并且传递应用参数时需要满足一定规范。

JAVA（无框架）

!!! note
- 下述操作使用于任何版本的 Sentinel SDK 版本。
- 使用下述方式接入服务时，使用界面创建规则并不会生效，因为客户端并无连接 Nacos 等存储获取规则，而是直接基于内存设置规则。
- 下述方式主要适用于 Sentinel 快速接入验证或者没有使用 Spring Cloud 框架的场景。

1. 在 pom.xml 文件中添加依赖项。

```
<!-- 添加 sentinel 核心依赖-->
    <dependency>
      <groupId>com.alibaba.csp</groupId>
      <artifactId>sentinel-core</artifactId>
      <version>1.8.6</version>
    </dependency>

    <!-- 添加 sentinel 控制台依赖-->
    <dependency>
      <groupId>com.alibaba.csp</groupId>
      <artifactId>sentinel-transport-simple-http</artifactId>
      <version>1.8.6</version>
    </dependency>
```

2. 使用 Sentinel API 设置规则。

```
private static void initFlowRules(){
        List<FlowRule> rules = new ArrayList<>();
        FlowRule rule = new FlowRule();
        rule.setResource("HelloWorld");
        rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
        // Set limit QPS to 20.
        rule.setCount(20);
        rules.add(rule);
        FlowRuleManager.loadRules(rules);
    }
```

3. 使用 Sentinel API 定义资源。

```
public static void main( String[] args ) {
        // 配置规则
        initFlowRules();

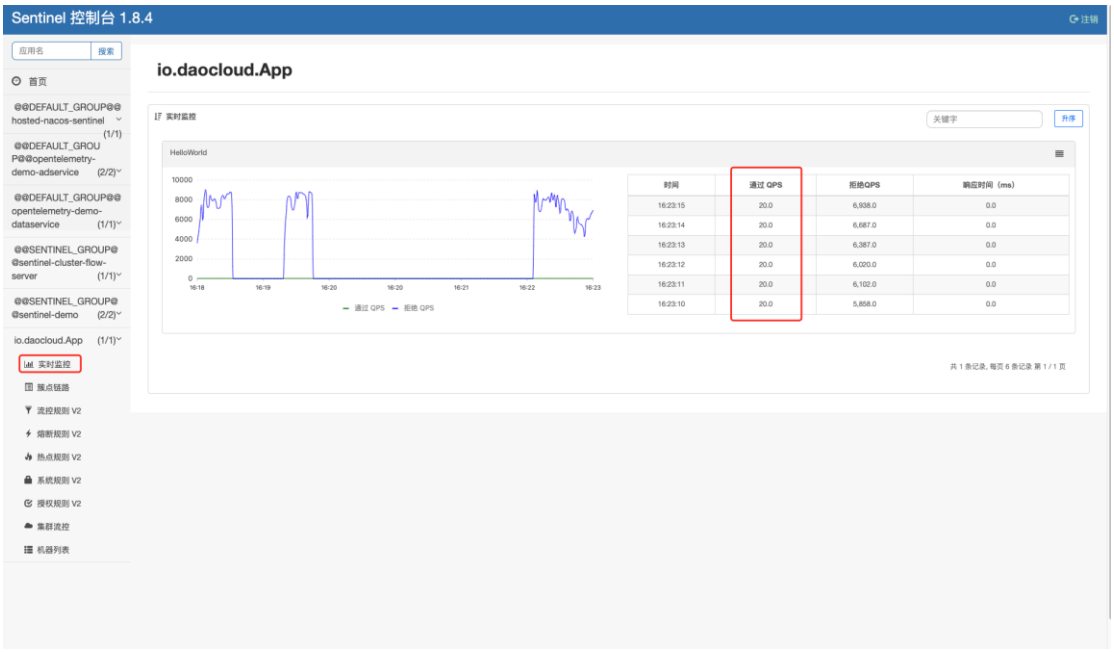
        while (true) {
```

```
        // 1.5.0 版本开始可以直接利用 try-with-resources 特性
        try (Entry entry = SphU.entry("HelloWorld")) {
        // 被保护的逻辑
        System.out.println("hello world");
        } catch (BlockException ex) {
        // 处理被流控的逻辑
        System.out.println("blocked!");
        }
        }
    }
```

4. 添加 jvm 参数连接 Sentinel 控制台，并启动应用。

```
# ip:port 为 Sentinel 控制台地址
    -Dcsp.sentinel.dashboard.server=10.6.176.50:32196
```

5. 打开 Sentinel 控制台，找到相应应用验证流控效果。



screenshot

JAVA（SpringCloud）框架

接入 SpringCloud 框架的微服务时，需要符合以下要求。

project.name 名称规范

应用接入 Sentinel 控制台的参数 project.name 必须使用如下格式：

{{nacos_namespace_id}}@@{{nacos_group}}@@{{appName}}
!!! note
- 符合此规范时，Sentinel 的治理规则会被推送到对应命名空间下，对应配置分组中的配置中心。
- 第一部分 `{{nacos_namespace_id}}` 指的是 Nacos 命名空间的 **ID**，而非命名空间的名称。
- Nacos 的 `public` 命名空间对应的 ID 是空字符 “”。如果想把应用接入 `public` 命名空间，必须使用空字符串，例如 `@@A@@appA`。
- 必须符合此规范，否则会引起未知错误。

Sentinel 写入 Nacos 配置中心 dataId 的命名规范

- 流控规则：{{appName}}-flow-rules
- 熔断规则：{{appName}}-degrade-rules
- 系统规则：{{appName}}-system-rules
- 授权规则：{{appName}}-authority-rules
- 热点规则：{{appName}}-param-rules

appName 为 project.name 三段式的最后一段。

Sentinel 与 Nacos 约定的内置通讯用户

username: skoala
password: 98985ba0-da90-41f6-b6dc-96f2ec49d973

推荐的接入方式

满足上述三项条件之后，推荐使用下述接入方式将 SpringCloud 框架的微服务接入 Sentinel。

!!! note
- 满足上述三项条件后，DCE 5.0 微服务引擎并不强制要求使用某种特定的接入方式。
- DCE 5.0 微服务引擎完全兼容相关开源框架使用，开发者可以根据需求灵活使用合适的接入方式。

1. 在 pom.xml 文件中添加依赖项，

版本对应关系参考：版本说明。

```
<!-- 添加 Sentinel 依赖-->
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
    </dependency>

    <!-- 添加 Sentinel Nacos 数据源依赖-->
    <dependency>
      <groupId>com.alibaba.csp</groupId>
      <artifactId>sentinel-datasource-nacos</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

2. 在项目中添加 application.yaml 配置文件。

```
```yaml title="application.yaml" nacos: # Nacos 地址 address: 10.6.176.50:30760 # Nacos 配置中心命名空间 namespace: # Nacos 配置中心分组 group: DEFAULT_GROUP # 接入 sentinel 控制台的应用名 project: name: nacos.namespace@@{nacos.group}@@${spring.application.name} spring: application: name: daocloud-springcloud cloud: sentinel: # Sentinel Nacos 数据源 datasource: flow: nacos: server-addr: ${nacos.address} dataId: ${spring.application.name}-flow-rules groupId: ${nacos.group} namespace: ${nacos.namespace} # 微服务引擎规定：Nacos 和 Sentinel 约定的用户及密码 username: skoala password: 98985ba0-da90-41f6-b6dc-96f2ec49d973 ruleType: flow degrade: nacos: server-addr: ${nacos.address} # 微服务引擎规定：Sentinel 写入 Nacos 的名称约定 dataId: ${spring.application.name}-degrade-rules groupId: ${nacos.group} namespace: ${nacos.namespace} # 微服务引擎规定：Nacos 和 Sentinel 约定的用户及密码 username: skoala password: 98985ba0-da90-41f6-b6dc-96f2ec49d973 rule-type: degrade system: nacos: server-addr: ${nacos.address} # 微服务引擎规定：Sentinel 写入 Nacos 的名称约定 dataId: ${spring.application.name}-system-rules groupId:
```
```

${nacos.group} namespace: ${nacos.namespace} # 微服务引擎规定：Nacos 和 Sentinel 约定的用户及密码 username: skoala password: 98985ba0-da90-41f6-b6dc-96f2ec49d973 rule-type: system authority: nacos: server-addr: ${nacos.address} # 微服务引擎规定：sentinel 写入 nacos 的名称约定 dataId: ${spring.application.name}-authority-rules groupId: ${nacos.group} namespace: ${nacos.namespace} # 微服务引擎规定：nacos 和 sentinel 约定的用户及密码 username: skoala password: 98985ba0-da90-41f6-b6dc-96f2ec49d973 rule-type: authority param-flow: nacos: server-addr: ${nacos.address} # 微服务引擎规定：sentinel 写入 nacos 的名称约定 dataId: ${spring.application.name}-param-rules groupId: ${nacos.group} namespace: ${nacos.namespace} # 微服务引擎规定：nacos 和 sentinel 约定的用户及密码 username: skoala password: 98985ba0-da90-41f6-b6dc-96f2ec49d973 rule-type: param-flow

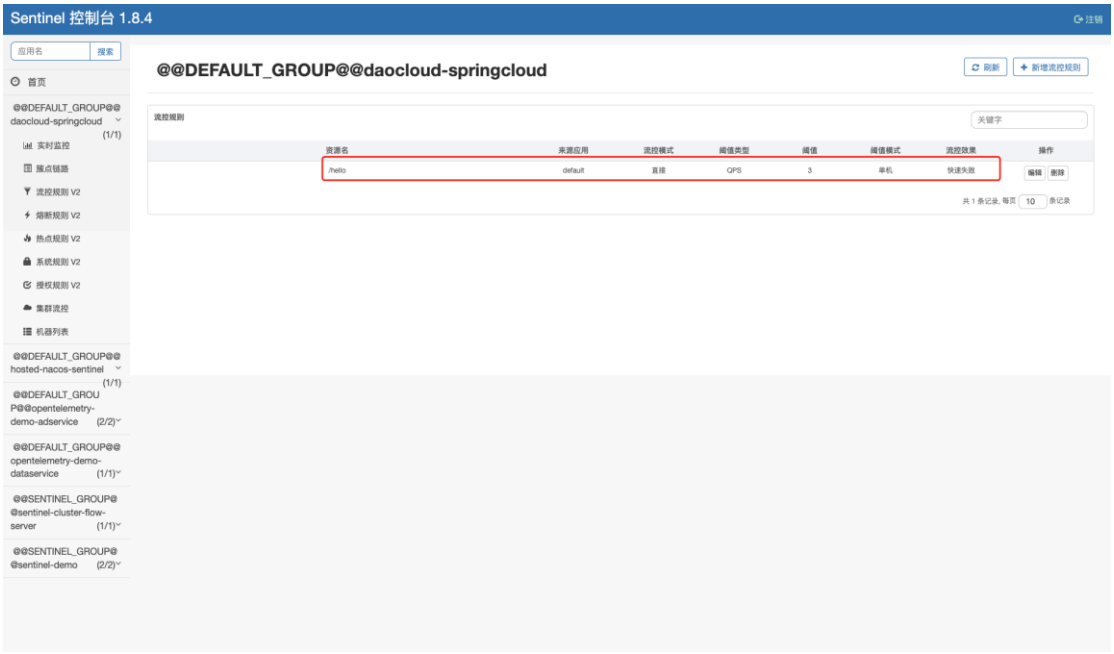
```

3. 编写相关的 Web API。

```
@SpringBootApplication
@RestController
@RequestMapping
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }

    @GetMapping("/hello")
    public String hello(){
        return "hello world";
    }
}
```

4. 在 Sentinel 控制台创建流控规则。
```



screenshot

5. 启动应用并添加连接 Sentinel 控制台的 jvm 参数

   -Dcsp.sentinel.dashboard.server=10.6.176.50:32196

6. 使用脚本访问设置了流控规则的 Web API。例如：

   ```
   while true; do curl http://localhost:8080/hello; done
   ```

7. Sentinel 控制台验证规则是否生效。

screenshot

服务接入 Nacos SDK

本文介绍不同框架的传统微服务如何接入 Nacos 的原生 SDK。

JAVA（无框架）

1. 在 pom.xml 文件中添加依赖项。目前的最新版本为 2.2.2

```
<dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>${latest.version}</version>
    </dependency>
```

2. 在服务中添加服务注册和服务发现的代码：

```
//添加配置变量 serverAddr: Nacos 的地址, e.g.,: 192.168.0.0:8848. namespace: Nacos 中
    的命名空间
    Properties properties = new Properties();
    properties.setProperty("serverAddr", System.getProperty("serverAddr"));
    properties.setProperty("namespace", System.getProperty("namespace"));

    NamingService naming = NamingFactory.createNamingService(properties);
    //注册实例: 注册时带上服务的 IP 和端口
    naming.registerInstance("sentinel-demo", "11.11.11.11", 8888, "DEFAULT");

    System.out.println(naming.getAllInstances("sentinel-demo"));

    naming.deregisterInstance("sentinel-demo", "11.11.11.11", 8888, "DEFAULT");

    System.out.println(naming.getAllInstances("sentinel-demo"));
    //添加对服务的订阅，在变更时获取事件通知
    naming.subscribe("sentinel-demo", new EventListener() {
        @Override
        public void onEvent(Event event) {
            System.out.println(((NamingEvent)event).getServiceName());
            System.out.println(((NamingEvent)event).getInstances());
            }
        });
```

3. 如需添加 Nacos 的更多特性，可参考更多使用方式

JAVA (SpringBoot) 框架

1. 在 pom.xml 文件中添加依赖项

```
<dependency>
        <groupId>com.alibaba.boot</groupId>
        <artifactId>nacos-config-spring-boot-starter</artifactId>
        <version>${latest.version}</version>
    </dependency>
```

!!! note

- 版本 [0.2.x.RELEASE](https://mvnrepository.com/artifact/com.alibaba.boot/nacos-config-spring-boot-starter) 对应的是 Spring Boot 2.x 版本。
  - 版本 [0.1.x.RELEASE](https://mvnrepository.com/artifact/com.alibaba.boot/nacos-config-spring-boot-starter) 对应的是 Spring Boot 1.x 版本。

2. 在项目中添加 bootstrap.yaml 配置文件

```
spring:
    application:
      name: demo
  nacos:
    config:
      data-id: test # Nacos 配置的 data-id
      server-addr: 127.0.0.1:8848 # Nacos 服务器地址
      group: DEFAULT_GROUP # 配置文件 Group
      namespace: public # 命名空间 ID
      type: yaml # Nacos 配置文件类型
      auto-refresh: true # 是否启用动态刷新配置
    discovery:
      server-addr: 127.0.0.1:8848 # Nacos 服务器地址
      group: DEFAULT_GROUP # 注册应用的 Group
      namespace: public # Nacos 的命名空间
```

3. 服务注册功能无需改动代码，直接启动项目就能在服务列表看到启动的服务。

4. 添加服务配置的代码

   1. 登录 Nacos 控制台添加配置文件。

      screenshot

      screenshot

   2. 然后在控制器代码中添加如下代码：

      ```
      @RestController
        @RequestMapping("config")
        public class Controller {
      ```

```
    @NacosValue(value = "${a.test}", autoRefreshed = true)
    private String name;

    @GetMapping("get")
    public String get() {
        return this.name;
    }
}
```

5. 如需添加 Nacos 的更多特性，可参考更多使用方式

JAVA (SpringCloud) 框架

1. 在 pom.xml 文件中添加依赖项。

   版本对应关系参考：版本说明

```
<dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
        <version>${latest.version}</version>
    </dependency>

    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>${latest.version}</version>
    </dependency>
```

2. 在项目中添加 bootstrap.yaml 配置文件

```
spring:
    application:
      name: demo
  ---
  spring:
    cloud:
      nacos:
        config:
          enabled: true
          server-addr: 127.0.0.1:8848 # nacos 服务器地址
          name: test # nacos 配置的 data-id
          group: DEFAULT_GROUP # 配置文件 Group
          namespace: public # 命名空间 ID
          file-extension: yaml # 配置文件后缀
```

```
    discovery:
      enabled: true
      server-addr: 127.0.0.1:8848 # nacos 服务器地址
      namespace: public # 命名空间 ID
      group: DEFAULT_GROUP # 应用分组名
```

3. 在服务添加 Nacos 服务注册的代码

在启动类上添加 @EnableDiscoveryClient 注解开启服务注册

```
@SpringBootApplication
  @EnableDiscoveryClient
  public class NacosProviderApplication {

    public static void main(String[] args) {
      SpringApplication.run(NacosProviderApplication.class, args);
    }
  }
```

4. 添加动态配置的代码

1. 在 Nacos 控制台添加配置文件

screenshot

screenshot

2. 在控制器中添加 @RefreshScope 和 @Value 注解

通过 springcloud 的 @RefreshScope 注解可以实现自动配置

```
@RestController
  @RequestMapping("config")
  @RefreshScope
  public class Controller {
      @Value(value = "${a.test}")
      private String name;

      @GetMapping("get")
      public String get() {
          return this.name;
      }
  }
```

5. 如需添加 Nacos 的更多特性，可参考更多使用方式

Go 框架

在 Go 框架微服务中添加 Nacos SDK 时，需要满足以下两个前提条件：

- Go 版本 1.15 以上
- Nacos 版本 2.x 以上

具体操作步骤如下：

1. 获取依赖

```
go get -u github.com/nacos-group/nacos-sdk-go/v2
```

2. 在服务中添加服务注册的代码

```
//create ServerConfig,配置 Nacos 服务器的地址
    sc := []constant.ServerConfig{
      *constant.NewServerConfig("127.0.0.1", 8848, constant.WithContextPath("/nacos")),
    }

    //create ClientConfig，配置客户端的链接配置
    cc := *constant.NewClientConfig(
      constant.WithNamespaceId("public"), //命名空间 ID
      constant.WithTimeoutMs(5000), // 超时时间
      constant.WithLogDir("/tmp/nacos/log"), //日志地址
      constant.WithCacheDir("/tmp/nacos/cache"), // Nacos 服务缓存的地址
      constant.WithLogLevel("debug"), // 日志级别
    )

    // 通过 ServerConfig 和 ClientConfig 创建 nacosclient 链接
    client, err := clients.NewNamingClient(
      vo.NacosClientParam{
        ClientConfig:  &cc,
        ServerConfigs: sc,
      },
    )

    //Register 注册服务
    registerServiceInstance(client, vo.RegisterInstanceParam{
      Ip:          "10.0.0.10",
      Port:        8848,
      ServiceName: "demo.go",
      GroupName:   "group-a",
      ClusterName: "cluster-a",
```

```
    Weight:      10,
    Enable:      true,
    Healthy:     true,
    Ephemeral:   true,
    Metadata:    map[string]string{"idc": "shanghai"},
})

//DeRegister 注销服务
deRegisterServiceInstance(client, vo.DeregisterInstanceParam{
    Ip:          "10.0.0.10",
    Port:        8848,
    ServiceName: "demo.go",
    GroupName:   "group-a",
    Cluster:     "cluster-a",
    Ephemeral:   true, //must be true
})
```

3.配置加载

```
//create ServerConfig,配置 Nacos 服务器的地址
sc := []constant.ServerConfig{
    *constant.NewServerConfig("127.0.0.1", 8848, constant.WithContextPath("/nacos")),
}

//create ClientConfig，配置客户端的链接配置
cc := *constant.NewClientConfig(
    constant.WithNamespaceId("public"), //命名空间 ID
    constant.WithTimeoutMs(5000), // 超时时间
    constant.WithLogDir("/tmp/nacos/log"), //日志地址
    constant.WithCacheDir("/tmp/nacos/cache"), // Nacos 服务缓存的地址
    constant.WithLogLevel("debug"), // 日志级别
)

// 通过 ServerConfig 和 ClientConfig 创建 nacosclient 链接
client, err := clients.NewNamingClient(
    vo.NacosClientParam{
        ClientConfig:  &cc,
        ServerConfigs: sc,
    },
)

//get config 获取配置
content, err := client.GetConfig(vo.ConfigParam{
    DataId: "test-data",
    Group:  "test-group",
```

```
})
fmt.Println("GetConfig,config :" + content)

//Listen config change,key=dataId+group+namespaceId. 监听配置改变
err = client.ListenConfig(vo.ConfigParam{
    DataId: "test-data",
    Group:  "test-group",
    OnChange: func(namespace, group, dataId, data string) {
        fmt.Println("config changed group:" + group + ", dataId:" + dataId + ", content:"
+ data)
    },
})
```

4. 如需添加 Nacos 的更多特性，可参考更多使用方式

Python 框架

1. 获取依赖

```
pip install nacos-sdk-python
```

2. 代码接入

```
import nacos

# Nacos server 地址
SERVER_ADDRESSES = "127.0.0.1:8848"
# 命名空间 ID
NAMESPACE = "public"

client = nacos.NacosClient(SERVER_ADDRESSES, namespace=NAMESPACE)

# 注册服务
client.add_naming_instance("test.service1", "1.0.0.7", 8080, "testCluster2", 0.2, "{}", Fals
e, True)
# 发送心跳
client.send_heartbeat("test.service", "1.0.0.7", 8080, "testCluster2", 0.1, "{}")


# get config
data_id = "dev-config"
group = "DEFAULT_GROUP"

# 全局服务配置
server_config = json.loads(client.get_config(data_id, group))
```

```
# 当服务配置发生变化
def config_update(data):
    global server_config
    server_config = json.loads(data['content'])
    print('new data->', server_config)

# 监听服务配置变化
client.add_config_watcher(data_id, group, config_update)
```

3. 如需添加 Nacos 的更多特性，可参考更多使用方式

Node.js 框架

1. 添加依赖

   需要使用 2.x 以上的版本

   ```
   npm install nacos --save
   ```

2. 添加服务发现的代码逻辑

```
'use strict';

const NacosNamingClient = require('nacos').NacosNamingClient;
const logger = console;

const client = new NacosNamingClient({
  logger,
  serverList: '127.0.0.1:8848', // Nacos 服务地址
  namespace: 'public', //命名空间 ID
});
await client.ready();

const serviceName = 'nodejs.test.domain';

// 注册服务
await client.registerInstance(serviceName, {
  ip: '1.1.1.1',
  port: 8080,
});
await client.registerInstance(serviceName, {
  ip: '2.2.2.2',
  port: 8080,
});
```

```
// 订阅服务
client.subscribe(serviceName, hosts => {
  console.log(hosts);
});

// 注销服务
await client.deregisterInstance(serviceName, {
  ip: '1.1.1.1',
  port: 8080,
});
```

3. 添加动态配置的代码逻辑

```
import {NacosConfigClient} from 'nacos';     // ts
const NacosConfigClient = require('nacos').NacosConfigClient; // js


const configClient = new NacosConfigClient({
  serverAddr: '127.0.0.1:8848',
  namespace: 'public', //命名空间 ID
});

// 获取配置文件
const content= await configClient.getConfig('test', 'DEFAULT_GROUP'); //dataID: test,g
roup: DEFAULT_GROUP
console.log('getConfig = ',content);

// 监听配置变化
configClient.subscribe({
  dataId: 'test',
  group: 'DEFAULT_GROUP',
}, content => {
  console.log(content);
});
```

4. 如需添加 Nacos 的更多特性，可参考更多使用方式

C++ 框架

1. 下载依赖

下载工程源代码并执行下述命令:

```
cd nacos-sdk-cpp
    cmake .
    make
```

执行命令后会产生一个 libnacos-cli.so 和一个 nacos-cli.out 文件

2. 运行 make install 将 libnacos-cli 安装到 lib 目录

3. 在服务代码中添加服务注册的代码逻辑

```
#include <iostream>
    #include <unistd.h>
    #include "Nacos.h"

    using namespace std;
    using namespace nacos;

    int main() {
        Properties configProps;
        configProps[PropertyKeyConst::SERVER_ADDR] = "127.0.0.1"; // nacos 服务地址
        configProps[PropertyKeyConst::NAMESPACE] = "public"; // 命名空间 ID
        NacosServiceFactory *factory = new NacosServiceFactory(configProps);
        ResourceGuard <NacosServiceFactory> _guardFactory(factory);
        NamingService *namingSvc = factory->CreateNamingService();
        ResourceGuard <NamingService> _serviceFactory(namingSvc);
        Instance instance;
        instance.clusterName = "DEFAULT";
        instance.ip = "127.0.0.1";
        instance.port = 2333;
        instance.instanceId = "1";
        instance.ephemeral = true;

        //模拟注册 5 个服务
        try {
            for (int i = 0; i < 5; i++) {
                NacosString serviceName = "TestNamingService" + NacosStringOps::value
Of(i);
                instance.port = 2000 + i;
                namingSvc->registerInstance(serviceName, instance);
            }
        }
        catch (NacosException &e) {
            cout << "encounter exception while registering service instance, raison:" << e.
what() << endl;
            return -1;
```

```
        }
        sleep(30);
        //注销服务
        try {
            for (int i = 0; i < 5; i++) {
                NacosString serviceName = "TestNamingService" + NacosStringOps::value
    Of(i);

                namingSvc->deregisterInstance(serviceName, "127.0.0.1", 2000 + i);
                sleep(1);
            }
        }
        catch (NacosException &e) {
            cout << "encounter exception while registering service instance, raison:" << e.
    what() << endl;
            return -1;
        }
        sleep(30);

        return 0;
    }
```

4. 添加动态配置的代码逻辑

```
#include <iostream>
    #include "Nacos.h"

    using namespace std;
    using namespace nacos;

    class MyListener : public Listener {
    private:
        int num;
    public:
        MyListener(int num) {
            this->num = num;
        }

        void receiveConfigInfo(const NacosString &configInfo) {
            cout << "==================================" << endl;
            cout << "Watcher" << num << endl;
            cout << "Watched Key UPDATED:" << configInfo << endl;
            cout << "==================================" << endl;
        }
    };
```

```
int main() {
    Properties props;
    props[PropertyKeyConst::SERVER_ADDR] = "127.0.0.1:8848"; //nacos 地址
    props[PropertyKeyConst::NAMESPACE] = "public"; // 命名空间 ID
    NacosServiceFactory *factory = new NacosServiceFactory(props);
    ResourceGuard <NacosServiceFactory> _guardFactory(factory);
    ConfigService *n = factory->CreateConfigService();
    ResourceGuard <ConfigService> _serviceFactory(n);

    MyListener *theListener = new MyListener(1);//You don't need to free it, since it will be deleted by the function removeListener
    n->addListener("dqid", "DEFAULT_GROUP", theListener);//dataID 为"dqid"并且 group 为"DEFAULT_GROUP"的配置改变都将监听到

    cout << "Input a character to continue" << endl;
    getchar();
    cout << "remove listener" << endl;
    n->removeListener("dqid", NULLSTR, theListener);//取消监听
    getchar();

    return 0;
}
```

5. 如需添加 Nacos 的更多特性，可参考[更多使用方式]

服务接入 Sentinel 监控

本文介绍如何为传统观念微服务接入 Sentinel 监控功能。

1. 添加依赖项

   sentinel metric exporter SDK 的版本需要 >= v2.0.0-alpha

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-metric-exporter</artifactId>
    <version>v2.0.0-alpha</version>
</dependency>
```

   如需了解相关原因，可参考：https://github.com/alibaba/Sentinel/pull/2976

2. 启动服务时添加 javaagent 参数，且 JMX 端口固定为 12345

-javaagent:/jmx_prometheus_javaagent-0.17.0.jar=12345:/prometheus-jmx-config.yaml

