Cloud Service Engine

Development Guide

Issue	01
Date	2024-05-16

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 Overview

1.1 Development Introduction

Overview

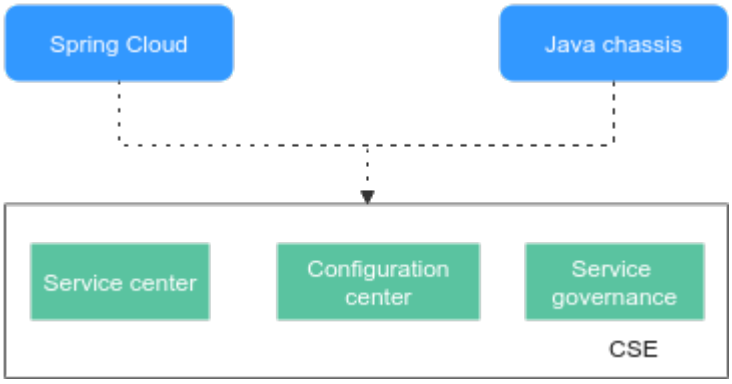
A stable and reliable microservice running environment is crucial as the microservice architecture has become the first option for developers to build applications.

Cloud Service Engine (CSE) is a one-stop management platform provided by ServiceStage for microservice solutions. It enables developers to focus on service development and improve product delivery efficiency and quality. The microservice architecture consists of the following:

- Remote Procedure Call (RPC) communication between microservices. The microservice architecture requires that microservices communicate with each other through RPC instead of other traditional communication modes, such as shared memory and pipes. Common communication protocols include REST (HTTP+JSON), gRPC (HTTP2+protobuffer) and Web Service (HTTP+SOAP). Using RPC for communication reduces coupling between microservices and makes the system more open with less technological restriction. You are advised to use standard protocols in the industry, such as REST. Proprietary protocols can also be used in scenarios requiring high performance.
- Distributed microservice instances and service discovery. The microservice architecture is highly elastic and needs to support multi-instance deployment of microservices to handle the dynamic service traffic. The microservice design is generally stateless. Increasing stateless microservice instances lets you improve processing performance. When there are a large number of instances, a middleware that supports service registry and discovery is required for microservice calling and addressing.
- Dynamic and centralized configuration management. The configuration of microservice management is increasingly complex as the number of microservices and instances increases. The configuration management middleware provides a unified view for all microservices, simplifying the configuration management of microservices. Such middleware works with the governance console to adjust microservice at microservice runtime to handle changing service scenarios without application upgrade.

- Microservice governance capabilities, such as circuit breaker, fault tolerance, rate limiting, load balancing, and service degradation. These governance capabilities can mitigate the impact of some common faults of the microservice architecture on the services.
- Tracing and centralized log collection and retrieval. Viewing logs remains the most commonly used method for analyzing system faults. Tracing information helps locate faults and analyze performance bottlenecks.

The microservice architecture has been implemented on many open-source frameworks, such as **Spring Cloud**, **Apache ServiceComb Java chassis** (Java chassis for short). ServiceComb engines support the access of these open-source microservice frameworks and use functions such as registry, discovery, centralized configuration, and service governance. The following figure shows the relationship.



You can use Spring Cloud and Java chassis microservice development frameworks to access the ServiceComb engine to obtain the best development experience and technical support. Using other development frameworks, such as Mesher to access the ServiceComb engine depends on the technical support of the open-source community.

This topic focuses on the development guide of Spring Cloud and Java chassis. Microservice applications developed using other frameworks such as Mesher use ServiceComb engine. See **Using ServiceComb Engines by Mesher**.

Development Capability Requirements

This document describes how the open-source microservice development frameworks are connected to a ServiceComb engine and use its functions. Assume that you have the following development capabilities:

- Using Java to develop microservices. You have developed an application based on a microservice development framework supported by ServiceStage and want to host the application on the ServiceComb engine. This document provides technical support for connecting microservice applications to the ServiceComb engine. This document does not describe how to use the open-source microservice development frameworks. You can obtain the basic materials and development guides of these frameworks in relevant open-source communities.
- Understanding the functions of the registry center and configuration center in microservice applications, and building and using the registry center in projects. Different microservice development frameworks support different

open-source registry centers by default. Therefore, understanding the functions of a registry center helps you change registry centers at ease.

- You are familiar with application deployment. For details, see **Creating and Deploying a Component**.

1.2 Related Concepts

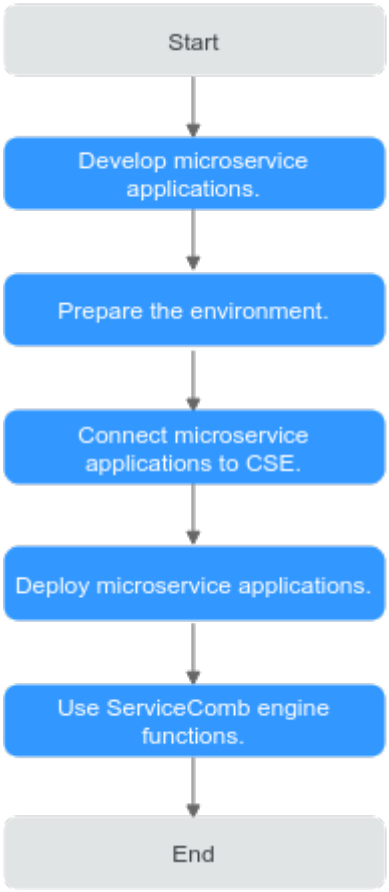
- Application: a software system that implements a complete service. An application consists of multiple microservices, which can discover and call each other.
- Microservice: a software system that implements a specific service function. Microservices are independently developed and deployed.
- Microservice instance: An instance is generated when a microservice is deployed in the runtime environment using the deployment system. An instance can be considered a process, and multiple instances can be deployed for a microservice.
- Microservice environment: a logical concept established by the service center, which can be development or production. Microservice instances in different environments are logically isolated and cannot be discovered or called by each other.

1.3 Development Process

Overview

Figure 1-1 shows how to develop an application and use a ServiceComb engine.

Figure 1-1 Development process



Description

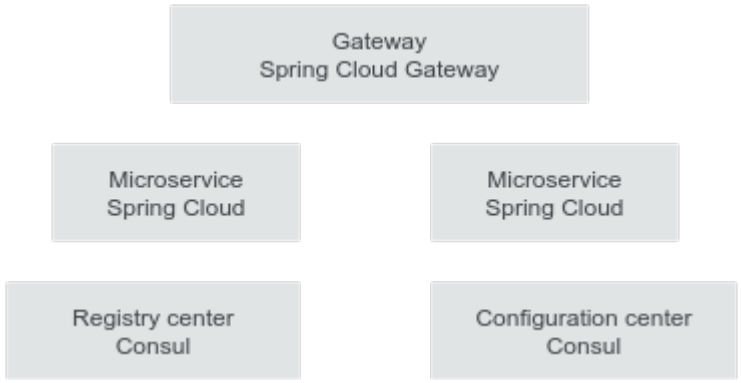
1. Developing Microservice Applications

   If you have developed a microservice application, skip this step and **prepare the environment**.

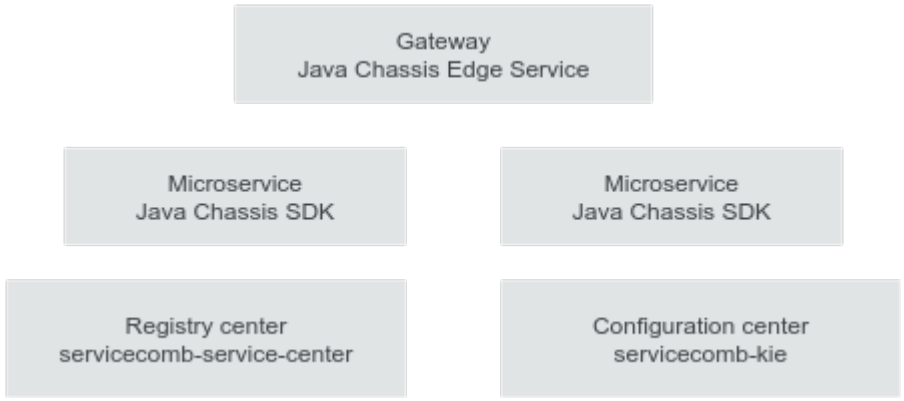
   Before developing a microservice application, you need to select a technology. To select an appropriate technology, technical decision makers need to consider whether team members can master the technology, and whether the technology can deliver the desired functions, performance, and reliability of the project. Many other factors, such as commercial services, should also be taken into account. This document does not discuss technology selection. Assume that the technical team has selected a proper development framework. Most technical teams build their services using open-source frameworks.

   For details about how to develop microservice applications, see **Developing Microservice Applications**.

   - For Spring Cloud, the following technology is used for local microservice development:



- For Java chassis, the following technology is used for local microservice development:



2. Preparing the Environment

   Create a cloud environment to support the ServiceComb engine connection test, cloud-based application deployment, and ServiceComb engine functions. Generally, a test environment and a production environment are created. ServiceStage facilitates cloud environment management. For details, see **Preparing the Environment**.

3. Connecting Microservice Applications to CSE

   Microservice applications are connected to the ServiceComb engine. To perform this step, you need to modify the configuration files and build scripts of developed applications. After the modification, recompile and package the applications and deploy the application package on the ServiceComb engine using ServiceStage. For details, see **Connecting Microservice Applications**.

4. Deploying Microservice Applications

   Deploy the developed microservice applications to the ServiceComb engine using ServiceStage. For details, see **Deploying Microservice Applications**.

5. Using ServiceComb Engine Functions

   An evolving application requires continuous improvement and iteration. In each iteration, microservice applications may need to be upgraded, requiring more ServiceComb engine functions. The preceding application development, compilation, packaging, and deployment will repeat during function iteration. For details, see **Using ServiceComb Engine Functions**.

1.4 Development Specifications

Development Language

Using Java to develop microservices.

Requirements for Microservice Development Framework of a ServiceComb Engine

The following table lists the recommended versions of the microservice development framework.

- If you have used the microservice development framework of an earlier version to build applications, you are advised to upgrade it to the recommended version to obtain the stable and rich function experience.
- If an application has been developed using the Spring Cloud microservice development framework, you are advised to use **Spring Cloud Huawei** to access the application.
- If new microservice applications are developed based on open source and industry ecosystem components, you are advised to use the Spring Cloud framework.
- If you want to use the out-of-the-box governance capability and high-performance RPC framework provided by ServiceComb engines, you are advised to use the Java chassis framework.

Framework	Recommended Versions	Description
Spring Cloud Huawei	1.10.9-2021.0.x or later	Uses **Spring Cloud Huawei** for connection. • Spring Cloud version 2021.0.5 • Spring Boot 2.6.13 Version description of the Spring Cloud microservice development framework: **https://github.com/huaweicloud/spring-cloud-huawei/releases**
Java Chassis	2.7.10 or later	Uses the software package provided by the open-source project for connection without introducing third-party software packages. Version description of the Java chassis microservice development framework: **https://github.com/apache/servicecomb-java-chassis/releases**.

NOTICE

During system upgrade and reconstruction, third-party software conflict is the most common issue. Traditional software compatibility management policies do not adapt to software development for fast software iteration. In this case, see **Third-Party Software Version Management Policy** for version compatibility.

2 Developing Microservice Applications

- If you have developed a microservice application, skip this section.

  The open-source community provides development documents and help channels to help you use the microservice development framework. For details about microservice application development in a specific microservice framework, see the reference documents provided in this section.

  The recommended ServiceComb engine samples offer you quick connection to the engine. Download the samples, modify the ServiceComb engine address and AK/SK information in the configuration file, and run the examples locally. These samples can be registered with the ServiceComb engine.

  – Spring Cloud

    Source code repository: **https://github.com/spring-cloud**

    Issues: For details, see the issues in each code repository of the source code repository.

