微服务引擎

# 开发指南

文档版本 01

发布日期 2024-05-16

华为云计算技术有限公司

**商标声明**


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

**注意**

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。


# 华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

# 目 录



# 1 概述

## 1.1 开发简介

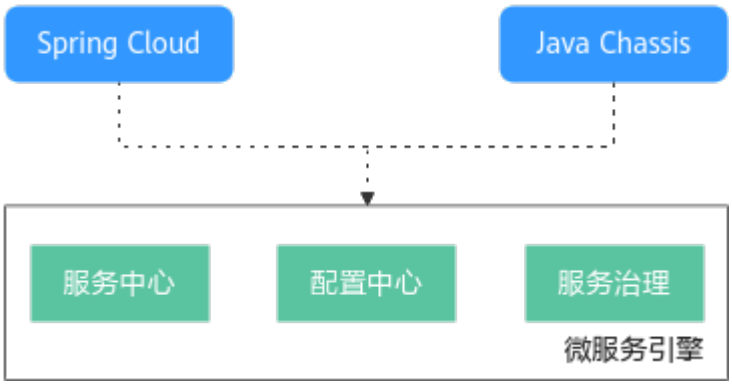
### 微服务简介

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（Remote Procedure Call，远程过程调用）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（HTTP+JSON）、gRPC（HTTP2+protobuffer）、Web Service（HTTP+SOAP）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、容错、限流、负载均衡、降级等微服务治理能力。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**Spring Cloud**、**Apache ServiceComb Java Chassis**（简称Java Chassis）等，实现了微服务架构模式。ServiceComb引擎支持这些开源的微服务框架接入并使用注册发现、集中配置、服务治理等功能。关系如下图所示：




使用Spring Cloud和Java Chassis微服务开发框架接入ServiceComb引擎，能够获得好的开发体验和技术支持。使用其他开发框架，比如使用Mesher接入ServiceComb引擎依赖于开源社区技术支持。

本文重点描述Spring Cloud和Java Chassis的开发指导，其他框架如Mesher开发的微服务应用使用ServiceComb引擎请参考**Mesher使用ServiceComb引擎指南**。

### 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ServiceComb引擎的功能，假设您已经熟悉和掌握如下开发能力：

- 使用Java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ServiceComb引擎上运行。本文档提供微服务应用接入ServiceComb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。
- 熟悉应用部署，请参考**创建并部署组件**。

## 1.2 常用概念

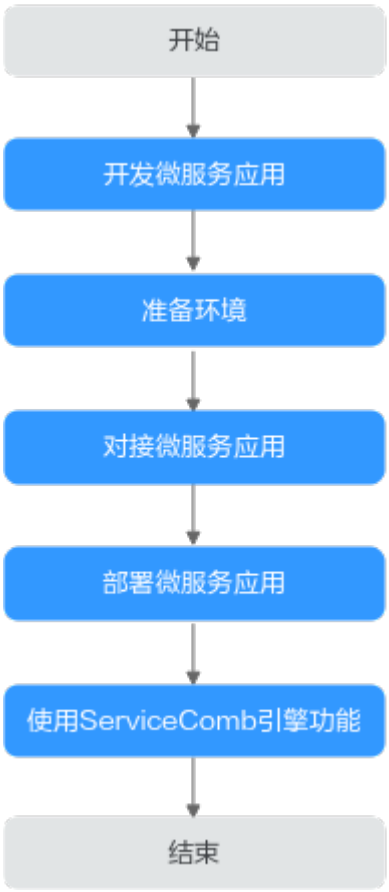
- 应用：可以将应用理解为完成某项完整业务场景的软件系统。应用一般由多个微服务组成，应用里面的微服务能够相互发现和调用。
- 微服务：完成某项具体业务功能的软件系统。微服务是独立开发、部署的单元。
- 微服务实例：将微服务采用部署系统部署到运行环境，就产生了实例。可以将实例理解为一个进程，一个微服务可以部署若干实例。
- 微服务环境：服务中心建立的一个逻辑概念，比如development、production等。不同环境里面的微服务实例逻辑隔离、无法相互发现和调用。

# 1.3 开发流程

## 开发流程概述

开发应用和使用ServiceComb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

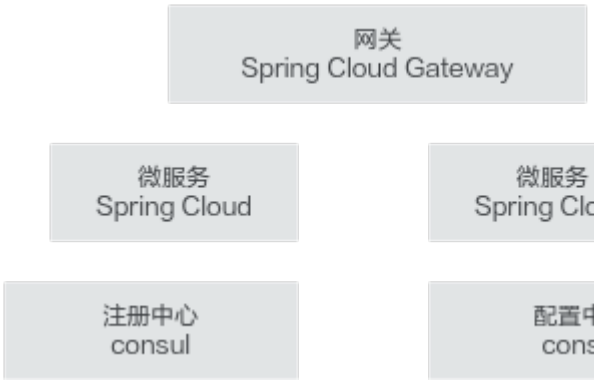
1. 开发微服务应用

   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**。

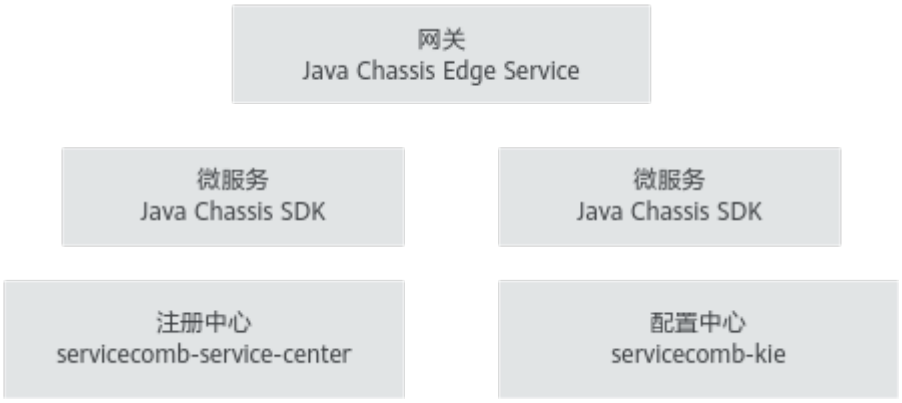
   进行微服务应用开发，首先需要进行技术选型。技术选型是一个复杂的问题，技术决策者需要考虑使用的技术是否容易被团队成员掌握，技术能否满足项目对于功能、性能、可靠性方面的要求，还需要考虑商业服务等多方面的因素。本文档不探讨技术选型，假设技术团队已经选择了适合自己的开发框架。大部分技术团队都会选择开源框架来构建业务。

   开发微服务应用的具体内容，请参考**开发微服务应用**。

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




- 使用Java Chassis，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ServiceComb引擎接入调试、云上应用部署和使用ServiceComb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ServiceComb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ServiceComb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ServiceComb引擎，详细内容请参考**部署微服务应用**。

5. 使用ServiceComb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ServiceComb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ServiceComb引擎功能**。

# 1.4 开发规范

## 开发语言要求

使用Java语言进行微服务开发。

## ServiceComb 引擎微服务开发框架版本要求

微服务开发框架推荐版本如下表所示。

- 如果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。
- 如果已使用Spring Cloud微服务开发框架开发了应用，推荐使用**Spring Cloud Huawei**接入应用。
- 如果基于开源开放和业界生态组件新开发微服务应用，可选择Spring Cloud框架。
- 如果希望使用ServiceComb引擎提供的开箱即用的治理能力和高性能的RPC框架，可选择Java Chassis框架。

| 框架 | 推荐版本 | 说明 |
| --- | --- | --- |
| Spring Cloud Huawei | 1.10.9-2021.0.x及以上 | 采用**Spring Cloud Huawei**项目提供接入支持：<br>● 适配的Spring Cloud版本为2021.0.5<br>● 适配的Spring Boot版本为2.6.13<br>Spring Cloud微服务开发框架的版本说明请参见：**https://github.com/huaweicloud/spring-cloud-huawei/releases**。 |
| Java Chassis | 2.7.10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。<br>Java Chassis微服务开发框架的版本说明请参见：**https://github.com/apache/servicecomb-java-chassis/releases**。 |