有关 JMX 的详细说明，可参考使用 JMX Exporter 暴露 JVM 监控指标。

3. 为服务创建 Kubernetes Service。重点包括以下参数：

   - labels 字段：固定为 skoala.io/type: sentinel
   - ports 字段：固定为 name: jmx-metrics，port: 12345，targetPort: 12345

```yaml
apiVersion: v1
  kind: Service
  metadata:
    labels:
      skoala.io/type: sentinel
    name: sentinel-demo
    namespace: skoala-jia
  spec:
    ports:
    - name: jmx-metrics
      port: 12345
      protocol: TCP
      targetPort: 12345
    selector:
      app.kubernetes.io/name: sentinel-demo
```

!!! note “如需了解相关原因，可参考 ServiceMonitor CR 的定义”

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    release: insight-agent
    operator.insight.io/managed-by: insight
  name: sentinel-service-monitor
spec:
  endpoints:
    - port: jmx-metrics
      scheme: http
  jobLabel: jobLabel
  namespaceSelector:
    any: true
  selector:
    matchLabels:
```

```
        skoala.io/type: sentinel
```

服务接入 Sentinel 集群流控

为服务接入 Sentinel 集群流控时，需要从服务器和客户端两方面调整配置。

集群流控服务器

DCE 5.0 不提供托管的 Sentinel 服务器，所以用户需自己启动 Sentinel 官方提供的集群流控服务器，且该服务器需要满足以下条件：

- 连接 Nacos，以便从 Nacos 读取持久化配置
- 为 Sentinel 服务器命名，以便作为配置前缀来区分多个集群流控服务器的 NamespaceSet 和 ServerTransport 配置

```
ReadableDataSource<String, Set<String>> namespaceDs = new NacosDataSource<>(nacosAddress, DEFAULT_GROUP,
                        serverName + NAMESPACESET_POSTFIX,
                        source -> JSON.parseObject(source, new TypeReference<Set<String>>() {}));
                ClusterServerConfigManager.registerNamespaceSetProperty(namespaceDs.getProperty());
                ReadableDataSource<String, ServerTransportConfig> transportConfigDs = new NacosDataSource<>(nacosAddress, DEFAULT_GROUP,
                        serverName + SERVERTRANSPORT_POSTFIX,
                        source -> JSON.parseObject(source, new TypeReference<ServerTransportConfig>() {}));
                ClusterServerConfigManager.registerServerTransportProperty(transportConfigDs.getProperty());
```

- 相关的代码实现，可参考 [sentinel-cluster-flow-control-java]

集群流控客户端

由于 Sentinel 官方提供的客户端 sentinel-cluster-client-default SDK 只适配内存模式使用。

而 DCE 5.0 也为集群流控客户端配置做了持久化，所以官方原始客户端 SDK 已无效。

1. 首先，用户需引入如下代码：

```
public class SentinelClusterClientInitFunc implements InitFunc {

    private String nacosAddress;
    private Properties properties = new Properties();
    private String groupId;
    private String dataId;

    @Override
    public void init() throws Exception {
        getNacosAddr();

        parseAppName();

        initDynamicRuleProperty();

        initClientConfigProperty();

        initClientServerAssignProperty();

        initStateProperty();
    }

    private void getNacosAddr() {
        nacosAddress = System.getProperty("nacos.address");
        if (StringUtil.isBlank(nacosAddress)){
            throw new RuntimeException("nacos address start param must be set");
        }
        System.out.printf("nacos address: %s\n", nacosAddress);
    }

    private void parseAppName() {
        String[] apps = APPNAME.split("@@");
        System.out.printf("app name: %s\n", APPNAME);
        if (apps.length != 3) {
            throw new RuntimeException("app name format must be set like this: {{
namespaceId}}@@{{groupName}}@@{{appName}}");
        } else if (StringUtil.isBlank(apps[1])){
            throw new RuntimeException("group name cannot be empty");
        }
        properties.put(PropertyKeyConst.NAMESPACE, apps[0]);
```

```
            properties.put(PropertyKeyConst.SERVER_ADDR, nacosAddress);
            properties.put(PropertyKeyConst.USERNAME, DEFAULT_NACOS_USERNAME
);
            properties.put(PropertyKeyConst.PASSWORD, DEFAULT_NACOS_PASSWORD
);
            groupId = apps[1];
            dataId = apps[2];
        }

        private void initDynamicRuleProperty() {
            ReadableDataSource<String, List<FlowRule>> ruleSource = new NacosDataSour
ce<>(properties, groupId,
                    dataId + FLOW_POSTFIX, source -> JSON.parseObject(source, new Type
Reference<List<FlowRule>>() {}));
            FlowRuleManager.register2Property(ruleSource.getProperty());

            ReadableDataSource<String, List<ParamFlowRule>> paramRuleSource = new N
acosDataSource<>(properties, groupId,
                    dataId +  PARAM_FLOW_POSTFIX, source -> JSON.parseObject(sou
rce, new TypeReference<List<ParamFlowRule>>() {}));
            ParamFlowRuleManager.register2Property(paramRuleSource.getProperty());
        }

        private void initClientConfigProperty() {
            ReadableDataSource<String, ClusterClientConfig> clientConfigDs = new Nacos
DataSource<>(properties, groupId,
                    dataId + CLUSTER_CLIENT_POSTFIX, source -> JSON.parseObject(
source, new TypeReference<ClusterClientConfig>() {}));
            ClusterClientConfigManager.registerClientConfigProperty(clientConfigDs.getProper
ty());
        }

        private void initClientServerAssignProperty() {
//            Cluster map format:
//            [
//                {
//                    "machineId": "10.64.0.81@8720",
//                    "ip": "10.64.0.81",
//                    "port": 18730,
//                    "clientSet": ["10.64.0.81@8721", "10.64.0.81@8722"]
//                }
//            ]
            ReadableDataSource<String, ClusterClientAssignConfig> clientAssignDs = new
NacosDataSource<>(properties, groupId,
```

```
                dataId + CLUSTER_MAP_POSTFIX, source -> {
            List<ClusterGroupDto> groupList = JSON.parseObject(source, new TypeReference<List<ClusterGroupDto>>() {});
            return Optional.ofNullable(groupList)
                .flatMap(this::extractClientAssignment)
                .orElse(null);
        });
        ClusterClientConfigManager.registerServerAssignProperty(clientAssignDs.getProperty());
    }

    private void initStateProperty() {
        ReadableDataSource<String, Integer> clusterModeDs = new NacosDataSource<>(properties, groupId,
                dataId + CLUSTER_MAP_POSTFIX, source -> {
            List<ClusterGroupDto> groupList = JSON.parseObject(source, new TypeReference<List<ClusterGroupDto>>() {});
            return Optional.ofNullable(groupList)
                .map(this::extractMode)
                .orElse(ClusterStateManager.CLUSTER_NOT_STARTED);
        });
        ClusterStateManager.registerProperty(clusterModeDs.getProperty());
    }

    private int extractMode(List<ClusterGroupDto> groupList) {
        if (groupList.stream().anyMatch(this::machineEqual)) {
            return ClusterStateManager.CLUSTER_SERVER;
        }

        boolean canBeClient = groupList.stream()
            .flatMap(e -> e.getClientSet().stream())
            .filter(Objects::nonNull)
            .anyMatch(e -> e.equals(getCurrentMachineId()));
        return canBeClient ? ClusterStateManager.CLUSTER_CLIENT : ClusterStateManager.CLUSTER_NOT_STARTED;
    }

    private Optional<ClusterClientAssignConfig> extractClientAssignment(List<ClusterGroupDto> groupList) {
        if (groupList.stream().anyMatch(this::machineEqual)) {
            return Optional.empty();
        }
        for (ClusterGroupDto group : groupList) {
            if (group.getClientSet().contains(getCurrentMachineId())) {
```

```
                    String ip = group.getIp();
                    Integer port = group.getPort();
                    return Optional.of(new ClusterClientAssignConfig(ip, port));
                }
            }
            return Optional.empty();
        }

        private boolean machineEqual(ClusterGroupDto group) {
            return getCurrentMachineId().equals(group.getMachineId());
        }

        private String getCurrentMachineId() {
            return HostNameUtil.getIp() + SEPARATOR + TransportConfig.getRuntimePort
    ();
        }

        private static final String SEPARATOR = "@";
    }
```

2. 接着在 Spring 容器中通过如下方式加载：

```
@PostConstruct
    public void initSentinelClusterFlow() throws Exception{
        new SentinelClusterClientInitFunc().init();
    }
```

微服务命名空间

微服务命名空间可用于隔离生产、开发、测试等不同环境下的服务、配置等资源。微服务引擎模块中的命名空间指的是微服务命名空间，即 Nacos 语境中的命名空间，并非 Kubernetes 场景下的命名空间。

!!! note
- 创建托管型注册中心实例时，系统会自动创建一个名为 **public** 的默认命名空间。该命名空间不可编辑、不可删除，属于系统自带的保留命名空间。
- 不同命名空间下的服务和配置严格隔离，不能互相引用。例如，A 命名空间下的服务不能引用 B 命名空间下的配置。

创建微服务命名空间

1. 进入 **微服务引擎** -> **传统微服务** -> **注册配置中心** 模块，点击目标注册中心的名称。

命名空间

命名空间

2. 在左侧导航栏点击 **微服务命名空间** ，然后在右上角点击 **创建** 。

创建命名空间

创建命名空间

3. 填写命名空间的 ID，名称，在页面右下角点击 **确定** 。

   如果不填写 ID，系统会自动生成一个 ID。**命名空间 ID 在创建之后不可更改**。

创建命名空间

创建命名空间

更新微服务命名空间

1. 在对应命名空间的右侧操作栏下点击 **编辑** ，进入更新页面。

更新命名空间

更新命名空间

2. 修改命名空间的名称，点击 **确定** 。

更新命名空间

更新命名空间

删除微服务命名空间

在对应命名空间的右侧操作栏下点击 **删除** ，然后在弹框中点击 **立即删除** 。

!!! note
删除命名空间之前，必须清理该命名空间下的所有资源，否则无法删除命名空间。

删除命名空间

删除命名空间

微服务列表

微服务列表页面列出了当前注册中心实例下的所有微服务，可以查看微服务的分组、健康状况、保护阈值、请求状况、链路追踪、治理状况等。在列表中点击微服务名称后，还可以进一步查看微服务实例列表、监控信息、接口列表、元数据等。

微服务列表

微服务列表

分组：指微服务的配置文件分组。将配置文件和和微服务实例互相分离，便于为不同的服务或组件应用相同的配置。

保护阈值：保护阈值是 0 到 1 之间的一个浮点数，表示集群中健康实例占比的最小值。如果实际健康实例的占比小于或等于该阈值时，就会触发阈值保护，即无论实例 (Instance) 是否健康，都会将这个实例 (Instance) 返回给客户端。阈值保护主要是为了防止故障实例过多时所有流量全部流入剩余实例，造成流量压力将剩余实例被压垮形成雪崩效应。

链路追踪：通过应用工作台创建服务时，可以选择否接入 OpenTelemetry 链路追踪组件。

是否可以治理：判断该微服务是否满足治理条件，例如是否开启了治理插件、是否正确配置插件信息、是否被网格纳管等要求。

查看微服务详情

在服务列表页面点击微服务名称可以查看服务详情，进一步查看实例列表、订阅者、监控、接口列表、元数据、服务治理等信息。

实例列表

首先需要进入目标注册中心，在左侧导航栏点击 **微服务列表** ，点击目标微服务的名称，进入微服务详情页面后方可执行后续操作。

进入微服务列表

进入微服务列表

- 服务流量权重的调整

  提供流量权重控制的能力，同时开放服务流量的阈值保护，帮助用户更好的地保护服务服务提供者集群不被意外打垮。可以点击实例的编辑按钮，修改实例的权重。如果想增加实例的流量，可以将权重调大，如果不想实例接收流量，则可以将权重设为 0。

  调整权重

  调整权重

- 实例上下线

  提供服务实例的上下线操作，在实例列表的操作列，可以点击实例的上线或者下线按钮。实例下线后状态会变为离线且不会被服务发现。

  上下线

  上下线

- 实例详情

  点击实例名称，**可**进入实例详情，查看实例的监控和元数据。

  实例详情

  实例详情

  - 实例监控

    实例监控功能用于监控服务实例的状态，**例如服**务实例的请求数，错误率、**响**应耗时、请求率、CPU **指**标、**内存指**标、读写速率、**接收**发送速率等指标的时序曲线。

    响应耗时中 p95 **代表**线上 95% **的**请求耗时都小于某个时间。

    实例监控

    实例监控

  - 实例元数据

    实例元数据

    实例元数据

订阅者

进入订阅者列表页面，**可**查看到订阅者的信息，**包括** IP **和端口**、**客**户端版本和应用名。

订阅者

订阅者

服务监控

服务监控用于查看特定时间范围内，**某个命名空**间下微服务的运行状态，**并且根据微服**务

的监控指标（请求数、错误率、响应耗时、请求率）初步判断是否出现异常。

在监控图表的右上角可以更改监控数据的时间范围。

监控

监控

微服务元数据

提供多个维度的服务元数据，帮助用户存储自定义的信息。这些信息都是以 Key-Value 的数据结构存储，在控制台上会以 k1=v1,k2=v2 这样的格式展示。

在右侧点击 编辑 可以修改元数据。

元数据

元数据

微服务治理

开启微服务治理插件后，可以通过 YAML 文件或页面表单为服务创建虚拟服务、目标规则、网关规则等三种治理规则。有关微服务治理的更多说明，可参考流量治理。

微服务治理

微服务治理

接口文档

接口文档显示服务对外暴露的 API 列表。点击右侧的导入接口文档可以手动录入 API，有地址导入和手工导入两种方式导入方式。

接口文档

接口文档

规则介绍

微服务引擎支持通过服务网格或 Sentinel 治理东西向流量。

Sentinel 治理规则

- 流控规则

  流控规则的原理是监控应用或服务流量的 QPS 指标，当指标达到阈值时根据预先设定的规则对请求流量进行控制，防止应用因短时间内无法处理过多流量而崩溃。使用流控规则后，系统可以在接下来的空闲期间逐渐处理堆积的请求，当指标重新恢复到阈值以下后，恢复正常的流量请求控制。

- 熔断降级

  在分布式系统中，各个服务通常需要调用其他的内部或外部服务才能正常运行，如果被调用的服务不够稳定，那么级联效应会导致调用者自身的响应时间也变长，产生线程堆积甚至导致服务不可用。为了避免这种情况出现，需要通过熔断机制根据预设的规则切断不稳定的调用链路，或者对下游服务进行降级，保护系统的整体可用性。

- 热点规则

  热点指经常被访问的数据。设置热点规则时，需要配置热点参数（即需要统计访问量的目标参数），然后系统会统计对该热点参数的请求量，当达到一定的阈值后，包含该热点参数的资源就是被限制调用。热点规则适用于统计被频繁访问的资源，达到一定的阈值后限制对该资源的访问。

- 系统规则

系统规则是指，由 Sentinel 综合系统容量、CPU 使用率、平均响应时间、入口 QPS 等数据，从整体维度出发自动选择流控规则控制请求流量。和其他规则不同的是，系统规则针对的是应用级别的入口流量，即仅对进入应用的流量生效，而其他规则通常是针对资源维度进行控制。

- 授权规则

  授权规则允许基于请求来源进行流量治理，例如仅放行白名单中的调用方发起的请求，不放行黑名单中的调用方发起的请求。

流控规则

Mesh 治理规则

流量治理为用户提供了三种资源配置，虚拟服务、目标规则、网关规则。 通过配置相应规则可以实现路由、重定向、熔断、分流等多项流量治理功能。用户可以通过向导或 YAML 形式创建、编辑治理策略。

- 虚拟服务主要用于对请求流量的路由定制规则，并可以对数据流做出分流、重定向、超时返回等处理。
- 目标规则则更关注流量本身的治理，为请求流量提供更强大的负载均衡、连接存活探寻、熔断等功能。
- 网关规则为 Istio 网关提供服务在网关的暴露方式。

有关这三项治理规则的详细说明，可参考文档流量治理。

虚拟服务 目标规则 网关规则

链路查询](../../../../insight/user-guide/trace/trace.md)。

资源信息

微服务引擎支持查看托管注册中心自身实例的运行状况、部署位置及运行日志。

- 与当前注册配置中心同名的工作负载就是该注册配置中心自身。
- 如果启用了 Sentinel 治理能力，则会出现另外一个实例，代表对应的 Sentinel 实例。

资源信息

资源信息

插件中心

插件中心提供 Sentinel 治理和 Mesh 治理两种插件，支持通过用户界面实现可视化配置。

安装插件后可以扩展微服务治理能力，满足不同场景下的业务诉求。

!!! info
同一个注册中心实例不能同时开启这两种插件，但可以根据不同的场景进行切换。

插件中心

插件中心

Sentinel 治理

Sentinel 插件主要适用于传统微服务的治理场景，支持流控规则、熔断规则、热点规则、系统规则、授权规则等多种治理规则。

开启 Sentinel 治理插件后会创建一个 Sentinel 实例，需要为其设置资源配额、选择部署模式（单节点/高可用）和访问方式。

Mesh 治理

Mesh 插件主要适用于云原生微服务的治理场景，提供虚拟服务、目标规则、网关规则、对等认证、请求身份认证、授权策略等治理规则。

开启 Mesh 治理插件需绑定一个网格实例，将微服务加入到网格中，并根据服务网格的要求配置边车等资源。

启用 Sentinel 治理插件

Sentinel 是面向分布式、多语言异构化服务架构的流量治理组件，主要以流量为切入点，从流量路由、流量控制、流量整形、熔断降级、系统自适应过载保护、热点流量防护等多个维度来帮助开发者保障微服务的稳定性。

操作步骤

1. 进入 微服务引擎 -> 传统微服务 -> 注册配置中心 模块，点击目标注册中心的名称。

   插件中心

   插件中心

2. 在左侧导航栏点击 插件中心 ，在 Sentinel 治理 卡片上点击 立即开启 。

   开启插件

   开启插件

3. 填写各项配置信息，然后在弹框底部点击 确定 。

   配置

配置

4. 如果满足前提条件并且配置正确，页面右上角会弹出”启用 Sentinel 插件成功”的消息。

配置

配置

启用 Mesh 治理插件

参照以下步骤启用 Mesh 治理插件：

1. 进入 微服务引擎 -> 传统微服务 -> 注册配置中心 模块，点击目标注册中心的名称。

插件中心

插件中心

2. 在左侧导航栏点击 插件中心 ，在 Mesh 治理 卡片上点击 立即开启 。

开启插件

开启插件

3. 选择想要绑定的服务网格，然后在弹框底部点击 确定 。

如果找不到想要的服务网格，可以去服务网格模块创建一个网格。

配置

配置

4. 如果满足前提条件并且配置正确，页面右上角会弹出”启用 Mesh 插件成功”的消息。

配置

配置

接入注册中心

注册中心支持接入 Nacos 注册中心、Eureka 注册中心、 Zookeeper 注册中心、Consul 注册中心、Kubernetes 注册中心、 Mesh 注册中心。

相对于托管型注册中心而言，接入型注册中心只支持一些基础操作，例如查看基本信息、监控信息等。 如需体验更高级更全面的管理操作，需要创建托管注册中心。

接入注册中心

接入注册中心的步骤如下：

1. 在左侧导航栏点击 传统微服务 -> 接入注册中心 ，然后在页面右上角点击 接入注册中心 。

   进入接入注册中心页面

   进入接入注册中心页面

2. 填写配置信息，然后在页面底部点击 确定 。

   接入不同类型的注册中心需要填写不同的配置信息。

   - Kubernetes/Mesh 注册中心：直接选择想要接入的集群或网格服务。
     - 如果找不到想要添加的 Kubernetes 集群，可以去容器管理模块接入集群或创建集群。
     - 如果找不到想要添加的网格服务，可以去网格服务模块创建网格。

       接入 Mesh/Kubernetes

       接入 Mesh/Kubernetes

   - Nacos/Zookeeper/Eureka/Consul 注册中心：填写注册中心的名称和地址，点

击 **确认** ，同时支持服务认证。

接入 Nacos/Zookeeper/Eureka

更新注册中心

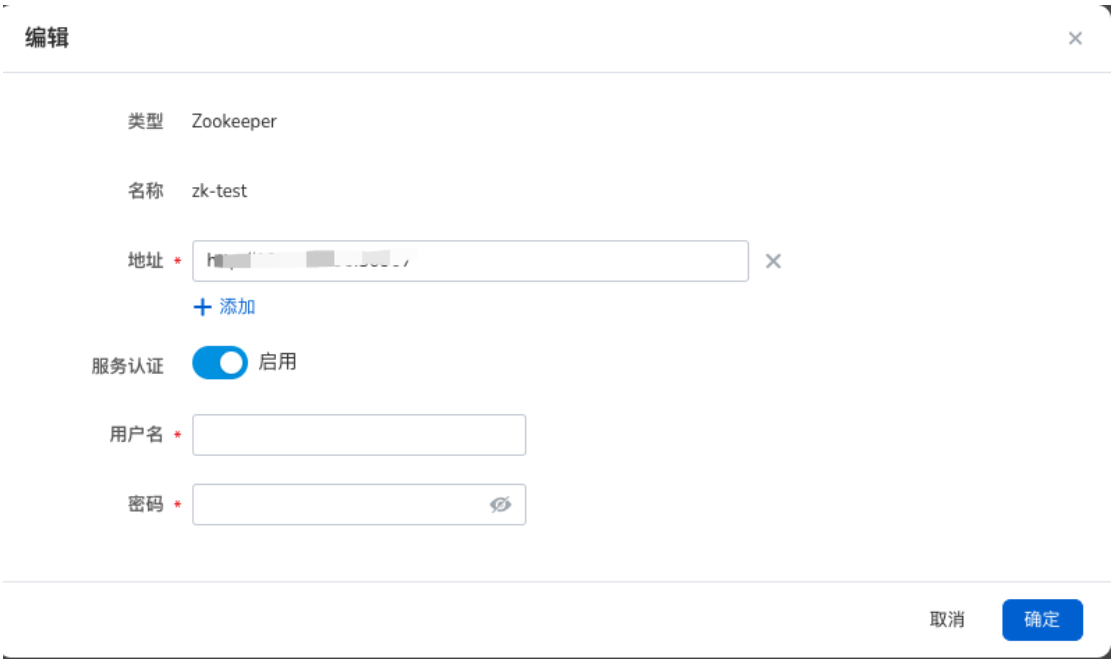
微服务引擎目前仅支持更新 Nacos/Zookeeper/Eureka **注册中心的配置**。

1. 在 **接入注册中心列表** 页选择需要更新的注册中心，在右侧点击 并选择 **编辑** 。

   进入更新页面

进入更新页面

2. 修改、增加或删除注册中心的地址，然后在页面底部点击 **确定** 。



进入更新页面

!!! note
如需更新 Kubernetes/Mesh 注册中心：

- 可以先[移除已经接入的注册中心](#_4)，然后再重新接入其他的注册中心。
- 也可以去容器管理模块[更新对应的集群](../../../kpanda/user-guide/clusters/upgrade-cluster.md)，或者去服务网格模块[更新对应的网格服务](../../../mspider/user-guide/service-mesh/index.md)。

移除注册中心

1. 在 **接入注册中心列表** 页选择需要移除的注册中心，**在右**侧点击 **并**选择 **移除** 。

进入移除页面

进入移除页面

2. 输入注册中心的名称，**点**击 **移除** 。

进入移除页面

进入移除页面

!!! note
接入型的注册中心仅支持`移除`操作，而托管型的注册中心仅支持`删除`操作。二者的区别在于：

- 移除：只是将注册中心从 DCE 5.0 的微服务引擎中移除，不会删除原有的注册中心和数据

，后续还可以再次接入该注册中心。
- 删除：删除注册中心及其中的所有数据，后续无法再次使用该注册中心，需要重新创建新的注册中心。

微服务管理

接入注册中心之后，可以通过注册中心对其中的微服务进行管理。微服务管理主要指查看注册中心下的微服务，

!!! note
接入型注册中心仅支持基础的管理操作。对于更复杂的管理场景，建议创建[托管注册中心](../hosted/index.md)以便执行更多高级操作。

1. 在 **接入注册中心列表** 页面点击目标注册中心的名称。

   点击注册中心名称

   点击注册中心名称

2. 在左侧导航栏点击 **微服务管理** ，查看微服务列表和基本信息。

   在当前页面，可以复制微服务的名称，可以查看当前注册中心下的所有微服务，以及各个微服务的所属命名空间、实例情况、请求统计数据等。

   点击注册中心名称

   点击注册中心名称

3. 点击微服务的名称，查看微服务的实例列表、**接口列表**、监控信息等。

   点击注册中心名称

   点击注册中心名称

   - 实例列表：查看实例状态、IP 地址、服务端口等。

     点击实例名称，还可以进一步查看该实例的监控信息和元数据。

     实例详情

     实例详情

- **接口列表**：查看微服务已经具有的接口，**或者**创建新的接口。

  实例详情

  实例详情

- 监控信息：查看微服务的监控信息，**包括**请求数、错误率、**响**应耗时、请求**率等**。

  支持自定义时间范围。

  实例详情

  实例详情

链路追踪

DCE 5.0 **支持服**务级别的链路追踪与查询，查看各个微服务的链路时延分布情况，**在出**现**故障**时可以快速定位故障。

更多详情，**可参考**可观测性 -> 链路查询。

插件中心介绍

插件中心是针对云原生网关和云原生微服务的统一插件管理门户，**其中涉及多种流控**，权**限及自定**义插件， **一次配置后即可在云原生网关和云原生微服**务中进行使用，**避免了相同**业务逻辑需求下的插件重复开发，为开发和运维人员带来了更多便利。

微服务引擎支持接入 Auth、JWT、**全局限流**、Wasm、ExtProc **插件**。 **只有先在插件中心接入插件，然后才能在云原生微服务或云原生网关中使用这些插件** 。

- JWT（JSON Web Token）**是一种用于身份**验证和授权的开放标准。**它由三部分**组成：

头部、载荷和签名。JWT 可以在客户端和服务器之间传递信息，并使用签名进行验证。 在云计算和云原生行业中，JWT 通常用于在微服务架构中进行身份验证和授权。

- Auth（ Authorization ）是一种用于验证用户身份和授权访问的插件。 它可以与 JWT 一起使用，对请求进行身份验证，并根据用户的权限决定是否允许访问特定资源。Auth 插件可以确保只有经过身份验证的用户才能访问特定的 API 或服务。
- Rate Limit（ 全局限流 ）是一种用于限制请求速率的插件。在云计算和云原生环境中，由于大量的请求可能会对系统造成负载压力，Rate Limit 插件可以帮助限制每个用户或 IP 地址的请求速率，以确保系统的稳定性和安全性。
- Wasm（ WebAssembly ）是一种低级别的字节码格式，可以在浏览器中运行高性能的编译代码。 在云计算和云原生行业中，Wasm 可以用作插件的运行环境，使开发人员能够使用更多的编程语言和工具来扩展和定制云服务。 通过使用 Wasm 插件，可以实现更高效、可扩展和灵活的云原生应用程序。
- ExtProc（ External Process ）是一种用于调用外部进程的插件。ExtProc 插件可以将请求和响应转发给外部进程， 由外部进程处理请求并返回结果。可以对请求进行额外处理，例如修改请求头、请求体、响应头、响应体等，或者根据请求的信息进行额外的自定义逻辑处理。

plugin list

plugin list

在云原生微服务中使用 JWT 插件

DCE 5.0 微服务引擎支持在云原生微服务中使用 JWT 插件，为服务添加安全认证。

前提条件

为了在云原生微服务中使用 JWT 插件，需要先满足下列前提条件。

创建服务网格

首先需要在 DCE 5.0 的服务网格模块中，为目标服务所在的集群创建服务网格。 目前支持创建托管网格、专有网格、外接网格三种类型的网格。

有关具体网格的创建步骤，可参考创建托管/专有网格或创建外接网格。

add-mesh

add-mesh

开启边车注入

在目标集群中部署应用，并在服务网格中为演示应用开启边车注入。

add-mesh

add-mesh

!!! note

- 由于此次演示使用的服务是 opentelemetry-demo-lite，所以使用 Helm 模板的形式部署应用。
- 如您使用其他服务，可以通过[创建工作负载](../../../kpanda/user-guide/workloads/create-deployment.md)或
  [Helm 模板](../../../kpanda/user-guide/helm/helm-app.md)或其他方式在集群中部署应用。

为网格绑定工作空间

为目标网格绑定一个工作空间，使得网格可以使用对应工作空间内的资源。

add-mesh

add-mesh

在微服务引擎中接入服务

进入微服务引擎模块，将演示应用导入云原生微服务模块，具体步骤可参考导入服务。

add-mesh

add-mesh

接入并绑定插件

满足上述前提条件之后，**接入** JWT **插件并将插件**绑定到微服务端口，**就可以在微服**务中**使用**该插件。

1. 进入微服务引擎，**在左**侧导航栏点击 **插件中心** ，**然后在**页面右上角点击 **接入插件** 。

   add-mesh

   add-mesh

2. **填写插件配置**，**最后在**页面右下角点击 **确定** **即可**。。

   add-mesh

   add-mesh

3. **在微服**务引擎的左侧导航栏点击 **云原生微服务** ，**然后点**击目标服务的名称。

   add-mesh

   add-mesh

4. **在服**务详情中点击 **插件能力** 页签，**然后在右**侧点击 **插件能力** 。

   add-mesh

   add-mesh

5. **填写插件配置**，**将插件和服**务端口绑定。

   add-mesh

   add-mesh

验证插件效果

完成前述操作后，**接下来通**过调用服务来验证插件效果。**按照**预期，**此**时访问服务时必须**要添加**认证请求头才能成功访问，**否**则无法访问服务。

1. **如果不**带请求头访问服务，**返回** missing，**服**务无法访问，**符合**预期效果。

   add-mesh

   add-mesh

2. 带上认证请求头再次访问，**此**时可以正常返回接口内容，**符合**预期效果。

   add-mesh

   add-mesh

微服务网关的 JWT 功能

微服务引擎网关支持 JWT 验证。下面介绍如何使用此功能。

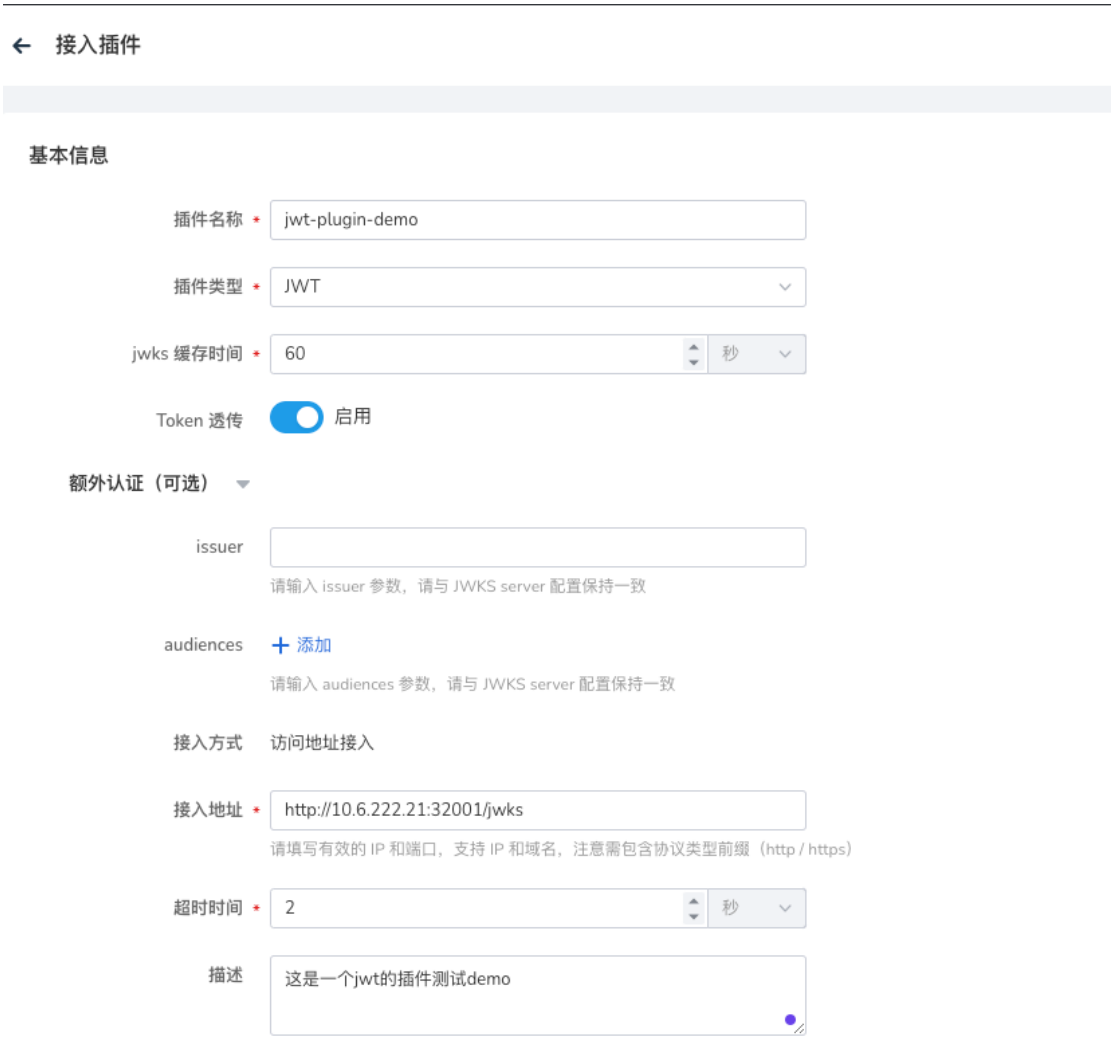
前提条件

- 创建一个集群或接入一个集群
- 创建一个网关
- 准备一个 Token 和用于验证 Token 的 JWKS 应用。如果尚且没有 JWKS 应用，可以参考创建 JWKS 应用创建一个。

操作步骤

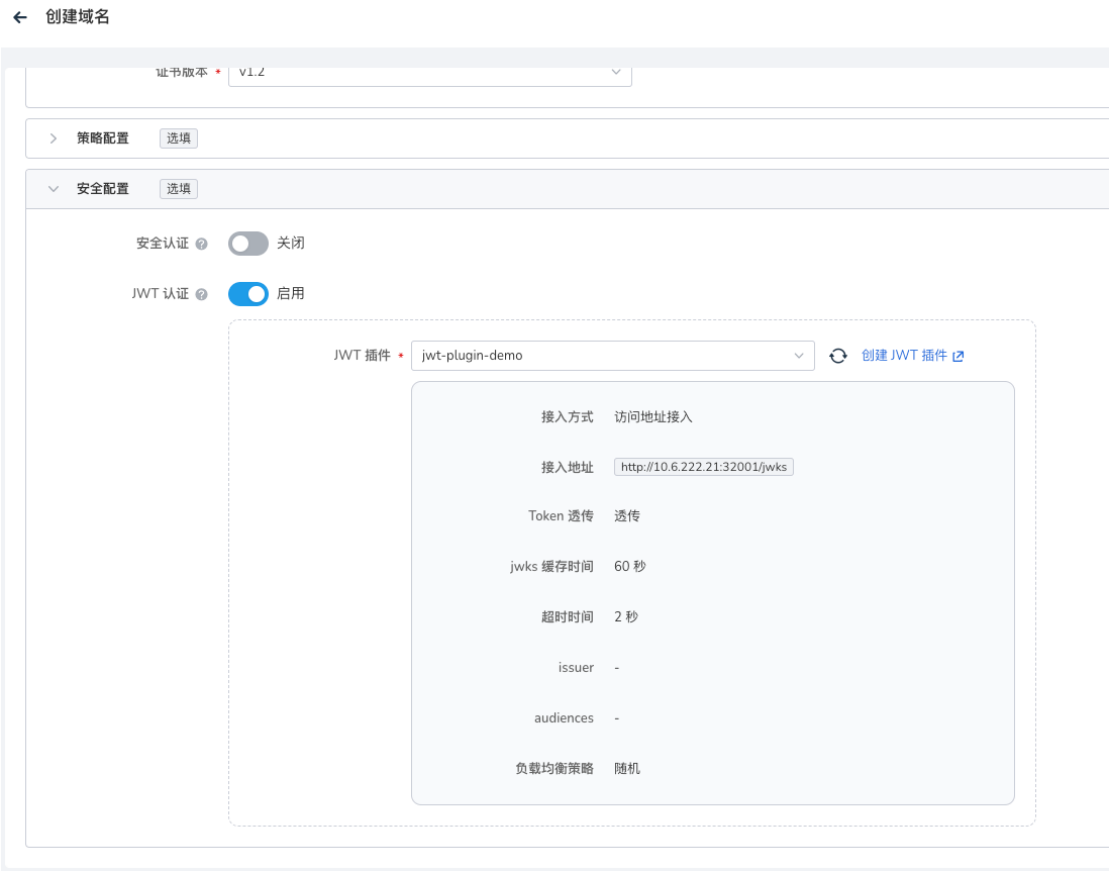
1. 在插件中心接入一个 JWT 插件
   - 插件名称：唯一的 JWKS 的名称，用于标识具体的 JWT 策略，必填
   - 插件类型：选择 JWT
   - JWKS 缓存时间：JWKS 在内存中的缓存时间，在缓存有效期内不会重复请求 JWKS 服务器地址
   - Token 透传：是否将 JWT 的 Token 信息发送到后端服务
   - JWKS 缓存时间：返回 JWKS 内容的 JWT 服务的完整 FQDN 地址，必填
   - Issuer：Token 颁发者认证，不填则不进行校验
   - Audiences：Token 的受众，不填则不进行校验
   - 接入地址：返回 JWKS 内容的 JWT 服务的完整 FQDN 地址，必填
   - 超时时间：JWKS 服务器的响应超时时间，超过超时时间获取 JWKS 失败

- 描述：插件的描述信息



JWT 插件

2. 参考创建域名创建协议为 https 的域名，在域名的安全策略中启用 JWT 认证， 并选择上一步创建的 JWT 插件。下方会回显出选择的 JWT 插件的配置信息。



域名选择插件

3. 参考添加 API 创建 API 并选择刚刚使用了 JWT 插件的域名。 默认情况下如果 API 使用的域名使用了 JWT 插件，API 也会启用 JWT 认证，但是在 API 级别可以禁用当前 API 的 JWT 认证； 如果 API 使用的域名未使用 JWT 插件，API 级别不可以开启 JWT 的认证。

API 选择插件 **域名启用**

API 选择插件 **域名未启用启用**

4. 带上 Token 访问验证，**如果**访问成功说明 JWT **策略配置成功**

基础配置

基础配置

创建 JWKS 应用

如果当前环境中尚没有 JWKS 应用，可以参考以下流程部署一个应用。

1. 将 JWKS 生成器代码下载到本地。

   ```
   git clone https://github.com/projectsesame/jwks-generator
   ```

2. 在本地运行 JWKS 生成器。

   ```
   mvn package -DskipTests && java -jar target/ROOT.war
   ```

   访问 http://localhost:8080，如果出现下方界面说明 JWKS 生成器已经在本地成功运行起来。

基础配置

基础配置

3. 参考下方说明填写信息，点击 **Generate** 生成 JWKS 内容。

   - KeySize：生成 secret 的大小，输入 256
   - KeyUse：用途，选择签名
   - Algorithm：算法，选择 HS256
   - KeyID：可选项，JWKS 有多个值时的匹配参数

基础配置

基础配置

4. 复制上图中 k 字段的取值，访问 <https: //jwt.io>，生成 Token。

   - 算法选择 HS256
   - 将复制好的 k 值粘贴到 secret 里面，并勾选 secret base64 encoded

基础配置

基础配置

5. 基于 YAML 模板创建 YAML 文件，然后使用 kubectl apply 命令安装 JWKS 应用：

   - 将 namespace 修改为网关所在的命名空间，在本例中使用 envoy-yang
   - 将 jwks.json 修改为上述第三步生成的 JWKS 内容

基础配置

基础配置

??? note “点击查看本例中配置的 YAML 文件”

