目录

Spring Cloud概述

KMSE支持原生 Spring Cloud 微服务框架，开发者只需要添加依赖和修改配置即可使用服务注册、调用链、分布式配置等能力。

版本配置关系说明：

Spring Cloud	Spring Boot	最新 KMSE SDK 版本
Greenwich	2.1.x	1.0-SNAPSHOT

兼容性说明：

Spring Cloud功能	开源实现	KMSE兼容性	说明
服务注册与发现	Netflix Eureka、Consul	兼容Consul	提供高可用注册中心，支持本地缓存
负载均衡	Netflix Ribbon	兼容	-
服务调用	RestTemplate、Feign	兼容	-
调用链	Spring Cloud Sleuth、jaeger	兼容jaeger	-
分布式配置	Spring Cloud Config、Consul Config	兼容Consul Config	支持通过控制台管理配置，发布配置和查看配置发布历史
熔断降级	Spring Cloud Hystrix、resilience4j	兼容resilience4j	-
微服务网关	Spring Cloud Gateway、NetflixZuul	兼容Spring Cloud Gateway	-
消息队列	Kafka	兼容	-

服务注册与发现

准备工作

开始实践服务注册发现功能前，请确保已完成了 SDK 下载。 服务注册依赖consul作为注册中心，需要本地安装consul，consul下载和安装参考官网https://www.consul.io。

实现服务注册和服务发现

通过一个简单的示例说明如何实践服务的注册和发现。 注意：需要完成“服务开发”的相关步骤后才进行以下操作。https://docs.ksyun.com/documents/37283

一、**创建服务提供者** 此服务提供一个简单的服务,并将自身注册到服务。

1. 创建server工程 从微服务平台下载一个项目，命名为server-demo。
   应用 > 服务开发

2. 修改pom依赖 修改pom.xml中dependency依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>com.ksyun.kmse</groupId>
    <artifactId>spring-cloud-kmse-starter-consul-discovery</artifactId>
    <version>${version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

3. 修改pom依赖 添加服务提供段代码，启动类添加注解。

```
@SpringBootApplication
@EnableDiscoveryClient
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

4. 提供接口服务 创建一个AccountController,提供一个简单的接口服务。

```
package com.demo.x.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RequestMapping({"/account"})
@RestController
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    public AccountController() {
    }

    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用account " + id);
        return id + "";
    }
}
```

5.修改配置 在resource目录下的bootstrap.yml文件中配置应用名与监听端口号等信息。

```
server:
  port: 8080
spring:
  application:
    name: server-demo
  cloud:
    host: http://127.0.0.1
    port: 8500
    discovery:
      healthCheckPath: /actuator/health
      healthCheckInterval: 15s
      register: true
      service-name: server-demo
      enabled: true
      instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
```

二、创建服务消费者 我们将创建一个服务消费者,消费者通过FeignClient客户端去调用服务提供者。

1. 创建client工程 从微服务平台下载一个项目，命名为client-demo。

2. 修改pom依赖 在pom.xml中引入需要的依赖内容：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>com.ksyun.kmse</groupId>
    <artifactId>spring-cloud-kmse-starter-consul-discovery</artifactId>
    <version>${version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

3. 开启服务注册与发现 启动类添加注解，添加代码调用服务方。

```
@SpringBootApplication
@EnableFeignClients
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

4. 设置调用信息 创建一个FeignClient去调用server-demo服务。

```
package com.demo.x.client;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@FeignClient("server-demo")
public interface OrderClient {

    @GetMapping({"/account/{id}"})
    String getById(@PathVariable Integer id);

}
```

```
package com.demo.x.controller;

import com.demo.x.client.OrderClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping({"/account"})
@RestController
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);
    @Autowired
    private OrderClient orderClient;

    @GetMapping({"/demo"})
    public String demo() {
        return orderClient.getById(12);
    }
}
```

5. 修改配置 在resource目录下的bootstrap.yml文件中配置应用名与监听端口号。

```
server:
  port: 8081
spring:
  application:
    name: client-demo
  cloud:
    host: http://127.0.0.1
    port: 8500
```

三、**调用服务** 分别启动server-demo和client-demo，然后调用client-demo的/demo接口，接口没有报错并成功返回，一个简单的服务注册与发现的例子就做完了。

配置中心

准备工作

开始实践配置中心功能前，请确保已完成了 SDK 下载。 配置中心依赖consul作为配置中心，需要本地安装consul，consul下载和安装参考官网https://www.consul.io。

一、**创建config-demo工程** 从微服务平台下载一个项目，命名为config-demo。

二、**依赖项** 修改pom.xml中dependency依赖如下：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.ksyun.kmse</groupId>
            <artifactId>spring-cloud-kmse-starter-consul-config</artifactId>
            <version>${version}</version>
        </dependency>
```

三、**修改配置** bootstrap.yml文件中添加如下配置。

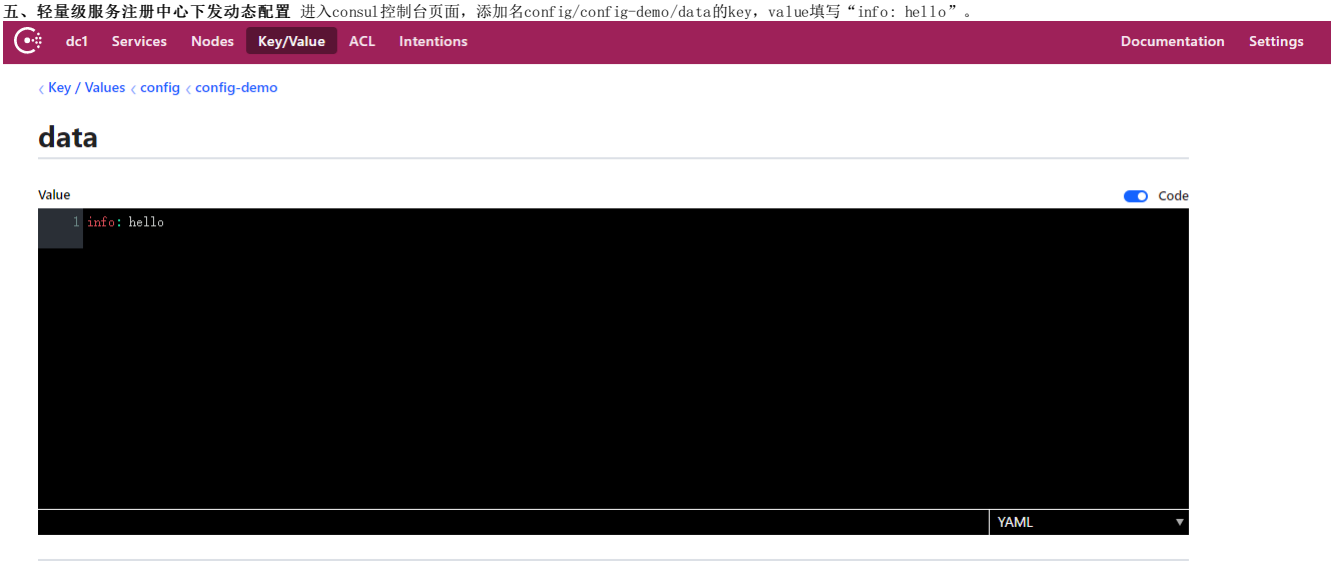
```
spring:
  application:
    name: config-demo
  cloud:
    consul:
      host: http://127.0.0.1
      port: 8500
      config:
        prefix: config
        enabled: true
        format: YAML
        data-key: data
```

四、添加代码 使用@Value和@RefreshScope注解，添加对应的配置项。

```
package com.demo.x.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.cloud.context.config.annotation.RefreshScope;

@RequestMapping({"/account"})
@RestController
@RefreshScope
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    @Value("${info}")
    private String info;
```

```
    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用account " + id);
        return id + ":"+info;
    }
}
```

五、轻量级服务注册中心下发动态配置 进入consul控制台页面，添加名config/config-demo/data的key，value填写“info: hello”。



配置加解密

操作场景
KMSE（微服务引擎）使用jasypt库来对配置进行加解密。

前提条件
向工程中添加依赖，在pom.xml中添加以下依赖：

```
<dependency>
  <groupId>com.github.ulisesbocchio</groupId>
  <artifactId>jasypt-spring-boot-starter</artifactId>
  <version>3.0.3</version>
</dependency>
```

kmse默认使用RSA进行加解密，用户从服务开发中下载工程模板中会随机生成一对RSA公私钥对，存放在resources目录下（public_key.pem/private_key.pem）。
bootstrap.properties默认配置如下：

```
jasypt.encryptor.privateKeyFormat=PEM
jasypt.encryptor.private-key-location= classpath:private_key.pem
```

密文内容被"ENC(xxx)"包裹，其中xxx为密文。
明文内容被"DEC(xxx)"包裹，其中xxx为明文。

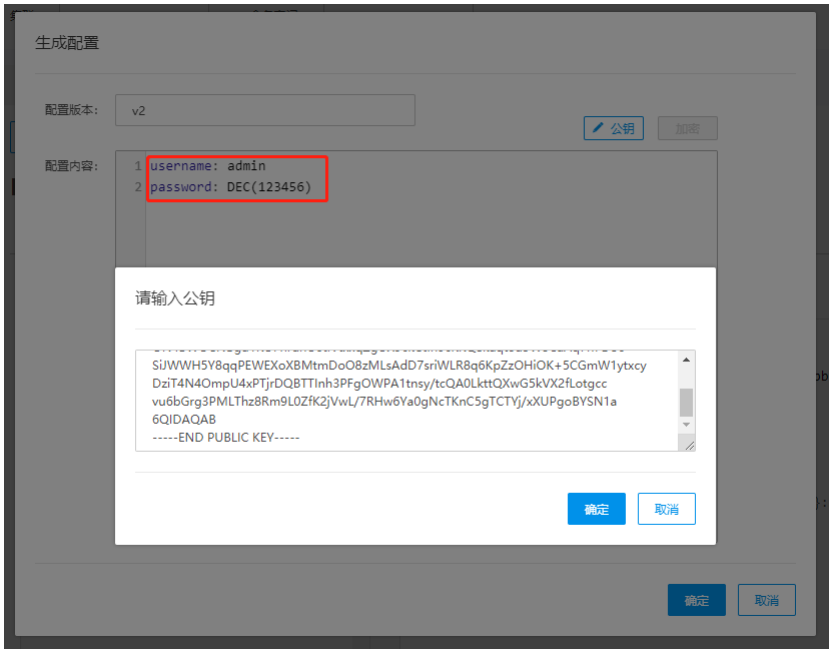
配置加密
例如正常配置为：

```
password: 123456
```

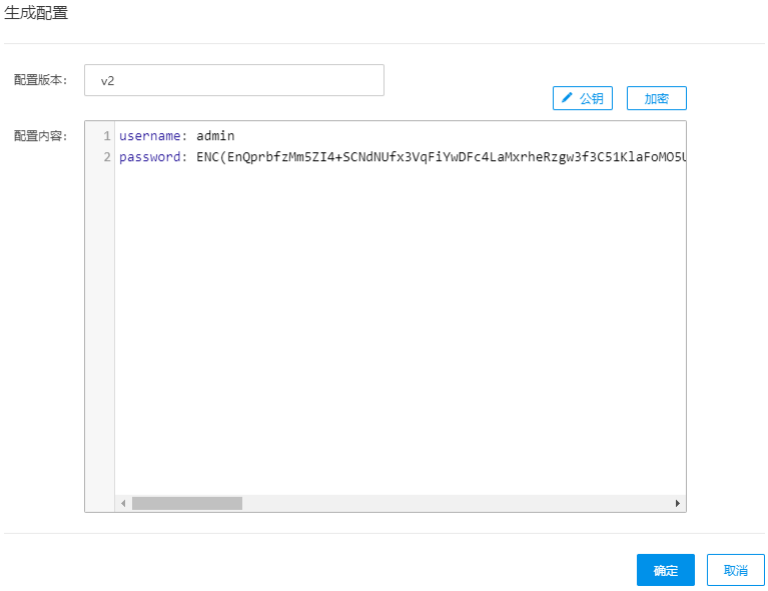
需要加密可写为：

```
password: DEC(123456)
```

1. 控制台使用
   在配置框中填入配置，对需要加密的内容使用"DEC()"包裹,如下对password内容需要加密。



输入公钥后点击加密按钮，获得加密后的配置。



将加密后的配置进行保存。

2. 本地使用
   本地可用maven插件来进行加密。例如在application.properties中填写配置，修改maven插件命令参数,指定加密方式与公钥。

```
-Djasypt.encryptor.publicKeyFormat=PEM
-Djasypt.encryptor.public-key-string="-----BEGIN PUBLIC KEY----- MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAv3A/w4LHCGQtmVVV1O1n 4ATHuJ4q3CqlzPq98a5xGc2IdPMwUspb6iloGCzq5Ef/OOdarkv7PWJjYolp49iK Sr/cBkCfcKzUu5QzsQoyYY+t8ZRoEp/OmZRwLUDnUn8SSm5KdMKjA+HjCpeRi1Tg SSQWazlTJmpUDQIIldFsMuCQkns1BYUXZ9GvLaZMuub1OXcmmJZJD1N55AHY5jjn pBBdiJOmWVti3UQmVMOgOIMEaZDgmCX1gImHCaG4eXT5neE6mLGtuQqCOZjjKLb6 ieOOp8jwVk9GYulnqmBzf5QuUSArDgeG1yahsXH2I8dtDGc7qzneB5sckEZ+Kh4F RwIDAQAB -----END PUBLIC KEY-----"
```

运行maven命令后得到密文如下：

password:
ENC(q2DOfazeu1uLWht//lmrdwpQEcRX+TYzDTAeqGivFcyiZDTV9GYzEEfyUh1JYjt/PrLTkp+XSv3Aps1Xqwmzji8480Z7nIyXgS1IBKwIZekiRiIEHpOTPg2Hc9IuZqaJ4c1OR6jf1VTrkZ8yGRPxdBOHvf9/UswtJ1cm45roErARj8

配置解密
用户可在控制台填入私钥对密文进行解密。

![4.png](http://fe-frame.ks3-cn-beijing.ksyun.com/project/cms/c26db378171d6b9ad6f076de26f042e6)

解密后

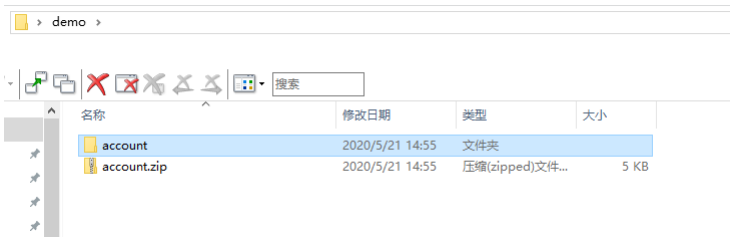
![5.png](http://fe-frame.ks3-cn-beijing.ksyun.com/project/cms/0b30efd6aa2516e8a34500d141087a2e)

服务开发

实现服务开发
通过一个简单的实例说明如何通过kmse创建一个用于“服务开发”的项目。

一、新建工程包

1. 按照所需填写各项参数，如图所示：
2. 点击“下载”按钮，下载解压后得到工程项目：

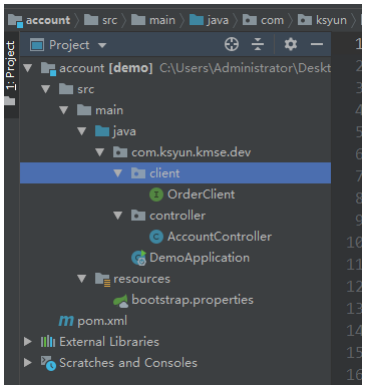


二、修改本地Maven的setting.xml文件

使用kmse的脚手架功能构建项目，为了连接到kmse的maven仓库，请编辑您的settings.xml。可以直接复制以下文件。一般情况 maven 的通用 settings.xml 在 .m2 文件夹下。

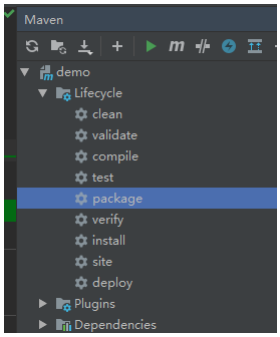
```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">

