# 天翼云 • 微服务云应用平台

# 开发指南

中国电信股份有限公司云计算分公司

# 目 录

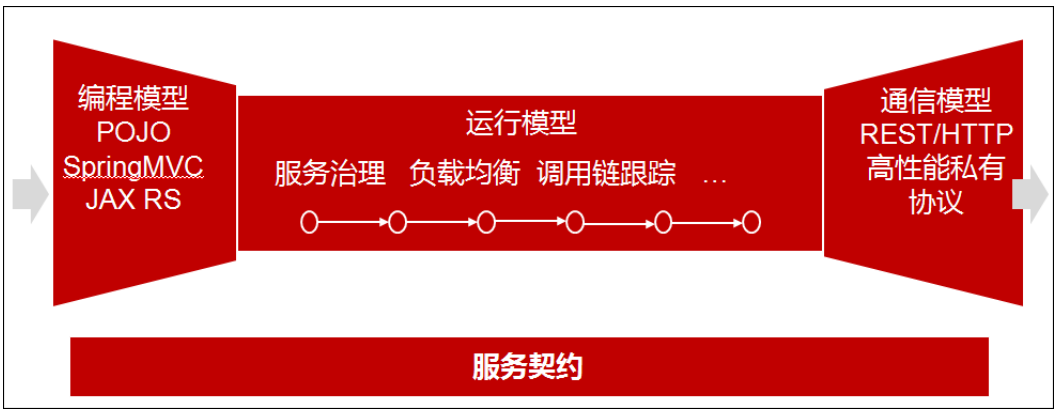
**1 微服务框架概述.....1**
1.1 微服务框架概述.....1
1.2 微服务开发框架特点.....1
1.3 SDK 组件组成.....1
**2 微服务定义.....4**
2.1 微服务元数据定义.....4
2.1.1 微服务元数据配置.....4
2.1.2 实例元数据配置.....5
2.2 微服务契约定义.....6
**3 开发配置.....8**
3.1 依赖包配置.....8
3.2 服务契约配置.....9
3.2.1 静态注册契约.....9
3.2.2 动态注册契约.....10
3.3 日志文件配置.....11
3.4 SDK 配置.....11
**4 应用开发.....13**
4.1 JAX RS 开发.....13
4.1.1 服务端开发.....13
4.1.2 消费端开发.....15
4.2 Spring MVC 开发.....15
4.2.1 服务端开发.....15
4.2.2 消费端开发.....16
4.3 透明 RPC 开发.....17
4.3.1 服务端开发.....17
4.3.2 消费端开发.....19
4.4 CodeFirst 开发方式.....20
4.4.1 服务端开发.....20
4.4.2 消费端开发.....22
**5 部署运行.....23**
5.1 standalone 模式.....23
5.2 web 容器模式.....23
**6 通信通道.....26**
6.1 Rest.....27

6.2 highway ..... 28
6.3 TLS 通信 ..... 29
**7 处理链开发 ..... 34**
7.1 处理链开发概述 ..... 34
7.2 服务治理 ..... 36
7.2.1 概念介绍 ..... 36
7.2.2 配置说明 ..... 37
7.3 负载均衡 ..... 40
7.3.1 场景介绍 ..... 40
7.3.2 配置说明 ..... 40
7.3.3 实施指导 ..... 42
7.4 调用性能统计 ..... 43
7.5 调用链跟踪 ..... 43
7.6 QPS 流控 ..... 45
7.7 TCC 事务 ..... 46
**8 灰度发布 ..... 49**
**9 服务中心 ..... 50**
**10 配置中心 ..... 52**
**11 预置环境变量 ..... 54**
**12 附录 ..... 55**
12.1 特殊开发场景支持 ..... 55
12.2 JAX-RS 注解支持说明 ..... 57
12.3 Spring MVC 注解支持说明 ..... 57
12.4 HttpServletRequest 参数使用说明 ..... 58
12.5 自定义路由策略 ..... 59

# 1 微服务框架概述

## 1.1 微服务框架概述

微服务开发框架包括编程模型、运行模型、通信模型和服务契约。编程模型支持透明 RPC（POJO）、Spring MVC 和 JAX-RS 三种编程模型。运行模型包括服务发现、熔断、负载均衡、配置及跟踪。通信模型支持序列化传输，包括 Rest 和 Highway 两个通道。通过服务契约的定义实现编程模型、运行模型、通信模型的灵活组合。微服务开发框架示意图：




## 1.2 微服务开发框架特点

1. 编程模型和通信模型分离，不同的编程模型可以灵活组合不同的通信模型。应用开发者在开发阶段只关注接口开发，部署阶段灵活切换通信方式；支持 legacy 系统的切换，legacy 系统只需要修改服务发布的配置文件（或者 annotation），而不需要修改代码。
2. 通过定义契约实现模块分离，实现跨语言的通信，并支持配套的软件工具链（契约生成代码、代码生成契约等）开发，构建完整的开发生态。

## 1.3 SDK 组件组成

1. 微服务开发框架组件表

| 类型 | Artifact ID | 是否可选 | 功能说明 |
|---|---|---|---|
| 编程模型 | cse-provider-pojo | 是 | 提供 RPC 开发模式。 |
| 编程模型 | cse-provider-jaxrs | 是 | 提供 JAX RS 开发模式。 |
| 编程模型 | cse-provider-springmvc | 是 | 提供 Spring MVC 开发模式。 |

| 类型 | Artifact ID | 是否可选 | 功能说明 |
| --- | --- | --- | --- |
| 通信模型 | cse-transport-rest-vertx | 是 | 运行于 HTTP 之上的开发框架，不依赖 WEB 容器，应用打包为可执行 jar。 |
| 通信模型 | cse-transport-rest-servlet | 是 | 运行于 WEB 容器的开发框架，应用打包为 war 包。 |
| 通信模型 | cse-transport-highway | 是 | 提供高性能的私有通信协议，仅用于 java 之间互通。 |
| 运行模型 | cse-handler-loadbalance | 是 | 负载均衡模块。提供各种路由策略和配置方法。一般客户端使用。 |
| 运行模型 | cse-handler-bizkeeper | 是 | 服务治理相关的功能，比如隔离、熔断、容错。 |
| 运行模型 | cse-handler-tracing | 是 | 调用链跟踪模块，对接监控系统，吐出打点数据。 |
| 运行模型 | cse-handler-tcc | 是 | 提供 TCC 事务开发管理能力。 |
| 运行模型 | cse-handler-flowcontrol-qps | 是 | QPS 限流。 |
| 运行模型 | cse-handler-performance-stats | 是 | 调用信息统计。 |

说明：编程模型、通信模型必须包含一种，运行模型全部是可选的。如果客户端没有配置负载均衡策略，将使用最简单的轮询策略进行路由。

2. 发布组件说明表

| 发布包名称 | 子项目-1 | 子项目-2 | 功能说明 |
| --- | --- | --- | --- |
| cse-service-center-standalone.tar.gz | - | - | 用于开发测试阶段的服务中心，可以在 windows/linux 系统单机运行。 |
| CSEMS-*.tgz | cse-paas.*.tar.gz | - | 包括服务中心、配置中心以及 ETCD 的 K8S 安装部署包。 |
| CSEMS-*.tgz | - | cse-config- | 配置中心 |

| 发布包名称 | 子项目-1 | 子项目-2 | 功能说明 |
| --- | --- | --- | --- |
| | | center.tar | |
| CSEMS-*.tgz | - | cse-service-center.tar | 服务中心 |
| CSEMS-*.tgz | - | cse-etcd.tar | ETCD |
| CSEMS-*.tgz | - | TOSCA-CSE.zip | 服务设计包。 |
| CSEMS-*.tgz | cse-sdk.*.tar.gz | - | SDK 发布的 jar 包，不包括依赖的三方件。依赖的三方件可以使用公司统一的 maven 仓库获取。 |
| CSEMS-*.tgz | cse.*.tar.gz | - | 包括服务中心、配置中心以及 ETCD 的 VM 安装部署包。 |
| CSEMS-*.tgz | - | cse-config-center.tar.gz | 配置中心 |
| CSEMS-*.tgz | - | cse-service-center.tar.gz | 服务中心 |
| CSEMS-*.tgz | - | etcd*tar.gz | ETCD |

# 2 微服务定义

微服务名支持数字、大小写字母和"-"、"_"、"."三个特殊字符，但是不能以特殊字符作为首尾字符，命名规范为：^[a-zA-Z0-9]+$|^[a-zA-Z0-9][a-zA-Z0-9_\-.]*[a-zA-Z0-9]$。

## 2.1 微服务元数据定义

开发者可以通过多种形式指定微服务信息，微服务的详细内容可以参考《微服务 API 参考》。

### 2.1.1 微服务元数据配置

1. 通过 microservice.yaml 文件配置；

```
APPLICATION_ID: helloTest
service_description:
  name: helloServer
  version: 0.0.1
  properties:
    a: b
    c: d
```

2. 通过实现接口 PropertyExtended 配置 properties，通过实现 com.huawei.paas.cse.serviceregistry.api.PropertyExtended 接口，业务可创建自己的 Properties 扩展类。在 services.yaml 里添加 propertyExtendedClass 配置项，写明业务自己实现的扩展类的路径，如下所示；

```
APPLICATION_ID: helloTest
service_description:
propertyExtendedClass: com.huawei.paas.cse.demo.pojo.server.MircroServicePropertyImpl
```

说明：当接口返回的 key 值和配置文件的 key 值存在重复的时候，会覆盖配置文件的值。初次注册服务时，会将服务的元数据一起注册到服务中心，之后如果要修改服务元数据，需要连同服务 version 一起变更。如果想在服务 version 不变的情况下更新元数据，需要通过服务管理中心统一变更。默认情况下，微服务只支持同一个 app 内部的服务调用。可在微服务的 properties 中配置 allowCrossApp=true 属性，开启可被跨 app 访问权限。

3. 微服务常用属性定义。

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| 服务信息 | service_descript | com.huawei.paas. | 服务名称 |

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| | ion.name | cse.MyService | |
| 服务信息 | service_description.version | 0.0.1 | 版本号 |
| 服务信息 | service_description.description | a description of myservice | 描述 |
| 服务信息 | service_description.role | FRONT | 微服务角色。描述微服务的层级信息，为 front |
| 服务信息 | service_description.role | MIDDLE | 微服务角色。描述微服务的层级信息，为 middle |
| 服务信息 | service_description.role | BACK | 微服务角色。描述微服务的层级信息，为 back |
| 服务信息 | service_description.properties | key/value 对 | 微服务 properties，可为服务添加非通用的属性 |

## 2.1.2 实例元数据配置

1. 静态配置

实例的元数据也是通过 2.1.1 微服务元数据配置中的两种方式进行配置，需要将 service_description 改为 instance_description，如下所示。

```
instance_description:
properties:
    a: b
    c: d
    propertyExtendedClass: com.huawei.paas.cse.demo.pojo.server.InstancePropertyImpl
```