```yaml title="all-in-one.yaml"
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: remote-jwks-go
  name: remote-jwks-go
  namespace: envoy-yang
spec:
  selector:
    matchLabels:
      app: remote-jwks-go
  template:
    metadata:
      labels:
        app: remote-jwks-go
    spec:
      containers:
        - args:
            - jwks
            - -c
            - /app/jwks.json
          command:
            - main
          image: release-ci.daocloud.io/skoala/demo/remote-jwks-go:0.1.0
          imagePullPolicy: IfNotPresent
          name: remote-jwks-go
          ports:
```

```
            - containerPort: 8080
              name: http
              protocol: TCP
          volumeMounts:
            - name: config
              mountPath: /app/jwks.json
              subPath: jwks.json
      volumes:
        - name: config
          configMap:
            name: jwks-config
      restartPolicy: Always
      securityContext:
        runAsNonRoot: true
        runAsUser: 65534
        runAsGroup: 65534

---
apiVersion: v1
kind: Service
metadata:
  name: remote-jwks-go
  namespace: envoy-yang
  labels:
    app: remote-jwks-go
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app: remote-jwks-go

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: jwks-config
  namespace: envoy-yang
  labels:
    app: remote-jwks-go
data:
```

```
    jwks.json: |+
      {
        "keys": [
          {
          "kty": "oct",
          "use": "sig",
          "k": "veb4HPc6oaEAsCikZ7rzTKmu9LkOU4LpDUKBxFjnBcc",
          "alg": "HS256"
      }
        ]
      }
```

6. 访问应用的 8080 端口，出现 success 说明应用安装成功。

   JWKS 地址应为 网关访问地址/jwks 构成，例如 http://13.5.245.34:31456/jwks

   在微服务引擎的网关概览页面可以查看网关的访问地址。

   基础配置

   基础配置

微服务网关接入认证服务器

微服务网关支持接入第三方认证服务器。

前提条件

- 创建一个集群或接入一个集群
- 创建一个网关

选用认证服务器

默认的认证服务器

1. 将认证服务器的代码模板克隆到本地。

   git clone https://github.com/projectsesame/envoy-authz-java

2. 直接使用 envoy-authz-java.yaml 以及文件下的默认镜像。

   kubectl apply -f envoy-authz-java.yaml

   默认镜像是 release.daocloud.io/skoala/demo/envoy-authz-java:0.1.0

3. 模板为简单的路径判断，当访问路径为 / 时通过认证，其余路径为拒绝访问。

自定义的认证服务器

1. 将认证服务器的代码模板克隆到本地。

   git clone https://github.com/projectsesame/envoy-authz-java

   该项目分为两个子模块：

   - API 模块是 Envoy 的 protobuf 文件的定义（无需修改）
   - authz-grpc-server 模块是认证服务器的认证逻辑处理地址（在这里填写认证逻辑）
   - release.daocloud.io/skoala/demo/envoy-authz-java:0.1.0

2. 使用如下命令编译 API 模块，解决找不到的问题

   mvn clean package

3. 成功编译之后，在 check 方法中编写自定义的认证逻辑。

   - check 方法位于 envoy-authz-java/authz-grpc-server/src/main/java/envoy/projectsesame/io/authzgrpcserver/AuthzService.java
   - 模板为简单的路径判断，当访问路径为 / 时通过认证，其余路径为拒绝访问。

4. 代码编写完成之后，使用 Docker 打包镜像。

   代码模板仓库中已存在 Dockerfile 文件，可以直接使用该模板构建镜像。

5. 将镜像地址填入 envoy-authz-java.yaml 文件中的 Deployment 下的 spec/template/spec/containers/image 字段。

   填写镜像

填写镜像

接入认证服务器

1. 在网关所在的集群内创建以下资源。使用 kubectl apply 命令基于 envoy-authz-java.yaml 文件可以一次性快速创建下述三项资源：
   - 认证服务器的 Deployment
   - 认证服务器的 Service
   - 认证服务器的 ExtensionService
2. 在插件中心接入一个 Auth 插件

   接入地址填写步骤 1 部署的应用外部访问地址，注意该应用的访问协议为 GRPC。

   Auth 插件

Auth 插件

配置认证服务器

在网关层面配置

!!! note
HTTP 和 HTTPS 域名都支持安全认证，如需使用 HTTPS 域名，网关需要开启 HTTPS。

1. 网关配置认证服务器。

网关配置认证服务器

网关配置认证服务器

2. 创建 HTTP 或者 HTTPS 域名，以 HTTP 域名为例，此时创建的域名都是默认开启安全认证的，并且无法关闭。

网关域名认证服务器

网关域名认证服务器

3. 在网关下创建一个 API，关联域名填写刚才新创建的域名，匹配路径为 /，并将 API 上线。API 默认状态是应用域名的安全认证配置，也可以自定义插件的生效与否和附加参数。

网关 API 认证服务器

网关 API 认证服务器

4. 现在即可通过认证服务器访问该 API 了。

   - 访问 /。

     curl -H 'header: true' http://gateway.test:30000/

     访问结果如下，可以看到请求通过了。

     adservice-springcloud: hello world!

   - 访问 /test1。

     curl -H 'header: true' http://gateway.test:30000/test1

     访问结果如下，可以看到请求被拦截了。

     No permission

在域名或 API 层面配置

!!! note
只有 HTTPS 域名支持安全认证，网关需要开启 HTTPS。

1. 创建 HTTPS 域名，并手动配置安全认证。

域名认证服务器

域名认证服务器

2. 在网关下创建一个 API，关联域名填写刚才新创建的域名，匹配路径为 /，并将 API 上线。API 默认状态是应用域名的安全认证配置，也可以自定义插件的生效与否和附加参数。

网关 API 认证服务器

网关 API 认证服务器

3. 现在即可通过认证服务器访问该 API 了。

   - 访问 /。

     curl -k -H 'header: true' https://gateway.test:30001/

     访问结果如下，可以看到请求通过了。

     adservice-springcloud: hello world!

   - 访问 /test1。

     curl -k -H 'header: true' https://gateway.test:30001/test1

     访问结果如下，可以看到请求被拦截了。

     No permission

微服务网关认证服务器 - 接入 AK/SK 认证

Demo 应用包含两个模块，分别是模拟签名应用和模拟认证应用，模拟签名应用用来生成对应的签名信息，模拟认证应用是基于 Envoy 的认证服务器的一个签名认证的实现。以下主要来讲述模拟签名应用的详细内容：

模拟签名应用

代码仓库地址：<https: //github.com/projectsesame/ak-sk-demo-java>

模拟签名应用分为三个接口，分别是：

1. /signstr：根据当前时间生成签名信息
2. /signstrbytime：根据传入的时间生成签名信息
3. /mock：根据当前时间生成签名信息并模拟通过网关访问请求。

这三个接口均需要使用 Post 请求，需要传入 Json 格式的 Body。具体字段如下表所示：

字段名称	字段含义	示例	备注
url	完整的请求网关路径	http://10.6.222.21:30080/yang?a=b	格式为：scheme://gatewayIP:gatewayPort/path?param
host	域名	10.6.222.21:30080	
method	请求方法	GET/POST	只支持 GET 和 POST
headers	请求头	{"User-Agent": "curl/8.1.2","Accept":"_/_","k":"v"}	请求网关时携带的请求头
signHeaders	需要进行签名的请求头	[“User-Agent”,“Accept”]	需要进行签名的请求头，会对在 headers 中查找 signHeaders 中的请求头

字段名称	字段含义	示例	备注
requestBody	请求体	this is request body	来进行签名
apiKey	请求认证的标识	api-key	请求认证的标识
secret	请求 apiKey 对应的密钥	secret	不需要填写，Demo 程序默认为 secret，可根据自己需要通过 apiKey 唯一获取到 secret 即可

Demo 应用会根据以上结构体进行签名，生成的签名字符串的结构为：

!!! info
x-data: 请求方法\n 请求路径\n 请求参数\n 请求时间\n 请求头\n 请求体

其中请求头按照字典序排序，多个请求头用逗号分隔，需要签名的请求头可以通过 `signHeaders` 字段指定，
不指定则只对 x-data 请求头进行签名，x-data 请求头为自定义请求头，为请求时间

如果存在请求体，请求体为 MD5 加密后再以 Base64 编码值

签名算法为根据 Secret 对 x-data 进行 hmac-sha1 加密，然后以 Base64 编码

签名认证信息字符串为：

- id=apiKey

- algorithm=签名算法
- headers=签名请求头
- signature=签名字符串

接口签名方式

1. 传入参数：

```
{
        "url":"http://10.6.222.21:30080/yang?a=b",
        "host":"10.6.222.21:30080",
        "apiKey": "key",
        "method":"POST",
        "headers":{"User-Agent":"curl/8.1.2","Accept":"*/*","k":"v"},
        "signHeaders":["User-Agent","Accept"],
        "requestBody":"hahha"
    }
```

2. 生成签名字符串：

```
x-data: POST
    /yang
    a=b
    1703573142130
    accept: */*
    user-agent: curl/8.1.2
    x-date: 1703573142130
    ODc5NWEzY2QyY2ExZjdmMTUzMGIzYmI0ZThiYWY2NTA=
```

3. 根据上一步生成的签名字符串和自定义获取的 Secret 进行加密，使用上述字符串根据 hmac-sha1 加密，然后用 Base64 编码，结果为：

```
SuRuXnwwgrv+0/TNbWQxkEIdnlA=
```

4. 生成签名认证信息：

```
id=key,algorithm=hmac-sha1,headers=User-Agent;Accept;x-date,signature=SuRuXnwwgrv+0/T
    NbWQxkEIdnlA=
```

示例代码接口返回结果

- 如果请求 /signstr 接口，会返回当前时间对应的签名信息，例如：