**须知**

系统升级、改造过程中，三方软件冲突是最常见的问题。随着软件迭代速度越来越快，传统的软件兼容性管理策略已经不适应软件的发展，您可以参考**三方软件版本管理策略**来解决版本冲突。

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

  - 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**

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

  - 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/**

    ServiceComb引擎推荐示例：**https://github.com/apache/servicecomb-samples/tree/master/basic**

# 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ServiceComb引擎专享版，并开放公网访问的IP，保证本地环境能够访问。

## 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：

- 创建ServiceComb引擎，请参考**创建ServiceComb引擎**。
- 创建环境，请参考**创建环境**。创建的环境，需包含CCE集群、ELB及ServiceComb引擎等资源。
- 创建应用，请参考**创建应用**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 3-1** 常用环境变量

| 环境变量名称 | 含义 |
| --- | --- |
| PAAS_CSE_SC_ENDPOINT | ServiceComb引擎注册中心地址信息。 |
| PAAS_CSE_CC_ENDPOINT | ServiceComb引擎配置中心地址信息。 |
| PAAS_PROJECT_NAME | 项目名称。 |

| 环境变量名称 | 含义 |
| --- | --- |
| 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 3-2** 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 |
| 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）是华为云提供的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 3-3** 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 |

- 云数据库

  **云数据库**（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。

  云数据库相关环境变量如下表所示。

**表 3-4** RDS 关系型数据库相关环境变量

| 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 |

# 4 对接微服务应用

## 4.1 Spring Cloud 接入 ServiceComb 引擎

本章节介绍Spring Cloud如何接入ServiceComb引擎，使得Spring Cloud能够对接ServiceComb引擎，并且方便的使用ServiceComb引擎提供的最常用的功能。在**使用ServiceComb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ServiceComb引擎需要使用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：版本要求请参见**ServiceComb引擎微服务开发框架版本要求**。
- 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>
```

**说明**

上述spring-cloud-starter-huawei-service-engine模块包含以下依赖模块：

```
<!--  注册发现模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-discovery</artifactId>
</dependency>
<!--  配置中心模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-config</artifactId>
</dependency>
<!--  服务治理模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-governance</artifactId>
</dependency>
<!--  灰度发布模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-router</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>
```

**说明**

上述spring-cloud-starter-huawei-service-engine-gateway模块包含以下依赖模块：

```
<!--  注册发现模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-discovery</artifactId>
</dependency>
<!--  配置中心模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-config</artifactId>
</dependency>
<!--  服务治理模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-governance</artifactId>
</dependency>
<!--  灰度发布模块  -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-router</artifactId>
</dependency>
<!--  网关模块  -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-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。ServiceComb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: 15
      config:
        serverAddr: http://127.0.0.1:{port}
        serverType: {servertype}
```

**说明**

- healthCheckInterval参数配置值的单位为秒。
- 当ServiceComb引擎版本为1.x时，{*port*}取值为30103，{*servertype*}取值为config-center。
- 当ServiceComb引擎版本为2.x时，{*port*}取值为30110，{*servertype*}取值为kie或config-center，推荐使用kie作为配置中心。

**步骤3** （可选）配置安全认证参数。

使用ServiceComb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。

ServiceComb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ServiceComb引擎获取用户名和密码，然后在配置文件中增加如下配置。

- 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算法

```
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RBAC功能，需要1.6.0-Hoxton及以上版本。

**----结束**

# 4.2 Java Chassis 接入 ServiceComb 引擎

本章节介绍Java Chassis如何接入ServiceComb引擎，使得Java Chassis能够对接ServiceComb引擎，并且方便的使用ServiceComb引擎提供的最常用的功能。在**使用ServiceComb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：请参见**ServiceComb引擎微服务开发框架版本要求**。
- 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>
```

**说明**

- 上述模块solution-basic包含常用的Java Chassis功能，例如配置中心模块和服务治理模块，方便一键式启用Java Chassis常见功能。

```
<!--  配置中心模块  -->
<dependency>
<groupId>org.apache.servicecomb</groupId>
<artifactId>config-cc</artifactId>
</dependency>
<!--  服务治理模块  -->
<dependency>
<groupId>org.apache.servicecomb</groupId>
<artifactId>handler-governance</artifactId>
</dependency>
```

- 上述模块servicestage-environment包含以下依赖模块：

```
<!--  注册发现模块  -->
<dependency>
 <groupId>org.apache.servicecomb</groupId>
 <artifactId>registry-service-center</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ServiceComb引擎最新的功能要求。如果期望后续升级新版本保持稳定而不是跟随ServiceComb引擎演进，您可以选择不依赖servicestage-environment，而是在您自己的项目中增加“mapping.yaml”。

“microservice.yaml”和“mapping.yaml”文件，一般都存放于您当前项目根目录下的“/src/main/resources/”路径下。

```
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ServiceComb引擎最新的功能要求。如果期望后续升级新版本保持稳定而不是跟随ServiceComb引擎演进，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）配置安全认证参数。

使用ServiceComb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。

ServiceComb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ServiceComb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ServiceComb引擎获取的用户名
      password: your password # 从ServiceComb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。通过自定义实现加密，如下所示：

- 自定义实现，首先实现接口“org.apache.servicecomb.foundation.auth.Cipher”，里面的两个方法：
  - String name()

    这个是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。

  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/service/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:
      rbac.enabled: true
      account:
        name: your account name
        password: your password # 加密后的密码
        cipher: youciphername  #实现类里面的name()方法返回的名称
  ```

**说明**

明文存储无法保证安全， 建议您对密码进行加密存储。

**----结束**

# 5 部署微服务应用

微服务应用部署，请参考**创建并部署组件**。

# 6 使用 ServiceComb 引擎功能

## 6.1 使用服务注册

ServiceComb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ServiceComb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。

### 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6-1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 6-1** 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 | - |

| 配置项 | 含义 | 缺省值 | 备注 |
| --- | --- | --- | --- |
| 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.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 | false | - |
| spring.cloud.servicecomb.discovery.healthCheckInterval | 发送心跳的时间间隔（秒） | 15 | 可配置的取值范围：≥ 1且≤600。 |
| 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。 |

## 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6-2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 6-2** 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 | - |
| 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6-3**所示。

**表 6-3** 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 |

# 6.2 使用配置中心

## 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。

ServiceComb引擎支持的配置中心为：config-center和kie。

**说明**

- 当ServiceComb引擎版本为1.x时，其配置中心为config-center。
- 当ServiceComb引擎版本为2.x时，其配置中心为kie或config-center，推荐使用kie作为配置中心。

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。

- ServiceComb引擎使用kie作为配置中心。

  微服务默认会读取配置中心应用配置、服务配置、自定义配置。应用配置指环境、应用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。微服务可以在配置文件中指定一个特定的label及label值，自定义配置指label及label值与微服务相同的配置。

  简单的场景，可以使用应用级配置和服务级配置。应用级配置被该应用下的所有微服务共享，是公共配置；服务级配置只对具体微服务生效，是独享配置。

  复杂的场景，可以通过使用customLabel和customLabelValue来定义配置。例如某些配置，是对所有应用共享的，那么就可以使用这个机制。在配置文件增加如下配置（以Spring Cloud为例）：

```
spring:
  cloud:
    servicecomb:
      config:
        kie:
          customLabel: public# 默认值是public
          customLabelValue: default # 默认值是空字符串
```

只要配置项带有public标签，并且标签值为default，这些配置项就会对该微服务生效。

a. 把配置中心当成数据库的一个表tbl_configurations，key是主键，每个label都是属性。

b. 客户端会根据如下3个条件查询配置：

- 自定义配置

  **select * from tbl_configurations where customLabel=customLabelValue & match=false**

- 应用级配置

  **select * from tbl_configurations where app=demo_app & environment=demo_environment & match=true**

- 服务级配置

  **select * from tbl_configurations where app=demo_app & environment=demo_environment & service=demo_service & match=true**

