微服务引擎

最佳实践

文档版本 01

发布日期 2023-10-13

华为云计算技术有限公司

商标声明


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

注意

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

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

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 其他框架改造为 Spring Cloud 接入 CSE ServiceComb

方案概述

本文描述如何将HSF、Dubbo框架改造为Spring Cloud框架并接入ServiceComb引擎的操作。

- 应用场景

  很多微服务框架只是提供了如何解决微服务运维问题的功能模块和工具，但并没有帮用户解决那些问题，用户自行解决这些问题的成本通常非常高，出于现有框架的使用成本和问题，以及对未来业务的发展是否需要选择更加合适的技术考虑，可将微服务框架进行迁移。

- 方案架构

  a. 将HSF、Dubbo框架改造为Spring Cloud框架。

  微服务框架HSF、Dubbo提供的主要功能是RPC框架，以及在RPC框架之上，提供相关的服务治理能力，包括注册发现、动态配置和限流熔断等。Spring Cloud提供REST框架，并在REST框架基础之上提供服务治理能力。因此实现微服务开发框架迁移主要是将RPC框架修改为REST框架，其操作主要包括两部分：

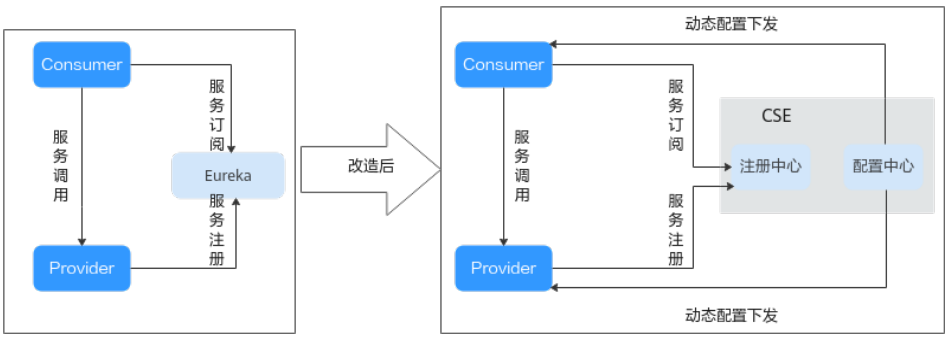
  - 将服务端的接口定义由RPC修改为REST。
  - 将客户端的调用方式由RPC修改为REST风格（包括RestTemplate，Feign等）。

  服务端的接口定义相对比较集中，客户端的使用则比较难于排查。为了尽可能减少客户端代码的排查和修改，采用Feign来实现客户端代码的替换。

  b. 将Spring Cloud+Nacos、Spring Cloud+Eureka接入到ServiceComb引擎。

  - 将Nacos、Eureka相关的依赖替换为ServiceComb引擎相关依赖。
  - 增加ServiceComb引擎相关配置。
  - Nacos、Eureka一些使用习惯的调整，比如如何规划服务配置和逻辑隔离等。

  下图以Eureka为例演示整个改造接入过程：



资源和成本规划

在操作前，需要的资源和成本规划见表1-1。

表 1-1 资源和成本规划

云资源	规格	数量	费用
微服务引擎	根据需要选择相应规格	1	根据选择的计费模式及规格确定相应费用
虚拟私用云	默认配置	1	00.00
子网	默认配置	1	00.00
安全组	根据需要开通入方向30100-30130等端口	1	00.00

实施步骤

1. 先将HSF或Dubbo框架改造为Spring Cloud框架。其基本操作步骤为：
   a. 修改POM和项目结构。
   b. 服务端RPC接口修改为REST接口。
   c. 客户端定义Feign引用。
   d. 删除HSF或Dubbo配置并增加Spring Cloud配置。
   e. 修改启动类。

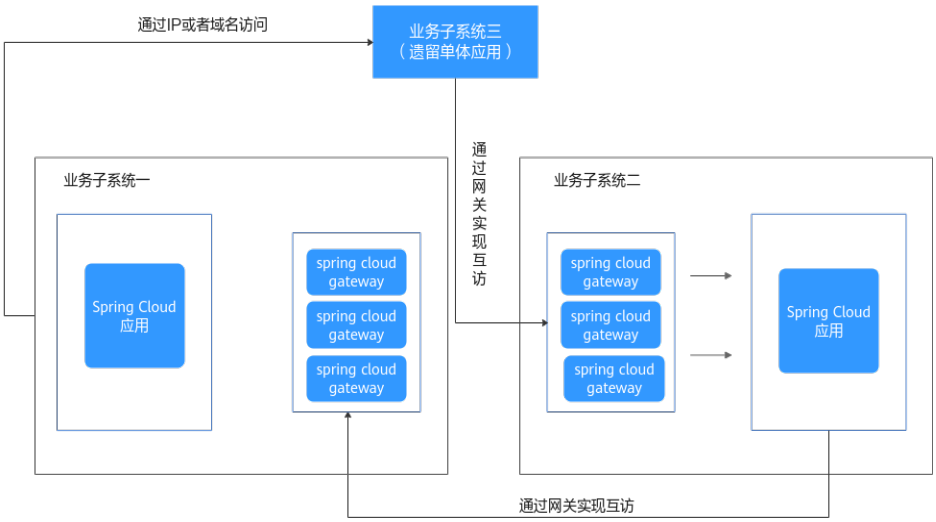
   您可以使用migrator工具一键将HSF、Dubbo等框架改造为Spring Cloud。
   - 将HSF框架改造为Spring Cloud框架详细操作指导请参考**HSF迁移Spring Cloud**。
   - 将Dubbo框架改造为Spring Cloud框架详细操作指导请参考**Dubbo迁移Spring Cloud**。
2. 将Spring Cloud+Nacos、Spring Cloud+Eureka接入到ServiceComb引擎。
   a. 修改pom文件将Nacos相关的依赖替换为ServiceComb引擎相关依赖。
   b. 修改bootstrap.yml文件增加ServiceComb引擎相关配置。
   c. 调整Nacos或Eureka的使用习惯。

您可以使用migrator工具一键将Spring Cloud+Nacos、Spring Cloud+Eureka接入到ServiceComb引擎。

- 将Spring Cloud+Eureka接入到ServiceComb引擎的详细操作指导请参考**Eureka+Spring Cloud迁移CSE**。
- 将Spring Cloud+Nacos接入到ServiceComb引擎的详细操作指导请参考**Nacos+Spring Cloud迁移CSE**。

系统集成场景

对于遗留系统、单体应用等，推荐使用网关实现互访，不采用mesher、sidecar等技术。一些未改造接入的服务，可通过如下结构图，同改造的服务协作以实现渐进式的改造。



常见问题

- 在进行微服务框架改造过程中的常见问题及解决思路请参考**HSF/Dubbo迁移Spring Cluod常见问题**。
- 将Spring Cloud、Nacos、Eureka迁移到ServiceComb引擎的常见问题及解决思路请参考**CSE注册发现和配置管理常见问题**。

2 ServiceComb 引擎托管应用

2.1 托管 Spring Cloud 应用

2.1.1 概述

适用场景

Spring Boot、Spring Cloud广泛应用于构建微服务应用。使用ServiceComb引擎托管Spring Cloud应用，主要目的是使用高可靠的商业中间件替换开源组件，对应用系统进行更好地管理和运维，改造过程应尽可能降低对业务逻辑的影响。适用于如下场景：

- 基于Spring Boot开发的应用系统，不具备微服务基本能力。应用系统通过集成Spring Cloud Huawei，具备服务注册发现、动态配置管理等能力。
- 基于Spring Cloud开源技术体系开发的应用系统，例如已经采用Eureka实现注册发现、采用Nacos实现动态配置，应用系统通过集成Spring Cloud Huawei，使用高可靠的商业中间件替换开源中间件，降低维护成本。
- 基于Spring Cloud其他开发体系，例如Spring Cloud Alibaba、Spring Cloud Azure等构建的云原生应用，使用Spring Cloud Huawei迁移到华为云运行。

应用建议

在开始使用ServiceComb引擎托管Spring Cloud应用前，可以参考如下建议评估改造的风险和工作量：

- 改造的基本原理是通过实现Spring Cloud提供的DiscoveryClient、PropertySource等接口，为Spring Cloud应用提供注册发现、动态配置等功能。这些实现独立于业务逻辑的开发，集成Spring Cloud Huawei不影响业务逻辑。Spring Cloud开源技术体系、Spring Cloud Alibaba、Spring Cloud Azure等，也都遵循这个设计模式。因此改造可以归纳为两种场景：集成和替换。不具备微服务能力的Spring Boot应用，只需要集成Spring Cloud Huawei；具备微服务能力的Spring Cloud应用，则需要使用Spring Cloud Huawei替换掉相关组件。
- 在替换场景，如果业务系统没有直接依赖实现组件的API，那么替换过程只需要移除原有依赖，添加Spring Cloud Huawei依赖，工作量非常小。如果业务系统大量

依赖实现组件的API，那么替换工作量会相应增加。根据实际经验，业务系统通常都不会直接依赖实现组件的API。

- 改造过程中最容易出现的问题是三方软件兼容性问题。处理兼容性问题的最佳策略是存在两个不同版本的三方软件时，优先使用新版本。对于Spring Boot、Spring Cloud版本，尽可能使用社区最新的版本，并紧跟社区的版本配套关系。例如使用Spring Cloud Hoxton.SR8版本，Spring Boot则使用2.3.5.RELEASE版本。虽然Spring Cloud Hoxton.SR8声称支持Spring Boot 2.2.x版本，但是多数组件都是集成2.3.5.RELEASE进行测试的，紧跟社区的版本配套关系，能够极大的减少兼容性问题的发生。**三方软件版本管理策略**会进一步说明三方软件兼容性问题的最佳实践。
- Spring Cloud最佳匹配ServiceComb引擎2.x版本，本最佳实践都是基于ServiceComb引擎2.x。ServiceComb引擎1.x和2.x具体改造过程的唯一差异是：配置中心类型ServiceComb引擎1.x使用的是config-center；ServiceComb引擎2.x使用的是kie。因此，ServiceComb引擎1.x的改造也可参考本最佳实践。

2.1.2 快速接入 ServiceComb 引擎

使用Spring Cloud Huawei接入ServiceComb引擎主要步骤可以归纳为如下两个步骤：

1. 增加/修改组件依赖。
2. 在配置文件“boostrap.yaml”中增加ServiceComb引擎配置信息。

具体操作，请参考**Spring Cloud接入ServiceComb引擎**。本章节补充在实际改造过程中需要注意的一些事项，特别是组件依赖有关的注意事项。

假设原来的业务系统都是基于Maven的项目。

第一步：熟悉原业务系统 pom 结构

Spring Cloud应用系统的pom结构一般分三种：

- 第一种使用Spring Boot或者Spring Cloud提供的公共pom作为parent，例如：

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

  或者：

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

- 第二种使用项目本身的parent，不使用Spring Boot或者Spring Cloud提供的公共pom作为parent。但是在项目中，通过dependency management引入了依赖管理，例如：

```
<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>
```

- 还有些应用系统会混合使用第一种和第二种，既使用了Spring Boot或者Spring Cloud提供的公共pom作为parent，又通过dependency management引入了依赖管理。

第二步：修改 parent 和 dependency management 避免三方软件冲突

parent和dependency management的修改是避免三方软件冲突的关键步骤。

1. 首先确定选用的Spring Cloud Huawei的版本，然后查询Spring Cloud Huawei版本对应的Spring Boot版本和Spring Cloud版本。Spring Cloud Huawei一般建议使用当前最新版本，配套的Spring Boot版本和Spring Cloud版本可以在**Spring Cloud Huawei官网**查询。
2. 比对当前项目parent的版本与Spring Cloud Huawei配套的Spring Boot版本和Spring Cloud版本，如果当前项目的parent版本比较低，修改为Spring Cloud Huawei的版本；如果当前版本的parent版本比较高，则无需修改。
3. 在当前项目的dependency management中独立引入Spring Boot、Spring Cloud、Spring Cloud Huawei的依赖管理。如果原来项目的Spring Boot或者Spring Cloud版本比较高，使用原来项目的版本；如果原来项目的版本比较低，则使用Spring Cloud Huawei的版本。需要注意依赖管理的顺序，排在前面的依赖管理会优先使用。Spring Boot、Spring Cloud的版本是基础，被这两个依赖关系管理的软件，建议都不提供额外的依赖管理，跟随社区的版本，可以有效减少冲突。这里之所以把3个依赖关系独立出来引入，是为了给以后独立升级其中一个组件提供便利。

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

如果业务系统集成了Spring Cloud Alibaba等依赖关系，在dependency management删除；一些已经被Spring Boot、Spring Cloud管理的不必要的依赖，也建议删除。常见的需要删除的依赖管理有：

```
<dependencyManagement>
 <dependencies>
```



```
    <!-- 第三方扩展的依赖 -->
    <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>
      <!-- 已经被Spring Cloud管理的依赖 -->
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
        <version>1.4.7.RELEASE</version>
      </dependency>
    </dependencies>
</dependencyManagement>
```

第三步：添加/删除依赖

添加的是服务注册发现、集中配置管理、服务治理相关的组件，删除的也是这些组件的第三方实现。这些功能以外的其他组件不需要变化，但需要关注Spring Boot、Spring Cloud版本升级后，这些软件可能需要配套升级。兼容性的问题通常会在编译阶段或者服务启动阶段发现。

添加依赖一般不指定版本号，把版本号交给parent和dependency management管理。

在微服务应用中引入：

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

在Spring Cloud Gateway应用中引入：

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

如果存在下面这些依赖，需要删除：

```
<!-- nacos场景 -->
<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>
```

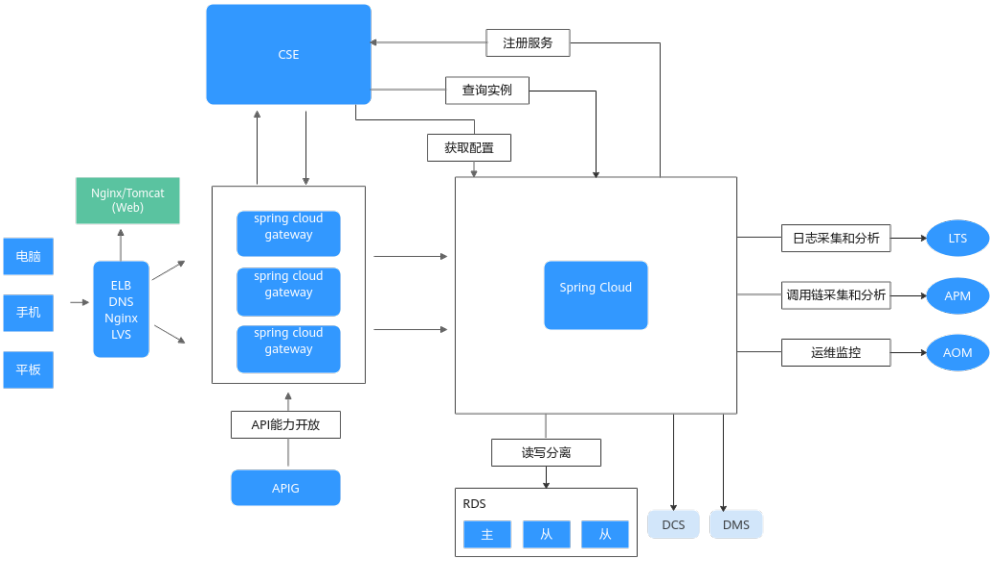
```
</dependency>

<!-- eureka场景 -->
<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>
```

通常下面的一些依赖，不需要删除：

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

2.1.3 合理的规划系统架构

Spring Cloud提供了丰富的组件，帮助搭建具备足够韧性的云原生系统。spring cloud gateway具备通用网关的大部分能力，并且集成了Spring Cloud的服务治理能力，可以实现Spring Cloud多协议转发。一个典型的Spring Cloud云原生架构如下：



该架构采用静态页面和服务分离，这样静态页面可以灵活的使用CDN、Nginx等形态部署。spring cloud gateway屏蔽了内部微服务的结构，一般会搭配流量控制、安全认证等服务治理策略，使得内部服务能够灵活的进行拆分合并，降低内部服务直接面对流量攻击的风险。

2.1.4 三方软件版本管理策略

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

本章节分享三方软件管理的最佳实践，帮助您打造一个持续演进的应用系统。

开源软件选型

主要的开源社区，例如**Spring Boot**、**Spring Cloud**等都会维护多个版本分支。以Spring Cloud为例，存在Hoxton、Greenwich、2020.0.x等分支，其中大部分分支都已经停止维护。开源软件多数维护的分支存在两个：一个为最新版本的开发分支；一个为最近的稳定维护分支。