```
{
        "x-data": 1703573142130,
```

"authorization": "id=key,algorithm=hmac-sha1,headers=User-Agent;Accept;x-date,signature=SuRuXnwwgrv+0/TNbWQxkEIdnlA="
}

将返回的两个请求头加入想要访问网关的请求中。

- 如果请求 /signstrbytime 接口，需要携带签名时间的毫秒值，会返回参数时间对应的签名信息，例如传入的时间参数为 1703573152130，返回为：

{
"x-data": 1703573152130,
"authorization": "id=key,algorithm=hmac-sha1,headers=User-Agent;Accept;x-date,signature=8zJJS6DVoGxlwi1K4vrK0QcdwVg="
}

- 如果请求 /mock 接口，会以当前时间生成对应的签名字符串后模拟请求网关，网关地址为请求参数重填写的 URL。

以上是签名模拟应用的签名逻辑和过程，签名结束后需要和认证服务相结合才可以实现 ak/sk 的认证逻辑， 如果采用以上签名模拟应用的签名，我们提供了相匹配的认证服务，只需要网关接入对应的认证服务即可进行认证。

认证服务器

代码仓库地址：<https: //github.com/projectsesame/envoy-authz-ak-sk-java>

根据 envoy-auzhe-java-aksk.yaml 中的编排文件将其部署到网关所在的集群中，然后接入认证服务器的网关。

微服务网关认证服务器 - 接入 OIDC 认证

微服务引擎网关的 OIDC 认证服务 demo 包含两个部分。

1. 认证服务，代码仓库地址：<https: //github.com/projectsesame/contour-authserver>
2. 身份提供商（以下简称 IDP），代码仓库地址：<https:

//github.com/projectsesame/dex>

部署 ldp 服务，根据 dex-all-in-one.yaml 文件中的内容将其部署在 k8s 集群中。 环境变量中 DEX_ISSUER 地址为 ldp 服务的 nodeport 或 loadbalance 地址+ /dex，该服务必须暴露为对外访问； 环境变量中 DEX_STATIC_CLIENT_REDIRECT_URI 是允许客户端重定向的地址列表，格式为 http(s): //+域名+port+redirectPath， 其中 path 为稍后需要部署的认证服务的 redirectPath。 其他环境变量不建议修改，想要尝试的可以参考文件 config.docker.yaml 修改并重新构建镜像。 部署成功后可以开始部署认证服务，根据 auth-oidc-all-in-one.yaml 文件中的内容将其部署在 k8s 集群中， 其中 configMap 中的部分内容需要根据自身环境来配置，以下是各个配置项的含义：

参数名称	参数类型	参数含义	参数示例	备注
address	字符串	提供服务的地址	: 10083	表明任何地址均可以通过 10083 端口访问该服务端口需要和

参数名称	参数类型	参数含义	参数示例	备注
issuerURL	string	Identify Provider	http://10.6.222.22:30051/dex	service 相同填写 *IDP* 服务 nodePort 或 loadBalance 的地址 +/dex 示例中是通过 nodePort 方式访问
clientID	string	客户端 ID	example-app	客户端 id 和密钥示例应

参数名称	参数类型	参数含义	参数示例	备注
				用为写死的状态
clientSecret	string	客户端密钥	ZXhhbXBsZS1hcHAtc2VjcmV0	
scopes	[]string	访问权限范围	openid，profile，email，offline_access	
redirectURL	string	重定向回调地址	https://yangyang.daocloud.io:30443	示例中为https://+域名+https端口
redirectPath	string	重定向路径	/oauth2/callback	需要保证网关 api 配置中该路径也包含

参数名称	参数类型	参数含义	参数示例	备注
				在认证路径中

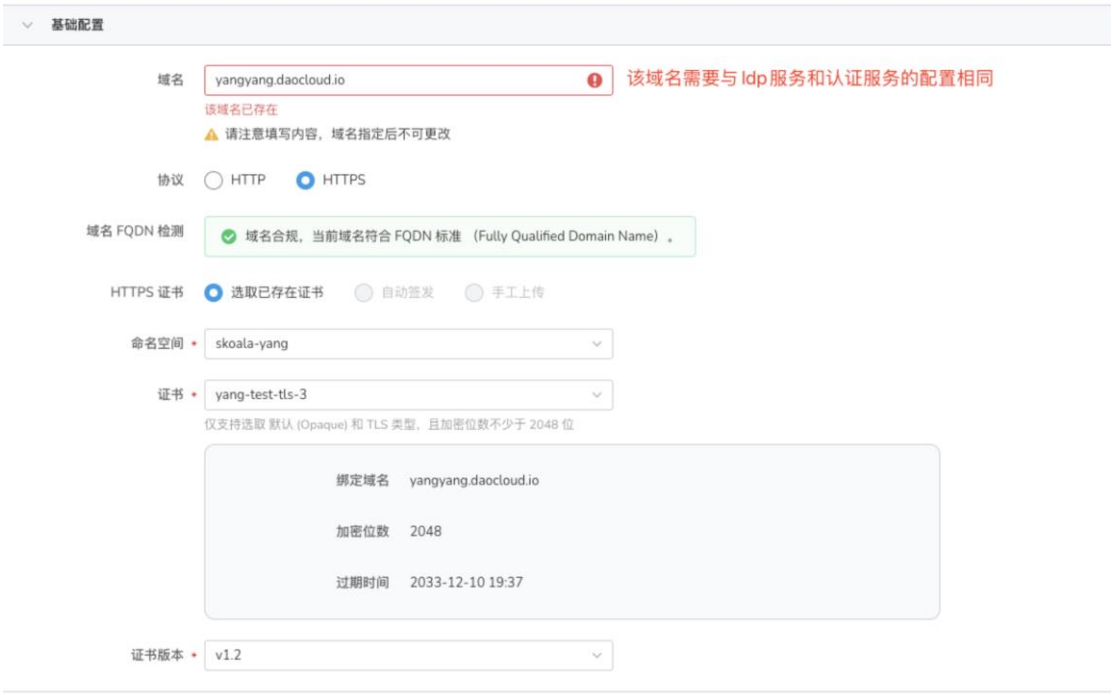
以上配置项在该 demo 的 issuerURL、redirectURL 中，这两个需要根据部署环境进行修改，其他参数不建议修改！以下文档均已上述示例配置的内容为例进行说明。

当这两项服务均部署成功后可以通过微服务引擎的网页进行后续配置。以下是配置步骤：

1. 创建插件

创建插件

2. 选择网关并创建域名，并选中刚刚第一步创建的安全认证插件



创建域名

安全认证插件

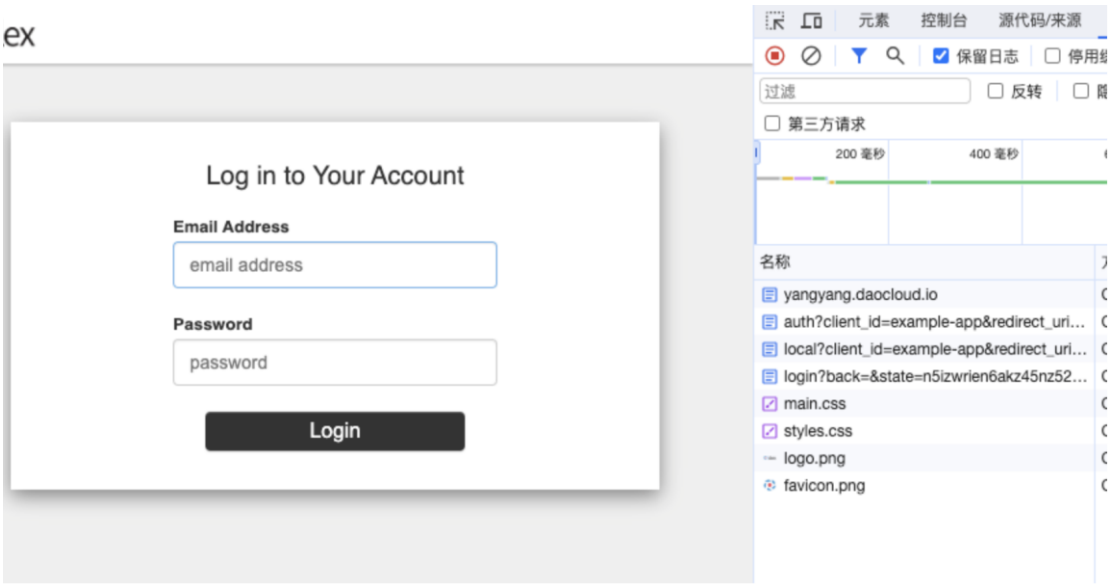
3. 创建 API，**并**选择刚刚创建的域名

创建 api

4. API 创建成功后通过通过浏览器访问访问刚才创建的 api

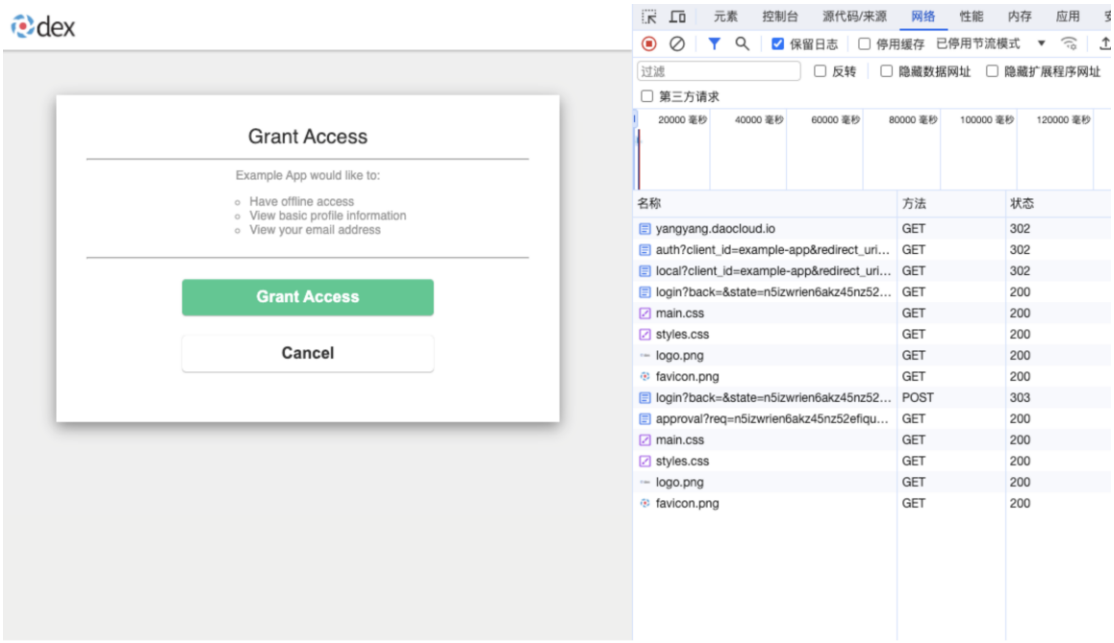
访问 https://yangyang.daocloud.io:30443/

由于我们对域名配置了认证服务，因为页面会跳转到 ldp 服务的登录页面。图片右侧显示了跳转的过程。 示例代码中用户名为：admin@example.com，密码为：password。



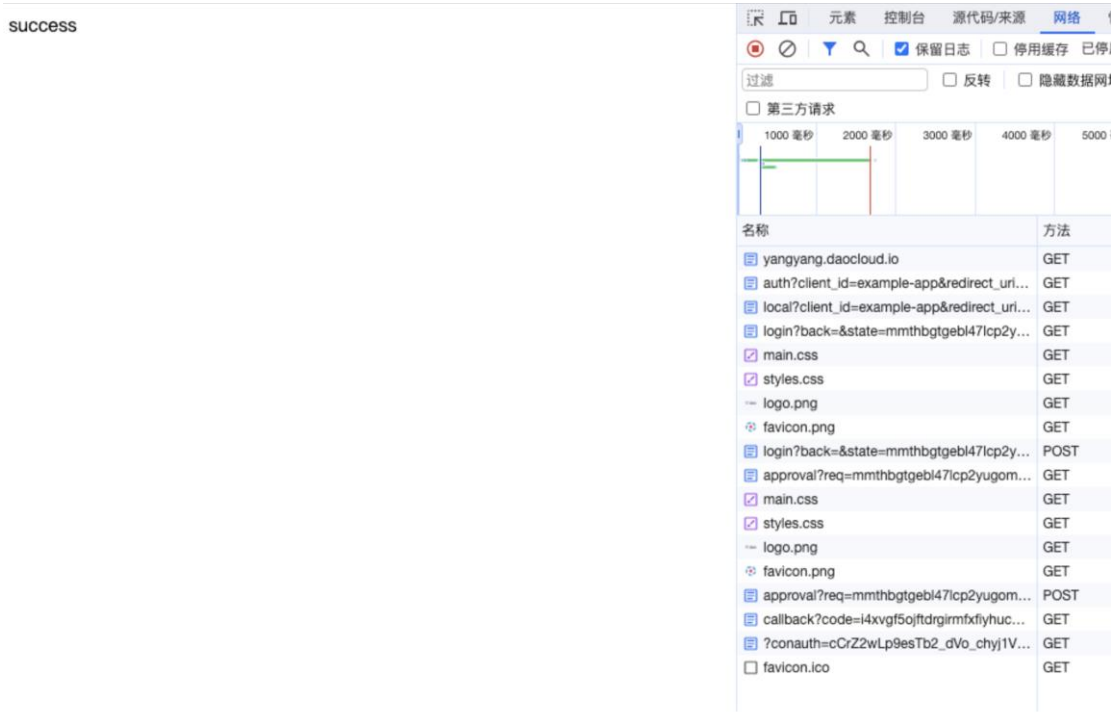
访问

登录后显示授权页面，点击 **Grant Access**



授权

经过一些跳转后可以正常访问后端接口：



正常访问

至此使用微服务引擎实现 OIDC 认证的完整逻辑已经完成。

微服务网关接入限流服务器

微服务网关支持接入第三方限流服务器，本文档演示使用默认的限流服务器的步骤。

前提条件

- 创建一个集群或接入一个集群
- 创建一个网关

选用限流服务器

你可以选择默认的限流服务器，也可以自己接入一个。

默认的限流服务器

直接应用提供的限流服务器模板，具体逻辑可参考限流服务器代码。

kubectl apply -f gateway-rls.yaml -n plugin-ns

??? note “默认的限流服务器”

```yaml title="gateway-rls.yaml"
---
# NOTE: this deployment is intended for demonstrating global
# rate limiting functionality only and should NOT be considered
# production-ready.

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: ratelimit
  name: gateway-rls
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      # This value of maxSurge means that during a rolling update
      # the new ReplicaSet will be created first.
      maxSurge: 50%
  selector:
    matchLabels:
      app: ratelimit
  template:
    metadata:
      labels:
        app: ratelimit
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchLabels:
                  app: ratelimit
              topologyKey: kubernetes.io/hostname
            weight: 100
```

```
containers:
  - name: redis
    image: release-ci.daocloud.io/skoala/redis:6.2.6
    env:
      - name: REDIS_SOCKET_TYPE
        value: tcp
      - name: REDIS_URL
        value: redis:6379
  - name: ratelimit
    image: release-ci.daocloud.io/skoala/envoy-ratelimit:v2   # latest a/o Mar 24 2022
    ports:
      - containerPort: 8080
        name: http
        protocol: TCP
      - containerPort: 8081
        name: grpc
        protocol: TCP
      - containerPort: 6070
        name: debug
        protocol: TCP
    volumeMounts:
      - name: ratelimit-config
        mountPath: /data/ratelimit/config
        readOnly: true
    env:
      - name: USE_STATSD
        value: "false"
      - name: LOG_LEVEL
        value: debug
      - name: REDIS_SOCKET_TYPE
        value: tcp
      - name: REDIS_URL
        value: localhost:6379
      - name: RUNTIME_ROOT
        value: /data
      - name: RUNTIME_SUBDIRECTORY
        value: ratelimit
      - name: RUNTIME_WATCH_ROOT
        value: "false"
      # need to set RUNTIME_IGNOREDOTFILES to true to avoid issues with
      # how Kubernetes mounts configmaps into pods.
      - name: RUNTIME_IGNOREDOTFILES
        value: "true"
    command: ["/bin/ratelimit"]
```

```
            livenessProbe:
              httpGet:
                path: /healthcheck
                port: 8080
              initialDelaySeconds: 5
              periodSeconds: 5
        volumes:
          - name: ratelimit-config
            configMap:
              name: gateway-rls

---
apiVersion: v1
kind: Service
metadata:
  name: gateway-rls
spec:
  ports:
  - port: 8081
    name: grpc
    protocol: TCP
  - port: 6070
    name: debug
    protocol: TCP
  selector:
    app: ratelimit
  type: NodePort

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: gateway-rls
data:
  ratelimit-config.yaml: |
    domain: gateway-rls.skoala-dev
    descriptors:
      - name: test1
        key: foo
        value: goo
        rate_limit:
          name: test1
          unit: Minute
          requests_per_unit: 20
```

```
      descriptors:
        - name: test2
          key: foo1
          value: goo1
          rate_limit:
            name: test2
            unit: Minute
            requests_per_unit: 15
          descriptors:
            - name: test3
              key: foo2
              value: goo2
              rate_limit:
                name: test3
                unit: Minute
                requests_per_unit: 10
```

接入限流服务器

1. 获取上述步骤部署的 gateway-rls **的外部**访问地址。

   kubectl get svc -n plugin-ns

   限流服务器地址: 10.6.222.21:32003

   限流服务器配置地址: http://10.6.222.21:32004

   ```
   NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
                                AGE
   gateway-rls             NodePort    10.233.56.164   <none>        8081:32003/TCP,6070:
   32004/TCP   1m
   ```

2. **在插件中心**创建全局限流插件。

   - **限流响**应头信息: **是否开启在响**应头中打印限流相关信息。
   - **快速成功**: **当限流服**务器无法正常访问时，**是否允**许继续访问请求。
   - **接入地址**: **限流服**务器的地址，8081 **端口**对应的地址，协议为 GRPC。
   - 负载均衡策略: **当存在多个限流服**务器时，**多个限流服**务器的访问策略。
   - **配置**获取接口: 获取限流服务器配置的地址，为端口 6070 对应的地址，协

议为 HTTP。

- 超时时间：限流服务器响应的超时时间。

RATELIMIT 插件

RATELIMIT 插件

3. 网关配置全局限流插件。

网关配置全局限流插件

网关配置全局限流插件

4. 创建域名并开启全局限流。

域名开启全局限流

域名开启全局限流

5. 在网关下创建一个 API，关联域名填写刚才新创建的域名，匹配路径为 /，并将 API 上线。API 默认状态是应用域名的全局限流配置，也可以自定义限流规则。

API 全局限流

API 全局限流

6. 现在即可通过限流服务器访问该 API 了。

```
while true; do curl -w " http_code: %{http_code}" http://gateway.demo:30000/; let count+=1; echo " count: ${count}"; done
```

访问结果如下，可以看到访问 10 次后，就被限流了。

```
adservice-springcloud: hello world! http_code: 200 count: 1
adservice-springcloud: hello world! http_code: 200 count: 2
adservice-springcloud: hello world! http_code: 200 count: 3
adservice-springcloud: hello world! http_code: 200 count: 4
adservice-springcloud: hello world! http_code: 200 count: 5
adservice-springcloud: hello world! http_code: 200 count: 6
adservice-springcloud: hello world! http_code: 200 count: 7
adservice-springcloud: hello world! http_code: 200 count: 8
adservice-springcloud: hello world! http_code: 200 count: 9
adservice-springcloud: hello world! http_code: 200 count: 10
```

```
http_code: 429 count: 11
http_code: 429 count: 12
http_code: 429 count: 13
http_code: 429 count: 14
http_code: 429 count: 15
http_code: 429 count: 16
http_code: 429 count: 17
http_code: 429 count: 18
http_code: 429 count: 19
http_code: 429 count: 20
...
```

基于 IP 的全局限流

!!! note
IP 限流规则的 Key 必须填写 remote_address。

针对所有 IP 进行限流

1. 编辑限流服务器的 configmap，在 descriptors 添加以下内容（注意格式）：

```
data:
    ratelimit-config.yaml: |
      domain: gateway-rls.test
      descriptors:
        - name: ip-rls
          key: remote_address
          rate_limit:
            name: ip-rule
            unit: Minute
            requests_per_unit: 5
```

2. 限流服务器会热加载配置，等待配置生效即可，当然，也可以访问限流服务器的配置接口，出现以下配置即可。

```
$ curl http://10.6.222.21:32004/rlconfig
  gateway-rls.test.remote_address: unit=MINUTE requests_per_unit=5, shadow_mode: false
```

3. 域名配置全局限流策略（当然，前提是网关需要开启全局限流插件）。

域名全局限流策略

域名全局限流策略

4. 基于该域名的 API 访问，执行以下命令访问。

```
while true; do curl -w " http_code: %{http_code}"  http://ip.test:30000/; let count+=1; ech
  o " count: ${count}"; done
```

访问结果如下，可以看到访问 5 次后，就被限流了。

```
adservice-springcloud: hello world! http_code: 200 count: 1
  adservice-springcloud: hello world! http_code: 200 count: 2
  adservice-springcloud: hello world! http_code: 200 count: 3
  adservice-springcloud: hello world! http_code: 200 count: 4
  adservice-springcloud: hello world! http_code: 200 count: 5
  http_code: 429 count: 6
  http_code: 429 count: 7
  http_code: 429 count: 8
  http_code: 429 count: 9
  http_code: 429 count: 10
  ...
```

针对指定 IP 进行限流

1. 编辑 gateway-rls 的 configmap，在 descriptions 添加以下内容（注意格式）：

   - 对所有 IP 进行限流，每分钟访问 10 次。
   - 对 IP 10.6.222.90 限流每分钟访问 5 次。
   - 对 IP 10.70.4.1（本机）限流每分钟访问 3 次。

```
data:
  ratelimit-config.yaml: |
    domain: gateway-rls.test
    descriptors:
      - name: ip-rls
        key: remote_address
        rate_limit:
          name: ip-rule
          unit: Minute
          requests_per_unit: 10
      - key: remote_address
        value: 10.6.222.90
        rate_limit:
          unit: Minute
          requests_per_unit: 5
      - key: remote_address
```

```
      value: 10.70.4.1
    rate_limit:
      unit: Minute
      requests_per_unit: 3
```

2. 限流服务器会热加载配置，等待配置生效即可，当然，也可以访问限流服务器的配置接口，出现以下配置即可。

```
$ curl http://10.6.222.21:32004/rlconfig
    gateway-rls.test.remote_address: unit=MINUTE requests_per_unit=10, shadow_mode: false
    gateway-rls.test.remote_address_10.6.222.90: unit=MINUTE requests_per_unit=5, shadow_mode: false
    gateway-rls.test.remote_address_10.70.4.1: unit=MINUTE requests_per_unit=3, shadow_mode: false
```

3. 域名配置全局限流策略。

4. 基于该域名的 API 访问，执行以下命令访问。

```
while true; do curl -w " http_code: %{http_code}"  http://ip.test:30000/; let count+=1; echo " count: ${count}"; done
```

在本机执行命令的访问结果如下，访问 3 次被限流。

```
adservice-springcloud: hello world! http_code: 200 count: 1
    adservice-springcloud: hello world! http_code: 200 count: 2
    adservice-springcloud: hello world! http_code: 200 count: 3
    http_code: 429 count: 4
    http_code: 429 count: 5
    http_code: 429 count: 6
    http_code: 429 count: 7
    http_code: 429 count: 8
    http_code: 429 count: 9
    http_code: 429 count: 10
    ...
```

在 10.6.222.90 主机执行命令的访问结果如下，访问 5 次被限流。

```
adservice-springcloud: hello world! http_code: 200 count: 1
    adservice-springcloud: hello world! http_code: 200 count: 2
    adservice-springcloud: hello world! http_code: 200 count: 3
    adservice-springcloud: hello world! http_code: 200 count: 4
    adservice-springcloud: hello world! http_code: 200 count: 5
    http_code: 429 count: 6
    http_code: 429 count: 7
    http_code: 429 count: 8
```

http_code: 429 count: 9
http_code: 429 count: 10
http_code: 429 count: 11
http_code: 429 count: 12
...

在其他未额外设置限流规则的主机执行命令的访问结果如下，访问 10 次被限流。

adservice-springcloud: hello world! http_code: 200 count: 1
adservice-springcloud: hello world! http_code: 200 count: 2
adservice-springcloud: hello world! http_code: 200 count: 3
adservice-springcloud: hello world! http_code: 200 count: 4
adservice-springcloud: hello world! http_code: 200 count: 5
adservice-springcloud: hello world! http_code: 200 count: 6
adservice-springcloud: hello world! http_code: 200 count: 7
adservice-springcloud: hello world! http_code: 200 count: 8
adservice-springcloud: hello world! http_code: 200 count: 9
adservice-springcloud: hello world! http_code: 200 count: 10
http_code: 429 count: 11
http_code: 429 count: 12
http_code: 429 count: 13
http_code: 429 count: 14
http_code: 429 count: 15
http_code: 429 count: 16
http_code: 429 count: 17
...

云原生自定义插件示例：envoy-extproc-caching-demo-go

Envoy-extproc-method-conv-demo-go 是一个基于 envoy-extproc-sdk-go 实现的，用以展示如何在 Go 语言中使用 Envoy 提供的 ext_proc 功能的示例。

功能

主要功能是使用一个非持久化存储来缓存由 Downstream 发起的 GET 请求的应答，针对某个 path 的第一次请求将由 Upstream 响应并被 Caching 缓存； 之后所有针对此 Path

的请求，将由 Caching 直接应答，而不再路由到 Upstream，直到 Caching 重启.以达到响应体缓存的目的。

前置条件

- 安装 Envoy (Version >= v1.29)
- 安装 Go (Verson >= v1.21) 如果只是运行，可跳过此步
- 支持 HTTP Method: GET 的目标服务(以下简称 Upstream)，且假设其支持以下 route：
  - /*
  - /no-extproc

编译

进入项目根目录（如果只是运行，可跳过此步）。

```
go build . -o extproc
```

运行

- Envoy：

  ```
  envoy -c ./envoy.yaml # (1)!
  ```

  1. 此文件位于项目根目录

- Caching：

  - 裸金属

    ```
    ./extproc caching --log-stream --log-phases
    ```

  - K8s

    ```
    kubectl apply -f ./deployment.yaml # (1)!
    ```

    1. 此文件位于项目根目录

- Curl：

  ```
  curl 127.0.0.1:8000/no-extproc # (1)!
  curl 127.0.0.1:8000/abc # (2)!
  ```

1. Caching 不会作用于此 route，每次请求都将由 Upstream 应答
2. 第一次请求将由 Upstream 应答，并被 Caching 缓存，后续针对 /abc 的请求将会由 Caching 直接应答

参数说明

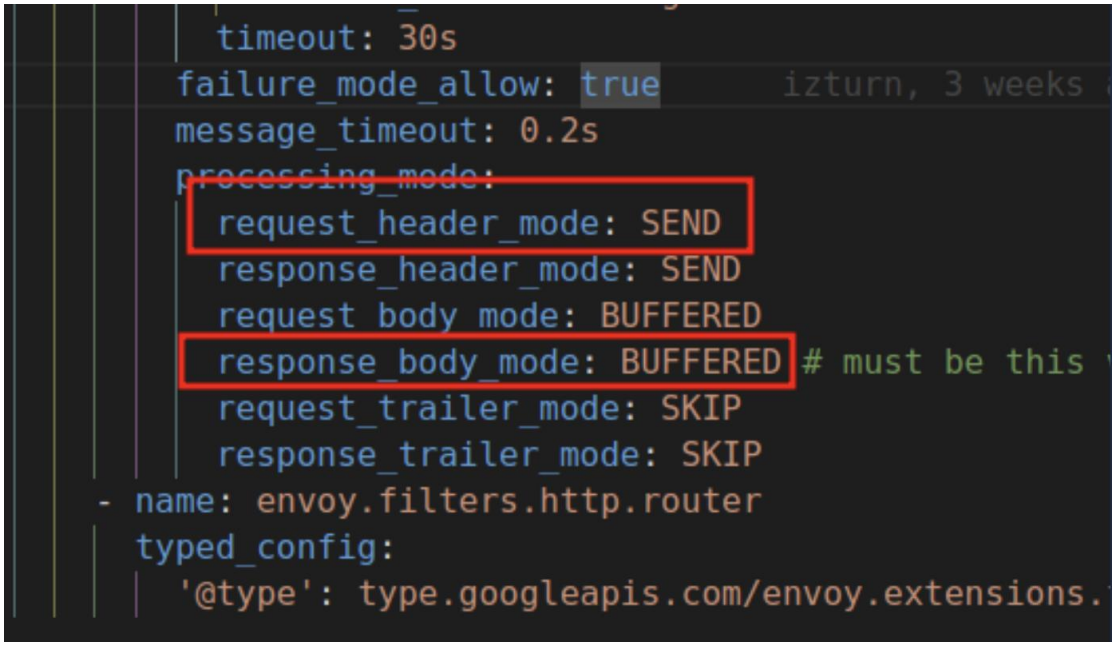
- log-stream：是否输出关于请求/响应流的日志
- log-phases：是否输出各处理阶段的日志
- update-extproc-header：是否在响应头中添加此插件的名字
- update-duration-header：在结束流时，响应头中添加总处理时间

以上参数默认均为 false

- payload-limit 32：请求体最大允许长度为 32 字节

注意事项

1. 此示例只支持 HTTP Method: GET
2. processing_mode 的配置项中的 **request_header_mode** 及 **response_body_mode** 必须配置为 **下图** 红框中的选项



添加自定义属性

云原生自定义插件示例

envoy-extproc-anti-replay-demo-go

Envoy-extproc-anti-replay-demo-go 是一个基于 envoy-extproc-sdk-go 实现的，用于展示如何在 Go 语言中使用 Envoy 提供的 ext_proc 功能示例。

功能

它的主要功能是在将 Downstream 提交的请求路由到 Upstream 之前，先审核其 sign、timestamp、nonce。 如果任何一个验证失败，则将直接应答 401，以达到防重放的目的。

前置条件

- 安装 Envoy (Version >= v1.29)
- 安装 Go (Verson >= v1.21)，如果只是运行，可跳过此步

- 支持 HTTP Method: POST 的目标服务（以下简称 Upstream），且假设其支持以下 route：
  - /*
  - /no-extproc

编译

进入项目根目录（如果只是运行，可跳过此步）。

```
go build . -o extproc
```

运行

- Envoy：

  ```
  envoy -c ./envoy.yaml # (1)!
  ```

  1. 此文件位于项目根目录

- Caching：
  - 裸金属：

    ```
    ./extproc anti-replay --log-stream --log-phases timespan "900"
    ```

  - k8s：

    ```
    kubectl apply -f ./deployment.yaml # (1)!
    ```

    1. 此文件位于项目根目录

- Curl