    <localRepository>/Users/chengjin/Public/work/maven/repository</localRepository>
    <mirrors>
        <mirror>
            <id>ezOne-maven</id>
            <mirrorOf>*,!ksyun_snapshot_maven_maven,!ksyun_release_maven_maven</mirrorOf>
            <name>ezOne maven</name>
            <url>https://ezone.work/ezPackage/mirrors/maven/</url>
        </mirror>
    </mirrors>
    <profiles>
        <profile>
            <id>profile_ksyun</id>
            <repositories>
                <repository>
                    <id>ksyun_snapshot_maven_maven</id>
                    <name>ksyun_snapshot_maven_maven</name>
                    <url>https://ezone.work/pkg/ksyun/maven/maven/snapshot/</url>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </repository>
                <repository>
                    <id>ksyun_release_maven_maven</id>
                    <name>ksyun_release_maven_maven</name>
                    <url>https://ezone.work/pkg/ksyun/maven/maven/release/</url>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                </repository>
            </repositories>
            <pluginRepositories>
                <pluginRepository>
                    <id>ksyun_snapshot_maven_maven</id>
                    <url>https://ezone.work/pkg/ksyun/maven/maven/snapshot/</url>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </pluginRepository>
                <pluginRepository>
                    <id>ksyun_release_maven_maven</id>
                    <url>https://ezone.work/pkg/ksyun/maven/maven/release/</url>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                </pluginRepository>
            </pluginRepositories>
        </profile>
    </profiles>
    <servers>
        <server>
            <id>ksyun_release_maven_maven</id>
            <username>ksyun_kmse</username>
            <password>e19da4105183477a873e9af4cb934335159290419 4285</password>
        </server>
        <server>
            <id>ksyun_snapshot_maven_maven</id>
            <username>ksyun_kmse</username>
            <password>e19da4105183477a873e9af4cb934335159290419 4285</password>
        </server>
    </servers>
    <activeProfiles>
        <activeProfile>profile_ksyun</activeProfile>
    </activeProfiles>
</settings>
```

三、工程包导入IDE
工程导入IDE后，如下图所示：

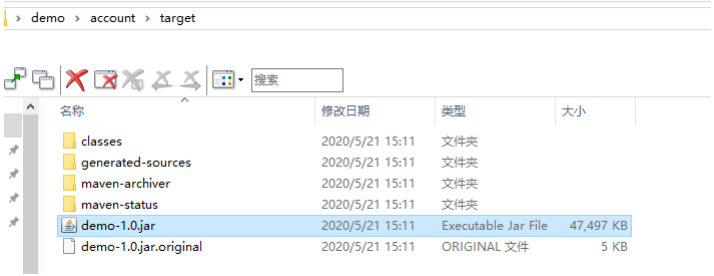


其中已经集成了springcloud体系中的大多数组件，工程中包含了远程调用的模板client（OrderClient）。开发者可以在其中添加各种业务逻辑。

四、将工程包打包成可上传的构建包
执行Maven的package命名，如下图所示：



构建完成后得到构建包，如下图所示：

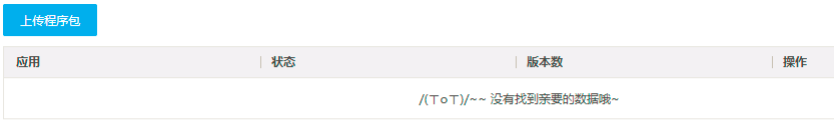


至此就完成了服务的开发。

服务管理

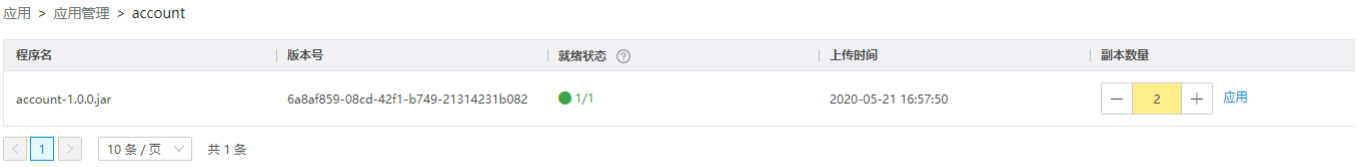
使用kmse实现应用的管理
通过一个简单的实例说明如何通过kmse上传和管理一个应用。

一、上传构建包
在服务管理模块，点击选择好对应的集群和命名空间后，点击上传按钮，如图所示：



之后上传在“服务开发”模块中构建完成的jar包，上传应用的各项参数可根据实际需要调整，如不填写可全部采用默认值：单击确定后开始上传，如下图所示：一段时间的打包运行后，可以看到程序已经在集群中的运行状态。

二、对应用进行管理
应用会根据程序中指定的spring.application.name自动分组，单击应用，进入到应用的管理界面，在这里可以根据业务负载的实际情况，对某个版本的实例做扩缩容，点击“副本数量”下方的加减号，点击“应用”按钮，即可对应用实例进行动态伸缩。



如果不需要该实例，可以点击“删除”按钮，对实例进行删除。

API列表

操作场景
KMSE提供API列表功能，采用的是swagger2，用户可在控制台查看以及调用API列表。

前提条件
向工程中添加依赖，在pom.xml中添加以下依赖：

```
<dependency>
  <groupId>com.ksyun.kmse</groupId>
  <artifactId>spring-cloud-kmse-swagger</artifactId>
  <version>${version}</version>
</dependency>
```

操作步骤

1. 给controller添加注解

常用注解

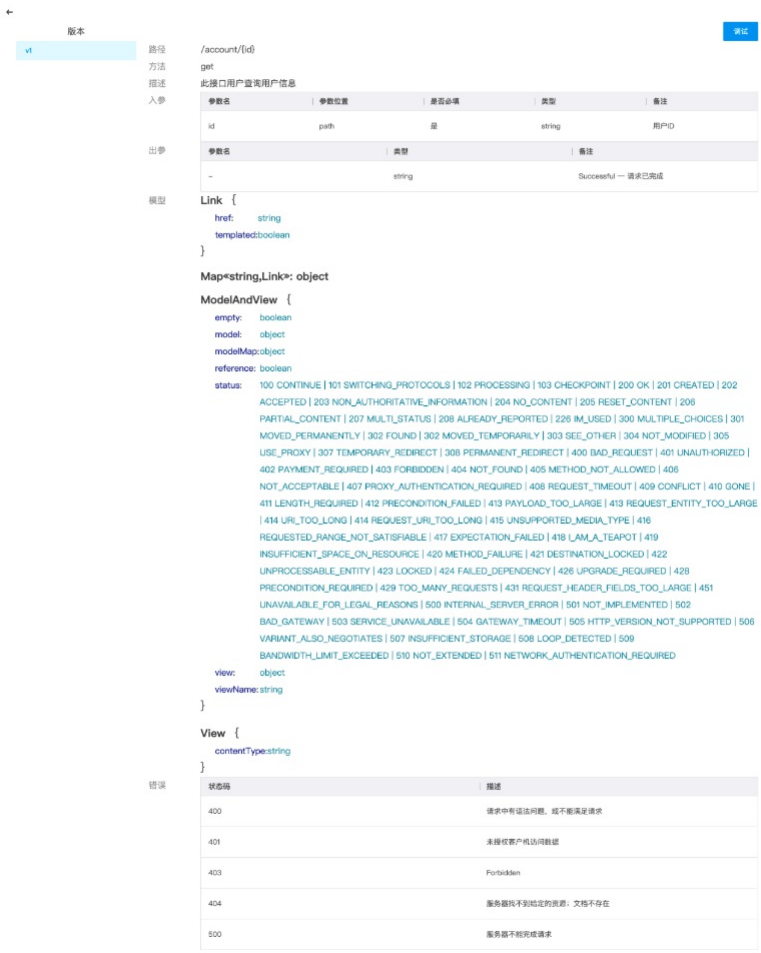
注解	示例	属性	值	备注
@Api	@Api(value = "xxx", description = "xxx")	value	字符串	可用在class头上,class描述
		description	字符串	-
@ApiOperation	@ApiOperation(value = "xxx", notes = "xxx")	value	字符串	可用在方法头上.参数的描述容器
		notes	字符串	-
@ApiImplicitParams	@ApiImplicitParams({@ApiImplicitParam{...},@ApiImplicitParam{...},...})	{}	@ApiImplicitParam数组	可用在方法头上.参数的描述容器
@ApiImplicitParam	@ApiImplicitParam(name = "xxx", value = "xxx", required = true, dataType = "xxx", paramType = "xxx", defaultValue = "xxx")	name	字符串与参数命名对应	可用在@ApiImplicitParams里
		value	字符串	参数中文描述
		required	布尔值	true/false
		dataType	字符串	参数类型
		paramType	字符串	参数请求方式:query/path
		defaultValue	字符串	在api测试中默认值
@ApiResponses	@ApiResponses({@ApiResponse{...},@ApiResponse{...},...})	{}	@ApiResponse数组	可用在方法头上.参数的描述容器
@ApiResponse	@ApiResponse(code = 200, message = "Successful")	code	整型	可用在@ApiResponses里
		message	字符串	错误描述

代码示例

```
@RequestMapping({"/account"})
@RestController
@Api(value = "API - AccountController")
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    public AccountController() {
    }

    @ApiOperation(value = "查询用户接口", notes="此接口用户查询用户信息", response= String.class)
    @ApiImplicitParams({
            @ApiImplicitParam(name = "id", value = "用户ID", required = true,
                    dataType = "string", paramType = "path", defaultValue = "1"),
    })
    @ApiResponses(value = {
            @ApiResponse(code = 200, message = "Successful — 请求已完成"),
            @ApiResponse(code = 400, message = "请求中有语法问题，或不能满足请求"),
            @ApiResponse(code = 401, message = "未授权客户机访问数据"),
            @ApiResponse(code = 404, message = "服务器找不到给定的资源：文档不存在"),
            @ApiResponse(code = 500, message = "服务器不能完成请求")}
    )
    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用account " + id);
        return id + "";
    }
}
```

2. 启动服务
   访问本地服务：

3. 将应用上传至KMSE控制台。



4. 控制台调试。

容错管理

操作场景
KMSE（微服务引擎）使用Resilience4j库，相比Hystrix更轻量，对外部依赖更少。

核心模块

- resilience4j-circuitbreaker: 断路器
- resilience4j-ratelimiter: 限流
- resilience4j-bulkhead: 隔离板
- resilience4j-retry: 自动重试

前提条件
向工程中添加依赖，在pom.xml中添加以下依赖

```
<dependency>
  <groupId>com.ksyun.kmse</groupId>
  <artifactId>spring-cloud-kmse-starter-fault-tolerant</artifactId>
  <version>${version}</version>
</dependency>
```

注意
如果Retry、CircuitBreaker、Bulkhead、RateLimiter同时注解在方法上，默认的顺序是Bulkhead->RateLimiter->CircuitBreaker->Retry，即先控制并发再限流然后熔断最后重试。

操作步骤
CircuitBreaker(断路器)使用

1. 创建配置
   在控制台“服务治理>>容错管理”页面新建容错类型规则，如下图

各项对应可配置参数如下：

配置参数	默认值	描述
failureRateThreshold	50	熔断器关闭状态和半开状态使用的同一个失败率阈值
ringBufferSizeInHalfOpenState	10	熔断器半开状态的缓冲区大小，会限制线程的并发量，例如缓冲区为10则每次只会允许10个请求调用后端服务
ringBufferSizeInClosedState	100	熔断器关闭状态的缓冲区大小，不会限制线程的并发量，在熔断器发生状态转换前所有请求都会调用后端服务
waitDurationInOpenState	60(s)	熔断器从打开状态转变为半开状态等待的时间,单位：（ms，s，m）
recordExceptions	empty	需要记录为失败的异常列表
ignoreExceptions	empty	需要忽略的异常列表
recordFailurePredicate	throwable -> true	自定义的谓词逻辑用于判断异常是否需要记录或者需要忽略，默认所有异常都进行记录

对应application.yml配置(本地开发可直接编写配置，以下例子依据此配置)

```
resilience4j.circuitbreaker.instances:
  backendA:
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
```

2. 调用方法
   CircuitBreaker目前支持两种方式调用，一种是程序式调用，一种是AOP使用注解的方式调用。

注解方式调用
首先在要保护的方法上使用@CircuitBreaker(name="",fallbackMethod="")注解，其中name是要使用的断路器的名称， fallbackMethod是要使用的降级方法，降级方法必须和原方法放在同一个类中，且降级方法的返回值和原方法相同，输入参数需要添加额外的exception参数，示例代码：

- BusinessService接口定义各类情况的方法

```
public interface BusinessService {
     String failure();

     String failureNotAOP() throws Throwable;

     String success();

     String failureWithFallback();

     String ignoreException();

}
```

- 实现类BusinessServiceImpl

```
import com.ksyun.exception.BusinessException;
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Service;
import org.springframework.web.client.HttpServerErrorException;

@Service
public class BusinessServiceImpl implements BusinessService {
  /**
   * 抛出record异常
   * @return
   */
  @Override
  @CircuitBreaker(name = "backendA")
  public String failure() {
      throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, "接口服务异常");
  }

  @Override
  public String failureNotAOP() throws Throwable {
      return null;
  }
```

```
    @Override
    public String success() {
        return null;
    }

    /**
     * 带fallback处理
     * @return
     */
    @Override
    @CircuitBreaker(name = "backendA", fallbackMethod = "fallback")
    public String failureWithFallback() {
        return failure();
    }

    /**
     * 降级处理
     * @param ex
     * @return
     */
    private String fallback(HttpServerErrorException ex) {
        return "fallback: Recovered HttpServerErrorException: " + ex.getMessage();
    }

    /**
     * 抛出ignore异常
     * @return
     */
    @CircuitBreaker(name = "backendA", fallbackMethod = "fallback")
    public String ignoreException() {
        throw new BusinessException("This exception is ignored by the CircuitBreaker of backend A");
    }
}
```

- 自定义异常BusinessException

```
public class BusinessException extends RuntimeException {

    public BusinessException(String message) {
        super(message);
    }
}
```

- 自定义失败谓词,返回true则记录异常，返回false则忽略异常

```
import java.util.function.Predicate;

public class RecordFailurePredicate implements Predicate<Throwable> {
    @Override
    public boolean test(Throwable throwable) {
        return !(throwable instanceof BusinessException);
    }
}
```

通过一个请求接口模拟各类情况

```
@RestController
@RequestMapping(value = "/backendA")
public class BackendAController {

    @Autowired
    private BusinessService businessService;

    @GetMapping("failure/{type}")
    public String failure(@PathVariable("type") String type) throws Throwable {
        if ("1".equals(type)){
            return businessService.failure();
        } else if ("2".equals(type)){
            return businessService.failureWithFallback();
        } else if("3".equals(type)){
            return businessService.ignoreException();
        } else if("4".equals(type)) {
            return businessService.failureNotAOP();
        } else {
            return businessService.success();
        }
    }
}
```

- 调用接口 （type=1）

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T13:59:49.260+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T13:59:51.689+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T13:59:52.549+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T13:59:53.228+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T13:59:53.947+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T13:59:54.578+0000","status":500,"error":"Internal Server Error","message":"CircuitBreaker 'backendA' is OPEN and does not perm
```

前5次返回失败错误信息后（配置中缓存区大小为5，缓存区被填满后开始计算失败率），第6次直接返回断路器backenA被打开，之后1分钟内的请求都将快速失败。

- 调用接口（type=2）,带fallback处理

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
this is fallback with ex: 500 接口服务异常%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
this is fallback with ex: 500 接口服务异常%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
this is fallback with ex: 500 接口服务异常%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
this is fallback with ex: 500 接口服务异常%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
this is fallback with ex: 500 接口服务异常%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
fallback: Recovered Throwable: CircuitBreaker 'backendA' is OPEN and does not permit further calls%
➜  ~
```

第6次断路器backenA被打开，请求被降级至fallback方法中处理。

- 调用接口（type=3）,ignore异常

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:47.537+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:50.025+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:50.860+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:51.592+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:52.220+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:52.809+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~ curl http://127.0.0.1:8080/backendA/failure/3
{"timestamp":"2020-05-20T14:18:53.592+0000","status":500,"error":"Internal Server Error","message":"This exception is ignored by the CircuitBreak
➜  ~
```