在开源软件选型上，应该紧跟社区的版本节奏，使用继续提供维护的分支的最新版本。在选择开发分支和维护分支上，没有严格的定论，需要视具体的产品功能确定。例如产品竞争力严重依赖某个三方软件的特性，那么更倾向于选择开发分支的最新版本；一个产品依赖某个三方软件的特性稳定，不使用新功能，那么更倾向于选择维护分支。

不建议选择社区已经停止维护的分支、以及虽然还在维护但发布时间超过半年以上的分支版本。虽然使用这些版本暂时没有发现功能问题，但是会给产品的持续演进带来严重影响：

1. 软件的安全漏洞得不到及时处理。安全漏洞的发现和利用都有一定的时间周期，长期使用老版本，安全漏洞被利用的可能性变大，使得系统更加容易被攻击。
2. 系统出现故障，更难寻求社区支持。停止维护的版本，或者已经发布超过半年以上的版本，很难得到社区的支持。
3. 系统演进变得更加困难。当系统需要增加新特性，引入新开发工具时，老版本更难与新开发工具兼容。
4. 还有很多的故障，可能在新版本已经修复，新版本在代码可维护性、性能等方面也都优于老版本。

因此，在开源软件选型问题上 ，最佳方案就是选择社区提供维护升级的开发分支或者维护分支，根据问题驱动升级到分支的最新版本，每季度周期性升级到分支的最新版本。

三方软件版本管理

首先通过一个简单的例子，介绍三方件冲突的原理。假设开发一个X项目，该项目需要同时引用项目A提供的组件，也需要引用项目B的组件，并且项目A和项目B同时依赖了项目C，但是版本号不同。

- 项目X的pom：

```
<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>
```

- 项目A的pom

```
<dependency>
  <groupId>groupC</groupId>
  <artifactId>artifactC</artifactId>
  <version>0.1.0</version>
</dependency>
```

- 项目B的pom：

```
<dependency>
  <groupId>groupC</groupId>
```

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

项目X在最终发布的时候，会出现如下几种情况：

- 使用项目C的0.2.0版本。由于项目A是使用0.1.0版本编译和测试的，那么组件A可能无法正常工作。例如0.2.0版本和0.1.0不兼容，并且项目A恰好使用了这些不兼容的接口。
- 使用项目C的0.1.0版本。由于项目B是使用0.2.0版本编译和测试的，那么组件B可能无法正常工作。例如项目B使用了0.2.0提供的新接口。

可以看出，如果项目A和项目B使用的C组件存在接口不兼容的情况，无论怎么调整，项目X都无法正常工作。必须修改项目A的代码，使用和B同样的或者兼容的版本进行测试，发布新的版本给X项目使用。

因此，进行依赖管理的最佳策略是保证公共组件的依赖，都使用较高的版本。尽管如此，通常还是会碰到一系列问题，特别是项目依赖关系非常复杂的情况。

目前主流的复杂项目，都采用dependency management机制管理依赖。使用dependency management已经被验证是比较有效的管理依赖的手段，因此被开源社区广泛使用。例如，Spring Boot、Spring Cloud和Spring Cloud Huawei，开发者可以通过查看Spring Cloud Huawei的源代码目录结构了解dependency management的具体使用。

对于复杂项目，以Spring Cloud Huawei项目为例，和依赖关系管理有关的pom文件包含：

```
/pom.xml # 项目的根目录
/spring-cloud-huawei-dependencies/pom.xml # 项目的主要依赖管理。和项目dependency management有关的声明都在这个文件。
/spring-cloud-huawei-parents/pom.xml # parents既给本项目的子module使用，也给项目开发者使用，类似Spring Boot提供的parent
/spring-cloud-huawei-bom/pom.xml # bom主要用于项目开发者期望将Spring Cloud Huawei自己提供的组件引入项目的依赖管理，而不期望引入Spring Cloud Huawei依赖的第三方软件的版本。
```

Spring Cloud Huawei的这几个pom是相对完整的pom结构，从不同的使用视角给开发者提供了可以引入的pom，比较适用于公共开发组件。

一般的微服务开发项目可能只包含“/pom.xml”，项目的parent和dependency management都在“/pom.xml”声明。引入dependency management以后，项目中的所有dependency声明，都不指定版本号。这样的好处是当需要升级三方软件版本的时候，只需要修改“/pom.xml”里面的dependency management，其他地方都不需要排查修改。

您可以通过**Spring Cloud Huawei的示例项目**，直观体会dependency management的原理和作用：

1. 执行命令**mvn dependency:tree**查看项目的依赖关系。
2. 修改“/pom.xml”中“spring-boot.version”，再执行命令**mvn dependency:tree**查看项目的依赖关系的变化。
3. 调整“/pom.xml”中“spring-boot-dependencies”和“spring-cloud-dependencies”的位置，再执行命令**mvn dependency:tree**查看项目的依赖关系的变化。

当项目依赖的三方软件增多的时候，识别软件之间的配套关系是非常困难的。一个比较好的策略是跟随Spring Boot、Spring Cloud的版本配套关系。将spring-boot-dependencies和spring-cloud-dependencies作为依赖关系管理的基础是非常好的选

择。因为Spring Boot、Spring Cloud使用比较广泛，社区能够及时修复兼容性问题，开发者只需要升级Spring Boot、Spring Cloud版本即可，而无需关注被其依赖的其他三方软件的版本号。

升级三方软件，能够推动软件的持续改善，但是也需要进行一些工程能力的提升，例如自动化测试能力的建设。

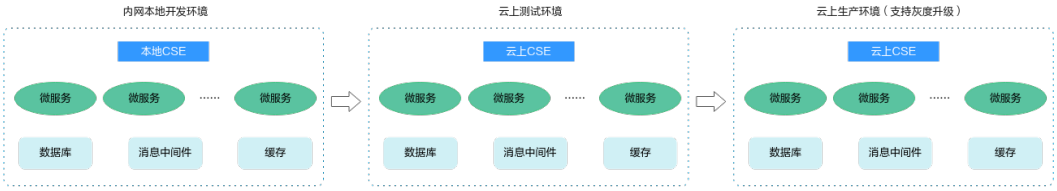
关于三方软件管理的更多内容，请参考**Java Chassis兼容问题和兼容性策略**。

2.1.5 开发环境规划管理

规划开发环境的目的是要保证开发人员更好的并行工作，减少依赖，减少搭建环境的工作量，降低生产环境上线的风险。

管理开发环境的目的是为了更好的进行开发测试，部署上线。

图 2-1 开发环境



结合项目经验，一般会按照**图2-1**规划开发环境：

- 搭建内网本地开发环境。本地开发环境的好处是各个业务/开发者可以搭建符合自己需要的最小功能集合环境，方便查看日志、调试代码等。本地开发环境能够极大的提升代码开发效率，减少部署和调试的时间。本地开发环境的不足之处是集成度不高，需要集成联调的时候，很难确保环境稳定。
- 云上测试环境是相对比较稳定的集成测试环境。本地开发测试完成后，各个业务将本领域的服务部署到云上测试环境，并且可以调用其他领域的服务进行集成测试。根据业务规模的复杂程度，可以将云上测试环境进一步分为α测试环境、β测试环境、γ测试环境等，这些测试环境集成程度由低到高。一般γ测试环境要求和生产环境一样的管理，确保环境稳定。
- 生产环境是正式业务环境，生产环境需要支持灰度升级功能，支持在线联调和引流，保证升级故障对服务造成的影响最小化。
- 云上测试环境可以通过开放CSE、中间件的公网IP，或者实现网络互通，这样可以使用云上的中间件替换本地环境，减少各个开发者自行安装环境的时间。这种情况也属于内网本地开发环境，微服务在本地开发环境的机器上运行。云上采用容器部署的微服务和本地开发环境机器上部署的微服务无法相互访问。为了避免冲突，云上测试环境只作为本地开发环境使用。

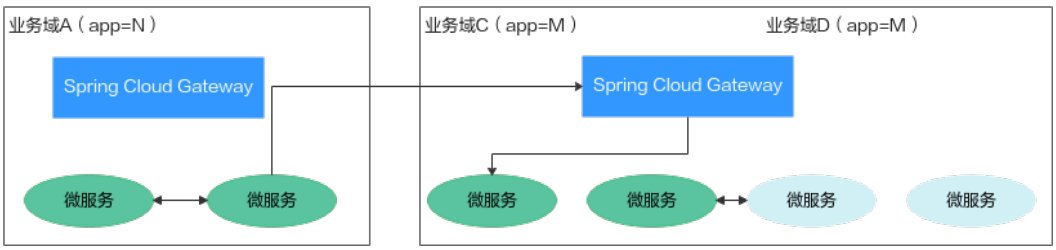
2.1.6 应用逻辑隔离管理

应用逻辑隔离主要用于不同的开发环境共享公共CSE资源的场景，降低成本。逻辑隔离还用于微服务之间的关系管理，通过配合合理的隔离策略，可以更好地控制微服务之间的可访问性、权限等。

服务发现

按照app隔离不同的业务域的微服务。

不同的业务域使用不一样的app名称。同一个业务域内的服务，能够相互发现和点到点访问。不同业务域的服务，不能相互发现，需要通过待访问微服务所在的业务域内的Spring Cloud Gateway去访问。



动态配置

动态配置按照公共、应用、服务三个层次进行管理。

简单的场景，可以使用应用级配置和服务级配置。应用级配置被该应用下的所有微服务共享，是公共配置；服务级配置只对具体微服务生效，是独享配置。复杂的场景，可以通过使用custom_tag和custom_value来定义配置。 例如某些配置，是对所有应用共享的，那么就可以使用这个机制。在配置文件增加如下配置：

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

只要配置项带有public标签，并且标签值为default，这些配置项就会对该微服务生效。可以这样理解配置中心：

1. 把配置中心当成数据库的一个表tbl_configurations， key是主键， 每个label都是属性。
2. 客户端会根据如下3个条件查询配置：
   - 自定义配置

     select * from tbl_configurations where custome_label=custome_label_value & withStrict=false
   - 应用级配置

     select * from tbl_configurations where app=demo_app & environment=demo_environment & withStrict=true
   - 服务级配置

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

   其中，withStrict为true的时候，表示有且只有条件里面指定的属性；withStrict为false的时候，表示除了条件里面的属性，允许有其他的属性。

   还可以给标签app指定多个应用，或者给标签service指定多个服务，这样配置项就可以对多个服务和应用生效，非常灵活。

2.1.7 配置文件加密方案

配置文件中经常会涉及一些敏感信息，例如帐号密码等参数。这时需对这些敏感信息进行加密，提供信息安全性。

本章节介绍使用jasypt-spring-boot-starter组件进行加密的实践，以RBAC认证中涉及的帐号名和密码作为示例。

1. 在pom文件中引入加密组件对应的依赖。

```
<dependency>
  <groupId>com.github.ulisesbocchio</groupId>
  <artifactId>jasypt-spring-boot-starter</artifactId>
  <version>2.1.2</version>
</dependency>
```

2. 配置密码。
   - 可将密码直接配置在配置文件（例如：application.properties）中。这种方式不安全，不建议使用。

     ```
     jasypt.encryptor.password=******
     ```

     ******请填写为实际加密使用的密码。
   - 在JVM启动参数中设置密码。

     ```
     -D jasypt.encryptor.password=******
     ```

     ******请填写为实际加密使用的密码。

3. 实现加密方法。

```
//此处设置为配置项jasypt.encryptor.password的密码
public static String salt = "GXXX6"(用户自定义);

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

//测试解密是否正常
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);
}
```

   此处使用的加密方法是jasypt默认的加密方法，用户也可以自定义扩展加解密方法，详情请参考**jasypt官方文档**描述。

4. 使用加密后的配置项。

   可以采用如下两种方式：
   - 写入配置文件方式

     ```
     spring:
       cloud:
         servicecomb:
           credentials:
             account:
               name: ENC(帐号名密文)
               password: ENC(密码密文)
     ```

     其中，帐号名密文、密码密文为**3**得到的结果。

说明

这种加密方式需要使用ENC()标记，用来识别是否启用了加密。ENC()为该加密方式的特殊标记，如果没有该标记，代表使用明文。

- 环境变量注入方式

```
spring_cloud_servicecomb_credentials_account_name = ENC(帐号名密文)
spring_cloud_servicecomb_credentials_account_password = ENC(密码密文)
```

其中，帐号名密文、密码密文为3得到的结果。

2.1.8 合理规划服务治理

2.1.8.1 滚动升级

推荐使用ServiceStage部署Spring Cloud应用，使用ServiceStage能够方便的实现滚动升级。

当使用ServiceStage部署应用的时候，可参考**设置应用健康检查**分别配置组件存活探针、组件业务探针，用以检测微服务的“存活”状态和“就绪”状态。

spring Boot提供了开箱即用的容器探针，LivenessStateHealthIndicator 、ReadinessStateHealthIndicator。

配置探针，需要启用spring-cloud-starter-huawei-actuator功能。

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

默认情况下LivenessStateHealthIndicator 、ReadinessStateHealthIndicator不包含任何的其他健康检查。Spring Cloud Huawei提供了一个健康检查功能，当服务注册成功，返回true。可以在ReadinessStateHealthIndicator包含这个检查：

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

然后配置如下**表2-1**设置组件业务探针。完成设置以后，服务注册成功的情况下，ServiceStage才会展示服务就绪状态，滚动升级的时候，也会在实例注册成功后，才停止老实例。

表 2-1 组件业务探针配置

参数	是否必填	参数含义
路径	是	请求的URL路径，例如/actuator/health/readiness。
端口	是	微服务的端口。
延迟时间/秒	否	开始检测的时间，对于启动时间较长的微服务，可以适当延长。
超时时间/秒	否	开始检测后，如果超过该时间未检测到探针状态则检测失败。

说明

Spring Cloud Huawei 1.9.0-Hoxton和1.9.0-2020.0.x及以上版本才提供这个模块。

除了设置探针，还需要设置滚动升级策略。核心的参数为“最大无效实例数”。“最大无效实例数”的默认值是0，当只有1个实例的情况下，滚动升级会存在中断。建议设置0<=最大无效实例数<实例数。

2.1.9 常见问题

2.1.9.1 Spring boot 从 2.0.x.RELEASE 升级到 2.3.x.RELEASE 兼容性问题

FeignClient 名问题

- 问题描述

  老版本的Spring Boot允许bean重名覆盖，新版本默认不允许，需要通过配置项启用。

- 解决方案

  增加配置：

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

Spring Data 接口变更

- 问题描述

  Spring Data接口经常发生变更。

- 解决方案

  使用新的接口修改代码，一般都有替换方案。例如new PageImpl修改为PageRequest.of，new Sort修改为Sort.of。

JPA 变更：多个 Entity 对应一个表

- 问题描述

  新版本要求一个Entity只能够对应一个表。

- 解决方案

  目前还没有简单方案，只能够一个表对应一个Entity，根据新版本的约束调整代码结构。

Mongo client 升级变更

- 问题描述

  MongoDbFactory的接口存在变更，需要调整为新版本的用法。

- 解决方案

```
@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();
}
```

2.1.9.2 动态配置常见问题

动态配置的类型选择

微服务引擎2.0的配置中心支持text、yaml等多种格式。

- 简单的key-value配置项
  可以使用text类型，配置中心的key对应于代码中的key。
- 大量的配置
  使用yaml格式，配置中心的key会被忽略，全量的key-value在yaml文件中定义。
  ServiceComb引擎1.x不支持yaml格式，可以通过Spring Cloud Huawei适配，来使用yaml，需要在微服务bootstrap中增加如下配置：

```
spring:
  cloud:
    servicecomb:
      config:
        fileSource: consumer.yaml # 需要按照yaml解析的配置项列表，以逗号分隔
```

- 初次使用ServiceComb引擎2.x
  建议选择Spring Cloud Huawei的最新版本，最新版本包含更多的特性并针对历史问题进行了较好的优化。

List 对象配置绑定

有些业务使用了List对象配置绑定，例如：

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

对于List对象，Spring Cloud默认都只会从一个PropertySource查询相关的配置项，如果其中一个PropertySource存在配置项的部分值，那么不会再查询其他值。 因此，在使用List对象绑定的时候，和这些List属性相关的配置，都必须全部放到配置中心，不支持部分元素在配置文件，部分元素在配置中心的场景。

可以将这个约束理解为“List配置的原子性”，即一个配置项（代码例子中的stringList或者modelList）不能被分割在不同的配置文件，保证配置项的原子性。

2.1.9.3 Spring Cloud 常见启动错误

2.1.9.3.1 注册中心地址错误

问题描述

当使用Spring Cloud Huawei时，启动微服务时，当报错示例如下：

```
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(DefaultHttpClientConnectionOperator.java:156) ~[httpclient-4.5.13.jar:4.5.13]
at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:376) ~[httpclient-4.5.13.jar:4.5.13]
```