  ```
  bash curl --request POST \ --url http://127.0.0.1:8080/ \ --data '{ "key": "value", "key2": "", "sign": "659876b30987883efdf178e69f062896", "nonce": "6062", "timestamp": "1712480920" }'
  ```

!!! note “参数说明”

- log-stream：是否输出关于请求/响应流的日志。
- log-phases：是否输出各处理阶段的日志。
- update-extproc-header：是否在响应头中添加此插件的名字。
- update-duration-header：在结束流时，响应头中添加总处理时间。

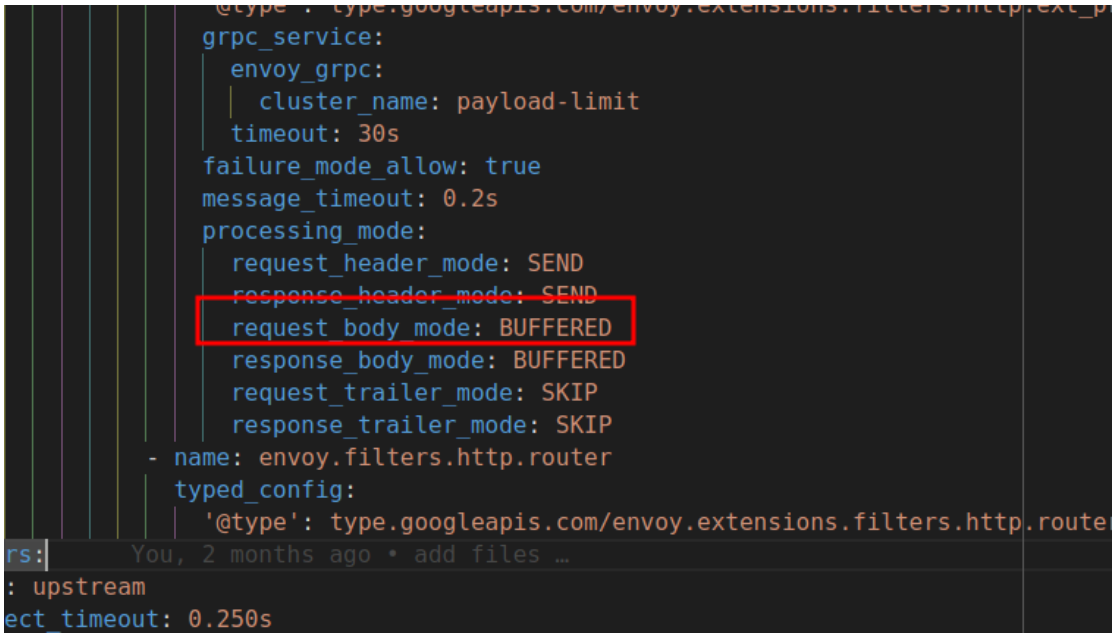
以上参数默认均为 false。

- timespan 900：请求的时间跨度（以 s 计）。

注意事项

1. 此命令行参数中的前 4 个为全局配置参数，即所有基于 envoy-extproc-sdk-go 实现的插件都会默认支持它们；而 timespan 900 为插件 (envoy-extproc-anti-replay-demo-go) 特定参数，由此插件解析与使用。
2. 在此示例中使用 md5 作为"签名"算法，仅是为了演示方便，在正式产品中请使用 SHA256WithRSA 等算法。
3. 以下 3 个字段为每个请求**必填字段**：
   - **sign** ：计算方式为 MD5(k1=v1&k2=v2...kN=vN)，生成原始字符串时按 key 的字母升序排列，且忽略掉值为空的 key-value 对。

     eg: sign= MD5("key=value&nonce=6062&timestamp=1712480920") = 659876b30987883efdf178e69f062896
   - **nonce** ：在时间跨度内同一个 nonce 只可使用一次。
   - **timestamp** ：以 s 计的当前时间。
4. processing_mode 配置项中的 **request_body_mode** 必须配置为 **下图** 红框中的选项：



添加自定义属性

云原生自定义插件示例：envoy-extproc-crc32-check-demo-go

Envoy-extproc-crc32-check-demo-go 是一个基于 envoy-extproc-sdk-go 实现的,用以展示如何在 Go 语言中使用 Envoy 提供的 ext_proc 功能的示例.

它的主要功能是在将 Downstream 提交的请求路由到 Upstream 之前，先对请求体执行 crc 校验，如果校验未通过，将直接应答 403。

前置条件

- 安装 Envoy (Version >= v1.29)
- 安装 Go (Verson >= v1.21)，如果只是运行，可跳过这一步
- 支持 HTTP Method: POST 的目标服务（以下简称 Upstream），且假设其支持以下

route：

- /*
- /no-extproc

编译

进入项目根目录（**如果只是运行，可跳**过这一步）。

```
go build . -o extproc
```

运行

- Envoy：

  ```
  envoy -c ./envoy.yaml # (1)!
  ```

  1. **此文件位于**项目根目录

- Caching：

  - **裸金属**：

    ```
    ./extproc crc32-check --log-stream --log-phases poly "0x82f63b78"
    ```

  - k8s：

    ```
    kubectl apply -f ./deployment.yaml # (1)!
    ```

    1. **此文件位于**项目根目录

- Curl

  ```
  curl --request POST \
       --url http://127.0.0.1:8080/post \
       --data '{
        "data": "1234567890",
        "crc32": "E7C41C6B",
       }'
  ```

参数说明

- log-stream：**是否**输出关于请求/**响**应流的日志
- log-phases：**是否**输出各处理阶段的日志

- update-extproc-header：是否在响应头中添加此插件的名字
- update-duration-header：在结束流时,响应头中添加总处理时间

以上参数默认均为 false。

- poly 0x82f63b78：生成 checksum 时使用的多项式，默认为 IEEE。

注意事项

1. 此命令行参数中的前 4 个为全局配置参数,即所有基于 envoy-extproc-sdk-go 实现的插件都会默认支持它们；而 poly 0x82f63b78 为插件(envoy-extproc-crc32-check-demo-go)特定之参数,由此插件解析与使用.
2. 在此示例中使用 md5 作为"签名"算法,仅是为了演示方便,在正式产品中请使用 SHA256WithRSA 等算法.
3. 以下几个字段为每个请求 **必填** 字段：
   - **data**：用以生成 crc32 的原始数据
   - **crc32**: 校验和，客户端计算时使用的 **多项式** ，必须与插件中的参数相同且其他配置参数必须为以下值，如下图 1 所示

     + **Bit Width**: 32
     + **REFIN**: true
     + **REFOUT**: true
     + **XOROUT (HEX)**: 0xFFFFFFFF
     + **Initial Value (HEX)**: 0xFFFFFFFF
     + **Polynomial Formula (HEX)**: 0x82F63B78
4. processing_mode 的配置项中的 **request_body_mode** 必须配置为下图 2 红框中的选项：

=== "图 1"

![添加自定义属性](docs/zh/docs/skoala/images/CRC1.png)

=== "图 2"

云原生自定义插件示例：envoy-extproc-method-conv-demo-go

Envoy-extproc-method-conv-demo-go 是一个基于 envoy-extproc-sdk-go 实现的， 用以展示如何在 Go 语言中使用 Envoy 提供的 ext_proc 功能的示例。

功能

它的主要功能是将由 Downstream 发起的 GET/POST 请求转换为 POST/GET 请求， 然后再发送到 Upstream，以达到请求方法转换的目的。

前置条件

- 安装 Envoy (Version >= v1.29)
- 安装 Go (Verson >= v1.21) 如果只是运行，可跳过此步
- 支持 HTTP Method: GET/POST 的目标服务(以下简称 Upstream)，且假设其支持以下

```
route:
  – /*
  – /no-extproc
```

编译

进入项目根目录(如果只是运行，可跳过此步)。

```
go build . -o extproc
```

运行

- Envoy:

```
envoy -c ./envoy.yaml # (1)!
```

1. 此文件位于项目根目录

- Caching:
  - 裸金属:

    ```
    ./extproc method-conv --log-stream --log-phases
    ```
  - k8s:

    ```
    kubectl apply -f ./deployment.yaml # (1)!
    ```

    1. 此文件位于项目根目录

- Curl

```
curl 127.0.0.1:8000/no-extproc # (1)!
curl 127.0.0.1:8000/foo # (2)!
curl -XPOST 127.0.0.1:8000/bar # (3)!
```

1. Method-conv 不会作用于此 route，每次请求都会原样路由到 Upstream
2. 此 GET 请求将被 Method-conv 转换成 POST 后再路由到 Upstream
3. 此 POST 请求将被 Method-conv 转换成 GET 后再路由到 Upstream

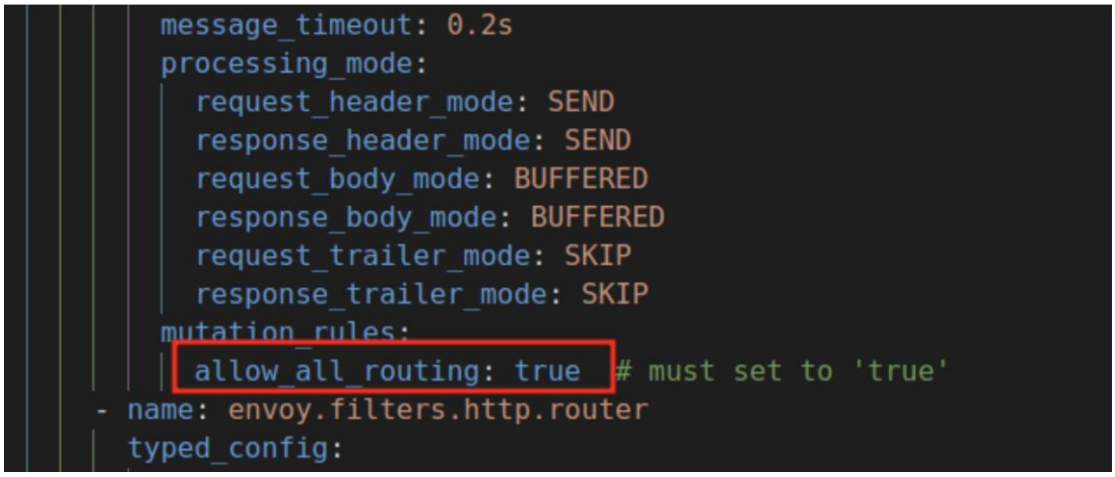
参数说明

- log-stream：是否输出关于请求/响应流的日志
- log-phases：是否输出各处理阶段的日志
- update-extproc-header：是否在响应头中添加此插件的名字
- update-duration-header：在结束流时,响应头中添加总处理时间

以上参数默认均为 false。

注意事项

1. 此示例只支持 HTTP Method：GET、POST 之间的转换
2. mutation_rules 的配置项中的 **allow_all_routing** 必须被设置为 **true** ，如下图红框中所示:



添加自定义属性

云原生自定义插件示例：envoy-extproc-payloadlimit-demo-go

Envoy-extproc-payloadlimit-demo-go 是一个基于 envoy-extproc-sdk-go 实现的，用以展示如何在 Go 语言中使用 Envoy 提供的 ext_proc 功能的示例。

功能

它的主要功能是在将 Downstream 提交的请求体路由到 Upstream 之前，先审核其大小，如果其大于准许发送的最大值，将直接应答 413，以达到限制请求体大小的目的。

前置条件

- 安装 Envoy (Version >= v1.29)
- 安装 Go (Verson >= v1.21)。如果只是运行，可跳过此步
- 支持 HTTP Method: POST 的目标服务（以下简称 Upstream），且假设其支持以下 route：

- /*
- /no-extproc

编译

进入项目根目录(**如果只是运行,可跳**过此步).

```
go build . -o extproc
```

运行

- Envoy

  ```
  envoy -c ./envoy.yaml # (1)!
  ```

  1. **此文件位于**项目根目录

- Caching

  - **裸金属**

    ```
    ./extproc payload-limit --log-stream --log-phases payload-limit 32
    ```

  - K8s

    ```
    kubectl apply -f ./deployment.yaml # (1)!
    ```

    1. **此文件位于**项目根目录

- curl

  ```
  curl -XPOST 127.0.0.1:8000/no-extproc # (1)!
  ```

  1. payload-limit **不会作用于此** route，**无**论请求体大小，请求都会路由到 Upstream

  以命令行参数 payload-limit 32 为例：

  ```
  bash curl -XPOST -H "Content-Type: application/json" -d '{"key1":"value1", "key2":"value2"}' 127.0.0.1:8000/bar
  ```

  如果请求体小于 32 **字**节，**将会正常路由到** Upstream，**否**则由 Payload-limit **直接以状**态码 :413 **响**应。

参数说明

- log-stream：**是否**输出关于请求/**响**应流的日志
- log-phases：**是否**输出各处理阶段的日志

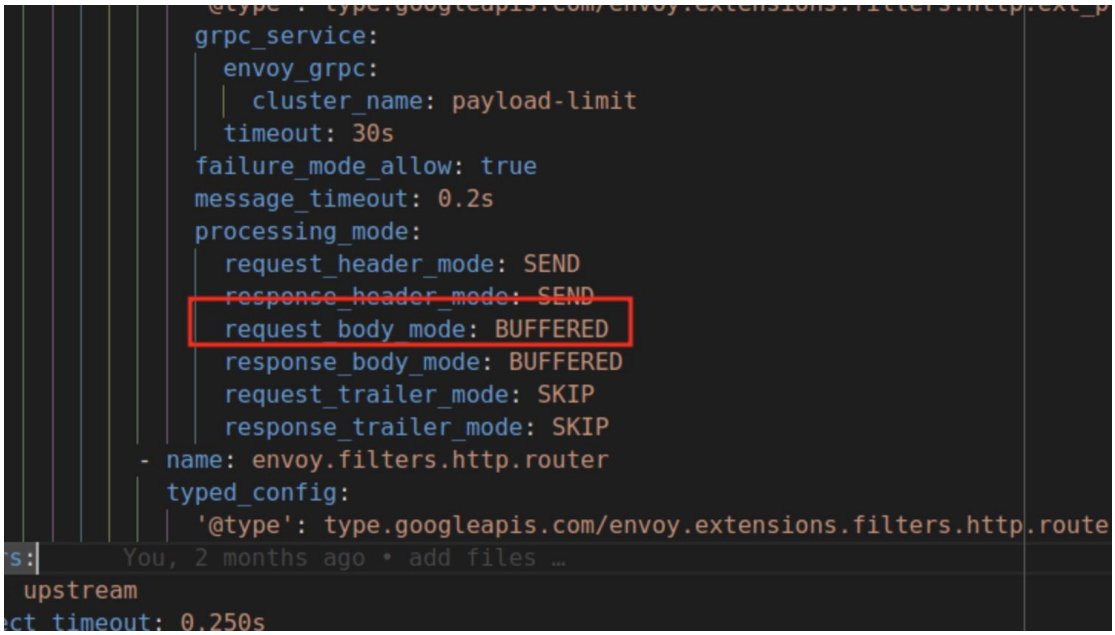
- update-extproc-header：是否在响应头中添加此插件的名字
- update-duration-header：在结束流时，响应头中添加总处理时间

以上参数默认均为 false。

- payload-limit 32：请求体最大允许长度为 32 字节

注意事项

1. 此命令行参数中的前 4 个为全局配置参数,即所有基于 envoy-extproc-sdk-go 实现的插件都会默认支持它们；而 payload-limit 32 为插件(envoy-extproc-payloadlimit-demo-go) 特定之参数，由此插件解析与使用。
2. processing_mode 的配置项中的 request_body_mode 必须配置为 下图 红框中的选项：



添加自定义属性

示例应用体验微服务治理

微服务引擎属于 DCE 5.0 高级版功能，其中包含注册中心、配置中心、微服务治理（传统微服务、云原生微服务）、云原生网关等功能。 本文将通过示例应用带您体验其中的微服务治理功能。

此次最佳实践包含的全部流程如下：

1. 在应用工作台部署示例应用，并启用微服务治理功能
2. 在微服务引擎中启用传统微服务治理插件
3. 在微服务引擎中配置对应的治理规则
4. 在微服务引擎暴露 API 并访问应用

示例应用介绍

本次实践使用的示例应用基于 OpenTelemetry 的标准演示应用，由 DaoCloud 大微服务团队根据 DCE 5.0 的功能加以优化，以便更好地体现云原生以及可观测能力，呈现微服务治理效果。

示例应用已经在 Github 开源，访问该应用的 Github 仓库地址可以获取详细信息。

示例应用的架构图如下：

image

image

应用部署

应用工作台是 DCE 5.0 的应用管理模块，支持创建/维护多种类型的应用、GitOps 和灰度发布等功能，可以快速将应用部署到任何集群。应用工作台支持基于 Git 仓、Jar 包、容

器镜像、Helm 模板部署应用。 本次实践基于 Helm 模板 部署示例应用。

image

image

部署应用之前需要满足如下的前提条件：

- 在容器管理中添加 Helm 仓库:

  image

  image

- 在微服务引擎中创建 Nacos 注册中心实例

  注意记录注册中心的地址信息，后续安装应用时需要用到。

  image

  image

基于 Helm 模板部署

1. 在 应用工作台 -> 向导 -> 基于 Helm 模板 中，找到 opentelemetry-demo 应用，点击应用卡片进行安装

   image

   image

   image

   image

2. 在 Helm 的安装界面，注意确认部署位置是否正确，然后按照下方要求更新 JAVA_OPTS 部分的参数配置。

   image

   image

   根据上方记录的注册中心地址，将下方带有注释的参数更新如下：

   ```
   -javaagent:./jmx_prometheus_javaagent-0.17.0.jar=12345:./prometheus-jmx-config.yaml
         -Dspring.extraAdLabel=Daocloud  -Dspring.randomError=false
         -Dspring.matrixRow=200  -Dmeter.port=8888
         -Dspring.cloud.nacos.discovery.enabled=true          # 修改，以启用 Nacos 服务注册发现
         -Dspring.cloud.nacos.config.enabled=true             # 修改，以启用 Nacos 配置管理能力
         -Dspring.cloud.nacos.config.server-addr=nacos-test.skoala-test:8848                # 修改，以配置 Nacos 注册中心地址
   ```

-Dspring.application.name=adservice-springcloud
-Dspring.cloud.nacos.discovery.server-addr=nacos-test.skoala-test:8848 # 修改，以配置 Nacos 注册中心地址
-Dspring.cloud.nacos.discovery.metadata.k8s_cluster_id=xxx # 修改，以配置 Nacos 注册中心所在集群 ID
-Dspring.cloud.nacos.discovery.metadata.k8s_cluster_name=skoala-dev # 修改，以配置 Nacos 注册中心所在集群名称
-Dspring.cloud.nacos.discovery.metadata.k8s_namespace_name=skoala-test # 修改，以配置 Nacos 注册中心所在命名空间
-Dspring.cloud.nacos.discovery.metadata.k8s_workload_type=deployment
-Dspring.cloud.nacos.discovery.metadata.k8s_workload_name=adservice-springcloud
-Dspring.cloud.nacos.discovery.metadata.k8s_service_name=adservice-springcloud
-Dspring.cloud.nacos.discovery.metadata.k8s_pod_name=${HOSTNAME}
-Dspring.cloud.sentinel.enabled=**false** # 修改，以启用 Sentinel
-Dspring.cloud.sentinel.transport.dashboard=nacos-test-sentinel.skoala-test:8080 # 修改，以配置 Sentinel 控制台地址

获取集群 ID、**集群名称**、**命名空**间名称的方法可参考：kubectl get cluster <clusername> -o json | jq .metadata.uid

3. 应用创建成功后，**会**显示在应用工作台的 Helm 应用列表。

image

image

Java 项目自行开发调试

如果采用其他部署方式，配置注册中心地址的方法可能有所不同。Java 项目在开发时需要**集成** Nacos **的** SDK，**而** DCE 5.0 **提供的注册中心完全兼容开源** Nacos，**所以可以直接使用开源** Nacos **的** SDK。 **具体操作步**骤可参考基于 Jar 包部署 Java 应用。

使用 java -jar **启**动项目时，**添加**对应的环境变量配置：

-Dspring.cloud.nacos.discovery.enabled=**false** # 启用 Nacos 服务注册发现
-Dspring.cloud.nacos.config.enabled=**false** # 启用 Nacos 配置管理能力
-Dspring.cloud.sentinel.enabled=**false** # 启用 Sentinel
-Dspring.cloud.nacos.config.server-addr=nacos-test.skoala-test:8848 # 配置 Nacos 注册中心地址
-Dspring.application.name=adservice-springcloud
-Dspring.cloud.nacos.discovery.server-addr=nacos-test.skoala-test:8848 # 配置 Nacos 注册中心地址
-Dspring.cloud.nacos.discovery.metadata.k8s_cluster_id=xxx # 配置 Nacos 注册中心所在集群 ID

-Dspring.cloud.nacos.discovery.metadata.k8s_cluster_name=skoala-dev # 配置 Nacos 注册中心所在集群名称
-Dspring.cloud.nacos.discovery.metadata.k8s_namespace_name=skoala-test # 配置 Nacos 注册中心所在命名空间
-Dspring.cloud.nacos.discovery.metadata.k8s_workload_type=deployment
-Dspring.cloud.nacos.discovery.metadata.k8s_workload_name=adservice-springcloud
-Dspring.cloud.nacos.discovery.metadata.k8s_service_name=adservice-springcloud
-Dspring.cloud.nacos.discovery.metadata.k8s_pod_name=${HOSTNAME}
!!! note
上面的 `metadata` 信息不能缺失，否则注册中心中呈现的服务会缺失这部分信息。

使用容器镜像部署

如果选择基于容器镜像部署应用，可以直接在用户界面配置中开启微服务治理并选取对应的注册中心模块，操作更简便。 具体步骤可参考基于 Git 仓构建微服务应用。

image

image

启用传统微服务治理

开始使用微服务治理功能之前，需要在对应注册中心下的插件中心开启对应的治理插件。插件中心提供 Sentinel 治理和 Mesh 治理两种插件，支持通过用户界面实现可视化配置。安装插件后可以扩展微服务治理能力，满足不同场景下的业务诉求。

本次实践采用传统微服务治理，即开启 Sentinel 治理插件。 如需了解详细步骤，可参考启用 Sentinel 治理插件。

image

image

配置对应的治理规则

应用部署成功后，可以在之前准备注册中心下的微服务列表中查看对应的服务。 微服务列表提供流控规则、熔断降级、热点规则、系统规则、授权规则等流量治理规则。本次实践以流控规则为例进行演示。

image

image

配置流控策略

这里限流策略示例，我们通过简单的配置即可为服务增加对应的限流策略。

image

image

测试流控策略

通过访问服务地址，我们可以看到在 1 分钟内请求次数大于 2 次之后，后续请求都会被拦截；在超过 1 分钟后自动恢复。

暴露 API 并访问应用

微服务应用部署完成后，需要通过 API 网关将应用入口开放给外部访问，完成这一步才是完成的服务使用体验。 为了暴露服务 API，需要创建云原生网关，将服务接入该网关，并创建对应的 API 路由。

创建云原生网关

首先需要创建一个云原生网关，具体操作步骤可以参考： 创建云原生网关

!!! note
创建网关时，应该将网关部署在示例应用所在的集群，并且该网关需要管辖示例应用所在的命名空间。

![image](https://docs.daocloud.io/daocloud-docs-images/docs/skoala/images/gatewaylist.png)

接入服务

基于 DCE 5.0 的特性，云原生网关可以自动发现所管辖命名空间内的服务，所以无需手动

接入服务。

本次演示采用 Nacos 注册中心的服务，很大程度上扩宽了网关可接入的服务数量，可以在服务接入 中选择接入 Nacos 注册中心的服务。

image

image
!!! info
当服务不在网关所管辖的命名空间内，或者想要接入注册中心或者其他外部服务时（使用域名/IP），可以采用手工接入服务的方式。

创建 API 路由

参考文档 添加 API 创建对应的 API 路由。

image

image

访问应用

当网关 API 创建完成后，使用创建 API 时配置的 域名 与 外部 API 路径 即可成功访问到应用页面，如下图所示。

示例应用的首页:

image

image

示例应用的订单确认页面:

image

image

结语

以上就是整个微服务引擎模块的体验之旅。在整个 DCE 5.0 的能力支持下我们顺利完成了应用部署、 启用微服务治理、配置并测试微服务治理策略、通过云原生网关开放 API、实际访问应用等操作。

更多能力

当我们的应用部署成功之后，对于后续的应用维护过程中，实际非常依赖 DCE 5.0 提供的可观测能力。接下来，我们会补充对应可观测能力实践。

- 查看应用部署后的拓扑结构
- 查看应用的日志内容
- 查看云网关网关 API 的访问日志

网关 API 策略

DCE 5.0 云原生网关支持支持负载均衡、超时重试、黑白名单等十几项 API 策略。本文借助真实的微服务应用演示如何配置各项策略以及每项策略的具体效果。

!!! note
- 有关各项策略的配置步骤、参数说明等细节，可参考[配置 API 策略](../gateway/api/api-policy.md)。
- 本文侧重于展示各项策略的使用效果，会省略一些配置步骤。

前提条件

- 准备一个演示服务，例如 my-otel-demo-adservice
- 创建网关、域名、API
- 将演示服务接入网关

负载均衡

1. 将演示服务扩展为多副本，便于演示负载均衡
2. 参考负载均衡配置负载均衡策略

3. 编写脚本记录各个服务副本接收到的流量

??? note “点击查看脚本内容”

