Cloud Service Engine

Best Practices

Issue 01

Date 2024-10-15

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

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 CSE Best Practices

This document summarizes operation practices of ServiceComb engines and registry/configuration centers in Cloud Service Engine (CSE) in common application scenarios. It provides detailed solutions each practice, helping users easily use CSE in different application scenarios.

Table 1-1 Best practices of ServiceComb engines

Best Practices	Description
Hosting Spring Cloud Applications Using Spring Cloud Huawei SDK	**Spring Boot** and **Spring Cloud** are widely used to build microservice applications. The main purpose of using ServiceComb engines to host Spring Cloud applications is to replace open-source components with highly reliable commercial middleware to better manage and maintain the application system. The reconstruction process should minimize the impact on service logic. This operation is applicable to the following scenarios: ● Application systems developed based on Spring Boot do not have basic microservice capabilities. The application system integrates Spring Cloud Huawei to provide capabilities such as service registration and discovery and dynamic configuration management. ● Application systems developed based on the Spring Cloud open-source technology system, for example, Eureka for registration and discovery and Nacos for dynamic configuration, integrate Spring Cloud Huawei and use highly reliable commercial middleware to replace open-source middleware, reducing maintenance costs. ● Cloud native applications built based on other Spring Cloud development systems, such as Spring Cloud Alibaba and Spring Cloud Azure, are migrated to Huawei Cloud using Spring Cloud Huawei.

Best Practices	Description
Hosting a Java Chassis Application	**Java chassis** is an open-source microservice development framework managed by the Apache Software Foundation. It was first donated by CSE. Till now, hundreds of developers have contributed to the project. Compared with Spring Cloud, Java chassis provides the following functions: ● Flexible and high-performance RPC implementation. Based on open APIs, Java chassis provides unified description of different RPC development modes, standardizing microservice API management and retaining flexible usage habits of developers. Based on reactive, Java chassis implements efficient communication protocols such as REST and Highway, and is compatible with traditional communication protocols such as Servlet. ● Rich service governance capabilities and unified governance responsibility chain. Common microservice governance capabilities, such as load balancing, rate limiting, and fault isolation, can be used out of the box. In addition, a unified governance responsibility chain is provided to simplify the development of new governance functions.

Table 1-2 Best practices of the registry/configuration center

Connecting Spring Cloud Eureka Applications to Nacos Engines	This section uses a demo to demonstrate how to connect Spring Cloud Eureka applications to Nacos engines.

2 ServiceComb Engines

2.1 ServiceComb Engine Application Hosting

2.1.1 Hosting Spring Cloud Applications Using Spring Cloud Huawei SDK

2.1.1.1 Introduction

Scenario

Spring Boot and **Spring Cloud** are widely used to build microservice applications. The main purpose of using ServiceComb engines to host Spring Cloud applications is to replace open-source components with highly reliable commercial middleware to better manage and maintain the application system. The reconstruction process should minimize the impact on service logic. This operation is applicable to the following scenarios:

- Application systems developed based on Spring Boot do not have basic microservice capabilities. The application system integrates Spring Cloud Huawei to provide capabilities such as service registration and discovery and dynamic configuration management.
- Application systems developed based on the Spring Cloud open-source technology system, for example, Eureka for registration and discovery and Nacos for dynamic configuration, integrate Spring Cloud Huawei and use highly reliable commercial middleware to replace open-source middleware, reducing maintenance costs.
- Cloud native applications built based on other Spring Cloud development systems, such as Spring Cloud Alibaba and Spring Cloud Azure, are migrated to Huawei Cloud using Spring Cloud Huawei.

Applying Suggestions

Before using ServiceComb engines to host Spring Cloud applications, evaluate the reconstruction risks and workload based on the following suggestions:

- The principle of reconstruction is to implement the DiscoveryClient and PropertySource interfaces provided by Spring Cloud to provide functions such as registration, discovery, and dynamic configuration for Spring Cloud applications. These implementations are independent of service logic development and do not affect service logic when integrated with Spring Cloud Huawei. Spring Cloud open-source technology system, Spring Cloud Alibaba, and Spring Cloud Azure also comply with this design mode. Therefore, the reconstruction can be classified into the integration and replacement scenarios. Spring Boot applications without microservice capabilities need to integrate with only Spring Cloud Huawei. For Spring Cloud applications with microservice capabilities, use Spring Cloud Huawei to replace related components.
- In the replacement scenario, if the service system does not directly depend on APIs that implement components, you only need to remove the original dependency and add the Spring Cloud Huawei dependency during the replacement, which requires small workload. If the service system depends on a large number of APIs that implement components, the replacement workload increases. Based on the actual experience, service systems do not directly depend on the APIs that implement components.
- The third-party software compatibility issues are the most likely to occur during the reconstruction. When there are two different versions of third-party software, use the later version preferentially. For Spring Boot and Spring Cloud, use the latest version in the community and follow the version mapping of the community. For example, if Spring Cloud Hoxton.SR8 is used, use Spring Boot 2.3.5.RELEASE. Although Spring Cloud Hoxton.SR8 claims to support Spring Boot 2.2.x, most components are integrated with 2.3.5.RELEASE for testing. Keeping up with the version mapping of the community can greatly reduce compatibility issues. For the best practices of third-party software compatibility issues, see **Third-Party Software Version Management Policy**.
- Spring Cloud best matches ServiceComb engine 2.x. This best practice is based on ServiceComb engine 2.x. The only difference between ServiceComb engine 1.x and 2.x is that 1.x uses config-center as the configuration center and 2.x uses kie. Therefore, you can also refer to this best practice to reconstruct ServiceComb engine 1.x.

2.1.1.2 Access to a ServiceComb Engine

To use Spring Cloud Huawei to access ServiceComb engines, perform the following steps:

1. Add or modify component dependencies.
2. Add the CSE ServiceComb engine configuration to the configuration file **boostrap.yaml**.

For details, see **Connecting Spring Cloud Applications to ServiceComb Engines**. This section describes the precautions during the reconstruction, especially the precautions related to component dependency.

Assume that the original service systems are all Maven-based projects.