其中，match为true的时候，表示有且只有条件里面指定的属性；match为false的时候，表示除了条件里面的属性，允许有其他的属性。还可以给标签app指定多个应用，或者给标签service指定多个服务，这样配置项就可以对多个服务和应用生效，非常灵活。

ServiceComb引擎的TEXT、XML等类型，SDK会简单的当成key-value对使用；YAML和Properties类型， SDK会解析内容，应用程序将内容作为实际的应用程序配置项。比如：

```
类型：TEXT
key: cse.examples.hello
value: World
```

应用程序会发现1个配置项: cse.examples.hello = World。

```
类型：YAML
key: cse.examples.hello
value: |
  cse:
    key1: value1
    key2: value2
```

应用程序会发现2个配置项: cse.key1 = value1和cse.key2 = value2。

- ServiceComb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。

  ServiceComb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，“配置项”及其对应的“值”的示例如下表所示。其中，值的格式为yaml。

| 配置项 | 值 |
| --- | --- |
| file1.yaml | cse.example.key1: value1<br>cse.example.key2: value2 |
| file2.yaml | cse.example.key3: value3<br>cse.example.key4: value4 |

配置创建方法请参考**配置管理**中的“创建配置”操作。

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。

## 6.2.2 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6-4**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 6-4** 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 | - |

| 配置项 | 含义 | 缺省值 | 备注 |
| --- | --- | --- | --- |
| spring.cloud.servicecomb.config.serverType | 配置中心类型 | config-center | • 当ServiceComb引擎为1.x版本时，需设置为config-center。<br>• 当ServiceComb引擎为2.x版本时，可设置为kie或config-center，推荐使用kie作为配置中心。 |
| 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 | - | 仅当配置中心类型为config-center时生效。 |

对于使用ServiceComb引擎1.x版本的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**。

## 6.2.3 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6-5**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 6-5** 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.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.tenantName | 应用的租户名称 | default | config-center |

- Java Chassis使用以kie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-kie</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6-6**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 6-6** 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 |

| 配置项 | 含义 | 缺省值 | 备注 |
| --- | --- | --- | --- |
| servicecomb.kie.serverUri | kie访问地址，格式为http(s)://{ip}:{port}，以“,”分隔多个地址 | - | kie |
| servicecomb.kie.firstRefreshInterval | 首次刷新配置项的时间间隔，单位为毫秒 | 3000 | kie |
| servicecomb.kie.refresh_interval | 刷新配置项的时间间隔，单位为毫秒 | 3000 | kie |
| servicecomb.kie.domainName | 应用的租户名称 | default | kie |

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**。

# 6.3 使用服务治理

## 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。
- 限流：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秒内到来的1000个请求平均分布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。
- 降级：降级治理是在业务高峰期时，需要临时减少对于目标服务的访问，达到降低目标服务负载；或者屏蔽对于非关键服务的访问，保持本服务的核心处理能力的治理措施。

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，ServiceComb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。如果使用微服务框架开发应用，在应用托管后启动应用，微服务会自动注册到对应的ServiceComb引擎，您可以到微服务引擎控制台进行服务治理的相关操作请参考**治理微服务**。

本章节重点介绍如何使用基于流量特征的服务治理能力。

## 6.3.2 流量标记

- 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。

  a. 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>
  ```

  b. 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。

  不同治理策略配置示例及在POM中添加依赖如下介绍。

  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;}
```

- 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
```

## 6.3.3 限流

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。

- Java Chassis的限流作用于微服务提供者，需要微服务应用集成流量控制模块，启用qps-flowcontrol-provider处理链。

  配置示例如下：

```
servicecomb:
  handler:
    chain:
      Provider:
        default: qps-flowcontrol-provider
```

  在POM中添加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>handler-flowcontrol-qps</artifactId>
  <version>${project.version}</version>
</dependency>
```

详细说明请参考**servicecomb限流开发指南**。

- Spring Cloud通过Aspect拦截RequestMappingHandlerAdater实现了限流，集成Spring Cloud Huawei以后，默认集成了限流模块spring-cloud-starter-huawei-governance， 只需要通过配置开启具体的限流策略。

  配置示例如下：

```
servicecomb:
  matchGroup:
    AllOperation: |
      matches:
        - apiPath:
            prefix: "/"
  rateLimiting:
    AllOperation: |
      rate: 10 #在一段时间内只允许10次请求
```

## 6.3.4 容错

容错的原理为：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。

- Java Chassis的容错作用于微服务消费者，需要微服务应用集成容错模块，启用**bizkeeper**处理链。

  配置示例如下：

```
servicecomb:
  handler:
    chain:
      Consumer:
        default: bizkeeper-consumer
```

在POM中添加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>handler-bizkeeper</artifactId>
  <version>${project.version}</version>
</dependency>
```

详细说明请参考**流量特征治理**。

**说明**

此处以微服务开发框架Java Chassis 2.x版本为例介绍。3.x版本配置详情请参考**流量特征治理**。

- Spring Cloud通过Aspect拦截RequestMappingHandlerAdater实现了容错，集成Spring Cloud Huawei以后，默认集成了客户端容错模块spring-cloud-starter-huawei-governance， 只需要通过配置开启具体的客户端容错策略。

  配置示例如下：

```
servicecomb:
  matchGroup:
    AllOperation: |
      matches:
        - apiPath:
            prefix: "/"
  retry:
    AllOperation: |
      maxAttempts: 3  # 重试次数
      retryOnSame: 1  # 重试请求发起的实例
```

```
    retryOnResponseStatus:  # 重试错误码
      - 502
      - 503
```

默认策略是在异常错误码为502、503情况下生效，1.11.4-2021.0.x/1.11.4-2022.0.x版本开始支持响应头header的特殊场景生效。

响应头header设置key默认为"X-HTTP-STATUS-CODE"，也支持自定义设置，配置如下：

```
spring:
  cloud:
    servicecomb:
      governance:
        response:
          header:
            status:
              key: 'X-HTTP-EEROR-STATUS-CODE'
```

同样响应头header中设置的响应码也支持自定义，但是需要在容错策略中增加对应的错误码，例如设置X-HTTP-STATUS-CODE=511，那么错误码中增加511错误码，配置如下：

```
servicecomb:
  matchGroup:
    AllOperation: |
      matches:
        - apiPath:
            prefix: "/"
  retry:
    AllOperation: |
      maxAttempts: 3  # 重试次数
      retryOnSame: 1  # 重试请求发起的实例
      retryOnResponseStatus:  # 重试错误码
        - 502
        - 503
        - 511
```

策略执行顺序为优先判断响应码，如果异常响应码满足策略设置，容错开启；响应码不满足后，再判断header设置的响应码是否满足条件。

## 6.3.5 熔断

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。

- Java Chassis的熔断作用于微服务消费者，需要微服务应用集成熔断模块，启用bizkeeper-consumer处理链。

  配置示例如下：

  ```
  servicecomb:
    handler:
      chain:
        Consumer:
          default: bizkeeper-consumer
  ```

  在POM中添加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>handler-bizkeeper</artifactId>
  <version>${project.version}</version>
</dependency>
```

详细说明请参考**流量特征治理**。

**说明**

此处以微服务开发框架Java Chassis 2.x版本为例介绍。3.x版本配置详情请参考**流量特征治理**。

- Spring Cloud Huawei通过Aspect拦截RequestMappingHandlerAdater实现了熔断，集成Spring Cloud Huawei以后，默认集成了客户端熔断模块spring-cloud-starter-huawei-governance， 只需要通过配置开启具体的客户端熔断策略。

  配置示例如下：

  ```
  servicecomb:
    matchGroup:
      AllOperation: |
        matches:
          - apiPath:
              prefix: "/"
    instanceIsolation:
      AllOperation: |
        minimumNumberOfCalls: 10
        slidingWindowSize: 10
        slidingWindowType: COUNT_BASED
        failureRateThreshold: 20
        recordFailureStatus:
          - 502
          - 503
  ```

