天翼云

微服务云应用平台

用户开发指南

天翼云科技有限公司

目 录

1 微服务开发指南

1.1 概述

1.1.1 开发简介

微服务简介

随着微服务架构模式被越来越多的开发者作为应用系统构建的首选，稳定可靠的微服务运行环境变的非常重要。

微服务引擎（CSE）是应用管理与运维平台（ServiceStage）针对微服务解决方案提供的一站式管理平台，使用微服务引擎，开发者可以更加专注于业务开发，提升产品交付效率和质量。微服务架构模式通常包含如下内容：

- 微服务之间的 RPC 通信。微服务架构模式要求微服务之间通过 RPC 进行通信，不采用其他传统的通信方式，比如共享内存、管道等。常见的 RPC 通信协议包括 REST、gRPC、Web Service 等。使用 RPC 通信，能够降低微服务之间的耦合，提升系统的开放性，减少技术选型的限制。一般建议采用业界标准协议，比如 REST。对于性能要求非常高的场景，也可以考虑私有协议。
- 分布式微服务实例和服务发现。微服务架构特别强调架构的弹性，业务架构需要支持微服务多实例部署来满足业务流量的动态变化。微服务设计一般会遵循无状态设计原则，符合该原则的微服务扩充实例，能够带来处理性能的线性提升。当

实例数很多的时候，就需要有一个支持服务注册和发现的中间件，用于微服务之间的调用寻址。

- 配置外置及动态、集中的配置管理。随着微服务和实例数的增加，管理微服务的配置会变得越来越复杂。配置管理中间件给所有微服务提供统一的配置管理视图，有效降低配置管理的复杂性。配置管理中间件搭配治理控制台，可以在微服务运行态对微服务的行为进行调整，满足业务场景变化、不升级应用的业务诉求。
- 提供熔断、隔离、限流、负载均衡等微服务治理能力。微服务架构存在一些常见的故障模式，通过这些治理能力，能够减少故障对于整体业务的影响，避免雪崩效应。
- 调用链、集中日志采集和检索。查看日志仍然是分析系统故障最常用的手段，调用链信息可以帮助界定故障和分析性能瓶颈。

有很多开源框架，比如 Apache ServiceComb Java Chassis（简称 Java Chassis）、Spring Cloud、Apache Dubbo（简称 Dubbo）、Go Chassis 等，实现了微服务架构模式。微服务引擎支持这些开源的微服务框架接入并使用微服务架构模式的注册发现、集中配置、服务治理等功能。

开发能力要求

本文档的主要目的就是说明这些开源微服务开发框架如何接入和使用微服务引擎的功能，假设您已经熟悉和掌握如下开发能力：

- 使用 Java 或者 Go 语言进行微服务开发。假设您已经基于一种 ServiceStage 支持的微服务开发框架开发了应用系统，并期望将应用系统托管在微服务引擎上运行。本文档提供微服务应用接入微服务引擎的相关技术支持。开源微服务开发框架如何使用不是本文档的范围，您可以通过开源社区获取相关微服务开发框架的入门材料和开发指南。
- 理解注册中心、配置中心在微服务应用中的作用，并在项目中搭建和使用注册中心。不同的微服务开发框架默认支持的开源注册中心会有差异，理解注册中心的作用，可以更加容易的更换注册中心。
- 熟悉虚机或者容器环境部署应用，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 应用组件部署”。

微服务开发框架版本说明

微服务开发框架支持的版本和推荐版本如下表所示，如果您已经使用低版本的微服务开发框架构建应用，建议升级到推荐版本，以获取最稳定和丰富的功能体验。

框架	支持版本	推荐版本	接入说明
Java Chassis	2.1.3 及以上	2.3.0 及以上	可以直接使用开源项目提供的软件包接入，不需要引用其他第三方软件包。
Java Chassis	2.0.0~2.1.2	-	可以直接接入微服务引擎专享版。 接入微服务引擎专业版，需要额外引入由 CSE SDK 提供的 foundation-auth 软件包。
Java Chassis	1.3.*x*	-	

框架	支持版本	推荐版本	接入说明
			说明 新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。
Go Chassis	2.x.x	2.2.0 及以上	可以直接使用开源项目提供的软件包接入，不需要引用其他第三方软件包。
Go Chassis	1.x.x	-	可以直接使用开源项目提供的软件包接入，不需要引用其他第三方软件包。
Spring Cloud	1.5.1-Hoxton 及以上	1.6.0-Hoxton 及以上	采用 Spring Cloud Huawei 项目提供接入支持。推荐版本为 Spring Cloud Huawei 的版本号。
Spring Cloud	1.5.1-Greenwich 及以上	-	采用 Spring Cloud Huawei 项目提供接入支持。
Spring Cloud	1.5.1-Finchley 及以上	-	采用 Spring Cloud Huawei 项目提供接入支持。
Apache Dubbo	1.3.x （Dubbo 2.7.x）	1.3.6 及以上	采用 Dubbo ServiceComb 项目提供接入支持。推荐版本为 Dubbo Serivcecomb 的版本号。
Apache Dubbo	1.1.x （Dubbo 2.6.x）	-	采用 Dubbo ServiceComb 项目提供接入支持。

1.1.2 常用概念

微服务引擎核心概念

- 应用：可以将应用理解为完成某项完整业务场景的软件系统。应用一般由多个微服务组成，应用里面的微服务能够相互发现和调用。
- 微服务：完成某项具体业务功能的软件系统。微服务是独立开发、部署的单元。
- 微服务实例：将微服务采用部署系统部署到运行环境，就产生了实例。可以将实例理解为一个进程，一个微服务可以部署若干实例。
- 微服务环境：服务中心建立的一个逻辑概念，比如 development、production 等。不同环境里面的微服务实例逻辑隔离、无法相互发现和调用。

ServiceStage 核心概念

ServiceStage 也存在几个对应的概念，在实际使用过程中容易发生混淆，包括：

- 应用：ServiceStage 的应用和微服务引擎的应用在含义上类似。ServiceStage 的应用名称是部署环节指定的，微服务引擎的应用是微服务开发环节指定的，两者没

有必然联系，可以不同。微服务开发框架都提供了环境变量映射的功能，通过环境变量映射的功能，开发者可以使用 ServiceStage 的应用名称作为微服务的应用名称。

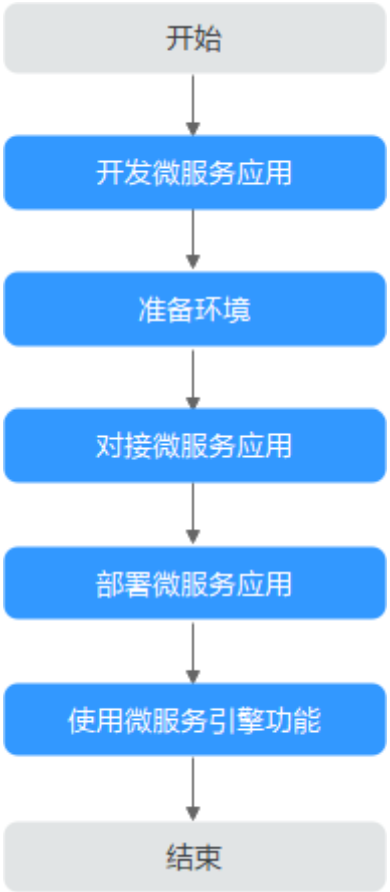
- 组件：ServiceStage 的组件对应于微服务引擎的微服务。和应用一样，名称分别是部署环节指定和开发环节指定，通过环境变量映射，微服务引擎可以使用 ServiceStage 的组件名称。
- 实例：ServiceStage 的实例对应于微服务引擎的实例。只要部署成功，ServiceStage 就有实例了。微服务引擎要求必须成功注册到服务中心，才会有实例。在服务正常注册的情况下，两者实例数是一样的，注册失败的情况下，微服务引擎没有实例，而 ServiceStage 有实例。
- 环境：ServiceStage 的环境和微服务引擎的环境概念不同。ServiceStage 的环境是一系列资源组成的运行环境，环境里面包含了微服务引擎、云容器引擎（CCE）等资源。

1.1.3 开发流程

开发流程概述

开发应用和使用微服务引擎，需要经过如图 1-1 所示的几个阶段。

图1-1 开发流程



开发流程说明

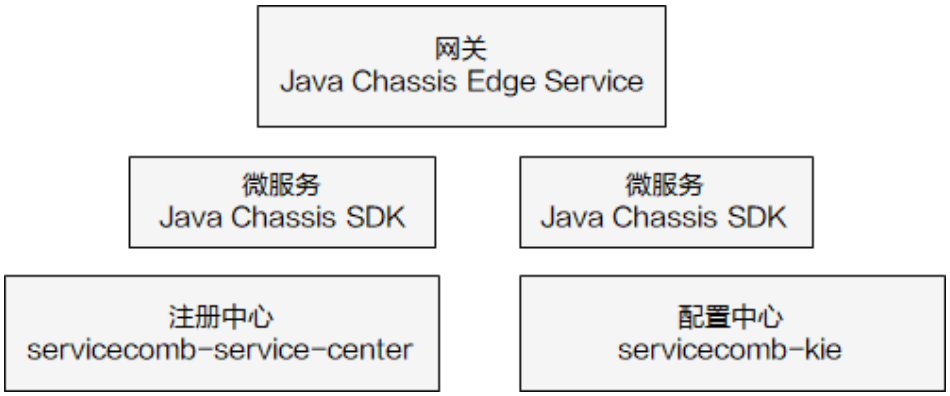
1. 开发微服务应用

   如果您已经完成了微服务应用的开发，可以跳过本流程，进入准备环境。

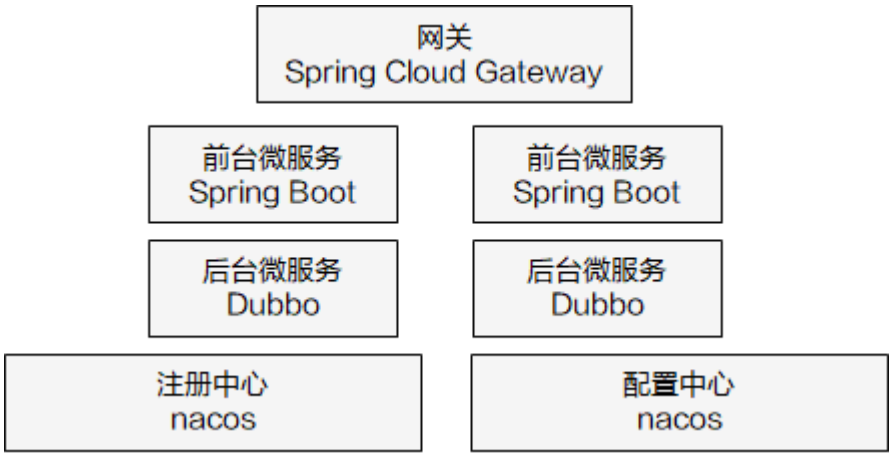
   进行微服务应用开发，首先需要进行技术选型。技术选型是一个复杂的问题，技术决策者需要考虑使用的技术是否容易被团队成员掌握，技术能否满足项目对于功能、性能、可靠性方面的要求，还需要考虑商业服务等多方面的因素。本文档不探讨技术选型，假设技术团队已经选择了适合自己的开发框架。大部分技术团队都会选择开源框架来构建业务。

   开发微服务应用的具体内容，请参考开发微服务应用。

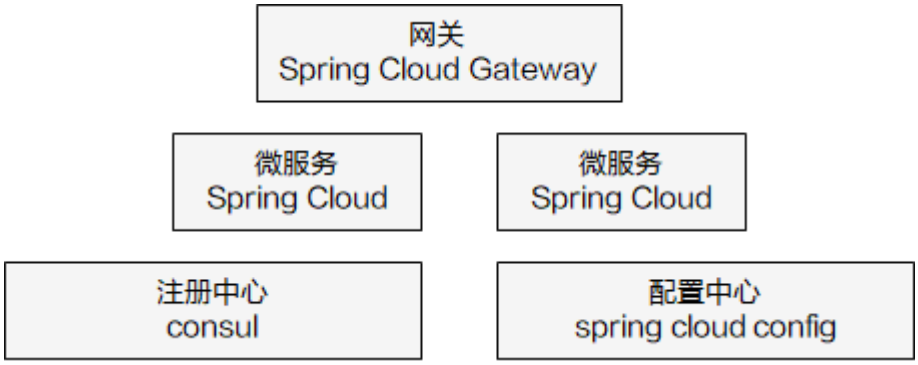
   - 使用 Java Chassis，通常会使用下面的技术进行本地微服务开发：