原因分析

上述报错，是微服务注册中心地址不可用导致的。

解决方法

- 启动服务在本地部署

  在本地机器上使用**curl https://*注册中心IP地址*:30100/health**命令检查注册中心工作状态，查看是否返回类似如下信息：

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

  如果是，请检查是否因注册中心IP地址错误、注册中心端口号错误或者网络被隔离，导致网络不通。

- 启动服务在云上微服务引擎部署

  微服务通过ServiceStage部署在微服务引擎，注册中心地址可以通过环境变量自动注入。请检查注入的注册中心地址是否正确。如果注册中心地址错误，请修改为正确地址并重新部署服务。

2.1.9.3.2 同一应用和环境下的不同服务无法互相调用

问题描述

同一个应用下的服务，其部署环境加载了开启安全认证的微服务引擎专享版。由于不同服务使用的帐号不同，导致服务之间无法互相发现，从而导致无法互相调用。

解决方法

对调用服务使用的帐号绑定该服务的全部权限，同时绑定其他服务的只读权限。

具体操作请参考**系统管理**。

2.2 托管 Java Chassis 应用

2.2.1 概述

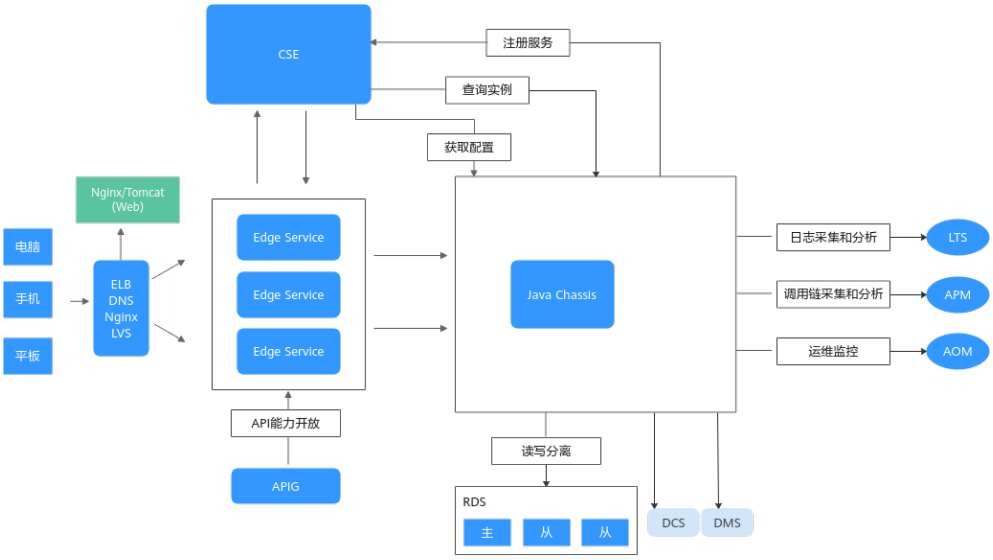
Java Chassis是Apache基金会管理的开源微服务开发框架，最早由CSE捐献，目前有上百个开发者为项目做出贡献。相对于Spring Cloud，Java Chassis，它提供了如下独特的功能：

- 灵活高性能的RPC实现。Java Chassis基于Open API，给出了不同RPC开发方式的统一描述，让微服务接口管理更加规范，同时保留了灵活的开发者使用习惯。Java Chassis基于Reactive，实现了高效的REST、Highway等通信协议，同时保留了传统Servlet等通信协议的兼容。
- 丰富的服务治理能力和统一的治理职责链。负载均衡、流量控制、故障隔离等常见的微服务治理能力都可以开箱即用，同时提供了统一的治理职责链，让新的治理功能的开发变得简单。

和Spring Cloud一样，Java Chassis也可以使用Spring、Spring Boot作为应用功能开发的基础组件，但是由于Java Chassis提供了独立的RPC实现，因此使用依赖于Spring MVC的功能组件会受到限制，比如使用Spring Securtiy，需要基于Java Chassis做一些适配。

2.2.2 合理的规划系统架构

Java Chassis提供了丰富的组件，帮助搭建具备足够韧性的云原生系统。Edge Service具备通用网关的大部分能力，并且集成了Java Chassis的服务治理能力，可以实现Java Chassis多协议转发。一个典型的Java Chassis云原生架构如下：



该架构采用静态页面和服务分离，这样静态页面可以灵活的使用CDN、Nginx等形态部署。Edge Service屏蔽了内部微服务的结构，一般会搭配流量控制、安全认证等服务治理策略，使得内部服务能够灵活的进行拆分合并，降低内部服务直接面对流量攻击的风险。

2.2.3 合理配置线程池参数

线程池是微服务的主要业务处理单元，合理的规划线程池不仅可以最大限度提升系统性能，还能防止异常情况导致系统无法给正常用户提供服务。线程池优化和业务自身的性能有很大关系，不同的场景参数设置不同，需要具体分析。下面分两种场景介绍。开始之前需要对业务的性能做一些基本的摸底，对常见的接口进行测试，查看时延。

- 业务性能很好的情况。

  即非并发场景，接口的平均时延小于10ms。

  业务性能很好的时候，为了让业务系统具备更好的可预测性，防止JVM垃圾回收、网络波动、突发流量等对系统的稳定性造成冲击，需要能够快速丢弃请求，

并配合重试等措施，以保障波动情况下系统性能可预测，同时不会出现偶然的业务失败，影响体验。

- **连接数和超时设置**

```
# 服务端verticle实例数，保持默认值即可。建议配置为8~10。
servicecomb.rest.server.verticle-count: 10
# 最大连接数限制。默认值为 Integer.MAX_VALUE。可以结合实际情况估算最大值，使系统具备更好的韧性。
servicecomb.rest.server.connection-limit: 20000
# 连接闲置时间。默认值60秒，一般不需要修改
servicecomb.rest.server.connection.idleTimeoutInSeconds: 60
# 客户端verticle实例数，保持默认值即可。建议配置为8~10。
servicecomb.rest.client.verticle-count: 0
# 一个客户端与服务器建立的最大连接数为 verticle-count * maxPoolSize，不要超过线程数。
#  这里是 10*50=500. 实例非常多的场景，要减小单个实例的连接数。
servicecomb.rest.client.connection.maxPoolSize: 50
# 连接闲置时间。默认值30 秒，一般不需要修改，要小于服务端的连接闲置时间。
servicecomb.rest.client.connection.idleTimeoutInSeconds
```

- **业务线程池配置**

```
# 线程池组数，建议 2~4
servicecomb.executor.default.group: 2
# 建议 50~200
servicecomb.executor.default.thread-per-group: 100
# 线程池排队队列大小，默认值为Integer.MAX_VALUE。高性能场景不要使用默认值，以快速丢弃请求
servicecomb.executor.default.maxQueueSize-per-group: 10000
# 队列最大等待时间，如果超过，理解丢弃请求的处理并返回。默认值为0。
# 高性能场景配置小的排队超时时间，快速丢弃请求
servicecomb.rest.server.requestWaitInPoolTimeout: 100
# 设置比较短的超时时间，快速丢弃请求， 但是不建议这个值小于1秒，可能导致很多问题。
servicecomb.request.timeout=5000
```

- 业务性能不那么好的情况。

  即非并发场景，接口的平均时延大于100ms。时延高通常是由于业务代码存在IO、资源等等待，CPU利用率上不去导致的。如果是由于计算复杂导致的，调优会变得复杂。

  当业务性能不太好的时候，下面几个参数值需要调大，否则业务会大量阻塞。业务性能不好，通过调大参数能够保证系统的吞吐量，应对突发流量来临时带来的业务失败。不过这个是以牺牲用户体验为代价的。

```
# 服务端连接闲置时间。
servicecomb.rest.server.connection.idleTimeoutInSeconds: 120000
# 客户端连接闲置时间。
servicecomb.rest.client.connection.idleTimeoutInSeconds: 90000
# 线程池组数。
servicecomb.executor.default.group: 4
# 线程池大小。
servicecomb.executor.default.thread-per-group: 200
# 线程池排队队列大小，性能不好的情况下需要排队
servicecomb.executor.default.maxQueueSize-per-group: 100000
# 配置较大的超时时间
servicecomb.rest.server.requestWaitInPoolTimeout: 10000
servicecomb.request.timeout=30000
```

2.2.4 合理配置日志文件

查看错误日志是定位问题的重要手段，需要合理规划日志输出，并且尽可能降低对系统性能的影响。规划日志文件有如下建议：

1. 使用**log4j2**或者**logback**输出日志。 将日志输出到文件，不要依赖于容器的stdout。
2. 打开metrics日志，将metrics日志输出到独立的文件，比如“metrics.log”， 而将业务日志输出到另外的文件，比如“servicecomb.log”。metrics参数配置如下：


```
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. 打开access log， 将access log输出到独立的日志文件。
4. 格式化打印业务日志，日志里面包含trace id，可以独立开发一个Handler，配置在Provider Handler的最前面，Handler在接收到请求后打印一条日志，处理完成了打印一条日志，对于问题界定，使用AOM快速检索相关日志等非常有帮助。

2.2.5 合理规划服务治理

2.2.5.1 滚动升级

推荐使用ServiceStage部署Java Chassis应用，使用ServiceStage能够方便的实现滚动升级。当使用ServiceStage部署应用的时候，可以配置组件业务探针，使得ServiceStage能够正确的检测微服务的状态。配置组件业务探针，需要启用metrics功能，然后将组件业务探针路径设置为“/health”。

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

除了设置探针，还需要设置滚动升级策略。核心的参数为“最大无效实例数”。“最大无效实例数”的默认值是0，当只有1个实例的情况下，滚动升级会存在中断。建议设置实例数>=2，0<=最大无效实例数<实例数 - 1即保证最小有2个可用的实例。

2.2.5.2 升级零中断

要实现升级零中断，通常需要解决如下问题：

1. 停止服务的时候，可能引起业务中断。在停止服务的过程中，可能服务正在处理请求，新的请求可能持续的发送到该服务。
2. 在微服务架构下，一般都会通过注册中心进行服务发现，客户端会缓存实例地址。停止服务的时候，使用者可能无法及时感知实例下线，并继续使用错误的实例进行访问，导致失败。
3. 实现升级零中断，需要进行滚动升级，在新版本功能就绪后，才能够停止老版本。

实现升级零中断需要很多的措施进行配合，比如**滚动升级**，实现零中断，建议保证最小有2个可用的实例。在本章节里面，主要描述从微服务的角度进行设置，更好的配合升级零中断。Java Chassis实现零中断的核心机制包括如下几个：

1. 优雅停机。服务停止的时候，需要等待请求完成，并拒绝新请求。

   Java Chassis优雅停机默认提供，在进程退出前，会进行一定的清理动作，包括等待正在处理的请求完成、拒绝未进入处理队列的新请求、调用注册中心接口进行

注销等动作。Java Chassis进程退出前，先将实例状态修改为DOWN，然后等待一段时间再进行后续的退出过程：

```
servicecomb:
  boot:
    turnDown:
      # 实例状态修改为DOWN以后等待时间，默认值为0，即不等待。
      waitInSeconds: 30
```

2. 重试：客户端对于网络连接错误，以及被拒绝等请求，需要选择新服务器进行重试。

   开启重试策略：

```
servicecomb:
  loadbalance:
    retryEnabled: true # 是否开启重试策略
    retryOnNext: 1  # 重新寻找一个实例重试的次数（不同于失败实例，依赖于负载均衡策略）
    retryOnSame: 0  # 在失败的实例上重试的次数
```

3. 隔离：对于失败超过一定次数的服务实例，进行隔离。

   开启实例隔离策略：

```
servicecomb:
  loadbalance:
    isolation:
      enabled: true
      enableRequestThreshold: 5 # 统计周期内实例至少处理的请求数，包括成功和失败。
      singleTestTime: 60000 # 实例隔离后，经过这个时间，会尝试访问。如果访问成功，则取消隔离，否则继续隔离。
      continuousFailureThreshold: 2 # 实例隔离的条件，连续两次失败。
```

2.2.6 升级到 Java Chassis 的最新版本

持续升级版本，可以更好的使用CSE的新功能和新特性，及时修复已知的质量和安全问题，降低维护成本。持续升级版本也会带来一些兼容性问题。一个比较好的策略是将持续升级纳入版本计划，安排足够的时间进行，而不是以问题驱动。持续升级还需要构建自动化测试能力，以减少版本升级的验证时间和控制版本升级的风险，及早发现问题。持续的构建自动化能力和升级版本，是被证明有效的构建高质量软件的最佳实践。

2.3 托管 Kratos 应用

概述

Kratos是轻量级Go微服务框架，包含大量微服务相关功能及工具，更多Kratos框架相关信息详见**Kratos官方文档**。

前提条件

创建ServiceComb引擎，请参考**快速创建ServiceComb引擎**。

代码接入

- Provider端：

```
package main

import (
	"crypto/tls"
	"log"
	"net/url"
	"github.com/go-chassis/sc-client"
```

```
        "github.com/go-kratos/kratos/contrib/registry/servicecomb/v2"
        "github.com/go-kratos/kratos/v2"
)
func main() {
    c, err := sc.NewClient(sc.Options{
                // EndPoints填写servicecomb中service center的地址
        Endpoints: []string{"127.0.0.1:30100"},
                EnableSSL: true,
                TLSConfig: &tls.Config{
                    InsecureSkipVerify: true,
                },
    })
    if err != nil {
        log.Panic(err)
    }
    r := servicecomb.NewRegistry(c)
    app := kratos.New(
                //需要传入注册实例的endpoint
                kratos.Endpoint(&url.URL{Scheme: "{endpoint}"}),
                //服务名
        kratos.Name("helloServicecomb"),
                //接入servicecomb注册中心service center
        kratos.Registrar(r),
    )
    if err := app.Run(); err != nil {
        log.Fatal(err)
    }
}
```

- Consumer端：

```
package main

import (
        "context"
        "crypto/tls"
        "log"
        "github.com/go-chassis/sc-client"
        "github.com/go-kratos/kratos/contrib/registry/servicecomb/v2"
         "github.com/go-kratos/kratos/v2/transport/grpc"
)
func main() {
    c, err := sc.NewClient(sc.Options{
                EnableSSL: true,
                TLSConfig:  &tls.Config{
                    InsecureSkipVerify: true,
                },
                // EndPoints填写servicecomb中service center的地址
        Endpoints: []string{"127.0.0.1:30100"},
    })
    if err != nil {
        log.Panic(err)
    }
    r := servicecomb.NewRegistry(c)
    ctx := context.Background()
    conn, err := grpc.DialInsecure(
        ctx,
                //Endpoint格式:<schema>://[namespace]/<service-name>
        grpc.WithEndpoint("discovery:///helloServicecomb"),
        grpc.WithDiscovery(r),
    )
    if err != nil {
        return
    }
    defer conn.Close()
}
```

应用部署

参考**快速体验ServiceStage**选择合适的方式部署应用。

2.4 托管 Kitex 应用

本文介绍如何将Kitex应用接入ServiceComb引擎。

概述

Kitex，作为Golang微服务RPC框架，具有高性能、强可扩展的特点。更多Kitex框架的详细信息，请参见**Kitex官方文档**。

前提条件

创建ServiceComb引擎，请参考**快速创建ServiceComb引擎**。

代码接入

- Provider端：

```
import (
    "context"
    "log"
    "net"
    "github.com/cloudwego/kitex-examples/hello/kitex_gen/api"
    "github.com/cloudwego/kitex-examples/hello/kitex_gen/api/hello"
    "github.com/cloudwego/kitex/pkg/rpcinfo"
    "github.com/cloudwego/kitex/server"
    "github.com/kitex-contrib/registry-servicecomb/registry"
)
type HelloImpl struct{}
func (h *HelloImpl) Echo(_ context.Context, req *api.Request) (resp *api.Response, err error) {
    resp = &api.Response{
        Message: req.Message,
    }
    return
}
func main() {
    // **初始化ServiceComb注册中心，默认从环境变量读取配置**
    r, err := registry.NewDefaultSCRegistry()
    if err != nil {
        panic(err)
    }
    svr := hello.NewServer(
        new(HelloImpl),
        server.WithRegistry(r),
        server.WithServerBasicInfo(&rpcinfo.EndpointBasicInfo{ServiceName: "Hello"}),
        server.WithServiceAddr(&net.TCPAddr{IP: net.IPv4(0, 0, 0, 0), Port: 8080}),
    )
    if err := svr.Run(); err != nil {
        log.Println("server stopped with error:", err)
    } else {
        log.Println("server stopped")
    }
}
```

- Consumer端：

```
import (
    "context"
    "log"
    "time"
    "github.com/cloudwego/kitex-examples/hello/kitex_gen/api"
    "github.com/cloudwego/kitex-examples/hello/kitex_gen/api/hello"
    "github.com/cloudwego/kitex/client"
    "github.com/kitex-contrib/registry-servicecomb/resolver"
)
```