  默认策略是在异常错误码为502、503情况下生效，1.11.4-2021.0.x/1.11.4-2022.0.x版本开始支持响应头header的特殊场景生效。

  响应头header设置key默认为"X-HTTP-STATUS-CODE"，也支持自定义设置，只需要在客户端配置如下配置：

  ```
  spring:
    cloud:
      servicecomb:
        governance:
          response:
            header:
              status:
                key: 'X-HTTP-EEROR-STATUS-CODE'
  ```

  同样响应头header中设置的响应码也支持自定义，但是需要在容错策略中增加对应的错误码，例如设置X-HTTP-STATUS-CODE=511，那么错误码中增加511错误码，配置如下：

  ```
  servicecomb:
    matchGroup:
      AllOperation: |
        matches:
          - apiPath:
              prefix: "/"
    instanceIsolation:
      AllOperation: |
        minimumNumberOfCalls: 10
        slidingWindowSize: 10
        slidingWindowType: COUNT_BASED
        failureRateThreshold: 20
        recordFailureStatus:
          - 502
          - 503
          - 511
  ```

上述配置对所有实例启用了客户端熔断策略。 该策略采用COUNT_BASED滑动窗口策略，窗口大小为10个请求，到达10个请求的时候，开始计算错误率，如果错误率达到20%，就会对后续请求进行熔断。默认的滑动窗口策略是TIME_BASED。策略执行顺序为优先判断响应码，如果异常响应码满足策略设置，容错开启；响应码不满足后，再判断header设置的响应编码是否满足条件。

## 6.3.6 隔离仓

隔离是一种异常检测机制，常用的检测方法是请求超时、流量过大等。一般的设置参数包括超时时间、同时并发请求个数等。

- Java Chassis的隔离作用于微服务消费者，需要微服务应用集成隔离模块，启用bizkeeper-consumer处理链。

  配置如下：

```
servicecomb:
  handler:
    chain:
      Consumer:
        default: bizkeeper-consumer
  isolation:
    Consumer:
      timeout:
        enabled: true  #是否启用超时检测
      timeoutInMilliseconds: 30000 #超时时间阈值
```

- Spring Cloud Huawei的隔离策略同熔断一致，配置示例请参考**熔断**中相关内容。

## 6.3.7 负载均衡

负载均衡作用在客户端，是高并发、高可用系统必不可少的关键组件，目标是尽力将网络流量平均分发到多个服务器上，以提高系统整体的响应速度和可用性。

- Java Chassis的负载均衡作用于微服务消费者，需要微服务应用集成负载均衡模块，启用loadbalance处理链。

  配置示例如下：

```
servicecomb:
  handler:
    chain:
      Consumer:
        default: loadbalance
  loadbalance:
    strategy:
      name: RoundRobin # 这是开启负载均衡轮询模式,随机模式为Random
```

  在POM中添加如下依赖：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>handler-loadbalance</artifactId>
  <version>${project.version}</version>
</dependency>
```

- Spring Cloud Huawei负载均衡规则使用了Spring Cloud里面Ribbon的思想，作用在客户端，其原理为：当使用随机规则时，客户端会在下游微服务实例中随机访问一个实例，当使用轮询规则时，客户端会在下游微服务实例中按顺序循环选择Server。

  配置示例如下：

```
servicecomb:
  loadbalance:
    userLoginAction: |
      rule: Random # 说明负载均衡规则为随机模式，默认是轮询模式
```

## 6.3.8 降级

降级治理是在业务高峰期时，需要临时减少对于目标服务的访问，达到降低目标服务负载；或者屏蔽对于非关键服务的访问，保持本服务的核心处理能力的治理措施。

- Java Chassis降级治理是作用于微服务消费者，需要微服务应用集成降级模块，启用bizkeeper-consumer处理链。

  配置示例如下：

  ```
  servicecomb:
    handler:
      chain:
        Consumer:
          default: bizkeeper-consumer
  ```

  在POM中添加如下依赖：

  ```
  <dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>handler-bizkeeper</artifactId>
    <version>${project.version}</version>
  </dependency>
  ```

  详细说明请参考**流量特征治理**。

  **说明**

  此处以微服务开发框架Java Chassis 2.x版本为例介绍。3.x版本配置详情请参考**流量特征治理**。

- Spring Cloud Huawei降级是一种治理规则，集成Spring Cloud Huawei以后，默认集成了客户端治理模块spring-cloud-starter-huawei-governance其原理为：当被流量标的路径被请求时，所有的请求百分百都会返回null，forceClosed是强制关闭降级治理的配置参数，当为true时降级治理会被强制关闭，默认值为false。

  配置示例如下：

  ```
  servicecomb:
    matchGroup:
      demo-test-fallback: |
        matches:
          - serviceName: "MyMicroservice"
            apiPath:
              prefix: "/"
    faultInjection:
      demo-test-fallback: |
        type: abort
        percentage: 100
        fallbackType: ReturnNull
        forceClosed: false
  ```

  当上述配置开启时访问MyMicroservice的任意接口的请求都会被阻拦并返回错误码为500的FaultInjectionException。

## 6.3.9 错误注入

用户在consumer端使用故障注入，可以设置发往指定微服务的请求的时延和错误及其触发概率用来在业务高峰期时保护核心业务只被关键微服务访问。

**说明**

Spring Cloud Huawei暂不支持错误注入治理策略。

Java Chassis错误注入治理是作用于微服务消费者，需要微服务应用集成错误注入模块，启用fault-injection-consumer处理链。

```
servicecomb:
  handler:
    chain:
      Consumer:
        default:  loadbalance,fault-injection-consumer
```

在POM中添加如下依赖：

```
<dependency>
   <groupId>org.apache.servicecomb</groupId>
   <artifactId>handler-fault-injection</artifactId>
   <version>${project.version}</version>
</dependency>
```

详细说明请参考**servicecomb错误注入开发指南**。

## 6.3.10 自定义治理

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ServiceComb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项目中的默认实现代码。

### 6.3.11 黑白名单

只有启用了公钥认证，设置的黑白名单才能生效。

## 6.4 使用灰度发布

为保障新特性平稳上线，可以先选择少部分用户试用，待新特性成熟以后，再让所有用户使用。灰度发布可以保证整体系统的稳定，在初始灰度的时候就可以发现、调整问题，以减少其影响。

基于Servicecomb Java Chassis和Spring Cloud Huawei框架注册到ServiceComb引擎的微服务，使用灰度功能只需通过配置下发即可使用。

Servicecomb Java Chassis依赖handler-router和Spring Cloud Huawei依赖spring-cloud-starter-huawei-router实现微服务灰度能力，下发规则遵守如下规范：

```
servicecomb:
  routeRule:
    provider: | #服务名
      - precedence: 2 #优先级
        match:        #匹配策略
          headers:          #header匹配
            region:
              exact: 'providerRegion'

            type:
              exact: gray
        route: #路由规则
          - weight: 100 #权重值
            tags:
              version: 1.0.0

      - precedence: 1
        route:
          - weight: 20
            tags:
              version: 0.0.1
              canaryProperty: group-a
          - weight: 80
            tags:
              version: 0.0.2
```

上述配置的具体含义如下：

- 匹配特定请求由match配置，匹配条件是headers。headers中的字段的匹配支持精准匹配。如果未定义match，则可匹配任何请求。
- 转发权重定义在routeRule.{targetServiceName}.route下，由weight配置，weight数值表示百分数，需要满足加和等于100，不满足100的部分会用最新版本填充。
- 服务分组定义在routeRule.{targetServiceName}.route下，由tags配置，version是特殊的tag，表示微服务版本。还可以配置其他属性，这些属性在实例的属性里面定义。
- 优先级数量越大优先级越高。

Servicecomb Java Chassis依赖darklaunch实现的微服务灰度能力，也是当前ServiceComb引擎页面下发的规则，遵守如下规范：