当抛出异常为ignore时，失败次数不会被记入缓冲区，也无法触发断路器开关

- 断路器打开一分钟之后变为半开状态，调用接口（type=5），正常响应

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/5
hello, success!%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/5
hello, success!%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/5
hello, success!%
```

3次请求成功率超过50%，断路器关闭，否则断路器重新开启

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/4
Hello World from backend A%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T14:25:42.360+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T14:25:43.853+0000","status":500,"error":"Internal Server Error","message":"500 接口服务异常","path":"/backendA/failure/1"}%
➜  ~ curl http://127.0.0.1:8080/backendA/failure/1
{"timestamp":"2020-05-20T14:25:45.181+0000","status":500,"error":"Internal Server Error","message":"CircuitBreaker 'backendA' is OPEN and does not perm
➜  ~
```

（失败率超过50%，断路器重新打开）

程序式调用
使用decorate包装服务的方法，再使用Try.of().recover()进行降级处理。

```
@Service(value = "businessAService")
public class BusinessAService implements BusinessService {
    @Autowired
    private CircuitBreakerRegistry circuitBreakerRegistry;
    @Override
    public String failureNotAOP() throws Throwable {
        CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker("backendA");
        CheckedFunction0<String> checkedSupplier=CircuitBreaker.decorateCheckedSupplier(circuitBreaker, backendAConnector::failureNotAOP);
        Try<String> result = Try.of(checkedSupplier).recover(CallNotPermittedException.class,    throwable -> {
            Log.info("服务降级");
            return "fallback";
        });
        return result.get();
    }
}
```

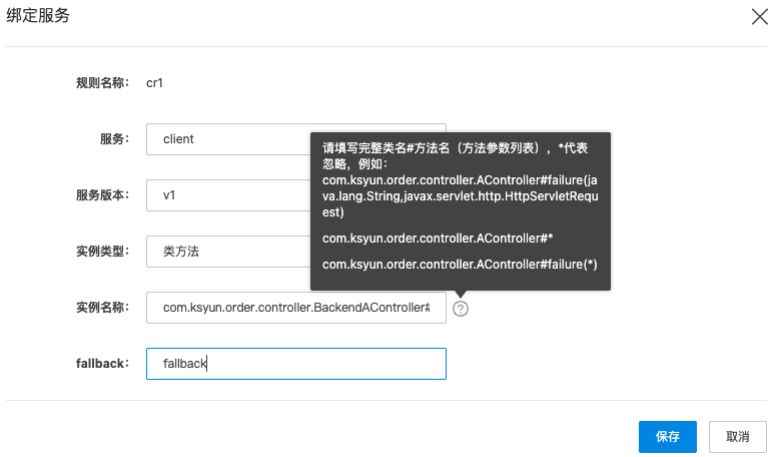
2.0
2.0版本配置实例新增类方法、URI

使用方法
启动类添加@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")开启功能支持,同时添加AspectPointcutScanPreparedEventListener监听事件。

```
@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")
public class DemoApplication extends SpringBootServletInitializer {

     public static void main(String[] args) {
         SpringApplication springApplication = new SpringApplication(DemoApplication.class);
         springApplication.addListeners(new AspectPointcutScanPreparedEventListener());
         springApplication.run(args);
```

```
    }
}
```

AOP新增类方法以及URI拦截，配置时可将实例名写为要拦截的方法或uri。
注意：注解、类方法、uri这3种实例配置方式本质上都是作用在方法上，所以不要对同一方法配置多套实例，会导致效果成指数级。例如：方法a上配置了注解，类方法以及uri3种重试规则（重试3次），实际会导致重试3x3x3=27次。
当yaml配置key存在特殊字符时，需要用"[]"包裹。

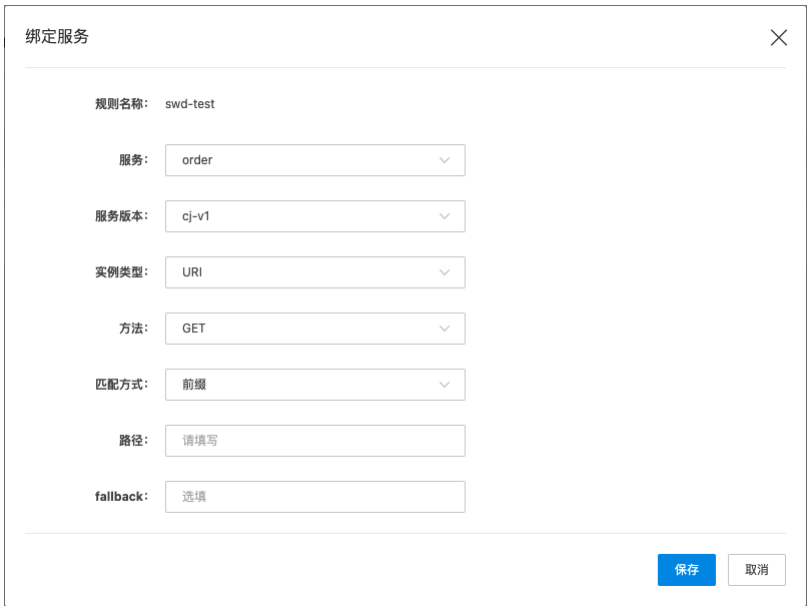


- 类名、方法实例配置

实例名称填写规则如下：

```
ClassName#MethodName(args...)

格式说明:
ClassName:完整的类名，例如：com.ksyun.order.controller.BackendAController
MethodName:方法名

例子:
# 匹配类下面所有方法（*代表所有方法）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#*]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 匹配某个名字的方法（因存在重载方法，*代表忽略参数）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(*)]":
  ...
# 匹配某个具体方法(需写上完整参数列表)：
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(java.lang.String,javax.servlet.http.HttpServletRequest)]":
  ...
```

- URI配置




```
matchType#method#uri

格式说明:
matchType:prefix、suffix、regex、match四种
method:request method,如:GET(大写)
uri:需要匹配的值

例子:
# 前缀匹配**
resilience4j.circuitbreaker.instances:
  "[prefix#GET#/failure/]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
```

```
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 后缀匹配
resilience4j.circuitbreaker.instances:
  "[suffix#GET#failure]":
   ...
# 完全匹配
resilience4j.circuitbreaker.instances:
  "[match#GET#/failure/1]":
   ...
# 正则
resilience4j.circuitbreaker.instances:
  "[regex#GET#/.*/failu.*]":
   ...
```

- Fallback 配置

```
resilience4j.circuitbreaker.fallback:
 "[com.ksyun.order.controller.BackendAController#*]": fallback
```

Retry(重试)使用

1. 创建配置
   在控制台“服务治理>>容错管理”页面新建重试类型规则，如下图：

各项对应可配置参数

配置参数	默认值	描述
maxAttempts	3	最大重试次数
waitDuration	500(ms)	固定重试间隔,单位（ms, s, m）
enableExponentialBackoff	false	是否允许使用指数退避算法进行重试间隔时间的计算
retryExceptionPredicate	throwable -> true	自定义异常重试规则，需要重试的返回true
retryExceptions	empty	需要重试的异常列表
ignoreExceptions	empty	需要忽略的异常列表

对应application.yml配置(本地开发可直接编写配置，以下例子依据此配置)

```
resilience4j.retry.instances:
  backendA:
    enableExponentialBackoff: true
    exponentialBackoffMultiplier: 2
    retryExceptionPredicate: com.ksyun.exception.RetryOnExceptionPredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    maxRetryAttempts: 3
    retryExceptions:
      - org.springframework.web.client.HttpServerErrorException
    waitDuration: 5s
```

2. 调用方法
   还是以之前的接口为例。Retry支持注解方式和程序式两种方式的调用。

注解方式调用
首先在方法上使用@Retry(name="",fallbackMethod="")注解，其中name是要使用的重试器实例的名称，fallbackMethod是要使用的降级方法：

```
/**
* 抛出需retry异常
* @return
*/
@Override
@Retry(name = "backendA")
public String failure() {
    throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, "接口服务异常");
}

/**
* 带fallback处理
* @return
*/
@Override
@Retry(name = "backendA", fallbackMethod = "fallback")
public String failureWithFallback() {
    return failure();
}

/**
* 降级处理
* @param ex
* @return
*/
private String fallback(HttpServerErrorException ex) {
    return "this is fallback with ex: " + ex.getMessage();
}

/**
* 抛出ignore异常
* @return
*/
@Override
@Retry(name = "backendA")
public String ignoreException() {
    throw new BusinessException("This exception is ignored by the CircuitBreaker of backend A");
}
```

- 自定义重试异常谓词(需要重试的异常返回true,不需要则返回false)

```
import java.util.function.Predicate;
```

```
public class RetryExceptionPredicate implements Predicate<Throwable> {
    @Override
    public boolean test(Throwable throwable) {
        return !(throwable instanceof BusinessException);
    }
}
```

- 调用接口（type=1）

```
➜  ~ ab http://127.0.0.1:8080/backendA/failure/1
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /backendA/failure/1
Document Length:        153 bytes

Concurrency Level:      1
Time taken for tests:   15.038 seconds
Complete requests:      1
Failed requests:        0
Non-2xx responses:      1
Total transferred:      272 bytes
HTML transferred:       153 bytes
Requests per second:    0.07 [#/sec] (mean)
Time per request:       15037.536 [ms] (mean)
Time per request:       15037.536 [ms] (mean, across all concurrent requests)
Transfer rate:          0.02 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing: 15037 15037   0.0  15037   15037
Waiting:    15037 15037   0.0  15037   15037
Total:      15037 15037   0.0  15037   15037
➜  ~
```

因为抛出异常为retryExceptions,所以请求重试了3次，等待时间为5s,最终耗时15s

- 调用接口（type=2）

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/2
this is fallback with ex: 500 接口服务异常%
➜  ~
```

重试之后仍抛出异常，则服务降级，调用fallback方法

- 调用接口（type=3）

```
➜  ~ ab http://127.0.0.1:8080/backendA/failure/3
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /backendA/failure/3
Document Length:        191 bytes

Concurrency Level:      1
Time taken for tests:   0.007 seconds
Complete requests:      1
Failed requests:        0
Non-2xx responses:      1
Total transferred:      310 bytes
HTML transferred:       191 bytes
Requests per second:    142.19 [#/sec] (mean)
Time per request:       7.033 [ms] (mean)
Time per request:       7.033 [ms] (mean, across all concurrent requests)
Transfer rate:          43.04 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     7    7   0.0      7       7
Waiting:        6    6   0.0      6       6
Total:          7    7   0.0      7       7
➜  ~
```

因抛出异常为ignoreExceptions,所以没有进行重试

程序式调用

```
@Service(value = "businessAService")
public class BusinessAService implements BusinessService {
    @Autowired
    private RetryRegistry retryRegistry;

    @Override
    public String failureNotAOP() throws Throwable {
        Retry retry = retryRegistry.retry("backendA");
        CheckedFunction0<String> checkedSupplier = Retry.decorateCheckedSupplier(retry, backendAConnector::failureNotAOP);
        Try<String> result = Try.of(checkedSupplier).recover(CallNotPermittedException.class, throwable -> {
            Log.info("服务降级");
            return "fallback";
        });
        return result.get();
    }
}
```

Retry可以与CircuitBreaker 一起使用有2种调用方式，一种是先用重试组件装饰，再用熔断器装饰，这时熔断器的失败需要等重试结束才计算，另一种是先用熔断器装饰，再用重试组件装饰，这时每次调用服务都会记录进熔断器的缓冲环中，需要注意的是，第二种方式需要把CallNotPermittedException放进重试组件的白名单中，因为熔断器打开时重试是没有意义的。

```
@Override
public String failureNotAOP() throws Throwable {
    CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker("backendA");
    Retry retry = retryRegistry.retry("backendA");
    // 先用重试组件包装，再用熔断器包装
    CheckedFunction0<String> retrySupplier= Retry.decorateCheckedSupplier(retry, backendAConnector::failureNotAOP);
    CheckedFunction0<String> checkedSupplier =CircuitBreaker.decorateCheckedSupplier(circuitBreaker, retrySupplier);
    // 使用Try.of().recover()调用并进行降级处理
     ry<String> result = Try.of(checkedSupplier).recover(CallNotPermittedException.class, throwable -> {
        Log.info("断路器打开");
        return "fallback";
    });
    return result.get();
}
```

但是需要注意同时注解重试组件和熔断器的话，是按照第二种方案，即每一次请求都会被熔断器记录。

```
/**
* 抛出record异常
* @return
*/
@Override
@Retry(name = "backendA")
@CircuitBreaker(name = "backendA", fallbackMethod = "fallback")
public String failure() {
    throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, "接口服务异常");
}
```

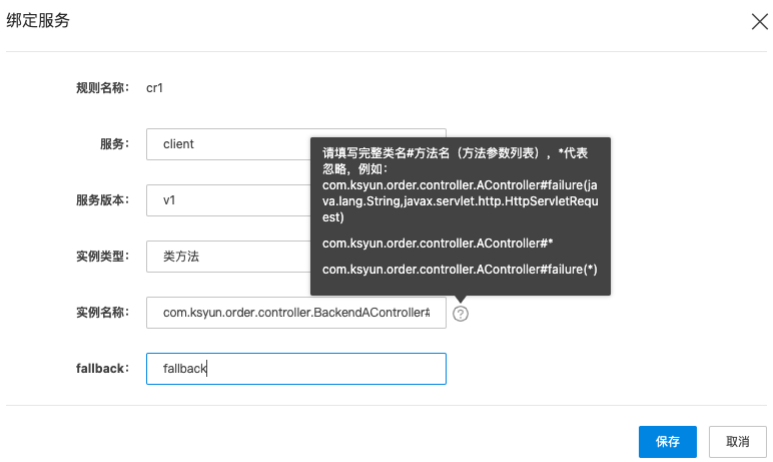
2.0
2.0版本配置实例新增类方法、URI

使用方法
启动类添加@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")开启功能支持,同时添加AspectPointcutScanPreparedEventListener监听事件。

```
@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")
public class DemoApplication extends SpringBootServletInitializer {

     public static void main(String[] args) {
         SpringApplication springApplication = new SpringApplication(DemoApplication.class);
         springApplication.addListeners(new AspectPointcutScanPreparedEventListener());
         springApplication.run(args);
     }
```

```
}
```

AOP新增类方法以及URI拦截，配置时可将实例名写为要拦截的方法或uri。
注意：注解、类方法、uri这3种实例配置方式本质上都是作用在方法上，所以不要对同一方法配置多套实例，会导致效果成指数级。例如：方法a上配置了注解，类方法以及uri3种重试规则（重试3次），实际会导致重试3x3x3=27次。
当yaml配置key存在特殊字符时，需要用"[]"包裹。

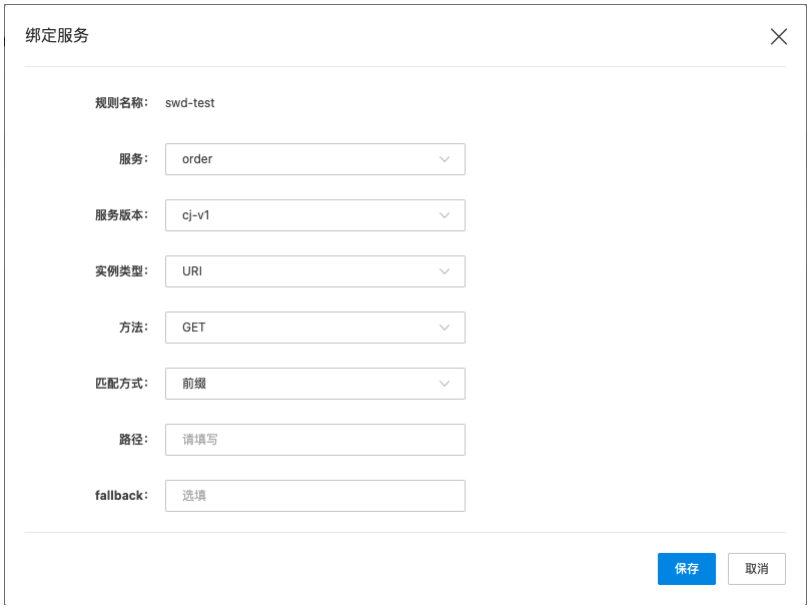


- 类名、方法实例配置

实例名称填写规则如下：

```
ClassName#MethodName(args...)