    Developer guide: **https://spring.io/projects/spring-cloud**

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

    Recommended ServiceComb engine samples: **https://github.com/huaweicloud/spring-cloud-huawei-samples/tree/master/basic**

  – Java Chassis

    Source code repository: **https://github.com/apache/servicecomb-java-chassis**

    Issues: **https://github.com/apache/servicecomb-java-chassis/issues**

    Developer guide: **https://servicecomb.apache.org/references/java-chassis/en_US/**

    Recommended ServiceComb engine samples: **https://github.com/apache/servicecomb-samples/tree/master/basic**

3 Preparing the Environment

You need to prepare the local development and commissioning environment and cloud environment.

Preparing a Local Development and Commissioning Environment

The local development and commissioning environment is used to set up a simple test environment. The options are as follows:

- **Download the local CSE**.
- Use the exclusive ServiceComb engine and open the IP address for public network access to ensure that the local environment can be accessed.

Preparing the Cloud Environment

Before deploying microservice applications on the cloud, you need to prepare the cloud environment. Perform the following procedure to prepare the environment:

- Create a ServiceComb engine. For details, see **Creating a ServiceComb Engine**.
- Create an environment. For details, see **Creating an Environment**. The created environment must contain resources such as CCE clusters, load balancers, and ServiceComb engines.
- Create an application. For details, see **Creating an Application**.

Common Environment Variables

Using ServiceStage to manage environments and deploy applications simplifies user configuration. ServiceStage sets some environment variables for applications. The following table lists some common environment variables:

Table 3-1 Common environment variables

Name	Description
PAAS_CSE_SC_ENDPOINT	Registry center address of a ServiceComb engine.

Name	Description
PAAS_CSE_CC_ENDPOINT	Configuration center address of a ServiceComb engine.
PAAS_PROJECT_NAME	Name of a project.
CAS_APPLICATION_NAME	Name of a ServiceStage application.
CAS_COMPONENT_NAME	Name of a ServiceStage component.
CAS_INSTANCE_VERSION	Version of the deployed ServiceStage.

You can use these variables based on the mechanisms of different microservice development frameworks, such as the Place Holder mechanism of Spring Cloud and the **mapping.yaml** mechanism of Java chassis, to reduce manual input during deployment.

When creating an application on ServiceStage, you can bind middleware, such as Distributed Cache Service (DCS) and Relational Database Service (RDS), to the application. You can obtain the configuration information about the middleware bound to applications by using the following environment variables.

- Distributed session

  Distributed sessions are stable and reliable session storage based on DCS, supporting automatic injection for mainstream web containers, such as tomcat context, node.js express-session, and PHP session handler.

  The following table describes the environment variables of distributed sessions.

  Table 3-2 Environment variables of DCS sessions

Name	Description
DISTRIBUTED_SESSION_CLUSTER	Whether the instance is in cluster mode. Value: **true** or **false**.
DISTRIBUTED_SESSION_TYPE	Storage type of a distributed session instance. Currently, only Redis is supported.
DISTRIBUTED_SESSION_VERSION	Version of a distributed session instance.
DISTRIBUTED_SESSION_NAME	Name of a distributed session instance.
DISTRIBUTED_SESSION_HOST	IP address for connecting to a distributed session instance.

Name	Description
DISTRIBUTED_SESSION_PORT	Port for connecting to a distributed session instance.
DISTRIBUTED_SESSION_PASSWORD	Password for connecting to a distributed session instance.

- Distributed cache

  DCS is an online, distributed, in-memory cache service compatible with Redis and Memcached. It combines high reliability and scalability with instant availability and easy management, delivering high read/write performance and fast data access.

  The following table describes the environment variables of DCS.

  Table 3-3 Environment variables of DCS

Name	Description
DISTRIBUTED_CACHE_CLUSTER	Whether the instance is in cluster mode. Value: **true** or **false**.
DISTRIBUTED_CACHE_TYPE	Storage type of a distributed cache instance. Currently, only Redis is supported.
DISTRIBUTED_CACHE_VERSION	Version of a DCS instance.
DISTRIBUTED_CACHE_NAME	Name of a DCS instance.
DISTRIBUTED_CACHE_HOST	IP address for connecting to a DCS instance.
DISTRIBUTED_CACHE_PORT	Port for connecting to a DCS instance.
DISTRIBUTED_CACHE_PASSWORD	Password for connecting to a DCS instance.

- Cloud database

  RDS for MySQL is a cloud-based web service that is reliable, scalable, easy to manage, and out of the box.

  The following table describes the environment variables of RDS.

  Table 3-4 Environment variables of RDS

Name	Description
RELATIONAL_DATABASE_NAME	Name of an RDS instance.

Name	Description
RELATIONAL_DATABASE_CONNECTION_TYPE	Connection type of an RDS instance. Value: **JNDI/ SPRING_CLOUD_CONNECTOR**.
RELATIONAL_DATABASE_JNDI_NAME	JNDI name of an RDS instance. This variable is used if the connection type is JNDI.
RELATIONAL_DATABASE_DB_NAME	Database name of an RDS instance.
RELATIONAL_DATABASE_DB_USER	Database user of an RDS instance.
RELATIONAL_DATABASE_DB_TYPE	Database type of an RDS instance. Currently, only MySQL is supported.
RELATIONAL_DATABASE_VERSION	Database version of an RDS instance.
RELATIONAL_DATABASE_HOST	Database IP address of an RDS instance.
RELATIONAL_DATABASE_PORT	Database port of an RDS instance.
RELATIONAL_DATABASE_PASSWORD	Database password of an RDS instance.

4 Connecting Microservice Applications

4.1 Connecting Spring Cloud Applications to ServiceComb Engines

This section describes how to connect Spring Cloud applications to ServiceComb engines and use the most common functions of ServiceComb engines. For details about the development guide, see **Using ServiceComb Engine Functions**.

In the **Spring Cloud Huawei Samples** project, you can find the code corresponding to the development methods in this section.

NOTE

Spring Cloud needs to use Spring Cloud Huawei to connect to ServiceComb engines. This document describes how to integrate and use Spring Cloud Huawei in Spring Cloud.

Prerequisites

- Microservice applications have been developed based on Spring Cloud.

  For details about microservice application development in the Spring Cloud microservice framework, see **https://spring.io/projects/spring-cloud**.
- Version requirements. See **Requirements for Microservice Development Framework of a ServiceComb Engine**.
- This document assumes that you use Maven for dependency management and packaging in your project. You are familiar with the Maven dependency management mechanism and are able to modify the **dependency management** and **dependency** in the **pom.xml** file.

Procedure

Step 1 Add dependencies to the **pom.xml** file of the project.

- If you develop microservices using Spring Cloud, introduce the following dependencies:

```
<dependency>
  <groupId>com.huaweicloud</groupId>
```

```
  <artifactId>spring-cloud-starter-huawei-service-engine</artifactId>
</dependency>
```

NOTE

The spring-cloud-starter-huawei-service-engine module consists of the following dependent modules:

```
<!-- Registry and discovery module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-discovery</artifactId>
</dependency>
<!-- Configuration center module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-config</artifactId>
</dependency>
<!-- Service governance module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-governance</artifactId>
</dependency>
<!-- Dark launch module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-router</artifactId>
</dependency>
```

- If you develop the gateway using Spring Cloud, introduce the following dependencies:

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


NOTE

The spring-cloud-starter-huawei-service-engine-gateway module consists of the following dependent modules:

```
<!-- Registry and discovery module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-discovery</artifactId>
</dependency>
<!-- Configuration center module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-config</artifactId>
</dependency>
<!-- Service governance module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-governance</artifactId>
</dependency>
<!-- Dark launch module -->
<dependency>
  <groupId>com.huaweicloud</groupId>
  <artifactId>spring-cloud-starter-huawei-router</artifactId>
</dependency>
<!-- Gateway module -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

You are advised to use Maven Dependency Management to manage the third-party software dependencies of a project. Introduce the following dependencies to the project:

```
<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>
```


Skip the operation if your project already contains the preceding dependencies.

If other registry and discovery libraries, such as Eureka, are used in your project, you need to adjust the project as follows:

- Delete the dependencies related to Eureka from the project. For example:
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```
- If **@EnableEurekaServer** is used in the code, delete it and replace it with **@EnableDiscoveryClient**.

  NOTE

  The spring-cloud-starter-huawei-service-engine component provides functions such as service registration, configuration center, service governance, dark launch, and contract management. Contract management is not mandatory for the running of Spring Cloud microservice applications. The ServiceComb engine limits the number of contracts. When the number of microservice application contracts exceeds the limit, the registry fails. If the legacy system cannot be properly split to reduce the number of contracts, the dependency can be excluded and the contract management function is not used.

```
<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>
```

Step 2 Configure microservice information.

Add the microservice description to the **bootstrap.yml** file. If the **bootstrap.yml** file is not available in the project, create one.

```
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}
```

NOTE

- **healthCheckInterval** is in seconds.
- For ServiceComb engine 1.x, {*port*} is **30103** and {*servertype*} is **config-center**.
- For ServiceComb engine 2.x, {*port*} is **30110** and {*servertype*} is **kie** (recommended) or **config-center**.

Step 3 (Optional) Configure security authentication parameters.

Perform this step only when you use the exclusive ServiceComb engine and enable security authentication. In other scenarios, skip this step.

After security authentication is enabled for a ServiceComb engine, all called APIs can be called only after a token is obtained. For details about the authentication process, see **RBAC**.

To use security authentication, obtain the username and password from the ServiceComb engine and then add the following configuration to the configuration file.

- Configuration in plaintext

```
spring:
  cloud:
    servicecomb:
      credentials:
        account:
          name: username
          password: password
          cipher: default
```

- Custom encryption algorithms for storage

Implement the com.huaweicloud.common.util.Cipher API using either of the following methods:
String name(), which is the name definition of **spring.cloud.servicecomb.credentials.cipher** and needs to be added to the configuration file.
char[] decode(char[] encrypted), which is the decryption API used to decrypt secretKey.
public class CustomCipher implements Cipher
To implement encryption and decryption, you need to use BootstrapConfiguration as the startup add-in. Add the following statement first:

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

Add the **META-INF/spring.factories** file to define the configuration:

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

After the custom configuration is complete, you can use the new decryption algorithm in the **bootstrap.yaml** file.

```
spring:
  cloud:
    servicecomb:
      credentials:
        account:
          name: username
          password: password
        cipher: user-defined algorithm name
```

NOTE

The RBAC function requires 1.6.0-Hoxton or later.

----End

4.2 Connecting Java Chassis Applications to ServiceComb Engines

This section describes how to connect Java chassis applications to ServiceComb engines and use the most common functions of ServiceComb engines. For details about the development guide, see **Using ServiceComb Engine Functions**.

In the **Apache ServiceComb Samples** project, you can find the code corresponding to the development methods in this section.

Prerequisites

- Microservice applications have been developed based on Java chassis.

  For details about microservice application development in the Java chassis framework, see **https://servicecomb.apache.org/references/java-chassis/en_US/**.
- Version requirements. See **Requirements for Microservice Development Framework of a ServiceComb Engine**.
- This document assumes that you use Maven for dependency management and packaging in your project. You are familiar with the Maven dependency management mechanism and are able to modify the **dependency management** and **dependency** in the **pom.xml** file.
- Java chassis can be used together with different technologies. The name of the configuration file is related to the technology you use. For example, if you use Java chassis in Spring mode, the configuration file name is **microservice.yaml**. If you use Java chassis in Spring Boot mode, the configuration file name is **application.yaml**. This document uses **microservice.yaml** to indicate the configuration file. You need to use a configuration file name corresponding to your project.

Procedure

Step 1 Add dependencies to the **pom.xml** file of the project.

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


NOTE

- The solution-basic module contains common Java chassis functions, such as the configuration center module and service governance module, which allow you to enable these functions in one-click.