```
{
  "policyType":"RULE",
  "ruleItems":[
    {
      "groupName":"self_rule_test",
      "groupCondition":"version=0.0.1",
      "policyCondition":"name=11111",
      "versions":["0.0.1"]
     }],
   "empty":false
}
```

上述配置的具体含义如下：

- policyCondition路由规则匹配条件，本条规则匹配的是请求参数为name，值为11111时匹配当前路由规则。
- groupName路由规则名称。
- groupCondition规则的目标分组，本条路由规则在匹配到name=11111的条件下，路由到version=0.0.1的微服务实例。
- 配置项固定为：cse.darklaunch.policy.${serviceName}。

## Spring Cloud Huawei

Spring Cloud Huawei使用灰度发布，需要在项目中增加如下依赖。如果项目中已经直接或者间接引入依赖，无需重复引入。

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

灰度规则依赖的headers参数设置：

- 1.10.7及之后版本header参数需要在配置文件中设置，用户请求当中的header不会透传到下游服务：

  ```
  spring:
    cloud:
      servicecomb:
        context:
          headerContextMapper:
            canary: canary
  ```

- 1.10.7之前的版本设置header参数：请求中设置的header参数进行透传。

## Java Chassis

Java Chassis使用灰度发布，需要在项目中增加如下依赖。如果项目中已经直接或者间接引入依赖，无需重复引入。

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

或者

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

并在配置文件中增加配置项：

```
servicecomb.router.type: router
```

Java Chassis默认不会传递非参数header到各个微服务，如果灰度发布依赖于非参数header，可以增加如下配置项：