2. 动态配置

实例元数据当前不支持通过配置动态修改，只支持通过直接调用接口与服务中心交互进行修改。

- JAVA 接口

可在业务代码中通过调用封装好的 Java 接口对实例 properties 进行修改，使用如下代码进行调用。

```
Map<String, String> properties = new HashMap<>();
properties.put("key0", "value0");
String serviceId = RegistryUtils.getMicroservice().getServiceId();
String instanceId = RegistryUtils.getMicroserviceInstance().getInstanceId();
ServiceRegistryClient client = RegistryClientFactory.getRegistryClient();
boolean success = client.updateInstanceProperties(serviceId, instanceId, properties );
```

- REST API

可以直接调用服务中心的 REST API 进行实例 properties 的修改。 在调用 api 时每次访问均需要携带 header：{"X-Tenant-Name": "default"}。

1. 首先使用 GET 操作访问 http://127.0.0.1:9980/registry/v3/microservices ，查询 serviceId。
2. 然后使用 GET 操作访问 http://127.0.0.1:9980/registry/v3/microservices/{serviceId}/instances ，另外需携带 header：{"X-ConsumerId": {serviceId}}，查询 instanceId。
3. 最后再使用 PUT 操作访问 http://127.0.0.1:9980/registry/v3/microservices/{serviceId}/instances/{instanceId}/properties ，body 参数格式如下：{"properties": {"tag1": "value1", "tag0": "value0"}}。

## 2.2 微服务契约定义

在进行应用开发之前，需要为微服务定义契约，用于服务端和消费端的解耦。服务端围绕契约进行服务的实现，消费端根据契约进行服务的调用，支持服务端和消费端采用不同的编程语言实现。

使用 yaml 文件格式定义服务契约，推荐使用 Swagger Editor 工具来编写契约，可检查语法格式及自动生成 API 文档。简单定义的 HelloWorld 服务的契约如下示例。

根据下面契约描述，可通过/helloworld/sayhello?name=xxx 调用 HelloWrold 接口的 sayHello 方法。

详细契约文件格式请参考 OpenAPI 官方文档：https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md。

```
swagger: '2.0'
info:
  title: rest test
  version: 1.0.0
  x-java-interface: com.huawei.paas.cse.HelloWorld      #对应schema interface
```

```
basePath: /helloworld
produces:
  - application/json


paths:
  /sayhello:
    get:
      operationId: sayHello   #对应interface的方法名
      parameters:
        - name: name
          in: query
          required: true
          type: string
      responses:
        200:
          description: say hello
          schema:
            type: string
```

说明：根据 swagger 标准，basePath 配置的路径需要包括 web server 的 webroot。

# 3 开发配置

无论是服务端还是消费端的实际业务开发之前都需要完成如下几项基本配置工作。

- 依赖包引入
- 服务契约配置
- 日志配置
- SDK 配置

## 3.1 依赖包配置

本节介绍开发过程依赖包的配置，请根据实际开发模式选择。

使用 dependencyManagement 来进行依赖包版本的管理，通过修改 version 字段来使用相应版本的依赖包 。

依赖包中包括三类：transport、provider、handler，分别用于引入网络通道、开发模式、治理链依赖包。其中 transport 和 provider 为必选，客户端需要增加一个 loadbalance handler 必选项。在 Pom 文件中增加如下内容：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.huawei.paas.cse</groupId>
            <artifactId>cse-dependency</artifactId>
            <version>1.0.21</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- transport -->
    <dependency>
        <groupId>com.huawei.paas.cse</groupId>
        <artifactId>cse-transport-rest-vertx</artifactId>
    </dependency>
```

```
<!-- provider -->
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-provider-pojo</artifactId>
</dependency>

<!-- handler -->
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-handler-performance-stats</artifactId>
</dependency>
</dependencies>
```

说明：详细的依赖包说明请参见 SDK 组件说明。

# 3.2 服务契约配置

微服务框架提供了三种契约配置方式。

- 静态注册：将契约文件置于指定的文件夹下，框架会自行加载契约。
- 动态注册：在运行阶段动态注册契约。
- 从服务中心拉取（尚未实现）：调用端在进行服务调用时如果没有契约，会自动从服务中心拉取。

## 3.2.1 静态注册契约

背景说明

maven 工程中"src/main/resources"是自动创建的，并且是添加到 Builder path 的，如果该目录被删除了，则需要手动创建，并手动添加到 Builder path 中。

前提条件

maven 工程中"src/main/resources"目录已经创建，并且添加到 Builder path。

操作步骤

1. 在"src/main/resources"下创建 microservices 文件夹或者 applications 文件夹，用于存放本地微服务和将要调用的微服务契约。
2. 将 2.2 微服务契约定义中定下的契约放置在"resources/microservices"或者“resources/applications”目录下，目录结构如下所示。

```
- resources
    - microservices
        - serviceName    # 微服务名
```

```
    - schemaId.yaml     # schema接口的契约


    - applications
        - appId       # appId
            - serviceName     # 微服务名
                - schemaId.yaml     # schema接口的契约
```

说明：microservices 目录下面的每一个文件夹表示一个微服务，每个微服务文件夹下的每个 yaml 文件表示一个 schema 的契约，文件名对应 schema-id 的值。 applications 目录下面可用于存放需要指明 appId 的服务契约，比如跨 app 调用场景。

契约中 info 下面的 x-java-interface 字段需要标明具体的 interface 路径，根据项目实际情况而定。

### 3.2.2 动态注册契约

如果在开发阶段不能确定 appId 和微服务名，那么就不能采用静态注册契约的方式了。框架提供了在运行阶段动态注册契约的方式，服务端和调用端使用方式如下。

服务端

将契约文件置于 src/main/resources 目录下的任意文件夹下，如 schemas 文件夹。 然后创建 BootListener 接口的实现类，在该类的 onBootEvent 方法中监听 EventType.BEFORE_PRODUCER_PROVIDER 事件，并注册契约，保证在进行服务注册时能够拥有契约并上传到服务中心，代码如下。

```
import com.huawei.paas.cse.core.BootListener;
import com.huawei.paas.cse.core.definition.loader.DynamicSchemaLoader;


@Component
public class SchemaRegisterListener implements BootListener {
    private static final Logger LOGGER = LoggerFactory.getLogger(SchemaRegisterListener.class);


    @Override
    public void onBootEvent(BootEvent event) {
        if (event.getEventType() == EventType.BEFORE_PRODUCER_PROVIDER) {
            try {
                // 动态注册schemas目录下面的契约到当前服务
                DynamicSchemaLoader.INSTANCE.registerSchemas("classpath*:schemas/*.yaml");
            } catch (Exception e) {
                LOGGER.error("register schema failed, exception: {}", e);
            }
        }
    }
```

```
}
```

另外需要将该类初始化为 Spring bean，可通过在类上面打上@Component 的方式实现，并将该类置于配置的 base-package 路径下，也可直接通过 spring xml 文件的方式初始化为 bean。框架在运行阶段会自动调用该类的 onBootEvent 方法注册契约。

调用端

调用端需要在进行服务调用之前的任意时刻注册契约，代码如下。

```
// 动态注册契约到appService服务中
DynamicSchemaLoader.INSTANCE.registerSchemas("appService", "classpath*:schemas/*.yaml");
```

说明：如果要调用的服务是需要跨 app 的，那么在注册契约时需要指定全名“appId:appService”进行注册。

# 3.3 日志文件配置

本节介绍日志文件的配置。

在 resources 目录下面创建 config 目录，添加 log4j.hello.properties 配置文件，日志配置文件命名规则为 log4j.*.properties，内容如下。

```
paas.logs.dir=../logs/  # 日志文件目录
paas.logs.file=hello.log     # 日志文件名


log4j.rootLogger=INFO,paas,stdout    # 日志级别和appender
```

# 3.4 SDK 配置

本节介绍 SDK 的配置。

操作步骤

1. 在 src/main/resources 目录下面创建 microservice.yaml 文件，用于存放 SDK 配置信息。
2. 服务端配置服务中心地址，网络通道地址，以及处理链信息，内容如下。

```
APPLICATION_ID: helloTest
service_description:
  name: helloServer
  version: 0.0.1
cse:
  service:
    registry:
      address: http://127.0.0.1:9980
  rest:
    address: 0.0.0.0:8080
  handler:
```

```
    chain:
      Provider:
        default: perf-stats
```

3. 消费端配置服务中心地址，网络通道地址，以及处理链信息，内容如下。

```
APPLICATION_ID: helloTest
service_description:
  name: helloClient
  version: 0.0.1
cse:
  service:
    registry:
      address: http://127.0.0.1:9980
  handler:
    chain:
      Consumer:
        default: loadbalance,perf-stats
  references:
    helloServer:
      version-rule: 0.0.1
      transport: rest
```

说明：references 用于为服务调用指定特定的 version 和通道，下面可以配置多个微服务，每个微服务都有自己的 version-rule、transport，未配置的微服务，version-rule 默认为 latest，transport 默认使用所有可用的 transport。这里的 version-rule 支持模糊匹配，例如 0.1+。

# 4 应用开发

微服务框架提供了三种业务开发模式供开发选择，包括透明 RPC，Spring MVC，JAX-RS。

服务端：

透明 RPC 开发：简单的基于接口和接口实现的开发模式。

JAX-RS 开发：使用 Jax-rs 标注进行服务开发。

Spring MVC 开发：使用 Spring mvc 标注进行服务开发。

消费端：

透明 RPC 开发：基于接口进行服务调用。

Spring MVC 开发：使用 Spring MVC 的 RestTemplate 客户端进行调用。

服务端模式与消费端模式没有强绑定关系，可以随意组合，例如：服务端使用透明 RPC 模式开发，消费端使用 Spring MVC 模式进行服务调用。

## 4.1 JAX RS 开发

本节从微服务的服务端和消费端说明如何使用 JAX-RS 开发模式进行业务开发。

### 4.1.1 服务端开发

本节主要介绍 JAX RS 的服务端的开发。

1. 服务接口

根据开发之前定义好的契约来编写 Java 业务接口，示例代码如下。

```
public interface HelloWorld {
    String sayHello(String name);
}
```

说明：该接口需要与契约中指定的 x-java-interface 路径保持一致。

2. 服务实现

使用 JAX-RS 标注进行业务代码的开发，HelloWorld 服务的实现如下。

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
```

```
@Path("/helloworld")
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorldImpl implements HelloWorld {

    @Path("/sayhello")
    @GET
    public String sayHello(@QueryParam("name") String name) {
        return "hello " + name;
    }

}
```

3. 服务发布

首先通过在服务实现类上面打上@RestSchema 标注，写明 schemaId，即表示该实现作为当前微服务的一个 schema 进行发布，如下所示。

```
import com.huawei.paas.cse.provider.rest.common.RestSchema;

......
@RestSchema(schemaId = "helloworld")
public class HelloWorldImpl implements HelloWorld {

    ......

}
```