格式说明：
ClassName:完整的类名，例如：com.ksyun.order.controller.BackendAController
MethodName:方法名

例子：
# 匹配类下面所有方法（*代表所有方法）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#*]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 匹配某个名字的方法（因存在重载方法，*代表忽略参数）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(*)]":
  ...
# 匹配某个具体方法(需写上完整参数列表)：
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(java.lang.String,javax.servlet.http.HttpServletRequest)]":
  ...
```

- URI配置




```
matchType#method#uri

格式说明：
matchType:prefix、suffix、regex、match四种
method:request method,如:GET(大写)
uri:需要匹配的值

例子：
# 前缀匹配**
resilience4j.circuitbreaker.instances:
  "[prefix#GET#/failure/]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
```

```
      ringBufferSizeInClosedState: 5
      ringBufferSizeInHalfOpenState: 3
      waitDurationInOpenState: 60s
  # 后缀匹配
  resilience4j.circuitbreaker.instances:
    "[suffix#GET#failure]":
      ...
  # 完全匹配
  resilience4j.circuitbreaker.instances:
    "[match#GET#/failure/1]":
      ...
  # 正则
  resilience4j.circuitbreaker.instances:
    "[regex#GET#/.*/failu.*]":
      ...
```

- Fallback 配置

```
resilience4j.circuitbreaker.fallback:
  "[com.ksyun.order.controller.BackendAController#*]": fallback
```

RateLimter(限流)使用

1. 创建配置
   在控制台“服务治理>>容错管理”页面新建限流类型规则，如下图：

各项对应可配置参数

配置参数	默认值	描述
timeoutDuration	5[s]	线程等待权限的默认等待时间，单位（ms, s, m）
limitRefreshPeriod	500[ns]	单位时间：权限刷新的时间，每个周期结束后，RateLimiter将会把权限计数设置为limitForPeriod的值
limiteForPeriod	50	单位时间通过量：一个限制刷新期间的可用权限数

对应application.yml配置(本地开发可直接编写配置，以下例子依据此配置)

```
resilience4j.ratelimiter.instances:
  backendA:
    limitForPeriod: 5
    limitRefreshPeriod: 20s
    timeoutDuration: 5s
```

2. 调用方式
   RateLimter目前支持两种方式调用，一种是程序式调用，一种是AOP使用注解的方式调用。
   注解方式调用
   调用一个正常接口，不抛出异常，为了让结果明显一些，程序中sleep5秒。

```
@Override
@RateLimiter(name="backendA")
public String success() {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "Hello World from backend A";
}
```

ab:Apache Bench,一个web压测工具，用来模拟并发，参考：http://httpd.apache.org/docs/2.4/programs/ab.html

用ab工具，同时发送5个请求，可以看到4个失败，只有1个成功。

```
➜  ~ ab -c5 -n5 http://127.0.0.1:8080/backendA/failure/5
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /backendA/failure/5
Document Length:        26 bytes

Concurrency Level:      5
Time taken for tests:   10.366 seconds
Complete requests:      5
Failed requests:        4
   (Connect: 0, Receive: 0, Length: 4, Exceptions: 0)
Non-2xx responses:      4
Total transferred:      1367 bytes
HTML transferred:       758 bytes
Requests per second:    0.48 [#/sec] (mean)
Time per request:       10365.701 [ms] (mean)
Time per request:       2073.140 [ms] (mean, across all concurrent requests)
Transfer rate:          0.13 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
Processing:  5119 5222  57.1   5247    5247
Waiting:     5118 5221  57.6   5247    5247
Total:       5120 5222  57.1   5247    5248
```

修改配置limitRefreshPeriod: 1s，重新测试，全部成功。可以看出即使服务还没完成，依然可以放入，只与时间有关，与线程无关。

```
➜  ~ ab -c1 -n5 http://127.0.0.1:8080/backendA/failure/5
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /backendA/failure/5
Document Length:        26 bytes

Concurrency Level:      1
Time taken for tests:   5.083 seconds
Complete requests:      5
Failed requests:        0
Total transferred:      795 bytes
HTML transferred:       130 bytes
Requests per second:    0.98 [#/sec] (mean)
Time per request:       1016.658 [ms] (mean)
Time per request:       1016.658 [ms] (mean, across all concurrent requests)
Transfer rate:          0.15 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:  1009 1016  14.2   1011    1042
Waiting:     1007 1015  14.0   1010    1040
Total:       1009 1017  14.2   1011    1042
```

程序式调用

调用方法与前面类似，装饰方法之后用Try.of().recover()来执行

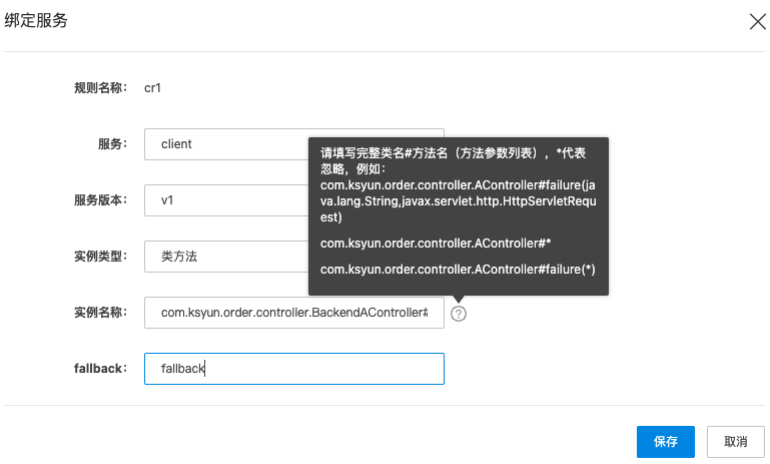
```
@Service(value = "businessAService")
public class BusinessAService implements BusinessService {
    @Autowired
    private RateLimiterRegistry rateLimiterRegistry;

    @Override
    public String failureNotAOP() throws Throwable {
        RateLimiter rateLimiter = rateLimiterRegistry.rateLimiter("name");
        CheckedFunction0<String> ratelimiterSupplier = RateLimiter.decorateCheckedSupplier(rateLimiter, backendAConnector::failureNotAOP);
        Try<String> result = Try.of(ratelimiterSupplier).recover(CallNotPermittedException.class, throwable -> {
            Log.info("服务降级");
            return "fallback";
        });
        return result.get();
    }
}
```

2.0
2.0版本配置实例新增类方法、URI

使用方法
启动类添加@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")开启功能支持,同时添加AspectPointcutScanPreparedEventListener监听事件。

```
@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")
public class DemoApplication extends SpringBootServletInitializer {

     public static void main(String[] args) {
         SpringApplication springApplication = new SpringApplication(DemoApplication.class);
         springApplication.addListeners(new AspectPointcutScanPreparedEventListener());
         springApplication.run(args);
     }
 }
```

AOP新增类方法以及URI拦截，配置时可将实例名写为要拦截的方法或uri。
注意：注解、类方法、uri这3种实例配置方式本质上都是作用在方法上，所以不要对同一方法配置多套实例，会导致效果成指数级。例如：方法a上配置了注解，类方法以及uri3种重试规则（重试3次），实际会导致重试3x3x3=27次。
当yaml配置key存在特殊字符时，需要用"[]"包裹。

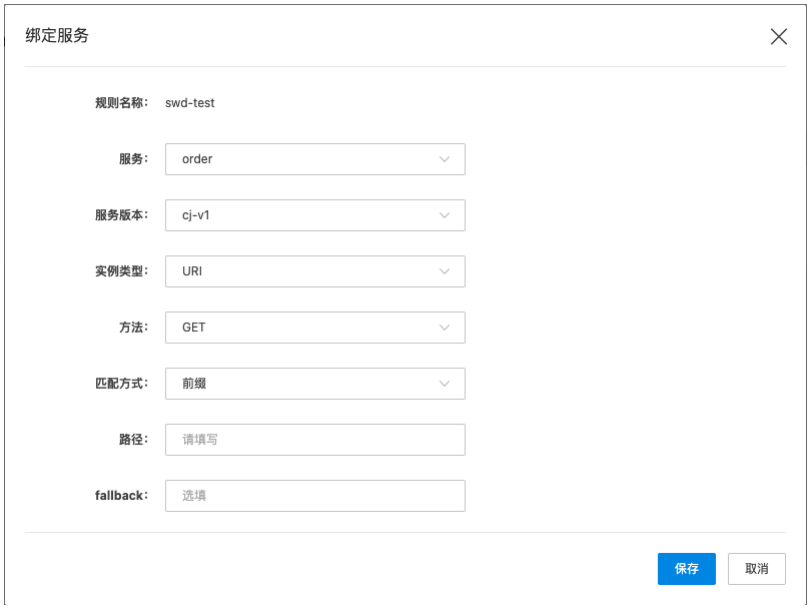


- 类名、方法实例配置

实例名称填写规则如下：

```
ClassName#MethodName(args...)

格式说明：
ClassName:完整的类名，例如：com.ksyun.order.controller.BackendAController
MethodName:方法名

例子:
# 匹配类下面所有方法（*代表所有方法）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#*]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 匹配某个名字的方法（因存在重载方法，*代表忽略参数）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(*)]":
  ...
# 匹配某个具体方法(需写上完整参数列表)：
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(java.lang.String,javax.servlet.http.HttpServletRequest)]":
  ...
```

- URI配置




```
matchType#method#uri

格式说明:
matchType:prefix、suffix、regex、match四种
method:request method,如:GET(大写)
uri:需要匹配的值

例子:
# 前缀匹配**
resilience4j.circuitbreaker.instances:
  "[prefix#GET#/failure/]":
     failureRateThreshold: 50
     recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
     ignoreExceptions:
       - com.ksyun.order.exception.BusinessException
     recordExceptions:
       - org.springframework.web.client.HttpServerErrorException
     ringBufferSizeInClosedState: 5
     ringBufferSizeInHalfOpenState: 3
     waitDurationInOpenState: 60s
# 后缀匹配
resilience4j.circuitbreaker.instances:
  "[suffix#GET#failure]":
    ...
# 完全匹配
resilience4j.circuitbreaker.instances:
  "[match#GET#/failure/1]":
    ...
# 正则
resilience4j.circuitbreaker.instances:
  "[regex#GET#/.*/failu.*]":
    ...
```

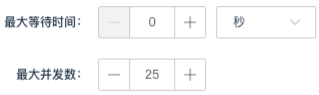
- Fallback 配置

```
resilience4j.circuitbreaker.fallback:
  "[com.ksyun.order.controller.BackendAController#*]": fallback
```

Bulkhead(舱壁隔离)使用

Resilence4j的Bulkhead提供两种实现，一种是基于信号量的，另一种是基于有等待队列的固定大小的线程的，由于基于信号量的Bulkhead能很好地在多线程和I/O模型下工作，所以推荐基于信号量的Bulkhead的使用。

1. 创建配置



各项对应可配置参数

配置参数	默认值	描述
maxConcurrentCalls	25	可允许的最大并发线程数
maxWaitDuration	0	尝试进入饱和舱壁时应阻止线程的最大时间

对应application.yml配置(本地开发可直接编写配置，以下例子依据此配置)

```
resilience4j.bulkhead.instances:
  backendA:
    maxConcurrentCalls: 2
    maxWaitDuration: 100
```

2. 调用方式
   Bulkhead目前支持两种方式调用，一种是程序式调用，一种是AOP使用注解的方式调用
   注解方式调用
   首先在连接器方法上使用@Bulkhead(name="", fallbackMethod="", type="")注解，其中name是要使用的Bulkhead实例的名称，fallbackMethod是要使用的降级方法，type是选择信号量或线程池的Bulkhead

```
@Override
@Bulkhead(name="backendA", type = Type.SEMAPHORE)
public String success() {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "Hello World from backend A";
}
```

用ab测试工具模拟并发请求，50个请求48个失败，只有2次成功。

```
➜  ~ ab -c5 -n50  http://127.0.0.1:8080/backendA/failure/5
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done


Server Software:
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /backendA/failure/5
Document Length:        26 bytes

Concurrency Level:      5
Time taken for tests:   10.011 seconds
Complete requests:      50
Failed requests:        48
   (Connect: 0, Receive: 0, Length: 48, Exceptions: 0)
Non-2xx responses:      48
Total transferred:      15246 bytes
HTML transferred:       9268 bytes
Requests per second:    4.99 [#/sec] (mean)
Time per request:       1001.098 [ms] (mean)
```

当最大并发达到上限时，此时请求将抛出异常。

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/5
{"timestamp":"2020-05-21T13:57:05.137+0000","status":500,"error":"Internal Serve
r Error","message":"Bulkhead 'backendA' is full and does not permit further call
s","path":"/backendA/failure/5"}%
```

若有fallback方法，则服务降级至fallback中处理。

```
➜  ~ curl http://127.0.0.1:8080/backendA/failure/5
fallback: Recovered Throwable: Bulkhead 'backendA' is full and does not permit f
urther calls%
```

程序式调用
调用方法都类似，装饰方法之后用Try.of().recover()来执行

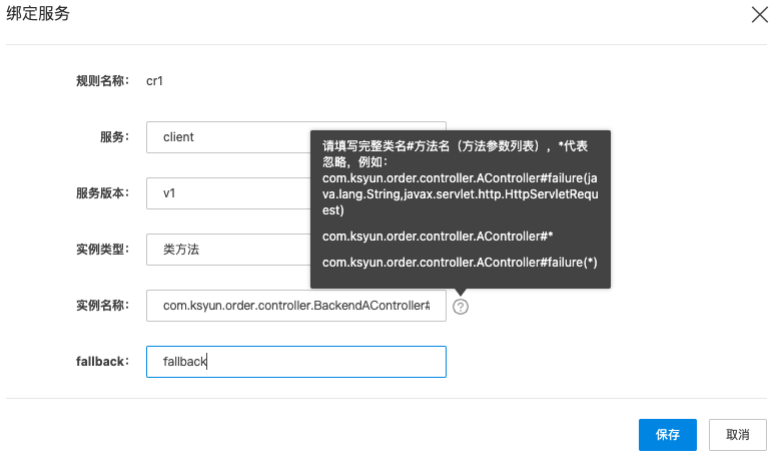
```
@Service(value = "businessAService")
public class BusinessAService implements BusinessService {
    @Autowired
    private BulkheadRegistry bulkheadRegistry;

    @Override
    public String failureNotAOP() throws Throwable {
        Bulkhead bulkhead = bulkheadRegistry.bulkhead("backendA");
        CheckedFunction0<String> bulkheadSupplier = Bulkhead.decorateCheckedSupplier(bulkhead, backendAConnector::failureNotAOP);
        Try<String> result = Try.of(bulkheadSupplier).recover(CallNotPermittedException.class, throwable -> {
            Log.info("服务降级");
            return "fallback";
        });
        return result.get();
    }
}
```

2.0
2.0版本配置实例新增类方法、URI

使用方法
启动类添加@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")开启功能支持,同时添加AspectPointcutScanPreparedEventListener监听事件。

```
@EnableKsyunFaultTolerant(basePackages = "com.ksyun.order")
public class DemoApplication extends SpringBootServletInitializer {

     public static void main(String[] args) {
         SpringApplication springApplication = new SpringApplication(DemoApplication.class);
         springApplication.addListeners(new AspectPointcutScanPreparedEventListener());
         springApplication.run(args);
     }
 }
```

AOP新增类方法以及URI拦截，配置时可将实例名写为要拦截的方法或uri。
注意：注解、类方法、uri这3种实例配置方式本质上都是作用在方法上，所以不要对同一方法配置多套实例，会导致效果成指数级。例如：方法a上配置了注解，类方法以及uri3种重试规则（重试3次），实际会导致重试3x3x3=27次。
当yaml配置key存在特殊字符时，需要用"[]"包裹。