```python
import requests

def lb_pod_count():
    pod1_ip = "10.244.1.135" # 服务副本所在节点的 IP
    pod2_ip = "10.244.2.207" # 服务副本所在节点的 IP
    pod3_ip = "10.244.3.193" # 服务副本所在节点的 IP
    pod1_count = 0
    pod2_count = 0
    pod3_count =0

    host = "http://10.6.222.24:30040/ip" # 服务访问地址
    headers = {"Host": "ad.service.virtualhost"}
    for i in range(300):
        res_ip = requests.get(host, headers=headers).text[11:]
        if res_ip == pod1_ip:
            pod1_count += 1
        elif res_ip == pod2_ip:
            pod2_count += 1
        elif res_ip == pod3_ip:
            pod3_count += 1

    print("指向%s 的流量为：%d" %(pod1_ip,pod1_count))
    print("指向%s 的流量为：%d" %(pod2_ip,pod2_count))
    print("指向%s 的流量为：%d" %(pod3_ip,pod3_count))

if __name__ == '__main__':
    lb_pod_count()
```

4. **随机**负载均衡的效果如下。**每个副本随机接受流量**，导致个别副本压力过大。

random

random

5. 轮询负载均衡的效果如下。**每个副本**轮流接受流量，**因此各个副本**处理的流量总数基本相同。

random

random

路径改写

1. 初始状态下，访问的是服务根路径，返回内容如图。

rewrite

rewrite

2. 配置路径改写策略，原先访问服务根路径，改为访问 /test2 接口。

rewrite

rewrite

3. 效果如图。从返回内容可以看出，现在访问的接口已经改变了，返回的数据也变了。

rewrite

rewrite

超时配置

1. 启用超时配置，设置响应超过 3 秒时判断访问失败。

rewrite

rewrite

2. 通过测试专用接口 timeout 让请求响应 1 秒，预期可以成功访问。

   下图说明访问成功，接口返回了所设置的请求时间

rewrite

rewrite

3. 通过测试专用接口 timeout 让请求响应 4 秒，预期会访问失败

   下图说明访问失败，接口返回上游请求超时的信息。

rewrite

rewrite

重试机制

1. 启用重试配置，访问失败时最多重试 6 次，重试响应时间超过 1 秒时视为重试失败。

rewrite

rewrite

2. 通过测试专用接口 set-retry-count 使得服务的 /retry 接口必须请求 3 次才能访问成功。

rewrite

rewrite

访问 retry 接口。

在重试次数为 6 的情况下，预期可以访问成功。

rewrite

rewrite

3. 通过测试专用接口 set-retry-count 使得服务的 /retry 接口必须请求 7 次才能访问成功。

rewrite

rewrite

访问 retry 接口。

在重试次数为 6 的情况下，预期也能访问成功。因为首次正常访问加失败后的 6 次重试，正好达到了所需的 7 次访问。

rewrite

rewrite

4. 通过测试专用接口 set-retry-count 使得服务的 /retry 接口必须请求 8 次才能访问成功。

rewrite

rewrite

访问 retry 接口。

在重试次数为 6 的情况下，预期第一次会访问失败。因为首次正常访问加失败后的 6 次重试，最多只能访问 7 次，达不到所需的 8 次访问。

但此时手动再次访问该接口，就会显示访问成功，因此此时累计的访问次数已经达到了 8 次。

rewrite

rewrite

请求头重写

1. 启用请求头重写策略，在请求服务时移除 user-agent （不区分大小写，并添加 demo-req。

   rewrite

   rewrite

2. 使用 postman 工具查看服务初始状态下带有的请求头。可以看到存在 user-agent，但没有 demo-req。

   rewrite

   rewrite

3. 通过测试专用接口 /request-header 查看请求服务时是否带上了 user-agent。

   返回 user-agent 的值为 null，说明该请求头被移除了，重写策略生效。

   rewrite

   rewrite

4. 通过测试专用接口 /request-header 查看请求服务时是否带上了 user-agent。

   返回了预先设置的值，说明添加了该请求头，重写策略生效。

   rewrite

   rewrite

响应头重写

1. 启用响应头重写策略，在服务响应头中移除 x-envoy-upstream-service-time （不区分大小写）响应头，并添加 demo-res。

   rewrite

   rewrite

2. 使用 postman 工具查看初始状态下的响应头。可以看到存在 x-envoy-upstream-service-time，但没有 demo-res。

   rewrite

   rewrite

3. 直接访问该服务

响应头中没有 x-envoy-upstream-service-time，但出现了 demo-res ，说明重写策略生效。

rewrite

rewrite

WebSocket

1. 编写测试 WebSocket 的脚本

??? note “点击查看脚本内容”

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>本地 websocket 测试</title>
        <meta name="robots" content="all" />
        <meta name="keywords" content="本地,websocket,测试工具" />
        <meta name="description" content="本地,websocket,测试工具" />
        <style>
            .btn-group{
                display: inline-block;
            }
        </style>
    </head>
    <body>
        <input type='text' value='通信地址, ws://开头..' class="form-control" style='width:390px;display:inline'
        id='wsaddr' />
        <div class="btn-group" >
            <button type="button" class="btn btn-default" onclick='addsocket();'>连接</button>
            <button type="button" class="btn btn-default" onclick='closesocket();'>断开</button>
            <button type="button" class="btn btn-default" onclick='$("#wsaddr").val("")'>清空</button>
        </div>
        <div class="row">
            <div id="output" style="border:1px solid #ccc;height:365px;overflow: aut
```

```
o;margin: 20px 0;"></div>
                <input type="text" id='message' class="form-control" style='width:810px'
placeholder="待发信息" onkeydown="en(event);">
                <span class="input-group-btn">
                    <button class="btn btn-default" type="button" onclick="doSend();">
发送</button>
                </span>
                </div>
            </div>
        </body>

            <script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
            <script language="javascript" type="text/javascript">
                function formatDate(now) {
                    var year = now.getFullYear();
                    var month = now.getMonth() + 1;
                    var date = now.getDate();
                    var hour = now.getHours();
                    var minute = now.getMinutes();
                    var second = now.getSeconds();
                    return year + "-" + (month = month < 10 ? ("0" + month) : mont
h) + "-" + (date = date < 10 ? ("0" + date) : date) +
                        " " + (hour = hour < 10 ? ("0" + hour) : hour) + ":" + (min
ute = minute < 10 ? ("0" + minute) : minute) + ":" + (
                            second = second < 10 ? ("0" + second) : second);
                }
                var output;
                var websocket;

                function init() {
                    output = document.getElementById("output");
                    testWebSocket();
                }

                function addsocket() {
                    var wsaddr = $("#wsaddr").val();
                    if (wsaddr == '') {
                        alert("请填写 websocket 的地址");
                        return false;
                    }
                    StartWebSocket(wsaddr);
                }

                function closesocket() {
```

```
            websocket.close();
        }

        function StartWebSocket(wsUri) {
            websocket = new WebSocket(wsUri);
            websocket.binaryType = "arraybuffer";
            websocket.onopen = function(evt) {
                onOpen(evt)
            };
            websocket.onclose = function(evt) {
                onClose(evt)
            };
            websocket.onmessage = function(evt) {
                onMessage(evt)
            };
            websocket.onerror = function(evt) {
                onError(evt)
            };
        }

        function onOpen(evt) {
            writeToScreen("<span style='color:red'>连接成功，现在你可以发送
信息啦！！！</span>");
        }

        function onClose(evt) {
            writeToScreen("<span style='color:red'>websocket 连接已断开!!!</spa
n>");
            websocket.close();
        }

        function onMessage(evt) {
            writeToScreen('<span style="color:blue">服务端回应 ' + formatDate(
new Date()) + '</span><br/><span class="bubble">' +
                evt.data + '</span>');
        }

        function onError(evt) {
            writeToScreen('<span style="color: red;">发生错误:</span> ' + evt.d
ata);
        }

        function doSend() {
            var message = $("#message").val();
```

```
                    if (message == '') {
                        alert("请先填写发送信息");
                        $("#message").focus();
                        return false;
                    }
                    if (typeof websocket === "undefined") {
                        alert("websocket 还没有连接，或者连接失败，请检测");
                        return false;
                    }
                    if (websocket.readyState == 3) {
                        alert("websocket 已经关闭，请重新连接");
                        return false;
                    }
                    console.log(websocket);
                    $("#message").val('');
                    writeToScreen('<span style="color:green">你发送的信息 ' + formatD
ate(new Date()) + '</span><br/>' + message);
                    websocket.send(message);
                }

                function writeToScreen(message) {
                    var div = "<div class='newmessage'>" + message + "</div>";
                    var d = $("#output");
                    var d = d[0];
                    var doScroll = d.scrollTop == d.scrollHeight - d.clientHeight;
                    $("#output").append(div);
                    if (doScroll) {
                        d.scrollTop = d.scrollHeight - d.clientHeight;
                    }
                }


                function en(event) {
                    var evt = evt ? evt : (window.event ? window.event : null);
                    if (evt.keyCode == 13) {
                        doSend()
                    }
                }
            </script>
    </html>
    ```
```

2. **在上述脚本保存**为 html **文件**，**双**击打开该文件。

rewrite
```

rewrite

3. 在未开启 WebSocket 策略，点击 连接 ，会显示访问失败。

rewrite

rewrite

4. 开启 WebSocket 策略。

rewrite

rewrite

5. 再次点击 连接 ，显示访问成功，说明 WebSocket 策略生效。

rewrite

rewrite

本地限流

1. 启用本地限流策略。

下图设置的含义是：每分钟只能正常请求 3 次，但允许溢出访问 2 次，所以每分钟累计允许访问 5 次。 超过 5 次的访问会返回 429 代码，并附上 ratelimit=done 和 ratelimit1= done1 的响应头。

rewrite

rewrite

2. 通过 curl 命令访问服务。

可以看到第六次访问时返回了 local_rate_limit 信息，表示由于限流而访问失败。

并且返回内容也出现了 429 代码以及新增的两个响应头

rewrite

rewrite

健康检查

1. 启用健康检查策略。

在没有设置路径改写的情况下，正常访问的是服务的根路径 /。但如果将检查路径设置为 /test，预期会健康检查失败，导致服务无法访问。

rewrite

rewrite

rewrite

rewrite

2. 修改为正确的检查路径。预期会通过健康检查，服务可以正常访问。

rewrite

rewrite

rewrite

rewrite

Cookie 重写

1. 启用 Cookie 重写策略。

   重写时需要确保 cookie 名称与已有 cookie 的名称相同，才能确保原先的属性被新设置的属性覆盖。

rewrite

rewrite

2. 使用测试专用接口 /cookie-set 设置请求时的 cookie 属性，并在响应头中携带实际生效的 cookie 属性。

   网关请求时设置的 cookie 为 Cookie{name='cookie-name', value='cookie-value', maxAge=PT-1S, domain='test.domain', path='/path', secure=false, httpOnly=false, sameSite='Lax'}

   而在响应头中 set-cookie 展示了实际生效的 cookie： cookie-name=cookie-value; Secure; Domain=rewrite.domain; SameSite=Strict; Path=/rewrite/path

rewrite

rewrite

通过网关访问服务

本文演示如何将微服务接入 DCE 5.0 云原生网关并通过网关访问该服务。

前提条件

- 准备一个服务。本文使用的是 my-otel-demo-adservice 服务，部署在 skoala-dev 集群下的 webstore-demo 命名空间。
- 访问 my-otel-demo-adservice 服务根路径 / 理论上应该返回 adservice-springcloud: hello world!。

服务接入网关

DCE 5.0 云原生网关支持通过手动接入和自动发现两种方式导入服务，操作时根据自身情况二选一即可。

自动发现服务

1. 参考文档创建网关创建一个网关。 将服务所在的命名空间添加为网关的管辖命名空间 。 此次演示使用的服务位于 webstore-demo 命名空间。所以创建网关时应该做如下配置:

   管辖命名空间

   管辖命名空间

2. 服务所在的命名空间被添加为管辖命名空间之后，该命名空间下的所有服务都会自动接入网关，无需手动添加。

   自动发现

   自动发现

3. 参考添加域名在网关下面创建域名，例如 adservice.virtualhost。
4. 参考添加 API 在网关下面创建 API。 需要将服务添加为 API 的后端服务 。

添加后端服务时，筛选自动发现类型的服务，然后勾选目标服务，点击 确定 即可。

后端服务

后端服务

手动接入服务

1. 参考创建网关创建网关。
2. 参考手动接入文档将服务接入网关

手动接入网关

手动接入网关

3. 参考添加域名在网关下面创建域名，例如 adservice.virtualhost。
4. 参考添加 API 在网关下面创建 API。

需要将服务添加为 API 的后端服务 。

添加后端服务时，筛选手工接入类型的服务，然后勾选目标服务，点击 确定 即可。

后端服务

后端服务

获取网关地址

1. 登录到网关所在集群的控制节点，使用 kubectl get po -n $Namespace 命令查看网关所在的节点。 以 envoy 开头的 Pod 是网关的数据面，查看这个 Pod 的位置即可。
2. 使用 ping 命令和网关所在节点通信，根据返回的数据得知该节点的 IP。

   在本次演示情形下，网关所在节点的 IP 为 10.6.222.24。

ping

ping

3. 使用 kubectl get svc -n $Namespace 命令查看网关暴露的端口。

   以 envoy 开头且以 gtw 结尾的便是网关对应的 Service。在本次演示情形下，网关 Service 的端口为 30040。

nodeport

nodeport

4. 根据上述信息得出，网关的访问地址为 10.6.222.24: 30040。

配置本地域名

使用 vim /etc/hosts 命令修改本地 hosts 文件，为网关访问地址配置本地域名。

hosts

hosts

访问服务

配置本地域名之后，即可使用域名通过外部和内部网络访问网关服务。此时应该可以正常访问，返回服务根路径下的 hello world 内容。

外部访问

在本次演示情形下，可以使用 curl adservice.virtualhost: 30040/。

internal visit

internal visit

内部访问

在网关所在集群的任意节点上都可以通过网关成功访问 adservice 服务。

在本次演示情形下，网关所在集群中有三个工作节点，分别名为 dev-worker1、dev-worker2、dev-worker3， 在这三个节点上，使用内网 IP 均可以访问成功。

public visit

public visit

网关对接服务网格

本文介绍如何对接 DCE 5.0 云原生网关和服务网格，使得可以通过网关访问服务网格下接入的服务， 并且可以正常使用网格的所有流量治理能力，例如虚拟服务、目标规则等。

现状

目前 DCE 5.0 云原生网关采用 contour 作为控制面，无法同步 Istio 的策略。通过云原生网关访问网格服务时， 网关 API 直接连接到网格服务，由于没有同步 Istio 的策略，所以无法应用虚拟服务、目标服务等规则，导致网格能力缺失。

对接思路

让云原生网关通过 Istio 的 Sidecar 访问网格服务，从而应用 Istio 的规则。

操作步骤

1. 创建云原生网关时，开启注入 Sidecar。

   创建网关

   创建网关

2. 为 Pod 设置 Annotation 注解 traffic.sidecar.istio.io/includeInboundPorts: ""。

   这样可以不让 Istio 处理入口流量，减少性能损耗。

3. 在 Envoy 的启动参数中增加 --base-id 10，允许同一个命名空间中存在两个 envoy 实例，防止冲突。

   参数示意图

   参数示意图

4. 在网关下添加 API 时，设置 Host 请求头。

   这样可以让网关在多集群场景下，通过 Sidecar 访问其他集群的服务，实现服务的跨集群负载均衡。

   参数示意图

   参数示意图

5. 在服务网格模块为网关 Sidecar 配置资源，不低于云原生网关的默认资源配置 1 核 1 G。

效果展示

完成上述配置之后，通过网关域名访问服务。可以看到每次请求的是不同集群中的服务，说明网关可以跨集群访问服务。

参数示意图

参数示意图

网关请求日志添加自定义属性

微服务引擎的网关底层使用的是 contour 作为控制面下发配置给 envoy， 其中请求日志的记录字段也会作为一项配置由 contour 下发给 envoy， 因此只需要在 contour 的配置文件中增加响应的需要记录的字段即可。

具体步骤如下：

1. 进入微服务引擎——云原生网关列表，找到需要修改配置的网关所在的集群和命名空间。

添加自定义属性

添加自定义属性

2. 在网关所在的集群和命名空间查询自定义 CR ContourConfiguration。

```
kubectl get contourconfig -n skoala-test
```

```
NAME                              AGE
contourconfig-vita-demo-gtw       21d
```

3. 编辑上一步查询到的 CR 资源，在 .spec.envoy.logging.accessLogJSONFields 中增加想要记录的日志字段属性，例如想要增加一个日志字段为响应头的 X-Envoy-Upstream-Service-Time 字段，可以增加一行 X-Envoy-Upstream-Service-Time=%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% 。

```
kubectl edit contourconfig -n skoala-test
```

编辑前:

添加自定义属性

添加自定义属性

编辑后：

添加自定义属性

添加自定义属性

4. 重启 contour 组件（不会对流量造成中断）

```
kubectl get pods -n skoala-test | grep contour | awk '{print $1}' | xargs kubectl delete pod -n skoala-test
```

```
pod "contour-vita-demo-gtw-785f495bdf-n7knb" deleted
pod "contour-vita-demo-gtw-b5fcc57bb-pbssx" deleted
```

5. Pod 启动成功后通过网关 API 访问接口，通过网关日志观察日志记录字段

添加自定义属性

添加自定义属性

选择某一条日志，查看原文：

添加自定义属性

添加自定义属性

可以看到自定义的响应头已经增加到了日志记录中。

!!! note
该功能适用于微服务引擎 0.29.0 及更高版本。

HTTP 转 Dubbo 协议动态路由

使用 Pixiu 可以支持协议转换功能，目前已支持 Http、Dubbo2、Triple、gRPC 协议代理和转换，本文主要介绍如何使用它将 HTPP 协议转换为 Dubbo 协议。

配置说明

以下内容以及注释仅涉及静态配置文件配置，不涉及配置下发。 Pixiu 静态启动配置文件，内容如下，相应字段含义均已注释：

```
- --
static_resources:
  listeners:
    # 名称，没有实际意义
    - name: "net/http"
      # 端口地址监听协议，这里固定为 *HTTP*
      protocol_type: "HTTP"
      # 实际代码未使用到，感觉是 bug，不需要处理，默认均为 20s
      #         config:
      #           read_timeout: 5s
      #           write_timeout: 5s
      #           idle_timeout: 5s
      # 监听地址
      address:
        socket_address:
          address: "0.0.0.0"
          port: 8881
      # 过滤器链
```

```
      filter_chains:
        filters:
          # 过滤器名称，这里固定 http 连接管理器
          - name: dgp.filter.httpconnectionmanager
            config:
              # 路由配置，这里固定填写即可
              route_config:
                routes:
                  - match:
                      prefix: "*"
              http_filters:
                # api 配置 filter
                - name: dgp.filter.http.apiconfig
                  config:
                    # 表明动态加载
                    dynamic: true
                    # 动态适配器的名称
                    dynamic_adapter: test
                # dubbo 代理 filter
                - name: dgp.filter.http.dubboproxy
                  config:
                    dubboProxyConfig:
                      registries:
                        "nacos":
                          protocol: "nacos"
                          timeout: "3s"
                          address: "127.0.0.1:8848"
                            # username: nacos
                            # password: nacos
                            # group: test-group
                            # namespace: test-namespace
# 超时时间，无效
# timeout: 2s
# 无效
# generate_request_id: false
# 服务名称，字段无效
# server_name: "test_http_to_dubbo"
# 适配器配置,自动配置动态 API
adapters:
  # id 和上面的 *dynamic_adapter* 的值对应
  - id: test
    name: dgp.adapter.dubboregistrycenter
    config:
      registries:
```

```
  # 注册中心 map 配置
  "nacos":
    # 协议，支持 nacos 和 zookeeper
    protocol: nacos
    # 注册中心的地址
    address: "127.0.0.1:8848"
    # 注册中心端口
    timeout: "5s"
    # 如果注册中心存在认证逻辑这里填写用户名
    #  username: nacos
    # 如果注册中心存在认证逻辑这里填写密码
    # password: nacos
    # 需要监听的注册中心分组
    # group: test-group
    # 注册中心的命名空间
    # namespace: test-namespace
```

完成该配置后，Pixiu 会自动从 adapters 中配置的注册中心获取所有服务的可用方法，并自动装配成 API，之后只需要按照固定格式请求 Pixiu， Pixiu 即可将流量转发到对应的 dubbo 服务的对应接口的对应方法，下面介绍这种方式下请求 Pixiu 的格式，一定要严格按照格式请求，否则无法匹配到对应的服务。

!!! info
1. 请求方法必须为 POST
2. 请求路径格式为：

    ```yaml
    IP: Port/服务名/接口名/版本号/方法名
    ```

    其中 IP 和 Port 均为 Pixiu 对外暴露的 IP 和端口，另外服务名、接口名、版本号和方法名均为对应想要请求的方法的属性.

3. 请求体必需为 json 格式，存在两个键，分别是 types 和 values，其中 types 为请求方法的参数类型，字符串类型，
    多个参数类型以逗号分隔，values 为对应参数类型的对应值，数组类型，多个参数以逗号分隔。

Demo 示例

Demo 示例代码地址：<https: //github.com/projectsesame/pixiu-demo-dubbo.git>

测试命令如下（**假定本地启**动的 Pixiu，**且**访问端口为 8881）：

```
# 127.0.0.1 是 Pixiu 部署的 ip，8881 是 Pixiu 启动的端口，dubbo3x-provider 是服务名
# io.daocloud.skoala.dubboapi.DubboDemoService 是接口名称，0.1.1 是版本号，sayHello 是方法名称
# 这里 sayHello 方法存在一个 String 类型的参数，由于是 java 语言开发的 demo，因此类型的完整名称是 java.lang.String
# 这里参数传值为 tt，因此这里传参 {\"types\": \"java.lang.String\",\"values\":[\"tt\"]}，下面例子同理
```

```
curl -XPOST http://127.0.0.1:8881/dubbo3x-provider/io.daocloud.skoala.dubboapi.DubboDemoService/0.1.1/sayHello -d "{\"types\":\"java.lang.String\",\"values\":[\"tt\"]}" -H 'Content-Type: application/json'
```

```
"[dubbo3x-provider] : Hello, tt
```

```
curl -XPOST http://127.0.0.1:8881/dubbo3x-provider/io.daocloud.skoala.dubboapi.DubboDemoService/0.1.1/returnUserInfo -d "{\"types\":\"io.daocloud.skoala.dubboapi.User\",\"values\":[{\"name\":\"yang\",\"age\":10}]}" -H 'Content-Type: application/json'
```

```
{"age": 10,"name":"yang"}
```

```
curl -XPOST http://127.0.0.1:8881/dubbo3x-provider/io.daocloud.skoala.dubboapi.DubboDemoService/0.1.1/returnUserInfoAndNameAndIntAge -d "{\"types\":\"java.lang.String,io.daocloud.skoala.dubboapi.User,int\",\"values\":[\"tt\",{\"name\":\"yang\",\"age\":10},20]}" -H 'Content-Type: application/json'
```

```
"method is returnUserInfoAndNameAndIntAge,userInfo is User{name='yang', age=10},name is tt, age is 20"
```

```
curl -XPOST http://127.0.0.1:8881/dubbo3x-provider/io.daocloud.skoala.dubboapi.DubboDemoService/0.1.1/returnUserInfoAndNameAndIntegerAge -d "{\"types\":\"java.lang.String,io.daocloud.skoala.dubboapi.User,java.lang.Integer\",\"values\":[\"tt\",{\"name\":\"yang\",\"age\":10},20]}" -H 'Content-Type: application/json'
```

```
""method is returnUserInfoAndNameAndIntegerAge,userInfo is User{name='yang', age=10},name is tt, age is 20"
```

Nacos 对接 LDAP 进行用户管理

Nacos 在 DCE 5.0 微服务引擎中名为注册中心。 本文说明如何通过 Nacos 对接 LDAP

（Lightweight Directory Access Protocol，轻量级目录访问协议）来管理用户。

在 Kubernetes 上部署 LDAP

先部署 ldap.yaml

kubectl apply -f ldap.yaml
yaml title="ldap.yaml" apiVersion: apps/v1 kind: Deployment metadata: name: ldap labels: app: ldap spec: replicas: 1 selector: matchLabels: app: ldap template: metadata: labels: app: ldap spec: containers: - name: ldap image: docker.m.daocloud.io/osixia/openldap:latest volumeMounts: - name: ldap-data mountPath: /var/lib/ldap - name: ldap-config mountPath: /etc/ldap/slapd.d - name: ldap-certs mountPath: /container/service/slapd/assets/certs ports: - containerPort: 389 name: openldap env: - name: LDAP_LOG_LEVEL value: "256" - name: LDAP_ORGANISATION value: "Example Inc." - name: LDAP_DOMAIN value: "example.org" - name: LDAP_ADMIN_PASSWORD value: "admin" - name: LDAP_CONFIG_PASSWORD value: "config" - name: LDAP_READONLY_USER value: "false" - name: LDAP_READONLY_USER_USERNAME value: "readonly" - name: LDAP_READONLY_USER_PASSWORD value: "readonly" - name: LDAP_RFC2307BIS_SCHEMA value: "false" - name: LDAP_BACKEND value: "mdb" - name: LDAP_TLS value: "true" - name: LDAP_TLS_CRT_FILENAME value: "ldap.crt" - name: LDAP_TLS_KEY_FILENAME value: "ldap.key" - name: LDAP_TLS_CA_CRT_FILENAME value: "ca.crt" - name: LDAP_TLS_ENFORCE value: "false" - name: LDAP_TLS_CIPHER_SUITE value: "SECURE256:+SECURE128:-VERS-TLS-ALL:+VERS-TLS1.2:-RSA:-DHE-DSS:-CAMELLIA-128-CBC:-CAMELLIA-256-CBC" - name: LDAP_TLS_VERIFY_CLIENT value: "demand" - name: LDAP_REPLICATION value: "false" - name: LDAP_REPLICATION_CONFIG_SYNCPROV value: "binddn=\"cn=admin,cn=config\" bindmethod=simple credentials=$LDAP_CONFIG_PASSWORD searchbase=\"cn=config\" type=refreshAndPersist retry=\"60 +\" timeout=1 starttls=critical" - name: LDAP_REPLICATION_DB_SYNCPROV value: "binddn=\"cn=admin,$LDAP_BASE_DN\" bindmethod=simple credentials=$LDAP_ADMIN_PASSWORD searchbase=\"$LDAP_BASE_DN\" type=refreshAndPersist interval=00:00:00:10 retry=\"60 +\" timeout=1 starttls=critical" - name: LDAP_REPLICATION_HOSTS value: "#PYTHON2BASH:['ldap://ldap-one-service', 'ldap://ldap-two-service']" - name: KEEP_EXISTING_CONFIG value: "false" - name: LDAP_REMOVE_CONFIG_AFTER_SETUP value: "true" - name:

LDAP_SSL_HELPER_PREFIX value: "ldap" volumes: - name: ldap-data hostPath: path: "/data/ldap/db" - name: ldap-config hostPath: path: "/data/ldap/config" - name: ldap-certs hostPath: path: "/data/ldap/certs" - -- apiVersion: v1 kind: Service metadata: labels: app: ldap name: ldap-service spec: ports: - port: 389 type: NodePort selector: app: ldap

再部署 phpadmin.yaml：

kubectl apply -f phpadmin.yaml
yaml title="phpadmin.yaml" apiVersion: apps/v1 kind: Deployment metadata: annotations: kompose.cmd: kompose convert -f docker-compose.yml kompose.version: 1.16.0 (0c01309) creationTimestamp: null labels: io.kompose.service: phpldapadmin name: phpldapadmin spec: replicas: 1 selector: matchLabels: io.kompose.service: phpldapadmin template: metadata: creationTimestamp: null labels: io.kompose.service: phpldapadmin spec: containers: - env: - name: PHPLDAPADMIN_HTTPS value: "false" - name: PHPLDAPADMIN_LDAP_HOSTS value: ldap-service image: docker.m.daocloud.io/osixia/phpldapadmin:latest name: phpldapadmin ports: - containerPort: 80 restartPolicy: Always - -- apiVersion: v1 kind: Service metadata: annotations: kompose.cmd: kompose convert -f docker-compose.yml kompose.version: 1.16.0 (0c01309) labels: io.kompose.service: phpldapadmin name: phpldapadmin spec: ports: - name: "8080" port: 8080 targetPort: 80 type: NodePort selector: io.kompose.service: phpldapadmin

部署 Nacos

在一台虚拟机上部署 Nacos。

1. 下载安装包

   cd /opt
   wget https://github.com/alibaba/nacos/releases/download/2.3.2/nacos-server-2.3.2.tar.gz

2. 配置 LDAP

   tar -zxvf nacos-server-2.3.2.tar.gz
   cd nacos/conf/

   编辑 application.properties 文件，开启认真并配置 ldap 连接信息：

   config title="application.properties" nacos.core.auth.system.type=ldap
   nacos.core.auth.enabled=true nacos.core.auth.server.identity.key=DefaultIdentityKey
   nacos.core.auth.server.identity.value=DefaultIdentityValue
   nacos.core.auth.plugin.nacos.token.secret.key=RGFvY2xvdWRTa29hbGFOYWNvc0F1dGh

   TZWNyZXRLZXk= nacos.core.auth.ldap.url=ldap://10.6.176.50:30326 # 上一步安装

的 ldap 连接地址，节点 ip:389 映射端口

nacos.core.auth.ldap.basedc=dc=example,dc=org
nacos.core.auth.ldap.userDn=cn=admin,${nacos.core.auth.ldap.basedc}
nacos.core.auth.ldap.password=admin
nacos.core.auth.ldap.userdn=cn={0},dc=example,dc=org
nacos.core.auth.ldap.filter.prefix=uid nacos.core.auth.ldap.case.sensitive=true
nacos.core.auth.ldap.ignore.partial.result.exception=false

3. 启动 Nacos

```
cd ../bin
./startup.sh -m standalone
```

启动后可通过 8848 端口页面访问 Nacos，如 http://10.6.222.10:8848/nacos

用户名密码为：nacos/nacos

4. 微服务部署 nacos-ldap.yaml