   - 使用 Dubbo，通常会使用下面的技术进行本地微服务开发：

   - 使用 Spring Cloud，通常会使用下面的技术进行本地微服务开发：

2. 准备环境

创建云上环境，以支持微服务引擎接入调试、云上应用部署和使用微服务引擎功能。一般情况下，会创建一个测试环境和一个生产环境。通过 ServiceStage，能够非常方便的管理云上环境，详细内容请参考准备环境。

3. 对接微服务应用

   用于微服务应用对接微服务引擎，涉及到对已经开发好的应用的配置文件、构建脚本的修改。修改完成后，需要对应用重新编译、打包，通过 ServiceStage 将应用包部署到微服务引擎，详细内容请参考对接微服务应用。

4. 部署微服务应用

   开发完成的微服务应用，通过 ServiceStage 部署到微服务引擎，详细内容请参考部署微服务应用。

5. 使用微服务引擎功能

   对于持续发展的应用系统，都会持续完善和迭代，每个迭代可能需要对微服务应用进行更新升级，需要使用更多的微服务引擎功能。持续迭代的功能演进，会重复上面的应用开发、编译、打包和部署环节。详细内容请参考使用微服务引擎功能。

1.2 开发微服务应用

如您已经完成了微服务应用的开发，请跳过本章节。

开源社区提供了丰富的开发资料和帮助渠道帮助您使用微服务开发框架。如您需深入了解具体微服务框架下的微服务应用开发，请参考本章节给出的参考资料链接。

体验微服务引擎最快捷的方式是使用“微服务引擎推荐示例”里面的例子。下载示例，修改配置文件中的微服务引擎地址，AK/SK 信息，在本地运行例子，这些例子可以注册到微服务引擎。

- Java Chassis

  源码仓库：https://github.com/apache/servicecomb-java-chassis

  问题咨询：https://github.com/apache/servicecomb-java-chassis/issues

  开发指南：https://servicecomb.apache.org/references/java-chassis/zh_CN/

  微服务引擎推荐示例：https://github.com/apache/servicecomb-samples/tree/master/basic

- Go Chassis

  源码仓库：https://github.com/go-chassis/go-chassis

  问题咨询：https://github.com/go-chassis/go-chassis/issues

  开发指南：https://go-chassis.readthedocs.io/en/latest/

  微服务引擎推荐示例：https://github.com/go-chassis/go-chassis/tree/master/examples/discovery

- Spring Cloud

  源码仓库：https://github.com/spring-cloud

  问题咨询：参考源码仓库的各个代码仓库下的 issues

  开发指南：https://spring.io/projects/spring-cloud

Spring Cloud Huawei 项目：https://github.com/huaweicloud/spring-cloud-huawei

微服务引擎推荐示例：https://github.com/huaweicloud/spring-cloud-huawei-samples/tree/master/basic

- Dubbo

  源码仓库：https://github.com/apache/dubbo

  问题咨询：https://github.com/apache/dubbo/issues

  开发指南：https://dubbo.apache.org/zh/

  Dubbo ServiceComb 项目：https://github.com/huaweicloud/dubbo-servicecomb

  微服务引擎推荐示例：https://github.com/huaweicse/dubbo-servicecomb-samples/tree/master/basic

1.3 准备环境

环境准备包括本地开发调试环境和云上环境准备。

准备本地开发调试环境

本地开发调试环境用于搭建一个简易的测试环境，可以有以下两种选择：

- 下载本地轻量化微服务引擎。
- 使用微服务引擎专业版或者使用微服务引擎专享版，并开放公网访问的 IP，保证本地环境能够访问。

说明

新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。

准备云上环境

微服务应用部署到云上，需要先准备云上环境。准备环境一般包含如下任务：

- 获取 AK/SK 及项目名称，请参考获取 AK/SK 与项目名称。

说明

- 如果使用微服务引擎专业版，需要配置 AK/SK。
- 如果使用微服务引擎专享版，不需要配置 AK/SK。

- 创建微服务引擎，请参考“帮助中心 > 应用管理与运维平台 >用户指南 > 基础设施 > 微服务引擎（CSE）> 创建微服务引擎专享版”章节。
- 创建环境，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 环境管理”章节。创建的环境，需包含 CCE 集群、ELB 及微服务引擎等资源。
- 创建应用，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 创建应用”章节。

常用环境变量说明

通过 ServiceStage 管理环境和部署应用，能够简化用户的配置。ServiceStage 会设置一些环境变量，供应用使用，常用的环境变量包括下表所示内容：

表1-1 常用环境变量

环境变量名称	含义
PAAS_CSE_ENDPOINT	CSE 注册中心、配置中心等服务的地址信息。这个环境变量在微服务引擎专业版通过 APIG 访问的时候使用，上述服务的外部访问地址是统一的域名。 说明 不建议使用这个环境变量，而是使用具体服务的环境变量，避免在微服务引擎专享版的场景下存在歧义，需要修改应用程序。
PAAS_CSE_SC_ENDPOINT	CSE 注册中心地址信息。
PAAS_CSE_CC_ENDPOINT	CSE 配置中心地址信息。
PAAS_PROJECT_NAME	项目名称。
CAS_APPLICATION_NAME	ServiceStage 的应用名称。
CAS_COMPONENT_NAME	ServiceStage 的组件名称。
CAS_INSTANCE_VERSION	ServiceStage 的部署版本号。

您可以结合不同微服务开发框架的机制，比如 Spring Cloud 提供的 Place Holder 机制、Java Chassis 提供的“mapping.yaml”机制等来合理使用这些变量，减少部署需要手工输入的内容。

ServiceStage 创建应用过程中，可以绑定中间件（如 DCS、RDS）。应用绑定的中间件配置信息可以通过以下环境变量获取。

- 分布式会话

  基于 DCS 实现的稳定可靠的会话存储，支持主流 Web 容器的自动注入，如 tomcat context，node.js express-session，php 的 session handler 等。

  分布式会话相关环境变量说明如下表所示。

表1-2 DCS 分布式会话相关环境变量

环境变量	说明
DISTRIBUTED_SESSION_CLUSTER	实例是否是集群模式，取值 true/false

环境变量	说明
DISTRIBUTED_SESSION_TYPE	分布式会话实例的存储类型，当前只支持 Redis
DISTRIBUTED_SESSION_VERSION	分布式会话实例的版本号
DISTRIBUTED_SESSION_NAME	分布式会话实例的名称
DISTRIBUTED_SESSION_HOST	分布式会话实例的连接 IP 地址
DISTRIBUTED_SESSION_PORT	分布式会话实例的连接 IP 端口
DISTRIBUTED_SESSION_PASSWORD	分布式会话实例的连接密码

- 分布式缓存

  分布式缓存服务（Distributed Cache Service，简称 DCS）是一款内存数据库服务，兼容了 Redis 和 Memcached 两种内存数据库引擎，为您提供即开即用、安全可靠、弹性扩容、便捷管理的在线分布式缓存能力，满足用户高并发及数据快速访问的业务诉求。

  分布式缓存相关环境变量如下表所示。

表1-3 DCS 分布式缓存相关环境变量

环境变量	说明
DISTRIBUTED_CACHE_CLUSTER	实例是否是集群模式，取值 true/false
DISTRIBUTED_CACHE_TYPE	分布式缓存实例的存储类型，当前只支持 Redis
DISTRIBUTED_CACHE_VERSION	分布式缓存实例的版本号
DISTRIBUTED_CACHE_NAME	分布式缓存实例的名称
DISTRIBUTED_CACHE_HOST	分布式缓存实例的连接 IP 地址
DISTRIBUTED_CACHE_PORT	分布式缓存实例的连接 IP 端口
DISTRIBUTED_CACHE_PASSWORD	分布式缓存实例的连接密码

- 关系型数据库

  关系型数据库（Relational Database Service，简称 RDS）是一种基于云计算平台的即开即用、稳定可靠、弹性伸缩、便捷管理的在线关系型数据库服务。

  关系型数据库相关环境变量如下表所示。

表1-4 RDS 关系型数据库相关环境变量

环境变量	说明

环境变量	说明
RELATIONAL_DATABASE_NAME	关系型数据库实例名称
RELATIONAL_DATABASE_CONNECTION_TYPE	关系型数据库实例的连接类型，取值为JNDI/SPRING_CLOUD_CONNECTOR
RELATIONAL_DATABASE_JNDI_NAME	关系型数据库实例的 JNDI 名称，如果连接类型为 JNDI
RELATIONAL_DATABASE_DB_NAME	关系型数据库实例的数据库名
RELATIONAL_DATABASE_DB_USER	关系型数据库实例的数据库用户
RELATIONAL_DATABASE_DB_TYPE	关系型数据库实例的数据库类型，当前只支持 MySQL
RELATIONAL_DATABASE_VERSION	关系型数据库实例的数据库版本
RELATIONAL_DATABASE_HOST	关系型数据库实例的数据库 IP 地址
RELATIONAL_DATABASE_PORT	关系型数据库实例的数据库端口
RELATIONAL_DATABASE_PASSWORD	关系型数据库实例的数据库密码

1.4 对接微服务应用

1.4.1 Spring Cloud 接入 CSE

本章节介绍 Spring Cloud 如何接入 CSE，使得 Spring Cloud 能够对接 CSE，并且方便的使用 CSE 提供的最常用的功能。在使用微服务引擎功能章节，会给出具体的开发指导。

本章节介绍的开发方法，可以在 Spring Cloud Huawei Samples 项目中找到对应的代码，供您在开发过程中参考。

说明

Spring Cloud 接入 CSE 需要使用 Spring Cloud Huawei，本文主要描述如何在 Spring Cloud 中集成和使用 Spring Cloud Huawei。

前提条件

- 已基于 Spring Cloud 开发好了微服务应用。

  Spring Cloud 微服务框架下的微服务应用开发，请参考 https://spring.io/projects/spring-cloud。
- 版本要求：Spring Cloud Huawei 1.6.0-Hoxton 及以上版本。

- 本文假设您的项目使用了 maven 管理打包，您熟悉 maven 的依赖管理机制，能够正确的修改 pom.xml 文件中的 dependency management 和 dependency。

操作步骤

步骤 1 在项目的“pom.xml”文件中引入依赖。

- 如果使用 Spring Cloud 开发微服务，引入：

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-service-engine</artifactId>
</dependency>
```

- 如果使用 Spring Cloud Gateway 开发网关，引入：

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-service-engine-
gateway</artifactId>
</dependency>
```

推荐使用 Maven Dependency Management 管理项目依赖的三方软件，在项目中引入：

```
 <dependencyManagement>
    <dependencies>
      <!-- configure user spring cloud / spring boot versions -->
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>${spring-boot.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <!-- configure spring cloud huawei version -->
      <dependency>
        <groupId>com.huaweicloud</groupId>
        <artifactId>spring-cloud-huawei-bom</artifactId>
        <version>${spring-cloud-huawei.version}</version>
        <type>pom</type>
```

```
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

如果您的项目中，已经包含了上述依赖，则不需要做任何处理。

如果您的项目中使用了其他注册发现库，比如 eureka，需要对项目进行适当调整，包括：

- 删除项目中 eureka 相关依赖，比如：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

- 如果代码中使用了@EnableEurekaServer， 需要删除并替换为@EnableDiscoveryClient。

说明

组件 spring-cloud-starter-huawei-service-engine 包含了服务注册、配置中心、服务治理、灰度发布、契约管理等功能。其中契约管理对于 Spring Cloud 微服务应用的运行不是必须的。微服务引擎对契约个数存在数量限制，当微服务应用契约个数超过限制，会注册失败。如果遗留系统无法进行合理的拆分减少契约个数，可以排除依赖，不使用契约管理功能。

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-service-engine</artifactId>
  <exclusions>
    <exclusion>
      <groupId>com.huaweicloud</groupId>
      <artifactId>spring-cloud-starter-huawei-swagger</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

步骤 2 配置微服务信息。

在“bootstrap.yml”增加微服务描述信息。如果项目中没有“bootstrap.yml”，则创建一个新的文件。

```
spring:
  application:
    name: basic-provider
  cloud:
    servicecomb:
      discovery:
        enabled: true
        address: http://127.0.0.1:30100
```