```
<!-- Configuration center module -->
<dependency>
<groupId>org.apache.servicecomb</groupId>
<artifactId>config-cc</artifactId>
</dependency>
<!-- Service governance module -->
<dependency>
<groupId>org.apache.servicecomb</groupId>
<artifactId>handler-governance</artifactId>
</dependency>
```

- The servicestage-environment module consists of the following dependent module:

```
<!-- Registry and discovery module -->
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>registry-service-center</artifactId>
</dependency>
```

You are advised to use Maven Dependency Management to manage the third-party software dependencies of a project. Add the following information to the **pom.xml** file of the project:

```
<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>
```

Skip the operation if your project already contains the preceding dependencies.

The **servicestage-environment** software package is optional. This software package provides the environment variable mapping function. When you use ServiceStage to deploy applications, you do not need to manually modify information such as the registry center address, configuration center address, and project name. The default configurations in the **microservice.yaml** file are overwritten by environment variables. The **mapping.yaml** file is contained in the software package. You can also add the **mapping.yaml** file to your own project.

NOTE

The **mapping.yaml** file may change in later versions to support the latest functions of ServiceComb engines. If you do not want the new version to evolve with ServiceComb engines, you can add the **mapping.yaml** file to your project instead of adding the **servicestage-environment** dependency.

Generally, the **microservice.yaml** and **mapping.yaml** files are stored in the **/src/main/resources/** directory in the root directory of the current project.

```
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
```

Common software packages are added to solution-basic, and the default **microservice.yaml** file is provided. This configuration file configures common Handlers and parameters as follows:

```
# 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
```

servicecomb-config-order is set to **-100** in the **microservice.yaml** configuration file, indicating that the priority of the configuration file is low. (A larger order indicates a higher priority. The default value is **0**.) If the same configuration item is added to the service, the configuration will be overwritten.

> **NOTE**
>
> The **microservice.yaml** file may change in later versions to support the latest functions of ServiceComb engines. If you do not want the new version to evolve with ServiceComb engines, you can write the configuration items to your own **microservice.yaml** file.

Step 2 (Optional) Configure security authentication parameters.

Perform this step only when you use the exclusive ServiceComb engine and enable security authentication. In other scenarios, skip this step.

After security authentication is enabled for a ServiceComb engine, all called APIs can be called only after a token is obtained. For details about the authentication process, see **RBAC**.

To use security authentication, obtain the username and password from the ServiceComb engine and then add the following configuration to the configuration file.

```
servicecomb:
  credentials:
    rbac.enabled: true
    account:
      name: your account name # Username obtained from the ServiceComb engine
      password: your password # Password obtained from the ServiceComb engine
    cipher: default # Returned name of the name() method in the implementation class of API
org.apache.servicecomb.foundation.auth.Cipher
```

cipher specifies the name of the algorithm used to encrypt the **password**. By default, the password is stored in plaintext. The encryption is implemented through customization. The details are as follows:

- Implement the **org.apache.servicecomb.foundation.auth.Cipher** API using either of the following methods:
  - String name()

    Name definition of **servicecomb.credentials.cipher**, which needs to be added to the configuration file.
  - char[] decode(char[] encrypted)

    Decrypt the API, which is used after secretKey is decrypted.

  The implementation class must be declared as **SPI**. For example:

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

  Create an SPI configuration file. The file name and path are **META-INF/service/org.apache.servicecomb.foundation.auth.Cipher**, and the file content is as follows:

  ```
  com.example.MyCipher
  ```

  Add the following configuration to the **microservice.yaml** file:

  ```
  servicecomb:
    credentials:
      rbac.enabled: true
      account:
        name: your account name
        password: your password # Encrypted password
      cipher: youciphername # Returned name of the name() method in the implementation
  class
  ```


NOTE

Plaintext storage cannot ensure security. You are advised to encrypt the password.

----End

5 Deploying Microservice Applications

For details about how to deploy microservice applications, see **Creating and Deploying a Component**.

6 Using ServiceComb Engine Functions

6.1 Using Service Registry

The service center of the ServiceComb engine provides the service registry function that registers basic information, such as the application to which a microservice belongs, microservice name, microservice version, and listening address, with the service center when the microservice is started.

During microservice running, the basic information about other microservices can be queried through the service center. The registered information varies with microservice development frameworks. For example, the service contract information is registered in Java chassis. The registered basic information and the process of registering and discovering other microservices are the same for all microservice development frameworks.

This section describes how different microservice development frameworks use the service center and configure their own registry information, as well as the configuration items related to the interaction between microservices and the registry center. After a microservice is registered, you can use the ServiceComb catalog, instance list, and dependencies in CSE.

Spring Cloud

When Spring Cloud uses service registry, you need to add the following dependencies to the project:

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

If the dependencies have been directly or indirectly included in the project, you do not need to add them. **Table 6-1** describes the configuration items of Spring Cloud. The values of these configuration items affect the basic information registered in the service center and the interaction between microservices and the service center, such as heartbeats. Information related to service registry needs to be configured in the **bootstrap.yml** file.

Table 6-1 Common configuration items of Spring Cloud

Item	Description	Default Value	Remarks
spring.cloud.servicecomb.discovery.appName	Application	default	-
spring.cloud.servicecomb.discovery.serviceName	Microservice name	-	If no service name exists, use **spring.application.name**.
spring.cloud.servicecomb.discovery.version	Microservice version	-	-
server.env	Environment	-	The value can be **production**, **development**, etc.
spring.cloud.servicecomb.discovery.enabled	Whether to enable service registry and discovery	true	-
spring.cloud.servicecomb.discovery.address	Registry center address	-	Use commas (,) to separate cluster addresses.
spring.cloud.servicecomb.discovery.watch	Whether to enable the watch mode	false	-
spring.cloud.servicecomb.discovery.healthCheckInterval	Interval for sending heartbeat messages, in seconds	15	Value range: 1 ≤ configuration item ≤ 600.
spring.cloud.servicecomb.discovery.datacenter.name	Data center name	-	-
spring.cloud.servicecomb.discovery.datacenter.region	Data center region	-	-
spring.cloud.servicecomb.discovery.datacenter.availableZone	AZ of the data center	-	-

Item	Description	Default Value	Remarks
spring.cloud.servicecomb.discovery.allowCrossApp	Whether cross-application calling is supported	false	Server configuration, indicating that clients in different applications are allowed to discover themselves.

Java Chassis

When Java chassis uses service registry, you need to add the following dependencies to the project:

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

If the dependencies have been directly or indirectly included in the project, you do not need to add them. **Table 6-2** describes the configuration items of Java chassis. The values of these configuration items affect the basic information registered in the service center and the interaction between microservices and the service center, such as heartbeats.

Table 6-2 Common configuration items of Java chassis

Item	Description	Default Value	Remarks
servicecomb.service.application	Application	default	-
servicecomb.service.name	Microservice name	defaultMicroservice	-
servicecomb.service.version	Microservice version	1.0.0.0	-
servicecomb.service.environment	Environment	-	The value can be **production**, **development**, etc.
servicecomb.service.registry.address	Registry center address	http://127.0.0.1:30100	Use commas (,) to separate cluster addresses.

Item	Description	Default Value	Remarks
servicecomb.service.registry.instance.watch	Whether to enable the watch mode	true	-
servicecomb.service.registry.instance.healthCheck.interval	Interval for sending heartbeat messages, in seconds	30	-
servicecomb.service.registry.instance.healthCheck.times	Indicates the allowed number of heartbeat failures. If the heartbeat fails for the consecutive times+1 times, the instance is brought offline by the service center. That is, interval x (times + 1) determines the time when an instance is automatically deregistered. If the service center does not receive a heartbeat message for a long time, the service center deregisters the instance.	3	-
servicecomb.datacenter.name	Data center name	-	-
servicecomb.datacenter.region	Data center region	-	-
servicecomb.datacenter.availableZone	AZ of the data center	-	-

The instance address and listening address registered by Java chassis are related to the release address specified by **servicecomb.service.publishAddress**. The configuration items of the service listening address are **servicecomb.rest.address** and **servicecomb.highway.address**, which correspond to the listening addresses of the REST and highway transmission modes, respectively. **Table 6-3** shows the relationship between the registered address, listening address, and release address.

Table 6-3 Effective rules of registered instance addresses

Listening Address	Release Address	Registered Instance Address
127.0.0.1	-	127.0.0.1

Listening Address	Release Address	Registered Instance Address
0.0.0.0	-	Set it to the IP address of a NIC. The wildcard address, loopback address, or broadcast address is not selected.
Specific IP address	-	Set it to the listening address.
*	Specific IP address	Set it to the release address.
*	"{NIC name}"	Specifies the IP address corresponding to the NIC name. Note that the IP address must be enclosed in quotation marks and brackets.

6.2 Using the Configuration Center

6.2.1 Configuration Center Overview

The configuration center is used to manage microservice application configurations. Microservices connect to the configuration center to obtain the information and changes of configurations. The configuration center is also the core component for the management functions of other microservices. For example, service governance rules are delivered through the configuration center.

ServiceComb engines support config-center and kie.

NOTE

- For ServiceComb engine 1.x, the configuration center is config-center.
- For ServiceComb engine 2.x, the configuration center is kie (recommended) or config-center.

This section describes the development details of different microservice development frameworks using the configuration center, including how to configure dependencies and connect to configuration items related to the configuration center, and how to read configurations and respond to configuration changes in microservice applications.

- ServiceComb engines use kie as the configuration center.

  By default, microservices read application configurations, service configurations, and custom configurations from the configuration center. Application configuration refers to the configuration of the same environment and application as the microservice. Service configuration refers to the configuration of the same environment, application, and microservice name as the microservice. A microservice can specify a specific label and label value in the configuration file. Custom configuration refers to the configuration of the same label and label value as the microservice.

  Application- and service-level configurations are applicable to simple scenarios. The application-level configuration is shared by all microservices of

the application. The service-level configuration is exclusive and takes effect only for specific microservices.

In complex scenarios, **customLabel** and **customLabelValue** can be used to define configurations. For example, if some configurations are shared by all applications, this method can be used. Add the following configuration to the configuration file (Spring Cloud is used as an example):

```
spring:
  cloud:
    servicecomb:
      config:
        kie:
          customLabel: public # The default value is public.
          customLabelValue: default # The default value is a null string.
```

If a configuration item has the **public** label and the label value is **default**, the configuration item takes effect for the microservice.

a. The configuration center is considered as the table **tbl_configurations** of the database. The key is the primary key, and each label is an attribute.

b. The client queries the configuration based on the following search criteria:

   - Custom configuration

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

   - Application-level configuration

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

   - Service-level configuration

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

     When **match** is set to **true**, only the attributes specified in the condition are available. When **match** is set to **false**, all attributes except those in the condition are allowed. You can also specify multiple applications for label **app** or services for label **service**. In this way, the configuration item takes effect for multiple services and applications.

For ServiceComb engines of the TEXT and XML types, SDK uses the content as key-value pairs. For CSE of the YAML and Properties types, SDK parses the content and the application uses the content as the actual application configuration items. For example,

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

One configuration item is found in the application: **cse.examples.hello = World**.

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


Two configuration items are found in the application: **cse.key1 = value1** and **cse.key2 = value2**.

- ServiceComb engines use config-center as the configuration center.

  By default, microservices read global configurations and service configurations from the configuration center. Global configuration refers to the environment shared by the microservice engine and microservice. Service configuration refers to the microservice engine's environment, application, and microservice name that are the same as the microservice's.

  ServiceComb engines support only key-value configuration items. To use a configuration file in YAML format, you can use the fileSource function provided by SDK. After the key list of fileSource is specified in the configuration file, SDK parses the values of these keys as YAML files. The following uses Spring Cloud as an example to describe how to add a configuration item to the **bootstrap.yml** file.