```
func main() {
    // **初始化ServiceComb注册中心，默认从环境变量读取配置**
    r, err := resolver.NewDefaultSCResolver()
    if err != nil {
        panic(err)
    }
    newClient := hello.MustNewClient(
        "Hello",
        client.WithResolver(r),
        client.WithRPCTimeout(time.Second*3),
    )
    for {
        resp, err := newClient.Echo(context.Background(), &api.Request{Message: "Hello"})
        if err != nil {
            log.Fatal(err)
        }
        log.Println(resp)
        time.Sleep(time.Second)
    }
}
```

应用部署

参考**快速体验ServiceStage**选择合适的部署方式。

验证

部署成功后，登录微服务引擎控制台，在左侧导航栏选择“ServiceComb引擎专享版”，单击**前提条件**创建的ServiceComb引擎，选择“微服务目录”，单击微服务名称，在“实例列表”页签查看服务实例是否已经成功注册。

您也可以验证Consumer调用Provider能够正常调用。

1. 设置环境变量serverAddr和serverPort为ServiceComb引擎服务注册发现地址的ip和port。

```
[root@ecs-test123 consumer]# export serverAddr=192.168.1.89
[root@ecs-test123 consumer]# export serverPort=30100
[root@ecs-test123 consumer]#
```

2. 运行consumer。说明成功从ServiceComb引擎的服务中心获取了provider的ip和port，并调用了provider。

```
[root@ecs-test123 consumer]# ./consumer
2022/08/30 20:03:33 INFO: Use Service center v4
2022/08/30 20:03:33 DEBUG: service center has new revision ed23ee99110031cca0905f51f01cea986aca7764
2022/08/30 20:03:33 Response({Message:Hello})
2022/08/30 20:03:34 Response({Message:Hello})
2022/08/30 20:03:35 Response({Message:Hello})
2022/08/30 20:03:36 Response({Message:Hello})
2022/08/30 20:03:37 Response({Message:Hello})
```

3 ServiceComb 引擎应用治理

3.1 SpringCloud 应用实现优雅上下线功能

3.1.1 功能介绍

在应用使用过程中，应用的发布、重启、扩缩容操作无法避免，为了保证应用正确上下线、流量不丢失，微服务引擎基于Sermant Agent提供了一套优雅上下线的方案，包括预热、延迟下线等，避免了请求超时、连接拒绝、流量丢失等问题的发生。

说明

- 此功能目前处于公测阶段，当前仅在华东-上海一支持。
- 当ServiceComb引擎为2.x版本且未开启安全认证时，支持此功能。
- 目前仅支持基于Spring Cloud框架开发java应用，支持的注册中心包含Eureka、Nacos、Consul以及Zookeeper。
- 优雅上下线基于Spring Cloud的负载均衡实现，若有实现自定义负载均衡，该能力将会失效。

优雅上线实现机制

预热是优雅上线的核心机制，Sermant Agent还提供了延迟注册机制，减少流量丢失，从而实现优雅上线。

- **延迟注册**

  在服务启动成功之后不立刻注册，而是延迟一段时间再去注册，目的是虽然服务启动成功了，但可能还有一些框架或者业务的代码没有初始化完成，可能会导致调用报错，可以通过设置延迟注册，让服务充分初始化后再注册到注册中心对外提供服务。

- **预热**

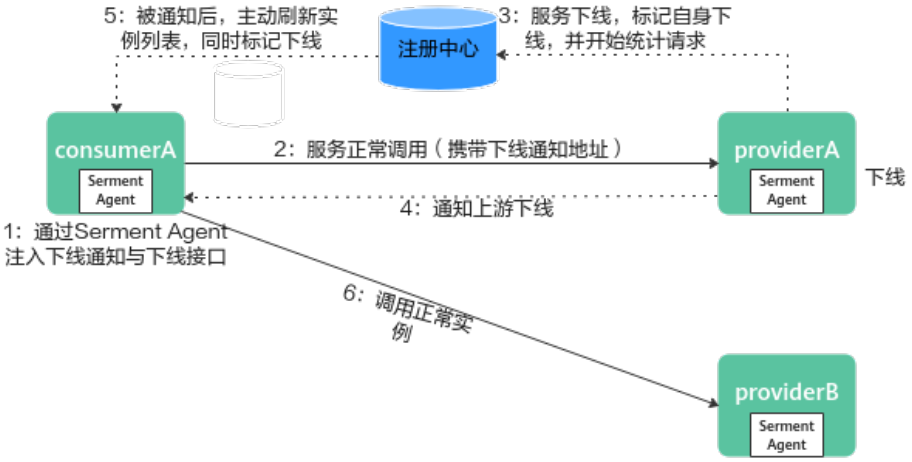
  该方式主要用于解决当流量突然增加时，可能瞬间把实例压垮的问题。通过预热，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，目的是采用少

流量对服务实例进行初始化，防止服务崩溃。预热是基于客户端实现的，当流量进入时，Sermant Agent会动态调整流量，根据服务的预热配置，对流量进行动态分配。对于开启服务预热的实例，在刚启动时，会给该实例分配较少的流量，随后流量将以曲线方式逐渐增加至与其他实例近乎持平。

优雅下线实现机制

延迟下线是优雅下线的核心机制，且Sermant Agent还提供了流量统计机制，即服务处理完所有统计的请求后再下线，减少流量丢失，从而实现了优雅下线。

图 3-1 优雅下线结构图



- **延迟下线**

  当服务提供者实例下线时，无法避免仍有业务请求还未处理完成，从而可能会出现请求丢失的现象。延迟下线即对下线的实例提供保护，优雅下线插件基于**下线实时通知+刷新缓存的机制**快速更新上游的实例缓存，服务消费者能尽早感知服务提供者实例下线的行为，同时基于**流量统计**的方式，确保即将下线的实例尽可能的将流量处理完成，最大程度避免流量丢失。

- **流量统计**

  当服务即将下线时，为确保当前请求已全部处理完成，Sermant Agent会尝试等待30s（可配置），定时统计和判断当前实例请求是否均处理完成，处理完成后最终下线。

版本支持

Spring Cloud Version	Spring Boot Version	Zookeeper Discovery Version	Nacos Discovery Version	Consul Discovery Version	Eureka Client Version
Edgware.SR2+	1.5.x	1.x.x、2.0.x	1.5.x	1.x.x、2.0.x、2.1.x	1.4.x、2.0.x、2.1.x
Finchley.x	2.0.x	2.x.x	1.5.x、2.0.x、2.1.x	1.3.x、2.0.x、2.1.x	1.4.x、2.0.x、2.1.x

Spring Cloud Version	Spring Boot Version	Zookeeper Discovery Version	Nacos Discovery Version	Consul Discovery Version	Eureka Client Version
Greenwich.x	2.1.x	2.x.x	1.5.x、2.0.x、2.1.x	1.3.x、2.0.x、2.1.x	1.4.x、2.0.x、2.1.x
Hoxton.x	2.2.x、2.3.x	2.x.x、3.0.0 - 3.1.0	2.x.x、2020.0.RC1、2021.1	1.3.x、2.0.x、2.1.x、2.2.x	1.4.4.RELEASE - 1.4.7.RELEASE、2.x.x、3.0.0 - 3.1.0
2020.0.x	2.4.x、2.5.x	3.0.0 - 3.1.0	2.x.x、2020.0.RC1、2021.1	3.0.0 - 3.1.0	2.1.x、2.2.x、3.0.0 - 3.1.0
2021.0.0	2.6.x	3.0.0 - 3.1.0	2.x.x、2020.0.RC1、2021.1	3.0.0 - 3.1.0	3.0.0 - 3.1.0

3.1.2 基于虚拟机场景

前提条件

- 已创建ECS实例， 创建ECS请参考**ECS快速入门**。
- 已安装JDK（版本为1.8及以上版本）并配置环境变量，详情请参考**Java Downloads**。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- ECS、ServiceComb引擎处于相同的VPC网络下。
- Sermant Agent开源版本要求1.0.3及以上。

操作步骤

步骤1 安装Sermant Agent，请参考**安装Sermant Agent**。

步骤2 启动应用并开启优雅上下线能力。

在应用的启动参数添加如下参数，添加启动参数后，待应用启动完成。

```
-javaagent:${HOME}/java-agent/java-agent.jar=appName=default
-Ddynamic_config_serverAddress={CSE_CONFIG_CENTER_ENDPOINTS}
-Dregister.service.address={CSE_REGISTRY_ENDPOINTS}
```

说明

相关配置介绍：

- appName为agent服务名称，该配置无需修改，使用default即可。
- ServiceComb引擎服务注册发现地址（CSE_REGISTRY_ENDPOINTS）与CSE配置中心地址（CSE_CONFIG_CENTER_ENDPOINTS）需替换为实际地址，可参考如下方式获取：
  - ServiceComb引擎服务注册发现地址：**获取ServiceComb引擎服务注册发现地址**。
  - CSE配置中心地址：**获取ServiceComb引擎配置中心地址**。

特别说明：

优雅下线是基于http协议进行通知，默认通知端口为16688，若您在虚机部署出现端口冲突问题（通常是单个ECS部署多个实例），请在启动时添加如下参数规避：

```
# 请更换下面的端口号
-Dgrace_rule_httpServerPort=16688
```

----结束

3.1.3 基于 CCE 场景

前提条件

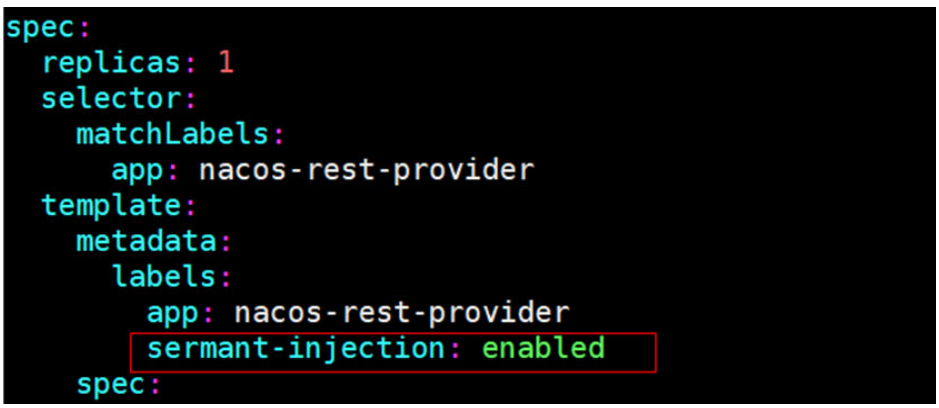
- 已创建CCE集群，详情请参考**创建CCE集群**。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- CCE与ServiceComb引擎处于相同VPC下。
- Sermant Injector版本要求1.0.11及以上，Sermant Agent镜像版本要求1.0.9及以上。

操作步骤

步骤1 为CCE集群安装sermant-injector，请参考**CCE部署场景接入指南**。

步骤2 为工作负载（deployment）打上标签并重启相关服务。

在下图所示位置为工作负载（deployment）打上标签sermant-injection: enabled。打上标签后，sermant-injector会在Pod重启时自动挂载Sermant Agent，并开启优雅上下线能力。



须知

优雅下线能力依赖k8s的preStop机制，若您的编排文件已配置preStop，sermant-injector将无法自动配置，需要您在**编排文件**位置“spec > containers > lifecycle > preStop > exec > command”添加如下命令：

curl -XPOST http://127.0.0.1:16688/\$\$sermant\$\$/shutdown 2>/tmp/null;sleep 30;exit 0

添加该命令会在POD停止前通知实例进行下线。其中16688为下线通知端口，默认为该值，可通过环境变量“grace.rule.httpServerPort”进行指定。

您的容器需要支持curl命令，否则会导致优雅下线失效。

----结束

3.1.4 配置及验证优雅上下线能力

本章节以Nacos demo应用为例，通过Sermant Agent接入ServiceComb引擎，并在CCE集群上验证优雅上下线功能。

说明

应用基于Sermant Agent会自动接入ServiceComb引擎，接入流程请参考**CCE部署场景接入指南**。

（可选）自定义优雅上下线配置

优雅上下线能力默认开启，若需自定义配置，请参考如下方式：

说明

如果方式一与方式二同时配置，将以方式二为准。

- 方式一：

  在启动SpringCloud应用时通过环境变量或者-D参数的形式进行配置，配置参数如下：

参数项	说明
grace_rule_startDelayTime	注册延迟时间，默认0秒，若大于0，则开启注册延迟。
grace_rule_enableWarmUp	开启优雅上线能力，默认开启。
grace_rule_warmUpTime	优雅上线时间，单位秒，该配置生效需开启优雅上线功能，默认120秒。
grace_rule_enableGraceShutdown	配置优雅下线能力开关， 默认开启。
grace_rule_shutdownWaitTime	下线前的最大等待时间，默认30S。
grace_rule_enableOfflineNotify	开启下线通知，默认开启。

- 方式二：

  通过**配置管理**进行配置，配置步骤如下：

a. 登录微服务引擎控制台。

b. 单击 ，选择区域。

c. 选择“ServiceComb引擎”。

d. 单击待操作的ServiceComb引擎。

e. 选择“配置管理”。

f. 单击右上角的“新建配置”进入新建配置页面。

g. 输入“配置项”，命名为sermant.agent.grace。

说明

此处名称固定为该值。

h. 选择“配置范围”，这里以微服务级配置为例，选择您要配置的服务，如下图所示：

i. “配置格式”选择“YAML”，并自定义优雅上下线配置， 如下：

```
rule:
  startDelayTime: 0          # 注册延迟时间，默认0秒，若大于0，则开启注册延迟。
  enableWarmUp: true         # 开启优雅上线能力，默认为true（开启），如需关闭，请设置为fasle。
  warmUpTime: 120            # 优雅上线时间，单位秒，该配置生效需开启优雅上线功能，默认120秒。
  enableGraceShutdown: true  # 配置优雅下线能力开关，默认为true（开启），如需关闭，请设置为fasle。
  shutdownWaitTime: 30       # 下线前的最大等待时间，默认30S。
  enableOfflineNotify: true  # 开启下线通知，默认开启。
```

j. 单击右下角的“立即创建”，然后重启对应服务实例即可。

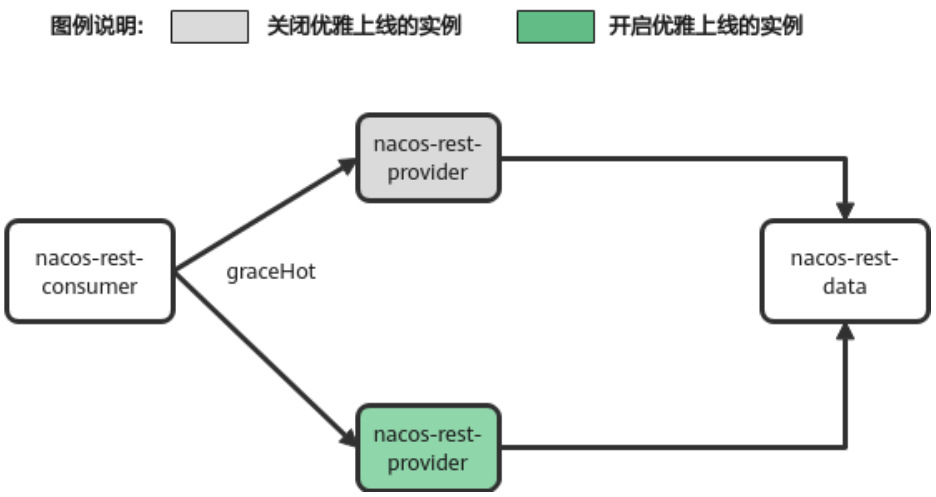
说明

下发配置时，请去掉注释，否则会导致下发配置失败。

验证优雅上下线能力

- 验证优雅上线能力。

图 3-2 优雅上线验证部署图



如上图所示，该套nacos应用有nacos-rest-consumer、nacos-rest-provider（两个实例）以及nacos-rest-data服务，其中灰色的实例已关闭开启优雅上线，而绿色实例开启优雅上线功能。

说明

若需**关闭预热**，请添加环境变量“grace_rule_enableWarmUp=false”进行指定。

而服务nacos-rest-consumer通过接口graceHot进行模拟调用。

a. 下载**demo应用**并打包。

b. 按照优雅上线验证部署图进行部署并将nacos-rest-provider的其中一个实例开启优雅上线能力。

c. 查看应用是否已全部接入ServiceComb引擎。

参考**查看微服务列表**查看您的应用是否已接入ServiceComb引擎。

d. 待应用接入ServiceComb引擎后，使用以下脚本**invoke-hot.sh**模拟调用过程。