```
      appName: basic-application
      serviceName: ${spring.application.name}
      version: 0.0.1
      healthCheckInterval: 30
    config:
      serverAddr:  http://127.0.0.1:30113
      serverType: config-center
```

步骤 3 （可选）配置 AK/SK。

如果使用微服务引擎专业版，需要配置 AK/SK；如果使用微服务引擎专享版，不需要配置 AK/SK，可以跳过这个步骤。

说明

新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。

AK/SK 在“bootstrap.yml”中配置，默认提供明文配置，支持用户自定义加密存储。

- 明文方法，在“bootstrap.yml”文件中增加配置。

```
spring:
  cloud:
    servicecomb:
      credentials:
        enabled: true
        accessKey: AK
        secretKey: SK
        akskCustomCipher: default
        project: 项目名称
```

- 自定义实现
  a. 首先实现接口“com.huaweicloud.common.util.Cipher”，里面有两个方法：
     - String name()，这个是 spring.cloud.servicecomb.credentials.akskCustomCipher 的名称定义，需要配置在配置文件中。
     - char[] decode(char[] encrypted)，解密接口，对 secretKey 进行解密后使用。

```
public class CustomCipher implements Cipher
```

加密解密的实现需要作为 BootstrapConfiguration，首先声明：

```
@Configuration
public class MyAkSKCipherConfiguration {
  @Bean
  public Cipher customCipher() {
    return new CustomCipher();
  }
}
```

然后增加文件 “META-INF/spring.factories”定义配置：

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
com.huaweicloud.common.transport.MyAkSKCipherConfiguration
```

b. 自定义完成，即可在“bootstrap.yaml”文件中使用新增加的解密算法：

```
spring:
  cloud:
    servicecomb:
      credentials:
        enabled: true
        accessKey: AK
        secretKey: SK
        akskCustomCipher: 自定义算法名称
        project: 项目名称
```

步骤 4 （可选）配置安全认证参数。

使用微服务引擎专享版，并且启用了安全认证，需要配置，其他场景可以跳过这个步骤。

微服务引擎开启了安全认证之后，所有调用的 API 都需要先获取 token 才能调用，认证流程请参考服务中心 RBAC 说明。

使用安全认证首先需要从微服务引擎获取用户名和密码，然后在配置文件中增加如下配置。和配置 AK/SK 一样，password 默认明文存储，开发者可以自定义 password 的加密存储算法，这里不重复描述。

- 明文方法

```
spring:
  cloud:
    servicecomb:
      credentials:
        account:
          name:用户名
          password:密码
          cipher: default
```

- 自定义实现加密存储方法

首先实现接口com.huaweicloud.common.util.Cipher，里面有两个方法：

String name()，这个是spring.cloud.servicecomb.credentials.cipher的名称定义，需要配置在配置文件中。

char[] decode(char[] encrypted)，解密接口，对secretKey进行解密后使用。

```
public class CustomCipher implements Cipher
```

加密解密的实现需要作为BootstrapConfiguration，首先声明：

```
@Configuration
public class MyCipherConfiguration {
  @Bean
  public Cipher customCipher() {
```

```
    return new CustomCipher();
  }
}
```

然后增加文件 META-INF/spring.factories定义配置：

```
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
com.huaweicloud.common.transport.MyCipherConfiguration
```

自定义完成，即可在bootstrap.yaml文件中使用新增加的解密算法：

```
spring:
  cloud:
    servicecomb:
      credentials:
        account:
          name:用户名
          password:密码
          cipher: 自定义算法名称
```

说明

使用安全认证功能，需要 1.6.0-Hoxton 及以上版本。

使用微服务引擎专业版，不能使用 watch 功能，需要在配置文件里面关闭，否则会周期性打印错误日志。服务中心设置 watch=false。1.6.0-Hoxton 及以上版本默认没有开启 watch 功能，不需要设置。

```
spring:
  cloud:
    servicecomb:
      discovery:
        watch: false
```

----结束

1.4.2 Java Chassis 接入 CSE

本章节介绍 Java Chassis 如何接入 CSE，使得 Java Chassis 能够对接 CSE，并且方便的使用 CSE 提供的最常用的功能。在使用微服务引擎功能章节，会给出具体的开发指导。

本章节介绍的开发方法，可以在 Apache ServiceComb Samples 项目中找到对应的代码，供您在开发过程中参考。

前提条件

- 已基于 Java Chassis 开发好了微服务应用。

  Java Chass 框架下的微服务应用开发，请参考 https://servicecomb.apache.org/references/java-chassis/zh_CN/。
- 版本要求：Java Chassis 2.3.0 及以上版本。

- 本文假设您的项目使用了 maven 管理打包，您熟悉 maven 的依赖管理机制，能够正确的修改“pom.xml”文件中的 dependency management 和 dependency。
- Java Chassis 支持和不同的技术进行组合使用，配置文件的名称和实际使用的技术有关。如果您采用 Spring 方式使用 Java Chassis，配置文件的名称一般为“microservice.yaml”，如果您采用 Spring Boot 方式使用 Java Chassis， 配置文件名称一般为“application.yaml”。本文统一使用“microservice.yaml”表示配置文件，请结合实际项目进行区分。

操作步骤

步骤 1 在项目的“pom.xml”文件中引入依赖。

```
<dependency>
   <groupId>org.apache.servicecomb</groupId>
   <artifactId>solution-basic</artifactId>
</dependency>
<dependency>
   <groupId>org.apache.servicecomb</groupId>
   <artifactId>servicestage-environment</artifactId>
</dependency>
```

推荐使用 Maven Dependency Management 管理项目依赖的三方软件，在项目的“pom.xml”文件中引入：

```
<dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>org.apache.servicecomb</groupId>
       <artifactId>java-chassis-dependencies</artifactId>
       <version>${java-chassis.version}</version>
       <type>pom</type>
       <scope>import</scope>
     </dependency>
   </dependencies>
 </dependencyManagement>
```

如果您的项目中，已经包含了上述依赖，则不需要做任何处理。

其中 servicestage-environment 软件包是可选的。这个软件包提供了环境变量映射的功能，依赖这个软件包以后，当您采用 ServiceStage 部署应用，不用手工修改注册中心地址、配置中心地址、项目名称等信息，会通过环境变量覆盖“microservice.yaml”中的默认配置，它包含“mapping.yaml”文件，在您自己的项目中增加“mapping.yaml”文件能够起到同样的效果。

说明

“mapping.yaml”在后续新版本可能会发生变化，以适配 CSE 最新的功能要求。如果期望后续升级新版本保持稳定而不是跟随 CSE 演进，您可以选择不依赖 servicestage-environment，而是在您自己的项目中增加“mapping.yaml”。

```
PAAS_CSE_ENDPOINT:
  - servicecomb.service.registry.address
  - servicecomb.config.client.serverUri
PAAS_CSE_SC_ENDPOINT:
  - servicecomb.service.registry.address
PAAS_CSE_CC_ENDPOINT:
  - servicecomb.config.client.serverUri
PAAS_PROJECT_NAME:
  - servicecomb.credentials.project

# CAS_APPLICATION_NAME:
#  - servicecomb.service.application
# CAS_COMPONENT_NAME:
#  - servicecomb.service.name
# CAS_INSTANCE_VERSION:
#  - servicecomb.service.version
```

solution-basic 里面引入了常用的软件包，并且提供了默认的“microservice.yaml”文件。这个配置文件配置了常用的 Handler 和参数。其内容如下：

```
# order of this configure file
servicecomb-config-order: -100

servicecomb:

# handlers
  handler:
    chain:
      Provider:
        default: qps-flowcontrol-provider
      Consumer:
        default: qps-flowcontrol-consumer,loadbalance,fault-injection-
consumer

# loadbalance strategies
  references:
    version-rule: 0
  loadbalance:
    retryEnabled: true
    retryOnNext: 1
    retryOnSame: 0

# metrics and access log
  accesslog:
    enabled: true
```

```
metrics:
  window_time: 60000
  invocation:
    latencyDistribution: 0,1,10,100,1000
  Consumer.invocation.slow:
    enabled: true
    msTime: 1000
  Provider.invocation.slow:
    enabled: true
    msTime: 1000
  publisher.defaultLog:
    enabled: true
    endpoints.client.detail.enabled: true
```

“microservice.yaml”配置文件设置了 servicecomb-config-order: -100 ， 表示配置文件的优先级很低（order 越大，优先级越高，缺省为 0），如果业务服务增加了同样的配置项，会覆盖这里的配置。

说明

“microservice.yaml”文件在后续新版本可能会发生变化，以适配 CSE 最新的功能要求。如果期望后续升级新版本保持稳定而不是跟随 CSE 演进，您可以考虑将配置项写到您自己的“microservice.yaml”文件中。

步骤 2 （可选）配置 AK/SK。

如果使用微服务引擎专业版，需要配置 AK/SK；如果使用微服务引擎专享版，不需要配置 AK/SK，可以跳过这个步骤。

说明

新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。

AK/SK 在“microservice.yaml”中配置，ServiceComb 默认提供明文配置，支持用户自定义加密存储方案。

- 明文方法，在“microservice.yaml”文件中增加配置。

```
servicecomb:
  credentials:
    accessKey: AK
    secretKey: SK
    project: 项目名称
    akskCustomCipher: default
```

- 自定义实现，首先实现接口“org.apache.servicecomb.foundation.auth.Cipher”，里面有两个方法：
  - String name()

    这个是 servicecomb.credentials.akskCustomCipher 的名称定义，需要配置在配置文件中。

- char[] decode(char[] encrypted)
  解密接口，对 secretKey 进行解密后使用。

实现类需要声明为 SPI，比如：

```
package com.example
public class MyCipher implements Cipher
```

创建 SPI 配置文件，文件名称和路径为 META-INF/serivces/org.apache.servicecomb.foundation.auth.Cipher， 文件内容为：

```
com.example.MyCipher
```

然后在“microservice.yaml”文件中增加配置。

```
servicecomb:
  credentials:
    accessKey: AK
    secretKey: SK  #对应的加密后的SK
    project:  项目名称
    akskCustomCipher: youciphername  #实现类里面的name()方法返回的名称
```

说明

如果不想将 AK/SK 写入配置文件，也可以使用如下两种方法，具体操作请参考 Java Chassis。

- 使用环境变量。操作系统环境变量名称不支持“.”，Java Chassis 能够自动处理 servicecomb_credentials_accessKey 环境变量，将其映射到 servicecomb.credentials.accessKey。
- 增加“mapping.yaml”文件，自定义环境变量名称。在步骤 1 介绍模块 servicestage-environment 的时候，已经用到了这个方法。

步骤 3 （可选）配置安全认证参数。

使用微服务引擎专享版，并且启用了安全认证，需要配置，其他场景可以跳过这个步骤。

微服务引擎开启了安全认证之后，所有调用的 API 都需要先获取 token，才能调用，认证流程请参考服务中心 RBAC 说明。

使用安全认证首先需要从微服务引擎获取用户名和密码，然后在配置文件中增加如下配置：

```
servicecomb:
  credentials:
    rbac.enabled: true
    account:
      name: your account name # 从微服务引擎获取的用户名
      password: your password # 从微服务引擎获取的密码
      cipher: default #接口org.apache.servicecomb.foundation.auth.Cipher的实现类里面的name()方法返回的名称
```

其中“cipher”指定了对“password”进行加密的算法名称，默认提供明文存储。

和 AK/SK 认证的加密方案类似，通过实现接口“org.apache.servicecomb.foundation.auth.Cipher”可以对密码进行加密存储。

说明

- 明文存储无法保证安全， 建议您对密码进行加密存储。
- 和配置 AK/SK 一样，也可以使用环境变量配置用户名和密码信息，请参考 Java Chassis。
- 使用微服务引擎专业版，不能使用 watch 功能，需要在配置文件里面关闭，否则会周期性打印错误日志。服务中心设置 watch=false，配置中心设置 refreshMode=1。

```
servicecomb:
  service:
    application: porter-application
    name: user-service
    version: 0.0.1
    registry:
      address: http://localhost:30100
      instance:
        watch: false
  config:
    client:
      serverUri: http://localhost:30113
      refreshMode: 1
```

----结束

1.4.3 Go Chassis 接入 CSE

本章节介绍 Go Chassis 如何接入 CSE，使得 Go Chassis 能够对接 CSE，并且方便的使用 CSE 提供的最常用的功能。在使用微服务引擎功能章节，会给出具体的开发指导。