然后在 resources/META-INF/spring 目录下创建 hello.bean.xml 文件，命名规则为*.bean.xml，配置 spring 进行服务扫描的 base-package，如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:util="http://www.springframework.org/schema/util"
xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
classpath:org/springframework/beans/factory/xml/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
```

```
    <context:component-scan base-package="com.huawei.paas.cse.helloworld" />


</beans>
```

说明：如果不按照上述命名规则及目录存放，则需要显示注明这些 spring 配置文件的加载路径。

### 4.1.2 消费端开发

## 4.2 Spring MVC 开发

本节从微服务的服务端和消费端说明如何使用 Spring MVC 开发模式进行业务开发。

### 4.2.1 服务端开发

本节主要介绍 Spring MVC 的服务端开发。

1. 服务接口

根据开发之前定义好的契约来编写 Java 业务接口，示例代码如下。

```
public interface HelloWorld {
    String sayHello(String name);
}
```

说明：该接口需要与契约中指定的 x-java-interface 路径保持一致，可通过工具根据契约自动化生成。

2. 服务实现

使用 Spring MVC 标注进行业务代码的开发，HelloWorld 服务的实现如下。

```
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;


@RequestMapping(path = "/helloworld", produces = MediaType.APPLICATION_JSON_VALUE)
public class HelloWorldImpl implements HelloWorld {


    @RequestMapping(path = "/sayhello", method = RequestMethod.GET)
    public String sayHello(@RequestParam("name") String name) {
        return "hello " + name;
    }


}
```

3. 服务发布

首先通过在服务实现类上面打上@RestSchema 标注，写明 schemaId，即表示该实现作为当前微服务的一个 schema 进行发布，代码如下所示：

```
import com.huawei.paas.cse.provider.rest.common.RestSchema;


......
@RestSchema(schemaId = "helloworld")
public class HelloWorldImpl implements HelloWorld {


    ......


}
```

然后在 resources/META-INF/spring 目录下创建 hello.bean.xml 文件，命名规则为*.bean.xml，配置 spring 进行服务扫描的 base-package，如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:util="http://www.springframework.org/schema/util"
xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
classpath:org/springframework/beans/factory/xml/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">


    <context:component-scan base-package="com.huawei.paas.cse.helloworld" />


</beans>
```

说明：如果不按照上述命名规则及目录存放，则需要显示注明这些 spring 配置文件的加载路径。

## 4.2.2 消费端开发

本节主要介绍 Spring MVC 的服务消费端开发。

1. 调用声明

Spring MVC 模式服务调用不需进行显式调用声明，直接使用 RestTemplate 通过自定义的 URL 进行调用。 不过使用的 RestTemplate 实例需要由 RestTemplateBuilder.create()进行创建。

2. 服务调用

进程在加载完日志配置和 sdk 配置之后，就可以对服务进行远程调用了。 先通过 RestTemplateBuilder 创建扩展的 Spring RestTemplate 实例对象，再使用该对象通过自定义的 URL 进行服务调用，如下所示。

```
import org.springframework.web.client.RestTemplate;
import com.huawei.paas.cse.provider.springmvc.reference.RestTemplateBuilder;
import com.huawei.paas.foundation.common.utils.BeanUtils;
import com.huawei.paas.foundation.common.utils.Log4jUtils;

public class MainClient {

    public void invoke() throws Exception {
        RestTemplate invoker = RestTemplateBuilder.create();
        String result = invoker.getForObject("cse://helloServer/helloworld/sayhello?name={name}",
                String.class,
                "world");
        System.out.println(result);
    }

}
```

说明：URL 格式为：cse://microserviceName/path?querystring。

# 4.3 透明 RPC 开发

本节分别从微服务的服务端和消费端说明如何使用透明 RPC 开发模式进行业务开发。

## 4.3.1 服务端开发

本节主要介绍透明 RPC 的服务端开发。

1. 服务接口

根据开发之前定义好的契约来编写 Java 业务接口，示例代码如下。

```
public interface HelloWorld {
    String sayHello(String name);
}
```

说明：该接口需要与契约中指定的 x-java-interface 路径保持一致，可通过工具根据契约自动化生成。

2. 服务实现

创建接口实现类 HelloWorldImpl.java，根据需求编写业务代码，内容如下：

```
public class HelloWorldImpl implements HelloWorld {
```

```
    public String sayHello(String name) {
        return "hello " + name;
    }
}
```

3. 服务发布

透明 RPC 开发模式支持通过 spring xml 声明和注解两种服务发布方式。

- spring xml

在 resources 目录下创建"META-INF/spring/hello.bean.xml"文件，内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:util="http://www.springframework.org/schema/util"
xmlns:cse="http://www.huawei.com/schema/paas/cse/rpc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
classpath:org/springframework/beans/factory/xml/spring-beans-3.0.xsd
        http://www.huawei.com/schema/paas/cse/rpc classpath:META-INF/spring/spring-paas-cse-rpc.xsd">

    <cse:rpc-schema schema-id="helloworld"
        implementation="com.huawei.paas.cse.helloworld.server.HelloWorldImpl"></cse:rpc-schema>
</beans>
```

说明：每个服务接口一个 schema 声明。

- 注解

```
import com.huawei.paas.cse.provider.pojo.RpcSchema;

@RpcSchema(schemaId = "helloworld")
public class HelloWorldImpl implements HelloWorld {
...
}
```

在 resources 目录下创建 META-INF/spring/hello.bean.xml，内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
```

```
classpath:org/springframework/beans/factory/xml/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.huawei.paas.cse.helloworld.server" />

</beans>
```

说明：这里的 base-package 指定的包应该包含之前的业务实现类路径。

## 4.3.2 消费端开发

本节主要介绍透明 RPC 的服务消费端开发。

1. 调用声明

消费端在进行服务调用之前需要对服务的调用进行声明。当前提供了 spring xml 和注解两种方式。

- spring xml

spring xml 使用 spring 配置进行调用声明，在文件 META-INF/spring/hello.bean.xml 中声明如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:util="http://www.springframework.org/schema/util"
xmlns:cse="http://www.huawei.com/schema/paas/cse/rpc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
classpath:org/springframework/beans/factory/xml/spring-beans-3.0.xsd
        http://www.huawei.com/schema/paas/cse/rpc classpath:META-INF/spring/spring-paas-cse-rpc.xsd">

    <cse:rpc-reference id="helloworld" schema-id="helloworld"
        microservice-name="helloServer"></cse:rpc-reference>

</beans>
```

然后在消费端代码中使用 HelloWorld hello = BeanUtils.getBean("helloworld")进行服务引用对象的获取。

说明：不要每次调用都去 getBean，spring 的 getBean 内部有一个必然加锁的调用，会影响并发度，getBean 得到的实例可以全局只存一份。

- RpcReference 注解

通过在消费对象的属性上打上@RpcReference 标注来进行服务调用的声明，该属性会自动被实例化，如下：

```
import com.huawei.paas.cse.provider.pojo.RpcReference;


@Component
public class Consumer {
    @RpcReference(microserviceName = "helloServer")
    private  HelloWorld hello;


    ...
}
```

说明：消费对象上面必须打上@Component 标注，Spring 会自动化将其实例化为 bean。

2. 服务调用

调用端在加载完日志配置、sdk 配置之后，就可以对服务进行远程调用了。 实际调用和本地通过接口调用完全一样，如下所示：

```
public class MainClient {
        ...


        # 服务调用
        System.out.println(hello.sayHello("world-rpc"));
    }
}
```

# 4.4 CodeFirst 开发方式

本节从微服务的服务端和消费端说明 CodeFirst 开发模式进行业务开发。

## 4.4.1 服务端开发

直接写实现类，不必预先定义契约，不必预先定义接口。启动时，会根据实现类自动生成契约，并注册到服务中心。支持下面几种规则。

- 透明 RPC 开发模式

```
@RpcSchema(schemaId = "codeFirst")
public class CodeFirstPojo {
    public int reduce(int a, int b) {
        return a - b;
    }


    public Person sayHello(Person user) {
```

```
        user.setName("hello " + user.getName());
        return user;
    }

    public String saySomething(String prefix, Person user) {
        return prefix + " " + user.getName();
    }
......
```

说明：因为纯粹从代码上，完全无法分辨微服务开发人员期望如何定义契约，所以生成的契约全是 POST 方法，所有 method 的入参被包装为一个 class，作为 body 参数传递。

- Jaxrs 开发模式

```
@RestSchema(schemaId = "codeFirst")
@Path("/codeFirstJaxrs")
@Produces(MediaType.APPLICATION_JSON)
public class CodeFirstJaxrs {
    @Path("/add")
    @POST
    public int add(@FormParam("a") int a, @FormParam("b") int b) {
        return a + b;
    }

    @Path("/sayhello")
    @POST
    public Person sayHello(Person user) {
        user.setName("hello " + user.getName());
        return user;
    }

    @Path("/saysomething")
    @POST
    public String saySomething(@HeaderParam("prefix") String prefix, Person user) {
        return prefix + " " + user.getName();
    }
......
```

- Springmvc 开发模式

```
@RestSchema(schemaId = "codeFirst")
@RequestMapping(path = "/codeFirstSpringmvc", produces = MediaType.APPLICATION_JSON_VALUE)
public class CodeFirstSpringmvc {
```

```
    @RequestMapping(path = "/sayhello", method = RequestMethod.POST)
    public Person sayHello(@RequestBody Person user) {
        user.setName("hello " + user.getName());
        return user;
    }

    @RequestMapping(path = "/saysomething", method = RequestMethod.POST)
    public String saySomething(@RequestHeader(name = "prefix") String prefix, @RequestBody Person user)
{
        return prefix + " " + user.getName();
    }
```

### 4.4.2 消费端开发

从服务中心下载或是通过线下手段，得到 Producer 的契约，通过契约生成相应的数据类型及接口，然后正常开发 consumer 逻辑。

# 5 部署运行

微服务框架当前提供了两种部署运行模式：standalone 模式和 web 容器模式。推荐使用 standalone 模式拉起服务进程。

## 5.1 standalone 模式

微服务框架提供了 standalone 部署运行模式，可直接在本地通过 Main 函数拉起。

编写 Main 函数，初始化日志和加载服务配置，内容如下。

```
import com.huawei.paas.foundation.common.utils.BeanUtils;
import com.huawei.paas.foundation.common.utils.Log4jUtils;

public class MainServer {

    public static void main(String[] args) throws Exception {
        Log4jUtils.init();            # 日志初始化
        BeanUtils.init();            # Spring bean初始化