```
#!/bin/bash
endpoint=127.0.0.1:31021
url=${endpoint}/graceHot
while true
do
  echo `curl -s ${url}`
done
```

说明

*endpoint*为nacos-rest-consumer服务的调用地址，即ip:port。需根据实际nacos-rest-consumer地址进行替换。

e. 查看服务调用情况。

使用脚本**stat.sh**查看服务调用情况，脚本如下：

```
#!/bin/bash
endpoint=127.0.0.1:31021
watch curl ${endpoint}/stat
```

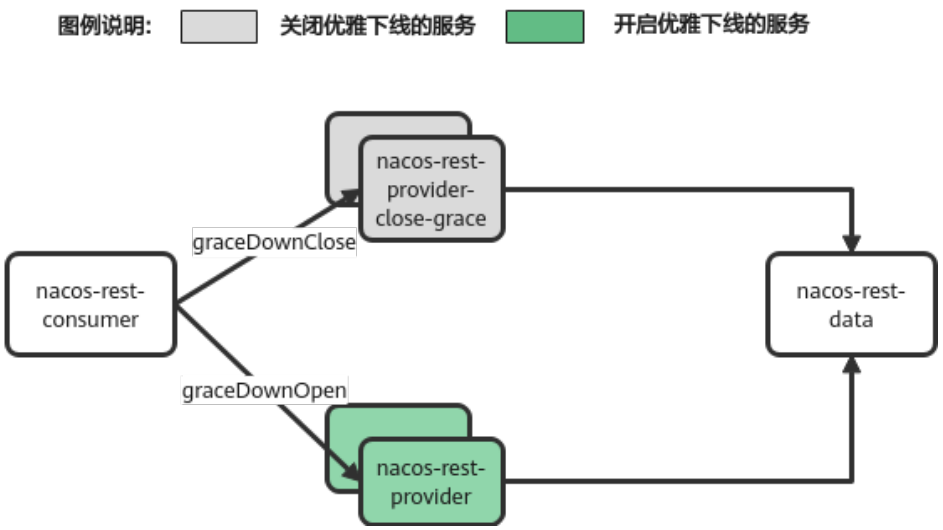
下图为某个时刻统计的调用结果：

```
{
        "graceHot":{
                "allCount":2821,
                "request":{
                        "[redacted].0.55:8009":{
                                "enableWarmUp":false,
                                "endpoint":"[redacted].0.55:8009",
                                "qps":85,
                                "requestCount":2710,
                                "serviceName":"nacos-rest-provider"
                        },
                        "[redacted].0.51:8004":{
                                "enableWarmUp":true,
                                "endpoint":"[redacted].0.51:8004",
                                "qps":8,
                                "requestCount":111,
                                "serviceName":"nacos-rest-provider"
                        }
                },
                "resource":"graceHot"
        }
}
```


上图中实例x.x.0.55:8009关闭了优雅上线， 实例x.x.0.51:8004开启了优雅上线，观察请求数（requestCount）与QPS，可观察到开启优雅上线的实例的QPS与请求数都小于关闭优雅上线的实例。持续观察流量请求情况，直到两个实例QPS基本持平则结束验证。

- 验证优雅下线能力。

  优雅下线验证部署图如下：



如上图，该套nacos应用有nacos-rest-consumer、nacos-rest-provider（两个实例）、nacos-rest-provider-close-grace（两个实例）以及nacos-rest-data服务，其中服务nacos-rest-provider-close-grace**关闭了优雅下线**，其他服务开启优雅下线。

说明

- 服务说明。

  nacos-rest-provider-close-grace与nacos-rest-provider**同属于一个jar应用**，nacos-rest-provider-close-grace请使用环境变量“spring_application_name=nacos-rest-provider-close-grace”进行**指定**。

- 关闭优雅下线的方法。

  添加环境变量“grace_rule_enableGraceShutdown=false”进行**指定**。

nacos-rest-consumer通过调用接口graceDownOpen测试验证优雅下线能力；调用接口graceDownClose对比未开启优雅下线时请求处理情况。查看调用情况可使用nacos-rest-consumer接口的stat方法进行请求统计。

a. 下载**demo应用**并打包。

b. 按照优雅下线验证部署图进行部署，并关闭nacos-rest-provider-close-grace的优雅下线能力。

c. 查看应用是否已全部接入ServiceComb引擎。

参考**查看微服务列表**查看您的应用是否已接入ServiceComb引擎。

d. 待应用接入ServiceComb引擎后，使用如下脚本**invoke.sh**模拟请求。

```
#!/bin/bash
endpoint=127.0.0.1:31021
openUrl=${endpoint}/graceDownOpen
closeUrl=${endpoint}/graceDownClose
while true
do
  echo `curl -s ${openUrl}`
  echo `curl -s ${closeUrl}`
done
```

说明

- *endpoint*为服务nacos-rest-consumer实例的请求地址，即ip:port。需根据实际nacos-rest-consumer地址进行替换。
- *openUrl*为开启优雅下线能力的调用接口。
- *closeUrl*为关闭优雅下线能力的调用接口。

e. 查看应用的请求情况。

参考如下脚本**stat.sh**查看请求调用情况:

```
#!/bin/bash
endpoint=127.0.0.1:31021
watch curl ${endpoint}/stat
```

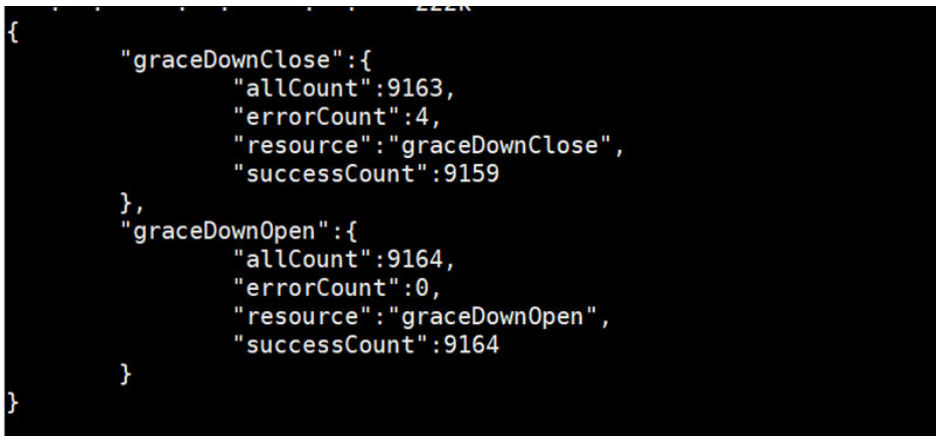
说明

*endpoint*为服务nacos-rest-consumer实例的请求地址，即ip:port。

f. 对服务进行缩容，模拟下线。

分别对nacos-rest-provider-close-grace与nacos-rest-provider进行缩容，将实例缩容为1个。

g. 使用**4**的脚本，查看服务调用情况。



上图中，graceDownClose为关闭优雅下线能力的请求统计，可以看到存在4个错误请求（errorCount），而下方graceDownOpen为开启优雅下线能力的请求统计，可以看到未出现错误请求。

3.2 SpringCloud 应用实现标签路由功能

3.2.1 功能介绍

在微服务存在多个版本、多个实例的情况下，需要管理服务之间的路由，达到无损升级、应用拨测等业务目的。Sermant Agent提供了标签路由的能力，标签路由通过匹配http请求头中的信息，把符合规则的流量转发到对应的标签应用中，从而实现标签路由的功能。

说明

- 此功能目前处于公测阶段，当前仅在华东-上海一支持。
- 当ServiceComb引擎为2.4.0及以上版本且未开启安全认证时，支持此功能。
- 目前支持将SpringCloud框架和Dubbo框架开发的java应用通过Sermant Agent注册到ServiceComb引擎中。
- Dubbo框架请参考**基于Sermant Agent接入的Dubbo应用实现标签路由功能**。
- 目前只支持基于线程池模型的异步调用。

版本支持

Spring Cloud Version	Spring Boot Version	Spring Cloud Openfeign Version	RestTemplate Version	Spring Cloud Loadbalancer Version	Spring Cloud Netflix Ribbon Version	Spring Cloud Gateway Version	Spring Cloud Netflix Zuul Version
Edgware.SR2+	1.5.x	1.4.3.RELEASE+	4.3.6.RELEASE+	-	1.4.3.RELEASE+	-	1.4.3.RELEASE+
Finchley.x	2.0.x	2.0.x	5.0.x	-	2.0.x	-	2.0.x
Greenwich.x	2.1.x	2.1.x	5.1.x	-	2.1.x	-	2.1.x
Hoxton.x	2.2.x、2.3.x	2.2.x	5.2.x	2.2.5.RELEASE+	2.2.x	2.2.x	2.2.x
2020.0.x	2.4.x、2.5.x	3.0.x	5.3.x	3.0.x	-	3.0.x	-
2021.0.0	2.6.x	3.1.x	5.3.x	3.1.x	-	3.1.x	-

说明

通过Sermant Agent注册到ServiceComb引擎中的应用可以使用标签路由功能。

3.2.2 基于虚拟机场景

前提条件

- 已创建ECS实例， 创建ECS请参考**ECS快速入门**。
- 已安装JDK（版本为1.8及以上版本）并配置环境变量，详情请参考**Java Downloads**。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- ECS与ServiceComb引擎处于相同的VPC网络下。
- Sermant Agent开源版本要求1.0.6及以上。

操作步骤

步骤1 安装Sermant Agent，请参考**安装Sermant Agent**。

步骤2 启动应用。

在应用的启动参数中添加如下参数，添加启动参数后，待应用启动完成。

```
-javaagent:${HOME}/java-agent/java-agent.jar=appName=default
-Ddynamic_config_serverAddress={CSE_CONFIG_CENTER_ENDPOINTS}
-Dregister.service.address={CSE_REGISTRY_ENDPOINTS}
-Dservice_meta_version={VERSION}
-Dservice_meta_parameters={PARAMETERS}
```

说明

相关配置介绍：

- appName为agent服务名称，该配置无需修改，使用default即可。
- ServiceComb引擎服务注册发现地址{CSE_REGISTRY_ENDPOINTS}与ServiceComb引擎配置中心地址{CSE_CONFIG_CENTER_ENDPOINTS}需替换为实际地址，可参考如下方式获取：
  - ServiceComb引擎服务注册发现地址：**获取ServiceComb引擎服务注册发现地址**。
  - ServiceComb引擎配置中心地址：**获取ServiceComb引擎配置中心地址**。
- {VERSION}需替换为服务注册时的版本号（形如a.b.c的格式，其中a、b、c均为数字，默认为1.0.0），标签应用需要修改为不同于正常应用的版本号。
- {PARAMETERS}需替换为服务注册时的自定义标签（形如tag1:value1,tag2:value2），即标签名与标签值以英文冒号分隔，多个标签之间以英文逗号分隔。
- 一般地，如果用版本号进行路由，则只需配置service_meta_version，如果用自定义标签进行路由，则只需配置service_meta_parameters。

----结束

3.2.3 基于 CCE 场景

前提条件

- 已创建CCE集群，详情请参考**创建CCE集群**。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- CCE与ServiceComb引擎处于相同VPC下。
- Sermant Injector版本要求1.0.11及以上，Sermant Agent镜像版本要求1.0.9及以上。

操作步骤

步骤1 为CCE集群安装sermant-injector，请参考**CCE部署场景接入指南**。

步骤2 为工作负载（deployment）中的微服务配置版本号或标签。

在下图所示位置为工作负载（deployment）配置环境变量，配置环境变量后，应用注册时，会使用该环境变量进行注册。

```
apiVersion: app/v1
kind: Deployment
metadata:
  name: cloud-providerB
  labels:
    app: cloud-providerB
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cloud-providerB
  template:
    metadata:
      labels:
        app: cloud-providerB
        sermant-injection: enabled
    spec:
      containers:
        - name: cloud-providerB
          image: cloud-providerB:1.0.0
          imagePullPolicy: IfNotPresent
          env:
            - name: "SERVICE_META_VERSION"
              value: "2.0.0"
        - name: "SERVICE_META_PARAMETERS"
            value: "group:gray"
          ports:
            - containerPort: 8004
      imagePullSecrets:
        - name: default-secret
```

其中：

- 键SERVICE_META_VERSION，值为服务注册的版本号（如a.b.c的格式，其中a、b、c均为数字），标签应用需要修改为不同于正常应用的版本号。
- 键SERVICE_META_PARAMETERS，值为服务注册时的自定义标签（形如tag1:value1,tag2:value2），即标签名与标签值以英文冒号分隔，多个标签之间以英文逗号分隔。

  说明

  当Sermant Agent为1.0.0及以下版本时，使用键为SERVICECOMB_INSTANCE_PROPS。
- 一般，如果用版本号进行路由，则只需配置SERVICE_META_VERSION，如果用自定义标签进行路由，则只需配置SERVICE_META_PARAMETERS。

步骤3 为工作负载（deployment）打上标签并重启相关服务。

在下图所示位置为工作负载（deployment）打上标签sermant-injection: enabled。打上标签后，sermant-injector会在Pod重启时自动挂载Sermant Agent，从而通过Sermant Agent注册到CSE上。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-router-gray-provider
  labels:
    app: spring-cloud-router-gray-provider
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-cloud-router-gray-provider
  template:
    metadata:
      labels:
        app: spring-cloud-router-gray-provider
        sermant-injection: enabled
```

----结束

3.2.4 配置路由规则

前提条件

SpringCloud应用已注册到ServiceComb引擎。

下发路由规则配置

下发路由规则配置，详情请参考**配置管理**。

路由规则说明示例：

```
---
- precedence: 2              # 优先级，数字越大，优先级越高。
  match:                     # 请求匹配规则。0..N个，不配置表示匹配。
    headers:                 # http header匹配
      id:                    # 如果配置了多个header，那么所有的header规则都必须和请求匹配
        exact: '1'           # 配置策略，等于1，详细配置策略见配置策略表。
        caseInsensitive: false  # false:不区分大小写（默认）,true:区分大小写。配置为false时，将统一转为大写进行比较
  route:                     # 路由规则
    - weight: 20             # 权重值
      tags:
        version: 1.0.0       # 实例标记。满足标记条件的实例放到这一组。
    - weight: 80             # 权重值
      tags:
        version: 1.0.1       # 实例标记。满足标记条件的实例放到这一组。
- precedence: 1
  route:
    - weight: 20
      tags:
        group: red
    - weight: 80
      tags:
        group: green
```

表 3-1 配置策略表

策略名	参数类型	策略值	匹配规则
精确匹配	int，String	exact	参数值等于配置值
正则	String	regex	参数值匹配正则表达式，由于部分正则表达式（如\w与\W等）区分大小写，所以使用正则策略时，请谨慎选择caseInsensitive（是否区分大小写）
不等于	int，String	noEqu	参数值不等于配置值
大于等于	int	noLess	参数值大于等于配置值
小于等于	int	noGreater	参数值小于等于配置值
大于	int	greater	参数值大于配置值
小于	int	less	参数值小于配置值

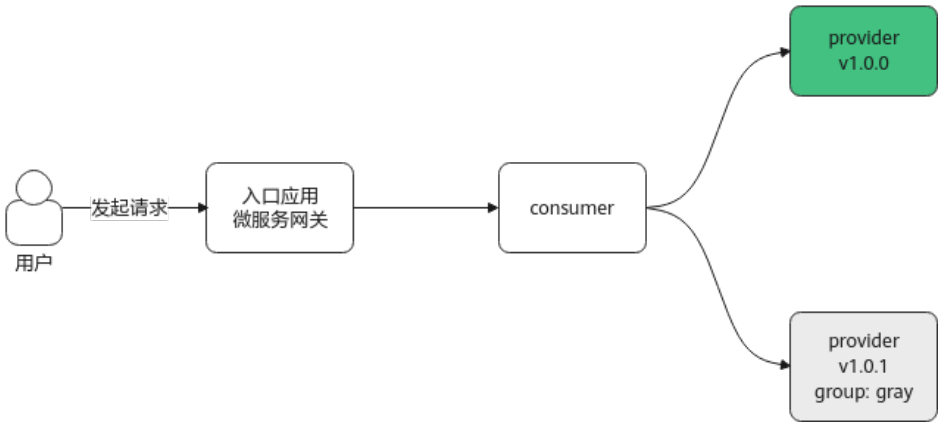
说明

- 配置路由规则时，需选择以下配置：
  - 配置项：servicecomb.routeRule.{目标应用名}。
  - 配置范围：应用级配置。
  - 应用：按应用和环境实际情况填写，默认为：default <空>。
  - 配置格式： yaml。
- 下发配置时，请去掉注释，否则会导致下发配置失败。
- 上述路由规则的解释如下：

  当请求头中包含id的时候，将80%流量转发到1.0.1版本，将20%流量转发到1.0.0 版本。当请求中不包含上述header的时候，将80%的流量转发到实例标签为group:green的实例，20%的流量转发到实例标签为group:red的实例。