本章节介绍的开发方法，可以在 Go Chassis Sample Discovery 项目中找到对应的代码，供您在开发过程中参考。

前提条件

- 已基于 Go Chassis 开发好了微服务应用。

  Go Chassis 微服务框架下的微服务应用开发，请参考 https://go-chassis.readthedocs.io/en/latest/。
- 相关软件版本要求：
  - Golang 1.14 及以上版本
  - Go Chassis 2.2.0 及以上版本
  - github.com/go-chassis/go-chassis-cloud v2.1.1 及以上版本

操作步骤

步骤 1 在项目的“go.mod”文件中引入依赖。

请在您的项目的“go.mod”中添加以下依赖。如果您的项目中已经包含以下依赖，则不需要做任何处理。

```
github.com/go-chassis/go-chassis-cloud v2.1.1
```

步骤 2 设置注册中心和配置中心。

在“chassis.yaml”文件中增加配置项：

```
servicecomb:
  registry:
    type: servicecenter
    address: 注册中心uri
  config:
    client:
      type: config-center
      serverUri: 配置中心uri
```

步骤 3 （可选）配置 AK/SK。

如果使用微服务引擎专业版，需要配置 AK/SK；如果使用微服务引擎专享版，不需要配置 AK/SK，可以跳过这个步骤。

说明

新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。

AK/SK 在“chassis.yaml”（或“auth.yaml”）中配置，Go Chassis 默认提供明文配置，支持用户自定义加密存储方案。

- 明文方法

  增加如下配置：

```
cse:
  credentials:
    accessKey: AK
    secretKey: SK
    project: 项目名称
    akskCustomCipher: default
```

- 自定义实现

  Go Chassis 读取“secretKey”后将使用 akskCustomCipher 插件进行解密，因此用户可使用 akskCustomCipher 插件对“secretKey”明文进行加密后再配置。

  增加如下配置：

```
cse:
  credentials:
    accessKey: AK
    secretKey: SK #对应的加密后的SK
    project:  项目名称
    akskCustomCipher: yourCipher
```

akskCustomCipher 插件的开发方法请参考：https://go-chassis.readthedocs.io/en/latest/dev-guides/how-to-write-cipher.html?highlight=cipher#protect-your-sk。

----结束

1.4.4 Dubbo 接入 CSE

本章节介绍 Dubbo 如何接入 CSE，使得 Dubbo 能够对接 CSE，并且方便的使用 CSE 提供的最常用的功能。在使用微服务引擎功能章节，会给出具体的开发指导。

本章节介绍的开发方法，可以在 Dubbo ServiceComb Samples 项目中找到对应的代码，供您在开发过程中参考。

Dubbo 未提供网关服务，在例子中使用了 Spring Cloud Gateway 作为网关，Spring Cloud Gateway 接入 CSE 的说明请参考 Spring Cloud 接入 CSE 章节。

前提条件

- 已基于 Dubbo 开发好了微服务应用。

  Dubbo 微服务框架下的微服务开发，请参考 https://dubbo.apache.org/zh/。
- Dubbo ServiceComb 版本为 1.3.6 及以上版本，并且使用 Spring Boot 作为开发底座。
- 本文假设您的项目使用了 maven 管理打包，您熟悉 maven 的依赖管理机制，能够正确的修改“pom.xml”文件中的 dependency management 和 dependency。

操作步骤

步骤 1 （可选）在项目的“pom.xml”文件中引入依赖。

如果您的项目中，已经包含了如下依赖，则不需要做任何处理。

如果您的项目中使用了其他注册发现库，比如 zookeeper、nacos 等，可以删除这些依赖，这个步骤不是必须的。

```
<dependency>
  <groupId>com.huaweicloud.dubbo-servicecomb</groupId>
  <artifactId>dubbo-servicecomb-solution-spring-boot</artifactId>
</dependency>
```

推荐使用 Maven Dependency Management 管理项目依赖的三方软件，在项目的“pom.xml”文件中引入：

```
<dependencyManagement>
  <dependencies>
   <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-dependencies</artifactId>
     <version>${spring-boot.version}</version>
     <type>pom</type>
     <scope>import</scope>
```

```
    </dependency>
    <dependency>
      <groupId>org.apache.dubbo</groupId>
      <artifactId>dubbo-bom</artifactId>
      <version>${dubbo.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>com.huaweicloud.dubbo-servicecomb</groupId>
      <artifactId>dubbo-servicecomb-bom</artifactId>
      <version>${dubbo-servicecomb.version}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
   </dependencies>
  </dependencyManagement>
```

步骤 2 在启动类引入 Dubbo ServiceComb 的 Bean。

下面的示例代码中，"classpath*:spring/dubbo-servicecomb.xml"是 Dubbo ServiceComb 的 Bean 信息，增加这个信息才会加载 Dubbo ServiceComb 的功能。

```
@SpringBootApplication
@ImportResource({"classpath*:spring/dubbo-provider.xml",
"classpath*:spring/dubbo-servicecomb.xml"})
public class ProviderApplication {
  public static void main(String[] args) throws Exception {
    try {
      new
SpringApplicationBuilder(ProviderApplication.class).web(WebApplicationTyp
e.NONE).run(args);
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }
}
```

步骤 3 配置微服务信息。

1. 在“dubbo.properties”增加微服务描述信息。如果项目中没有“dubbo.properties”，则创建一个新的文件。使用 Spring Boot 集成 Dubbo，也可以将配置信息写到“application.yaml”中。为了符合 Dubbo 的使用习惯，本文档使用 properties 格式进行描述。在推荐的例子中，使用的是“application.yaml”，使用 yaml 格式可以更好的利用 profile 机制管理不同环境的配置。

```
#### 服务配置信息 ####
# 所属应用。默认为 default
```

```
dubbo.servicecomb.service.application=basic-application
# 服务名称。默认为 defaultMicroserviceName
dubbo.servicecomb.service.name=price-provider
# 版本。默认为 1.0.0.0
dubbo.servicecomb.service.version=1.0.0
# 环境。默认为空。可选值：development,testing,acceptance,production
# dubbo.servicecomb.service.environment=production
# project。 默认为default
# dubbo.servicecomb.service.project=

#### 实例配置信息 ####
# 实例初始状态。可选值：UP,DOWN,STARTING,OUTOFSERVICE
# dubbo.servicecomb.instance.initialStatus=UP

#### 服务中心配置信息 ####
dubbo.servicecomb.registry.address=http://127.0.0.1:30100

#### 配置中心配置信息 ####
dubbo.servicecomb.config.address=http://127.0.0.1:30113
dubbo.servicecomb.config.fileSource=provider.yaml
#### SSL 配置信息  ####
# dubbo.servicecomb.ssl.enabled=true
dubbo.servicecomb.ssl.enabled=true
```

2. 修改 dubbo registry 为 CSE 的服务中心。这个配置项一般在 spring 的配置文件中，比如“spring/dubbo-provider.xml”文件中。

```
<dubbo:registry address="sc://127.0.0.1:30100"/>
```

这个配置项的地址信息不会使用，只使用了协议名称 sc。

3. （可选）如果使用基于流量的微服务治理的重试功能，需要修改 dubbo cluster 配置。如果无此配置项，请新增。详细请参考使用服务治理。

```
<dubbo:consumer cluster="dubbo-servicecomb"></dubbo:consumer>
```

步骤 4 （可选）配置 AK/SK。

如果使用微服务引擎专业版，需要配置 AK/SK；如果使用微服务引擎专享版，不需要配置 AK/SK，可以跳过这个步骤。

说明

新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。

AK/SK 在“dubbo.properties”中配置，默认提供明文配置，示例如下：

```
dubbo.servicecomb.credentials.enabled=true
dubbo.servicecomb.credentials.accessKey=AK
dubbo.servicecomb.credentials.secretKey=SK
```

```
dubbo.servicecomb.credentials.cipher=default
dubbo.servicecomb.credentials.project=项目名称
```

其中，项目名称的获取请参考获取 AK/SK 与项目名称。

步骤 5 （可选）使用环境变量。

可以通过环境变量简化部署配置。使用 ServiceStage 部署可以使用的环境变量参考准备环境。

“dubbo.properties”和“application.yaml”中均可以使用 place holder 来使用环境变量。以“application.yaml”为例，服务中心的地址和配置中心的地址使用环境变量，那么采用 ServiceStage 部署的时候，环境变量的实际值会覆盖配置文件里面的缺省值，减少了重新编译打包的步骤。

```
PAAS_CSE_SC_ENDPOINT: http://127.0.0.1:30100
PAAS_CSE_CC_ENDPOINT: http://127.0.0.1:30113


dubbo:
  servicecomb:
    registry:
      address: ${PAAS_CSE_SC_ENDPOINT}
    config:
      address: ${PAAS_CSE_CC_ENDPOINT}
```

----结束

1.4.5 Mesher 接入 CSE

1.4.5.1 Mesher 简介

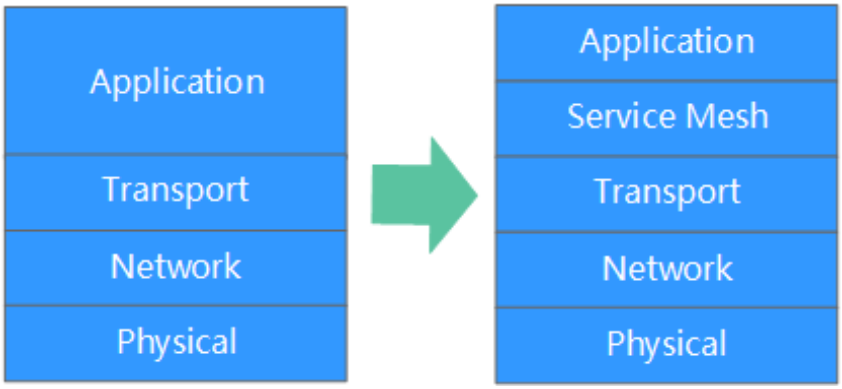
什么是 Mesher

Mesher 是 Service Mesh 的一个具体的实现，是一个轻量的代理服务以 Sidecar 的方式与微服务一起运行。

Service Mesh 是由 William Morgan 定义：

Service Mesh 是一个基础设施层，用于处理服务间通信。云原生应用有着复杂的服务拓扑，Service Mesh 保证请求可以在这些拓扑中可靠地传输。在实际应用当中，Service Mesh 通常是由一系列轻量级的网络代理组成的，它们与应用程序部署在一起，但应用程序不需要知道它们的存在。

随着云原生应用的崛起，Service Mesh 逐渐成为一个独立的基础设施层。在云原生模型里，一个应用可以由数百个服务组成，每个服务可能有数千个实例，而每个实例可能会持续地发生变化。服务间通信不仅异常复杂，而且也是运行时行为的基础。管理好服务间通信对于保证端到端的性能和可靠性来说是非常重要的。



Service Mesh 实际上就是处于 TCP/IP 之上的一个抽象层，假设底层的 L3/L4 网络能够点对点地传输字节（同时，也假设网络环境是不可靠的，所以 Service Mesh 必须具备处理网络故障的能力）。

从某种程度上说，Service Mesh 有点类似 TCP/IP。TCP 对网络端点间传输字节的机制进行了抽象，而 Service Mesh 则是对服务节点间请求的路由机制进行了抽象。Service Mesh 不关心消息体是什么，也不关心它们是如何编码的。应用程序的目标是“将某些东西从 A 传送到 B”，而 Service Mesh 所要做的就是实现这个目标，并处理传送过程中可能出现的任何故障。

与 TCP 不同的是，Service Mesh 有着更高的目标：为应用运行时提供统一的、应用层面的可见性和可控性。Service Mesh 将服务间通信从底层的基础设施中分离出来，让它成为整个生态系统的一等公民——它因此可以被监控、托管和控制。