```
kubectl apply -f nacos-ldap.yaml
```

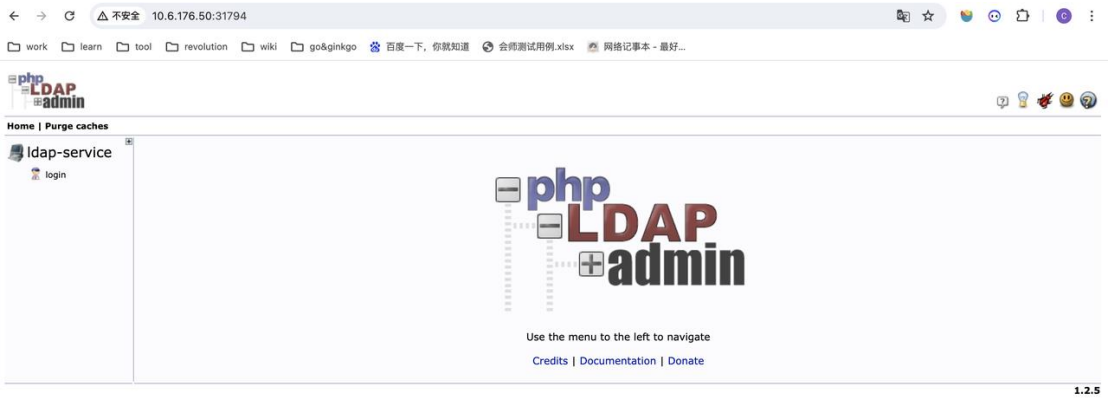
```
yaml title="nacos-ldap.yaml" apiVersion: nacos.io/v1alpha1 kind: Nacos metadata: labels: sidecar.istio.io/inject: 'false' skoala.io/type: nacos name: nacos-ldap namespace: skoala-demo spec: certification: enabled: true ldap: admin: admin # login DN 中的 cn 值，如 login DN 为 "cn=admin,dc=example,dc=org" 时，该值为 admin basedc: dc=example,dc=org # ldap 的 basedc password: admin # ldap 的管理账号密码 url: ldap://10.6.176.50:30326 # ldap 地址 token: RGFvY2xvdWRTa29hbGFOYWNvc0F1dGhTZWNyZXRLZXk= token_expire_seconds: '18000' type: ldap database: mysqlDb: nacosjvm mysqlHost: 10.6.222.10 mysqlPassword: dangerous mysqlPort: '3306' mysqlUser: root type: mysql image: docker.m.daocloud.io/nacos/nacos-server:v2.3.2-slim jvm_percentage: 0.75 mysqlInitImage: docker.m.daocloud.io/arey/mysql-client:latest replicas: 1 resources: limits: cpu: '1' memory: 2Gi requests: cpu: 500m memory: 500Mi serviceType: NodePort type: standalone volume: requests: storage: 1Gi storageClass: default
```

用户管理

通过 LDAP 创建用户：

1. 页面访问 phpLDAPadmin

URL 为前文部署的 phpadmin 的地址：节点 IP:8080 映射端口



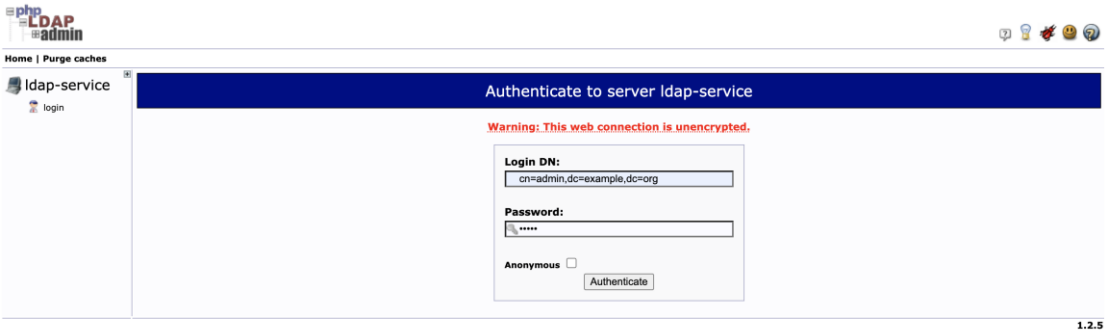
go to webpage

2. 登录 phpLDAPadmin

点击左侧 **login** 登录：

- Login DN 为 cn=admin,dc=example,dc=org
- Password 为 admin

输入 Login DN 和 Password 后，点击 **Authenticate**

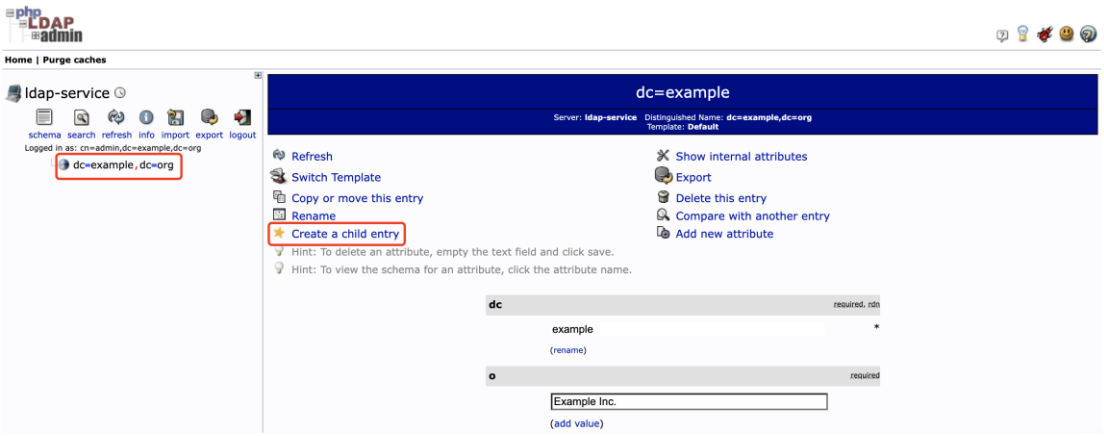


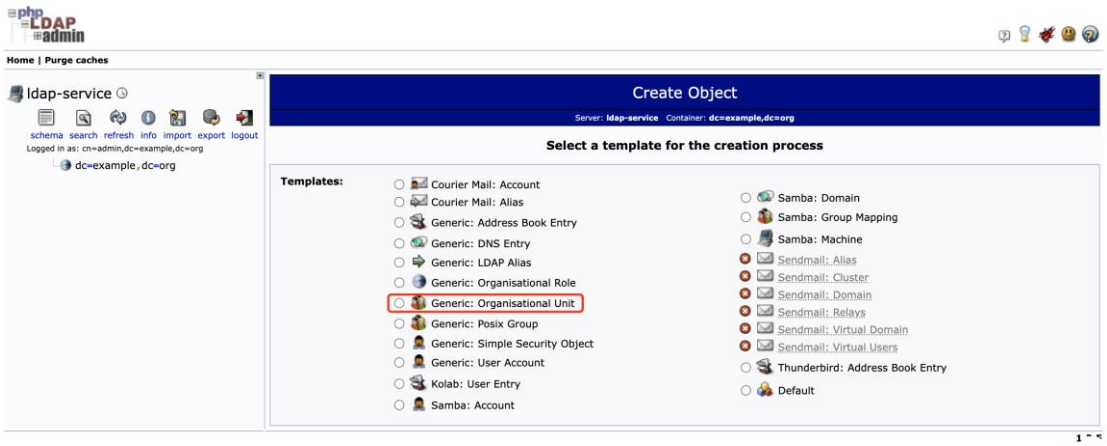
login

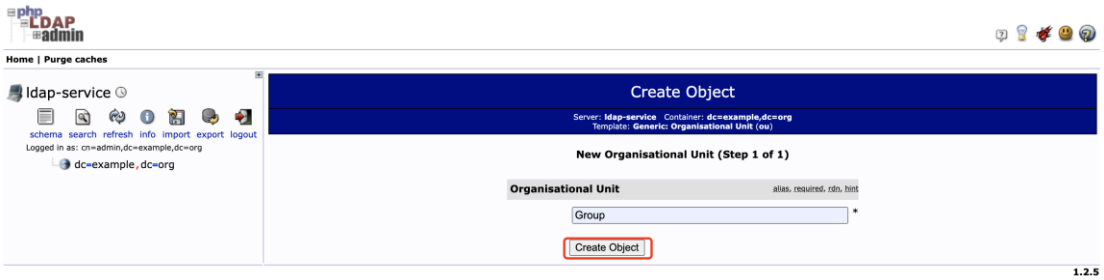
Authenticate

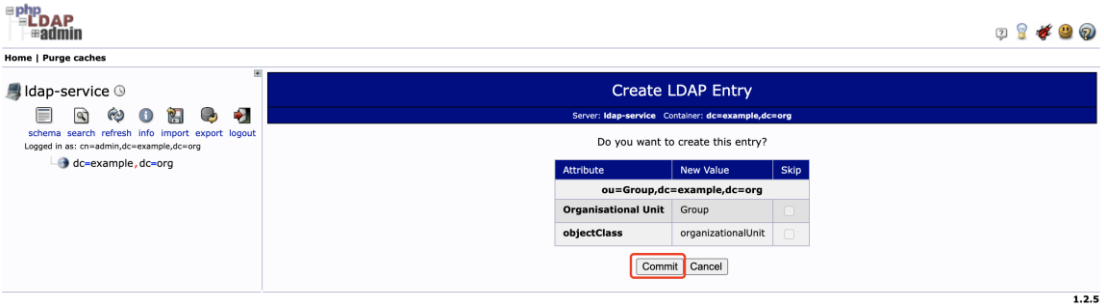
3. 创建用户组和用户

1. **点**击左侧 **dc=example,dc=org 后再点**击 **Create a child entry** ， 选择 **Generic: Organisational Unit** 创建 Group **分**组

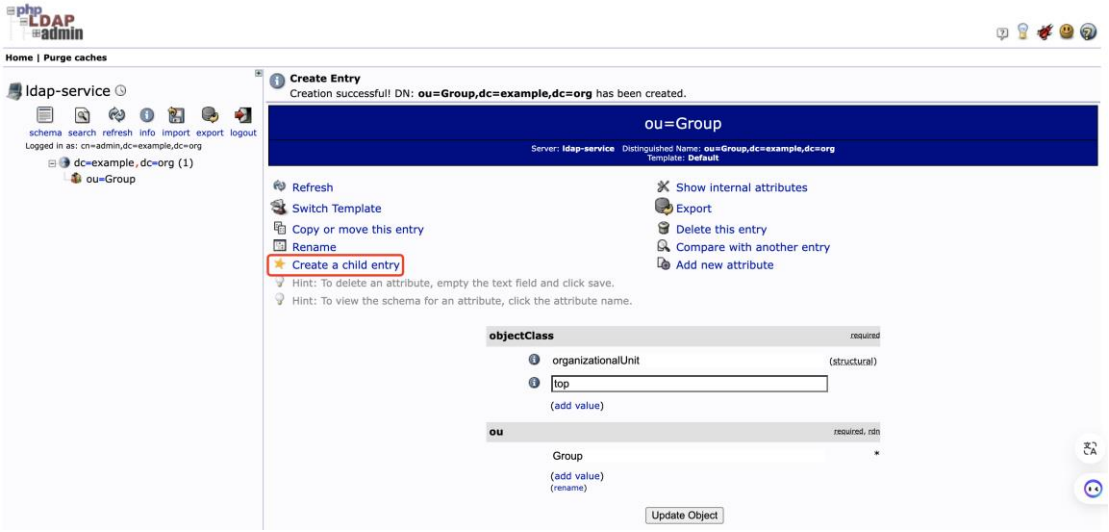


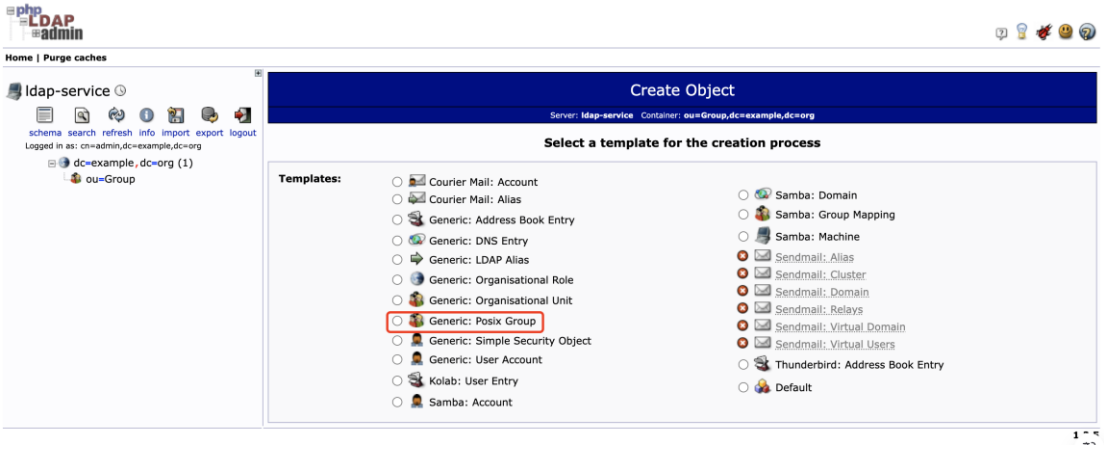


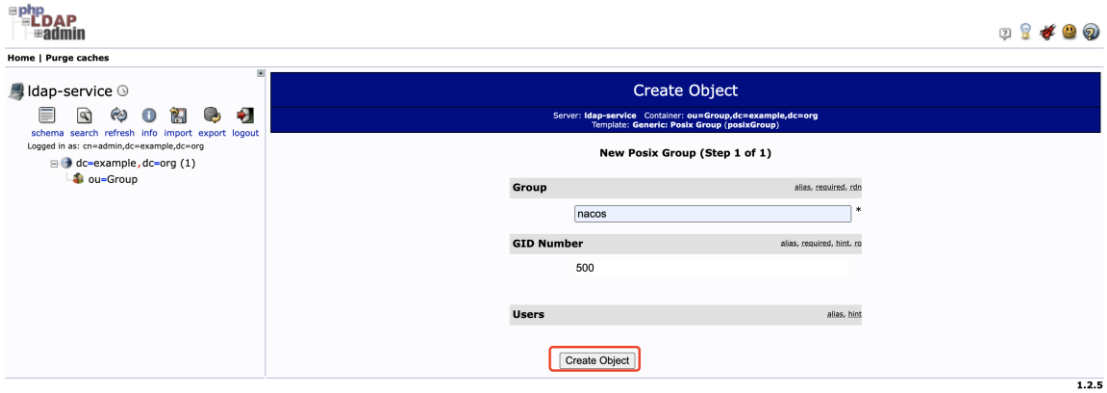


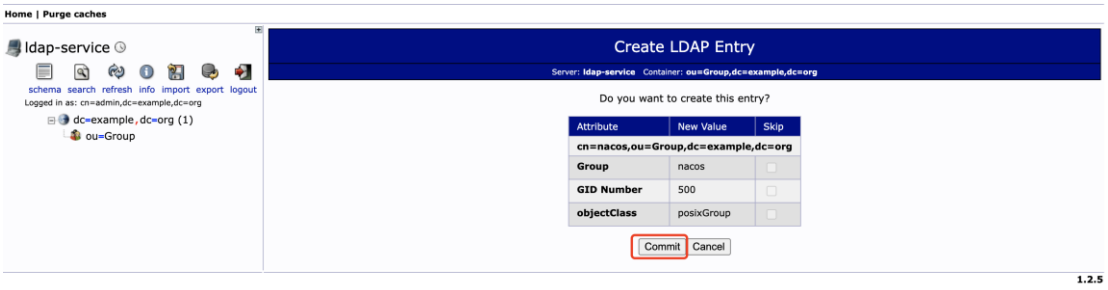


2. 在 **ou=Group** 下创建用户组 nacos

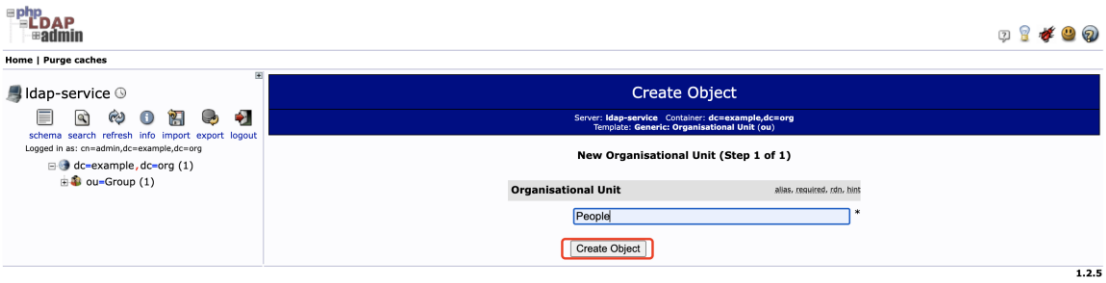




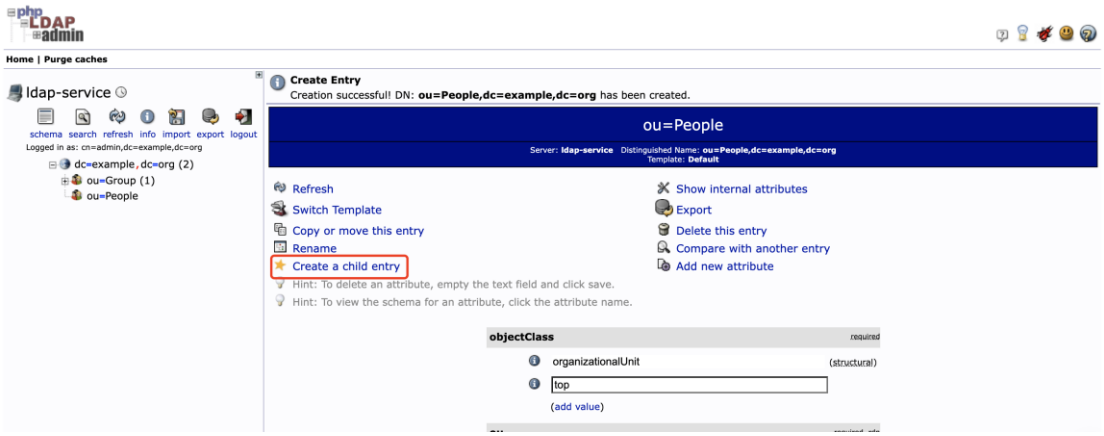


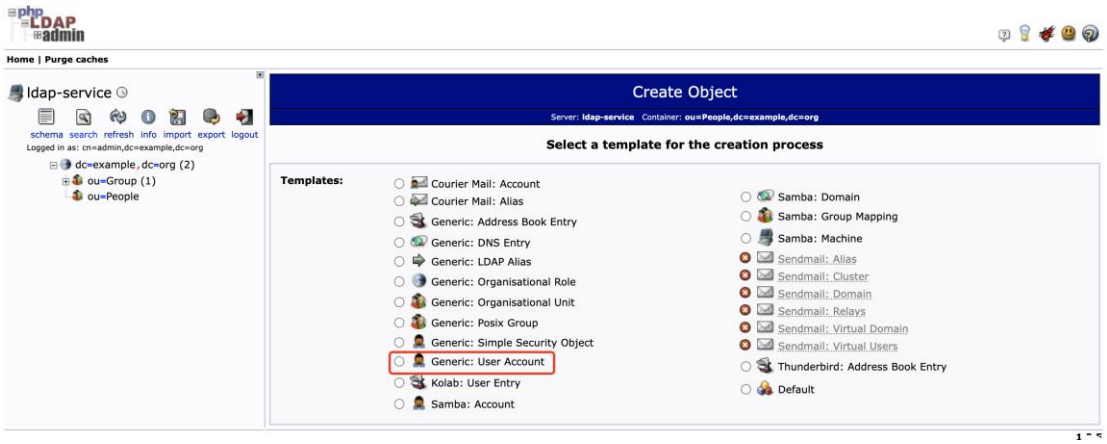


3. 同样点击左侧 **dc=example,dc=org** 后再点击 **Create a child entry** ，选择 **Generic: Organisational Unit** 创建 People 分组

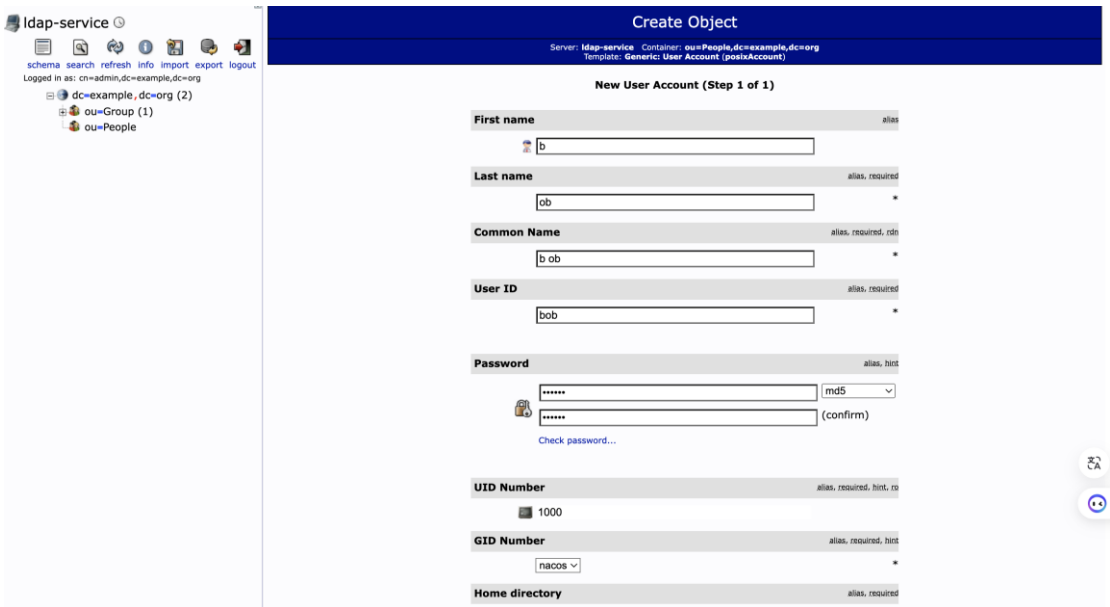


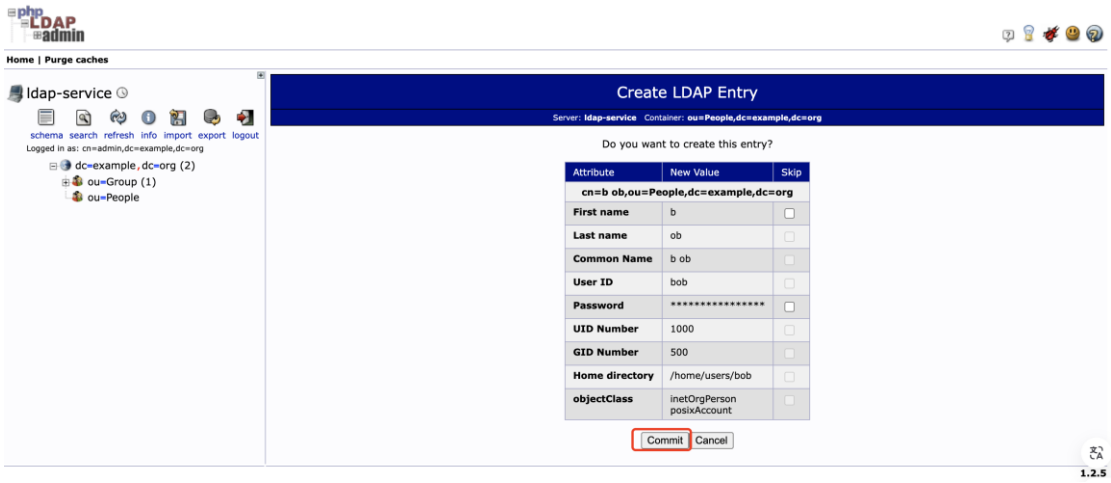
4. 在 **ou=People** 下创建用户 bob


Home | Purge caches
ldap-service
schema search refresh info import export logout
Logged in as: cn=admin,dc=example,dc=org
dc=example,dc=org (2)
ou=Group (1)
ou=People
Create Entry
Creation successful! DN: ou=People,dc=example,dc=org has been created.
ou=People
Server: ldap-service Distinguished Name: ou=People,dc=example,dc=org
Template: Default
Refresh
Switch Template
Copy or move this entry
Rename
Create a child entry
Show internal attributes
Export
Delete this entry
Compare with another entry
Add new attribute
Hint: To delete an attribute, empty the text field and click save.
Hint: To view the schema for an attribute, click the attribute name.
objectClass
organizationalUnit
(structural)
top
(add value)


Home | Purge caches
ldap-service
schema search refresh info import export logout
Logged in as: cn=admin,dc=example,dc=org
dc=example,dc=org (2)
ou=Group (1)
ou=People
Create Object
Server: ldap-service Container: ou=People,dc=example,dc=org
Select a template for the creation process
Templates:
Courier Mail: Account
Courier Mail: Alias
Generic: Address Book Entry
Generic: DNS Entry
Generic: LDAP Alias
Generic: Organisational Role
Generic: Organisational Unit
Generic: Posix Group
Generic: Simple Security Object
Generic: User Account
Kolab: User Entry
Samba: Account
Samba: Domain
Samba: Group Mapping
Samba: Machine
Sendmail: Alias
Sendmail: Cluster
Sendmail: Domain
Sendmail: Relays
Sendmail: Virtual Domain
Sendmail: Virtual Users
Thunderbird: Address Book Entry
Default

5. 参照下图填写参数：


ldap-service
schema search refresh info import export logout
Logged in as: cn=admin,dc=example,dc=org
dc=example,dc=org (2)
ou=Group (1)
ou=People
Create Object
Server: ldap-service Container: ou=People,dc=example,dc=org
Template: Generic: User Account (posixAccount)
New User Account (Step 1 of 1)
First name
b
Last name
ob
Common Name
b ob
User ID
bob
Password
md5
(confirm)
Check password...
UID Number
1000
GID Number
nacos
Home directory



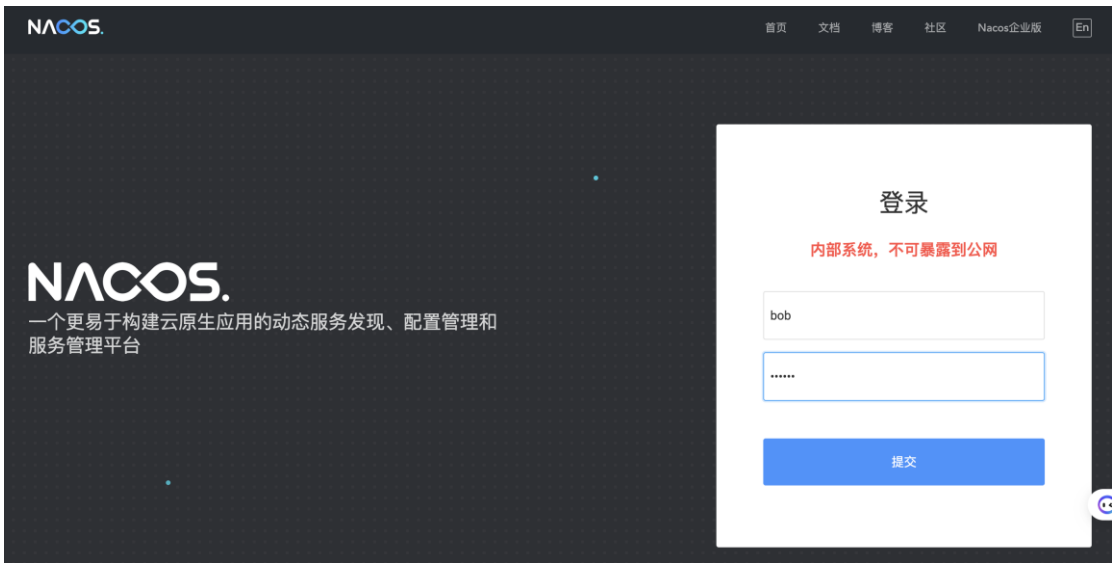
- first name 为 b
- last name 为 ob
- 密码设为 123456
- GID number 选择 nacos

然后点击 **Create Object** ，再点击 **Commit**

4. 在 Nacos 进行用户授权

    1. 使用 LDAP 创建的用户 bob，登录 Nacos 页面

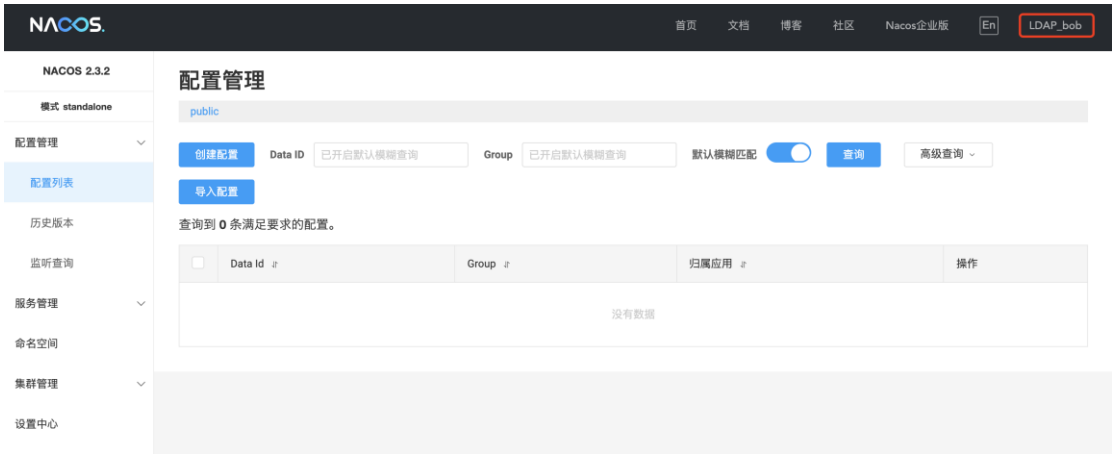
        用户名/密码为：bob/123456



bob

登录后用户名变成 LDAP_bob，此后可使用 LDAP_bob 作为用户名进行登录。

但要注意 LDAP_bob 的密码为 nacos，而不是 123456；bob 登录时密码仍为 123456



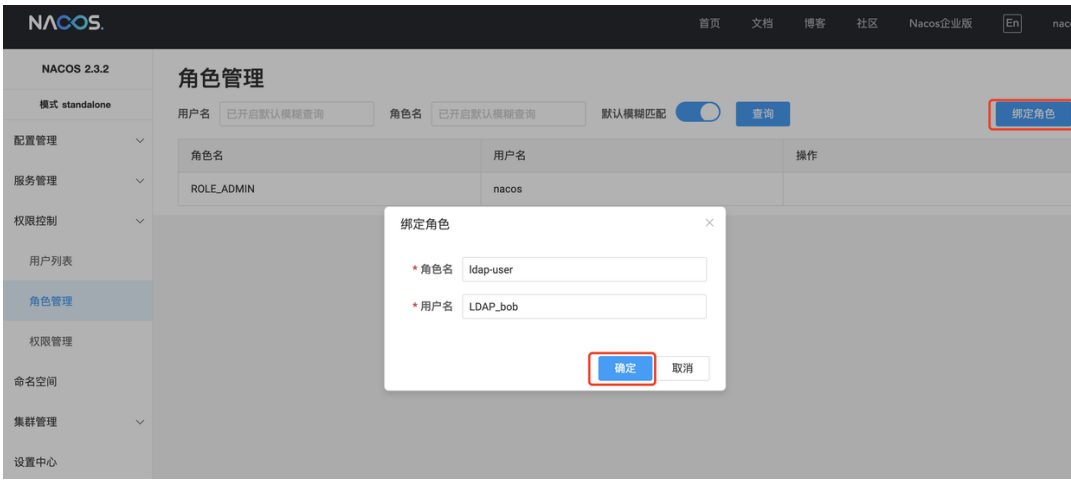
bob

2. 给 LDAP_bob 用户授权

    1. 重新以用户 nacos 登录 nacos 页面，进入 权限控制 -> 用户列表 页面可看到 LDAP_bob 用户

nacos

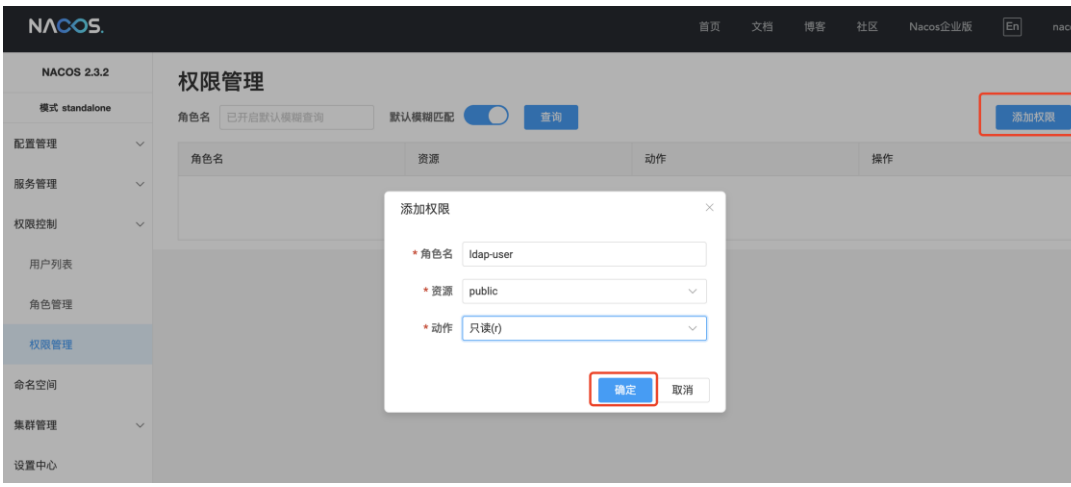
2. 进入角色管理页面，**点**击 **绑定角色** 给用户创建角色



bind

3. 进入权限管理页面，**点**击 **添加权限** 给角色绑定 public **的只**读权限，

则 LDAP_bob **用**户只具有 public **命名空**间下的只读权限。



permission

!!! note

ldap 添加用户 bob，同步到 nacos 那边之后是 LDAP_bob。

ldap 删除用户 bob 后，nacos 端的 LDAP_bob 还能正常使用。

skoala-init 的 x-kubernets-validations 报错问题

将 skoala-init 从 v0.11.0 升级到 v0.12.0 时报错导致无法升级：

unknown field "x-kubernets-validations" in io.k8s.apiextensions-apiserver....

error

error

排查过程

1. Q: 操作步骤是否正确?

   A: 先 kubectl apply CRD(s)，再 helm upgrade，所以此操作步骤无误，排除操作问题.

2. Q: 版本是 QA 测试通过版本，且内部测试时升级功能正常?

   A: 确认无误。

3. Q: 确认使用版本与测试时使用的 k8s 集群版本?

   A: v1.21 (使用版本) ，v1.26 (测试版本)。

4. Q: 至此及根据相应的错误截图，初步推断是客户 k8s 集群版本与相应 CRD 不匹配，同时 skoala-init 为微服务相关的所有的服务提供初始条件配置，即其包含有多个 CRD， 而根据 skoala-init v0.12.0 的 ChangeLog 可知，在此版本中 Gateway-API 的 CRD 有过升级，所以得出：

   A: 由 Gateway-API 相关升级引发。

参考文档

- Rollout, Upgrade and Rollback Planning

  Rollout

  Rollout

- KEP-2876：使用通用表达式语言（CEL）来验证 CRD

  KEP-2876

  KEP-2876
- CRD validation error for x-kubernetes-preserve-unknown-fields: true #88252

解决方案

根据实际情况，选择适合的解决方案。

- 方案一：升级使用的 k8s 集群版本为至少 v1.25
- 方案二：按 CRD validation error for x-kubernetes-preserve-unknown-fields: true #88252

  中所示，添加：

  –validate=false

Nacos 版本降级

如果你使用的是高于 Nacos 2.1.x 的版本，想要将版本降级到低于 2.1.x，并且使用了外置数据库， 可以参照以下步骤来降级版本：

1. 找到需要降级的 Nacos CR 资源

   ```
   kubecl get nacos -A
   ```

2. 修改 image 字段，替换成需要降级的版本
3. 修改数据库

   ```
   alter table config_info drop column encrypted_data_key;
   alter table config_info_beta drop column encrypted_data_key;
   alter table his_config_info drop column encrypted_data_key;
   ```

4. 重启此 Nacos 的 StatefulSet。

网关压力测试报告

本文介绍 DCE 5.0 云原生网关在不同场景下的性能表现，便于您根据需求为网关配置合适的资源。

测试环境

开始测试之前，需要部署 DCE 5.0，下载安装测试工具，并准备好用于测试的压力机。

对象	角色	说明
DCE 5.0 云原生网关	测试对象	采用一主一从的模式，部署位

对象	角色	说明
Locust	测试工具	置：172.30.120.211采用1+4的主从分布式运行模式，四台

对象	角色	说明
Nginx	用于测试网关性能的演示服务	压测机的资源配置均为 8 核 8 G，通过 DCE 5.0 云原生网

对象	角色	说明关访问，访问地址为：http://172.30.120.211:30296/
contour	DCE 5.0 云原生网关的控制面	版本为 1.23.1
envoy	DCE 5.0 云原生网关的数据面	版

对象	角色	说明本为1.24.0版本为0.12.1版本为0.13.1版本为0.15.1
全局管理	DCE 5.0 云原生网关依赖的组件	
容器管理	DCE 5.0 云原生网关依赖的组件	
微服务引擎	DCE 5.0 云原生网关依赖的组件	

性能指标

- 吞吐量（RPS）：每秒处理的请求数。结合 CPU 使用率，判断 DCE 5.0 云原生网关在特定资源配置下每秒可以处理的并发请求数。 吞吐量越高，说明网关性能越好
- CPU 使用率：测试处理特定数量的并发请求时，DCE 5.0 云原生网关实例的 CPU 使用

情况。 当 CPU 使用量达到 90% 以上时，认为 CPU 接近满载，此时的吞吐量（RPS）是当前配置能够正常处理的最大并发请求数。

测试脚本

- 在 Locust Web 机器执行如下命令，收集压测结果