3.2.5 验证标签路由功能

本文以consumer调用provider服务为例，在CCE场景验证标签路由功能（CCE场景部署请参考**基于CCE场景**）。部署图如下所示：



其中，微服务网关为Spring Cloud Netflix Zuul，provider存在2个版本，一个是版本为1.0.0（绿色背景）的provider，一个是版本为1.0.1、自定义标签为group:gray（灰色背景）的provider。

验证步骤

步骤1 下载**demo应用**并打包，镜像打包文件和k8s部署文件参考**部署文件**，制作docker镜像请参考**使用容器引擎客户端上传镜像**。

步骤2 按照部署图部署入口应用(spring-cloud-router-zuul)、consumer(spring-cloud-router-consumer)、provider(spring-cloud-router-provider v1.0.0)3个应用

步骤3 查看应用是否已全部接入CSE的ServiceComb引擎。

参考**查看微服务列表**查看您的应用是否已接入CSE的ServiceComb引擎。

步骤4 待应用接入ServiceComb引擎后，参考**配置路由规则**对spring-cloud-router-provider应用配置路由规则，路由规则如下：

```
---
- precedence: 1
  match:
    headers:
      id:
        exact: '1'
        caseInsensitive: false
  route:
    - tags:
        group: gray
      weight: 100
- precedence: 2
  match:
    headers:
      id:
        exact: '2'
        caseInsensitive: false
  route:
    - tags:
        version: 1.0.1
      weight: 100
```

说明

配置路由规则时，需选择以下配置：

- 配置项：servicecomb.routeRule.spring-cloud-router-provider。
- 配置范围：应用级配置。
- 应用：default <空>。
- 配置格式： yaml。

步骤5 部署标签应用gray-provider(服务名为spring-cloud-router-provider，版本为1.0.1，标签为group:gray)。

注意

必须先配置路由规则，再部署标签应用。否则，如果在规则生效之前部署标签应用，流量就会马上请求到标签应用。

步骤6 查看标签应用是否已接入ServiceComb引擎。

参考**查看微服务列表**查看服务名为spring-cloud-router-provider的服务，是否存在2个实例（v1.0.0与v1.0.1的实例）。

步骤7 验证标签路由功能。

- 使用以下命令访问标签为group:gray的provider：
  curl -H "id: 1" http://127.0.0.1:30000/consumer/hello/feign
- 使用以下命令访问版本为1.0.1的provider：
  curl -H "id: 2" http://127.0.0.1:30000/consumer/hello/feign
- 使用以下命令访问版本为1.0.0的provider：
  curl -H "id: 3" http://127.0.0.1:30000/consumer/hello/feign
  或者
  curl http://127.0.0.1:30000/consumer/hello/feign

说明

执行命令时，需要把ip替换成实际宿主机的ip，把端口替换成容器的8170端口映射到宿主机的端口，访问时，会请求到入口应用（spring-cloud-router-zuul）中。

可以得出结论，当请求头为id: 1或者id: 2时，会路由到版本为1.0.1的provider，当不满足以上条件时，会访问到版本为1.0.0的provider。

----结束

3.3 SpringCloud 应用实现全链路灰度

3.3.1 概述

背景信息

在微服务架构下，有些开发需求会使微服务调用链路上的多个微服务同时发生了改动，通常每个微服务都会有灰度环境或分组来接收灰度流量。此时希望通过进入上游灰度环境的流量，也能进入下游灰度的环境中，确保一个请求始终在灰度环境中传递，即使这个调用链路上有一些微服务没有灰度环境，这些应用请求在下游的时候依然能够回到灰度环境中。通过Sermant Agent提供的全链路灰度能力，可以在不需要修改任何您的业务代码的情况下，能够轻松实现上述能力。

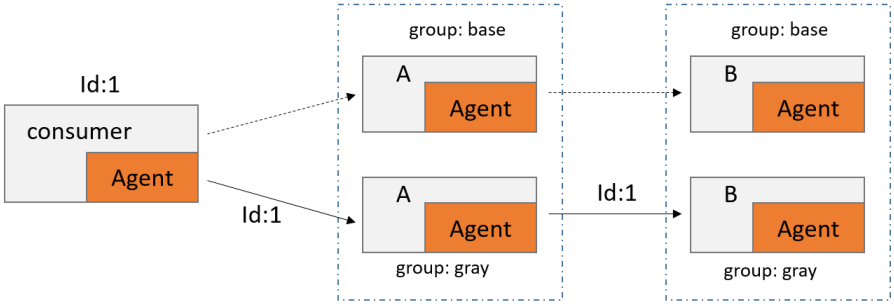
本文通过示例为您演示Sermant Agent全链路灰度功能。假设应用的架构由Sermant Agent以及后端的微服务架构（Spring Cloud）组成，后端调用链路：providerA->providerB，通过consumer调用后端服务，所有服务挂载的Sermant Agent使用CSE注册中心。

说明

- 此功能目前处于公测阶段，当前仅在华东-上海一支持。
- 当ServiceComb引擎为2.4.0及以上版本且未开启安全认证时，支持此功能。

方案

通过设置Head参数，保证访问到灰度环境实例：



前提条件

- 已创建云容器引擎（CCE），创建CCE请参考**创建CCE集群**。
- CCE集群版本需要大于等于1.15。
- 已安装kubectl命令，安装kubectl命令请参考**通过kubectl连接集群**相关操作。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- 本地编译构建打包机器环境已安装了Java JDK、Maven，并且能够访问Maven中央库。
- Sermant Agent开源版本要求1.0.6及以上。

3.3.2 准备工作

spring-cloud应用基于Sermant Agent接入CSE请参考**SpringCloud应用通过Sermant Agent接入CSE的ServiceComb引擎**。

部署 Demo 应用程序

步骤1 登录云容器引擎。

步骤2 选择已创建的CCE集群节点，单击目标集群名称。

步骤3 在集群管理页面左侧导航栏中，选择“工作负载 > 无状态负载 > YAML创建”。

步骤4 通过yaml模板进行相关配置，启动容器。

本文示例中部署一个comsumer端，providerA、providerB分别部署一个基线版本和一个灰度版本，样例参考**spring-cloud-demo**。

说明

- 因为需要使用CSE下发的灰度策略，所以Sermant Agent的配置中心必须为CSE的配置中心。
- 因为需要使用灰度发布功能，所以在发布服务时需要增加版本或者灰度标签，在本文示例中，consumer作为客户端，仅设置基线版本，服务端均设置基线版本、灰度版本，灰度版本添加版本号：SERVICE_META_VERSION=2.0.0，或者添加灰度标识：SERVICE_META_PARAMETERS=group:gray。

- provider A应用基线版本YAML配置(仅供参考)：

```
spec:
  template:
    metadata:
      labels:
        sermant-injection: enabled
    spec:
      containers:
```

```
      - name: providerA
        env:
          - name: "SERVICE_META_PARAMETERS"
            value: "group:base"
```

- provider A应用灰度版本YAML配置(仅供参考)：

```
spec:
  template:
    metadata:
      labels:
        sermant-injection: enabled
    spec:
      containers:
        - name: providerA
          env:
            - name: "SERVICE_META_PARAMETERS"
              value: "group:gray"
```

- provider B应用基线版本YAML配置(仅供参考)：

```
spec:
  template:
    metadata:
      labels:
        sermant-injection: enabled
    spec:
      containers:
        - name: providerB
          env:
            - name: "SERVICE_META_PARAMETERS"
              value: "group:base"
```

- provider B应用灰度版本YAML配置(仅供参考)：

```
spec:
  template:
    metadata:
      labels:
        sermant-injection: enabled
    spec:
      containers:
        - name: providerB
          env:
            - name: "SERVICE_META_PARAMETERS"
              value: "group:gray"
```

说明

- 在YAML文件中的“spec > template > metadata > labels”层级下加入内容：sermant-injection: enabled即可挂载Sermant Agent代理。
- SERVICE_META_PARAMETERS路由标签设置。

----结束

3.3.3 灰度规则设置

Sermant Agent 监听配置范围

Sermant Agent使用CSE作为配置中心时，监听的范围有以下三个：

1. app=default&environment=&service={服务名}
2. app=default&environment=
3. public=default

说明

- app对应为“应用”值；
- environment对应为“环境”；
- service对应为“微服务名称”；
- public为公共配置。

一般设置为app=default&environment=作为通用路由设置。

CSE 下发路由配置

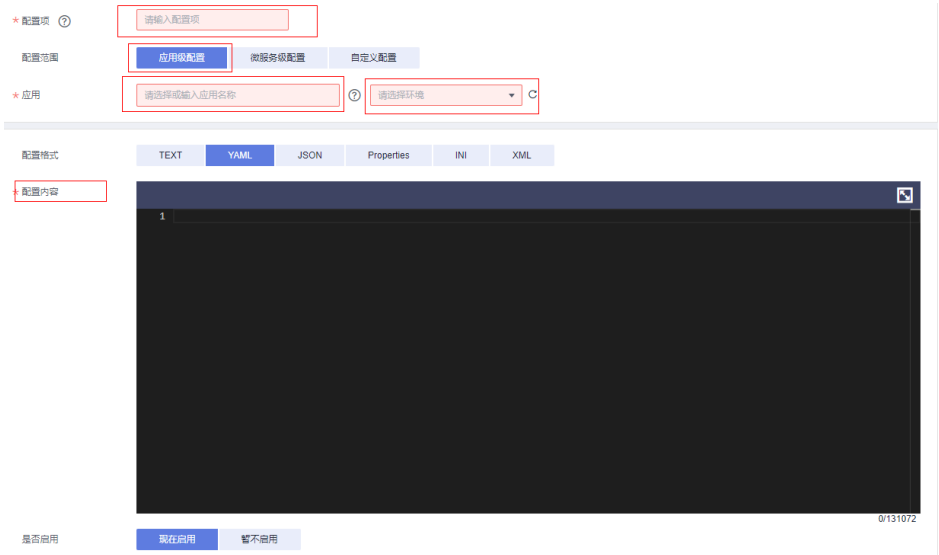
步骤1 登录微服务引擎控制台。

步骤2 在左侧导航栏选择“ServiceComb引擎”。

步骤3 单击待操作的引擎。

步骤4 选择“配置管理”。

步骤5 单击“新建配置”，配置以以下内容。



说明

- 配置项：servicecomb.routeRule；
- 配置项范围：选择“应用级配置”；
- 应用：应用名称输入“default”，环境选择“空”；
- 配置格式：选择YAML；
- 配置内容：参考**设置服务路由策略**配置路由策略。

----结束

设置服务路由策略

配置项：servicecomb.routeRule，配置内容：

```
providerA: |
  - precedence: 2
    match:
```



```
    headers:
      id:
        exact: '1'
        caseInsensitive: false
  route:
    - weight: 0
      tags:
        group: base
    - weight: 100
      tags:
        group: gray
- precedence: 1
  route:
    - weight: 100
      tags:
        group: base
    - weight: 0
      tags:
        group: gray
providerB: |
- precedence: 2
  match:
    headers:
      id:
        exact: '1'
        caseInsensitive: false
  route:
    - weight: 0
      tags:
        group: base
    - weight: 100
      tags:
        group: gray
- precedence: 1
  route:
    - weight: 100
      tags:
        group: base
    - weight: 0
      tags:
        group: gray
```

服务路由策略设置说明：

1. 请求头的id参数值精确匹配为1时，consumer的所有请求流量都是从providerA-gray流向providerB-gray。
2. 请求头的id参数值为其他任意值，consumer的所有请求流量都是从providerA流向providerB。

3.3.4 结果验证

- 访问灰度环境实例。

  执行以下命令请求consumer：

  curl -H "id:1" http://{endpoint}/sayHello?name={name}

  执行结果：

  ```
  consumer -> providerA[group:gray] -> providerB[group:gray]
  providerA-gray:name---{name}---version----2.0.0-------parameters----group:grayProviderB-gray:name---{name}---version---2.0.0---parameters---group:gray
  ```

- 访问基线环境实例。

  执行以下命令请求consumer：

  curl -H "id:2" http://{endpoint}/sayHello?name={name} 或者 curl http://{endpoint}/sayHello?name={name}

执行结果：

```
consumer -> providerA[group:base] -> providerB[group:base]
providerA:name---{name}---version----1.0.0-------parameters----group:baseProviderB:name---{name}---version---1.0.0---parameters---group:base
```

3.4 Dubbo 应用实现标签路由功能

3.4.1 功能介绍

在微服务存在多个版本、多个实例的情况下，需要管理服务之间的路由，达到无损升级、应用拨测等业务目的。Sermant Agent提供了标签路由的能力，标签路由通过匹配dubbo attachment中的信息，把符合规则的流量转发到对应的标签应用中，从而实现标签路由的功能。

须知

- 此功能目前处于公测阶段，当前仅在华东-上海一支持。
- 当ServiceComb引擎为2.4.0及以上版本且未开启安全认证时，支持此功能。
- 目前支持将SpringCloud框架和Dubbo框架开发的java应用通过Sermant Agent注册到ServiceComb引擎中。
- SpringCloud框架请参考**基于Sermant Agent接入的SpringCloud应用实现标签路由功能**。
- 目前只支持基于线程池模型的异步调用。

版本支持

当前支持的版本为Dubbo 2.6.x - 2.7.x。

说明

目前标签路由功能仅支持通过Sermant Agent注册到ServiceComb引擎中的应用。

3.4.2 基于虚拟机场景

前提条件

- 已创建ECS实例， 创建ECS请参考**ECS快速入门**。
- 已安装JDK（版本为1.8及以上版本）并配置环境变量，详情请参考**Java Downloads**。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- ECS与ServiceComb引擎处于相同的VPC网络下。
- Sermant Agent开源版本要求1.0.6及以上。

操作步骤

步骤1 安装Sermant Agent，请参考**安装Sermant Agent**。

步骤2 启动应用。

在应用的启动参数添加如下参数，添加启动参数后，待应用启动完成。

```
-javaagent:${HOME}/java-agent/java-agent.jar=appName=default
-Ddynamic_config_serverAddress={CSE_CONFIG_CENTER_ENDPOINTS}
-Dregister.service.address={CSE_REGISTRY_ENDPOINTS}
-Dgrace_rule_enableSpring=false
-Dservice_meta_version={VERSION}
-Dservice_meta_parameters={PARAMETERS}
```

说明

相关配置介绍：

- appName为agent服务名称，该配置无需修改，使用default即可。
- ServiceComb引擎服务注册发现地址{CSE_REGISTRY_ENDPOINTS}与ServiceComb引擎配置中心地址{CSE_CONFIG_CENTER_ENDPOINTS}需替换为实际地址，可参考如下方式获取：
  - ServiceComb引擎服务注册发现地址：**获取ServiceComb引擎服务注册发现地址**。
  - ServiceComb引擎配置中心地址：**获取ServiceComb引擎配置中心地址**。
- grace_rule_enableSpring为SpringCloud框架优雅上下线功能，所以Dubbo框架需要手动关闭（设置为fasle），否则可能会存在端口冲突的问题。
- {VERSION}需替换为服务注册时的版本号（形如a.b.c的格式，其中a、b、c均为数字，默认为1.0.0），标签应用需要修改为不同于正常应用的版本号。
- {PARAMETERS}需替换为服务注册时的自定义标签（形如tag1:value1,tag2:value2），即标签名与标签值以英文冒号分隔，多个标签之间以英文逗号分隔。
- 一般地，如果用版本号进行路由，则只需配置service_meta_version，如果用自定义标签进行路由，则只需配置service_meta_parameters。

----结束

3.4.3 基于 CCE 场景

前提条件

- 已创建CCE集群，详情请参考**创建CCE集群**。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参考**快速创建ServiceComb引擎**。
- CCE与ServiceComb引擎处于相同VPC下。
- Sermant Injector版本要求1.0.11及以上，Sermant Agent镜像版本要求1.0.9及以上。

操作步骤

步骤1 为CCE集群安装sermant-injector，请参考**CCE部署场景接入指南**。

步骤2 为工作负载（deployment）中的微服务配置版本号或标签。

在下图所示位置为工作负载（deployment）配置环境变量，配置环境变量后，应用注册时，会使用该环境变量进行注册。

```
apiVersion: app/v1
kind: Deployment
```

```
metadata:
  name: dubbo-providerB
  labels:
    app: dubbo-providerB
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dubbo-providerB
  template:
    metadata:
      labels:
        app: dubbo-providerB
        sermant-injection: enabled
    spec:
      containers:
        - name: dubbo-providerB
          image: dubbo-providerB:1.0.0
          imagePullPolicy: IfNotPresent
          env:
            - name: "SERVICE_META_VERSION"
              value: "2.0.0"
        - name: "SERVICE_META_PARAMETERS"
            value: "group:gray"
          ports:
            - containerPort: 8004
      imagePullSecrets:
        - name: default-secret
```