为什么要使用 Mesher

- 业务代码无须改造
- 支持老旧应用接入
- 普通应用快速成为云原生应用
- 业务代码零侵入

基本实现原理

Mesher 是 L7 层协议代理，Mesher 以 Sidecar 模式运行在应用所在的 Pod 内，与 Pod 共享网络与存储：

1. Pod 中的应用使用 Mesher 作为 http 代理，可以自动发现其他服务。
2. Mesher 会代替 Pod 中的应用向注册中心注册应用相关信息，以便让其他应用发现。

发起一次网络请求的过程中存在微服务消费者和提供者，场景如下：

- 场景一：仅消费者使用 Mesher 作为 Sidecar。

  提供者需要自己实现服务注册发现，或者使用 Java 或 GO chassis 开发框架，否则无法发现 AccountService。

  应用间的网络请求如下：

  Store web -> Mesher -> Account service
- 场景二：消费者提供者均使用 Mesher 作为 Sidecar。

  此场景无需再使用微服务开发框架。

应用间的网络请求如下：

Store web -> Mesher -> Mesher -> Account service

- 场景三：仅提供者使用 Mesher 作为 Sidecar

  消费者需要使用 Java 或者 Go Chassis 开发框架。

  应用间的网络请求如下：

  Store web -> Mesher -> Account service

注意事项

应用上云后需要作出一定的配置变更。例如在 Mesher 所处环境外，StoreWeb 在访问 AccountService 时使用 http://IP:port/进行访问。在使用 Mesher 后，使用 http://AccountService:port/即可进行访问，文档中将详细讲解。

1.4.5.2 接入说明

须知

不同于微服务开发框架，Mesher 的能力是由 ServiceStage 平台提供的。您必须在 ServiceStage 平台创建 Mesher 组件，才能使用 Mesher。

本章节介绍 http 应用如何通过 Mesher 接入 CSE。由于 Mesher 支持多语言，因此本章仅描述通过 Mesher 接入 CSE 时的规范要求。具体的代码样例可以参考：

- .Net core 接入服务网格
- PHP 接入服务网格

前提条件

已开发好了一个 http 应用（支持多语言）。

操作步骤

步骤 1 修改微服务调用的 URL，将 URL 中的${IP:Port}修改为服务名。

例如调用一个名为“provider”的微服务，API 为“/hello”，则调用 URL 通常为：http://${IP:Port}/hello。例如：

```
http://127.0.0.1:80/hello
```

您需要将调用的 URL 修改为：

```
http://provider/hello
```

步骤 2 为微服务调用请求设置 http proxy，proxy 的地址为：http://${IP:Port}。例如：

```
http://127.0.0.1:30101
```

若一个微服务被其他微服务调用，则该微服务的监听地址中必须包含 127.0.0.1（监听端口和其他监听地址不做限制）。

步骤 3 在 ServiceStage 平台创建 Mesher 组件，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 新建应用组件 > 创建微服务组件”章节。

须知

Mesher 组件名称必须和微服务名称一致。

----结束

1.5 部署微服务应用

微服务应用部署，请参考“帮助中心 > 应用管理与运维平台 > 用户指南> 应用管理> 应用组件部署 > 部署组件”。

1.6 使用微服务引擎功能

1.6.1 使用服务注册

微服务引擎的服务中心提供了服务注册的功能。服务注册是指微服务启动的时候，将基本信息，比如所属应用、微服务名称、微服务版本、监听的地址信息等注册到服务中心。

微服务运行的过程中，也通过服务中心查询其他微服务的基本信息。不同的微服务开发框架注册的信息会有差异，比如 Java Chassis 还会注册服务契约等信息。不同微服务开发框架注册的基本信息、注册和发现其他微服务的流程是相同的。

本章节重点介绍不同的微服务开发框架如何使用服务中心和配置自己的注册信息，同时也会介绍微服务和注册中心之间交互有关的配置项。微服务注册成功后，可以在微服务引擎使用微服务目录、微服务实例列表、微服务依赖关系等功能。

Java Chassis

Java Chassis 使用服务注册，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>registry-service-center</artifactId>
</dependency>
```

如果项目已经直接或者间接包含这个依赖，则无需添加。Java Chassis 包含如表 1-5 所示配置项。这些配置项的值影响在服务中心注册的基本信息，以及微服务和服务中心之间的交互，比如心跳等。

表1-5 Java Chassis 常用配置项

配置项	含义	缺省值	备注
servicecomb.service.application	所属应用	default	-
servicecomb.service.name	微服务名称	defaultMicroservice	-
servicecomb.service.version	微服务版本	1.0.0.0	-
servicecomb.service.environment	环境	-	production、development等
servicecomb.service.registry.address	注册中心地址	http://127.0.0.1:30100	集群地址使用“,”分隔
servicecomb.service.registry.instance.watch	是否开启 watch 模式	true	微服务引擎专业版需要设置为 false 说明 新安装的环境下，ServiceStage不再提供对微服务引擎专业版的支持。
servicecomb.service.registry.instance.healthCheck.interval	发送心跳的时间间隔（秒）	30	-
servicecomb.service.registry.instance.healthCheck.times	允许的心跳失败次数。当连续第 times+1 次心跳仍然失败时则实例被服务中心下线。即 interval * (times + 1)决定了实例被自动注销的时间。如果服务中心等待这么长的时间没有收取到心跳，会注销实例	3	-
servicecomb.datacenter.name	数据中心名称	-	-
servicecomb.datacenter.region	数据中心区域	-	-
servicecomb.datacenter.availableZone	数据中心可用区	-	-

Java Chassis 注册的实例地址信息、监听地址，和配置项 servicecomb.service.publishAddress 指定的发布地址有关。服务监听地址的配置项是 servicecomb.rest.address 和 servicecomb.highway.address，分别对应 rest 传输方式和 highway 传输方式的监听地址。注册的地址信息和监听地址、发布地址的关系如表 1-6 所示。

表1-6 注册的实例地址生效规则

监听地址	发布地址	注册的实例地址
127.0.0.1	-	127.0.0.1
0.0.0.0	-	指定选取一张网卡的 IP 地址作为发布地址。不会选择通配符地址、回环地址或广播地址
具体 IP 地址	-	与监听地址一致
*	具体 IP 地址	与发布地址一致
*	"{网卡名}"	指定网卡名对应的 IP，注意需要加上引号和括号

Spring Cloud

Spring Cloud 使用服务注册，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-servicecomb-
discovery</artifactId>
</dependency>
```

如果项目已经直接或者间接包含这个依赖，则无需添加。Spring Cloud 包含如表 1-7 所示配置项，这些配置项的值影响在服务中心注册的基本信息，以及微服务和服务中心之间的交互，比如心跳等。和服务注册有关的信息，需要在“bootstrap.yml”配置。

表1-7 Spring Cloud 常用配置项

配置项	含义	缺省值	备注
spring.cloud.servicecomb.discovery.appName	所属应用	default	-
spring.cloud.servicecomb.discovery.serviceName	微服务名称	-	如果没有，使用 spring.application.name
spring.cloud.servicecomb.discovery.version	微服务版本	-	-

配置项	含义	缺省值	备注
server.env	环境	-	production，development 等
spring.cloud.servicecomb.discovery.enabled	是否启用服务注册发现	true	-
spring.cloud.servicecomb.discovery.address	注册中心地址	-	集群地址使用“,”分隔
spring.cloud.servicecomb.discovery.watch	是否开启 watch 模式	true	微服务引擎专业版需要设置为 false 说明 新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。
spring.cloud.servicecomb.discovery.healthCheckInterval	发送心跳的时间间隔（秒）	10	-
spring.cloud.servicecomb.discovery.datacenter.name	数据中心名称	-	-
spring.cloud.servicecomb.discovery.datacenter.region	数据中心区域	-	-
spring.cloud.servicecomb.discovery.datacenter.availableZone	数据中心可用区	-	-
spring.cloud.servicecomb.discovery.allowCrossApp	是否支持跨应用调用	false	服务端配置，表示允许不同应用下的客户端发现自己

Dubbo

Dubbo 使用服务注册，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>com.huaweicloud.dubbo-servicecomb</groupId>
  <artifactId>dubbo-servicecomb-service-center</artifactId>
</dependency>
```

如果项目已经直接或者间接包含这个依赖，则无需添加。Dubbo ServiceComb 包含如表1-8 所示配置项，这些配置项的值影响在服务中心注册的基本信息，以及微服务和服务中心之间的交互，比如心跳等。和服务注册有关的信息，需要在“dubbo.properties”配置。

表1-8 Dubbo 常用配置项

配置项	含义	缺省值	备注
dubbo.servicecomb.service.application	所属应用	default	-
dubbo.servicecomb.service.name	微服务名称	defaultMicroserviceName	-
dubbo.servicecomb.service.version	微服务版本	1.0.0.0	-
dubbo.servicecomb.service.environment	环境	-	production，development 等
dubbo.servicecomb.registry.address	注册中心地址	-	集群地址使用“,”分隔
dubbo.servicecomb.ssl.enabled	是否启用 SSL	false	如果服务中心是 SSL，需要设置为 true

1.6.2 使用配置中心

1.6.2.1 配置中心概述

配置中心用来管理微服务应用的配置。微服务连接配置中心，能够从配置中心获取配置信息及其变化。配置中心还是其他微服务管控功能的核心部件，比如服务治理规则的下发，也是通过配置中心实现的。

微服务引擎支持的配置中心有：config-center。

本章节介绍不同微服务开发框架使用配置中心的一些开发细节，包括如何配置依赖、连接配置中心有关的配置项等，并简单的介绍微服务应用中如何读取配置和响应配置变化。

微服务引擎

微服务引擎使用 config-center 作为配置中心。微服务默认会读取配置中心全局配置、服务配置。全局配置指环境和微服务相同的配置；服务配置指环境、应用、微服务名称和微服务相同的配置。

微服务引擎只支持 key-value 的配置项。如果用户需要使用 yaml 格式的配置文件，可以使用具体 SDK 提供的 fileSource 功能。通过在配置文件中指定 fileSource 的 key 列表，SDK 会将这些 key 对应的 value 全部当成 yaml 解析。以 Spring Cloud 为例，在 bootstrap.yml 中增加配置项：

```
spring:
  cloud:
    servicecomb:
      config:
```

```
      fileSource: file1.yaml,file2.yaml
```

并且在配置中心创建配置：

```
file1.yaml: |
  cse.example.key1: value1
  cse.example.key2: value2
file2.yaml: |
  cse.example.key3: value3
  cse.example.key4: value4
```

应用程序中会发现 4 个配置项：cse.example.key1=value1，cse.example.key2=value2，cse.example.key3=value3 和 cse.example.key4=value4。

1.6.2.2 Java Chassis 使用配置中心

Java Chassis 使用以 config-center 命名的配置中心，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>config-cc</artifactId>
</dependency>
```

如果项目已经直接或者间接包含如上依赖，则无需添加。Java Chassis 包含如表 1-9 所示配置项，这些配置项的值指定了微服务在配置中心的身份，以及微服务和配置中心之间的交互。

表1-9 Java Chassis 常用配置项

配置项	含义	缺省值	备注
servicecomb.service.application	所属应用	default	-
servicecomb.service.name	微服务名称	defaultMicroservice	-
servicecomb.service.version	微服务版本	1.0.0.0	-
servicecomb.service.environment	环境	-	production，development 等
servicecomb.config.client.serverUri	访问地址，格式为 http(s)://{ip}:{port}，以“,”分隔多个地址	http://127.0.0.1:30103	config-center
servicecomb.config.client.refreshMode	配置更新模式，0 为主动 push，1 为微服务周期 pull	0	config-center

配置项	含义	缺省值	备注
servicecomb.config.client.refreshPort	主动 push 端口	30104	config-center
servicecomb.config.client.tenantName	应用的租户名称	default	config-center

Java Chassis 有多种方式可以读取动态配置，第一种是使用 archaius API，例子如下：

```
DynamicDoubleProperty myprop = DynamicPropertyFactory.getInstance()
  .getDoubleProperty("trace.handler.sampler.percent", 0.1);
```

archaius API 支持 callback 处理配置变更：

```
myprop.addCallback(new Runnable() {
     public void run() {
         // 当配置项的值变化时，该回调方法会被调用
         System.out.println("trace.handler.sampler.percent is changed!");
     }
  });
```

第二种方式是使用 Java Chassis 提供的配置注入机制，使用这种方式能够非常简单的处理复杂配置，和配置优先级，例子如下：

```
  @InjectProperties(prefix = "jaxrstest.jaxrsclient")
  public class Configuration {
    /*
     * 方法的 prefix 属性值 "override" 会覆盖标注在类定义的 @InjectProperties
     * 注解的 prefix 属性值。
     *
     * keys属性可以为一个字符串数组，下标越小优先级越高。
     *
     * 这里会按照如下顺序的属性名称查找配置属性，直到找到已被配置的配置属性，则停止查找：
     * 1) jaxrstest.jaxrsclient.override.high
     * 2) jaxrstest.jaxrsclient.override.low
     *
     * 测试用例：
     * jaxrstest.jaxrsclient.override.high: hello high
     * jaxrstest.jaxrsclient.override.low: hello low
     * 预期：
     * hello high
     */
    @InjectProperty(prefix = "jaxrstest.jaxrsclient.override", keys =
```

```
{"high", "low"})
   public String strValue;
```