Step 1: Get Familiar with the POM Structure of the Original Service System

The POM structure of the Spring Cloud application system is classified into the following types:

- The first method is to use the public POM provided by Spring Boot or Spring Cloud as the parent. For example:

  ```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.5.RELEASE</version>
  </parent>
  ```

  Alternatively, ensure that the following dependencies are introduced to the project:

  ```
  <parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-build</artifactId>
    <version>2.2.3.RELEASE</version>
  </parent>
  ```

- The second method is to use the parent of the project instead of the public POM provided by Spring Boot or Spring Cloud as the parent. However, dependency management is introduced into the project. For example:

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

- Some application systems use both the first and second methods. They use the public POM provided by Spring Boot or Spring Cloud as the parent, and introduce dependency management.

Step 2: Modify Parent and Dependency Management to Avoid Conflicts with Third-Party Software

Modifying parent and dependency management is a key step to prevent third-party software conflicts.

1. Determine the Spring Cloud Huawei version, and then query the Spring Boot version and Spring Cloud version corresponding to the Spring Cloud Huawei version. You are advised to use the latest version of Spring Cloud Huawei. You can query the mapping Spring Boot version and Spring Cloud version on the **official Spring Cloud Huawei website**.
2. Compare the parent version of the current project with the Spring Boot version and Spring Cloud version that match Spring Cloud Huawei. If the parent version of the current project is earlier, change it to the Spring Cloud Huawei version. Otherwise, no modification is required.
3. Independently introduce the dependency management of Spring Boot, Spring Cloud, and Spring Cloud Huawei to the dependency management of the current project. If the Spring Boot or Spring Cloud of the original project is of

a later version, use the version of the original project. Otherwise, use the Spring Cloud Huawei version. Pay attention to the sequence of dependency management. The dependency management in the front will be used first. Spring Boot and Spring Cloud versions are the basis. You are advised not to provide additional dependency management for the software managed by the two dependencies. You can follow the community version to effectively reduce conflicts. The three dependencies are introduced separately to facilitate the upgrade of a component.

```
<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>
```

If the service system integrates dependencies such as Spring Cloud Alibaba, delete the dependencies from dependency management. You are advised to delete unnecessary dependencies that have been managed by Spring Boot and Spring Cloud. Common dependencies that need to be deleted are as follows:

```
<dependencyManagement>
  <dependencies>
    <!-- Dependency of third-party extension -->
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>2.1.0.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
      </dependency>
      <!-- Dependency managed by Spring Cloud -->
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
        <version>1.4.7.RELEASE</version>
      </dependency>
    </dependencies>
</dependencyManagement>
```

Step 3: Add or Delete Dependencies

The components related to service registration and discovery, centralized configuration management, and service governance are added. The third-party implementations of the components are deleted. Other components do not need to be changed. However, after Spring Boot and Spring Cloud are upgraded, these components may need to be upgraded accordingly. Compatibility issues are usually found in the compilation phase or service startup phase.

Generally, you do not need to specify the version number when adding a dependency. The version number is managed by the parent and dependency management.

Introduce the following in microservice applications:

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

Introduce the following to the Spring Cloud Gateway application:

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

If the following dependencies exist, delete them:

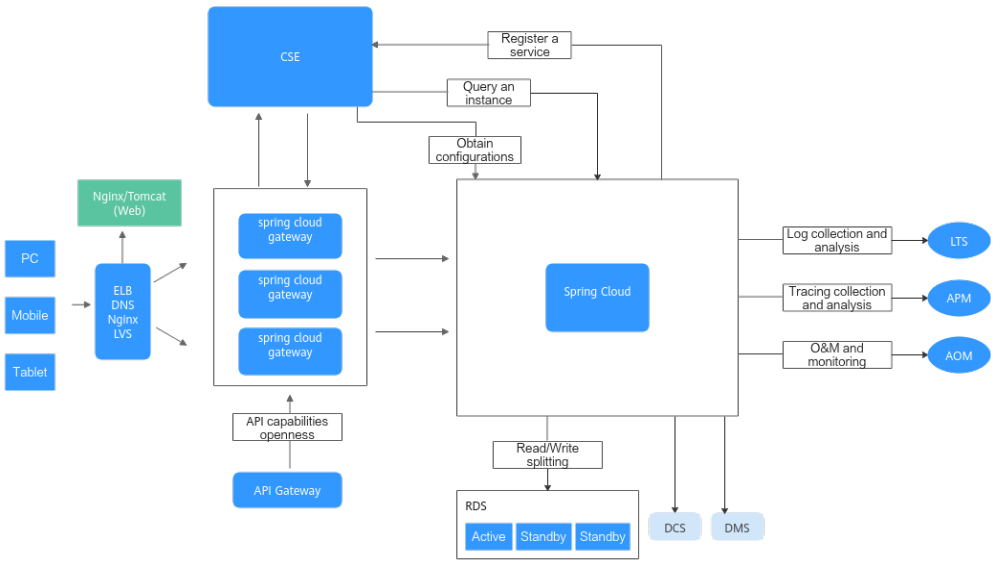
```
<!-- Nacos scenario-->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>spring-cloud-gateway-starter-ahas-sentinel</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>spring-boot-starter-ahas-sentinel-client</artifactId>
</dependency>

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

Generally, the following dependencies do not need to be deleted:

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.0.28</version>
</dependency>
```

2.1.1.3 System Architecture Planning

Spring Cloud provides various components to help build a resilient cloud-native system. Spring Cloud gateway has most capabilities of the common gateway and integrates the service governance capabilities of Spring Cloud to implement multi-protocol forwarding. The following figure shows a typical cloud-native Spring Cloud architecture.



This architecture separates static pages (web applications) from services so that static pages can be flexibly deployed in CDN or Nginx mode. Spring Cloud gateway shields the internal microservice structure and works with service governance policies such as rate limiting and security authentication so that internal services can be flexibly split and combined, reducing the risks of traffic attacks on internal services.

2.1.1.4 Third-Party Software Version Management Policy

During system upgrade and reconstruction, third-party software conflict is the most common issue. With the rapid development of software iteration, the traditional software compatibility management policy is no longer suitable for the development of software.