  ```
  spring:
    cloud:
      servicecomb:
        config:
          fileSource: file1.yaml,file2.yaml
  ```

  In addition, create configurations in the configuration center. The following table lists the configuration items and their values. The value is in YAML format.

Item	Value
file1.yaml	cse.example.key1: value1 cse.example.key2: value2
file2.yaml	cse.example.key3: value3 cse.example.key4: value4

  For details about how to create a microservice, see **Configuration Management (Applicable to Engine 1.x)**.

  Four configuration items are found in the application: **cse.example.key1=value1**, **cse.example.key2=value2**, **cse.example.key3=value3**, and **cse.example.key4=value4**.

6.2.2 Using the Configuration Center in Spring Cloud

When the configuration center is used in Spring Cloud, you need to add the following dependencies to the project:

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

If the dependencies have been directly or indirectly included in the project, you do not need to add them. Spring Cloud contains the configuration items listed in **Table 6-4**. The values of these configuration items specify the identity of microservices in the configuration center and the interaction between microservices and the configuration center.

Table 6-4 Common configuration items of Spring Cloud

Item	Description	Default Value	Remarks
spring.cloud.servicecomb.discovery.appName	Application	default	-
spring.cloud.servicecomb.discovery.serviceName	Microservice name	-	If no service name exists, use **spring.application.name**.
spring.cloud.servicecomb.discovery.version	Microservice version	-	-
server.env	Environment	-	The value can be **production**, **development**, etc.
spring.cloud.servicecomb.config.enabled	Whether to enable dynamic configuration	true	-
spring.cloud.servicecomb.config.serverType	Configuration center type	config-center	• For ServiceComb engine 1.x, set it to **config-center**. • For ServiceComb engine 2.x, set it to **kie** (recommended) or **config-center**.
spring.cloud.servicecomb.config.serverAddr	Access address. The format is **http(s)://{*ip*}:{*port*}**. Use commas (,) to separate multiple addresses.	-	-
spring.cloud.servicecomb.config.fileSource	List of YAML configuration items, which are separated by commas (,)	-	This parameter is valid only when the configuration center is config-center.

Spring Cloud users who use ServiceComb engine 1.x need to frequently add configuration files in YAML format to the configuration center. Spring Cloud Huawei provides the configuration item

spring.cloud.servicecomb.config.fileSource to enable users to configure configuration files in YAML format. The value of this configuration item is the key list of the key-value system. Multiple keys are separated by commas (,). The values of these keys are text content in YAML format. Spring Cloud Huawei performs special processing and parsing on the values of these keys.

After accessing the configuration center, you can use the @Value and @ConfigurationProperties labels to inject configurations for Spring Cloud applications. Alternatively, you can also use **Environment** to read configurations and @RefreshScope to dynamically change configurations. For details, see the **developer guide of the community**.

6.2.3 Using the Configuration Center in Java Chassis

- Java chassis uses the configuration center named **config-center**.

  You need to add the following dependencies to the project:

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

  If the preceding dependencies have been directly or indirectly included in the project, you do not need to add them. Java chassis contains the configuration items listed in **Table 6-5**. The values of these configuration items specify the identity of microservices in the configuration center and the interaction between microservices and the configuration center.

  Table 6-5 Common configuration items of Java chassis

Item	Description	Default Value	Remarks
servicecomb.service.application	Application	default	-
servicecomb.service.name	Microservice name	defaultMicroservice	-
servicecomb.service.version	Microservice version	1.0.0.0	-
servicecomb.service.environment	Environment	-	The value can be **production**, **development**, etc.
servicecomb.config.client.serverUri	Access address. The format is **http(s)://{*ip*}:{*port*}**. Use commas (,) to separate multiple addresses.	http://127.0.0.1:30103	config-center

Item	Description	Default Value	Remarks
servicecomb.config.client.tenantName	Tenant name of the application	default	config-center

- Java chassis uses the configuration center named **kie**.

  You need to add the following dependencies to the project:

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

  If the preceding dependencies have been directly or indirectly included in the project, you do not need to add them. Java chassis contains the configuration items listed in **Table 6-6**. The values of these configuration items specify the identity of microservices in the configuration center and the interaction between microservices and the configuration center.

  Table 6-6 Common configuration items of Java chassis

Item	Description	Default Value	Remarks
servicecomb.service.application	Application	default	-
servicecomb.service.name	Microservice name	defaultMicroservice	-
servicecomb.service.version	Microservice version	1.0.0.0	-
servicecomb.service.environment	Environment	-	The value can be **production**, **development**, etc.
servicecomb.kie.serverUri	Address for accessing kie. The format is **http(s)://{*ip*}:{*port*}**. Use commas (,) to separate multiple addresses.	-	kie
servicecomb.kie.firstRefreshInterval	Interval for updating configuration items for the first time (ms)	3000	kie
servicecomb.kie.refresh_interval	Interval for updating configuration items (ms)	3000	kie

Item	Description	Default Value	Remarks
servicecomb.kie.domainName	Tenant name of the application	default	kie

Java chassis provides multiple methods to read dynamic configurations.

- The first method is to use the archaius API, for example,

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

  The archaius API supports callback to process configuration change:

```
myprop.addCallback(new Runnable() {
      public void run() {
          // When the value of a configuration item changes, the callback method is invoked.
          System.out.println("trace.handler.sampler.percent is changed!");
      }
  });
```

- The second method is to use the configuration injection mechanism provided by Java chassis. This method can easily handle complex configurations and configuration priorities. For example,

```
  @InjectProperties(prefix = "jaxrstest.jaxrsclient")
  public class Configuration {
    /*
     * The prefix attribute override of a method will overwrite @InjectProperties defined in the class.
     * The prefix attribute of an annotation.
     *
     * The keys attribute can be a string array. A smaller subscript indicates a higher priority.
     *
     * The system searches for configuration attributes in the following sequence until the configured configuration attributes are found:
     * 1) jaxrstest.jaxrsclient.override.high
     * 2) jaxrstest.jaxrsclient.override.low
     *
     * Test case:
     * jaxrstest.jaxrsclient.override.high: hello high
     * jaxrstest.jaxrsclient.override.low: hello low
     * Expected result:
     * hello high
     */
    @InjectProperty(prefix = "jaxrstest.jaxrsclient.override", keys = {"high", "low"})
    public String strValue;
```

  Inject configurations.

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

- The third method is used when Spring and Spring Boot are integrated. The configuration can be read in the native mode of Spring and Spring Boot, for example, @Value and @ConfigurationProperties. Java chassis applies configuration hierarchy to Spring Environment. Spring and Spring Boot can also read the dynamically configured values and the values in the **microservice.yaml** file.

  For more information about the read configurations of Java chassis, see the **developer guide of the community**.

6.3 Using Service Governance

6.3.1 Overview

Service governance is a broad concept. Generally, it refers to some measures that ensure the reliable system running and are independent of business logic. The following assurance measures are provided to deal with the common fault modes in microservice scenarios:

- Load balancing management: provides load balancing policy management in multi-instance scenarios. For example, the polling mode is used to ensure that traffic is balanced among different instances. When an instance is faulty, the instance can be temporarily isolated to prevent access to the instance from causing request timeout.
- Rate limiting: provides load protection and prevents the system from breaking down when the external traffic exceeds the processing capability of the system. Rate limiting is also used to smooth requests so that requests are evenly distributed to services, preventing the impact of burst traffic on the system.
- Retry: prevents random failures, which often occur in the microservice system, which often occur in the microservice system due to many reasons. Take the request timeout of a Java microservice application as an example. This may occur due to network fluctuation or software/hardware upgrade, which may interrupt services for several seconds. The increasing latency due to JVM garbage collection and thread scheduling may also be the cause. The system is more prone to time out when traffic is not even, such as 1000 concurrent requests and 1000 requests within 1s. The interaction between applications, systems, and networks can also cause random failures. The burst traffic of an application may affect the bandwidth and consequently the running of other applications. In other application-related scenarios, for example, SSL needs to obtain the OS entropy. If the entropy is too low, a latency of several seconds will occur. The system must be capable of protection against the inevitable random faults.
- Bulkhead: protects resource-consuming services. For example, if a time-consuming service shares a thread pool with other services, other services will wait when the service receives a large number of sudden requests, compromising the performance of the entire system. The bulkhead allocates an independent resource pool (usually implemented through the semaphore or thread pool) to resource-intensive services to prevent other services from being affected.
- Degradation: During peak hours, access to the target service needs to be temporarily reduced to decrease the load of the target service. Alternatively, access to non-key services needs to be shielded to maintain the core processing capability of the service.

However, no governance measure is applicable to all scenarios. That is, a governance measure that works well in one scenario may cause problems in another. Therefore, it is important to dynamically update the governance policy based on the service running status and metrics.

To use service governance in a service system, perform the following steps:

1. Develop a service. This step focuses more on service function delivery than on service governance. The microservice development framework provides assurance measures by default against common system faults. Selecting an appropriate microservice development framework can save the DFx time.
2. Conduct performance tests and fault drills. Many system instability issues are found in this process. The service governance policies are applied to resolving these issues and written to the configuration file as the default values of the application.
3. Bring the service online. If an unexpected scenario occurs during service rollout, you need to use the configuration center to dynamically adjust governance parameters for stable service running.

The preceding three steps will be continuously optimized throughout the software lifecycle. ServiceComb engines provide unified service governance capability based on request markers for different microservice development frameworks. If a microservice framework is used to develop an application, the microservice is automatically registered with the corresponding ServiceComb engine after the application is hosted and started, and you can perform service governance on the CSE console. For details, see **Governing Microservices**.

This section describes how to use the service governance capability based on request markers.

6.3.2 Request Marking

- Java chassis implements the request marker-based governance capability using Handler. The Provider implements rate limiting, circuit breaker, and bulkhead, and the Consumer implements retry.
  a. To use the request marker-based governance capability, you need to introduce the following dependencies to the code:
  ```
  <dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>handler-governance</artifactId>
  </dependency>
  ```
  b. Configure the Handler chain.
  ```
  servicecomb:
    handler:
      chain:
        Consumer:
          default: governance-consumer,loadbalance
        Provider:
          default: governance-provider
  ```

  Java chassis is a REST/RPC framework based on open APIs. The model is different from the REST framework. Java chassis provides REST-based and RPC-based matching. You can use the **servicecomb.governance.{operation}.matchType** configuration item to specify the matching rule. By default, REST-based matching is used. If the highway protocol in Java chassis is used for calling, set **matchType** to **rpc**. For example,
  ```
  servicecomb:
    governance:
      matchType: rest #Set the global default matching mode to REST and set the highway protocol to rpc.
      GovernanceEndpoint.helloRpc:
        matchType: rpc # Set the interface helloRpc on the server to use RPC-based matching.
  ```

  In REST-based matching, **apiPath** uses a URL, for example,
  ```
  servicecomb:
    matchGroup:
  ```



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

In RPC-based matching, **apiPath** uses an operation, for example,

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

For server governance, such as rate limiting, the header is obtained from HTTP in REST-based matching. For client governance, such as retry, the header is obtained from **InvocationContext** in REST-based matching.

The following describes how to configure different governance policies and add dependencies to the POM file.

One request corresponds to one key. For example, **userLoginAction** is the name of a key. Multiple marking rules can be defined for one request. In each marking rule, the matching rules for apiPath, method, and headers can be defined. The relationship between marking rules is OR, and the relationship between matching rules is AND.

A series of operators are provided in **match** to match apiPath or headers.

- exact: exact match
- prefix: prefix match
- suffix: suffix match
- contains: whether the target string contains the scheme string
- compare: supporting the match in >, <, >=, <=, =, or != mode During the processing, the scheme string and the target string are converted into the Double type for comparison. The supported data range is the Double data range. If the difference between the values of = and != is less than 1e-6, the two values are considered equal. For example, if the scheme string is > –10, the target string greater than –10 is matched.

Request marking can be implemented at different application layers. For example, on the server that provides REST APIs, request information can be obtained through the HttpServletRequest API. The client called by the RestTemplate can obtain request information from the RestTemplate.

The methods for information extraction vary with frameworks and application layers. The implementation layer shields differences by mapping features to GovernanceRequest. In this way, the governance capability can be used in different frameworks and application layers.

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

  private String uri;

  private String method;}
```

- Spring Cloud uses Aspect to intercept the RequestMappingHandlerAdater class to implement rate limiting, circuit breaker, and bulkhead, and intercept RestTemplate and FeignClient to implement retry.