```
docker run -p 8089:8089 --network=host -v $PWD:/mnt/locust locustio/locust -f /mnt/locust/gateway-external-nginx.py --master
```

- 在 Locust 压测机执行如下命令，模拟用户访问，执行压力测试

```
docker run -p 8089:8089 --network=host -v $PWD:/mnt/locust locustio/locust -f /mnt/locust/gateway-external-nginx.py --worker --master-host=172.30.120.210
```

- 压测脚本 gateway-external-nginx.py

```
from locust import task
    from locust.contrib.fasthttp import FastHttpUser

    class ShellCard(FastHttpUser):

      host = "http://172.30.120.211:30296" # 被测服务的访问地址
      @task
      def test(self):
        header = {"Host": "external.nginx"}
        self.client.get("/", headers=header)
```

测试 nginx 吞吐量：三副本，不限制资源

测试结果

并发用户数

每秒吞吐量

CPU 使用量

分析

4
4300

58%——70%

资源未充分利用，**理**论上仍有能力处理更多请求。

8
5700
77%——83%

仍有少量资源空闲，**可以**尝试增加并发请求数，测试吞吐量上限。

12
6700
90%——95%

只有极少量资源空闲，CPU **接近**满载，视为达到吞吐量上限。

!!! success
综上，当服务部署三副本且不限制资源用量时，DCE 5.0 云原生网关能够处理的最大并发请求数大约为 6000 到 7000，较同类产品而言可谓性能优异。

测试过程截图

- **并**发用户数为 4

  4 **个并**发用户

  4 **个并**发用户

  4 **个并**发用户

  4 **个并**发用户

- **并**发用户数为 8

  8 **个并**发用户

  8 **个并**发用户

  8 **个并**发用户

  8 **个并**发用户

- **并**发用户数为 12

  12 **个并**发用户

12 个并发用户

12 个并发用户

12 个并发用户

探究 contour 资源配置对 envoy 的性能影响

DCE 5.0 云原生网关是在开源项目 contour 和 envoy 的基础上进一步研发优化而来。contour 充当网关的控制平面，envoy 充当数据平面。

创建 DCE 5.0 云原生网关时，系统要求必须为网关配置不低于 1 核 1 G 的资源。因此，本次测试中 contour 的资源限制最低为 1 核 1 G。

此外，为了更好地体现 contour 的资源配置的影响，将 envoy 的资源限制设置为 6 核 3 G， 保证 envoy 自身始终具有较高的性能，不会因为自身资源不足而影响测试结果。

为保证压测机器资源负载正常，默认 Locust users 为 8。

测试结果

contour 资源规格

每秒吞吐量

CPU 使用量

分析

1 核 1 G

3700
53%——69%

contour 的资源配置从 1 核 1 G 到 2 核 1 G，再到 3 核 2 G，但网关每秒吞吐量一直维持在 3700 左右，上下变动幅度只有 100。CPU 使用率也维持在 50%——70%左右，整体

变化非常小。

2 核 1 G

3600
55%——72%

3 核 2 G

3800
57%——69%
!!! success
这说明 contour 的资源配置对 envoy 的性能几乎没有影响。

测试过程截图

- contour 资源 1 核 1 G

  1c1g

  1c1g

  1c1g

  1c1g

- contour 资源 2 核 1 G

  2c1g

  2c1g

  2c1g

  2c1g

- contour 资源 3 核 2 G

  3c2g

  3c2g

  3c2g

  3c2g

探究 envoy 资源配置对吞吐量的影响

envoy 固定为 1 副本，contour 配置为 1 核 1 G，被测试服务 nginx 配置为 3 副本，不限制资源用量。

测试结果

envoy 资源规格

并发用户数

每秒吞吐量

CPU 使用量

分析

1 核 1 G

4
1016
18%——22%

当 envoy 的资源配置不变时，即使并发用户数成倍增加，网关吞吐量也变化不大。

8
1181
19%——20%
16
1090
19%——22%

2 核 1 G

4
2103
28%——41%

并发用户数为 4 时，当资源配置从 1 核增加到 2 核，吞吐量也增加了 1000 左右。

8
2284
38%——47%

3 核 1 G

8
3355
59%——70%

并发用户数为 8 时，当资源配置从 1 核增加到 3 核，吞吐量也增加了 2000 左右。

12

3552
52%——59%

4 核 2 G

8
3497
58%——80%

并发用户数为 12 时，当资源配置从 3 核增加到 5 核，吞吐量也增加了 1000 左右。

12
4250
78%——86%

5 核 2 G

8
3573
60%——81%
12
4698
68%——78%

6 核 2 G

12
4574
78%——85%

当配置为 6 核 2 G 时，吞吐量为 5400 左右，CPU 使用量也到达了 90% 以上，接近满载。

16
5401

90%以上

!!! success
综上：

- envoy 的 CPU 配置对吞吐量起决定性因素。
- 在当前压测资源下，通过 envoy 访问 nginx 的吞吐量能够达到直接访问 nginx 的吞吐量的 80% 以上。

测试过程截图

当 envoy 配置为 1 核 1 G

- **并**发用户**数**为 4

  4

  4

  4

  4

- **并**发用户数为 8

  8

  8

  8

  8

- **并**发用户数为 16

  16

  16

  16

  16

当 envoy 配置为 2 核 1 G

- **并**发用户数为 4

  4

  4

  4

  4

- **并**发用户数为 8

  8

  8

  8

  8

当 envoy 配置为 3 核 1 G

- 并发用户数为 8

  8

  8

  8

  8

- 并发用户数为 12

  12

  12

  12

  12

当 envoy 配置为 4 核 2 G

- 并发用户数为 8

  8

  8

  8

  8

- 并发用户数为 12

  12

  12

  12

  12

当 envoy 配置为 5 核 2 G

- 并发用户数为 8

  8

  8

  8

  8

- 并发用户数为 12

  12

  12

12

12

当 envoy 配置为 6 核 2 G

- 并发用户数为 12

12

12

12

12

- 并发用户数为 16

16

16

16

16

Nacos 压力测试报告

本文通过 JVM 虚拟机测试托管 Nacos 注册配置中心在不同资源规格下的性能表现，便于您根据实际的业务场景为 Nacos 提供合适的资源配额。

性能指标

TPS：通常指服务器每秒处理的事务数 (Transactions Per Second)。在本文中具体指 Nacos 服务器每秒可以处理多少个服务注册请求。

测试命令

java -jar -Xms2g -Xmx2g -Dnacos.server=10.6.222.21: 30168 -Dthead.count=100 -Dservice.count =500 nacos-client-1.0-SNAPSHOT-jar-with-dependencies.jar

- -Xms2g：JVM 的初始内存
- -Xmx2g：JVM 可使用的最大内存

- -Dnacos.server：Nacos 注册配置中心的访问地址
- -Dthead.count：线程数，对应测试结果中的并发数
- -Dservice.count：每个线程的请求数，对应测试结果中的服务数
- nacos-client-1.0-SNAPSHOT-jar-with-dependencies.jar：压测使用的 Jar 包，实际测试过程中在本地将其重命名为 nacos-benchmark.jar

测试过程截图

process

process

测试结果

规格

节点数

并发数

服务数

执行时间(毫秒)

TPS
1 Core 2 GiB
1
30
1000
17086
1755.8
50
1000
23315
2144.5
100
500
26194
1908.4

2 Core 4 GiB
1
30
1000
8794
3409.1
50
1000
15460
3225.8
100
500
15699
3184.7
4 Core 8 GiB
1
50
1000
11843
4221.9
80
1000
14510
5513.4
100
500
9501
5262.6
8 Core 16 GiB
1
50
2000
14391
6948.8
80
2000
18695
8558.4
100
1000
11686
8557.2

探究 JVM 资源配置对 Nacos 的性能影响

为探究 JVM 的资源配置对托管 Nacos 注册配置中心的性能影响，在 Nacos 申请资源为 2 Core 4 GiB 时， 为 JVM 也配置相同的资源，即 2 Core 4 GiB。

这种情况下得出的测试结果如下：

规格

节点数

并发数

服务数

执行时间(毫秒)

TPS
2 Core 4 GiB
1
30
1000
11003
2727.3
50
1000
15006
3333.3
100
500
13963
3580.9
!!! success

TPS 与调整前的数据不相上下。这说明，调整 JVM 对 Nacos 的性能基本没有影响。

微服务

微服务可以指一种通过多个小型服务组合来构建单个应用的架构风格，也可以指构成该应

用的各个小型服务，即一些协同工作、小而自治的服务。 微服务架构具有技术异构、故障隔离、敏捷开发、独立部署，弹性扩缩、代码可复用等优势，因此越来越多的企业开始采用微服务架构开发应用。 但是，随着业务的不断扩展，业务逻辑越来越复杂，微服务的数量呈现出爆发式增长的态势。错综复杂的服务间调用关系、依赖关系影响着整个应用的稳定性和可用性，大大增加了运维的成本。因此，微服务治理工具应运而生。

- 传统微服务

  传统微服务指基于传统的微服务注册中心发展而来的微服务治理模式和技术体系。 传统的微服务框架主要有 Spring Cloud、Dubbo 等，传统的注册中心主要有 Eureka、Nacos、Zookeeper、Consul、etcd 等。

- 云原生微服务

  云原生微服务指基于 Kubernetes 发展而来的跨技术栈微服务治理模式，自无框架以来逐步向服务网格发展。 这种模式使用云原生的框架（如 Kubernetes、Istio、Linkered），云原生的注册中心（如 Kubernetes 注册中心和 Mesh 注册中心）和云原生的网关（如 Nginx、Envoy、Contour）。

- 微服务框架

  微服务框架可以分为侵入式和非侵入式两种。侵入式框架以 Spring Cloud 和 Dubbo 为代表，需要对旧代码进行改造。 非侵入式框架以 Istio 和 Linkerd 为代表，在原有基础上加入边车和服务网格，无需更改旧代码。

- Spring Cloud 框架

  Spring Cloud 是一款基于 Spring Boot 的微服务框架。 Spring Cloud 将市面上成熟的、经过验证的微服务框架整合起来，在 Spring Boot 的基础上进行再封装，屏蔽复杂的配置和实现原理， 最终为开发人员提供了一套简单易懂、易部署和易维护的分

布式系统开发工具包。 Spring Cloud 包括 Spring Cloud Config、Spring Cloud Bus 等 20 多个子项目，提供了服务治理、服务网关、智能路由、负载均衡、熔断器、监控跟踪、分布式消息队列、配置管理等领域的解决方案。

- Dubbo 框架

  Apache Dubbo 微服务开发框架提供了 RPC 通信与微服务治理两大关键能力。 这意味着，使用 Dubbo 开发的微服务将具备相互之间的远程发现与通信能力。 同时，利用 Dubbo 丰富的服务治理能力，可以实现诸如服务发现、负载均衡、流量调度等服务治理诉求。 Dubbo 是高度可扩展的，用户几乎可以在任意功能点去定制自己的实现，从而改变框架的默认行为以满足自己的业务需求。 Dubbo 目前支持 Consul、Nacos、ZooKeeper、Redis 等多种开源组件作为注册中心。

- Sentinel

  Sentinel 是面向分布式服务架构的流量控制组件，主要以流量为切入点，从流量控制、熔断降级、系统自适应保护等多个维度来保障微服务的稳定性。 Sentinel 具有丰富的应用场景，承接了阿里巴巴近 10 年的双十一大促流量的核心场景。Sentinel 具有完备的实时监控，用户可以在控制台中看到接入应用的单台机器的秒级数据。 Sentinel 提供与其它开源框架 / 库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合，可以开箱即用。 此外，Sentinel 还能提供简单易用、完善的 SPI 扩展接口，用户可以通过实现扩展接口来快速地定制逻辑。

注册中心

注册中心相当于微服务架构中的”通讯录“，负责记录服务和服务地址的映射关系。 微服务在启动时，将自己的网络地址等信息注册到注册中心，注册中心存储这些数据。 服务消费

者从注册中心查询服务提供者的地址，并通过该地址调用服务提供者的接口。 各个微服务与注册中心使用一定机制通信。如果注册中心与某微服务长时间无法通信，就会注销该实例。如果微服务网络地址发生变化时，会重新注册到注册中心。

传统的服务注册与发现

服务注册是指，微服务启动时通过集成的服务发现 SDK 向注册中心发送注册信息，注册中心在收到服务注册请求后将该服务的基本信息存储下来。 服务发现是指，在服务注册之后，如有服务消费者要调用该服务，调用方可以通过服务发现组件从注册中心查询目标微服务的地址列表，并通过获取到的服务地址列表，以某种负载策略向目标微服务发起调用。传统的服务注册与发现主要由传统的注册中心提供支持，例如 Nacos、Eureka、Zookeeper 等。

- Eureka 注册中心

  Eureka 是 Spring Cloud 官方推荐使用的注册中心。 Eureka 主要适用于通过 Java 实现的分布式系统，或是与 JVM 兼容语言构建的系统。 但是，由于 Eureka 服务端的服务治理机制提供了完备的 RESTful API，所以也支持将非 Java 语言构建的微服务应用纳入 Eureka 的服务治理体系中来。 只是在使用其他语言平台的时候，需要自己来实现 Eureka 的客户端程序。 Eureka 集群采用非 Master/Slave 架构，集群中所有节点角色一致，数据写入集群任意一个节点后，再由该节点向集群内其他节点进行复制实现弱一致性同步。

- Nacos 注册中心

  Nacos 注册中心是阿里巴巴推出的开源项目，可以轻松构建云原生应用程序和微服务平台。 Nacos 致力于发现、配置和管理微服务。Nacos 提供了一组简单易用的特

性集，主要包括服务发现和服务健康监测、动态配置服务、动态 DNS 服务、服务及其元数据管理。 Nacos 支持几乎所有类型的服务，例如，Dubbo/gRPC 服务、Spring Cloud RESTful 服务或 Kubernetes 服务等。

- Zookeeper 注册中心

  Apache ZooKeeper 是开放源码的分布式应用程序协调组件，为分布式应用提供一致性服务的软件。 Zookeeper 提供的功能包括配置维护、域名服务、分布式同步、组服务等。 在微服务项目开发中，ZooKeeper 经常和 Dubbo 配合使用，充当服务注册中心。

云原生的服务注册与发现

- Kubernetes 注册中心

  在 Kubernetes 中能够为微服务提供内部服务注册发现的基础就是 Service 类型的资源。Kubernetes 使用 DNS 作为服务注册表。 为了满足这一需要，每个 Kubernetes 集群都会在 kube-system 命名空间中用 Pod 的形式运行一个 DNS 服务 (kube-dns/coredns)，通常称之为集群 DNS。 服务注册过程是：

  1. 向 API Server 用 POST 方式提交一个新的 Service 定义。
  2. 该请求需要经过认证、鉴权以及其他的准入策略检查过程之后才会放行。
  3. Service 得到一个 ClusterIP（虚拟 IP 地址），并保存到集群数据仓库。
  4. 在集群范围内传播 Service 配置。
  5. 集群 DNS 服务得知该 Service 的创建，据此创建必要的 DNS 记录。

     Service 对象注册到集群 DNS 之中后，就可以通过一个单一稳定的 IP 地址访问该 Service 中的 Pod。

Service Mesh 注册中心

Service Mesh 作为微服务的注册中心是基于 Kubernetes 实现的。 在 Kubernetes 集群环境中安装 Istio 后，Kubernetes 在创建 Pod 时， 就会通过 Kube-API 服务器调用控制面组件的 Sidecar-Injector 服务、自动修改应用程序的描述信息并将其注入 Sidecar， 之后在创建业务容器的 Pod 中同时创建 Sidecar 代理容器。SideCar 则会通过 xDS 协议与 Istio 控制面各组件连接。

云原生网关

网关是介于客户端和服务器端之间的中间层，所有的外部请求都需要经过网关层。 而云原生网关指的是基于云原生可声明式 API 概念发展而来的、与业务解耦的网关，例如 Zuul、Kong、Nginx、Spring Cloud Gateway、Envoy 等。 云原生网关最显著的特点是，可以通过声明的方式定义网关的运行时配置，可以通过控制面的声明自动生成配置。 云原生网关的功能和微服务网关基本类似，例如身份验证、路由、监控、负载均衡、缓存、服务升降级、静态响应处理、流量控制、日志、重试、熔断等。

- Envoy

  Envoy 是面向服务架构的高性能网络代理，拥有强大的定制化能力。 Envoy 是面向服务架构设计的 L7 代理和通信总线，核心是一个 L3/L4 网络代理。 Envoy 通常以 Sidecar 的方式运行在应用程序的周边，也可以作为网络的边缘代理来运行。Envoy 的特性包括：进程外体系结构、L3/L4 过滤器体系结构、HTTP L7 过滤器体系结构、一流的 HTTP/2 支持、 HTTP/3 支持、HTTP L7 路由、gRPC 支持、服务发现和动态配置、健康检查、高级负载平衡、前端/边缘代理支持、一流的可观察性等。

- Contour

  Contour 将 Envoy 部署为反向代理和负载均衡器，从而充当 Kubernetes 的入口控制器。 Contour 支持动态配置更新，而且还引入了一个新的入口 API (HTTPProxy)。该 API 通过自定义资源定义 (CRD) 实现，目标是扩展 Ingress API 的功能，提供更丰富的用户体验并解决原始设计中的缺陷。 Contour 架构灵活，可以部署为 Kubernetes Deployment 或 Daemonset。此外，Contour 还具有 TLS 证书授权特性，管理员可以安全地委派通配符证书访问。