This section describes the best practices of third-party software management to help you build a continuously evolving application system.

Open-Source Software Selection

Major open-source communities, such as **Spring Boot** and **Spring Cloud**, maintain multiple versions. For example, Spring Cloud has versions such as Hoxton, Greenwich, and 2020.0.x, most of which are no longer maintained. Most open-source software has two versions: one is the latest development version, and the other is the recent maintenance version.

For open-source software selection, follow the version development pace of the community and use the latest maintenance version. There is no strict conclusion on the selection of development and maintenance versions. The selection depends on the specific product functions. For example, if the product competitiveness depends heavily on the features of a third-party software, the development version is preferred. If the product depends on stable features of a third-party software and does not use new functions, the maintenance version is preferred.

You are not advised to select a version that is no longer maintained by the community or a version that is still under maintenance but has been released for more than half a year. Although no function issues are found in these versions, the continuous evolution of the product will be severely affected:

1. Software security vulnerabilities cannot be handled in a timely manner. The discovery and exploitation of security vulnerabilities take a certain period of time. If the earlier version is used for a long time, security vulnerabilities are more likely to be exploited, making the system more vulnerable to attacks.
2. When the system is faulty, it is more difficult to seek community support. Versions that are no longer maintained or have been released for more than half a year can hardly be supported.
3. System evolution becomes more difficult. When new features need to be added to the system and new development tools need to be introduced, it is more difficult for earlier versions to be compatible with the new development tools.
4. Many faults may have been rectified in later versions, and the code maintainability and performance of later versions are better than those of earlier versions.

Therefore, the best solution to open-source software selection is to select the development or maintenance version provided by the community for maintenance and upgrade, drive the upgrade to the latest version based on the issue, and periodically upgrade to the latest version every quarter.

Third-Party Software Version Management

The following uses an example to describe the principle of third-party component conflict. Assume that project X needs to reference the components provided by both project A and project B. Project A and project B depend on project C, but the version numbers of the two projects are different.

- POM of project X
```
<dependency>
  <groupId>groupA</groupId>
  <artifactId>artifactA</artifactId>
   <version>0.1.0</version>
</dependency>
<dependency>
  <groupId>groupB</groupId>
  <artifactId>artifactB</artifactId>
  <version>0.1.0</version>
</dependency>
```
- POM of project A
```
<dependency>
  <groupId>groupC</groupId>
  <artifactId>artifactC</artifactId>
  <version>0.1.0</version>
</dependency>
```

- POM of project B

```
<dependency>
  <groupId>groupC</groupId>
  <artifactId>artifactC</artifactId>
  <version>0.2.0</version>
</dependency>
```

When project X is finally released, the following situations may occur:

- Version 0.2.0 of project C is used. Because project A is compiled and tested using version 0.1.0, component A may not work properly. For example, version 0.2.0 is incompatible with version 0.1.0, and project A uses these incompatible interfaces.
- Version 0.1.0 of project C is used. Because project B is compiled and tested using version 0.2.0, component B may not work properly. For example, project B uses the new interfaces provided by version 0.2.0.

If the interface of component C used by project A is incompatible with that used by project B, project X cannot work properly no matter how the interface is adjusted. The code of project A must be modified and tested using the same or compatible version as that of project B. A new version must be released for project X.

Therefore, the best policy for dependency management is to ensure the dependency of common components and use a later version. However, there are often a number of issues, especially when the project dependencies are very complex.

Currently, the dependency management mechanism is used to manage the dependency of mainstream complex projects. Dependency management has been proved to be an effective method for managing dependencies and therefore is widely used in open-source communities. For example, for Spring Boot, Spring Cloud, and Spring Cloud Huawei, you can view the source code directory structure of Spring Cloud Huawei to learn about how to use dependency management.

For a complex project, for example, the Spring Cloud Huawei project, the POM file related to dependency management includes:

```
/pom.xml # Root directory of the project.
/spring-cloud-huawei-dependencies/pom.xml # Main dependency of the project . All the
statements related to the dependency management are in this file.
/spring-cloud-huawei-parents/pom.xml # parents is used by sub-modules of the project and
project developers, which is similar to the parent provided by Spring Boot
/spring-cloud-huawei-bom/pom.xml # BOM is used when project developers expect to
introduce the components provided by Spring Cloud Huawei into the dependency management
of the project instead of introducing the third-party software versions on which Spring Cloud
Huawei depends.
```

The POMs of Spring Cloud Huawei are relatively complete. They provide developers with POMs that can be introduced from different perspectives and are applicable to common development components.

Generally, a microservice development project may contain only the **/pom.xml** file. The parent and dependency management of the project are declared in the **/pom.xml** file. After dependency management is introduced, all dependency statements in the project do not specify the version number. In this way, when the third-party software version needs to be upgraded, you only need to modify dependency management in the **/pom.xml** file.

You can use the **Spring Cloud Huawei samples** to understand the principle and function of dependency management:

1. Run the **mvn dependency:tree** command to check the project dependency.
2. Modify **spring-boot.version** in the **/pom.xml** file and run the **mvn dependency:tree** command to check the dependency relationship changes of the project.
3. Adjust the positions of **spring-boot-dependencies** and **spring-cloud-dependencies** in the **/pom.xml** file, and run the **mvn dependency:tree** command to check the dependency relationship changes of the project.

When the number of third-party software on which a project depends increases, it is difficult to identify the mapping between software. In this case, you can follow the version mapping of Spring Boot and Spring Cloud. It is a good choice to use **spring-boot-dependencies** and **spring-cloud-dependencies** as the basis for dependency management. Because Spring Boot and Spring Cloud are widely used, the community can fix compatibility issues in a timely manner. Developers only need to upgrade Spring Boot and Spring Cloud versions and do not need to pay attention to the versions of other third-party software on which Spring Boot and Spring Cloud depend.