  To use the request marker-based governance capability, you need to introduce the following dependencies to the code:


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

Spring Cloud is based on the REST framework and can better match the matching semantics of marker-based governance. **apiPath** and **headers** correspond to the HTTP protocol:

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

6.3.3 Rate Limiting

The rate limiting rule is based on Resilience4j and works on the server. The principles are as follows: A maximum of **rate** requests can be accepted at the interval specified by **limitRefreshPeriod**. If the number of requests exceeds the value of **rate**, the traffic is limited and the response code 429 is returned.

- Rate limiting of Java chassis is used for microservice providers. The rate limiting module must be integrated into microservice applications and the **qps-flowcontrol-provider** processing chain must be enabled.

  Configuration example:

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

  Add the following dependency to the POM file:

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

  For details, see ***ServiceComb Rate Limiting Development Guide***.

- Spring Cloud uses Aspect to intercept RequestMappingHandlerAdater to implement rate limiting. After Spring Cloud Huawei is integrated, the rate limiting module spring-cloud-starter-huawei-governance is integrated by default. You only need to enable a specific rate limiting policy.

  Configuration example:

  ```
  servicecomb:
    matchGroup:
      AllOperation: |
        matches:
          - apiPath:
              prefix: "/"
    rateLimiting:
      AllOperation: |
        rate: 10 # A maximum of 10 requests are allowed in a period of time.
  ```


6.3.4 Fault Tolerance

Based on whether the retry interval is fixed, retry policies are classified into fixed interval and exponential interval. The default retry policy is fixed interval.

- Fault tolerance of Java chassis is used for microservice consumers. The fault tolerance module must be integrated into microservice applications and the **bizkeeper** processing chain must be enabled.

  Configuration example:

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

  Add the following dependency to the POM file:

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

  NOTE

  The microservice development framework Java Chassis 2.x is used as an example.

- Spring Cloud uses Aspect to intercept RequestMappingHandlerAdater to implement fault tolerance. After Spring Cloud Huawei is integrated, the client fault tolerance module spring-cloud-starter-huawei-governance is integrated by default. You only need to enable a specific client fault tolerance policy.

  Configuration example:

  ```
  servicecomb:
    matchGroup:
      AllOperation: |
        matches:
          - apiPath:
              prefix: "/"
    retry:
      AllOperation: |
        maxAttempts: 3  # Number of retries
        retryOnSame: 1  # Instance initiated by retry
        retryOnResponseStatus:  # Retry error code
          - 502
          - 503
  ```

  The default policy takes effect when the error code is 502 or 503. In 1.11.4-2021.0.x/1.11.4-2022.0.x and later versions, the response header takes effect in special scenarios.

  The response header is defaulted to **X-HTTP-STATUS-CODE**. You can also customize the key as follows:

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

  The response code set in the response header can also be customized. However, you need to add the corresponding error code to the fault tolerance policy. For example, if you set **X-HTTP-STATUS-CODE=511**, add error code 511. The configuration is as follows:


```
servicecomb:
  matchGroup:
    AllOperation: |
      matches:
        - apiPath:
            prefix: "/"
  retry:
    AllOperation: |
      maxAttempts: 3  # Number of retries
      retryOnSame: 1  # Instance initiated by retry
      retryOnResponseStatus:  # Retry error code
        - 502
        - 503
        - 511
```

The system checks the response code first. If the abnormal response code meets the policy setting, the fault tolerance function is enabled. If the abnormal response code does not meet the policy setting, the system checks whether the response code set in the header meets the requirement.

6.3.5 Circuit Breaker

The circuit breaker rule is based on Resilience4j and works on the server. The principles are as follows:

When the specified value of **failureRateThreshold** or **slowCallRateThreshold** is reached, the circuit breaker is triggered and response code 429 is returned. **SlowCallDurationThreshold** indicates the slow call duration threshold. **minimumNumberOfCalls** indicates the minimum number of requests that meet the circuit breaker requirement. For example, if the value of **minimumNumberOfCalls** is **10**, at least 10 calls must be recorded to calculate the failure rate. If only nine calls are recorded, CircuitBreaker will not be enabled even if all the nine calls fail. **slidingWindowType** specifies the type of the sliding window. The default value is **count** (based on the number of requests) or **time** (based on the time window). If the sliding window type is **count**, the latest **slidingWindowSize** calls are recorded and counted. If the sliding window type is **time**, the calls in the latest **slidingWindowSize** seconds are recorded and counted. **slidingWindowSize** specifies the size of the sliding window. The unit can be the number of requests or second, depending on the sliding window type.

- Circuit breaker of Java chassis is used for microservice consumers. The circuit breaker module must be integrated into microservice applications and the **bizkeeper-consumer** processing chain must be enabled.

  Configuration example:

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

  Add the following dependency to the POM file:

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

  NOTE

  The microservice development framework Java Chassis 2.x is used as an example.

- Spring Cloud Huawei uses Aspect to intercept RequestMappingHandlerAdater to implement circuit breaker. After Spring Cloud Huawei is integrated, the

client circuit breaker module spring-cloud-starter-huawei-governance is integrated by default. You only need to enable a specific client circuit breaker policy.

Configuration example:

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

The default policy takes effect when the error code is 502 or 503. In 1.11.4-2021.0.x/1.11.4-2022.0.x and later versions, the response header takes effect in special scenarios.

The default key of the response header is **X-HTTP-STATUS-CODE**. You can also customize the key by configuring the following on the client:

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

The response code set in the response header can also be customized. However, you need to add the corresponding error code to the fault tolerance policy. For example, if you set **X-HTTP-STATUS-CODE=511**, add error code 511. The configuration is as follows:

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

The preceding configuration enables the client circuit breaker policy for all instances. This policy uses the COUNT_BASED sliding window policy. The window size is 10 requests. When the number of requests reaches 10, the error rate starts to be calculated. If the error rate reaches 20%, circuit breaker is performed for subsequent requests. The default sliding window policy is **TIME_BASED**. The system checks the response code first. If the abnormal response code meets the policy setting, the fault tolerance function is enabled. If the abnormal response code does not meet the policy setting, the

system checks whether the response code set in the header meets the requirement.

6.3.6 Bulkhead

Bulkhead is an exception detection mechanism. It is used when a request timeout or large traffic occurs. Generally, you need to set the timeout duration and the number of concurrent requests.

- Bulkhead of Java chassis is used for microservice consumers. The bulkhead module must be integrated into microservice applications and the **bizkeeper-consumer** processing chain must be enabled.

  Configuration example:

  ```
  servicecomb:
    handler:
      chain:
        Consumer:
          default: bizkeeper-consumer
    isolation:
      Consumer:
        timeout:
          enabled: true #Whether to enable timeout detection
        timeoutInMilliseconds: 30000 #Timeout threshold
  ```

- The bulkhead policy of Spring Cloud Huawei is the same as that of circuit breaker. For details about the configuration example, see **Circuit Breaker**.

6.3.7 Load Balancing

Load balancing functions on the client and is an indispensable key component of a high-concurrency and high-availability system. It aims to evenly distribute network traffic to multiple servers to improve the overall response speed and availability of the system.

- Load balancing of Java chassis is used for microservice consumers. The load balancing module must be integrated into microservice applications and the **loadbalance** processing chain must be enabled.

  Configuration example:

  ```
  servicecomb:
    handler:
      chain:
        Consumer:
          default: loadbalance
    loadbalance:
      strategy:
        name: RoundRobin #The polling mode is enabled.
  ```

  Add the following dependency to the POM file:

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

- Spring Cloud Huawei load balancing is based on Ribbon in Spring Cloud and works on the client. The principles are as follows: When a random rule is used, the client randomly accesses an instance in the downstream microservice instance. When a polling rule is used, the client cyclically selects a server in the downstream microservice instance in sequence.

  Configuration example:


```
servicecomb:
  loadbalance:
    userLoginAction: |
      rule: Random #The random mode is enabled. The default mode is polling.
```

6.3.8 Service Degradation

During peak hours, access to the target service needs to be temporarily reduced to decrease the load of the target service. Alternatively, access to non-key services needs to be shielded to maintain the core processing capability of the service.

- Service degradation of Java chassis is used for microservice consumers. The service degradation module must be integrated into microservice applications and the **bizkeeper-consumer** processing chain must be enabled.

  Configuration example:

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

  Add the following dependency to the POM file:

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

  NOTE

  The microservice development framework Java Chassis 2.x is used as an example.

- Spring Cloud Huawei degradation is a governance rule. After Spring Cloud Huawei is integrated, the client governance module spring-cloud-starter-huawei-governance is integrated by default. The principles are as follows: When the traffic target path is requested, null is returned for all requests. **forceClosed** is a parameter for forcibly disabling degradation governance. When **forceClosed** is set to **true**, degradation governance is forcibly disabled. The default value is **false**.

  Configuration example:

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

  When the preceding configuration is enabled, requests for accessing any API of MyMicroservice will be blocked and FaultInjectionException with error code 500 will be returned.

6.3.9 Fault Injection

You can use fault injection on the consumer side to configure the latency, fault, and triggering probability of requests sent to a specified microservice to ensure that core services are accessed only by key microservices during peak hours.

NOTE

Spring Cloud Huawei does not support fault injection.

Fault injection of Java chassis is used for microservice consumers. The fault injection module must be integrated into microservice applications and the fault-injection-consumer processing chain must be enabled.

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

Add the following dependency to the POM file:

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

For details, see ***ServiceComb Fault Injection Development Guide***.

6.3.10 Customized Governance

The default implementation of service governance does not solve all service problems. The customized governance function allows you to use request marker-based governance capability in different scenarios, for example, rate limiting in the gateway scenario, and URL matching in the Java chassis scenario. The SDK is based on Spring, and all Spring-based frameworks can flexibly use these APIs with similar methods.

The following uses rate limiting as an example to describe how to use an API. You can also use custom API-based code to deliver services and governance rules on the ServiceComb engine management console.

The basic code process includes declaring the reference of RateLimitingHandler, creating GovernanceRequest, intercepting (packaging) the service logic, and handling governance exceptions.

```
 @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();
}
```

This section describes the customized development. For in-depth usage, you can also refer to the default implementation code in the Java chassis and Spring Cloud projects.

6.3.11 Blacklist/Whitelist

The configured blacklist/whitelist takes effect only after public key authentication is enabled.

6.4 Using Dark Launch

In dark launch, a small number of users test the trial version, ensuring the smooth rollout of new features. Once new features become mature, a formal version is released for all users. Dark launch ensures stability of the entire system. During initial dark launch, problems can be detected and fixed.

For ServiceComb Java chassis and Spring Cloud Huawei microservices registered with the ServiceComb engines, deliver configurations to use dark launch.

If ServiceComb Java chassis depends on handler-router and Spring Cloud Huawei depends on spring-cloud-starter-huawei-router to implement microservice dark launch, the delivery rules comply with the following specifications:

```
servicecomb:
    routeRule:
      provider: | #Service name.
      - precedence: 2 #Priority.
          match:        #Matching policy.
            headers:          #Header matching.
              region:
                exact: 'providerRegion'

              type:
                exact: gray
          route: #Routing rule.
          - weight: 100 #Weight value.
            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
```

The preceding configurations are described as follows:

- **match** specifies the requests to be matched. The matching condition is **headers**. Fields in headers support exact match. If **match** is not defined, any request can be matched.
- The forwarding weight is defined in **routeRule.{targetServiceName}.route** and is configured by **weight**. The value of **weight** indicates the percentage. The sum of the values must be equal to 100. If the sum is smaller than 100, the value in the latest version is calculated.
- The service group is defined under **routeRule.***{targetServiceName}***.route** and is configured by **tags**. **version** is a special tag, indicating the microservice version. You can also configure other properties, which are defined in the properties of the instance.
- A larger priority value indicates a higher priority.