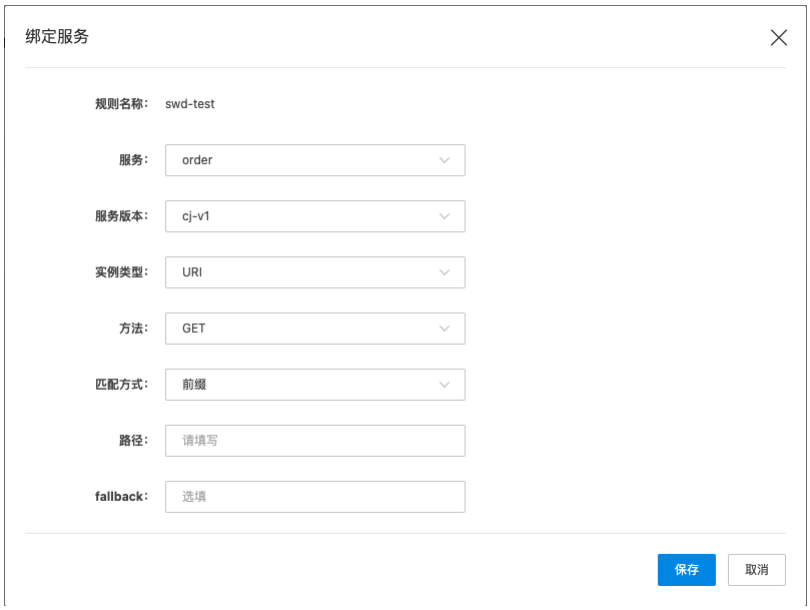


- 类名、方法实例配置

实例名称填写规则如下：

```
ClassName#MethodName(args...)

格式说明:
ClassName:完整的类名，例如：com.ksyun.order.controller.BackendAController
MethodName:方法名

例子:
# 匹配类下面所有方法（*代表所有方法）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#*]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 匹配某个名字的方法（因存在重载方法，*代表忽略参数）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(*)]":
  ...
# 匹配某个具体方法(需写上完整参数列表)：
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(java.lang.String,javax.servlet.http.HttpServletRequest)]":
  ...
```

- URI配置




```
matchType#method#uri

格式说明:
matchType:prefix、suffix、regex、match四种
method:request method,如:GET(大写)
uri:需要匹配的值

例子:
# 前缀匹配**
resilience4j.circuitbreaker.instances:
  "[prefix#GET#/failure/]":
     failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 后缀匹配
resilience4j.circuitbreaker.instances:
  "[suffix#GET#failure]":
    ...
# 完全匹配
```

```
  resilience4j.circuitbreaker.instances:
    "[match#GET#/failure/1]":
      ...
  # 正则
  resilience4j.circuitbreaker.instances:
    "[regex#GET#/.*/failu.*]":
      ...
```

- Fallback 配置

```
  resilience4j.circuitbreaker.fallback:
    "[com.ksyun.order.controller.BackendAController#*]": fallback
```

DistributedRateLimter(分布式限流)使用

启动类添加@EnableDistributedRateLimiter(basePackages = "com.ksyun.order")开启功能支持,同时添加AspectPointcutScanPreparedEventListener监听事件。

```
@EnableDistributedRateLimiter(basePackages = "com.ksyun.order")
public class DemoApplication extends SpringBootServletInitializer{

    public static void main(String[] args) {
        SpringApplication springApplication = new SpringApplication(DemoApplication.class);
        springApplication.addListeners(new AspectPointcutScanPreparedEventListener());
        springApplication.run(args);
    }
}
```

分布式限流采用redisson实现，需配置redis

```
spring:
  redis:
    host: 127.0.0.1
    port: 6379
    password: 123456
```

1. 创建配置
   在控制台“服务治理>>容错管理”页面新建”分布式限流“类型规则，如下图

各项对应可配置参数

配置参数	默认值	描述
rate	10	单位时间通过量
rate-interval	1	单位时间：权限刷新的时间
rate-interval-uint	MILLISECONDS	单位时间（单位）
timeout	5	线程等待权限的默认等待时间
timeout-unit	SECONDS	线程等待权限的默认等待时间单位

对应application.yml配置(本地开发可直接编写配置，以下例子依据此配置)

```
ksyun.cloud.distributed-ratelimiter.instances:
  backendA:
    rate: 1
    rate-interval: 10
    rate-interval-unit: SECONDS
    timeout: 15
    timeout-unit: SECONDS
```

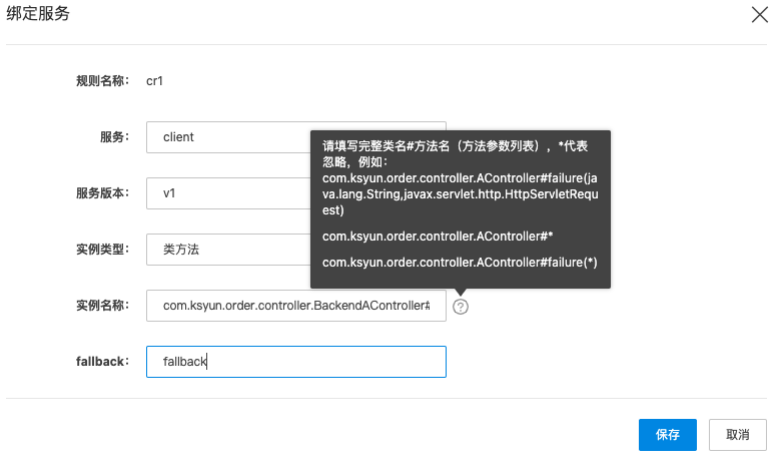
2. 调用方式
   DistributedRateLimter目前支持AOP使用注解的方式调用

注解方式调用
调用一个正常接口，不抛出异常，为了让结果明显一些，程序中sleep5秒。

```
@DistributedRateLimiter(name = "backendA")
public String success() {
  try {
    Thread.sleep(5000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  return "Hello World from backend A";
}
```

2.0
2.0版本配置实例新增类方法、URI

新增类方法以及URI拦截，配置时可将实例名写为要拦截的方法或uri。
注意：注解、类方法、uri这3种实例配置方式本质上都是作用在方法上，所以不要对同一方法配置多套实例，会导致效果成指数级。例如：方法a上配置了注解，类方法以及uri3种重试规则（重试3次），实际会导致重试3x3x3=27次。
当yaml配置key存在特殊字符时，需要用"[]"包裹。

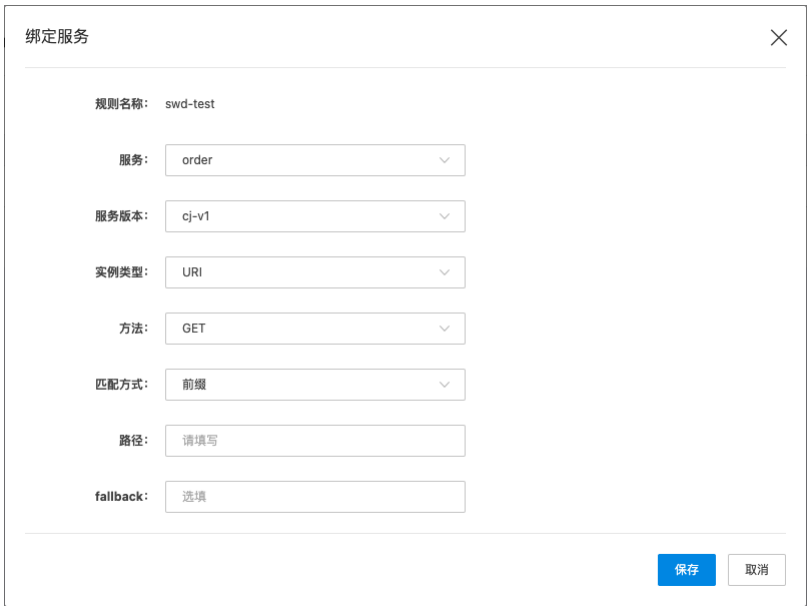


- 类名、方法实例配置

实例名称填写规则如下：

```
ClassName#MethodName(args...)

格式说明:
ClassName:完整的类名，例如：com.ksyun.order.controller.BackendAController
MethodName:方法名

例子:
# 匹配类下面所有方法（*代表所有方法）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#*]":
    failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 匹配某个名字的方法（因存在重载方法，*代表忽略参数）
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(*)]":
  ...
# 匹配某个具体方法(需写上完整参数列表)：
resilience4j.circuitbreaker.instances:
  "[com.ksyun.order.controller.BackendAController#failure(java.lang.String,javax.servlet.http.HttpServletRequest)]":
  ...
```

- URI配置




```
matchType#method#uri

格式说明:
matchType:prefix、suffix、regex、match四种
method:request method,如:GET(大写)
uri:需要匹配的值

例子:
# 前缀匹配**
resilience4j.circuitbreaker.instances:
  "[prefix#GET#/failure/]":
     failureRateThreshold: 50
    recordFailurePredicate: com.ksyun.exception.RecordFailurePredicate
    ignoreExceptions:
      - com.ksyun.order.exception.BusinessException
    recordExceptions:
      - org.springframework.web.client.HttpServerErrorException
    ringBufferSizeInClosedState: 5
    ringBufferSizeInHalfOpenState: 3
    waitDurationInOpenState: 60s
# 后缀匹配
resilience4j.circuitbreaker.instances:
  "[suffix#GET#failure]":
    ...
# 完全匹配
```

```
resilience4j.circuitbreaker.instances:
  "[match#GET#/failure/1]":
    ...
# 正则
resilience4j.circuitbreaker.instances:
  "[regex#GET#/.*/failu.*]":
    ...
```

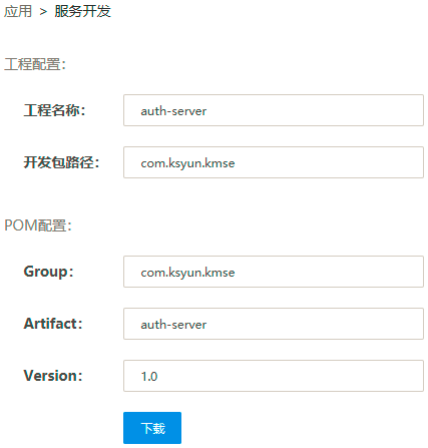
- Fallback 配置

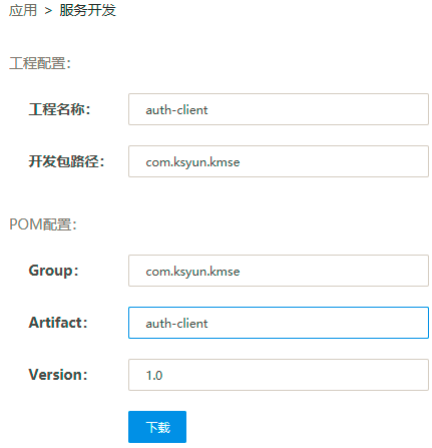
```
resilience4j.circuitbreaker.fallback:
  "[com.ksyun.order.controller.BackendAController#*]": fallback
```

服务鉴权

使用kmse实现服务的权限校验
通过一个简单的实例说明开发者如何通过kmse进行服务间的权限校验。

一、准备客户端和服务端两个demo
这里演示如何快速实践服务鉴权功能。假如现在有两个微服务 auth-client 和 auth-server，想实现 auth-client 调用 auth-server 时，auth-server 对请求做鉴权。参考服务开发文档，下载auth-server和auth-client两个demo。





查看依赖，实践服务鉴权只需要依赖以下maven组件，调用端和被调用端都只需要如下依赖。

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
        <dependency>
            <groupId>com.ksyun.kmse</groupId>
            <artifactId>spring-cloud-kmse-starter-authentication</artifactId>
            <version>${version}</version>
        </dependency>
```

因为auth-server是被调用方，所以在auth-server的bootstrap.yaml文件中写入鉴权配置，配置中两个版本属性（VERSION，subset），两个服务名属性（spring.application.name,auth-policy.http[0].route[0].destination.host）必须一致。配置的意思是创建一个名称为auth-rule-1的鉴权规则，该条规则的意思是禁止应用名称前缀为“auth-client”的请求来访问auth-server应用。

```
VERSION: v1
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            prefix: "auth-client"
      name: auth-rule-1
      route:
        - destination:
            host: auth-server
            subset: v1
      type: black-list
spring:
  application:
    name: auth-server
server:
  port: 8080
```

在auth-client的yaml中写入鉴权需要的参数，这些参数在系统中会自动注入，现在手工填写，应用名称为auth-client，版本为v1：

```
VERSION: v1
server:
  port: 8081
spring:
  application:
    name: auth-client
```

准备测试的java代码，auth-server端提供服务的controller:

```
package com.ksyun.kmse.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping({"/server"})
@RestController
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    public AccountController() {
    }

    @RequestMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用server " + id);
        return id + "";
    }
}
```

auth-client端提供的远程调用client:

```
package com.ksyun.kmse.client;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@FeignClient(name = "auth-server", url = "http://127.0.0.1:8080")//如果使用注册中心可以不使用显式的url配置
public interface OrderClient {

    @GetMapping("/server/{id}")
    String getById(@PathVariable Integer id);

}
```

auth-client端提供的测试访问入口controller：

```
package com.ksyun.kmse.controller;

import com.ksyun.kmse.client.OrderClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping({"/client"})
@RestController
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    public AccountController() {
    }

    @Autowired
    private OrderClient server;

    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用参数 " + id);
        String result = server.getById(id);
        log.info("远程调用结果 " + result);
        return result;
    }
}
```

至此两个测试应用准备完毕。

二、对服务鉴权进行测试
步骤一中的鉴权配置含义是"不允许applicationname前缀等于'auth-client'的请求访问"。 调用auth-client的测试接口 http://127.0.0.1:8081/client/1。

```
[C:\~]$ curl http://127.0.0.1:8081/client/1
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   168    0   168    0     0    168      0 --:--:-- --:--:-- --:--:--  5419
{"timestamp":"2020-05-22T07:46:25.793+0000","status":500,"error":"Internal Server Error","message":"status 403 reading OrderClient#getById(Integer)","path":"/client/1"}
[C:\~]$
```

发现auth-server返回http验证码为403。
将auth-server的配置改为如下：

```
VERSION: v1
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            prefix: "Aclient"
      name: auth-rule-1
      route:
        - destination:
            host: auth-server
            subset: v1
      type: black-list
spring:
  application:
    name: auth-server
```

这个配置的含义是"不允许applicationname前缀等于Aclient的请求访问"。 重启auth-server后，再次调用测试接口，返回http状态码为200。

```
[C:\~]$ curl -I http://127.0.0.1:8081/client/1
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     1    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8
Content-Length: 1
Date: Fri, 22 May 2020 07:49:57 GMT
```

将auth-server的配置改为如下，测试后缀拦截：

```
VERSION: v1
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            suffix: "ent"
      name: auth-rule-1
      route:
        - destination:
            host: auth-server
            subset: v1
      type: black-list
spring:
  application:
    name: auth-server
```

这个配置的含义是"不允许applicationname后缀等于ent的请求访问"。 重启auth-server后，再次调用测试接口，返回http状态码为403，请求被拦截。

```
[C:\~]$ curl http://127.0.0.1:8081/client/1
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   168    0   168    0     0    168      0 --:--:-- --:--:-- --:--:-- 10500
{"timestamp":"2020-05-22T07:53:07.078+0000","status":500,"error":"Internal Server Error","message":"status 403 reading OrderClient#getById(Integer)","path":"/client/1"}
[C:\~]$
```

依次类推还有如下的请求场景：

```
#匹配请求来源url
auth-policy:
  http:
    - match:
      - APIPath: "/auth-server/1"
#匹配请求来源ip
auth-policy:
  http:
    - match:
      - IP: "127.0.0.1"
#匹配请求http方法
auth-policy:
  http:
    - match:
      - Method: "GET"
#匹配应用版本
auth-policy:
  http:
    - match:
      - applicationVersion: "v1"

#前缀匹配
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            prefix: "a"
#后缀匹配
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            suffix: "b"

#精准匹配
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            exact: "c"

#正则匹配，例如正整数
auth-policy:
  http:
    - match:
      - applicationName:
          endUser:
            regular: "[1-9]\d*"
```

负载均衡

准备工作
开始实践负载均衡功能前，请确保已完成了 SDK 下载。
负载均衡依赖consul作为注册中心和配置中心，需要本地安装consul，consul下载和安装参考官网https://www.consul.io。

一、创建服务端

1. 创建lb-server项目 ，从微服务平台下载一个项目，命名为lb-server。

2.依赖项
修改pom.xml中dependency依赖如下：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-consul-discovery</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
```

