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 ServiceComb 引擎托管应用

1.1 托管 Spring Cloud 应用

1.1.1 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。

1.1.2 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CSE的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>
```

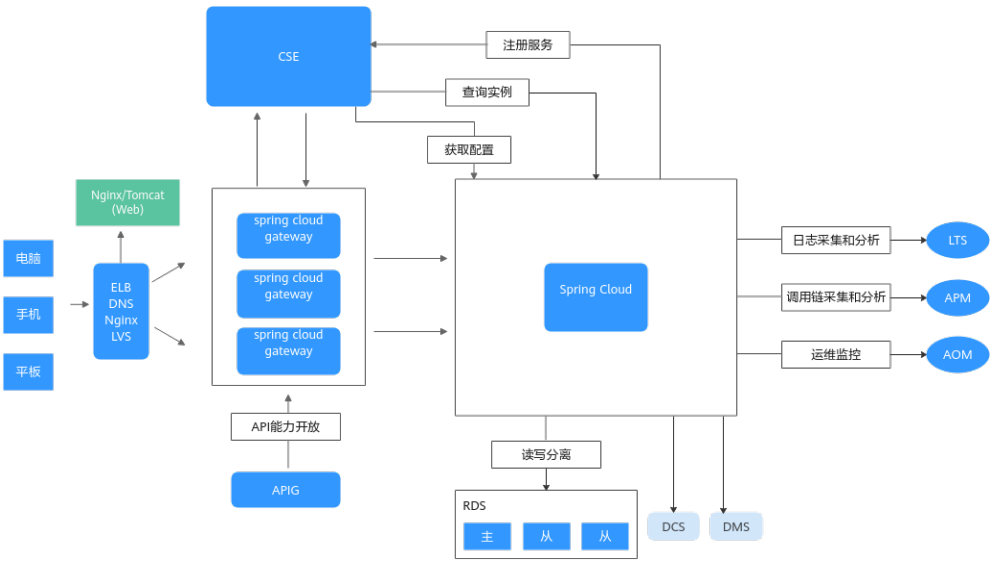
```
</dependency>

<!-- 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>
```

1.1.3 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。

1.1.4 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。

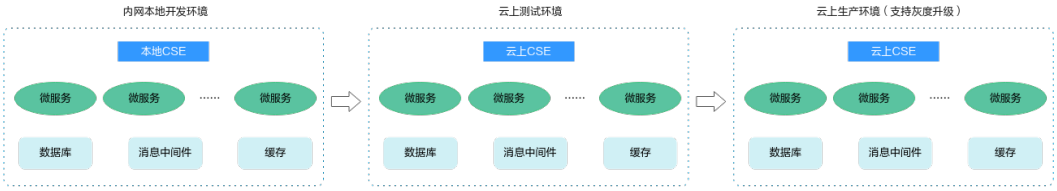
关于三方软件管理的更多内容，请参考**Java Chassis兼容问题和兼容性策略**。

1.1.5 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 1-1 开发环境



结合项目经验，一般会按照**图1-1**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。

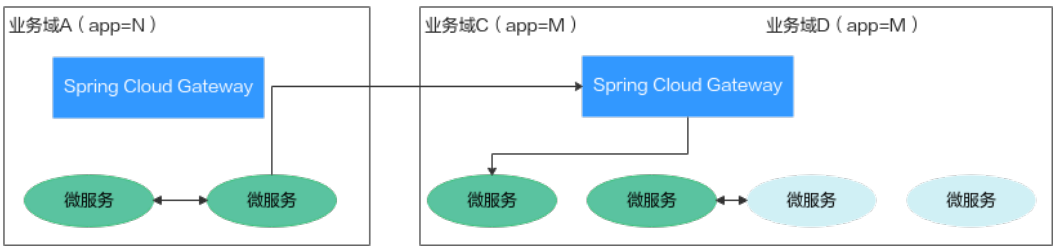
1.1.6 应用逻辑隔离管理

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。

1.1.7 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。

1.1.8 合理规划服务治理

1.1.8.1 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1-1**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 1-1 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。

1.1.9 常见问题

1.1.9.1 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();
}
```

1.1.9.2 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。

1.1.9.3 Spring Cloud 常见启动错误

1.1.9.3.1 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。

1.1.9.3.2 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**。

1.2 托管 Java Chassis 应用

1.2.1 概述

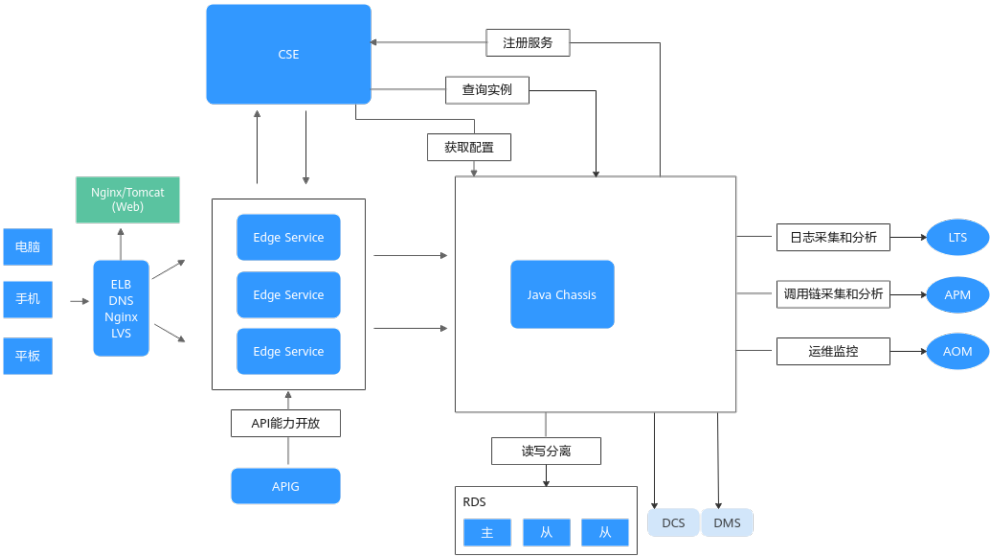
Java Chassis是Apache基金会管理的开源微服务开发框架，最早由CSE捐献，目前有上百个开发者为项目做出贡献。相对于Spring Cloud，Java Chassis，它提供了如下独特的功能：

- 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。

1.2.2 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。

1.2.3 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
```

1.2.4 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。

1.2.5 合理规划服务治理

1.2.5.1 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。

1.2.5.2 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。
```

1.2.6 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。