If ServiceComb Java chassis depends on darklaunch to implement microservice dark launch, the rules are also delivered on the ServiceComb engine page and comply with the following specifications:

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

The preceding configurations are described as follows:

- **policyCondition**: matching condition of the routing rule. This rule matches the request parameter **name**. When the value is **11111**, the current routing rule is matched.
- **groupName**: name of the routing rule.
- **groupCondition**: target group of the rule. When **name=11111** is matched, the route is routed to the microservice instance with **version=0.0.1**.
- The configuration item is fixed to **cse.darklaunch.policy.***${serviceName}*.

Spring Cloud Huawei

When Spring Cloud Huawei uses the dark launch, you need to add the following dependencies to the project. If the dependencies have been directly or indirectly included in the project, you do not need to add them.

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

Set the **headers** parameter on which the dark launch rule depends.

- In 1.10.7 and later versions, set the **header** parameter in the configuration file. The header in the user request is not transparently transmitted to the downstream service.

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

- In versions earlier than 1.10.7, the **header** parameters set in the request are transparently transmitted.

Java Chassis

When Java chassis uses dark launch, you need to add the following dependencies to the project. If the dependencies have been directly or indirectly included in the project, you do not need to add them.

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

or

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

Add the following configuration item to the configuration file:

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

By default, Java chassis does not transfer non-parameter headers to microservices. If dark launch depends on non-parameter headers, you can add the following configuration items:

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

Java chassis uses these non-parameter headers for dark launch matching.

If the request is forwarded by EdgeService, you also need to add configurations related to dark launch to EdgeService.

6.5 Using Dashboard

The dashboard provides some basic capabilities for monitoring microservice running. Microservices report running status data using SDKs. The reported data includes request statistics, such as the quantity, latency, and error rate, as well as governance-related status, such as the circuit breaker status.

- Spring Cloud directly uses the dashboard without dependencies. Spring Cloud contains the configuration items listed in **Table 6-7**. These configuration items specify information such as the dashboard reporting address.

Table 6-7 Common configuration items of Spring Cloud Huawei

Item	Description	Default Value
spring.cloud.servicecomb.dashboard.invocationProviderEnabled	Request-based interface count is used.	true
spring.cloud.servicecomb.dashboard.governanceProviderEnabled	Circuit breaker-based interface count is used.	false
spring.cloud.servicecomb.dashboard.enabled	Whether to enable the dashboard data reporting function	false
spring.cloud.servicecomb.dashboard.address	Address to which dashboard data is reported. The format is **http://***{ip}*:*{port}*. Use commas (,) to separate multiple addresses. **NOTE** Change the port number to **30109**. For details about how to obtain the address for reporting dashboard data, see **Obtaining the Configuration Center Address of a ServiceComb Engine**.	-

NOTE

Either request-based interface count or circuit breaker-based count takes effect each time.

- When Java chassis uses the dashboard, you need to add the following dependencies to the project:

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

If the dependencies have been directly or indirectly included in the project, you do not need to add them. Java chassis contains the configuration items listed in **Table 6-8**. These configuration items specify information such as the dashboard reporting address.

Table 6-8 Common configuration items of Java chassis

Item	Description	Default Value
servicecomb.monitor.client.serverUri	Address to which dashboard data is reported. The format is **http://***{ip}*:*{port}*. Use commas (,) to separate multiple addresses. **NOTE** Change the port number to **30109**. For details about how to obtain the address for reporting dashboard data, see **Obtaining the Configuration Center Address of a ServiceComb Engine**.	-
servicecomb.monitor.client.enabled	Whether to enable data reporting	true
servicecomb.monitor.client.interval	Report period (ms)	10000

6.6 Using Security Authentication

6.6.1 Security Authentication Overview

The exclusive ServiceComb engine with security authentication enabled provides the system management function using the role-based access control (RBAC) through the microservice console. You can use an account associated with the **admin** role to create an account and associate a proper role with the account based on service requirements. The user who uses this account has the access and operation permissions on the ServiceComb engine.

After security authentication is enabled for an exclusive ServiceComb engine, all called APIs can be called only after a token is obtained. For details about the authentication process, see **RBAC**.

For an exclusive ServiceComb engine with security authentication enabled, perform the following operations before using security authentication:

1. **Creating a Security Authentication Account and Password**
2. **Configuring the Security Authentication Account and Password for a Microservice**

NOTE

- Spring Cloud must integrate Spring Cloud Huawei 1.6.1 or later and Java chassis 2.3.5 or later to support security authentication.
- The exclusive ServiceComb engine with security authentication disabled is upgraded to the new version and security authentication is enabled. For details, see **Managing Security Authentication for a ServiceComb Engine**.

6.6.2 Creating a Security Authentication Account and Password

Create an account name and password for the exclusive ServiceComb engine with security authentication enabled. For details, see **System Management**.

6.6.3 Configuring the Security Authentication Account and Password for a Microservice

After enabling programming interface security authentication of an exclusive ServiceComb engine, you need to enable the same function of microservice components connected to the engine. Programming interface security authentication is triggered by configuring the security authentication account and password. Currently, the configuration file configuration mode and environment variable injection mode are supported.

For security purposes, you are advised to encrypt the account and password before using them.

NOTE

If programming interface security authentication is not enabled for the exclusive ServiceComb engine, but the security authentication account name and password are configured for the microservice component, the engine will verify the account configured for the microservice component.

Configuring the Security Authentication Account and Password for a Spring Cloud Microservice Component

- Configure the configuration file

  Add the following configurations to the **bootstrap.yml** file of the microservice. If they are configured, skip this step.

```
spring:
  cloud:
    servicecomb:
      credentials:
        account:
          name: test # Set this parameter based on the actual value.
          password: mima # Set this parameter based on the actual value.
          cipher: default
```

  NOTE

  By default, the user password is stored in plaintext, which cannot ensure security. You are advised to encrypt the password for storage. For details, see **Custom encryption algorithms for storage**.

- Enter environment variables

  Add the environment variables listed in **Table 6-9** to the microservice.

Add environment variables. For details, see **Managing Application Environment Variables**.

Table 6-9 Environment variables

Name	Description
spring_cloud_servicecomb_credentials_account_name	Set it based on the actual value.
spring_cloud_servicecomb_credentials_account_password	Set it based on the actual value. **NOTE** By default, the user password is stored in plaintext, which cannot ensure security. You are advised to encrypt the password for storage. For details, see **Custom encryption algorithms for storage**.

Configuring the Security Authentication Account and Password for a Java Chassis Microservice Component

- Configure the configuration file

  Add the following configurations to the **microservice.yml** file of the microservice. If they are configured, skip this step.

  ```
  servicecomb:
    credentials:
      rbac.enabled: true # Set this parameter based on the actual value.
      cipher: default
      account:
        name: test # Set this parameter based on the actual value.
        password: mima # Set this parameter based on the actual value.
        cipher: default
  ```

  NOTE

  By default, the user password is stored in plaintext, which cannot ensure security. You are advised to encrypt the password for storage. For details, see **Configure security authentication parameters**.

- Enter environment variables

  Add the environment variables listed in **Table 6-10** to the microservice.

  Add environment variables. For details, see **Managing Application Environment Variables**.

Table 6-10 Environment variables

Name	Description
servicecomb_credentials_rbac_enabled	• **true**: security authentication enabled. • **false**: security authentication disabled.
servicecomb_credentials_account_name	Set it based on the actual value.
servicecomb_credentials_account_password	Set it based on the actual value. **NOTE** By default, the user password is stored in plaintext, which cannot ensure security. You are advised to encrypt the password for storage. For details, see **Configure security authentication parameters**.

7 Appendix

7.1 Java Chassis Version Upgrade Reference

- Java chassis earlier than 2.1.3 is used to the ServiceComb engine.
  a. The CSE SDK needs to be introduced using the following dependencies:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.huawei.paas.cse</groupId>
            <artifactId>cse-dependency</artifactId>
            <version>version</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

  b. Add dependencies.

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

  The CSE SDK is introduced, and an extra repository needs to be added to Maven settings.

```
<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>
```

- Upgrade the version to 2.1.3 or later.
  a. Modify 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. Add dependencies.

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

If this package depends on another software package whose **groupId** is **com.huawei.paas.cse**, delete the depended software package. For 2.1.3 or later, all software packages can be obtained from the Maven central repository. You do not need to configure the Maven repository.

7.2 Local Development Tool

The local development tool includes the local lightweight ServiceComb engine 2.x and provides a lightweight service center, configuration center, and easy-to-use UI for local development.

For details, see the **README.md** file in the local development tool package.

Table 7-1 Local engine resource quota

Function	Resource	Quota
Microservice management	Microservice versions	10,000
	Number of instances of a single microservice	100
	Number of contracts of a single microservice	500
Configuration management	Configuration items	600

Table 7-2 Local ServiceComb engine versions

Version	ServiceComb Engine Version	Release Date	How to Obtain
2.1.8	2.x	2023.9.25	Local-CSE-2.1.8-windows-amd64.zip
			Local-CSE-2.1.8-linux-amd64.zip

Version	ServiceComb Engine Version	Release Date	How to Obtain
			Local-CSE-2.1.8-linux-arm64.zip
			Local-CSE-2.1.8-darwin-amd64.zip
			Local-CSE-2.1.8-darwin-arm64.zip

NOTE

- Local ServiceComb engines are used only for local development and debugging and are not commercial.
- The local ServiceComb engine can be used in Windows, Mac, and Linux OSs. If Mac OS is used, you need to run the downloaded Mac package in the **Users/***xxx* directory. *xxx* indicates the user name for logging in to the system.

7.3 Using ServiceComb Engines by Mesher

7.3.1 Mesher Overview

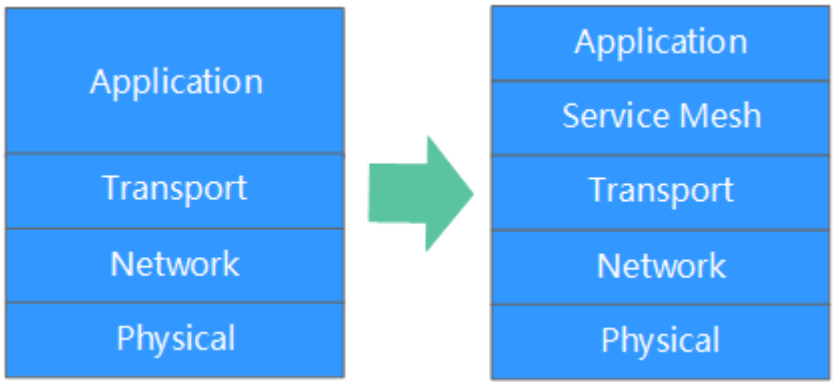
What Is Mesher?

Mesher provides the Service Mesh, which is a lightweight proxy service that runs together with microservices in Sidecar mode.

Service Mesh is defined by William Morgan.

A Service Mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the Service Mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.

The concept of the Service Mesh as a separate layer is tied to the rise of the cloud native application. In the cloud native model, a single application might consist of hundreds of services; each service might have thousands of instances; and each of those instances might be in a constantly-changing state. Not only is service communication in this world incredibly complex, it's a pervasive and fundamental part of runtime behavior. Managing it is vital to ensuring end-to-end performance and reliability.



The Service Mesh is a networking model that sits at a layer of abstraction above TCP/IP. It assumes that the underlying L3/L4 network is present and capable of delivering bytes from point to point. (It also assumes that this network, as with every other aspect of the environment, is unreliable; the Service Mesh must therefore also be capable of handling network failures.)

In some ways, the Service Mesh is analogous to TCP/IP. Just as the TCP stack abstracts the mechanics of reliably delivering bytes between network endpoints, the Service Mesh abstracts the mechanics of reliably delivering requests between services. Like TCP, the Service Mesh does not care about the actual payload or how it is encoded. The application has a high-level goal ("send something from A to B"), and the job of the Service Mesh, like that of TCP, is to accomplish this goal while handling any failures along the way.