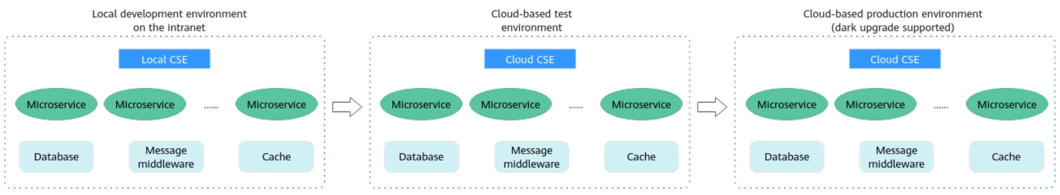
Upgrading third-party software can promote continuous software improvement, but engineering capabilities, such as automatic test capabilities, need to be improved.

2.1.1.5 Development Environment Planning

The purpose of planning the development environment is to ensure that developers can better work in parallel, reduce dependencies, reduce the workload of environment setup, and reduce the risks of bringing the production environment online.

The purpose of managing the development environment is to better develop, test, and deploy services.

Figure 2-1 Development environment



Based on the project experience, the development environment is planned according to **Figure 2-1**:

- Set up a local development environment on the intranet. The advantage of the local development environment is that each service or developer can set up a minimum function set environment that meets their requirements to facilitate log viewing and code debugging. The local development environment greatly improves the code development efficiency and reduces the deployment and debugging time. The disadvantage of local development environment is the low integration. When the integration and joint commissioning are required, it is difficult to ensure environment stability.

- The cloud-based test environment is a relatively stable integration test environment. After the local development and test are complete, each service domain deploys the services in its own domain to the cloud test environment and can invoke services in other domains for integration tests. Based on the service scale, the cloud test environment can be further divided into the α, β, and γ test environments. These test environments are integrated in ascending order. Generally, the γ test environment must be managed in the same way as the production environment to ensure environment stability.
- The production environment is a formal service environment. It needs to support dark upgrade, online joint commissioning, and traffic diversion to minimize the impact of upgrade faults on services.
- In the cloud-based test environment, the public IP addresses of CSE and middleware can be opened, or network interconnection can be implemented. In this way, the middleware on the cloud can be used to replace the local environment, reducing the time for developers to install the environment. This situation also belongs to the local development environment on the intranet where microservices run in the local development environment. Microservices deployed in containers on the cloud and those deployed on the local development environment cannot access each other. To avoid conflicts, the cloud-based test environment is used only as the local development environment.

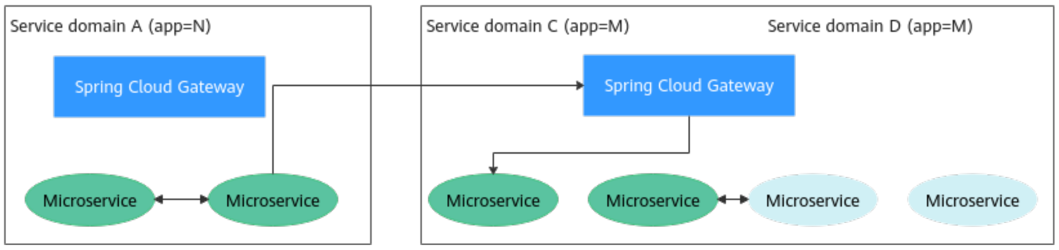
2.1.1.6 Application Logical Isolation

Application logical isolation is used in scenarios where different development environments share public CSE resources to reduce cost. Logical isolation is also used to manage the relationships between microservices. With proper isolation policies, the accessibility and permissions between microservices can be better controlled.

Service Discovery

Microservices in different service domains are isolated by applications.

Different service domains use different application names. Services in the same service domain can discover each other and access each other in point-to-point mode. Services in different service domains cannot discover each other. They need to access each other through Spring Cloud Gateway in the service domain where the microservice to be accessed is located.



Dynamic Configuration

Dynamic configuration is managed at the public, application, and service layers.

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, **custom_tag** and **custom_value** can be used to define configurations. For example, if some configurations are shared by all applications, this method can be used. Add the following configurations to the configuration file:

```
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. The configuration center can be described as follows:

1. 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.
2. The client queries the configuration based on the following search criteria:
   - Custom configuration

     select * from tbl_configurations where custome_label=custome_label_value & withStrict=false
   - Application-level configuration

     select * from tbl_configurations where app=demo_app & environment=demo_environment & withStrict=true
   - Service-level configuration

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

   When **withStrict** is set to **true**, only the attributes specified in the condition are available. When **withStrict** 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.

2.1.1.7 Configuration File Encryption Scheme

The configuration file often contains sensitive information, such as account and passwords. In this case, the sensitive information needs to be encrypted to ensure security.

This section describes how to use jasypt-spring-boot-starter to encrypt data. The account names and passwords involved in RBAC authentication are used as examples.

1. Add the dependency corresponding to the encryption component to the POM file.
   ```
   <dependency>
     <groupId>com.github.ulisesbocchio</groupId>
     <artifactId>jasypt-spring-boot-starter</artifactId>
     <version>2.1.2</version>
   </dependency>
   ```
2. Configure the password.
   – You can directly configure the password in the configuration file (for example, **application.properties**). However, this method is not recommended because it is insecure.
     ```
     jasypt.encryptor.password=******
     ```
     Set ****** to the password used for encryption.
   – Set the password in the JVM startup parameter.
     ```
     -D jasypt.encryptor.password=******
     ```
     Set ****** to the password used for encryption.
3. Implement the encryption method.
   ```
   // Set this parameter to the password of the jasypt.encryptor.password configuration item.
     public static String salt = "GXXX6" (user-defined);

    // Encryption method.
     public static String demoEncrypt(String value) {
         BasicTextEncryptor textEncryptor = new BasicTextEncryptor();
         textEncryptor.setPassword(salt);
         return textEncryptor.encrypt(value);
     }