        ...
    }
}
```

说明：如果使用的是 rest 网络通道，需要将 pom 中的 transport 改为使用 cse-transport-rest-vertx 包。运行 MainServer 即可启动该微服务进程，向外暴露服务。

## 5.2 web 容器模式

如果需要将该微服务加载到 web 容器中启动运行时，需要新建一个 servlet 工程包装一下。

1. web.xml 文件配置

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath*:META-INF/spring/*.bean.xml
            classpath*:app-config.xml
        </param-value>
    </context-param>
```

```
    <listener>
        <listener-
class>com.huawei.paas.cse.transport.rest.servlet.RestServletContextListener</listener-class>
    </listener>

    <servlet>
        <servlet-name>RestServlet</servlet-name>
        <servlet-class>com.huawei.paas.cse.transport.rest.servlet.RestServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    <servlet-mapping>
        <servlet-name>RestServlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

RestServletContextListener 用于初始化微服务环境，包括日志、spring 等。

需要指定 spring 配置文件加载路径，通过 context-param 来配置。 上例中，classpath*:META-INF/spring/*.bean.xml 是微服务的内置规则，如果配置中未包含这个路径，则 RestServletContextListener 内部会将它追加在最后。

另外可以配置 servlet-mapping 中的 url-pattern 规则将满足指定规则的 URL 路由到该 servlet 中来。

2. pom.xml 文件配置

当前只有 Rest 网络通道支持此种运行模式，使用 web 容器模式时需要将 pom 文件中的 transport 改为依赖 cse-transport-rest-servlet 包。

设置 finalName，是方便部署，有这一项后，maven 打出来的 war 包，部署到 web 容器中，webroot 即是这个 finalName。

```
<dependencies>
    <dependency>
        <groupId>com.huawei.paas.cse</groupId>
        <artifactId>cse-transport-rest-servlet</artifactId>
    </dependency>

    ...
</dependencies>
```

```
<build>
    <finalName>test</finalName>
</build>
```

说明：

- RESTful 调用应该与 web 容器中其他静态资源调用（比如 html、js 等）隔离开来，所以 webroot 后一段应该还有一层关键字，比如上面 web.xml 中举的例子（/test/rest）中的 rest。
- 以 tomcat 为例，默认每个 war 包都有不同的 webroot，这个 webroot 需要是 basePath 的前缀，比如 webroot 为 test，则该微服务所有的契约都必须以/test 打头。
- 当微服务加载在 web 容器中，并直接使用 web 容器开的 http、https 端口时，因为是使用的 web 容器的通信通道，所以需要满足 web 容器的规则。

# 6 通信通道

微服务框架实现了两种网络通道，包括 Rest 和 Highway，均支持 TLS 加密传输。

通信通道的公共配置：

| 配置项 | 配置值参考 | 说明 |
| --- | --- | --- |
| cse.request.timeout | 30000 | 请求超时时间，rest/highway 的客户端都共用这个配置项。 |
| cse.references.[服务名].transport | rest | 当一个服务实例存在多种不同类型的 transport 的时候，可以指定客户端只访问某一个 transport。 |
| cse.references.[服务名].version-rule | latest | 当多个版本实例共存的时候，指定只访问哪些版本。支持 latest，1.0.0+，1.0.0-2.0.2，精确版本。详细参考服务中心的接口描述。 |

服务监听与发布：

服务监听的地址和上报给注册中心的地址是分开的。

服务监听地址由 transport 层 cse.<transport_name>.address 决定，服务上报给注册中心的地址由 cse.service.publishAddress 决定。若用户未配置 cse.service.publishAddress，此时若服务监听地址为 0.0.0.0，默认随机选择一个网卡上报给注册中心，若监听地址非 0.0.0.0，默认上报服务监听地址。如 0 所示。服务监听 0.0.0.0 时可能带来的安全风险请用户自行分析。

服务监听端口同监听地址一样，监听和注册的值可以分开设置。实际监听的值，在 cse.<transport_name>.address 中指定，同样，注册到 SC 中的值可以通过 cse.<transport_name>.publishPort 指定。

| cse.<transport_name>.address | 服务监听地址 | cse.service.publishAddress | 服务发布地址 |
| --- | --- | --- | --- |
| 10.93.100.154:8080 | 10.93.100.154:8080 | - | 10.93.100.154:8080 |
| 0.0.0.0:8080 | 0.0.0.0:8080 | - | 10.93.100.154:8080/127.0.0.0:8080/其它网卡:8080 |

| cse.<transport_name>.address | 服务监听地址 | cse.service.publishAddress | 服务发布地址 |
|---|---|---|---|
| 10.93.100.154:8080 | 10.93.100.154:8080 | 10.93.100.153 | 10.93.100.153:8080 |
| 0.0.0.0:8080 | 0.0.0.0:8080 | 10.93.100.154 | 10.93.100.154:8080 |
| ......:8080 | ......:8080 | {ethX} | IP of ethX:8080 |

服务发布地址规则：

- 可以以指定地址的方式发布

将 cse.service.publishAddress 环境变量的值设置为一个固定的 IP 地址，以将其注册到注册中心中。例如设置 cse.service.publishAddress=10.93.100.154

- 指定获取某个网卡的地址作为发布地址

将 cse.service.publishAddress 环境变量的值设置为一个网卡的名字，微服务在启动的时候可以自动将这个网卡的地址注册到注册中心。例如设置 cse.service.publishAddress={eth0}，则最终 eth0 网卡的地址会被注册。

## 6.1 Rest

Rest 网络通道将服务以标准 RESTful 的形式向外发布，调用端兼容直接使用 http client 使用标准 RESTful 形式进行调用。

Rest 网络通道提供了两种模式：vertx 和 servlet，分别对应两种部署运行模式，用户可根据自己的运行模式来选择 Rest 网络通道。

1. Rest on Vertx

Rest on Vertx 网络通道对应的依赖包如下。

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-transport-rest-vertx</artifactId>
</dependency>
```

使用 Rest on Vertx 通道需要如下配置项：

| 分类 | 配置项 | 配置值参考 | 说明 |
|---|---|---|---|
| REST + Vertx | cse.rest.address | 0.0.0.0:8080 | 服务监听的地址 |
| REST + Vertx | cse.rest.server.thread-count | 1 | 服务端线程个数 |
| REST + Vertx | cse.rest.client.thread-count | 1 | 客户端网络线程个数 |
| REST + Vertx | cse.rest.client. | 1 | 客户端每个网络线 |

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| | connection-pool-per-thread | | 程中的连接池个数 |

服务供给端进程需要配置 cse.rest.address，向外暴露服务。服务消费端进程如果仅仅作为消费端则可不配置 cse.rest.address 项。

2. Rest on Servlet

Rest on Servlet 网络通道对应的依赖包如下：

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-transport-rest-servlet</artifactId>
</dependency>
```

使用 Rest on Servlet 通道需要如下配置项：

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| REST + Servlet | cse.rest.address | 0.0.0.0:8080 | transport 监听地址 |
| REST + Servlet | cse.rest.timeout | 3000（缺省值） | 超时时间（ms） |

3. 序列化

- 参数

当前 Rest 通道的 body 参数只支持 application/json 序列化方式。 如果需要向服务端发送 form 类型参数，那么需要在调用端构造好 application/json 格式的 body 数据，如{"a": 3, "b": 4}，不能直接通过 multipart/form-data 格式传递 form 类型参数，服务端无法解析。

- 返回值

当前 Rest 通道的返回值支持 application/json 和 text/plain 两种序列化方式。服务供给端通过 produces 声明可提供的序列化能力，调用端通过请求的 Accept 头指明返回值序列化方式，默认返回 application/json 格式数据。

## 6.2 highway

Highway 通道是高性能私有协议通道，在有特殊性能需求场景时可选用。

1. Highway 通道对应的依赖包如下。

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-transport-highway</artifactId>
</dependency>
```

2. Highway 通道需要的配置项如下。

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| highway + tcp | cse.highway.address | 0.0.0.0:7070 | 服务监听的地址 |
| highway + tcp | cse.highway.server.thread-count | 1（缺省值） | 服务端网络线程个数 |
| highway + tcp | cse.highway.client.thread-count | 1（缺省值） | 客户端网络线程个数 |
| highway + tcp | cse.highway.client.connection-pool-per-thread | 1（缺省值） | 客户端每个网络线程的连接池个数 |

## 6.3 TLS 通信

通过简单的配置，开发者可以启用 TLS 通信，保障数据的传输安全。

1. SDK 和外部服务的交互如下所示。

| 交互方 | 主体方 | tag | 启用实例 | 说明 |
| --- | --- | --- | --- | --- |
| 服务中心 | 微服务 SDK | sc.consumer | cse.service.registry.address=https://host:port | 微服务支持通过 TLS 访问服务中心 |
| 配置中心 | 微服务 SDK | cc.consumer | cse.config.client.serverUri=https://host:port | 微服务支持通过 TLS 访问配置中心 |
| 微服务 highway consumer | 微服务 highway provider | highway.provider | cse.highway.address=0.0.0.0:7070?sslEnabled=true | hiway provider |
| 微服务 highway provider | 微服务 highway consumer | highway.consumer | NA | hiway consumer |
| 微服务 rest consumer | 微服务 rest provider | rest.provider | cse.rest.address=0.0.0.0:8080?sslEnabled=true | rest provider |
| 微服务 rest | 微服务 rest | rest.consumer | NA | rest consumer |

| 交互方 | 主体方 | tag | 启用实例 | 说明 |
|---|---|---|---|---|
| provider | consumer | | | |

2. SDK 的证书配置

目前支持给微服务的各个交互方统一指定证书，也可以单独指定证书，加上 tag 的配置会覆盖默认配置。

ssl.[tag].[property]，示例如下。

```
ssl.ciphers : TLS_RSA_WITH_AES_128_GCM_SHA256(微服务公共配置)
ssl.rest.consumer.ciphers: TLS_RSA_WITH_AES_256_GCM_SHA384(SDK开发的REST消费者)
ssl.rest.provider.ciphers:TLS_RSA_WITH_AES_256_GCM_SHA384(SDK开发的REST提供者)
ssl.highway.consumer.ciphers: TLS_RSA_WITH_AES_256_GCM_SHA384(SDK开发的Highway消费者)
ssl.highway.provider.ciphers:TLS_RSA_WITH_AES_256_GCM_SHA384(SDK开发的Highway提供者)
```

SDK 的证书配置项说明如下所示。

| 配置项名称 | 配置项说明 | 示例和缺省值 |
|---|---|---|
| ssl.protocols | 协议列表，逗号分隔 | TLSv1.2 |
| ssl.ciphers | 算法列表，逗号分隔 | TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_RSA_WITH_AES_128_GCM_SHA256 |
| ssl.authPeer | 是否认证对端 | TRUE |
| ssl.checkCN.host | 是否对证书的 CN 进行检查（该配置项只对 Consumer 端，并且使用 http 协议有效，即 Consumser 端使用 rest 通道有效。对于 Provider 端、highway 通道等无效。检查 CN 的目的是防止服务器被钓鱼，参考标准定义：https://tools.ietf.org/ | TRUE |

| 配置项名称 | 配置项说明 | 示例和缺省值 |
| --- | --- | --- |
| | html/rfc2818。） | |
| ssl.trustStore | 信任证书文件 | trust.jks |
| ssl.trustStoreType | 信任证书类型 | JKS |
| ssl.trustStoreValue | 信任证书密码 | - |
| ssl.keyStore | 身份证书文件 | server.p12 |
| ssl.keyStoreType | 身份证书类型 | PKCS12 |
| ssl.keyStoreValue | 身份证书密码 | - |
| ssl.crl | 吊销证书文件 | revoke.crl |
| ssl.sslCustomClass | SSLCustom 类的实现，用于开发者转换密码、文件路径等。 | 实现了 com.huawei.paas.foundation.ssl.SSLCustom 的类名。 |