Unlike TCP, the Service Mesh has a significant goal beyond "just make it work": it provides a uniform, application-wide point for introducing visibility and control into the application runtime. The explicit goal of the Service Mesh is to move service communication out of the realm of the invisible, implied infrastructure, so that it can be monitored, managed and controlled in the ecosystem.

Why Mesher?

- Service codes do not need to be reconstructed.
- Existing applications can be accessed.
- Common applications quickly become cloud-native.
- Service codes do no need to be modified.

Basic Implementation Principle

Mesher is the proxy of Layer 7 protocols. It runs in Sidecar mode in a pod where applications reside, and shares network and storage resources with the pod.

1. Applications in the Pod use Mesher as the HTTP proxy to automatically discover other services.
2. Instead of applications in the Pod, Mesher registers with the registry center to be discovered by other services.

The network request process of a consumer and provider using Service Mesh is as follows:

- Scenario 1. Only the consumer uses Mesher in Sidecar mode.

Provider needs to implement service registry and discovery or use the Java development framework. Otherwise, the consumer connected through Mesher cannot discover the provider.

The network request process between applications is as follows:

consumer-> Mesher -> provider

- Scenario 2: Both the consumer and provider use Mesher in Sidecar mode.

  In this scenario, the microservice development framework is not required.

  The network request process between applications is as follows:

  consumer -> Mesher -> Mesher ->provider
- Scenario 3. Only the provider uses Mesher in Sidecar mode.

  Consumer needs the Java development framework.

  The network request process between applications is as follows:

  consumer -> Mesher -> provider

Precautions

Configurations need to be modified after applications are deployed on the cloud. For example, before Mesher is deployed, the consumer uses **http://IP:port/** to access the provider; after Mesher is deployed, the consumer uses **http://provider:port/** to access the provider. For details, see **Connecting Mesher Applications to CSE**.

7.3.2 Connecting Mesher Applications to CSE

NOTICE

Unlike the microservice development framework, the Mesher capability is provided by ServiceStage. You must enable multi-language access to Service Mesh on ServiceStage.

This section describes how to connect HTTP applications to ServiceComb engines using Mesher. Mesher supports multiple languages. This section describes only the specifications for connecting Mesher applications to ServiceComb engines. For details about the sample code, see the following:

- **Connecting to the Service Mesh Through .NET Core**
- **Connecting PHP to Service Mesh**

Prerequisites

An HTTP application (supporting multiple languages) has been developed.

Procedure

Step 1 Change **${IP:Port}** in the URL called by the microservice to a service name.

For example, if the microservice name is **provider** and the API is **/hello**, the URL is **http://${IP:Port}/hello**. For example:


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

You need to change the called URL to:

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

Step 2 Deploy components on ServiceStage, bind the ServiceComb engine, and connect the components to the ServiceComb engine. You can select the bound

ServiceComb engine in **Advanced Settings**, click  , and enter the listening port number of the application process to enable multi-language access to Mesher. For details, see **Creating and Deploying a Component** .

NOTE

If the component is deployed in a container, multi-language access to Service Mesh is supported. If the component is deployed on a VM, multi-language access to Service Mesh is not supported.

----End

7.4 Resolved Issues in Earlier Versions of Spring Cloud Huawei and Java Chassis

This section lists all the issues that have been resolved in earlier versions of Spring Cloud Huawei and Java chassis.

Spring Cloud Huawei

Spring Cloud Huawei Version	Resolved Issues
1.11.6-2023.0.x 1.11.6-2022.0.x 1.11.6-2021.0.x	• Security vulnerabilities existed in the **snakeyaml**, **jackson**, and **guava** configuration files. • After the routing was disabled, the microservice application failed to be started and the nacos/ servicecomb adaptation implementation class cannot be found. • When a configuration update event was released in the same configuration center configurations, the pooling configuration may not be found during the request.
1.11.4-2022.0.x 1.11.4-2021.0.x	• When RBAC security authentication was not enabled, the framework still listened to authentication expiration events. • The gateway and webflux routes cannot obtain the request header setting information. • When microservice API security authentication was enabled, all requests failed because the rule was not set. • The circuit breaker rule on the server did not take effect.

1.11.3-2022.0.x 1.11.3-2021.0.x	When a service name was specified, the instance isolation policy did not take effect.
1.11.2-2022.0.x 1.11.2-2021.0.x	• When no blacklist or whitelist policy was configured for API security authentication, a null pointer exception occurred. • When the same request header was set on both the server and client, the key did not take effect.
1.11.0-2022.0.x 1.11.0-2021.0.x	The trace context configuration did not take effect based on dynamic configuration.
1.10.13-2021.0.x	When multiple services were called at the same time, the degradation did not take effect.
1.10.11-2021.0.x	Instance isolation governance did not take effect.
1.10.9-2021.0.x	• When a service name was specified, the retry policy did not take effect. • When a service degradation error occurred, the response was empty instead of the **null** string.
1.10.8-2021.0.x 1.10.8-2020.0.x	The load balancing rule did not take effect.
1.10.7-2021.0.x 1.10.7-2020.0.x	When service registry and discovery was disabled, the start failed.
1.10.6-2021.0.x 1.10.6-2020.0.x	Environment information was missing from the monitoring information.
1.10.5-2021.0.x 1.10.5-2020.0.x	There were too many retries. As a result, the request was not responded for a long time.
1.10.4-2021.0.x 1.10.4-2020.0.x	The identifierRateLimiting context failed to be obtained.
1.10.3-2021.0.x 1.10.3-2020.0.x	The governance configuration did not take effect when it was changed for the first time.
1.10.2-2021.0.x 1.10.2-2020.0.x	• The default configuration update interval was changed to 15s. • The null pointer of the instance isolation filter was abnormal.

1.10.1-2021.0.x 1.10.1-2020.0.x	• A null pointer exception occurred in the context of a non-client request. • The routing client failed to compile the request header. • The dark launch version policy took effect. • A conversion exception occurred when **ClientRequest** was not of the **RequestData** type.
1.10.0-2021.0.x 1.10.0-2020.0.x	• When a service was deleted or restarted, the Ribbon cache cannot be updated. As a result, the request was sent to an unavailable service, and the error "no host to route" was displayed. • After a dark launch release rule was configured in dynamic configuration, the modification did not take effect. • The startup application failed to be started at the outer layer of the service package. • The maximum number of gateway retries did not take effect.
1.9.1-2020.0.x	• In some scenarios, the ags attribute of the startup class was incorrectly loaded. • The maximum number of retries was unlimited. • The dark launch dynamic configuration did not take effect.
1.9.0-2020.0.x	When **instance.healthCheck.mode** was **pull**, the custom **healthCheckInterval** did not take effect.
1.8.0-2020.0.x	• When too many query tasks were configured in non-long polling mode, the tasks were triggered at no interval. • After a service instance was brought offline on the CSE page, the service instance can still be called. • **management.server.port** and **server.port** were different, causing an error during startup. • The gateway obtained instances in different environments.
1.7.0-2020.0.x	• The gateway failed to be started because Web MVC was combined with the route. • The gateway cannot implement route definition based on service discovery. • The gateway cannot implement service discovery across applications.
1.6.1-2020.0.x	**NOTE** Major problems exist. You are not advised to use it: • The configuration center was queried frequently. • Incorrect configurations were found.

1.9.4-Hoxton	When a configuration update event was released in the same configuration center configurations, the pooling configuration may not be found during the request.
1.9.3-Hoxton	When a service was deleted or restarted, the Ribbon cache cannot be updated. As a result, the request was sent to an unavailable service.
1.9.2-Hoxton	After an instance was deleted and another instance was registered on the server, the client selected an incorrect server instance.
1.9.1-Hoxton	• In some scenarios, the ags attribute of the startup class was incorrectly loaded. • The maximum number of retries was unlimited. • The dark launch dynamic configuration did not take effect.
1.9.0-Hoxton	When **instance.healthCheck.mode** was **pull**, the custom **healthCheckInterval** did not take effect.
1.8.0-Hoxton	• When too many query tasks were configured in non-long polling mode, the tasks were triggered at no interval. • After a service instance was brought offline on the CSE page, the service instance can still be called. • **management.server.port** and **server.port** were different, causing an error during startup. • The gateway obtained instances in different environments.
1.7.0-Hoxton	• The gateway failed to be started because Web MVC was combined with the route. • The gateway cannot implement route definition based on service discovery. • The gateway cannot implement service discovery across applications.
1.6.0-Hoxton	• SDK sent a retry request to the engine based on error codes 401 and 403. • The default routing rule of the gateway did not take effect. • Cross-application calling was not supported.
1.5.9-Hoxton	• RBAC authentication did not take effect. • The latest microservice version cannot be obtained during dark launch routing. • Some JDK versions did not support swagger cyclic dependency.

1.5.8-Hoxton	• An incorrect instance was selected during gateway service discovery. • A calling exception occurred when the instance status was empty.
1.5.6-Hoxton	• The AK/SK did not take effect after being configured, and the authentication failed. • The **server.env** configurations did not take effect and were empty. • Governance configuration items in the configuration center can still be used after being deleted. • The circuit breaker configuration attribute of the sliding window did not take effect. • A null pointer exception occurred after the watch mode was enabled in the registry center. • Only the first address can be read when the environment variable **PAAS_CSE_SC_ENDPOINT** was read.
1.5.0-Hoxton	• Incorrect governance rules caused a null pointer exception. • A null pointer exception occurred when the AK/SK was not configured. • The traffic limiting policy did not take effect for the first concurrent request after the service was started. • An incorrect service was selected for governance. As a result, the request was abnormal. • When the environment was production and the schema remained unchanged, the service failed to be restarted.
1.6.4-Greenwich	When a configuration update event was released in the same configuration center configurations, the pooling configuration may not be found during the request.
1.6.3-Greenwich	When a service was deleted or restarted, the Ribbon cache cannot be updated. As a result, the request was sent to an unavailable service.
1.6.1-Greenwich	An exception occurred during cross-application service discovery of the gateway.

1.6.0-Greenwich	• SDK sent a retry request to the engine based on error codes 401 and 403. • The routing rule of the gateway did not take effect. • The latest microservice version cannot be obtained during dark launch routing. • The AK/SK configuration did not take effect. • The **server.env** configurations did not take effect and were empty. • Service center failed to be started in watch mode. • Only the first address can be read when the environment variable **PAAS_CSE_SC_ENDPOINT** was read. • Cross-application calling was not supported.
1.5.0-Greenwich	• Incorrect governance rules caused a null pointer exception. • A null pointer exception occurred when the AK/SK was not configured. • The traffic limiting policy did not take effect for the first concurrent request after the service was started. • A service forwarding error occurred in governance. • When the environment was production, the schema was repeatedly registered. As a result, the startup failed.
v1.3.3-Greenwich	The listening function of the registry center did not take effect.
1.6.1-Finchley	• The calling failed randomly during the first concurrency after the service was started. • Microservices failed to be called across applications.

1.6.0-Finchley	• SDK sent a retry request to the engine based on error codes 401 and 403. • The routing rule of the gateway did not take effect. • The latest microservice version cannot be obtained during dark launch routing. • The AK/SK configuration did not take effect. • The **server.env** configurations did not take effect and were empty. • Service center failed to be started in watch mode. • Only the first address can be read when the environment variable **PAAS_CSE_SC_ENDPOINT** was read. • Cross-application calling was not supported.
1.5.1-Finchley	Governance configurations in the configuration center can still be used after being deleted.
v1.3.9	**NOTE** Major problems exist. You are not advised to use it: A critical service forwarding error occurred in governance.
v1.3.8	**NOTE** Major problems exist. You are not advised to use it: A critical service forwarding error occurred in governance.
v1.3.4	• The registry thread pool cannot be closed correctly, causing leakage. • The microservice failed to be registered after the actuator was enabled. • There were too many heartbeats in some scenarios.
v1.3.3	• WebSocket failed to send a WebService request. • The watch of the registry center did not take effect.
v1.3.2	• When the environment was production, the schema failed to be registered. • Random address selection failed when the registry center URL was not set. • The watch was abnormal when the domain name was configured in the registry center.
v1.2.0	When the default AK/SK configuration was read from ServiceStage, too many objects were initialized, causing memory leakage.

v1.1.0	• There were too many heartbeat information logs. • SSL calling between microservices did not take effect. • The request was abnormal when the URL contained spaces.
v1.0.0	Automatic service discovery cannot be performed in some scenarios.
v0.0.3	• The service discovered instances in the down state. • The configured path was too long. • The microservice failed to connect to the local CSE engine.