    // Test whether the decryption is normal.
     public static String demoDecrypt(String value) {
         BasicTextEncryptor textEncryptor = new BasicTextEncryptor();
         textEncryptor.setPassword(salt);
         return textEncryptor.decrypt(value);
     }

     public static void main(String[] args) {
         String username = demoEncrypt("root");
         System.out.println(username);
         System.out.println(username);
     }
   ```
   The default encryption method of jasypt is used. You can also customize extended encryption and decryption methods. For details, see the **official jasypt document**.
4. Use the encrypted configuration item.

   You can use either of the following methods:
   – Write the configuration file
     ```
     spring:
       cloud:
         servicecomb:
            credentials:
               account:
                  name: ENC (ciphertext of the account name)
                  password: ENC (ciphertext of the password)
     ```
     Ciphertexts of the account name and password are obtained in **3**.

NOTE

This encryption mode requires the ENC() flag to identify whether encryption is enabled. ENC() is the special mark of the encryption mode. If ENC() does not exist, the plaintext is used.

– Enter environment variables

```
spring_cloud_servicecomb_credentials_account_name = ENC (ciphertext of the account name)
spring_cloud_servicecomb_credentials_account_password = ENC (ciphertext of the password)
```

Ciphertexts of the account name and password are obtained in **3**.

2.1.1.8 Service Governance Planning

2.1.1.8.1 Rolling Upgrade

You are advised to use ServiceStage to deploy Spring Cloud applications, which facilitates rolling upgrade.

When using ServiceStage to deploy applications, you can configure the liveness probe and service probe of a component by referring to **Configuring Health Check** to check the Liveness and Ready statuses of microservices.

Spring Boot provides out-of-the-box container probes, LivenessStateHealthIndicator, and ReadinessStateHealthIndicator.

To configure a probe, you need to enable the spring-cloud-starter-huawei-actuator function.

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

By default, LivenessStateHealthIndicator and ReadinessStateHealthIndicator do not contain any other health check items. Spring Cloud Huawei provides health check. When a service is registered successfully, **true** is returned. This check can be included in ReadinessStateHealthIndicator:

```
management.endpoint.health.group.readiness.include=registry
```

Then, set the service probe of the component as **Table 2-1**. After configuration and the service is successfully registered, ServiceStage displays the service ready status. During the rolling upgrade, the old instance is stopped only after the instance is successfully registered.

Table 2-1 Component service probes

Parameter	Mandatory	Description
Path	Yes	Request URL, for example, /actuator/health/readiness.
Port	Yes	Microservice port.

Parameter	Mandatory	Description
Latency (s)	No	Detection start time. For microservices that take a long time to start, you can prolong the time.
Timeout Period (s)	No	After the detection starts, if the probe status is not detected within the period specified by this parameter, the detection fails.

NOTE

This module is provided only in Spring Cloud Huawei 1.9.0-Hoxton, 1.9.0-2020.0.x, and later versions.

In addition to setting probes, you need to set the rolling upgrade policy. The core parameters include the **Max. Unavailable Pods**. The default values of these two parameters are both 0. If there is only one instance, the rolling upgrade will be interrupted. You are advised to set 0 ≤ **Max. Unavailable Pods**.

2.1.1.9 FAQs

2.1.1.9.1 Incompatibilities During Spring Boot Upgrade from 2.0.x.RELEASE to 2.3.x.RELEASE

FeignClient Name Issue

- Description

  In earlier Spring Boot versions, Bean name can be overwritten. In new versions, this function is disabled by default and can be enabled by using the configuration item.
- Solutions

  Configure the following to enable overwriting:

```
spring:
  main:
    allow-bean-definition-overriding: true
```

Spring Data API Change

- Description

  The Spring Data API fluctuates.
- Solutions

  Use the new API to modify the code. For example, change **new PageImpl** to **PageRequest.of** and **new Sort** to **Sort.of**.

JPA Change: Multiple Entities Correspond to One Table

- Description

  In later versions, one entity corresponds to one table.

- Solutions

  Currently, there is no solution. Alternatively, adjust the code structure to the constraints of the new version.

Mongo Client Upgrade Change

- Description

  The MongoDbFactory API has changed and needs to be adjusted to the new version.

- Solutions

```
@Bean
public MappingMongoConverter mappingMongoConverter(MongoDbFactory factory,
MongoMappingContext context, BeanFactory beanFactory) {
   DbRefResolver dbRefResolver = new DefaultDbRefResolver(factory);
   MappingMongoConverter mappingConverter = new
MappingMongoConverter(dbRefResolver, context);

mappingConverter.setCustomConversions(beanFactory.getBean(MongoCustomConversions.
class));
   // other customization
   return mappingConverter;
}

@Bean
public MongoClientOptions mongoOptions() {
       return
MongoClientOptions.builder().maxConnectionIdleTime(60000).socketTimeout(60000).buil
d();
}
```

2.1.1.9.2 Dynamic Configuration Issues

Selecting Dynamic Configuration Type

Microservice engine 2.0's configuration center supports multiple formats such as TEXT and YAML.

- Simple key-value configuration

  Use TEXT format. The keys in the configuration center and code are the same.

- Many configurations

  Use YAML format. Ignore the configuration center key. All key-value pairs are defined in the YAML file.

  ServiceComb engine 1.x does not support YAML, but Spring Cloud Huawei does. Add the following configuration to the microservice Bootstrap:

```
spring:
  cloud:
    servicecomb:
      config:
        fileSource: consumer.yaml # List of configuration items that need to be parsed based
on YAML. Use commas (,) to separate multiple configuration items.
```

- Initial use of ServiceComb engine 2.x

  You are advised to select the latest version of Spring Cloud Huawei, which contains more features and has been optimized based on historical issues.

Binding List Object Configuration

Some services use the list object configuration binding. For example:

```
@ConfigurationProperties("example.complex")
public class ComplexConfigurationProperties {
  private List<String> stringList;
  private List<Model> modelList;
  ... ...
}
```

For the list object, Spring Cloud queries related configuration items from only one PropertySource by default. If a PropertySource has some values of the configuration items, Spring Cloud does not query other values. For such services, ensure that configurations related to these list attributes have been placed in the configuration center. Do not place some elements in the configuration file or other elements in the configuration center.

This restriction is due to the atomicity of the list configuration – a configuration item (stringList or modelList in the code example) cannot be separated in different configuration files.

2.1.1.9.3 Incorrect Registry Center Address

Description

When Spring Cloud Huawei is used, the following error is displayed during microservice startup:

```
send request to https://192.168.10.1:30100/v4/default/registry/microservices failed and retry to
https://192.168.10.1:30100/v4/default/registry/microservices once.
org.apache.http.conn.HttpHostConnectException: Connect to 192.168.10.1:30100 [/127.0.0.2] failed:
Connection refused: connect
at
org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOpera
tor.java:156) ~[httpclient-4.5.13.jar:4.5.13]
at
org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionMana
ger.java:376) ~[httpclient-4.5.13.jar:4.5.13]
```

Analysis

The error is reported when the microservice registry center address is unavailable.

Solutions

- Start the service and deploy it on the local host

  On the local host, run the **curl https://***IP address of the registry center***:30100/health** command to check the working status of the registry center. Check whether information similar to the following is displayed:

  ```
  curl: Failed to connect to xxx.xxx.xxx.xxx port 30100: Connection refused
  ```

  If yes, check whether the network is disconnected because the IP address or port number of the registry center is incorrect or the network is isolated.

- Start service deployment on the microservice engine on the cloud

  The microservice is deployed on the microservice engine through ServiceStage. The registry center address can be automatically injected using environment variables. Check whether the address of the injected registry center is correct. If no, correct it and deploy the service again.

2.1.1.9.4 Different Services of the Same Application in the Same Environment Cannot Invoke Each Other

Description

The exclusive microservice engine with security authentication enabled is loaded in the environment where services of the same application are deployed, however, different services use different accounts. As a result, these services of the same application cannot discover each other and cannot invoke each other.

Solutions

Grant the account invoking this service all permissions on this service and the read-only permission on other services.

For details, see **System Management**.

2.1.2 Hosting a Java Chassis Application

2.1.2.1 Introduction

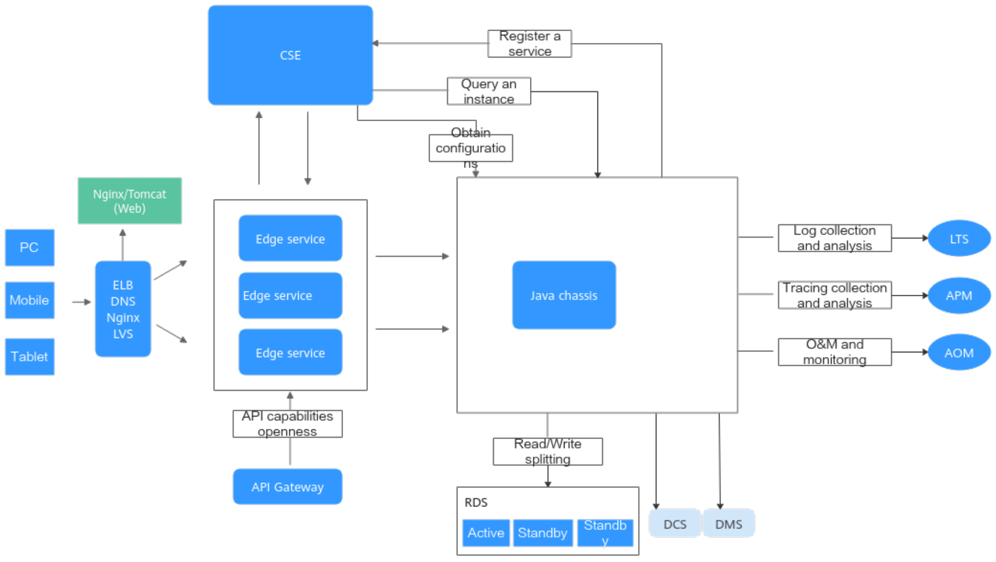
Java chassis is an open-source microservice development framework managed by the Apache Software Foundation. It was first donated by CSE. Till now, hundreds of developers have contributed to the project. Compared with Spring Cloud, Java chassis provides the following functions:

- Flexible and high-performance RPC implementation. Based on open APIs, Java chassis provides unified description of different RPC development modes, standardizing microservice API management and retaining flexible usage habits of developers. Based on reactive, Java chassis implements efficient communication protocols such as REST and Highway, and is compatible with traditional communication protocols such as Servlet.
- Rich service governance capabilities and unified governance responsibility chain. Common microservice governance capabilities, such as load balancing, rate limiting, and fault isolation, can be used out of the box. In addition, a unified governance responsibility chain is provided to simplify the development of new governance functions.

Like Spring Cloud, Java chassis can also use Spring and Spring Boot as basic components for application development. However, Java chassis provides independent RPC implementation. Therefore, functional components that depend on Spring MVC are restricted, for example, Spring Security.

2.1.2.2 System Architecture Planning

Java chassis provides various components to help build a resilient cloud-native system. The edge service has most capabilities of the common gateway and integrates the service governance capabilities of Java chassis to implement multi-protocol forwarding. The following figure shows a typical cloud-native Java chassis architecture.



This architecture separates static pages (web applications) from services so that static pages can be flexibly deployed in CDN or Nginx mode. The edge service shields the internal microservice structure and works with service governance policies such as rate limiting and security authentication so that internal services can be flexibly split and combined, reducing the risks of traffic attacks on internal services.

2.1.2.3 Thread Pool Parameters Configuration

Thread pools are the main service processing units of microservices. Proper thread pools planning can maximize system performance and prevent the system from failing to provide services for normal users due to exceptions. The optimization of the thread pool is closely related to the service performance. The parameter settings vary according to scenarios. The following describes two scenarios. Before setting, you need to check the service performance, test common APIs, and check the latency.

- The service performance is good.

  That is, in non-concurrent scenarios, the average API latency is less than 10 ms.

  When the service performance is good, to make the service system more predictable and prevent the JVM garbage collection, network fluctuation, and burst traffic from affecting the system stability, the system needs to quickly discard requests and take measures such as retry to ensure that the system performance is predictable in the case of fluctuation and normal service running.

  – Number of connections and timeout settings