执行注入：

```
 ConfigWithAnnotation config =
SCBEngine.getInstance().getPriorityPropertyManager()
     .createConfigObject(Configuration.class,
          "key", "k");
```

第三中方式在和 Spring、Spring Boot 集成的时候使用，可以按照 Spring、Spring Boot 的原生方式读取配置，比如@Value、@ConfigurationProperties。Java Chassis 将配置层次应用于 Spring Environment 中，Spring 和 Spring Boot 读取配置的方式，也能够读取到 microservice.yaml 和动态配置的值。

有关 Java Chassis 读取配置的更多内容，请参考社区开发指南。

1.6.2.3 Spring Cloud 使用配置中心

Spring Cloud 使用配置中心，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-config</artifactId>
</dependency>
```

如果项目已经直接或者间接包含这个依赖，则无需添加。Spring Cloud 包含如表 1-10 所示配置项，这些配置项的值指定了微服务在配置中心的身份，以及微服务和配置中心之间的交互。

表1-10 Spring Cloud 常用配置项

配置项	含义	缺省值	备注
spring.cloud.servicecomb.discovery.appName	所属应用	default	-
spring.cloud.servicecomb.discovery.serviceName	微服务名称	-	如果没有，使用 spring.application.name
spring.cloud.servicecomb.discovery.version	微服务版本	-	-
server.env	环境	-	production，development 等
spring.cloud.servicecomb.config.enabled	是否启用动态配置	true	-
spring.cloud.servicecomb.con	配置中心类型	config-	配置中心类型，可

配置项	含义	缺省值	备注
fig.serverType		center	选值: config-center
spring.cloud.servicecomb.config.serverAddr	访问地址，格式为http(s)://{ip}:{port}，以“,”分隔多个地址	-	-
spring.cloud.servicecomb.config.fileSource	内容为 yaml 的配置项列表，使用“,”分割	-	主要用途是 config-center 支持 yaml 文件，配置项的 yaml 内容被解析，然后将 yaml 的配置项值设置到配置系统里面。
spring.cloud.servicecomb.config.watch.delay	pull 模式轮询时间间隔（毫秒）	10000	config-center

config-center 是一个配置管理器，支持使用 key-value 的格式添加配置项。对于 Spring Cloud 用户，经常需要在配置中心增加 yaml 格式的配置文件。Spring Cloud Huawei 提供了配置项 spring.cloud.servicecomb.config.fileSource，使得用户在使用 config-center 的情况下，也能够配置 yaml 格式的配置文件。这个配置项的值是 key-value 系统的 key 列表，多个 key 以逗号分隔，这些 key 的值是 yaml 格式的文本内容，Spring Cloud Huawei 会对这些 key 的值进行特殊处理和解析。

接入配置中心以后，Spring Cloud 应用可以采用@Value、@ConfigurationProperties 等标签的方式注入配置，也可以使用 Environment 读取配置，并且使用@RefreshScope 来支持配置的动态更新，详细内容请参考社区开发指南。

1.6.2.4 Dubbo 使用配置中心

Dubbo 当前版本只支持 config-center。Dubbo 使用配置中心，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>com.huaweicloud.dubbo-servicecomb</groupId>
  <artifactId>dubbo-servicecomb-config-center</artifactId>
</dependency>
```

如果项目已经直接或者间接包含这个依赖，则无需添加。Dubbo ServiceComb 包含如表 1-11 所示配置项，和配置中心有关的信息，需要在“dubbo.properties”配置。

表1-11 Dubbo 常用配置项

配置项	含义	缺省值	备注
dubbo.servicecomb.service.application	所属应用	default	-

配置项	含义	缺省值	备注
dubbo.servicecomb.service.name	微服务名称	defaultMicroserviceName	-
dubbo.servicecomb.service.version	微服务版本	1.0.0.0	-
dubbo.servicecomb.service.environment	环境	-	production，development 等
dubbo.servicecomb.config.address	配置中心地址	-	集群地址使用“,”分割
dubbo.servicecomb.config.type	配置中心类型	config-center	可选值：config-center
dubbo.servicecomb.config.fileSource	内容为 yaml 的配置项列表，使用“,”分割	-	主要用途是 config-center 支持 yaml 文件，配置项的 yaml 内容被解析，然后将 yaml 的配置项值设置到配置系统里面。
dubbo.servicecomb.ssl.enabled	是否启用 SSL	false	如果服务中心是 SSL，需要设置为 true

接入配置中心以后，可以采用 Spring、Spring Boot 原生的方式读取配置，比如@Value、@ConfigurationProperties 等标签的方式注入配置，也可以使用 Environment 读取配置，其中使用 Environment 读取配置能够获取到变化的配置值。详细内容请参考社区开发指南。

1.6.3 使用服务治理

1.6.3.1 服务治理概述

服务治理是一个非常宽泛的概念，一般指独立于业务逻辑之外，给系统提供一些可靠运行的系统保障措施。针对微服务场景下的常用故障模式，提供的保障措施包括：

- 负载均衡管理：提供多实例情况下的负载均衡策略管理，比如采用轮询的方式保障流量在不同实例均衡。当一个实例发生故障的时候，能够暂时隔离这个实例，防止访问这个实例造成请求超时等。
- 流控：流控的主要目的是提供负载保护，防止外部流量超过系统处理能力，导致系统崩溃。流控还被用于平滑请求，让请求以均匀分布的方式到达服务，防止突发的流量对系统造成冲击。
- 重试：重试的主要目的是保障随机失败对业务造成影响。随机失败在微服务系统经常发生，产生随机失败的原因非常多。以 Java 微服务应用为例，造成请求超时这种随机失败的原因包括：网络波动和软硬件升级，可能造成随机的几秒中断；JVM 垃圾回收、线程调度导致的时延增加；流量并不是均匀的，同时到来的 1000

个请求和 1 秒内平均来的 1000 个请求对系统的冲击是不同的，前者更容易导致超时；应用程序、系统、网络的综合影响，一个应用程序突然的大流量可能会对带宽产生影响，从而影响到其他应用程序运行；其他应用程序相关的场景，比如 SSL 需要获取操作系统熵，如果熵值过低，会有几秒钟的延迟。 系统不可避免地面临随机故障，必须具备一定的随机故障保护能力。

- 隔离仓：隔离仓通常针对系统资源占用比较多的业务进行保护。比如一个业务非常耗时，如果这个业务和其他业务共享线程池，当这个业务被大量突发访问时，其他业务都会等待，造成整个系统性能下降。隔离仓通过给资源占用比较多的业务分配独立的资源池（一般通过信号量或者线程池实现），避免对其他业务造成影响。

服务治理的复杂性在于没有任何治理措施是适用于所有场景的。对于一个应用场景工作良好的治理手段，在另外一个场景可能成为问题。在业务运行周期，根据业务运行状态和指标，动态的更新治理策略非常重要。

在业务系统中使用服务治理，通常包括下面几个步骤：

1. 开发业务。这个过程一般比较少关注服务治理的内容，以交付业务功能为重心。微服务开发框架针对常用的系统故障，一般都默认提供了保障措施，选择合适的微服务开发框架，可以节省 DFx 的时间。
2. 性能测试和故障演练。这个过程中会发现非常多的系统不稳定问题，服务治理的策略会在解决这些问题的过程中应用，并写入配置文件作为应用程序缺省值。
3. 业务上线运行。上线运行的过程中碰到未考虑的场景，需要采用配置中心动态调整治理参数，以保障业务平稳运行。

上面的 3 个步骤在整个软件生命周期会不断迭代完善。描述如何使用所有的治理能力是复杂的，微服务引擎针对不同的微服务开发框架，提供了一个统一的基于流量特征的服务治理能力。本章节重点介绍如何使用基于流量特征的服务治理能力。

1.6.3.2 基于动态配置的流量特征治理介绍

基于动态配置的流量特征治理旨在提供一种通用的，适合不同语言、不同微服务开发框架的治理规则。治理规则规定了微服务治理的过程、治理的策略，可以使用不同的开发框架、技术实现治理规则约定的治理能力。

您可以在 Java Chassis、Go Chassis、Spring Cloud、Dubbo 中使用该功能。

Java Chassis 提供了实现 SDK，可以将其用于其他开发框架。SDK 默认采用 Resilience4j 实现治理过程。规范没有约束治理过程的实现框架，可以很方便的使用其他的治理框架实现治理过程。

治理过程

治理过程可以从如下两个不同的角度进行描述：

- 从管理流程上，可以分为流量标记和设置治理规则两个步骤。系统架构师将请求流量根据特征打上标记，用于区分一个或者一组代表具体业务含义的流量，然后对这些流量设置治理规则。
- 从处理过程上，可以分为下发配置和应用治理规则两个步骤。 可以通过配置文件、配置中心、环境变量等常见的配置管理手段下发配置。微服务 SDK 负责读取配置，解析治理规则，实现治理效果。

治理策略

治理策略分两部分进行描述，一部分描述流量标记， 一部分描述治理规则。

可以根据请求的特征进行流量标记，示例如下：

```
servicecomb:
  matchGroup:
    userLoginAction: |
      matches:
        - apiPath:
            exact: "/login"
          method:
            - POST
        - headers
          Authentication:
            prefix: Basic
```

示例定义了一个流量特征 userLoginAction，如果流量的 apiPath=/login&method=POST，或者请求头 Authentication=Basic*，那么认为这个流量是一个登录操作。

定义好流量特征后，就可以设置治理规则，示例如下：

```
servicecomb:
  rateLimiting:
    userLoginAction: |
      rate: 100
```

示例设置流量特征 userLoginAction 的限流策略是 100TPS。

规范参考

- 流量标记

  示例如下：

```
servicecomb:
  matchGroup:
    userLoginAction: |
      matches:
        - name: loginMethod
          apiPath:
            exact: "/login"
          method:
            - POST
        - name: authHeader
          headers
            Authentication:
              prefix: Basic
```

一个流量对应一个 Key，userLoginAction 为 Key 的名称。 一个流量可以定义多个标记规则，每个标记规则里面可以定义 apiPath，method，headers 匹配规则。 不同标记规则是或的关系，匹配规则是与的关系。

在 match 中提供了一系列的算子来对 apiPath 或者 headers 进行匹配：

- exact：精确匹配。
- prefix: 前缀匹配。
- suffix: 后缀匹配。
- contains: 包含， 目标字符串是否包含模式字符串。
- compare: 比较，支持>、<、>=、<=、=、!=符号匹配。处理时会把模式字符串和目标字符串转化为 Double 类型进行比较，支持的数据范围为 Double 的数据范围。在进行=和!=判断时，如果二者的差值小于 1e-6 就视为相等。例如模式串为>-10，会对大于-10 以上的目标串匹配成功。

流量标记可以在不同的应用层实现，比如：在提供 REST 接口的服务端，可以通过 HttpServletRequest 获取流量信息；在 RestTemplate 调用的客户端，可以从 RestTemplate 获取流量信息。

不同的框架和应用层，提取信息的方式不一样。实现层通过将特征映射到 GovernanceRequest 来屏蔽差异。使得在不同的框架、不同的应用层都可以使用治理。

```
public class GovernanceRequest {
  private Map<String, String> headers;

  private String uri;

  private String method;
}
```

- 限流

  示例如下：

```
servicecomb:
  rateLimiting:
    userLoginAction: |
      limitRefreshPeriod: 1000 //新增许可间隔时间，单位默认毫秒，支持Duration类型时间
      rate: 1 //许可数量
```

限流规则借鉴了 Resilience4j 的思想，作用在服务端，其原理为：

每隔 limitRefreshPeriod 的时间会加入 rate 个新许可，就可以最多接受 rate 个请求，超过的将被限流，返回响应码 429。

- 重试

  根据重试时间间隔的是否固定，分为固定间隔重试和指数间隔重试两种策略，默认重试策略为固定间隔重试。

  固定间隔重试示例如下：

```
servicecomb:
```

```
  retry:
    userLoginAction: |
      maxAttempts: 3 //最大重试次数
      retryOnResponseStatus: [502,503，5xx] //响应码，通过响应码列表指定重试场景，默认是502,5xx表示匹配500-599的响应码
      waitDuration: 0 //重试，单位默认毫秒，支持Duration类型时间
```