其中：

- 键SERVICE_META_VERSION，值为服务注册的版本号（如a.b.c的格式，其中a、b、c均为数字），标签应用需要修改为不同于正常应用的版本号。
- 键SERVICE_META_PARAMETERS，值为服务注册时的自定义标签（形如tag1:value1,tag2:value2），即标签名与标签值以英文冒号分隔，多个标签之间以英文逗号分隔。

  说明

  当Sermant Agent为1.0.0及以下版本时，使用键为SERVICECOMB_INSTANCE_PROPS。

- 一般，如果用版本号进行路由，则只需配置SERVICE_META_VERSION，如果用自定义标签进行路由，则只需配置SERVICE_META_PARAMETERS。

步骤3 为工作负载（deployment）打上标签并重启相关服务。

在下图所示位置为工作负载（deployment）打上标签sermant-injection: enabled。打上标签后，sermant-injector会在Pod重启时自动挂载Sermant Agent，从而通过Sermant Agent注册到CSE上。


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gray-dubbo-b
  labels:
    app: gray-dubbo-b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gray-dubbo-b
  template:
    metadata:
      labels:
        app: gray-dubbo-b
        sermant-injection: enabled
```

----结束

3.4.4 配置路由规则

前提条件

Dubbo应用已注册到ServiceComb引擎。

下发路由规则配置

下发路由规则配置，详情参考**配置管理**。

路由规则说明示例：

```
---
- precedence: 2                # 优先级，数字越大，优先级越高。
  match:                       # 请求匹配规则。0..N个，不配置表示匹配。
    attachments:               # dubbo attachment匹配
      id:                      # 如果配置了多个attachment，那么所有的attachment规则都必须和请求匹配
        exact: '1'             # 配置策略，等于1，详细配置策略参考配置策略表。
        caseInsensitive: false  # false:不区分大小写（默认）,true:区分大小写。配置为false时，将统一转为大写进行比较
  route:                       # 路由规则
    - weight: 20               # 权重值
      tags:
        version: 1.0.0         # 实例标记。满足标记条件的实例放到这一组。
    - weight: 80               # 权重值
      tags:
        version: 1.0.1         # 实例标记。满足标记条件的实例放到这一组。
- precedence: 1
```

```
route:
  - weight: 20
    tags:
      group: red
  - weight: 80
    tags:
      group: green
```

表 3-2 配置策略表

策略名	参数类型	策略值	匹配规则
精确匹配	int，String	exact	参数值等于配置值
正则	String	regex	参数值匹配正则表达式，由于部分正则表达式（如\w与\W等）区分大小写，所以使用正则策略时，请谨慎选择caseInsensitive（是否区分大小写）
不等于	int，String	noEqu	参数值不等于配置值
大于等于	int	noLess	参数值大于等于配置值
小于等于	int	noGreater	参数值小于等于配置值
大于	int	greater	参数值大于配置值
小于	int	less	参数值小于配置值

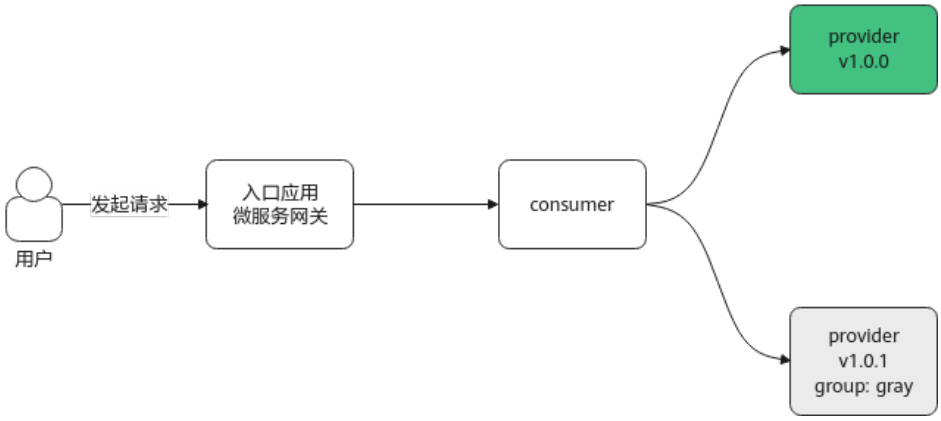
说明

- 配置路由规则时，需选择以下配置：
  - 配置项：servicecomb.routeRule.{目标应用名}。
  - 配置范围：应用级配置。
  - 应用：按应用和环境实际情况填写，默认为：default <空>。
  - 配置格式： yaml。
- 下发配置时，请去掉注释，否则会导致下发配置失败。
- 上述路由规则的解释如下：

  当attachment中包含id的时候，将80%流量转发到1.0.1版本，将20%流量转发到1.0.0版本。当请求中不包含上述attachment的时候，将80%的流量转发到实例标签为group:green的实例，20%的流量转发到实例标签为group:red的实例。

3.4.5 验证标签路由功能

本文以consumer调用provider服务为例，在CCE场景验证标签路由功能（CCE场景部署请参考**基于CCE场景**）。部署图如下所示：



其中，微服务网关为Spring Cloud Gateway，provider存在2个版本，一个是版本为1.0.0（绿色背景）的provider，一个是版本为1.0.1、自定义标签为group:gray（灰色背景）的provider。

验证步骤

步骤1 下载**demo应用**并打包，镜像打包文件和k8s部署文件参考**部署文件**，制作docker镜像请参考**使用容器引擎客户端上传镜像**。

步骤2 按照部署图部署入口应用(gateway)、consumer(dubbo-a)、provider(dubbo-b v1.0.0)3个应用。

步骤3 查看应用是否已全部接入CSE的ServiceComb引擎。

参考**查看微服务列表**查看您的应用是否已接入CSE的ServiceComb引擎。

步骤4 待应用接入ServiceComb引擎后，参考**配置路由规则**对dubbo-b应用配置路由规则，路由规则如下：

```
---
- precedence: 1
  match:
    attachments:
      id:
        exact: '1'
        caseInsensitive: false
  route:
    - tags:
        group: gray
      weight: 100
- precedence: 2
  match:
    attachments:
      id:
        exact: '2'
        caseInsensitive: false
  route:
    - tags:
        version: 1.0.1
      weight: 100
```


说明

配置路由规则时，需选择以下配置：

- 配置项：servicecomb.routeRule.dubbo-b。
- 配置范围：应用级配置。
- 应用：default <空>。
- 配置格式： yaml。

步骤5 部署标签应用gray-provider(服务名为dubbo-b，版本为1.0.1，标签为group:gray)。

须知

必须先配置路由规则，再部署标签应用。否则，如果在规则生效之前部署标签应用，流量就会马上请求到标签应用。

步骤6 查看标签应用是否已接入ServiceComb引擎。

参考**查看微服务列表**查看服务名为dubbo-b的服务，是否存在2个实例（v1.0.0与v1.0.1的实例）。

步骤7 验证标签路由功能。

- 使用以下命令访问标签为group:gray的provider：
  curl -H 'id: 1' http://127.0.0.1:30000/object
- 使用以下命令访问版本为1.0.1的provider：
  curl -H 'id: 2' http://127.0.0.1:30000/object
- 使用以下命令访问版本为1.0.0的provider：
  curl -H 'id: 3' http://127.0.0.1:30000/object
  或者
  curl http://127.0.0.1:30000/object

说明

执行命令时，需要把ip替换成实际宿主机的ip，把端口替换成容器的28030端口映射到宿主机的端口，访问时，会请求到入口应用（gateway）中。

可以得出结论，当请求头为id: 1或者id: 2时，会路由到版本为1.0.1的provider，当不满足以上条件时，会访问到版本为1.0.0的provider。

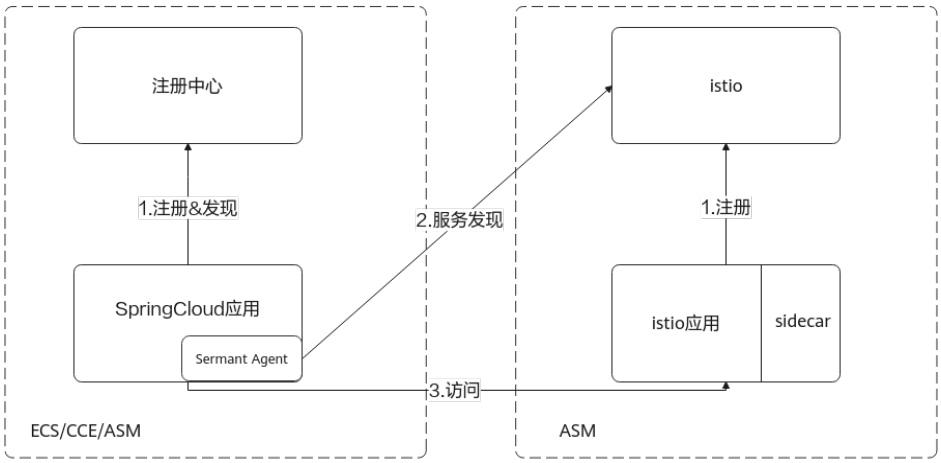
----结束

3.5 SpringCloud 应用对接 ASM 进行服务发现

功能介绍

随着istio服务网格的发展，越来越多的应用都会接入服务网格，Sermant Agent提供了一种能让SpringCloud应用也能访问服务网格中的应用的解决方案。

SpringCloud应用访问istio应用的部署图如下：



istio应用需要注册到istio中，不限制语言，SpringCloud应用不必注册到istio中，SpringCloud应用部署的环境可以是ECS/CCE/ASM。SpringCloud应用会通过Sermant Agent进行istio中的服务发现，然后SpringCloud应用便可以像调用其它SpringCloud应用一样，通过服务名调用istio中的服务。

说明

- 此功能目前处于公测阶段，当前仅在华东-上海一支持。
- 当ServiceComb引擎为2.x版本且未开启安全认证时，支持此功能。
- Sermant Injector版本要求1.0.11及以上，Sermant Agent镜像版本要求1.0.9及以上。

版本支持

Spring Cloud Edgware.SR2 - 2021.0.0

前提条件

- 已创建ASM，具体操作请参考**创建ASM**。
- 已部署istio应用，具体操作请参考**创建无状态负载**，并保证istio应用能注册到istio中。
- 已创建未开启安全认证的ServiceComb引擎实例，详情请参见**快速创建ServiceComb引擎**。
- SpringCloud应用环境的网络能够访问istio应用所在的ASM。
- 确保CCE集群的安全组正常放通，详情请参考**集群安全组规则配置** 。

操作步骤

步骤1 挂载Sermant Agent并启动SpringCloud应用，请参考**SpringCloud应用通过Sermant Agent接入ServiceComb引擎**。

步骤2 开启Sermant Agent istio服务发现功能，相关配置如下所示，可在启动SpringCloud应用时通过环境变量或者-D参数的形式进行配置。

配置项	配置说明	默认值	备注
XDS_PLUGIN_ENABLED	是否开启istio服务发现	false	开启istio服务发现需要设置成true

配置项	配置说明	默认值	备注
XDS_PLUGIN_SECURE	访问istio时，是否开启tls	true	当前ASM只支持tls访问
XDS_PLUGIN_ADDRESS	istio地址	istiod.istio-system.svc:15012	-
XDS_PLUGIN_NAMESPACE	istio系统所在的命名空间	istio-system	-
XDS_PLUGIN_TOKEN	istio token，开启tls时必填	<空>	istio token为istio-system命名空间的**default-token-xxxxx**（xxxx为随机字符串）中的token值，可在CCE管理控制台，在左侧导航栏选择“配置项与密钥”，在“密钥”页签下选择istio-system命名空间后查看。

说明

当CCE集群版本大于等于1.25时，需要手动创建 istio token，操作如下：

1. **通过kubectl连接istio应用所在的集群**或者**通过CloudShell连接istio应用所在的集群**。
2. 执行以下命令创建token：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: default-token
  namespace: istio-system
  annotations:
    kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
EOF
```

3. 执行成功后，便可在CCE管理控制台，单击istio应用所在的CCE集群，在左侧导航栏选择“配置项与密钥”在“密钥”页签下选择istio-system命名空间后查看到default-token，单击default-token，便可查看到token值。

步骤3 完成SpringCloud应用与istio应用部署之后，SpringCloud应用便可以通过服务名调用istio应用。例如，通过spring-cloud-starter-openfeign调用istio应用（假设服务名为istio-provider），请求路径为"/hello"，请求方法为get的接口的示例如下：

```
@FeignClient(name = "istio-provider")
public interface IstioClient {
    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    String hello();
}
```

您也可以选用RestTemplate等其它支持SpringCloud服务发现机制的客户端，在此不做限制。

说明

SpringCloud应用调用istio应用的服务名需要与istio中的注册的服务名保持一致。

----结束

更新 token（可选）

按需更新token，由于istio token与ASM相关，如果重装了ASM或者由于其它原因导致了token发生变化，则会导致istio服务发现失效，此时可以使用配置管理，在不重启服务的情况下，动态更新token。在更新token前需要SpringCloud应用已接入ServiceComb引擎。

更新token，即创建token配置，具体操作请参考**配置管理**。

token配置示例：

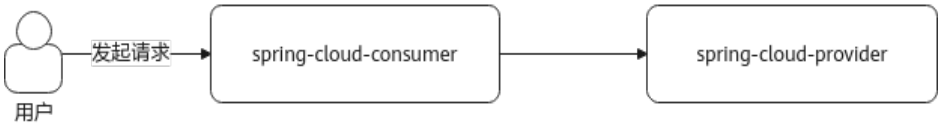
```
token: "xxxx"
```

说明

- 配置项：sermant.xds.discovery。
- 配置范围：按需选择。
- 应用：按应用和环境实际情况填写，默认为：default <空>。
- 配置格式： yaml。
- 更新token时，请把xxxx替换为实际的token，token值来源请参考**配置说明**。

验证 ASM 服务发现功能

本文以spring-cloud-router-consumer调用spring-cloud-router-provider服务为例，在CCE场景验证ASM服务发现（CCE场景部署请参考**基于CCE场景**），部署图如下所示：



其中spring-cloud-router-consumer通过Sermant Agent注册到ServiceComb中并对ASM进行服务发现，spring-cloud-router-provider不会挂载Sermant Agent，不会注册到ServiceComb中，只会部署到ASM中。

说明

为了方便验证，spring-cloud-router-consumer与spring-cloud-router-provider部署到了一个CCE集群中，在实际环境中，不局限于这一点，只要保证SpringCloud应用能够访问istio，并且保证SpringCloud应用能够通过istio应用的pod ip进行访问即可。

步骤1 下载**demo应用**并打包，在**镜像文件夹**获取制作镜像的文件，制作docker镜像请参考**使用容器引擎客户端上传镜像**。

步骤2 **创建ASM**并部署spring-cloud-router-provider，部署时的istio服务名需要为spring-cloud-router-provider。

步骤3 部署spring-cloud-router-consumer，部署时需要在“spec > template > metadata > labels”层级下加入：sermant-injection: enabled以挂载Sermant Agent，在“spec > template > metadata > annotations”层级下加入：sidecar.istio.io/inject: 'false'以屏蔽ASM服务发现。

步骤4 进入ASM管理控制台，选择您的ASM集群，在服务管理页面中，命名空间选择default，如果您的部署正确，则可以看到spring-cloud-rouer-provider状态为正常，spring-cloud-router-consumer的状态为异常（不会注入sidecar）。

步骤5 查看spring-cloud-router-consumer是否已接入ServiceComb引擎。

参考**查看微服务列表**查看spring-cloud-router-consumer是否已接入ServiceComb引擎。

步骤6 验证是否对spring-cloud-rouer-provider进行了服务发现。

使用以下命令访问spring-cloud-router-consumer：

curl http://127.0.0.1:31002/hello/feign

说明

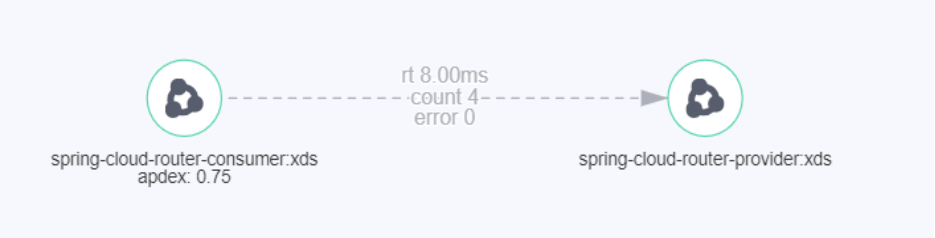
执行命令时，需要把ip替换成实际宿主机的ip。

步骤7 如果返回结果如下所示，则说明验证成功。

```
Hello world! My port is 8162, my version is 1.0.0.
```

说明

如果您把spring-cloud-router-consumer与spring-cloud-router-provider接入了**APM**，也可以在**应用拓扑**中查看到如下调用关系：