```
# Number of verticle instances on the server. Retain the default value. It is recommended that this parameter be set to a value in the range of 8 to 10.
servicecomb.rest.server.verticle-count: 10
# Maximum number of connections. The default value is Integer.MAX_VALUE. The maximum value can be estimated based on the actual situation so that the system has better resilience.
servicecomb.rest.server.connection-limit: 20000
# Connection idle time. The default value is 60s. Generally, you do not need to change the value.
servicecomb.rest.server.connection.idleTimeoutInSeconds: 60
# Number of verticle instances on the client. Retain the default value. It is recommended that this parameter be set to a value in the range of 8 to 10.
servicecomb.rest.client.verticle-count: 0
```

```
# Maximum number of connections between a client and the server is verticle – count * maxPoolSize, which cannot exceed the number of threads.
#In this example, the number of connections is 500 (10 x 50). If there are a large number of instances, reduce the number of connections of a single instance.
servicecomb.rest.client.connection.maxPoolSize: 50
# Connection idle time. The default value is 30s. Generally, you do not need to change the value. The value must be shorter than the connection idle time of the server.
servicecomb.rest.client.connection.idleTimeoutInSeconds
```

– Service thread pool configuration

```
# Number of thread pool groups. The recommended value is 2 to 4.
servicecomb.executor.default.group: 2
# Recommended value range: 50–200
servicecomb.executor.default.thread-per-group: 100
# Size of a queue in the thread pool. The default value is Integer.MAX_VALUE. Do not use the default value in high-performance scenarios to quickly discard requests.
servicecomb.executor.default.maxQueueSize-per-group: 10000
# Maximum waiting time of a queue. If the waiting time exceeds the maximum value, the request is discarded and a response is returned. The default value is 0.
# In high-performance scenarios, set the queuing timeout interval to a small value to quickly discard requests.
servicecomb.rest.server.requestWaitInPoolTimeout: 100
# Set a short timeout period to quickly discard requests. However, you are advised to set the timeout period to a value greater than or equal to 1s. Otherwise, many problems may occur.
servicecomb.request.timeout=5000
```

- The service performance is not good.

  That is, in non-concurrent scenarios, the average API latency is longer than 100 ms. High latency is usually caused by low CPU usage due to I/O and resource waiting in service code. If the high latency is caused by complex calculation, the optimization becomes complex.

  When the service performance is not good, you need to increase the values of the following parameters. Otherwise, a large number of services will be blocked. Increasing these parameters ensures the system throughput and avoids service failures caused by burst traffic. However, user experience will be affected.

```
# Server connection idle time.
servicecomb.rest.server.connection.idleTimeoutInSeconds: 120000
# Client connection idle time.
servicecomb.rest.client.connection.idleTimeoutInSeconds: 90000
# Number of thread pool groups.
servicecomb.executor.default.group: 4
# Size of the thread pool.
servicecomb.executor.default.thread-per-group: 200
# Size of the queue in the thread pool. Threads will be queued when the performance is not good.
servicecomb.executor.default.maxQueueSize-per-group: 100000
# Set the timeout period to a large value.
servicecomb.rest.server.requestWaitInPoolTimeout: 10000
servicecomb.request.timeout=30000
```

2.1.2.4 Log Files Configuration

Viewing error logs is an important method for locating faults. Therefore, you need to properly plan log output to minimize the impact on system performance. Suggestions for planning log files:

1. Use **log4j2** or **logback** to output logs. Output logs to a file without depending on stdout of the container.
2. Open the **metrics** log, export this log to an independent file, for example, **metrics.log**, and export service logs to another file, for example, **servicecomb.log**. Configure **metrics** parameters as follows:

```
servicecomb:
  metrics:
```

```
  window_time: 60000
  invocation:
    latencyDistribution: 0,1,10,100,1000
  Consumer.invocation.slow:
    enabled: true
    msTime: 3000
  Provider.invocation.slow:
    enabled: true
    msTime: 3000
  publisher.defaultLog:
    enabled: true
    endpoints.client.detail.enabled: true
```

3. Open **access log** and export it to an independent log file.
4. The service log containing **trace ID** is printed in a formatted manner. You can develop a handler and configure it before **Provider Handler**. After receiving and processing a request, the handler prints logs respectively. This helps you locate faults and quickly search for related logs using AOM.

2.1.2.5 Service Governance Planning

2.1.2.5.1 Rolling Upgrade

You are advised to use ServiceStage to deploy Java chassis applications, which facilitates rolling upgrade.

When using ServiceStage to deploy applications, you can configure component service probes so that ServiceStage can correctly detect microservice statuses. To configure the component service probe, you need to enable the metrics function and set the component service probe path to **/health**.

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>metrics-core</artifactId>
</dependency>
```

In addition to setting probes, you need to set the rolling upgrade policy. The core parameters include the **Max. Unavailable Pods**. The default values of these two parameters are both 0. If there is only one instance, the rolling upgrade will be interrupted. You are advised to set **Max. Unavailable Pods** ≥ 2, 0 ≤ **Max. Unavailable Pods** < Pods–1. That is, at least 2 instances are available.

2.1.2.5.2 Hitless Upgrade

To achieve a hitless upgrade, the following problems need to be solved:

1. Service interruption during stopping a service. During the process of stopping a service, the service may be processing requests, and new requests may be continuously sent to the service.
2. In the microservice architecture, service discovery is usually performed through the service center. The client caches the instance address. Access failure. This is because when the service is stopped, users may not be aware that the instance is offline in a timely manner and continue to use the incorrect instance for access.
3. Rolling upgrades. The old version can be stopped only after the new version is ready.

Many measures are required to achieve a hitless upgrade, for example, **Rolling Upgrade**. Therefore, you are advised to ensure that at least two instances are available. Java chassis implements hitless upgrades:

1. Graceful shutdown: When the service is stopped, the system waits for the request to be completed and rejects the new request.

   Graceful shutdown is provided by default. Before a process exits, certain cleanup actions are performed, including waiting for the requests that are being processed to complete, rejecting new requests that are not in the processing queue, and invoking the registry center API to deregister the process. Before exiting a Java chassis process, change the instance status to DOWN and wait for a period of time.

```
servicecomb:
  boot:
    turnDown:
      # Wait time after the instance status is changed to Down. The default value is 0, indicating no waiting.
      waitInSeconds: 30
```

2. Retry: If the client fails to connect to the network or rejects the request, a new server needs to be selected for retry.