指数间隔重试策略示例如下：

```
servicecomb:
  retry:
    userLoginAction: |
      maxAttempts: 3 //最大重试次数
      retryOnResponseStatus: [502,503,5xx] //响应码，通过响应码列表指定重试场景，默认是502，5xx表示匹配500-599的响应码
      retryStrateg: RandomBackoff //重试策略，默认为FixedInterval固定时间间隔策略
      initialInterval: 1000 //基准时间间隔，最小值为10，单位默认毫秒，支持Duration类型时间
      multiplier: 2.0f //乘积因子，单位为float,取值范围大于1.0
```

重试规则借鉴了 Resilience4j 的思想，作用在客户端，其原理为：

如果响应的错误码（502,503）和返回值的计算结果满足重试条件，并且异常在重试异常清单里面，则进行重试，下一次重试等待时间为 waitDuration。重试等待时间和具体的框架与运行机制有关：

同步框架，重试等待时间必须大于等于 0。

异步框架，重试等待时间必须大于 0，否则不会重试，并且重试是在独立的线程池里面执行的。

重试条件是框架相关的，通过扩展 RetryExtension， 不同框架实现机制可能不同：

```
public interface RetryExtension {
  boolean isRetry(List<Integer> statusList, Object result);
  Class<? extends Throwable>[] retryExceptions();
}
```

- 熔断

  示例如下：

```
servicecomb:
  circuitBreaker:
    userLoginAction: |
      failureRateThreshold: 50 //失败率(请求)百分比阈值，取值范围（0,100.0]
      slowCallRateThreshold: 100 //慢调用率阀值，取值范围取值范围（0,100.0]
      slowCallDurationThreshold: 60000 //慢调用请求阈值定义，响应时间超过该阈值的请求都是慢调用，单位默认毫秒，支持Duration类型时间
      minimumNumberOfCalls: 100 //达到熔断条件的请求数量下限
```

```
    slidingWindowType: count //滑动窗口计数类型,默认为time类型
    slidingWindowSize: 100 //滑动窗口大小，当为滑动窗口类型为time类型时，窗口大小表示时间，单位为秒
```

熔断规则借鉴了 Resilience4j 的思想，作用在服务端，其原理为：

达到指定 failureRateThreshold 错误率或者 slowCallRateThreshold 慢请求率时进行熔断，返回响应码 429，慢请求通过 SlowCallDurationThreshold 定义。minimumNumberOfCalls 是达到熔断要求的最低请求数量门槛，例如，若 minimumNumberOfCalls 是 10，为计算失败率，则最小要记录 10 个调用。若只记录了 9 个调用，即使 9 个都失败，CircuitBreaker 也不会打开。slidingWindowType 指定滑动窗口类型，默认可选 count/time， 分别是基于请求数量窗口和基于时间窗口。若滑动窗口为 count，则最近 slidingWindowSize 次的调用会被记录和统计。若滑动窗口为 time，则最近 slidingWindowSize 秒中的调用会被记录和统计。slidingWindowSize 指定窗口大小，根据滑动窗口类型，单位可能是请求数量或者秒。

- 隔离仓

  示例如下：

```
servicecomb:
  bulkhead:
    userLoginAction: |
      maxConcurrentCalls: 1000 //最大信号量
      maxWaitDuration: 0 //最大等待时间间隔，单位默认毫秒，支持Duration类型时间
```

隔离仓规则借鉴了 Resilience4j 的思想，作用在服务端，其原理为：

隔离仓使用了信号量机制，当大量并发请求进入时，如果信号量存在剩余，进入系统的请求会直接获取信号量并开始业务处理。当信号量全被占用时，请求将会进入阻塞状态，如果 maxWaitDuration 时间内无法获取到信号量则系统会拒绝这些请求。若请求在阻塞计时 maxWaitDuration 内获取到了信号量，那么将直接获取信号量并执行相应的业务处理。开启隔离仓时，返回错误码 429。在异步框架，建议 maxWaitDuration 设置为 0，防止阻塞事件派发线程。

1.6.3.3 基于动态配置的流量特征治理开发

微服务引擎提供了简单易用的基于流量标记治理的能力。您可以通过微服务引擎的服务治理功能定义业务特征和治理规则，也可以通过动态配置管理配置项的方式下发治理规则，详情请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 微服务引擎（CSE）> 服务治理 > 治理微服务”章节。

本章节重点介绍和代码开发有关的内容，包括如何配置依赖、涉及的配置项，以及不同微服务开发框架对于流量特征治理支持的差异的内容。

Java Chassis

Java Chassis 通过 Handler 实现了基于流量标记治理能力。其中 Provider 实现了限流、熔断和隔离仓，Consumer 实现了重试。

1. 使用流量标记治理能力，首先需要在代码中引入依赖：

```
<dependency>
```

```
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>handler-governance</artifactId>
</dependency>
```

2. 然后配置 Handler 链：

```
servicecomb:
  handler:
    chain:
      Consumer:
        default: governance-consumer,loadbalance
      Provider:
        default: governance-provider
```

Java Chassis 是基于 Open API 的 REST/RPC 框架，在模型上和单纯的 REST 框架存在差异。Java Chassis 提供两种模式匹配规则，第一种是基于 REST 的，第二种是基于 RPC 的。可以通过配置项：servicecomb.governance.{operation}.matchType 指定匹配规则，默认使用 REST。如果使用 Java Chassis 中的 highway 协议调用，需要指定 matchType 类型为 rpc。比如：

```
servicecomb:
  governance:
    matchType: rest # 设置全局默认是rest匹配模式,highway协议设置为rpc
    GovernanceEndpoint.helloRpc:
      matchType: rpc # 设置服务端的接口helloRpc采用RPC匹配模式
```

在 REST 匹配模式下， apiPath 使用 url， 比如：

```
servicecomb:
  matchGroup:
    userLoginAction: |
      matches:
        - apiPath:
            exact: "/user/login"
```

在 RPC 匹配模式下, apiPath 使用 operation，比如：

```
servicecomb:
  matchGroup:
    userLoginAction: |
      matches:
        - apiPath:
            exact: "UserSchema.login"
```

对于服务端治理，比如限流，REST 模式下从 HTTP 取 header；对于客户端治理，比如重试，REST 模式下从 InvocationContext 取 header。

Spring Cloud

Spring Cloud 通过 Aspect 拦截 RequestMappingHandlerAdater 实现了限流、熔断和隔离仓，通过拦截 RestTemplate 和 FeignClient 实现了重试。

使用流量标记治理能力，首先需要在代码中引入依赖：

```
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-governance</artifactId>
</dependency>
```

Spring Cloud 是基于 REST 的框架，能比较好的符合流量特征治理的匹配语义，apiPath 和 headers 分别对应 HTTP 协议的概念：

```
servicecomb:
  matchGroup:
    userLoginAction: |
      matches:
        - apiPath:
            exact: "/user/login"
          method:
            - POST
        - headers:
          Authentication:
            prefix: Basic
```

Dubbo

Dubbo 的 Provider 通过 Filter 拦截请求实现了限流、熔断和隔离仓，通过拦截 ClusterInvoker 实现了重试。

使用流量标记治理能力，首先需要在代码中引入依赖：

```
<dependency>
  <groupId>com.huaweicloud.dubbo-servicecomb</groupId>
  <artifactId>dubbo-servicecomb-governance-center</artifactId>
  <version>${project.version}</version>
</dependency>
```

如果要使用重试，需要修改 Dubbo 的 Spring 配置文件，将 Dubbo 默认的 ClusterInvoker 修改为 dubbo-servicecomb：

```
 <dubbo:consumer cluster="dubbo-servicecomb"></dubbo:consumer>
```

Dubbo 是一个 RPC 框架，需要定义 operation 和 apiPath 的映射关系，比如，服务端限流、熔断、隔离仓场景：com.huaweicloud.it.order.OrderGovernanceService.hello；客户端重试场景：com.huaweicloud.it.order.OrderGovernanceService.retry。headers 使用 Attachments，需要包含在 Attachments 里面的头才会参与匹配。

```
servicecomb:
```

```
matchGroup:
  userLoginAction: |
    matches:
      - apiPath:
          exact: "com.huaweicloud.it.order.OrderGovernanceService.hello"
        method:
          - POST
      - headers:
          Authentication:
            prefix: Basic
```

自定义

服务治理的默认实现并不一定能够解决业务的所有问题。自定义治理功能可以方便的在不同的场景下使用基于流量的治理能力，比如在网关场景下进行流控，在 Java Chassis 场景下支持 URL 匹配等。SDK 基于 Spring，使用 Spring 的框架都能够灵活的使用这些 API，方法类似。

下面以流控为例，说明如何使用 API。使用 API 开发的自定义代码，也可以通过微服务引擎的管理控制台下发业务和治理规则。

代码的基本过程包括声明 RateLimitingHandler 的引用，创建 GovernanceRequest，拦截（包装）业务逻辑，处理治理异常。

```
 @Autowired
private RateLimitingHandler rateLimitingHandler;

GovernanceRequest governanceRequest = convert(request);

CheckedFunction0<Object> next = pjp::proceed;
DecorateCheckedSupplier<Object> dcs = Decorators.ofCheckedSupplier(next);

try {
  SpringCloudInvocationContext.setInvocationContext();

  RateLimiter rateLimiter = rateLimitingHandler.getActuator(request);
  if (rateLimiter != null) {
    dcs.withRateLimiter(rateLimiter);
  }

  return dcs.get();
} catch (Throwable th) {
  if (th instanceof RequestNotPermitted) {
    response.setStatus(429);
    response.getWriter().print("rate limited.");
    LOGGER.warn("the request is rate limit by policy : {}",
```

```
        th.getMessage());
  } else {
    if (serverRecoverPolicy != null) {
      return serverRecoverPolicy.apply(th);
    }
    throw th;
  }
} finally {
  SpringCloudInvocationContext.removeInvocationContext();
}
```

上面简单的介绍了自定义开发。对于更加深入的使用方式，也可以直接参考 Java Chassis、Spring Cloud、Dubbo 项目中的默认实现代码。

1.6.4 使用 AZ 亲和

在应用没有容器化之前，原先一个虚机上会装多个组件，进程间会有通信。

但在做容器化拆分的时候，往往直接按进程拆分容器。比如：业务进程放在一个容器，监控日志处理或者本地数据放在另一个容器，并且有独立的生命周期。如果两个容器分布在网络中地理距离较远的两个点，请求经过多次转发，性能会变差。

微服务侧配置 AZ 亲和性可以实现就近部署，增强网络能力实现通信上的就近路由，减少网络的损耗。微服务侧的 AZ 亲和是一种特殊的服务治理，和基于流量的服务治理功能一样，AZ 亲和中服务调用优先级顺序为：相同 region 和 availableZone>相同 region 和不同 availableZone>不同 region。

本章节重点介绍微服务侧如何进行 AZ 亲和配置。

说明

本章节描述的 AZ 亲和适用于多个微服务框架：

- Java Chassis 要求 2.3.0 及以上版本。
- Spring Cloud Huawei 要求 1.7.0 及以上版本。

Java Chassis

Java Chassis 默认开启 AZ 亲和调用。若需关闭，将 servicecomb.datacenter.name 参数值设置为空或者在配置信息中删除该参数。AZ 亲和相关信息配置如下所示：

```
servicecomb:
   datacenter:
      name: test #数据中心名称
      region: region1 #数据中心区域
      availableZone: zone1
```

上述配置的具体含义如下：

- servicecomb.datacenter.name：数据中心名称，String 类型。

- servicecomb.datacenter.region：数据中心区域。
- servicecomb.datacenter.availableZone：数据中心可用区。

Spring Cloud

Spring Cloud 默认不开启 AZ 亲和调用。若需开启，请将 spring.cloud.servicecomb.discovery.enabledZoneAware 参数值设置为 true。AZ 亲和相关信息配置如下所示：

```
spring:
  cloud:
    servicecomb:
      discovery:
        enabledZoneAware: true #AZ亲和开关，默认为false
        datacenter:
          name: test #数据中心名称
          region: region1 #数据中心区域
          availableZone: zone1 #数据中心可用区
```

上述配置的具体含义如下：

- spring.cloud.servicecomb.discovery.enabledZoneAware：AZ 亲和开关，boolean 类型，缺省值为 false。
- spring.cloud.servicecomb.discovery.datacenter.name：数据中心名称，String 类型。
- spring.cloud.servicecomb.discovery.datacenter.region：数据中心区域。
- spring.cloud.servicecomb.discovery.datacenter.availableZone：数据中心可用区。