3.修改配置
在bootstrap.yml中添加如下配置。

```
server:
  port: 8080
spring:
  application:
    name: lb-server
  cloud:
    host: http://127.0.0.1
    port: 8500
    discovery:
      healthCheckPath: /actuator/health
      healthCheckInterval: 15s
      register: true
      service-name: lb-server
      enabled: true
      instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
      tag: version=v0
```

4.代码调整
启动类添加注解。

```
@SpringBootApplication
@EnableDiscoveryClient
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

新增接口供客户端调用。

```
package com.demo.x.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping({"/account"})
@RestController
public class AccountController {

    private static final Logger log = LoggerFactory.getLogger(AccountController.class);

    @Value("${spring.cloud.discovery.instanceId}")
    private String instanceId;

    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用account " + id);
        return id + ":"+instanceId;
    }
}
```

二、创建客户端
1.创建lb-client项目 从微服务平台下载一个项目，命名为lb-client。

2.依赖项
修改pom.xml中dependency依赖如下：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.ksyun.kmse</groupId>
            <artifactId>spring-cloud-kmse-loadbalancer-core</artifactId>
            <version>${version}</version>
        </dependency>
        <dependency>
```

```
            <groupId>com.ksyun.kmse</groupId>
            <artifactId>spring-cloud-kmse-starter-loadbalancer</artifactId>
            <version>${version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-consul-discovery</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
```

3.修改配置
在bootstrap.yml中添加如下配置。

```
server:
  port: 8081
spring:
  application:
    name: lb-client
  cloud:
    host: http://127.0.0.1
    port: 8500
    discovery:
      healthCheckPath: /actuator/health
      healthCheckInterval: 15s
      register: true
      service-name: lb-client
      enabled: true
      instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:${random.value}}}
      version=v0
ksyun:
  cloud
    loadbalancer:
      destinationRule:
      - host: lb-server
        subsets:
        - labels:
            version: v0
          name: v0
          trafficPolicy:
            loadBalancer:
              simple: rr
        trafficPolicy:
          loadBalancer:
            simple: rr
```

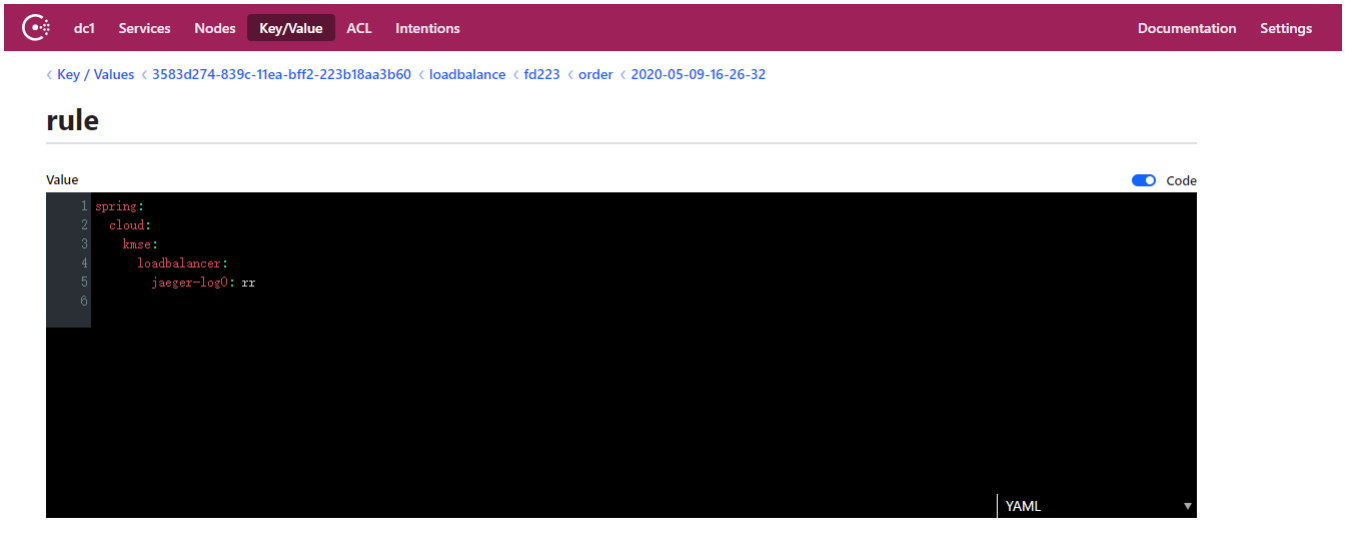
4.代码调整
启动类添加注解。

```
@SpringBootApplication
@EnableFeignClients
@EnableDiscoveryClient
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

新增接口调用服务端

```
package com.demo.x.client;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@FeignClient("lb-server")
public interface OrderClient {

    @GetMapping({"/account/{id}"})
    String getById(@PathVariable Integer id);

}
```

三、服务注册中心下发配置
在consul的kv中新建名为/lb/lb-client/v0/data的key,并添加如下配置，lb-server为服务名称，v0为服务的版本，rr为轮询策略，目前共支持三种负载均衡策略， 分别是随机(random)、轮询(rr)、响应时间权值(wr)，也可以选择其他策略。



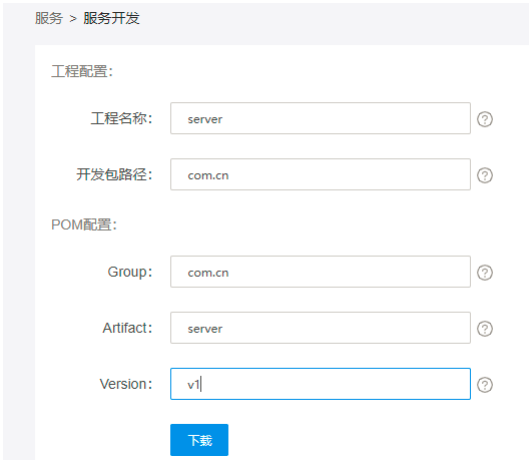

```
ksyun:
  cloud:
    loadbalancer:
      destinationRule:
      - host: lb-server
        subsets:
        - labels:
            version: v0
          name: v0
```

```
      trafficPolicy:
        loadBalancer:
          simple: rr
    trafficPolicy:
      loadBalancer:
        simple: rr
```

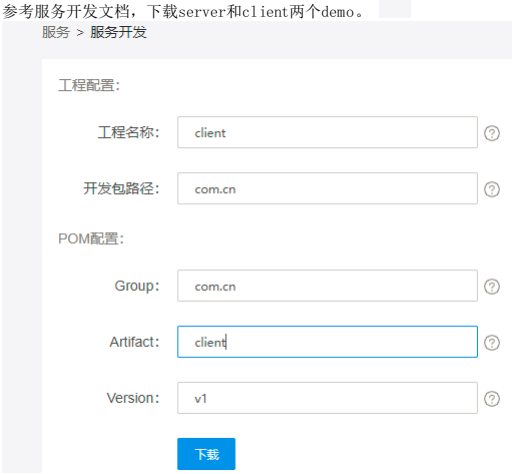
灰度发布

使用kmse实现服务间的灰度发布功能
通过一个简单的实例说明开发者如何通过kmse进行服务间的灰度发布功能。kmse的灰度发布功能可分为基于流量灰度和全链路灰度。

一、准备客户端和服务端两个demo
这里演示如何快速实践服务路由功能。假如现在有两个微服务 client 和 server，想实现 client 调用 server 时，通过灰度规则对服务间的请求流量做定向路由。



参考服务开发文档，下载server和client两个demo。



查看依赖，实践服务鉴权只需要依赖以下maven组件，调用端和被调用端都只需要如下依赖。

```
<dependency>
    <groupId>com.ksyun.kmse</groupId>
    <artifactId>spring-cloud-kmse-starter-route</artifactId>
    <version>1.0-release</version>
</dependency>
```

准备测试的java代码，因为需要灰度功能，所以我们在controller加入特殊的逻辑用来体现灰度功能，server端提供服务的controller:

```
package com.cn.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.PostConstruct;

@RequestMapping({"/server"})
@RestController
@Configuration
public class ServerController {

    private static final Logger log = LoggerFactory.getLogger(ServerController.class);

    public ServerController() {
    }

    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        System.out.println(appName);
        System.out.println(instance);
        return version + "#" + instance;
    }

    @Value("${spring.cloud.consul.discovery.tags:version=v1}")
    private String tag;

    private String version;

    @Value("${spring.cloud.consul.discovery.instanceId:1}")
    private String instance;

    @GetMapping
    public String routeServer() {
        System.out.println(appName);
        System.out.println(instance);
        return version + "#" + instance;
```

```
    }

    @PostConstruct
    private void parseVersion() {
        System.out.println(String.format("configName={%s}", configName));
        String[] split = tag.split(",");
        for (String s : split) {
            if (s.startsWith("version=")) {
                version = s.substring(s.indexOf("version=") + 8);
                break;
            }
        }
    }
}
```

client端提供的远程调用client：

```
package com.cn.controller;

import com.cn.client.Client;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

@RequestMapping({"/client"})
@RestController
public class ClientController {

    private static final Logger log = LoggerFactory.getLogger(ClientController.class);

    @Autowired
    private Client client;

    public ClientController() {
    }

    @GetMapping({"/{id}"})
    public String account(@PathVariable("id") Integer id) {
        log.info("调用client " + id);
        Map<String, Integer> mapVersion = new HashMap<>(), mapInstance = new HashMap<>();

        StringBuilder lbResult = new StringBuilder();
        lbResult.append("\n");
        for (int i = 0; i < id; i++) {
            String result = client.getById(i);
            String[] split = result.split("#");
            String version = split[0];
            String instance = split[1];

            if (!mapVersion.containsKey(version)) {             mapVersion.put(version, 1);
            } else {
                mapVersion.put(version, mapVersion.get(version) + 1);
            }
            if (!mapInstance.containsKey(instance)) {
                mapInstance.put(instance, 1);
            } else {
                mapInstance.put(instance, mapInstance.get(instance) + 1);
            }
            lbResult.append(instance).append("\n");
        }
        log.info("版本统计 = {}", mapVersion.toString());
        log.info("实例统计 = {}", mapInstance.toString());
        log.info("负载均衡顺序 = {}", lbResult.toString());
        return mapVersion.toString() + "     " + mapInstance.toString() + " " + lbResult.toString();
    }
}
```

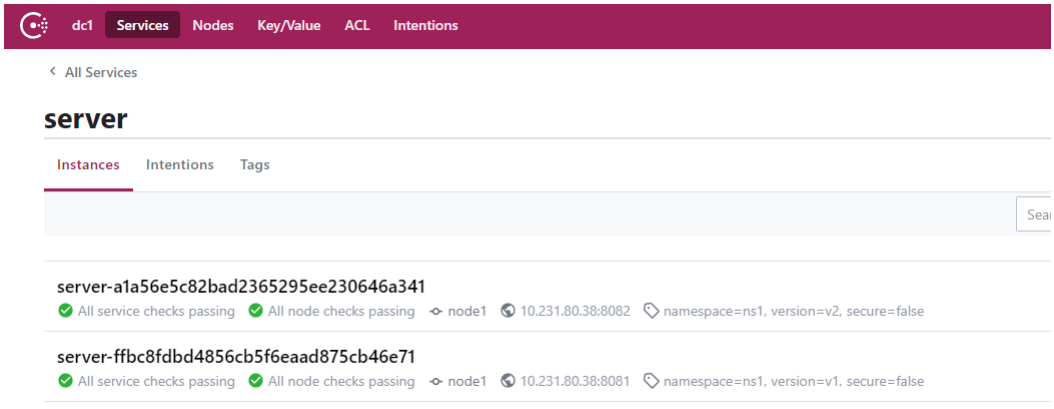
至此两个测试应用的java代码准备完毕。

二、对基于流量的灰度进行测试
将server服务用maven（mvn package）命令打包，然后在本地启动consul作为注册中心。

使用如下两条命令分别运行server的v1和v2版本：

```
java -Dspring.cloud.consul.discovery.tags=namespace=ns1,version=v1  -Dserver.port=8081 -jar server.jar
java -Dspring.cloud.consul.discovery.tags=namespace=ns1,version=v2  -Dserver.port=8082 -jar server.jar
```

运行成功后，在consul-ui上看到如下的注册信息，可以看到分别有version不同的tag：



因为client是调用方，所以在client的resources文件夹中新增application.yaml文件，写入灰度配置。配置的意思是该应用访问server的流量，90%的流量发送到v2版本，10%的流量发送到v1版本。

```
ksyun:
  cloud:
    route:
      rule:
        client:
          virtualService:
            - route:
                - destination:
                    host: server
                    subset: v1
                  weight: 10
                - destination:
                    host: server
                    subset: v2
```

```
            weight: 90
```

因为本地调试的特殊性，所以还需要加入一个命名空间的参数，然后运行client，访问测试接口：

```
http://127.0.0.1:8080/client/100
```

从请求返回体和日志都可以看到，v1、v2的流量确实是按照规则来分配的。

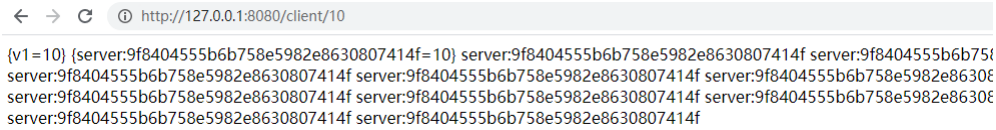
三、对全链路的灰度进行测试
1. 简单演示
这里采用uri来进行演示

```
ksyun:
  cloud:
    route:
      rule:
        mytest:
          virtualService:
          - match:
            - uri:
                endUser:
                  prefix: /client/10
            route:
            - destination:
                host: server
                subset: v1
```

这个配置的含义是"uri前缀等于/client/10的请求流量全部路由到v1版本"。 重启client后，再次调用测试接口。我们期望形如"/client/10"的请求全部打到v1版本。

```
http://127.0.0.1:8080/client/10
```

从请求返回体和日志都可以看到，流量全部路由到了v1版本，这就符合了我们的预期。



这里我们继续刚才的配置测试后缀拦截，将client的配置改为如下，这种类型的配置会将形如以下的请求流量都达到server的v2。

- /a/client/10
- /b/client/10

```
ksyun:
  cloud:
    route:
      rule:
        mytest:
          virtualService:
            - match:
                - uri:
                    endUser:
                      suffix: /client/10
              route:
                - destination:
                    host: server
                    subset: v2
```

2. 更多的场景展示
由于参数类型，匹配规则，逻辑关系三者都能组合出众多的规则，所以下文例举了一些请求场景来作为参考，可以根据实际业务中的场景来做自定义扩展。

2.1 header前缀匹配
如果场景中有headers: name前缀匹配zhangsan 的请求，那么这种类型的配置会将形如以下的请求流量都路由到server的v1。

- curl --location --request GET '127.0.0.1/aa' --header 'name: zhangsan1'
- curl --location --request GET '127.0.0.1/aa/bb' --header 'name: zhangsanaaa'

```
ksyun:
  cloud:
    route:
      rule:
        mytest:
          virtualService:
            - match:
                - headers:
                    endUser:
                      prefix: "zhangsan"
                      param: name
              route:
                - destination:
                    host: server
                    subset: v1
```

2.2 uri正则匹配
如果场景中有queryParams:type正则匹配-?[1-9]*)$ 的请求，那么这种类型的配置会将形如以下的请求流量都路由到server的v1。

- curl --location --request GET '127.0.0.1/a?type=123'
- curl --location --request GET '127.0.0.1/b/c?type=456&name=zhangsan'

```
ksyun:
  cloud:
    route:
      rule:
        test:
          virtualService:
          - match:
            - queryParams:
                endUser:
                  regular: ^(-?[1-9]\d*)$
                  param: type
            route:
            - destination:
                host: server
                subset: v1
```

2.3 多规则匹配,"且"逻辑关系
如果场景中有headers:name=zhangsan, 且headers:age=20的请求，那么这种类型的配置会将形如以下的请求流量都路由到server的v1。

- curl --location --request GET '127.0.0.1/aa' --header 'name: zhangsan' --header 'age: 20'
- curl --location --request GET '127.0.0.1/aa/bb' --header 'name: zhangsan' --header 'age: 20' --header 'role: student'