   Enable the retry policy.

```
servicecomb:
  loadbalance:
    retryEnabled: true # Whether to enable the retry policy.
    retryOnNext: 1  # Number of retry times for searching for an instance (different from a failed instance; depending on the load balancing policy)
    retryOnSame: 0  # Number of retries on the failed instance.
```

3. Isolation: Service instances that fail to be processed for a specified number of times are isolated.

   Enable the instance isolation policy.

```
servicecomb:
  loadbalance:
    isolation:
      enabled: true
      enableRequestThreshold: 5 # Minimum number of successful and failed requests processed by the instance in a statistical period.
      singleTestTime: 60000 # Time after which the system attempts to access the instance isolated. If the access is successful, isolation will be canceled. Otherwise, isolation will continue.
      continuousFailureThreshold: 2 # Condition for isolating a instance: the instance fails to be isolated for two consecutive times.
```

2.1.2.6 Java Chassis Upgrade

Continuous version upgrades can better use new functions and features of CSE, fix known quality and security issues in a timely manner, and reduce maintenance costs.

Continuous version upgrades also bring some compatibility issues. Therefore, you are advised to include continuous upgrades to your plan.

In addition, automatic test capabilities need to be built for continuous upgrades to reduce the verification time and control the risks. Continuously building automation capabilities and upgrading versions are proven best practices for building high-quality software.

3 Registry/Configuration Centers

3.1 Connecting Spring Cloud Eureka Applications to Nacos Engines

This section uses a demo to demonstrate how to connect Spring Cloud Eureka applications to Nacos engines.

Connect a provider service and a consumer service to a Nacos engine.

Prerequisites

- You have created a Nacos engine. For details, see **Creating a Registry/Configuration Center**.
- You have downloaded the **demo source code** from GitHub to the local host and decompressed it.
- The Java JDK and Maven have been installed on the local host for compilation, building, and packaging, and the Maven central library can be accessed.

Restrictions

- Nacos is compatible with Eureka APIs on the Eureka server side and saves and updates client instance information registered on the service side. Therefore, if you use only Eureka as the registry center, many features of Nacos, such as namespace and configuration management, cannot be used.
- Eureka is used as the client and can be viewed only on the **Service Management** page of Nacos. Eureka services are displayed using the default attributes of Nacos.
  - Default namespace: **public**
  - Default group: **DEFAULT_GROUP**
- The value of **Protection Threshold** for creating a service on the **Service Management** page of Nacos is a feature of Nacos and does not apply to the Eureka service.

Connecting Spring Cloud Eureka Applications to Nacos Engines

Step 1 Log in to CSE.

Step 2 Obtain the registry center address of a Nacos engine.

1. In the left navigation pane, choose **Registry/Configuration Center** and click the Nacos engine instance.
2. In the **Connection Information** area on the **Basic Information** page, obtain the service center address.

Connection Information

Private IP	[redacted]	Private Port	8848,9848
Virtual Private Cloud	vpc-[redacted]	Subnet	lbcdzw

Step 3 Change the registry center address and microservice name in the demo.

1. Configure the Nacos service center address and microservice name in the **application.properties** file.
   - Find the **eureka-demo-master\eureka-consumer\src\main\resources\application.properties** file in the demo source code directory downloaded to the local host and configure the consumer service.

```
server.port=9001
spring.application.name= eureka-client-consumer //Microservice name
eureka.client.serviceUrl.defaultZone= XXX.nacos.cse.com:8848/nacos/eureka //Service center address of Nacos
eureka.instance.lease-renewal-interval-in-seconds=15 //Service heartbeat update interval
eureka.client.registry-fetch-interval-seconds=15 //Interval for pulling the registry center. It is recommended that the value be the same as the heartbeat interval.
```

   - Find the **eureka-demo-master\eureka-provider\src\main\resources\application.properties** file in the demo source code directory downloaded to the local host and configure the provider service.

```
server.port=9000
spring.application.name= eureka-client-provider //Microservice name
eureka.client.serviceUrl.defaultZone= XXX.nacos.cse.com:8848/nacos/eureka //Service center address of Nacos
eureka.instance.lease-renewal-interval-in-seconds=15 //Service heartbeat update interval
eureka.client.registry-fetch-interval-seconds=15 //Interval for pulling the registry center. It is recommended that the value be the same as the heartbeat interval.
```

Step 4 Pack the demo source code into a JAR package.

1. In the root directory of the demo source code, open the Command Prompt and run the **mvn clean package** command to package and compile the project.
2. After the compilation is successful, two JAR packages are generated, as shown in Table 3-1.

Table 3-1 Software packages

Software Package Directory	Software Package Name	Description
\eureka-consumer\target	eureka-client-consumer-1.0.0-SNAPSHOT.jar	Service consumer

Software Package Directory	Software Package Name	Description
\eureka-provider\target	eureka-client-provider-1.0.0-SNAPSHOT.jar	Service provider

Step 5 Deploy Spring Cloud applications.

Deploy provider and consumer on the ECS node in the VPC where the Nacos engine is located.

1. Create an ECS node in the VPC where the engine instance is located and log in to the ECS node. For details, see **Purchasing and Logging In to a Linux ECS**.
2. Install JRE to provide a running environment for services.
3. Upload the JAR package generated in **Step 4** to the ECS node.
4. Run the **java -jar** *{JAR package}* command to run the generated JAR package.

Step 6 Confirm the deployment results.

1. **Optional:** On the CSE console, choose **Registry/Configuration Center** and click the Nacos engine created in **Prerequisites**.
2. Choose **Service Management** and check the number of instances of microservices **eureka-client-consumer** and **eureka-client-provider**.
   - If **Instances** is not **0**, the demo has been connected to the Nacos engine.
   - If **Instances** is **0** or the **eureka-client-consumer** and **eureka-client-provider** services cannot be found, the demo fails to be connected to the Nacos engine.

----End