```
servicecomb.router.header: canaryHeader1,canaryHeader2
```

Java Chassis会将这些非参数header用于灰度发布的匹配。

如果请求经过Edge Service转发，Edge Service也需要增加灰度发布相关的配置。

# 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。

- Spring Cloud使用仪表盘，不需加入依赖，可直接使用。Spring Cloud包含如**表6-7**所示配置项，指定仪表盘上报地址等信息。

  **表 6-7** Spring Cloud Huawei 常用配置项

| 配置项 | 含义 | 缺省值 |
| --- | --- | --- |
| spring.cloud.servicecomb.dashboard.invocationProviderEnabled | 使用基于请求的接口计数。 | true |
| spring.cloud.servicecomb.dashboard.governanceProviderEnabled | 使用基于熔断器的计数。 | false |
| spring.cloud.servicecomb.dashboard.enabled | 是否开启仪表盘数据上报功能 | false |
| spring.cloud.servicecomb.dashboard.address | 仪表盘数据上报的地址，格式为http://{ip}:{port}，以“,”分隔多个地址。<br>**说明**<br>仪表盘数据上报地址获取方式请参考**获取ServiceComb引擎配置中心地址**，且将端口号改为：30109。 | - |

  **说明**

  其中，基于请求的接口计数和基于熔断器的计数的两种计数机制只能开启一个。

- 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6-8**所示配置项，指定仪表盘上报地址等信息。

**表 6-8** Java Chassis 常用配置项

| 配置项 | 含义 | 缺省值 |
| --- | --- | --- |
| servicecomb.monitor.client.serverUri | 仪表盘数据上报的地址，格式为http://{ip}:{port}，以“,”分隔多个地址。<br>**说明**<br>仪表盘数据上报地址获取方式请参考**获取ServiceComb引擎配置中心地址**，且将端口号改为：30109。 | - |
| servicecomb.monitor.client.enabled | 是否启用数据上报 | true |
| servicecomb.monitor.client.interval | 上报周期（毫秒） | 10000 |

# 6.6 使用安全认证

## 6.6.1 安全认证概述

开启了安全认证的ServiceComb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角色权限的账号创建新账号，根据实际业务需求把合适的角色同账号关联。使用该账号的用户则具有对该ServiceComb引擎的相应的访问和操作权限。

ServiceComb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ServiceComb引擎专享版，在使用安全认证前需要完成以下工作：

1. **创建安全认证账号名和密码**
2. **配置微服务安全认证的账号名和密码**

**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ServiceComb引擎专享版，升级到新版本并开启安全认证的场景，请参考**管理ServiceComb引擎专享版安全认证**。

## 6.6.2 创建安全认证账号名和密码

为开启了安全认证的ServiceComb引擎专享版创建账号名和密码，请参考**系统管理**。

## 6.6.3 配置微服务安全认证的账号名和密码

ServiceComb引擎专享版开启编程接口安全认证后，需要对连接到该引擎的微服务组件开启编程接口安全认证。开启编程接口安全认证是通过配置安全认证账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。

由于账号和密码涉及安全问题，建议加密后使用。

**说明**

若ServiceComb引擎专享版未开启编程接口安全认证，但微服务组件配置了安全认证账号名和密码，引擎会对微服务组件配置的账号进行校验。

### Spring Cloud 微服务组件配置安全认证账号名和密码

- 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6-9**所示环境变量。

  添加环境变量，请参考**管理应用环境变量**。

**表 6-9** 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。<br>**说明**<br>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账号名和密码

- 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6-10**所示环境变量。

  添加环境变量，请参考**管理应用环境变量**。

**表 6-10** 环境变量

| 环境变量 | 说明 |
| --- | --- |
| servicecomb_credentials_rbac_enabled | • true：开启安全认证。<br>• false：不开启安全认证。 |
| 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。<br>**说明**<br>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**。 |

# 7 附录

## 7.1 Java Chassis 版本升级参考

- 使用2.1.3版本之前的Java Chassis接入ServiceComb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。

# 7.2 本地开发工具说明

本地开发工具包含了ServiceComb引擎2.x的本地轻量化版本，提供用于本地开发的轻量服务中心、配置中心，和简单易用的界面。

使用说明请参考本地开发工具压缩包中的README.md文件。

**表 7-1** 本地引擎资源配额限制

| 功能 | 资源 | 最大配额 |
| --- | --- | --- |
| 微服务管理 | 微服务版本数量（个） | 10,000 |
| | 单个微服务实例数量（个） | 100 |
| | 单个微服务契约数量（个） | 500 |
| 配置管理 | 配置数量（个） | 600 |

**表 7-2** 本地轻量化 ServiceComb 引擎版本说明

| 版本 | 对应ServiceComb引擎版本 | 发行时间 | 获取路径 |
| --- | --- | --- | --- |
| 2.1.8 | 2.x | 2023.9.25 | Local-CSE-2.1.8-windows-amd64.zip |
| | | | Local-CSE-2.1.8-linux-amd64.zip |
| | | | Local-CSE-2.1.8-linux-arm64.zip |
| | | | Local-CSE-2.1.8-darwin-amd64.zip |
| | | | Local-CSE-2.1.8-darwin-arm64.zip |

**说明**

- 本地轻量化ServiceComb引擎仅作为本地开发调测，请勿用于商业使用。
- 本地轻量化ServiceComb引擎支持在Windows、Linux和Mac系统下使用。当运行环境为Mac系统时，需要将下载的Mac包放在“Users/xxx”目录下运行，其中xxx为当前登录系统的用户名。

# 7.3 Mesher 使用 ServiceComb 引擎指南

## 7.3.1 Mesher 简介

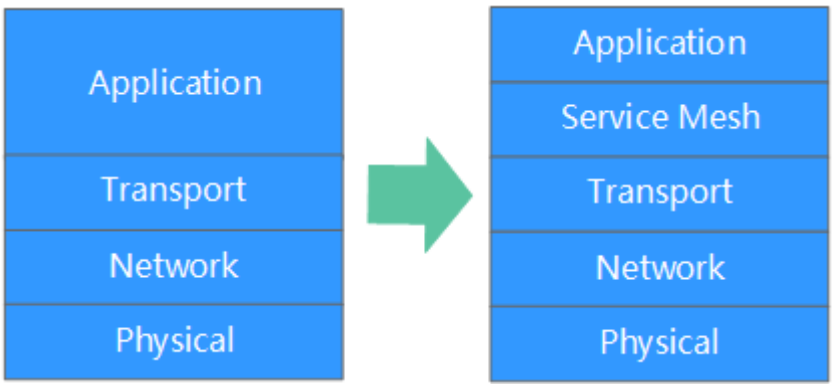
### 什么是 Mesher

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consumer和提供者provider，场景如下：

- 场景一：仅consumer使用Mesher作为Sidecar。

  provider需要自己实现服务注册发现，或者使用Java开发框架，否则通过Mesher接入的consumer无法发现provider。

  应用间的网络请求如下：

  consumer-> Mesher -> provider
- 场景二：consumer、provider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：

  consumer -> Mesher -> Mesher ->provider
- 场景三：仅provider使用Mesher作为Sidecar

  consumer需要使用Java开发框架。

  应用间的网络请求如下：

  consumer-> Mesher -> provider

## 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，consumer在访问provider时使用http://IP:port/进行访问。在使用Mesher后，使用http://provider:port/即可进行访问，**接入说明**将详细讲解。

# 7.3.2 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平台开启多语言接入Mesher服务网格。

本章节介绍http应用如何通过Mesher接入ServiceComb引擎。由于Mesher支持多语言，因此本章仅描述通过Mesher接入ServiceComb引擎时的规范要求。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** 在ServiceStage平台部署组件，绑定ServiceComb引擎，将组件接入ServiceComb引擎，可在“高级设置”中选择已绑定的ServiceComb引擎，单击  ，输入应用进程的监听端口号，开启多语言接入Mesher服务网格，具体操作请参考**创建并部署组件** 。

**说明**

组件部署环境为容器场景时，支持开启多语言接入Mesher服务网格；当部署环境为虚拟机场景时，不支持开启多语言接入Mesher服务网格。

**----结束**

# 7.4 Spring Cloud Huawei 与 Java-chassis 历史版本修复问题

本章列出了在Spring Cloud Huawei与Java-chassis框架历史版本中修复的所有问题。

## Spring Cloud Huawei 历史版本及版本修复问题

| spring-cloud-huawei版本 | 主要修复问题 |
| --- | --- |
| 1.11.6-2023.0.x | • snakeyaml、jackson、guava版本安全漏洞。<br>• 路由开关关闭后，微服务应用启动失败，找不到nacos/servicecomb适配实现类。<br>• 当前服务配置中心配置未变化，发布配置刷新事件，请求过程中可能出现池化配置找不到。 |
| 1.11.6-2022.0.x | |
| 1.11.6-2021.0.x | |

| | |
|---|---|