说明：默认的协议算法是高强度加密算法，JDK 需要安装对应的策略文件，参考：http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html。 您可以在配置文件配置使用非高强度算法。微服务消费者，可以针对不同的提供者指定证书（当前证书是按照 HOST 签发的，不同的提供者都使用一份证书存储介质，这份介质同时给微服务访问服务中心和配置中心使用）。

3. 微服务上公有云配置

微服务上公有云主要解决服务中心、配置中心和 IAM 认证服务的认证关系，系统已经提供了默认实现，只需要在 microservices.yaml 中增加如下几个配置项。

```
ssl.cc.consumer.sslOptionFactory: com.huawei.paas.foundation.auth.SSLOptionFactoryCloud
ssl.sc.consumer.sslOptionFactory: com.huawei.paas.foundation.auth.SSLOptionFactoryCloud
ssl.apiserver.consumer.sslOptionFactory: com.huawei.paas.foundation.auth.SSLOptionFactoryCloud
```

4. 服务管理中心的证书配置

目前支持使用环境变量来配置的 TLS 认证方式，默认开启 TLS 通信，双向认证模式，认证对端时同时校验对端是否匹配证书（CommonName）字段。

服务管理中心的证书配置项说明如下所示。

| 配置项名称 | 配置项说明 | 示例和缺省值 |
| --- | --- | --- |
| CSE_SSL_MODE | 设置协议模式，取值 1 或 0（HTTPS/HTTP）。 | 1 |
| CSE_SSL_VERIFY_CLIENT | 设置 HTTPS 模式下是否认 | 1 |

| 配置项名称 | 配置项说明 | 示例和缺省值 |
| --- | --- | --- |
| | 证对端，取值 1 或 0（认证/不认证）。 | |
| CSE_SSL_PASSPHASE | 设置 HTTPS 模式下的证书密钥访问密码。 | - |

服务管理中心配置文件$APP_ROOT/conf/app.conf 配置说明如下所示，暂不支持环境变量方式设置。

| 配置项名称 | 配置项说明 | 示例和缺省值 |
| --- | --- | --- |
| ssl_protocols | 通信使用的 SSL 版本 | TLSv1.2 |
| ssl_ciphers | 配置使用算法列表 | TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,<br>TLS_RSA_WITH_AES_256_GCM_SHA384,<br>TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,<br>TLS_RSA_WITH_AES_128_GCM_SHA256,<br>TLS_RSA_WITH_AES_128_CBC_SHA。 |

说明：由于服务中心支持 HTTP/2 协议，所以 ssl_ciphers 必须配置有 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 算法。TLS_RSA_WITH_AES_128_GCM_SHA256、TLS_RSA_WITH_AES_128_CBC_SHA 被列为 HTTP/2 协议的不安全算法黑名单，但为了客户端算法兼容性，存在时必须配置到最后一位。

密钥物料及证书存放路径说明如下：

| 目录 | 说明 | 对应环境变量 |
| --- | --- | --- |
| / | - | - |
| /opt | - | - |
| /opt/CSE | - | INSTALL_ROOT |
| /opt/CSE/etc | - | - |
| /opt/CSE/etc/cipher | 密钥物料存放目录。 | CIPHER_ROOT |
| /opt/CSE/etc/ssl | 证书存放目录。 | SSL_ROOT |
| /opt/CSE/etc/ssl/trust.cer | 授信 CA。 | - |
| /opt/CSE/etc/ssl/server_key.pem | 已加密服务端私钥文件。 | - |

| 目录 | 说明 | 对应环境变量 |
| --- | --- | --- |
| /opt/CSE/etc/ssl/server.cer | 服务器证书。 | - |
| /opt/CSE/etc/ssl/cert_pwd | 用于存放解密私钥的对称加密密文文件。 | - |
| /opt/CSE/apps | - | - |
| /opt/CSE/apps/ServiceCenter | - | APP_ROOT |
| /opt/CSE/apps/ServiceCenter/conf | 服务管理中心配置文件目录。 | - |
| /opt/CSE/apps/ServiceCenter/conf/app.conf | 应用配置文件。 | - |

# 7 处理链开发

## 7.1 处理链开发概述

- 平台默认提供的处理链：

| 名称 | 模块 | 适用范围 | 说明 |
| --- | --- | --- | --- |
| loadbalance | cse-handler-loadbalance | Consumer | 提供客户端路由发现、负载均衡。 |
| bizkeeper-consumer | cse-handler-bizkeeper | Consumer | 客户端服务治理能力。 |
| bizkeeper-provider | cse-handler-bizkeeper | Provider | 服务端服务治理能力。 |
| tracing-consumer | cse-handler-tracing | Consumer | 客户端调用链数据生成（span 数据，和监控服务对接）。一般客户端服务端都启用才有效。 |
| tracing-provider | cse-handler-tracing | Provider | 服务端调用链数据生成（span 数据，和监控服务对接）。一般客户端服务端都启用才有效。 |
| sla-consumer | cse-handler-tracing | Consumer | 客户端调用链数据生成（SLA 数据，和监控服务对接）。一般客户端服务端都启用才有效。 |
| sla-provider | cse-handler-tracing | Provider | 服务端调用链数据生成（SLA 数据，和监控服务对接）。一般 |

| 名称 | 模块 | 适用范围 | 说明 |
| --- | --- | --- | --- |
| | | | 客户端服务端都启用才有效。 |
| tcc-client | cse-handler-tcc | Consumer | 进行 TCC 事务管理。 |
| tcc-server | cse-handler-tcc | Provider | 进行 TCC 事务管理。 |
| qps-flowcontroll-consumer | cse-handler-flowcontrol-qps | Consumer | 客户端做 qps 限流。 |
| qps-flowcontroll-provider | cse-handler-flowcontrol-qps | Provider | 服务端做 qps 限流。 |
| perf-stats | cse-handler-performance-stats | Consumer/Provider | 调用信息统计。 |

- 使用处理链

可以通过如下配置控制 Provider 和 Consumer 使用的处理链。

- Provider: cse.handler.chain.Provider.default= bizkeeper-provider,tracing-provider,sla-provider
- Consumer: cse.handler.chain.Consumer.default=loadbalance,bizkeeper-consumer,tracing-consumer,sla-consumer

给不同的微服务指定不一样的处理链。

- cse.handler.chain.Consumer.service.[MicroServiceName]=loadbalance,bizkeeper-consumer,tracing-consumer,sla-consumer

- 处理链的顺序

处理链的顺序不同，系统工作行为也不同。 下面列举一下常见问题。

- loadbalance 和 bizkeeper-consumer 这两个顺序可以随机组合。但是行为是不一样的。 当 loadbalance 在前面的时候，那么 loadbalance 提供的重试功能会在 bizkeeper-consumer 抛出异常时发生，比如超时等。如果已经做了 fallbackpolicy 配置，比如 returnnull，那么 loadbalance 则不会重试。 如果 loadbalance 在后面，那么 loadbalance 的重试会延长超时时间，即使重试成功，如果 bizkeeper-consumer 设置的超时时间不够，那么最终的调用结果也是失败。
- tracing-consumer,sla-consumer,tracing-provider,sla-provider 这些调用链建议放到调用链的最开始位置，保证成功、失败的情况都可以记录日志（由于记录日志需要 IP 等信息，对于消费者，只能放到 loadbalance 后面）。如果不需要记录客户端返回的异常，则可以放到末尾，只关注网络层返回的错误。但是如果 bizkeeper-consumer 等超时提前返回的话，则可能不会记录日志。

- 建议的顺序

  Consumer: loadbalance,tracing-consumer,sla-consumer,bizkeeper-consumer

  Provider: tracing-provider,sla-provider,bizkeeper-provider

  这种顺序能够满足大多数场景，并且不容易出现不可理解的错误。

- 处理链扩展

可以通过实现 AbstractHandler 来扩展处理链。 具体包含如下几个步骤：

1. 实现 AbstractHandler

```
public class MyHandler extends AbstractHandler {
    @Override
    public void handle(Invocation invocation, AsyncResponse asyncResp) throws Exception {
        // code before

        invocation.next(response -> {
            // code after
            asyncResp.handle(response);
        });
    }
}
```

2. 给处理链增加一个名字 增加 config/cse.handler.xml 文件，内容如下。

```
<config>
<handler id="myhandler" class="xxx.xxx.xxx.MyHandler" />
</config>
```

3. 在 microservices.ymal 文件指定需要使用该处理链。

```
cse:
handler:
chain:
  Provider:
    default: bizkeeper-provider,tracing-provider,sla-provider,perf-stats,myhandler
```

处理链的顺序和上面定义的顺序有关，注意加到一个合适的位置。

# 7.2 服务治理

## 7.2.1 概念介绍

服务治理主要涉及“隔离”、“熔断”、“容错”、“负载均衡”等技术措施，这些技术措施共同完成了服务治理的能力。为了让开发者能够正确配置服务治理的参数，微服务提供了一种技术措施上的定义。

- “隔离”是一种异常检测机制，常用的检测方法是请求超时、流量过大等。一般的设置参数包括超时时间、同时并发请求个数等。
- “熔断”是一种异常反应机制，“熔断”依赖于“隔离”。熔断通常基于错误率来实现。一般的设置参数包括统计请求的个数、错误率等。
- “容错”是一种异常处理机制，“容错”依赖于“熔断”。熔断以后，会调用“容错”的方法。一般的设置参数包括调用容错方法的次数等。

把这些概念联系起来（配置参考）： 当"隔离"措施检测到 N 次请求中共有 M 次错误的时候，"熔断"不再发送后续请求，调用"容错"处理函数。

这个技术上的定义，是和 Netflix Hystrix 一致的，通过这个定义，非常容易理解它提供的配置项，参考：https://github.com/Netflix/Hystrix/wiki/Configuration。

说明：SDK 开发指南里面描述的都是技术概念，最终用户并不会用到这里的概念。系统提供的治理功能请参考治理中心的文档说明。

## 7.2.2 配置说明

1. 配置项生效范围

- 按照类型(type)：配置项能够针对 Provider, Consumer 进行配置。
- 按照范围(scope)：配置项能够针对 MicroService 进行配置, 也可以针对【x-schema-id + operationId】进行配置。

```
cse.[namespace].[type].[MicroServiceName].[接口名称].[property name]
```