Java Chassis

Java Chassis Version	Resolved Issues
3.1.0	• "UnsupportedOperationException" was reported when the @RestController annotation class was scanned. • idleTimeout was incorrectly set. As a result, some connections were closed.
3.0.2	• When multiple identical keys existed in the configuration center, the updated configuration did not take effect. • The node information was not correctly destroyed. As a result, the request was abnormal.
3.0.1	Fallback label-based routing was supported.
2.8.16	idleTimeout was incorrectly set. As a result, some connections were closed.
2.8.15	When multiple identical keys existed in the configuration center, the updated configuration did not take effect.
2.8.14	• The request context was lost. • An exception occurred when SCBEngine was shut down. • When the content was empty, buffer reader was incorrect.
2.8.13	• The connection timed out, and the metric statistics were incorrect. • SDK was abnormal when the attachment to be uploaded was empty.

2.8.12	• The registry center frequently changed microservices. As a result, the newly deployed instance cannot be found and the cache instance was still used. • Vertx was occasionally shut down abnormally, and the blocking service was shut down normally.
2.8.11	When DefaultHttpClientFilter failed to deserialize the response body, the exception information was incorrect.
2.8.10	• **Content-Type** set for ResponseEntity did not take effect. • The registry center frequently changed microservices. As a result, the newly deployed instance cannot be found and the cache instance was still used.
2.8.9	• The bean of the service startup registry class was initialized twice. • The configuration was compatible with the registry center configuration of engine 1.x. • The cache instance information was not updated during concurrent registry.
2.8.8	• Microservice routing should obtain attributes from instances to determine the route distribution policy. • An error occurred when HttpUtils parsed Content-Disposition. As a result, ReadStreamPart cannot obtain the file name.
2.8.7	The available AZ was null by default.
2.8.6	• Configuration item **keep-alive** of the HTTP client was modified. • The microservice version number was obtained during concurrent access.
2.8.5	If the number of consecutive errors was 0 or fewer, the error percentage cannot take effect.
2.8.4	The request was incorrect because the instance change time was abnormal.
2.8.3	The definition of **caseInsensitive** in the routing rule was modified.
2.8.2	• During the effective matching of a governance rule, the apiPath matching was incorrect. • An exception occurred when the **body** parameter in the MVC request was null.
2.8.1	The load isolation switch did not take effect.

2.8.0	• **idleTimeout** and **keepAliveTimeout** were incorrect. • The highway protocol time in the edge gateway was incorrectly parsed. • A null pointer exception occurred on the rate limiting governance rule.
2.7.10	• The environment variables reported by the dashboard were incorrect. • When registry was unavailable, the cached service was cleared. As a result, the request was abnormal.
2.7.9	A null pointer exception occurred when autoDiscovery was enabled.
2.7.8	• A metadata memory leak occurred. • The findByContext method caused memory overflow. • PriorityInstancePropertyDiscoveryFilter generated debug logs.
2.7.6	• The ProducerOperationHandler error log recorded a wrong traceID. • The instance isolation filter did not take effect. • A null pointer exception occurred when the routing rule version was empty.
2.7.5	• Too many events were queued each time an instance was pulled to release events. • Span was empty in some scenarios of Zipkin. • The kie configuration center did not support filesource separation by commas (,).
2.7.3	• The dynamic configuration modification of dark launch release was invalid. • The request context combination logic conflicted.
2.7.0	IsolationDiscoveryFilter created too many objects.
2.6.3	• The retry was abnormal when a service failed to be selected for load balancing. • The calculation of retries did not take effect.
2.6.0	• During file download, the part content was empty. As a result, a null pointer exception occurred. • The empty string of the highway protocol was serialized. • The key pulled the configuration too fast in non-long pulling mode. • A null pointer exception occurred when the SDK was deployed for the first time after an engine was created. • The highway protocol was added and the default response body size limit was 20 MB.

2.5.2	The default value of **idleTimeout** for the ServiceComb client connection was changed from 60s to 30s.
2.5.1	An exception was thrown when the kie pulled an empty configuration for the first time.
2.5.0	RPC failed to be called in the vert.x work pool.
2.3.0 2.2.4 2.2.3 2.2.2 2.2.1	• After the configuration in the configuration center was updated, the rate limiting component was abnormal. • The custom executor failed to be added to the edge gateway service. • The AK/SK switch was on, and the encryption class failed to be obtained for the first time. • Placeholder were not supported. • slidingWindowType did not take effect for circuit breaker. • OOM occurred when a request was forwarded to a microservice that did not exist. • Error 404 occurred when a wrong address was retried. • The registration center failed to listen to the instance. • A deadlock occurred when a bean class was injected in XmlViewResolver and SPI modes at the same time. • After the configuration was changed, the dark launch route did not use the latest version. • The configuration change event was incorrectly triggered when **filesource** was configured. • The registry center address supported array placeholder configuration. • When the dashboard was enabled, the startup failed due to a null pointer exception.
2.2.0 2.1.6	• The governance configuration did not correctly process empty configurations and print incorrect operations. • RestTemplate failed to upload multiple files randomly. • When **deleteAfterSuccess** was set to **true**, the downloaded file in the servlet context failed to be deleted.
2.1.5	• The start failed when the main method did not contain the package name. • An exception occurred when the third-party service was registered and started. • An exception occurred when setting the response header for the HTTP2 request. • YAML was parsed insecurely. • The collection placeholders were parsed. • APP_MAPPING failed to parse the ServiceStage environment variables.

2.1.3	• The isolated instance was not restored after the response was correctly returned. • Asynchronous callback was triggered when the connection was closed due to timeout. • The service configuration and the default **spring value** did not take effect. • **servicecomb.references.version-rule** did not take effect. • Service isolation did not take effect due to timeout.
2.1.2	• The API remained unchanged and the JDK version was different. The schema reading sequence was inconsistent. As a result, the schema verification failed. • The service status was incorrect. As a result, the isolated instance cannot be called. • A null pointer exception occurred when the gateway read the microservice name.
2.1.1	• When the native parameters were empty, a null pointer exception was thrown during highway protocol parsing. • When the connection timed out, the obtained connection time was 0. • The **yaml** and **properties** configurations cannot be correctly processed. • RegistryUtils API did not use the version rule parameters.
2.1.0	• OOM occurred because the instance information failed to be set continuously. • The client label in the microservice metadata was incorrect. • A service calling exception occurred on the instances of different versions and APIs. • A request exception occurred when the highway protocol contained an **appId**.
2.0.2	• The jackson conversion overwrote all objects. • The kie parsing configuration was incorrect. • The version number query configuration was added to kie.
2.0.1	• **idleTimeoutInSeconds** in HTTP2 was not used. • A null pointer exception occurred when **@RequestHeader(value ="xxx")** and the aggregation parameters were used at the same time. • A request exception occurred when the interface parameter was defined as an object.

1.3.11	• Configuration item **keep-alive** of the HTTP client was modified. • If the number of consecutive errors was 0 or fewer, the error percentage cannot take effect.
1.3.10	• The load isolation switch did not take effect. • An exception occurred when setting the response header for the HTTP2 request.
1.3.9	A null pointer exception occurred on the rate limiting governance rule.
1.3.8	The instance isolation filter did not take effect.
1.3.7	IsolationDiscoveryFilter created too many objects.
1.3.5	• **idleTimeout** of the request client did not take effect. • The server was suspended and there was no stack information.
1.3.3	• YAML was parsed insecurely. • The isolated instance was not restored after the response was correctly returned.
1.3.2	
1.3.1	• The API remained unchanged and the JDK version was different. The schema reading sequence was inconsistent. As a result, the schema verification failed. • A null pointer exception occurred when **@RequestHeader(value ="xxx")** and the aggregation parameters were used at the same time. • The service status was incorrect. As a result, the isolated instance cannot be called. • The CPU was overloaded when there were too many instances. • The file upload was abnormal. As a result, no response was returned. • The default status was reset when the registry center was re-registered due to an exception. • SwaggerProducerOperation printed sensitive information logs.

1.3.0	• No exception was returned for FallbackPolicy. • CseAsyncRestTemplate did not support header settings. • RSAProviderTokenManager caused a memory leak. • ServiceCombServerStats.getFailedRate threw **ArithmeticException: /by zero**. • The swagger failed to generate a schema when a rewriting method existed. • When the RPC calling method had multiple parameters and the first parameter was **Object**, the second parameter was empty.
1.2.0	• The @ConfigurationProperties annotation was not supported. • HTTP2 did not support file download. • When **acceptType** existed on the client, the file download was abnormal. • When the startup failed, the destruction method overwrote the original exception information. • An exception occurred when metrics-prometheus adapted to prometheus-2.x. • The CPU usage calculated by the framework was incorrect. • An exception occurred when the schema was empty. • The Spring MVC application developed based on ServiceComb failed to be started. • The collected information was incorrect when an exception occurred during REST calling. • A null pointer exception occurred when the return type was defined as **ResponseEntity<Void>**. • The AZ affinity was abnormal when the instance was empty. • The request policy was rejected when the thread queue was full. • httpClientResponse.exceptionHandler threw an exception.

1.1.0	• A service thread deadlock occurred after delayed error injection was enabled. • The latest service failed to be accessed. • The request was abnormal because unavailable services were not cleared in a timely manner. • The edge service did not correctly return the servlet response code. • The **mvn install** command was used for packaging in Ubuntu. • The sequence of parameters generated by **BeanParamAnnotationProcessor** was unstable. • In the callback scenario, the instance attribute of the old version was used. As a result, the request was incorrect. • The downloaded content was incorrect. • The @ApiResponse annotation lost the response type. • The encoding and decoding of request parameters were incorrect. • JSON parsing was incorrect and the return code was incorrect. • servicecomb.xx.xx did not take effect as a governance rule. • The request connection was suspended when the request path contained invalid characters. • The NotFoundException exception was thrown during internal class parsing.
1.0.0	• The traffic limiting policy did not take effect. • The configurations starting with **cse** and **servicecomb** were set at the same time. As a result, the configuration values cannot be obtained. • An exception occurred when RPC requested the void method. • After a client requested an unregistered service, the request still cannot find the server even if the service was registered again. • A null pointer was returned when an unregistered service as queried.

• 1.0.0-m2	• When Zuul was used as the gateway for routing, and the same microservice API was opened in REST and highway modes at the same time, the client cannot call the API. • When the service name or schema ID contained a period (.), the QPS handler was abnormal. • The weighted load policy initialization and stateless access were responded. • Files with Chinese names cannot be downloaded. • An exception occurred during re-registration when the service contained environment variables. • A null pointer exception was thrown when the producer implementation class did not have an implementation method. • A null pointer exception was thrown when the uploaded content was empty. • When the size of the file to be uploaded exceeded the upper limit on the server, the client returned an error response. • The edge gateway threw an exception when the server returned **Transfer-Encoding header**. • Concurrent service discovery occurred when concurrent requests were sent. • The edge gateway service covered all error codes and returned error code 502. • Graceful shutdown did not take effect in some scenarios.
0.5.0	• RSAConsumerTokenManager created an authentication token. • When a service was shut down, the service was not deregistered from the registry center. • Automatic service discovery did not take effect. • The edge gateway service did not correctly process the 404 request.
0.4.0	• When a servlet request timed out, the async call was not supported or a null pointer exception was thrown. • No response can be obtained when an exception occurred during calling. • The REST producer returned an error type.
0.2.0	• Re-registration cannot be performed in the retry thread. • When a schema failed to be registered, the logs did not contain the exception information. • When the service was started, a null pointer exception occurred during registry.