| 1.11.4-2022.0.x<br>1.11.4-2021.0.x | • RBAC安全认证未开启，框架依然会监听认证过期事件。<br>• gateway/webflux路由无法获取请求头设置信息。<br>• 微服务API安全认证开启后，规则未设置，所有请求不通过。<br>• 服务端熔断规则不生效。 |
| 1.11.3-2022.0.x<br>1.11.3-2021.0.x | 指定服务名情况下，实例隔离策略不生效。 |
| 1.11.2-2022.0.x<br>1.11.2-2021.0.x | • API安全认证未设置黑、白名单策略时，空指针异常。<br>• 服务端、客户端同时设置相同请求头，key不生效。 |
| 1.11.0-2022.0.x<br>1.11.0-2021.0.x | trace上下文配置基于动态配置不生效。 |
| 1.10.13-2021.0.x | 同时调用多个服务情况下，降级不生效。 |
| 1.10.11-2021.0.x | 实例隔离治理不生效。 |
| 1.10.9-2021.0.x | • 指定服务名设置重试策略不生效。<br>• 服务降级错误返回“null”字符串，修改为返回null。 |
| 1.10.8-2021.0.x<br>1.10.8-2020.0.x | 负载均衡规则不生效。 |
| 1.10.7-2021.0.x<br>1.10.7-2020.0.x | 服务注册发现开关关闭，启动失败。 |
| 1.10.6-2021.0.x<br>1.10.6-2020.0.x | 监控信息中缺少环境信息。 |
| 1.10.5-2021.0.x<br>1.10.5-2020.0.x | 重试次数太多导致请求长时间无响应。 |
| 1.10.4-2021.0.x<br>1.10.4-2020.0.x | identifierRateLimiting限流上下文获取失败。 |
| 1.10.3-2021.0.x<br>1.10.3-2020.0.x | 治理配置第一次变化时不生效。 |
| 1.10.2-2021.0.x<br>1.10.2-2020.0.x | • 调整默认配置刷新时间为15s。<br>• 实例隔离过滤器空指针异常。 |

| | |
|---|---|
| 1.10.1-2021.0.x<br>1.10.1-2020.0.x | • 非客户端请求上下文空指针异常。<br>• 路由客户端编译请求头失败。<br>• 灰度版本策略生效。<br>• ClientRequest非RequestData类型下转化异常。 |
| 1.10.0-2021.0.x<br>1.10.0-2020.0.x | • 服务删除、重启无法刷新ribbon缓存，导致请求到不可用服务，报no host to route。<br>• 动态配置下灰度发布配置规则，修改配置规则不生效。<br>• 启动类application在业务包外层启动失败。<br>• 网关最大重试次数不生效。 |
| 1.9.1-2020.0.x | • 某些场景下，启动类ags属性加载不正确。<br>• 网关配置最大重试次数无限重试。<br>• 灰度动态配置不生效。 |
| 1.9.0-2020.0.x | instance.healthCheck.mode值为pull，自定义配置healthCheckInterval健康检查时间不生效。 |
| 1.8.0-2020.0.x | • 非long polling模式下大量配置查询任务，触发查询任务无间隔时间。<br>• cse操作页面对服务实例下线后，服务实例调用依然正常。<br>• management.server.port和server.port两个端口不一致，启动报错。<br>• gateway查询到不同环境的实例。 |
| 1.7.0-2020.0.x | • webmvc与路由结合导致gateway启动失败。<br>• gateway无法实现基于服务发现的路由定义功能。<br>• gateway无法跨应用服务发现。 |
| 1.6.1-2020.0.x | **说明**<br>存在重大问题，不建议使用：<br>• 非常频繁地查询配置中心。<br>• 查询到错误的配置。 |
| 1.9.4-Hoxton | 当前服务配置中心配置未变化，发布配置刷新事件，请求过程中可能出现池化配置找不到。 |
| 1.9.3-Hoxton | 服务删除、重启无法刷新ribbon缓存，导致请求到不可用服务，路由不到可用服务。 |
| 1.9.2-Hoxton | 服务端删除实例后重新注册实例，客户端选择错误的服务端实例。 |
| 1.9.1-Hoxton | • 某些场景下，启动类ags属性加载不正确。<br>• 网关配置最大重试次数无限重试。<br>• 灰度动态配置不生效。 |

| | |
|---|---|
| 1.9.0-Hoxton | instance.healthCheck.mode值为pull，自定义配置healthCheckInterval健康检查时间不生效。 |
| 1.8.0-Hoxton | • 非long polling模式下大量配置查询任务，触发查询任务无间隔时间。<br>• cse操作页面对服务实例下线后，服务实例调用依然正常。<br>• management.server.port和server.port两个端口不一致启动报错。<br>• gateway查询到不同环境的实例。 |
| 1.7.0-Hoxton | • webmvc与路由结合导致gateway启动失败。<br>• gateway无法实现基于服务发现的路由定义功能。<br>• gateway无法跨应用服务发现。 |
| 1.6.0-Hoxton | • SDK针对401、403错误码，对引擎做出重试请求。<br>• gateway默认路由规则不生效。<br>• 无法跨应用调用。 |
| 1.5.9-Hoxton | • RBAC鉴权功能不生效。<br>• 灰度路由时无法获取到微服务最新版本。<br>• 某些JDK版本不支持swagger循环依赖。 |
| 1.5.8-Hoxton | • gateway服务发现中选择错误实例。<br>• 实例状态为空时调用异常。 |
| 1.5.6-Hoxton | • AK/SK配置后不生效，鉴权失败。<br>• server.env配置不生效，全部为空。<br>• 配置中心治理配置项删除后仍然能使用。<br>• 滑动窗口熔断配置属性不生效。<br>• 注册中心开启watch模式后，空指针异常。<br>• 读取环境变量PAAS_CSE_SC_ENDPOINT时只能读取第一个地址，无法读取第二个。 |
| 1.5.0-Hoxton | • 错误的治理规则导致空指针异常。<br>• AK/SK未配置时启动，报空指针异常。<br>• 服务启动后首次并发请求，限流策略不生效。<br>• governance治理选择错误服务导致请求异常。<br>• 当环境配置为production时，契约不变，服务重启会失败。 |
| 1.6.4-Greenwich | 当前服务配置中心配置未变化，发布配置刷新事件，请求过程中可能出现池化配置找不到。 |
| 1.6.3-Greenwich | 服务删除、重启无法刷新ribbon缓存，导致请求到不可用服务，路由不到可用服务。 |
| 1.6.1-Greenwich | gateway跨应用服务发现异常。 |

| | |
|---|---|
| 1.6.0-Greenwich | • SDK针对401、403错误码，对引擎做出重试请求。<br>• gateway路由规则不生效。<br>• 灰度路由时无法获取到微服务最新版本。<br>• AK/SK配置不生效。<br>• server.env配置不生效，全部为空。<br>• Servicecenter在watch模式下启动失败。<br>• 读取环境变量PAAS_CSE_SC_ENDPOINT时只能读取第一个地址，无法读取第二个。<br>• 不支持跨应用调用。 |
| 1.5.0-Greenwich | • 错误的治理规则导致空指针异常。<br>• AK/SK未配置时空指针异常。<br>• 服务启动后首次并发请求，限流策略不生效。<br>• governance服务转发错误。<br>• 环境是production时，重复注册契约导致启动失败。 |
| v1.3.3-Greenwich | 注册中心监听不生效。 |
| 1.6.1-Finchley | • 服务启动后第一次并发，调用随机失败。<br>• 微服务跨应用调用失败。 |
| 1.6.0-Finchley | • SDK针对401、403错误码，对引擎做出重试请求。<br>• gateway路由规则不生效。<br>• 灰度路由时无法获取到微服务最新版本。<br>• AK/SK配置不生效。<br>• server.env配置不生效，全部为空。<br>• Servicecenter在watch模式下启动失败。<br>• 读取环境变量PAAS_CSE_SC_ENDPOINT时只能读取第一个地址，无法读取第二个。<br>• 不支持跨应用调用。 |
| 1.5.1-Finchley | 配置中心治理配置删除后仍然可以使用。 |
| v1.3.9 | **说明**<br>存在重大问题，不推荐使用。<br>governance存在服务转发严重错误。 |
| v1.3.8 | **说明**<br>存在重大问题，不推荐使用。<br>governance存在服务转发严重错误。 |
| v1.3.4 | • 注册线程池无法正确关闭并导致泄漏。<br>• actuator开启后微服务注册失败。<br>• 某些场景心跳次数过多。 |

| | |
|---|---|
| v1.3.3 | • websocket在wss协议请求失败。<br>• 注册中心watch不生效。 |
| v1.3.2 | • 当环境设置为production时契约注册失败。<br>• 注册中心url未设置时随机选择地址失败。<br>• 注册中心配置域名时watch异常。 |
| v1.2.0 | 从ServiceStage读取默认的ak/sk配置时，初始化大量对象，导致内存泄漏。 |
| v1.1.0 | • heartbeat信息日志过多。<br>• 微服务间SSL调用不生效。<br>• url包含空格时请求异常。 |
| v1.0.0 | 某些场景无法自动服务发现。 |
| v0.0.3 | • 服务发现down状态实例。<br>• 配置的路径过长。<br>• 微服务连接本地CSE引擎失败。 |

## Java Chassis 历史版本及版本修复问题

| java-chassis版本 | 主要修复问题 |
|---|---|
| 3.1.0 | • 扫描@RestController注解类报UnsupportedOperationException异常。<br>• idleTimeout未正确设置带来的一些connection closed问题。 |
| 3.0.2 | • 配置中心存在多条相同key时，更新配置不生效。<br>• 节点信息未正确销毁导致请求异常。 |
| 3.0.1 | 开始支持fallback标签路由。 |
| 2.8.16 | idleTimeout未正确设置带来的一些connection closed问题。 |
| 2.8.15 | 配置中心存在多条相同key时，更新配置不生效。 |
| 2.8.14 | • 请求上下文丢失。<br>• SCBEngine关闭异常。<br>• 当内容为空时buffer reader不正确。 |
| 2.8.13 | • 连接超时，metrics信息统计不正确。<br>• 上传附件为空时SDK异常。 |
| 2.8.12 | • 注册中心频繁变更微服务，导致新部署实例无法找到，仍然使用缓存版本实例。<br>• vertx偶尔关闭异常，阻塞服务正常关闭。 |

| | |
|---|---|
| 2.8.11 | DefaultHttpClientFilter反序列化响应body失败时，异常信息不正确。 |