说明：

- type 指 Provider 或者 Consumser。
- 对于全局配置，不需要指定 MicroServiceName。针对特定的微服务的配置，需要增加 MicroServiceName。
- 对于适用于接口配置的，需要指定接口名称，接口名称由【x-schema-id + operationId】组成。

隔离示例如下。

```
cse.isolation.Consumer.enabled
cse.isolation.Consumer.DemoService.enabled
cse.isolation.Consumer.DemoService.hello.sayHello.enabled
cse.isolation.Provider.enabled
cse.isolation.Provider.DemoService.enabled
cse.isolation.Provider.DemoService.hello.sayHello.enabled
```

2. 配置项列表

| 类 | 配置项 | 默认配置值参考 | 说明 |
| --- | --- | --- | --- |

| 类 | 配置项 | 默认配置值参考 | 说明 |
| --- | --- | --- | --- |
| 隔离 | cse.isolation.timeout.enabled | FALSE | 是否启用超时检测。 |
| 隔离 | cse.isolation.timeoutInMilliseconds | 30000 | 超时时间，超过时间，记录一次错误。 |
| 隔离 | cse.isolation.maxConcurrentRequests | 10 | 通过并发数检测错误。配置最大并发数。 |
| 熔断 | cse.circuitBreaker.enabled | TRUE | 是否启用熔断措施。 |
| 熔断 | cse.circuitBreaker.forceOpen | FALSE | 不管失败次数，都进行熔断。 |
| 熔断 | cse.circuitBreaker.forceClosed | FALSE | 任何时候都不熔断，forceOpen 优先。 |
| 熔断 | cse.circuitBreaker.sleepWindowInMilliseconds | 15000 | 熔断后，多长时间恢复。恢复后，会重新计算失败情况。注意：如果恢复后的调用立即失败，那么会立即重新进入熔断。 |
| 熔断 | cse.circuitBreaker.requestVolumeThreshold | 20 | 10s 内统计的请求个数，10s 内统计的请求必须大于这个值，并且错误率达到阈值的时候才熔断。由于 10 秒还会被划分为 10 个 1 秒的统计周期，经过 1s 中后 |

| 类 | 配置项 | 默认配置值参考 | 说明 |
| --- | --- | --- | --- |
| | | | 才会开始计算错误率，因此从调用开始至少经过 1s，才会发生熔断。 |
| 熔断 | cse.circuitBreaker.errorThresholdPercentage | 50 | 错误率，达到错误率的时候熔断。 |
| 容错 | cse.fallback.enabled | TRUE | 是否启用出错后的故障处理措施。支持返回 null 和抛出 Exception 两种处理措施。 |
| 容错 | cse.fallback.maxConcurrentRequests | 10 | 并发调用容错处理措施（cse.fallbackpolicy.policy）的请求数。超过这个值，不再调用处理措施，直接返回异常。 |
| 容错 | cse.fallbackpolicy.policy | throwexception | 出错后的处理策略，默认抛出异常。可选值有：returnnull，throwexception |

说明：在表格里，全部省略 type 和 MicroServiceName。未特殊说明，配置项都支持 Provider 和 Consumer。例如：对于服务消费者，需要配置为：cse.isolation.Consumer.enabled，对于服务提供者，需要配置为：cse.isolation.Provider.enabled。

谨慎使用 cse.isolation.timeout.enabled=true。因为系统处理链都是异步执行，中间处理链的返回，会导致后面处理链的逻辑处理效果丢失。尽可能将 cse.isolation.timeout.enabled 保持默认值 false，并且正确设置网络层超时时间 cse.request.timeout=30000。

# 7.3 负载均衡

## 7.3.1 场景介绍

当应用访问一个集群部署的服务时，会涉及到路由负载均衡。 当前 CSE2.1 提供的基于 Ribbon 的方案，可以通过配置文件配置负载均衡策略，当前支持随机，顺序，基于响应时间的权值等多种负载均衡路由策略。这些策略可以根据用户的场景进行修改配置。

## 7.3.2 配置说明

1. 命名规范如下：

cse.loadbalance.[MicroServiceName].[property name]

- 只能配置客户端。
- 对于全局配置，不需要指定 MicroServiceName。针对特定的微服务的配置，需要增加 MicroServiceName。

  负载均衡示例如下：.

```
cse.loadbalance.NFLoadBalancerRuleClassName
cse.loadbalance.DemoService.NFLoadBalancerRuleClassName
```

2. 配置项

负载均衡配置项如下：

| 配置项 | 配置值参考 | 说明 |
| --- | --- | --- |
| cse.loadbalance.NFLoadBalancerRuleClassName | com.netflix.loadbalancer.RoundRobinRule | 基于轮询的路由选择策略。 |
| cse.loadbalance.NFLoadBalancerRuleClassName | com.netflix.loadbalancer.RandomRule | 基于随机的路由选择策略。 |
| cse.loadbalance.NFLoadBalancerRuleClassName | com.netflix.loadbalancer.WeightedResponseTimeRule | 基于服务器响应时间的路由选择策略。 |
| cse.loadbalance.NFLoadBalancerRuleClassName | com.huawei.paas.cse.loadbalance.SessionStickinessRule | 会话保持路由选择策略。 |
| cse.loadbalance.SessionStickinessRule.sessionTimeoutInSeconds | 30 | 客户端闲置时间，超过限制后选择后面的服务器。（说明：暂不支持微服务配置。例如： |

| 配置项 | 配置值参考 | 说明 |
| --- | --- | --- |
| | | cse.loadbalance.SessionStickinessRule.sessionTimeoutInSeconds，不能配置为cse.loadbalance.DemoService.SessionStickinessRule.sessionTimeoutInSeconds）。 |
| cse.loadbalance.SessionStickinessRule.successiveFailedTimes | 5 | 客户端失败次数，超过后会切换服务器（暂不支持微服务配置）。 |
| cse.loadbalance.retryEnabled | FALSE | 负载均衡捕获到服务调用异常，是否进行重试。 |
| cse.loadbalance.retryOnNext | 0 | 尝试新的服务器的次数。 |
| cse.loadbalance.retryOnSame | 0 | 同一个服务器尝试的次数。 |
| cse.loadbalance.isolation.enabled | FALSE | 是否开启故障实例隔离功能。 |
| cse.loadbalance.isolation.enableRequestThreshold | 20 | 当实例的调用总次数达到该值时开始进入隔离逻辑门槛，需为整数，默认值为 20。 |
| cse.loadbalance.isolation.errorThresholdPercentage | 20 | 实例故障隔离错误百分比，需为(0,100]区间整数，默认值为 20，大于该值时该实例被隔离。 |
| cse.loadbalance.isolation.singleTestTime | 10000 | 故障实例单点测试时间，单位为 ms，默认值为 10000，当前时间与该实例上次被调用到的时间差大 |

| 配置项 | 配置值参考 | 说明 |
| --- | --- | --- |
| | | 于该值时该实例有机会被调用到。 |
| cse.loadbalance.transactionControl.policy | com.huawei.paas.cse.loadbalance.filter.SimpleTransactionControlFilter | 动态路由分流策略，框架提供了简单的分流机制，开发者也可以实现自定义的分流过滤策略。 |
| cse.loadbalance.transactionControl.options | key/value pairs | 针对 SimpleTransactionControlFilter 分流策略的配置项，可添加任意项过滤标签。 |

## 7.3.3 实施指导

配置文件路径：src\main\resources\microservice.yaml。

实施负载均衡的步骤如下。

1. 在处理链配置项当中增加负载均衡一项，代码如下。

```
cse:
  ......
  handler:
    chain:
      Consumer:
        default: loadbalance
  ......
```

2. 增加路由策略，代码如下。

```
cse:
  ......
  loadbalance:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
  ......
```

说明：

- 路由策略默认配置为 com.netflix.loadbalancer.RoundRobinRule。
- 其中 NFLoadBalancerRuleClassName 为具体的路由策略，通过配置不同的实现类的路径来实现不同的路由策略，通常的路由策略配置如下：

A. 轮询： com.netflix.loadbalancer.RoundRobinRule

B. 随机：com.netflix.loadbalancer.RandomRule

C. 权值：com.netflix.loadbalancer.WeightedResponseTimeRule

D. 会话保持：com.huawei.paas.cse.loadbalance.SessionStickinessRule

自定义路由策略。

如果微服务开发者对路由策略有特殊的需求，可以自定义实现自己的路由策略，只要实现 com.netflix.loadbalancer.IRule 接口，并实现其相应的接口方法即可。具体可以参考 7.3.2 配置说明 会话保持的策略实现：SessionStickinessRule。此外可参考 12.5 自定义路由策略。

# 7.4 调用性能统计

调用性能统计模块用于在运行日志中输出请求的执行时间和系统吞吐量等信息，让开发者能够看到系统的性能状况。

1. 软件包

调用性能统计模块对应的 Pom 软件包如下所示。

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-handler-performance-stats</artifactId>
</dependency>
```

2. 配置项

调用性能统计配置项如下所示。

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| 性能日志 | cse.metrics.enabled | TRUE | 是否输出性能日志统计。 |
| 性能日志 | cse.metrics.cycle.ms | 60000 | 输出性能日志统计的时间间隔。 |

# 7.5 调用链跟踪

调用跟踪模块主要实现与“服务监控”的对接，按照服务监控的要求，产生 span 和 SLA 数据。

1. 软件包

调用跟踪模块对应的 pom 依赖包如下：

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-handler-tracing</artifactId>
</dependency>
```

2. 配置项

调用跟踪配置项如下所示。

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| 调用跟踪 | cse.tracing.enabled | TRUE | 是否启用调用跟踪。只控制 span 数据，不控制 SLA 数据的生成。 |
| 调用跟踪 | cse.tracing.samplingRate | 1.0 | 采样的频率。只控制 span 数据，不控制 SLA 数据的生成。 |

定义日志文件格式参考

开发者可以在 log4j 配置文件中定制监控日志输出的格式，必须按照下面的要求对 SLA 和调用链数据进行覆盖，代码如下。

说明：开发者不需要覆盖所有的配置项，只需要根据需要覆盖必要的配置项。配置项的默认值微服务会根据实际情况进行调整，这样系统可以使用到最新的配置功能。 比如，多数情况开发者只需要覆盖文件路径： log4j.appender.sla.File=/var/log/metrics.dat

```
// sla properties
log4j.logger.com.huawei.paas.cse.sla.fileslasender=ERROR, sla
log4j.additivity.com.huawei.paas.cse.sla.fileslasender=false
log4j.appender.sla=org.apache.log4j.RollingFileAppender
log4j.appender.sla.layout=org.apache.log4j.PatternLayout
log4j.appender.sla.Append=true
log4j.appender.sla.bufferedIO=false
log4j.appender.sla.immediateFlush=true
log4j.appender.sla.layout.ConversionPattern=%m%n
log4j.appender.sla.Encoding=UTF-8
log4j.appender.sla.File=m.dat
// span properties
log4j.logger.com.huawei.paas.cse.tracing.filespansender=ERROR, talc
log4j.additivity.com.huawei.paas.cse.tracing.filespansender=false
log4j.appender.talc=org.apache.log4j.RollingFileAppender
log4j.appender.talc.layout=org.apache.log4j.PatternLayout
log4j.appender.talc.Append=true
log4j.appender.talc.bufferedIO=false
log4j.appender.talc.immediateFlush=true
log4j.appender.talc.layout.ConversionPattern=%m%n
log4j.appender.talc.Encoding=UTF-8
```

```
log4j.appender.talc.File=t.dat
```