----结束

4 迁移应用到 CSE Nacos

4.1 用户自建 Nacos 迁移到 CSE Nacos

4.1.1 方案介绍

方案简介

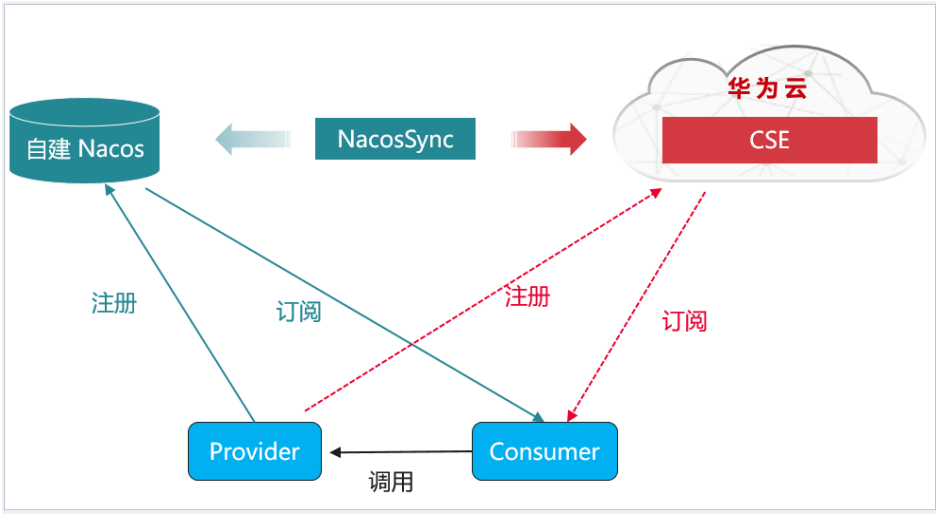
本文介绍一种基于Nacos Sync的注册中心迁移方案，**方案适用于在华为云上自建Nacos的用户**。

Nacos Sync是一款开源组件，支持注册中心的双向同步与平滑迁移，通过配置同步任务的方式，进行服务的迁移，Nacos Sync能使源集群的服务信息和CSE Nacos的服务信息保持同步，从而实现自建配置注册中心和CSE Nacos之间的平滑迁移，适用于对业务不中断有诉求的用户。

说明

Nacos Sync支持开源Nacos版本为1.4.1至2.1.0。

如图示，用户通过NacosSync组件，将注册订阅在自建Nacos的服务实例，实时同步至CSE的Nacos，实现服务实例的双向注册。



前提条件

- 自建Nacos迁移，需创建CSE Nacos服务作为目标端，创建CSE Nacos实例，详情请参考**快速创建Nacos引擎**。
- 使用Nacos Sync组件进行同步迁移，Nacos Sync组件依赖Mysql进行部署，需要创建Mysql实例，并创建数据库与表，请参考**MySQL实例创建**创建按需计费的MySQL 5.7版本的实例，其规格为通用型2vCPUs|4GB，参考**安全组配置**修改安全组配置，开放所有端口。
- 组件部署在ECS中，请参考**购买弹性云服务器**创建按需计费、X86架构的4vCPUs|8GB规格、公共镜像为EulerOS-EulerOS 2.9 64bit的弹性云服务器，且网络配置需**绑定弹性公网IP**，使用与MySQL同样的**安全组**，还需参考**入门实践**安装JRE 8。
- **同步任务需要双边网络互通，所以您自建的Nacos、CSE Nacos、Mysql和ECS实例需要处于同一个网段下，能够网络互通，并且组件的安全组配置完成。**

4.1.2 部署 Nacos Sync 同步组件

操作步骤

步骤1 获取Nacos Sync部署包。

在**nacos-sync**中获取Nacos Sync部署包，选择**nacos-sync-0.4.8.tar.gz**进行下载。

步骤2 创建Nacos Sync所需要的数据库与表。

1. 创建数据库实例，具体操作请参考**创建Mysql数据库实例**。
2. 连接MySQL实例，具体操作请参考**连接MySQL实例**。
3. 创建数据库，数据库名称为nacos_sync，字符集选择utf8mb4，具体操作请参考**创建数据库**。
4. 解压nacos-sync-0.4.8.tar.gz，获取“nacos-sync/bin/”下的nacosSync.sql文件，并执行该文件，具体操作请参考**执行SQL**，执行完成后，会生成三张表。

```
cluster # 存储集群信息
task    #存储同步任务信息
system_config # 系统配置信息
```

步骤3 部署Nacos Sync至ECS服务器。

1. 请参考**Linux弹性云服务器登录方式概述**选择相应方式登录弹性云服务器。
2. 将获取的压缩包上传至ECS服务器的“/tmp/”文件夹下。
3. 在压缩包所在路径下，执行解压命令，解压至当前文件夹下。

   cd /tmp/

   tar -zxvf nacos-sync-0.4.8.tar.gz
4. 修改application.properties配置文件的数据库信息。

   cd nacos-sync/conf

   vi application.properties

   修改文件中的数据库连接信息为**步骤2**创建的数据库信息，然后保存。

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/nacos_sync?characterEncoding=utf8   # 修改为申请的数据库ip信息与自创建的数据库信息
spring.datasource.username=root   # 数据库用户名
spring.datasource.password=xxxxxx   # 数据库密码
```

5. 启动Nacos Sync服务。

   cd ..

   cd bin/

sh startup.sh start

日志的路径在“nacos-sync/logs”下，可检查是否有异常信息。

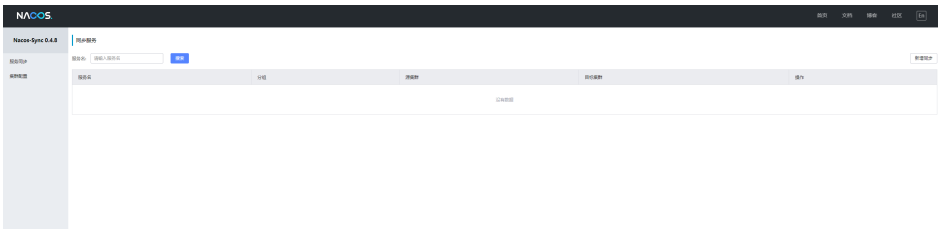
可修改startup.sh中的JAVA_OPT参数，自定义设置JVM堆内存大小。

6. 访问Nacos Sync服务地址。

   访问链接为IP+端口号，其中IP为ECS实例绑定的**弹性公网IP**，端口号为“application.properties”文件中配置的端口号。

```
server.port=8083
server.servlet.context-path=/

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=false

spring.cloud.discovery.enabled=false


spring.datasource.url=jdbc:mysql://127.0.0.1:3306/nacos_sync?characterEncoding=utf8
spring.datasource.username=root
spring.datasource.password=root
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
~
~
~
~
~
~
```

   Nacos Sync的服务界面如下：



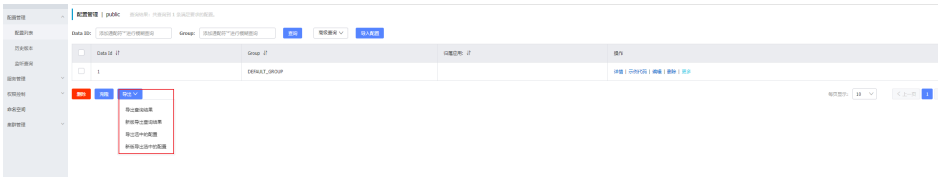
----结束

4.1.3 迁移操作

配置迁移

配置迁移可使用Nacos自身的导出功能将配置文件导出，再通过CSE Nacos的导入配置功能导入。

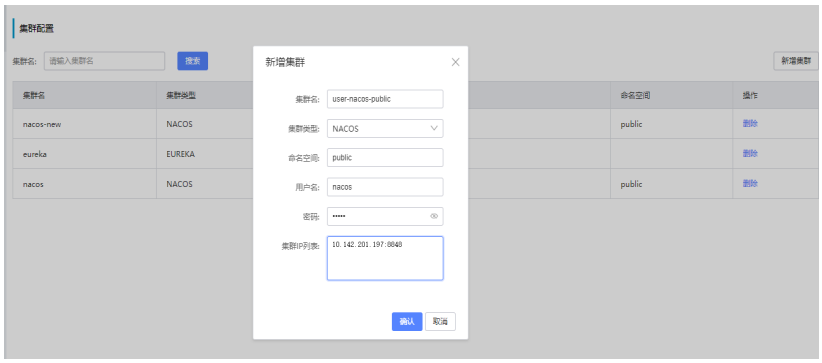
1. 自建Nacos配置导出：可在自建Nacos的配置管理页面，选择“导出 > 导出选中配置”，将配置文件导出。



2. 配置导入CSE Nacos，参考**导入配置**，将**1**中获取的配置文件导入到CSE Nacos引擎中。

应用迁移

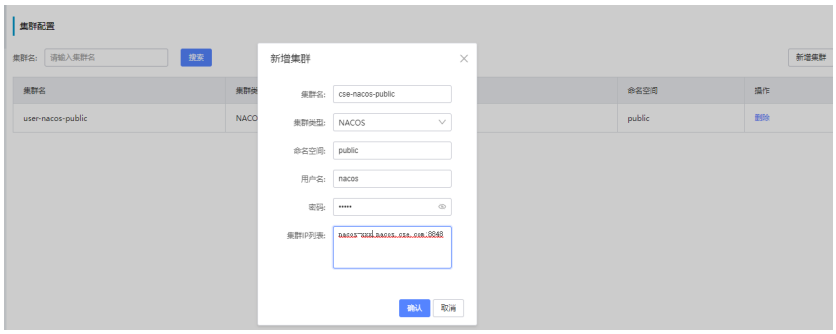
1. 添加注册中心集群信息。
   a. 参考**步骤3.6**访问Nacos Sync服务。
   b. 在左侧导航栏选择“集群配置”，增加一个您自建的Nacos集群，若使用多个命名空间，需要多次添加。



说明

- **命名空间为您自建待迁移应用所在的命名空间ID**，用户密码为默认管理员的用户密码，若未使用鉴权功能修改过，默认为nacos，集群IP为您自己部署Nacos的IP与端口号。
- 集群名字可以自定义，但是一旦确认，不能被修改，否则基于此集群增加的任务，在Nacos Sync重启后，将不会恢复成功。

c. 同**1.b**增加一个CSE Nacos的集群，若使用多个命名空间，需要多次添加。



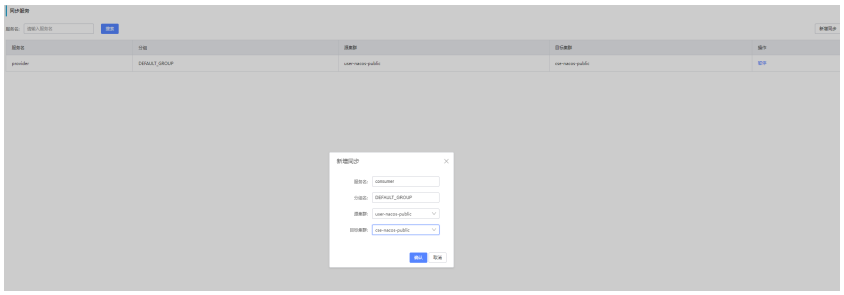
说明

命名空间为您想要迁移至CSE Nacos的命名空间ID，用户密码为默认管理员的用户密码，若未使用鉴权功能修改过，默认为nacos，集群IP为CSE Nacos的内网地址+端口号，其获取方式可参考**查看Nacos引擎详细信息**，端口号为8848。

d. 添加完成后，可以在列表中查询到增加的集群。

2. 添加同步任务。

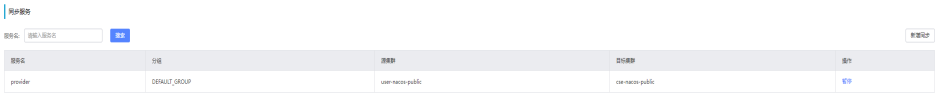
a. 参考**步骤3.6**访问Nacos Sync服务，增加一个同步任务，从您自建的Nacos集群同步到CSE Nacos集群，同步的粒度是服务，其中源集群为您自建的Nacos，目标集群为CSE Nacos。



说明

目标集群的服务名与分组名，需要与源集群的服务名和分组名一致，否则同步不了，不同的服务需要新建不同的任务。

添加完成之后，可以在服务同步列表中，查看已添加的同步任务：



b. 确认是否完成数据同步，检查数据是否同步成功到CSE Nacos集群，可参考**查看服务列表**查看CSE Nacos中出现同步任务中的服务，即表示同步成功。

c. 对所有待迁移的应用进行迁移与同步任务，并准备升级客户端应用的Nacos连接配置。

注册中心替换

1. 自建Nacos当中的服务提供实例，需要增加CSE Nacos到自建Nacos的反向同步任务，以保证业务不中断。

2. 观察自建的Nacos集群和CSE 的 Nacos集群，查看两个集群的服务实例是否一致，健康状态是否正常。
3. 修改客户端连接Nacos的配置，将服务中原有的自建Nacos集群访问地址更新成CSE Nacos集群的客户端访问地址，获取方式请参考**查看Nacos引擎详细信息**。

```
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848    # 将该地址改为CSE Nacos的集群地址
```

4. 依次重新部署自建Nacos的服务实例，此时您的服务只在CSE Nacos 集群中进行了注册，完成了替换。

   图示为Nacos Sync同步后的CSE Nacos服务实例。

说明

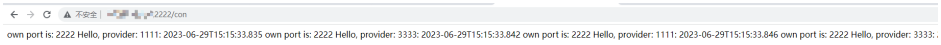
进行服务端的重启替换时，需新增从CSE Nacos到自建Nacos的服务端任务，保证业务不中断。

Nacos Sync同步任务正常后，在升级provider服务时，新建CSE Nacos到自建Nacos的同步任务，再逐步替换配置、重启，最后下线自建Nacos与Nacos Sync组件。

4.1.4 结果验证及资源释放

结果验证

1. 验证应用接入CSE的Nacos专享版引擎。
   参考**服务管理**检查应用是否接入Nacos专享版引擎。
2. 验证服务调用结果。
   通过Nacos服务中心Consumer调用Provider服务，在浏览器中输入http://Nacos注册中心的Consumer服务外部端点地址，显示服务调用成功：



资源释放

迁移完成后，即可释放Mysq实例与Nacos Sync部署所在的ECS实例。

1. 参考**删除弹性云服务器**将ECS实例释放。
2. 参考**删除实例**将Mysql实例释放。

5 通过应用网关访问 gRPC 协议的服务

概述

应用网关支持访问gRPC协议的服务。下文以固定地址服务为例，演示如何配置gRPC协议服务的接入与路由。

前提条件

- 已创建ECS实例用来部署服务， 创建ECS请参考**购买弹性云服务器**。
- 已创建应用网关，具体操作请参考考**创建应用网关**。
- ECS、应用网关处于相同的VPC网络下。

操作步骤

步骤1 在**GRPC获取路径**下载生产者端geeter_server和消费端greeter_client的代码。

步骤2 在本地go环境构建出二进制软件包grpc-provider和 grpc-consumer，执行命令如下。

GOOS=linux go build -o grpc-provider main.go

GOOS=linux go build -o grpc-consumer main.go

步骤3 登录Linux弹性云服务器。

请参考**Linux弹性云服务器登录方式概述**选择相应方式登录弹性云服务器。

步骤4 将构建好的软件包上传到ECS上合适的路径下，如根目录“/”。

步骤5 执行如下命令对grpc-provider和grpc-consumer赋予可执行权限。

chmod +x grpc-provider

chmod +x grpc-consumer

步骤6 执行命令启动生产者。

./grpc-provider

此时gRpc的生产者端已经被启动了，监听端口是50051。

```
[root@e      ~]# ./grpc-provider
2023/12/13 10:29:08 server listening at [::]:50051
2023/12/13 10:32:13 Received: world
```


步骤7 参考**创建服务**创建来源类型为“固定IP”的服务。其中“服务地址”为ECS的IP地址，端口为grpc-provider的监听端口，即50051。

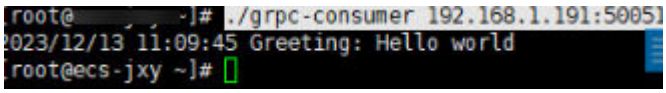
步骤8 参考**创建路由**在应用网关中创建路由，域名为“*”的路由，在路由规则中，添加路由规则。

具体路由规则为：“服务地址”的匹配类型选择“前缀匹配”，服务地址输入“/”，“请求方法”选择“POST”、“GET”、“DELETE”和“PUT”，“目标服务”为**步骤7**中创建的服务，“分组”选择“不限制”，“权重”为100。

步骤9 执行命令使用consumer端进行访问。

./grpc-consumer {网关IP}:50051

其中网关IP是网关的私网IP，访问成功则有以下结果。



----结束