```
ksyun:
  cloud:
    route:
      rule:
        test:
          virtualService:
          - match:
            - headers:
                endUser:
                  exact: "20"
                  param: age
            - headers:
```

```
              endUser:
                exact: zhangsan
                param: name
          route:
          - destination:
              host: server
              subset: v1
```

2.4 多规则匹配,"或"逻辑关系
如果场景中有headers:age前缀匹配20, 或headers:name后缀匹配zhangsan 的请求，那么这种类型的配置会将形如以下的请求流量都路由到server的v1中。

- curl --location --request GET '127.0.0.1/aa' --header 'age: 200'
- curl --location --request GET '127.0.0.1/aa/bb' --header 'name: zhangsanl' --header 'role: student'
- curl --location --request GET '127.0.0.1/aa/cc' --header 'name: zhangsan' --header 'age: 20'

```
ksyun:
  cloud:
    route:
      rule:
        test:
          virtualService:
          - match:
            - headers:
                endUser:
                  prefix: "20"
                  param: age
            route:
            - destination:
                host: server
                subset: v1
          - match:
            - headers:
                endUser:
                  suffix: zhangsan
                  param: name
            route:
            - destination:
                host: server
                subset: v1
```

调用链

操作场景
KMSE 微服务管理平台提供调用链查询功能，主要包括链路追踪、动态拓扑两部分。

链路追踪用来查询和定位具体某一次调用的情况，提供了 Spring Cloud 全链路跟踪功能，包括服务监控、mysql、redis、mongo、kafka、rabbitmq。使用者可以通过具体的服务、接口定位、IP 等查询具体的调用过程，包括调用过程所需要的时间和运行情况。还可以根据 TraceID 查询调用链的详细信息。调用链详情是为了定位在分布式链路调用过程中每个环节的耗时、日志和异常。

动态拓扑包含了查询服务之间相互依赖调用的拓扑关系，查询特定集群特定命名空间下服务之间调用的统计结果等功能。

前提条件
向工程中添加依赖，在 pom.xml 中添加以下依赖：

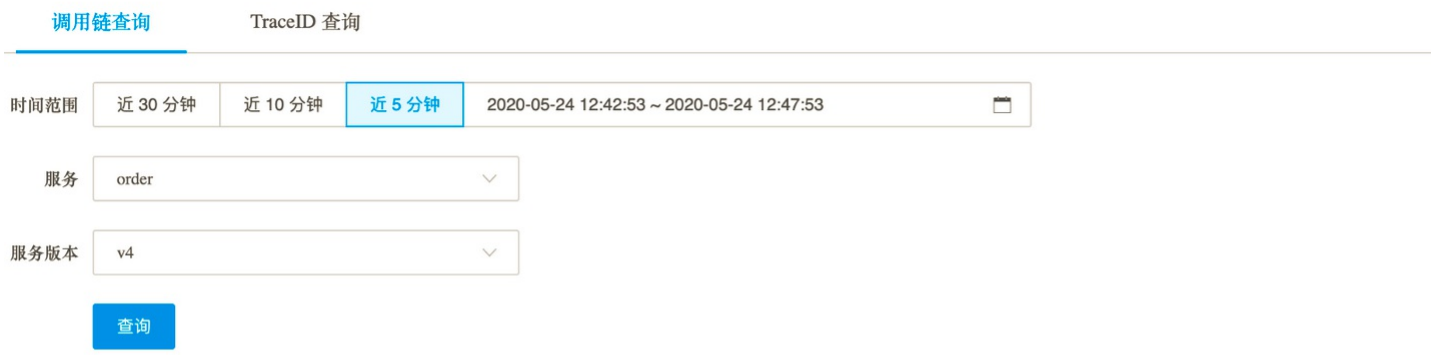
```
<dependency>
  <groupId>com.ksyun.kmse</groupId>
  <artifactId>spring-cloud-kmse-starter-jaeger</artifactId>
  <version>${version}</version>
</dependency>
```

bootstrap.yaml 中添加链路相关配置（本地开发自测配置，线上无需配置）：

```
opentracing:
  jaeger:
    log-spans: true
    # 是否开启 Zipkin 兼容模式
    enable-b3-propagation: true
    # 概率采样
    # probabilistic-sampler:
    #    sampling-rate: 0.1
    #常量采样
    const-sampler:
      decision: true
    # trace 上报接口
    http-sender:
      url: http://0.0.0.0:14268/api/traces
    # 启用 trace
    enabled: true
    # trace 上报服务的版本
    tags.svc-version: v4
```

调用链查询

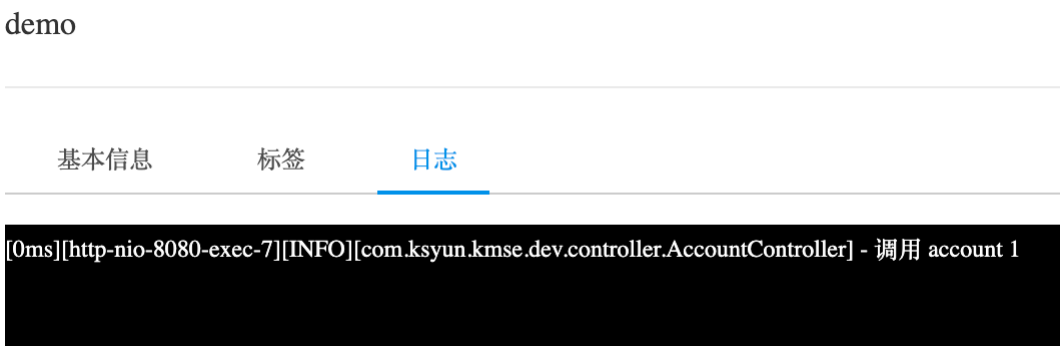
1. 登录微服务管理平台，在左侧导航栏中选择【调用链-链路追踪】，切换至【调用链查询】标签页。
2. 在调用链查询中，设置查询条件，单击【查询】。
   - **时间范围**：支持特定和自定义时间范围选择。特定时间范围包括：5分钟前、10分钟前和30分钟前。
   - **服务**：单击下拉框，在下拉框中选择服务。
   - **服务版本**：单击下拉框，在下拉框中选择服务版本。



Trace ID	产生日期	服务	耗时（毫秒）
1e7e3efecc65a8a5	2020-05-24 12:45:45	order	1.61
4a5b1323328d3345	2020-05-24 12:45:44	order	1.59
fd0ff5c119ceba23	2020-05-24 12:45:42	order	53.73
95031d1c658411e1	2020-05-24 12:45:37	order	9.94

3. 根据查询结果，可以单击【Trace ID】进入具体慢业务或出错业务，查看调用链详情。

4. 点击时间轴可查看 Span 详情，包括基本信息、标签、日志。

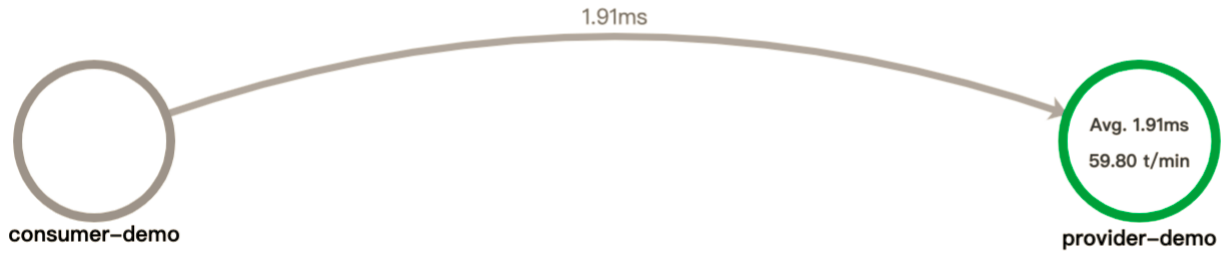


动态拓扑图查询

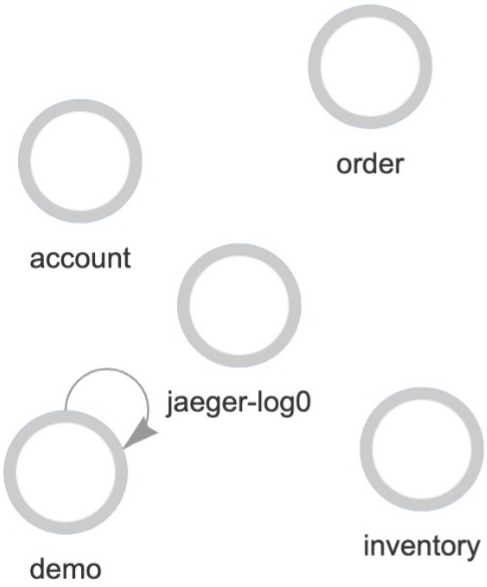
1. 登录微服务管理平台，在左侧导航栏中选择【调用链-动态拓扑】。
2. 在页面顶部数据中心位置，选择需要查看的服务所属命名空间。
3. 下方按钮选择需要依赖拓扑的时间，近30分钟、近10分钟、近5分钟以及选择特定时间段（特定时间段的时间跨度最长为3天）。
4. 选择之后将在下方空白处出现对应的服务依赖调用关系。

服务依赖拓扑图仅能查询仅1个月的数据

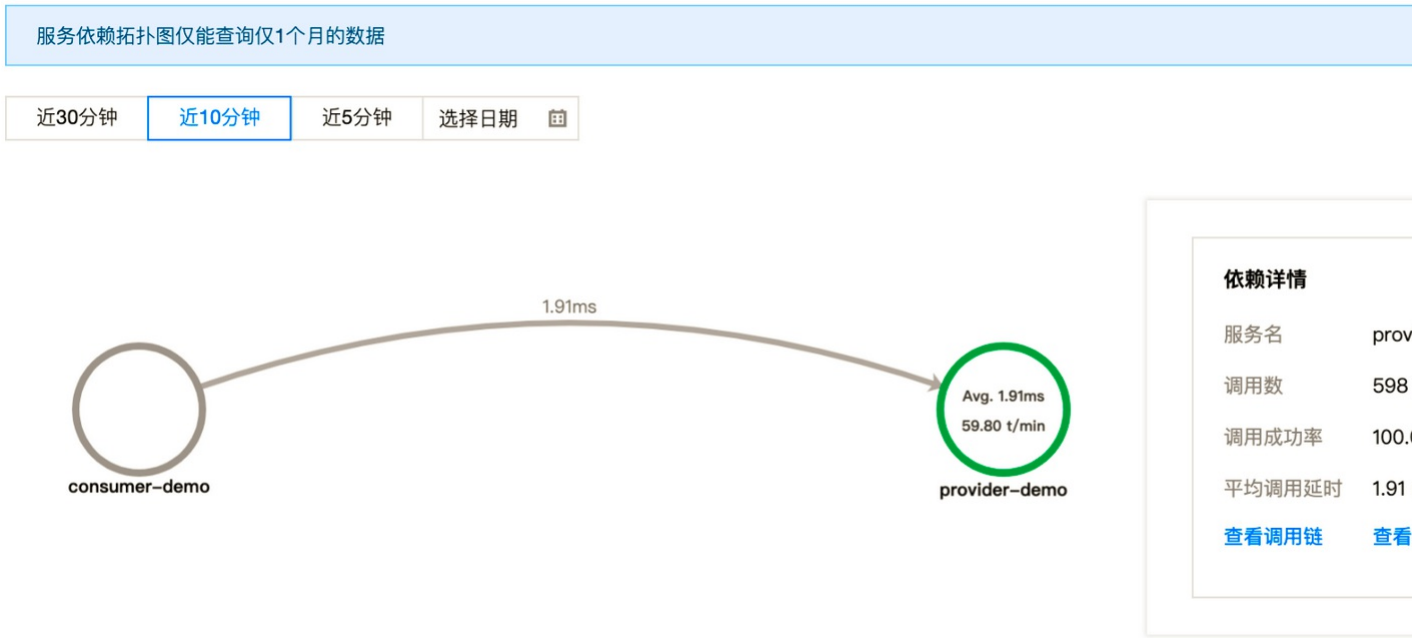
近30分钟 | 近10分钟 | 近5分钟 | 选择日期



查询依赖详情
鼠标放置到图上特定位置可以显示调用依赖详情。



单击服务圈内（白色底），可以展示被调用次数，单击弹出框上的“查看调用链”可以进入到调用链查询界面。



附：微服务对下游组件访问的链路支持
微服务对 JDBC、 Redis、Memcached、MongoDB、消息队列 RabbitMQ、Kafka 等的访问操作会产生跟踪日志， KMSE 会对该日志进行采集、分析、统计，这些组件的调用会展现在 KMSE 平台的链路追踪中。

JDBC 链路使用说明
支持各种实现 JDBC 规范的 MySQL / SQL Server / Oracle 等关系型数据库驱动器和各类连接池（如 Tomcat-JDBC、DBCP、Hikari、Druid），在使用时需引入 SpringBoot 相关依赖和 相关的 db 驱动依赖即可，如：

```
<!-- JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- mysql -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

引入依赖后，根据需要添加相关数据库连接池依赖或直接使用 SpringBoot 默认选项。
配置启用 JDBC tracing：

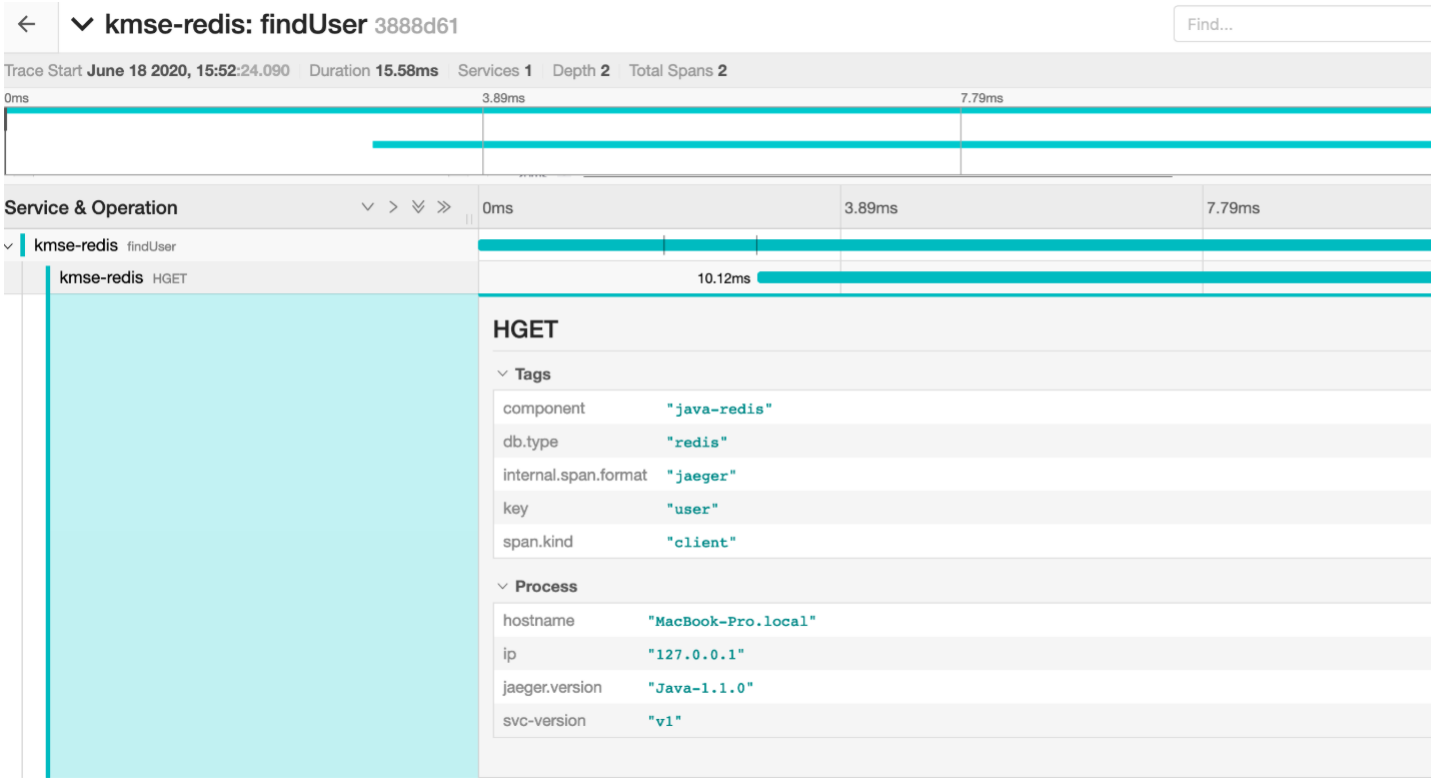
Redis 链路使用说明
考虑到 Redis 库的多样性，以及 spring-data-redis 库的易用性，目前只对spring-boot-starter-data-redis进行支持，在引用 spring-boot-starter-data-redis 时不要指定版本，只需要整个工程依赖 parent pom 即可：

```
<!-- Redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
 </dependency>
```

spring-boot-starter-data-redis的版本为 parent pom 文件管理的 Redis Starter 的版本。在代码中具体使用时，引入 RedisTemplate，然后使用其方法即可。不建议直接引用 Jedis 和 Lettuce 相关的依赖，spring-boot-starter-data-redis 会自动引用相关的依赖，并做适配。
配置启用 Redis tracing：

```
opentracing.spring.cloud.redis.enabled=true
```