| 2.8.10 | • ResponseEntity设置的Content-Type不生效。<br>• 注册中心频繁变更微服务，导致新部署实例无法找到，仍然使用缓存版本实例。 |
| 2.8.9 | • 服务启动注册类bean初始化2次问题。<br>• 配置兼容1.x引擎注册中心配置。<br>• 并发注册时，缓存实例信息不刷新。 |
| 2.8.8 | • 微服务路由应从实例中获取属性以决定路由分发策略。<br>• HttpUtils解析Content-Disposition错误导致ReadStreamPart无法获取到文件名。 |
| 2.8.7 | 可用az默认配置为null。 |
| 2.8.6 | • 调整HTTP客户端keep-alive配置项。<br>• 并发访问时获取微服务版本号。 |
| 2.8.5 | 连续错误次数为0或小于0时，错误百分比无法生效。 |
| 2.8.4 | 实例改变时间异常导致请求错误。 |
| 2.8.3 | 路由规则中caseInsensitive定义。 |
| 2.8.2 | • 治理规则生效匹配中，apiPath比较匹配存在问题。<br>• mvc请求中body参数为null时异常。 |
| 2.8.1 | 负载隔离开关未生效。 |
| 2.8.0 | • idleTimeout和keepAliveTimeout未正确设置。<br>• edge网关中highway协议时间解析错误。<br>• 限流治理规则空指针异常。 |
| 2.7.10 | • 仪表盘环境变量上报不正确。<br>• 注册不可用时，已缓存服务被清除导致请求异常的。 |
| 2.7.9 | autoDiscovery开关打开时空指针异常。 |
| 2.7.8 | • 元数据内存泄漏。<br>• findByContext方法导致内存溢出。<br>• PriorityInstancePropertyDiscoveryFilter输出debug日志。 |
| 2.7.6 | • ProducerOperationHandler错误日志打印traceID。<br>• 实例隔离过滤器未生效。<br>• 当路由规则版本信息为空时出现空指针异常。 |
| 2.7.5 | • 每次拉取实例发布事件而导致过多事件排队。<br>• zipkin某些场景Span为空。<br>• kie配置中心不支持filesource逗号分割。 |

| | |
|---|---|
| 2.7.3 | • 灰度发布动态修改配置无效。<br>• 请求上下文合并逻辑冲突。 |
| 2.7.0 | IsolationDiscoveryFilter创建过多对象。 |
| 2.6.3 | • 负载均衡选择服务异常时重试异常。<br>• 重试次数计算不生效。 |
| 2.6.0 | • 文件下载时，part内容为空导致空指针异常。<br>• highway协议空字符串系列化。<br>• kie在非long pulling模式下过快拉取配置。<br>• 新引擎创建后首次部署sdk报空指针异常。<br>• 增加highway协议，增加默认响应体大小限制20M。 |
| 2.5.2 | servicecomb客户端连接idleTimeout默认时间从60s调整为30s。 |
| 2.5.1 | kie第一次拉取配置为空时抛异常。 |
| 2.5.0 | 在vert.x工作池中RPC调用失败。 |
| 2.3.0<br>2.2.4<br>2.2.3<br>2.2.2<br>2.2.1 | • 配置中心配置更新，流控组件异常。<br>• edge网关服务添加自定义executor失败。<br>• AK/SK开关打开，第一次获取加密类失败。<br>• 配置不支持占位符配置。<br>• 熔断配置slidingWindowType不生效。<br>• 转发请求到不存在的微服务时出现OOM。<br>• 重试错误地址出现404异常。<br>• 注册中心监听实例失败。<br>• XmlViewResolver和SPI两种方式同时注入某个bean类出现死锁。<br>• 配置变更后灰度路由未使用到最新版本。<br>• 配置filesource时触发配置变化事件不正确。<br>• 注册中心地址支持数组占位符配置。<br>• 开启仪表盘功能时，空指针异常导致启动失败。 |
| 2.2.0<br>2.1.6 | • 治理配置未正确处理空配置和打印错误操作。<br>• RestTemplate请求上传多个文件出现随机失败。<br>• deleteAfterSuccess为true时servlet上下文中的下载文件删除失败。 |

| | |
|---|---|
| 2.1.5 | • main方法不存在包名时启动异常。<br>• 第三方服务注册启动时异常。<br>• http2请求设置响应头信息异常。<br>• yaml不安全解析。<br>• 解析集合占位符。<br>• APP_MAPPING解析servicestage环境变量失败。 |
| 2.1.3 | • 被隔离实例尝试请求正确返回后未正常恢复。<br>• 超时异常连接关闭触发异步回调。<br>• 服务配置和spring value默认值不生效。<br>• servicecomb.references.version-rule未生效。<br>• 超时异常时服务隔离未生效。 |
| 2.1.2 | • 接口未变化，JDK版本不同，契约读取顺序不一致导致契约校验失败。<br>• 服务状态不正确导致隔离实例无法被尝试调用。<br>• 网关读取微服务名称抛空指针异常。 |
| 2.1.1 | • 原生参数为空时highway协议解析抛空指针异常。<br>• 当连接超时时获取连接时间为0。<br>• 无法正确处理yaml和properties配置。<br>• RegistryUtils API不使用版本规则参数。 |
| 2.1.0 | • 设置实例信息持续失败导致OOM。<br>• 微服务元数据中客户端标签错误。<br>• 多实例在不同版本和api时服务调用异常。<br>• highway协议中带appId请求异常。 |
| 2.0.2 | • jackson转化会覆盖所有对象。<br>• kie解析配置错误。<br>• kie增加版本号查询配置。 |
| 2.0.1 | • Http2中idleTimeoutInSeconds未使用。<br>• 同时使用@RequestHeader(value ="xxx")和聚合参数导致空指针异常。<br>• 接口参数定义为对象时请求异常。 |
| 1.3.11 | • HTTP客户端keep-alive配置项。<br>• 连续错误次数为0或小于0时，错误百分比无法生效。 |
| 1.3.10 | • 负载隔离开关未生效。<br>• http2请求设置响应头信息异常。 |
| 1.3.9 | 限流治理规则空指针异常。 |
| 1.3.8 | 实例隔离过滤器不生效。 |

| | |
|---|---|
| 1.3.7 | IsolationDiscoveryFilter创建过多对象。 |
| 1.3.5 | • 请求客户端idleTimeout未生效。<br>• 服务器挂起而且没有任何堆栈信息。 |
| 1.3.3 | • yaml不安全解析。<br>• 被隔离实例尝试请求正确返回后未正常恢复。 |
| 1.3.2 | |
| 1.3.1 | • 接口未变化，JDK版本不同，契约读取顺序不一致导致契约校验失败。<br>• 同时使用@RequestHeader(value ="xxx")和聚合参数导致空指针异常。<br>• 服务状态不正确导致隔离实例无法被尝试调用。<br>• 实例过多时CPU负载过高。<br>• 文件上传异常导致无法返回。<br>• 注册中心异常重新注册时默认状态重置。<br>• SwaggerProducerOperation打印敏感信息日志。 |
| 1.3.0 | • FallbackPolicy无异常信息返回。<br>• CseAsyncRestTemplate不支持设置header。<br>• RSAProviderTokenManager导致内存泄漏。<br>• ServiceCombServerStats.getFailedRate抛ArithmeticException: /by zero。<br>• 存在重写方法时，swagger生成契约异常。<br>• RPC调用方法多个参数且第一个参数是Object时，第二个参数为空的。 |
| 1.2.0 | • 不支持@ConfigurationProperties注解。<br>• http2不支持文件下载。<br>• 当存在客户端acceptType时文件下载异常。<br>• 启动失败时，销毁方法将原始异常信息覆盖。<br>• metrics-prometheus适配prometheus-2.x出错。<br>• 框架计算CPU使用率不准确。<br>• 契约为空时异常。<br>• 基于ServiceComb开发的springMVC应用启动失败。<br>• rest调用异常时收集信息错误。<br>• 当返回类型定义为ResponseEntity<Void>时报空指针异常。<br>• AZ亲和在空实例时异常。<br>• 线程队列占满时拒绝请求策略。<br>• httpClientResponse.exceptionHandler抛异常。 |

| | |
|---|---|
| 1.1.0 | ● 启用延迟错误注入后业务线程死锁。<br>● 访问最新服务异常。<br>● 由于不可用服务清理未及时导致请求异常。<br>● edge服务未正确返回servlet的响应码。<br>● 解决ubuntu中使用mvn install命令打包。<br>● BeanParamAnnotationProcessor生成的参数顺序不稳定。<br>● 回调场景使用老版本实例属性导致请求错误。<br>● 下载内容不正确。<br>● @ApiResponse注解丢失响应类型。<br>● 请求参数编码和解码不正确。<br>● json解析错误返回码错误。<br>● servicecomb.xx.xx作为治理规则时不生效。<br>● 请求path中包含非法字符时请求连接挂起。<br>● 内部类解析抛NotFoundException异常。 |
| 1.0.0 | ● 限流策略不生效。<br>● 同时设置以cse和servicecomb开头的配置导致无法获取配置值。<br>● RPC请求void方法时异常。<br>● 客户端请求完一个未注册服务后，即使后面这个服务重新注册了，请求还是找不到服务端。<br>● 查询未注册服务空指针。 |
| ● 1.0.0-m2 | ● 使用zuul做网关进行路由，当同一个微服务接口同时开放rest和highway方式时，出现客户端无法调用。<br>● 当服务名或契约id中包含“.”时，qps handler异常。<br>● 响应加权负载策略初始化和无状态访问。<br>● 不支持中文名称的文件下载。<br>● 服务包含环境变量时重新注册异常。<br>● producer实现类没有实现方法时抛空指针异常。<br>● 上传内容为空时抛空指针异常。<br>● 当上传文件大小超过服务端限制时，客户端返回一个错误的响应。<br>● 服务端返回Transfer-Encoding header时edge网关抛异常。<br>● 并发请求时存在服务并发发现的问题。<br>● edge网关服务覆盖所有错误码统一返回502错误码。<br>● 某些场景优雅下线不生效。 |

| | |
|---|---|
| 0.5.0 | ● RSAConsumerTokenManager创建鉴权token。<br>● 服务下线时，未从注册中心取消注册。<br>● 服务自动发现不生效。<br>● edge网关服务未正确处理404请求。 |
| 0.4.0 | ● servlet请求超时时，同时引发不支持async调用或空指针异常。<br>● 调用出现异常时无法获取响应。<br>● rest producer返回错误类型。 |
| 0.2.0 | ● 在重试线程中无法重新注册。<br>● 契约注册失败无法从日志反映异常信息。<br>● 服务启动是，注册报空指针异常。 |