开发者可以通过在日志配置文件中配置 log4j.appender.sla.File=m.dat 和 log4j.appender.talc.File=t.dat 两个配置项来修改 SLA 和调用链数据文件的生成路径，日志配置文件命名格式为 log4j.*.properties。

## 7.6 QPS 流控

QPS 流控用于限制每秒钟最大的请求数量。在 consumer 端，用于限制发往某个微服务的请求频率；在 provider 端，用于限制某个微服务过来的请求频率。consumer 端和 provider 端的流控有些不同，请注意区分。provider 端的流控不是安全意义上的流量控制，而是业务层面的功能。为了防止 DOS 攻击，需要结合其他的一些列措施。

说明：QPS 流控不是绝对精确的，可能会有少量误差。

1. QPS 配置项参考

| 配置项 | 默认值 | 说明 |
| --- | --- | --- |
| cse.flowcontrol.Consumer.qps.enabled | TRUE | 是否启用 Consumer 流控。 |
| cse.flowcontrol.Provider.qps.enabled | TRUE | 是否启用 Provider 流控。 |
| cse.flowcontrol.Consumer.qps.limit.[ServiceName].[Schema].[operation] | 2147483647 （max int） | 每秒钟允许的 QPS 数，支持 microservice、schema、operation 三个级别的配置。 |
| cse.flowcontrol.Provider.qps.limit.[ServiceName] | 2147483647 （max int） | 每秒钟允许的 QPS 数，仅支持 microservice 一个级别的配置。 |

2. Consumer 流控

流控是微服务级的，不是进程级的。

假设调用目标微服务：ms，目标 schema：s1，目标 operation：op。

启用流控后，如果不配置 limit，QPS 最高允许为 2147483647。如果配置 cse.flowcontrol.Consumer.qps.limit.ms 为 10000。调用 ms，QPS 最高允许为 10000。

启用流控后，如果配置 cse.flowcontrol.Consumer.qps.limit.ms 为 10000，cse.flowcontrol.Consumer.qps.limit.ms.s1 为 20000。调用 ms.s1，QPS 最高允许为 20000，除 ms.s1 以外的，对 ms 的调用，QPS 最高允许为 10000。

启用流控后，如果配置 cse.flowcontrol.Consumer.qps.limit.ms 为 10000，cse.flowcontrol.Conusmer.qps.limit.ms.s1 为 20000，cse.flowcontrol.Consumer.qps.limit.ms.s1.op 为 30000。调用 ms.s1.op，QPS 最高允许为 30000。除 ms.s1.op 以外的，对 ms.s1 的调用，QPS 最高允许为 20000。除 ms.s1 以外的，对 ms 的调用，QPS 最高允许为 10000。

3. Provider 流控

Provider 流控配置比较简单， cse.flowcontrol.Provider.qps.limit.ms=1000，表示允许来自于微服务 ms 的请求每秒最大为 1000。

# 7.7 TCC 事务

在一个长事务中，一个由两台服务器一起参与的事务，服务器 A 发起事务，服务器 B 参 与事务，B 的事务需要人工参与，所以处理时间可能很长。如果按照 ACID 的原则，要 保持事务的隔离性、一致性，服务器 A 中发起的事务中使用到的事务资源将会被锁定， 不允许其他应用访问到事务过程中的中间结果，直到整个事务被提交或者回滚。这就 造成事务 A 中的资源被长时间锁定，系统的可用性将不可接受。

为了解决在事务运行过程中大颗粒度资源锁定的问题，业界提出一种新的事务模型， 它是基于业务层面的事务定义。锁粒度完全由业务自己控制。它本质是一种补偿的思 想。它把事务运行过程分成 Try、Confirm/Cancel 两个阶段。在每个阶段的逻辑由业务 代码控制。这样就事务的锁粒度可以完全自由控制。业务可以在牺牲强隔离性的情况 下，获取更高的性能。

场景描述


使用指南

1. TCC 事务对应的依赖包如下所示。

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-handler-tcc</artifactId>
</dependency>
```

2. 然后将 tcc-server 添加进处理链中，如下：

```
cse:
  ......
  handler:
    chain:
      Provider:
        default: perf-stats,tcc-server
  ......
```

3. 定义服务实现类，如下所示。

```
@RpcSchema(schemaId = "helloworld")
public class TccHelloworldImpl implements ITccHelloworld {
    private static final Logger LOGGER = LoggerFactory.getLogger(TccHelloServiceImpl.class);

    @Override
    @TccTransaction(confirmMethod = "confirm", cancelMethod = "cancel")
    public String sayHello(String name) {
        LOGGER.info("Try say hello from client, {}", name);
        return "Hello, " + name;
    }

    public String confirm(String name) {
        LOGGER.info("Confirm say hello from client, {}", name);
        return null;
    }

    public String cancel(String name) {
        LOGGER.info("Cancel say hello from client, {}", name);
        return null;
    }
}
```

在支持 TCC 事务的方法上打上@TccTransaction 标注，并注明 confirmMethod 和 cancelMethod 方法。

confirmMethod 和 cancelMethod 的参数和返回值必须和服务提供函数相同。

如果 Try 正常执行，confirm 方法也会被执行。如果 Try 抛出异常，则 cancel 会被执行。

说明：

- 如果不使用标注的方式发布服务，那么需要在实现类上面打上@Component 标注。
- TCC 事务对业务逻辑定义需要有一定要求，TCC 操作应该支持幂等性原则，否则容 易产生过度补偿等问题。
- TCC 支持运行在微服务多实例中，其原理是将事务数据统一存储到数据库中，TCC 组件提供统一的事务存储接口，以便开发对接不同的数据库，只需要继承实现 com.huawei.paas.cse.tcc.repository.CachableTransactionRepository 类即可完成多实例 TCC 支持；也可参考 com.huawei.paas.cse.tcc.repository 包中简单的数据库存储类实现，已简单实现了 Jdbc/redis/ZooKeeper 的对接。

配置项说明如下：

| 配置项 | 分类 | 默认配置值参考 | 说明 |
| --- | --- | --- | --- |
| cse.tcc.transaction.repository | TCC | com.huawei.paas.cse.tcc.repository.FileSystemTransactionRepository | 事务存储仓库。 |
| cse.tcc.transaction.repository.file.path | TCC | tcc | 使用文件系统存储事务时，指定文件存储根路径，默认在当前目录的 tcc 文件夹下。 |
| cse.tcc.transaction.recover | TCC | TRUE | 是否启动恢复机制。 |

# 8 灰度发布

功能描述

该功能对应于微服务目录灰度发布功能。管理员可以通过下发规则，对服务进行灰度发布管理。本章节描述如何在微服务中启用这个功能。

前提条件

微服务成功对接配置中心。

配置参考

1. 增加依赖关系(pom.xml)

```
<dependency>
    <groupId>com.huawei.paas.cse</groupId>
    <artifactId>cse-handler-cloud-extension</artifactId>
</dependency>
```

引入必要的 jar 包。

2. 在 Consumer 端配置负载均衡(microservice.yaml)

```
cse:
  loadbalance:
    serverListFilters: darklaunch
    serverListFilter:
      darklaunch:
        className: com.huawei.paas.darklaunch.DarklaunchServerListFilter
```

在负载均衡模块启用了灰度发布的 filter。

注意事项

由于客户端默认只会下载最新版本的微服务实例，在启用灰度发布的时候，灰度规则默认只应用于最新版本，灰度发布的效果可能和实际的预期效果不同。如果一个微服务需要支持灰度发布，建议增加如下配置项：

cse.references.[服务名].version-rule=1.0.0+

其中服务名是依赖的服务的名称。这样配置后，客户端可以收取到大于 1.0.0 所有版本的服务端实例，然后针对这些实例组做灰度发布。

目前灰度发布的自定义策略，支持的数据类型只包含 String 和 Integer。 使用灰度发布涉及业务提前规划，在做规划的时候，如果基于请求参数，请使用 String 或者 Integer 两种类型的参数。

# 9 服务中心

本节主要介绍服务中心所需配置项。服务中心客户端配置说明如下

| item | 2.1 | description |
| --- | --- | --- |
| 服务中心地址 | cse.service.registry.address=http(s)://{ip:port} | 连接服务注册中心地址，支持配置多个地址，地址之间以逗号隔开：http://10.21.253.148:9980,http://10.21.221.112:9980。 |
| HTTP 协议版本 | cse.service.registry.client.httpVersion=HTTP_1_1 | 连接使用的 HTTP 协议版本。 |
| 线程数 | cse.service.registry.client.workerPoolSize=1 | 工作线程数。 |
| 连接超时时间 | cse.service.registry.client.timeout.connection=30000 | 请求连接超时值(ms)。 |
| 处理超时时间 | cse.service.registry.client.timeout.request=30000 | 请求处理超时值(ms)。 |
| 动态发现注册中心 | cse.service.registry.autodiscovery=false | 是否启用动态发现注册中心机制 |
| 定义租户 ID | cse.config.client.tenantName=default | 定义租户 ID |
| 自注册 | cse.service.registry.instance.preferIpAddress | 是否以 IP 地址方式注册实例的 HostName。 |
| 健康检查 | cse.service.registry.instance.healthCheck.interval | 检查时延(s)，上报心跳时间间隔。 |
| 健康检查 | cse.service.registry.instance.healthCheck.times | 重试次数，服务管理中心更新心跳失败重试次数。 |
| 实例变化推送 | cse.service.registry.instance.watch | 是否 Watch 实例变化。 |

| item | 2.1 | description |
| --- | --- | --- |
| 服务发布 IP | cse.service.publishAddress | 服务向注册中心发布的 IP 地址。 |

# 10 配置中心

本节主要介绍配置中心所需配置项。

SDK 中配置中心配置项

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| 配置中心 | cse.config.client.serverUri | https://10.22.87.59:30103 | 配置中心的地址，支持多个，用逗号分隔。若 cse.service.registry.autodiscovery 配置为 true，则通过该配置项地址发现 config-center 集群，若配置为 false，则直接使用该配置项地址为 config-center 集群地址。 |
| 服务信息 | cse.config.client.refreshPort | 30104 | 配置中心动态下发配置端口。 |
| 服务信息 | cse.config.client.refreshMode | 0 | 配置动态刷新模式，0 为 configcenter 在发生变化时主动推送，1 为 client 端周期拉取，其他值均为非法，不会去连配置中心。 |
| 服务信息 | cse.config.client.refresh_interv | 10000 | refreshMode 配置为 1 时，client 端 |

| 分类 | 配置项 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| | al | | 主动从配置中心拉取配置的周期，单位毫秒。 |
| 服务信息 | cse.config.client.tenantName | default | 租户名。 |
| 服务信息 | service_description.name | testclient | 应用名。 |
| 服务信息 | cse.service.registry.autodiscovery | FALSE | 是否通过配置的地址自动发现集群。 |

配置中心环境变量配置

| 配置项 | 分类 | 配置值参考 | 说明 |
| --- | --- | --- | --- |
| ENCRYPT_KEY | 配置中心 | Changeme_123 | 配置证书密码。 |
| CSE_CC_SERVER_ADDR | 配置中心 | 本机地址 | 配置监听地址，请指定本机 IP 地址。 |
| CSE_ETCD_SERVICE_ADRR | 配置中心 | 10.10.10.1:2379,<br>10.10.10.2:2379 | 配置 ETCD 节点，表示方法为“ip:port”，支持多个节点，例如：10.10.10.1:2379,<br>10.10.10.2:2379。 |
| CSE_RATE_LIMIT_TIME | 配置中心 | 1000 | 配置限流时间，表示统计该事件长度内的流量，单位为 ms，若为“0”，则不限流。 |
| CSE_RATE_LIMIT_COUNT | 配置中心 | 10 | 配置流量限制参数，连接最大值，表示每段限流时间内最大的连接数，若为“0”，则不限流。 |
| CSE_CC_SSL_MODE | 配置中心 | 1 | 配置协议模式，取值 “1”或“0” ，默认为“0”。其中“0”表示“http”， “1”表示“https”。 |

# 11 预置环境变量

本节主要介绍微服务从环境中获取的变量信息。微服务应用从环境中获取变量列表如下所示。

| 类别 | 名称 | 含义 | 可选值 | 可选 |
| --- | --- | --- | --- | --- |
| 注册中心 | SERVICE_CENTER_ADRESS | 注册中心的地址信息 | 是 IP:Port 的组合， 多个 IP:PORT 之间采用逗号（;）分开。 | required |
| 位置信息 | REGION_ID | 应用所在 region 的 id | 可以是字母和数字、下划线等符号的组合，避免冒号、引号等字符。 | - |
| | AVAILABILITYA_ZONE_ID | 应用所在可用区的 ID | - | - |
| | DATACENTER_ID | 应用所在数据中心的 ID | - | - |
| 应用信息 | APPLICATION_ID | 应用的 id | • 可以是字母和数字、下划线等符号的组合，避免冒号、引号等字符 default 租户信息 tenant_id 租户的 ID。<br>• 可以是字母和数字、下划线等符号的组合，避免冒号、引号等字符。 | - |
| 租户信息 | TENANT_ID | 租户的 ID | • 可以是字母和数字、下划线等符号的组合，避免冒号、引号等字符 default 租户信息 tenant_id 租户的 ID。<br>• 可以是字母和数字、下划线等符号的组合，避免冒号、引号等字符。 | - |

# 12 附录

## 12.1 特殊开发场景支持

- 2xx 状态码支持

框架支持根据场景返回不同的 2xx 状态码。 首先需要在契约里面每个 operation 的 responses 部分定义好特定返回值与返回类型，以便生成文档。

说明：所有 2xx 状态码场景的返回值类型需要保持一致

使用 ContextUtils.getInvocationContext().setStatus(202) 指定返回特定的状态码。 如果是多线程或异步场景，使用如下方式指定：