1.6.5 使用仪表盘

仪表盘提供一些基础的微服务运行监控能力。微服务通过 SDK 上报运行状态数据，上报的数据内容包括请求统计数据，比如请求数、时延、错误率等，还包括和治理有关的一些状态，比如熔断状态等。

说明

目前仅支持 Java Chassis 使用仪表盘。

Java Chassis 使用仪表盘，需要在项目中增加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>dashboard</artifactId>
</dependency>
```

如果项目已经直接或者间接包含这个依赖，则无需添加。Java Chassis 包含如表 1-12 所示配置项，指定仪表盘上报地址等信息。

表1-12 Java Chassis 常用配置项

配置项	含义	缺省值
servicecomb.monitor.client.serverUri	上报地址	-
servicecomb.monitor.client.enabled	是否启用数据上报	true
servicecomb.monitor.client.interval	上报周期（毫秒）	10000

1.6.6 使用安全认证

1.6.6.1 安全认证概述

开启了“安全认证”的微服务引擎专享版，通过微服务控制台提供了基于 RBAC（Role-Based Access Control，基于角色的访问控制）的系统管理功能。权限与角色相关联，您可以使用关联了 admin 角色权限的帐号创建新帐号，根据实际业务需求把合适的角色同帐号关联。使用该帐号的用户则具有对该微服务引擎的相应的访问和操作权限。

微服务引擎专享版开启了安全认证之后，所有调用的 API 都需要先获取 token 才能调用，认证流程请参考服务中心 RBAC 说明。

开启了安全认证的微服务引擎专享版，在使用安全认证前需要完成以下工作：

1. 创建安全认证帐号名和密码
2. 配置微服务安全认证的帐号名和密码

说明

- 框架支持安全认证功能的版本要求：Spring Cloud 需要集成 Spring Cloud Huawei 1.6.1 及以上版本，Java Chassis 需要 2.3.5 及以上版本。
- 老版本未开启安全认证的微服务引擎专享版，升级到新版本并开启安全认证的场景，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 基础设施 > 微服务引擎（CSE） > 微服务引擎（CSE）管理 > 管理微服务引擎专享版安全认证”。

1.6.6.2 创建安全认证帐号名和密码

为开启了安全认证的微服务引擎专享版创建帐号名和密码，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 基础设施 > 微服务引擎（CSE）> 系统管理”章节。

1.6.6.3 配置微服务安全认证的帐号名和密码

微服务引擎专享版开启安全认证后，需要对连接到该引擎的微服务组件开启安全认证。开启安全认证是通过配置安全认证帐号名和密码的方式触发。目前支持通过配置文件配置方式和环境变量注入的方式。

由于帐号和密码涉及安全问题，建议加密后使用。

Spring Cloud 微服务组件配置安全认证帐号名和密码

- 配置文件配置方式

  为微服务的“bootstrap.yml”文件增加以下配置，若已配置请忽略。

```
spring:
  cloud:
    servicecomb:
      credentials:
        account:
          name: test   #结合用户实际值配置
          password: mima  #结合用户实际值配置
          cipher: default
```

说明

用户密码 password 默认为明文存储，无法保证安全。建议您对密码进行加密存储，请参考自定义实现 password 的加密存储算法。

- 环境变量注入方式

  为微服务添加如表 1 所示环境变量。

  添加环境变量，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 应用高级设置 > 设置应用环境变量”章节。

表1-13 环境变量

环境变量	说明
spring_cloud_servicecomb_credentials_account_name	结合用户实际值配置。
spring_cloud_servicecomb_credentials_account_password	结合用户实际值配置。 说明 用户密码 password 默认为明文存储，无法保证安全。建议您对密码进行加密存储，请参考自定义实现 password 的加密存储算法。

Java Chassis 微服务组件配置安全认证帐号名和密码

- 配置文件配置方式

  为微服务的“microservice.yml”文件增加以下配置，若已配置请忽略。

```
servicecomb:
  credentials:
    rbac.enabled: true  #结合用户实际值配置
    cipher: default
    account:
      name: test #结合用户实际配置
      password: mima #结合用户实际配置
      cipher: default
```

说明

用户密码 password 默认为明文存储，无法保证安全。建议您对密码进行加密存储，请参考配置安全认证参数。

- 环境变量注入方式

  为微服务添加如表 2 所示环境变量。

  添加环境变量，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 应用高级设置 > 设置应用环境变量”章节。

表1-14 环境变量

环境变量	说明
servicecomb_credentials_rbac_enabled	• true：开启RBAC。 • false：不开启RBAC。
servicecomb_credentials_account_name	结合用户实际值配置。
servicecomb_credentials_account_password	结合用户实际值配置。 说明 用户密码 password 默认为明文存储，无法保证安全。建议您对密码进行加密存储，请参考配置安全认证参数。

1.7 附录

1.7.1 Java Chassis 版本升级参考

- 使用 2.1.3 版本之前的 Java Chassis 接入微服务引擎

  a. 需要额外引入 CSE SDK。 引入 CSE SDK 使用如下 Maven Dependency Management:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.huawei.paas.cse</groupId>
            <artifactId>cse-dependency</artifactId>
```

```
            <version>版本号</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

b. 并引入依赖：

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-solution-service-engine</artifactId>
</dependency>
```

引入 CSE SDK，Maven Settings 需要增加额外的仓库：

```
    <repositories>
        <repository>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            <id>huaweicloudsdk-releases</id>
            <name>huaweicloudsdk</name>

<url>https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
        </repository>
    </repositories>
```

- 升级到 2.1.3 及以上版本

a. 需要修改 Maven Dependency Management：

```
 <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.apache.servicecomb</groupId>
        <artifactId>java-chassis-dependencies</artifactId>
        <version>${java-chassis.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

b. 并引入依赖：

```
<dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>solution-basic</artifactId>
</dependency>
<dependency>
```

```
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>servicestage-environment</artifactId>
</dependency>
```

如果依赖了其他 groupId 为 com.huawei.paas.cse 的软件包，删除即可。2.1.3 之后，所有软件包可以从 Maven 中央库获取，不需要额外配置 Maven 仓库。

1.7.2 AK/SK 认证方式排查与切换指导

说明

新安装的环境下，ServiceStage 不再提供对微服务引擎专业版的支持。

步骤 1 请确认您是否使用微服务引擎专业版。

- 是，执行步骤 2。
- 否，操作结束。

步骤 2 请确认您是否使用 ServiceStage 容器部署方式部署您的微服务应用。

- 是，执行步骤 3。
- 否，操作结束。

步骤 3 为微服务应用配置 AK/SK，请参考为微服务应用配置 AK/SK。

----结束

1.7.3 为微服务应用配置 AK/SK

1.7.3.1 Java Chassis

方法一

为微服务的“microservice.yml”文件增加以下配置，若已配置请忽略。

AK/SK 与项目名称获取方法，请参考获取 AK/SK 与项目名称。

```
servicecomb:
  credentials:
    accessKey: AK  #结合用户实际值配置
    secretKey: SK  #结合用户实际值配置
    project: 项目名称  #结合用户实际值配置
    akskCustomCipher: default
```

方法二

为微服务添加如表 1-15 所示环境变量。

- 添加环境变量，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 应用高级设置 > 设置应用环境变量”章节。
- AK/SK 获取方法，请参考获取 AK/SK 与项目名称。

表1-15 环境变量

环境变量	说明
servicecomb_credentials_accessKey	AK，结合用户实际值配置。
servicecomb_credentials_secretKey	SK，结合用户实际值配置。

1.7.3.2 Go Chassis

方法一

请为微服务的“chassis.yaml”或“auth.yaml”文件增加以下配置，若已配置请忽略。

AK/SK 与项目名称获取方法，请参考获取 AK/SK 与项目名称。

```
cse:
  credentials:
    accessKey: AK  #结合用户实际值配置
    secretKey: SK  #结合用户实际值配置
    project: 项目名称  #结合用户实际值配置
    akskCustomCipher: default
```

方法二

为微服务添加如表 1-16 所示环境变量。

- 添加环境变量，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 应用高级设置 > 设置应用环境变量”。
- AK/SK 获取方法，请参考获取 AK/SK 与项目名称。

表1-16 环境变量

环境变量	说明
cse_credentials_accessKey	AK，结合用户实际值配置。
cse_credentials_secretKey	SK，结合用户实际值配置。

1.7.3.3 Spring Cloud

方法一

为微服务的“bootstrap.yml”文件增加以下配置，若已配置请忽略。

AK/SK 与项目名称获取方法，请参考获取 AK/SK 与项目名称。

```
spring:
  cloud:
    servicecomb:
      credentials:
        enabled: true
        accessKey: AK  #结合用户实际值配置
        secretKey: SK  #结合用户实际值配置
        project: 项目名称  #结合用户实际值配置
        akskCustomCipher: default
```

方法二

为微服务添加如表 1-17 所示环境变量。

- 添加环境变量，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理 > 应用高级设置 > 设置应用环境变量”章节。
- AK/SK 获取方法，请参考获取 AK/SK 与项目名称。

表1-17 环境变量

环境变量	说明
spring_cloud_servicecomb_credentials_accessKey	AK，结合用户实际值配置。
spring_cloud_servicecomb_credentials_secretKey	SK，结合用户实际值配置。

1.7.3.4 Mesher

参考以下操作步骤创建一个名为“mesher-secret”密钥。创建密钥前：

1. 已获取 AK/SK，请参考获取 AK/SK 与项目名称。
2. 对获取到的 AK/SK 进行 Base64 编码。

   可以直接使用 **echo -n** '*要编码的内容*' **| base64** 命令即可，示例如下：

   ```
   root@ubuntu:~# echo -n '3306' | base64
   MzMwNg==
   ```

   其中，3306 为要编码的内容。

操作步骤

步骤 1 登录 ServiceStage 控制台。

步骤 2 创建密钥，请参考“帮助中心 > 应用管理与运维平台 > 用户指南 > 应用管理> 应用配置管理 > 创建密钥”章节。

1. “创建方式”选择“可视化”。
2. “密钥名称”填写为“mesher-secret”。
3. “所在集群”选择部署应用的集群。

4. “命名空间”选择部署应用的命名空间。
5. “密钥类型”选择“Opaque”。
6. “密钥数据”请按以下表格填写。

表1-18 秘钥数据

键	值
cse_credentials_accessKey	AK，结合用户实际值配置，需进行 base64 编码。
cse_credentials_secretKey	SK，结合用户实际值配置，需进行 base64 编码。

----结束

1.7.4 获取 AK/SK 与项目名称

获取 AK/SK 访问密钥

说明

请以应用所需权限用户登录 ServiceStage 控制台。

步骤 1 登录 ServiceStage 控制台。

步骤 2 鼠标移动到用户名，在下拉菜单选择“我的凭证”。

步骤 3 在导航栏，单击“访问密钥”。

步骤 4 单击“新增访问密钥”，通过身份认证后成功创建 AK/SK。

步骤 5 单击“立即下载”。

下载成功后，在 credentials 文件中获取 AK 和 SK 信息：

- Access Key Id 的值即为 AK。
- Secret Access Key 的值即为 SK。

须知

- 每个用户仅允许保留 2 个有效的访问秘钥。
- 为保证访问密钥的安全，访问密钥仅在初次生成时自动下载，后续不可再次通过管理控制台界面获取。请妥善保管访问密钥。

----结束

获取项目名称

步骤 1 登录 ServiceStage 控制台。

步骤 2 鼠标移动到用户名，在下拉列表选择“统一身份认证”。

步骤 3 在中间导航栏，单击“项目”。

在“项目”列中查看项目名称。

----结束

1.7.5 本地开发工具说明

本地开发工具包含了本地轻量化微服务引擎，用于本地开发的轻量服务中心、配置中心，并提供简单的界面。

使用说明请参考 README 文件。

表1-19 本地轻量化微服务引擎版本说明

版本	发行时间	获取路径
1.0.10	2021.09.14	Local-CSE-1.0.10.zip 说明 1.0.10 版本本地轻量化微服务引擎仅作为本地开发调测，请勿用于商业使用。

2 修订记录

发布日期	修订记录
2022-06-22	第二次正式发布。 修改“Spring Cloud 接入 CSE”章节部分描述错误。
2022-05-06	第一次正式发布。