```
public String sayHello(String name) {
    final InvocationContext context = ContextUtils.getInvocationContext();
    new Thread(new Runnable() {
        public void run() {
            ...
            context.setStatus(202);
        }
    }).start().join();
    return name;
}
```

说明：如果不使用 ContextUtils 设置状态码，那么服务端将返回默认的 200 状态码。

- 异常处理

框架支持根据不同的异常情况返回不同状态码以及异常信息。

首先需要在契约中定义好每个异常返回状态码与返回类型。 然后在业务接口实现中抛出 InvocationException 异常，设置好状态码和异常信息，框架即可根据契约将特定异常返回给调用端。使用方法如下。

```
public String sayHello(String name) {
    throw new InvocationException(code, codeMsg, codeBody);
}
```

- 上下文数据透传

通过 cse-context 在框架中进行上下文数据的透传，不影响业务接口。

1. 服务端

A. 业务代码中使用 cse-context，在业务代码中可通过 ContextUtils 获取到透传过来的上下文数据。

```
Map<String, Object> context = ContextUtils.getInvocationContext().getContext();
System.out.println(context.get("token"));
```

B. handler 中使用 cse-context

```
Map<String, Object> context = invocation.getContext();
System.out.println(context.get("token"));
```

1. 调用端

A. 使用原生 Rest 客户端将所有需要透传到服务端的上下文数据打包成一个 Map<String, Object>，然后序列化成 json 串之后放入到请求的 header 参数中，key 为 x-cse-context。

```
Map<String, Object> context = new HashMap<>();
context.put("token", "token-abcd");      # 需要透传的上下文数据
String jsonContext = new ObjectMapper().writeValueAsString(context);

HttpHeaders headers = new HttpHeaders();
headers.add("x-cse-context", jsonContext);
HttpEntity<Object> requestEntity = new HttpEntity<>(headers);
ResponseEntity<ResponseCls> resEntity = template.exchange(url, httpmethod, requestEntity,
ResponseCls.class);
```

B. 使用微服务客户端，在下个版本中支持。

- Raw Json 字符串参数支持

框架支持由业务自己做序列化，框架中对参数直接进行透传。 首先需要在契约中给参数加上 x-raw-json 属性，只针对 body 类型参数，如下。

```
parameters:
  - name: name
    in: body
    required: true
    x-raw-json: true
    type: string
```

在调用端需要业务自己对参数进行序列化为 json 字符串，然后进行调用，服务端的业务接口中直接收到原生 json 字符串，由业务自己反序列化成对象使用。

- 使用 Java 接口进行服务调用

支持直接通过调用内部 Java 接口进行服务调用，代码如下。

```
import com.huawei.paas.cse.core.consumer.InvokerUtils;

......
```

```
Object result = InvokerUtils.syncInvoke("serviceName", "schemaId", "operationName", new Object[]
{arg0});
......
```

说明：当前跨 app 调用只支持通过这种方式进行调用，不支持通过标准的编程接口模式进行访问。

注意：在进行跨 app 的服务调用时，微服务名需要指定 appId，格式如：appId:serviceName。

## 12.2 JAX-RS 注解支持说明

本节介绍 JAX-RS 开发模式微服务当前版本支持的注解。

JAX-RS 注解支持

| 注解 | 位置 | 描述 |
| --- | --- | --- |
| javax.ws.rs.Path | schema/operation | URL 路径 |
| javax.ws.rs.Produces | schema/operation | 方法支持的编解码能力。 |
| javax.ws.rs.DELETE | operation | http method |
| javax.ws.rs.GET | operation | http method |
| javax.ws.rs.POST | operation | http method |
| javax.ws.rs.PUT | operation | http method |
| javax.ws.rs.QueryParam | parameter | 从 query string 中获取。 |
| javax.ws.rs.PathParam | parameter | 从 path 中获取，必须在 path 中定义好该参数。 |
| javax.ws.rs.HeaderParam | parameter | 从 header 中获取。 |
| javax.ws.rs.CookieParam | parameter | 从 cookie 中获取。 |

说明：

- 当方法参数没有注解，且不为 HttpServletRequest 类型参数时，默认为 body 类型参数，一个方法只支持最多一个 body 类型参数。
- 打在参数上面的注解建议显示定义出 value 值，否则将直接使用契约中的参数名，例如应该使用@QueryParam("name") String name，而不是@QueryParam String name。

## 12.3 Spring MVC 注解支持说明

本节介绍 Spring MVC 开发模式当前支持 org.springframework.web.bind.annotation 包下的注解。

Spring MVC 注解支持

Spring MVC 开发模式当前支持 org.springframework.web.bind.annotation 包下的注解如 0 所示，所有注解的使用方法参考 Spring MVC 官方文档：Spring MVC 官方文档。

| 注解 | 位置 | 描述 |
| --- | --- | --- |

| 注解 | 位置 | 描述 |
| --- | --- | --- |
| RequestMapping | schema/operation | 支持标注 path/method/produces 三种数据，operation 默认继承 schema 上的 produces。 |
| PathVariable | parameter | 从 path 中获取，写好注解的 value 值。 |
| RequestParam | parameter | 从 query string 中获取，写好注解的 value 值。 |
| RequestHeader | parameter | 从 header 中获取，写好注解的 value 值。 |
| RequestBody | parameter | 从 body 中获取，每个 operation 只能有一个 body 参数。 |

说明：当方法参数没有注解，且不为 HttpServletRequest 类型参数时，默认为 RequestParam 类型参数，参数名为契约中对应的参数名。

## 12.4 HttpServletRequest 参数使用说明

本节主要介绍 HttpServletRequest 参数使用说明。

HttpServletRequest 参数

微服务框架支持在 operation 中直接使用 HttpServletRequest 参数，由开发者自己处理 http request 中的相关数据。该参数可用于 JAX-RS 和 Spring MVC 两种开发模式中。

业务代码中通过 HttpServletRequest 使用到 query/path/header/cookie/body 五个类型的参数数据时必须在契约中将参数定义出来，否则无法在业务代码中获取到相关参数。

业务接口实现如下。

```
@RequestMapping(path = "/sayhi", method = RequestMethod.GET)
public String sayHi(HttpServletRequest request) {
    return "hi " + request.getParameter("name");
}
```

对应契约如下。

```
/sayhi:
  get:
    operationId: sayHi
```

```
parameters:
  - name: name
    in: query
    required: true
    type: string
responses:
  200:
    description: say hi
    schema:
      type: string
```

# 12.5 自定义路由策略

场景描述

SDK 提供了多种不同的路由策略，用户开发应用时通过配置即可方便使用。但实际使用中，对于某些特定的应用场景，当以上提供的几种路由策略不能满足用户使用时，用户可以在 SDK 提供的路由策略的框架下根据业务需要通过编程的方式来开发满足需要的路由策略 。

实施指导

1. 实现接口 com.netflix.loadbalancer.IRule 中定义的接口方法。路由选择逻辑在 public Server choose(Object key)方法中实现。 LoadBalancerStats 是一个封装了负载均衡器当前运行状态的一个结构。通过获取 stats 中各个实例的运行指标，在 choose 方法中，判定将当前请求路由到那个实例上进行处理。处理风格可以参照 com.huawei.paas.cse.loadbalance.SessionStickinessRule。
2. 编译开发的策略，保证生成的 class 在 classpath 下。
3. 通过 SDK 配置该路由策略，假如是 AbcRule。则配置如下。

```
cse.loadbalance.NFLoadBalancerRuleClassName=com.huawei.cse.ribbon.rule.AbcRule
```