如果通过其他方式引入 Redis 客户端（例如直接 new Jedis），则将无法在 KMSE 的链路中查看到相应的信息。



Memcached 链路使用说明
考虑到 Spring Data 暂未提供 Memcached 操作依赖库，我们可以使用 spymemcached 第三方依赖，示例如下：

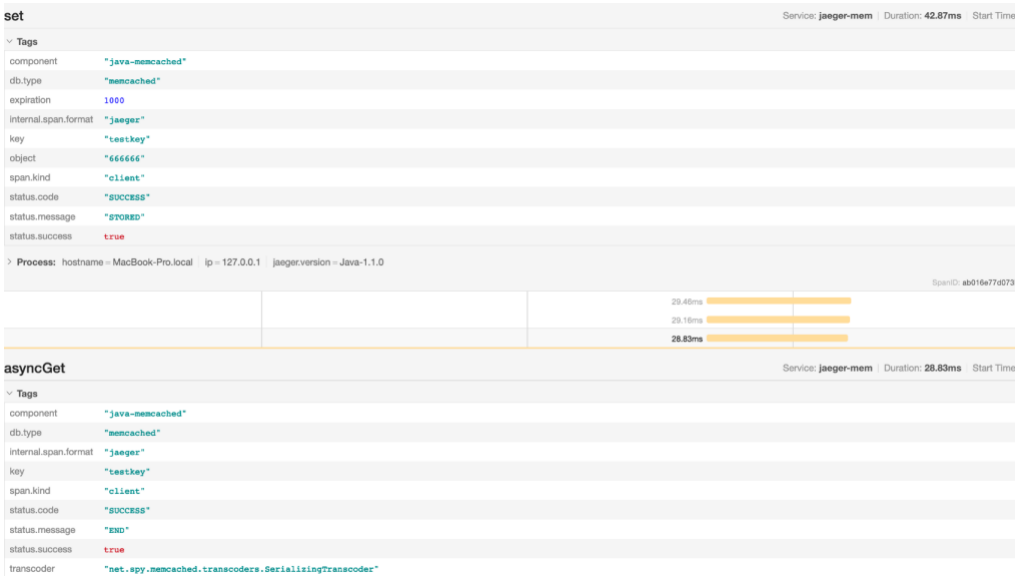
```
<!-- Memcached -->
<dependency>
    <groupId>net.spy</groupId>
    <artifactId>spymemcached</artifactId>
    <version>2.12.2</version>
</dependency>
```

我们需要创建一个 Tracing 增强的 Memcached Client：

```
MemcachedClient memcachedClient = new TracingMemcachedClient(tracer, false,
                new InetSocketAddress("10.69.70.68", 9011));
memcachedClient.set("testkey", 1000, "888888");
```

此处的 tracer 需要我们将 io.opentracing.Tracer 实例化后进行依赖注入，如：

```
@ConditionalOnProperty(value = "opentracing.jaeger.enabled", havingValue = "true", matchIfMissing = true)
@Configuration
public class MyTracerConfiguration {

    @Bean
    public io.opentracing.Tracer jaegerTracer() {
        io.jaegertracing.Configuration config = new io.jaegertracing.Configuration("jaeger-mem");
        io.jaegertracing.Configuration.SenderConfiguration sender = new io.jaegertracing.Configuration.SenderConfiguration();
        sender.withEndpoint("http://jaeger-collector.kmse-system:14268/api/traces");
        config.withSampler(new io.jaegertracing.Configuration.SamplerConfiguration().withType("const").withParam(1));
        config.withReporter(new io.jaegertracing.Configuration.ReporterConfiguration().withLogSpans(true).withSender(sender).withMaxQueueSize(10000));
        return config.getTracer();
    }
}
```



MongoDB 链路使用说明
考虑到 spring-data-mongodb 库的易用性，目前只对spring-boot-starter-data-mongodb进行支持，在引用 spring-boot-starter-data-mongodb 时不要指定版本，只需要整个工程依赖 parent pom 即可，示例如下：

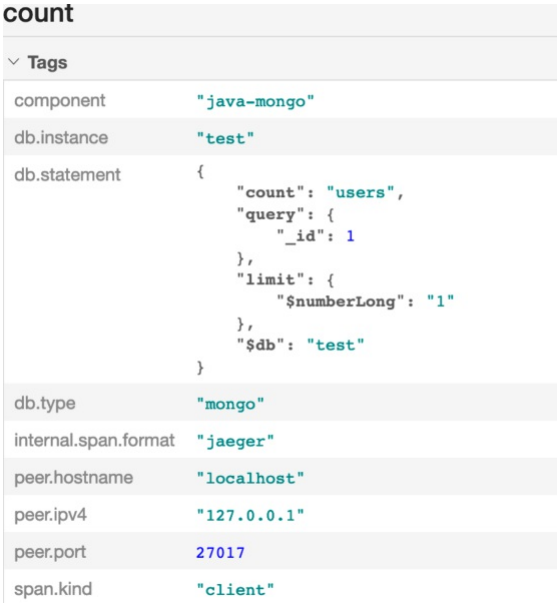
```
<!-- MongoDB -->
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

spring-boot-starter-data-mongodb的版本即是 parent pom 文件管理的 mongoldb starter 的版本。在代码中具体使用时，引入MongoTemplate，然后使用其方法即可。

配置启用 MongoDB tracing：

```
opentracing.spring.cloud.mongo.enabled=true
```

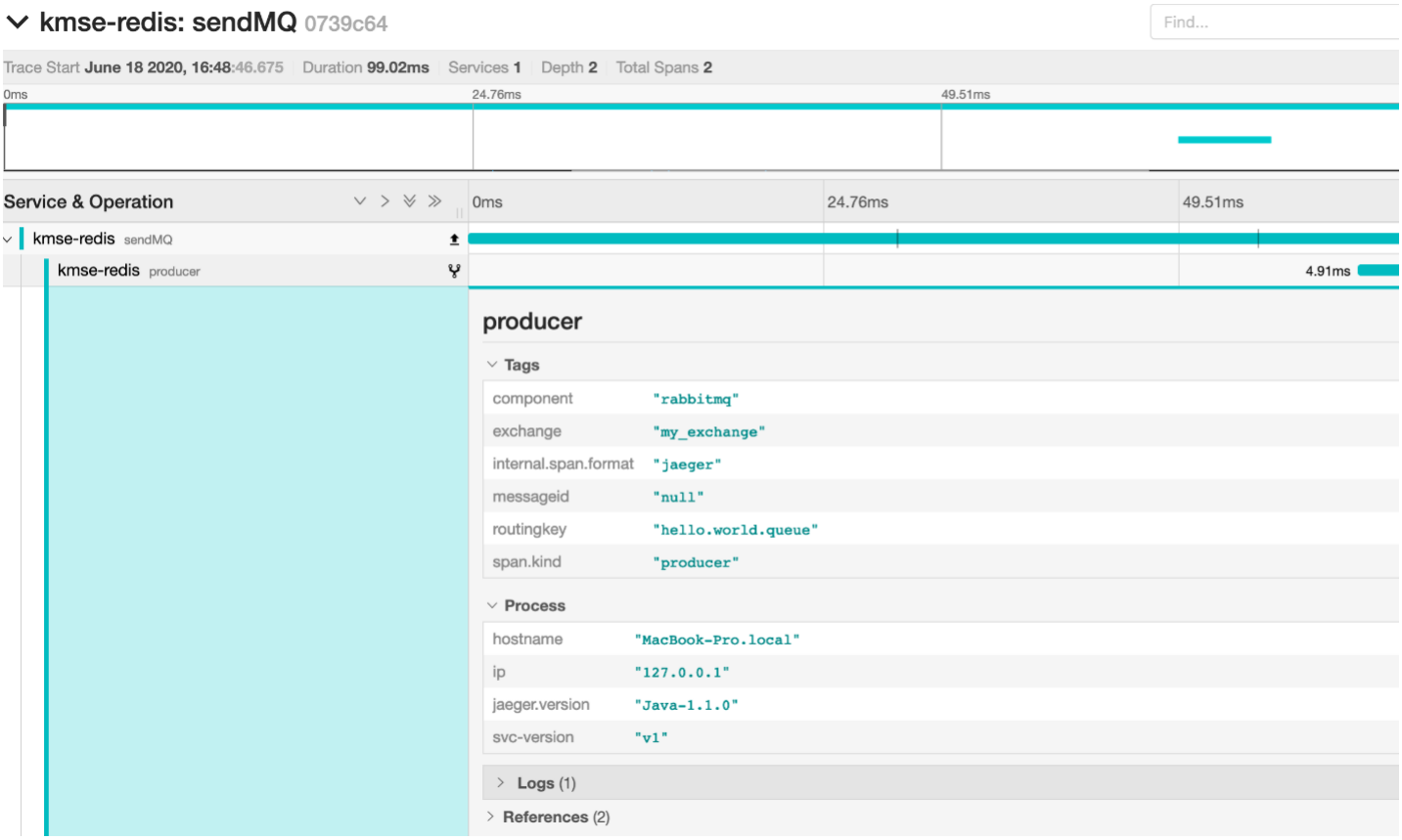


消息队列 RabbitMQ 链路使用说明
RabbitMQ 组件目前通过 Spring Cloud Stream 方式接入Spring Cloud 体系，对于 RabbitMQ 组件的全链路追踪目前基于spring-cloud-stream-rabbit扩展实现，使用时需在上下游服务中添加spring-cloud-stream-rabbit依赖并按照规范进行 RabbitMQ 配置。

```
<!-- Stream -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<!-- RabbitMQ -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

配置启用 JMS tracing：

```
opentracing.spring.cloud.jms.enabled=true
```



消息队列 Kafaka 链路使用说明
Kafaka 组件目前通过 Spring Cloud Stream 方式接入Spring Cloud 体系，对于 Kafaka 组件的全链路追踪目前基于spring-cloud-stream-kafka 扩展实现，使用时需在上下游服务中添加spring-cloud-stream-rabbit依赖并按照规范进行 Kafaka 配置。

```
<!-- Stream -->
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<!-- RabbitMQ -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

配置启用 Kafaka tracing：

```
opentracing.spring.cloud.kafka.enabled=true
```

Kafka 配置参考：

```
spring:
  cloud:
    stream:
      kafka:
        binder:
          brokers: localhost:9092
          auto-create-topics: true
          configuration:
            auto.offset.reset: latest
      bindings:
        greetings-in:
          destination: greetings-sample
          group: greetings-in-group
          contentType: application/json
        greetings-out:
          destination: greetings-sample
          contentType: application/json
```

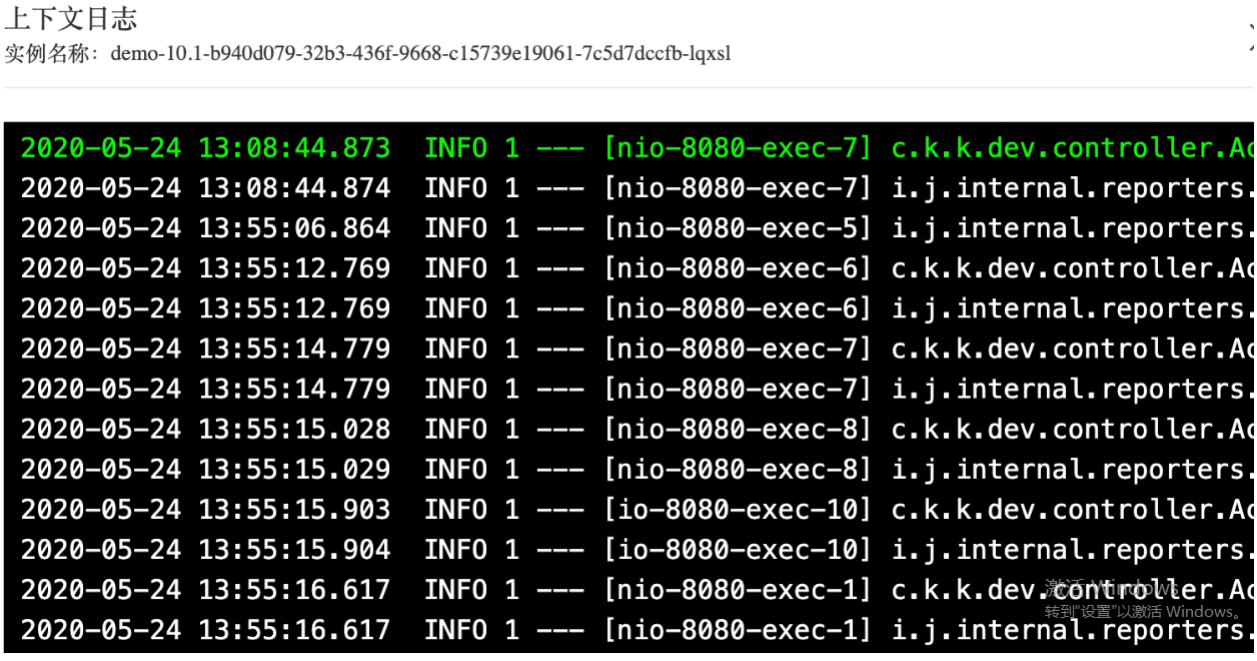
日志

操作场景
KMSE 微服务管理平台提供日志查询功能，主要包括服务日志、实时日志两部分。为用户提供一站式日志服务，从日志采集、日志存储到日志内容搜索，帮助用户轻松定位业务问题。用户可以通过 KMSE 微服务控制台查看服务日志、实时日志，并根据关键词来检索日志。

前提条件
使用 SpringBoot 标准输出或使用 logback、log4j、log4j2 等日志框架输出到控制台 stdout（日志格式无要求），不支持输出到自定义文件。

服务日志

1. 登录微服务管理平台，在左侧导航栏中选择【日志-服务日志】。
2. 在服务日志查询中，设置查询条件，单击【查询】。
   - **时间范围**：支持特定和自定义时间范围选择。特定时间范围包括：近 1 小时、近 6 小时、近 12 小时、近 1 天和近 3 天。
   - **服务**：单击下拉框，在下拉框中选择服务。
   - **服务版本**：单击下拉框，在下拉框中选择服务版本。
   - **关键字**：在文本框中填写查询关键字，可按关键字进行过滤。
3. 查看上下文日志，点击日志列表中任意一行，即可显示上下文日志。



实时日志

1. 登录微服务管理平台，在左侧导航栏中选择【日志-实时日志】。
2. 在实时日志服务列表中，选择服务对应版本，单击“日志”图标，会弹出日志侧边栏，选择对应 pod 实例的 对应容器，即可查看其实时日志。

观测

操作场景 KMSE 微服务管理平台提供监控查询功能，主要包括服务统计、服务观测、容错观测、配置下发观测四部分。

- 服务统计，支持以被调视角展示服务指标的统计信息。用户可以通过统计信息了解服务指标的变化情况。
- 服务观测，支持以图表形式展示 Java 服务的观测指标信息。
- 容错观测，支持以图表形式展示服务容错的指标统计信息。
- 配置下发观测，支持查询某服务版本实例的配置下发状态，了解应用配置下发情况。

前提条件 向工程中添加依赖，在 pom.xml 中添加以下依赖：

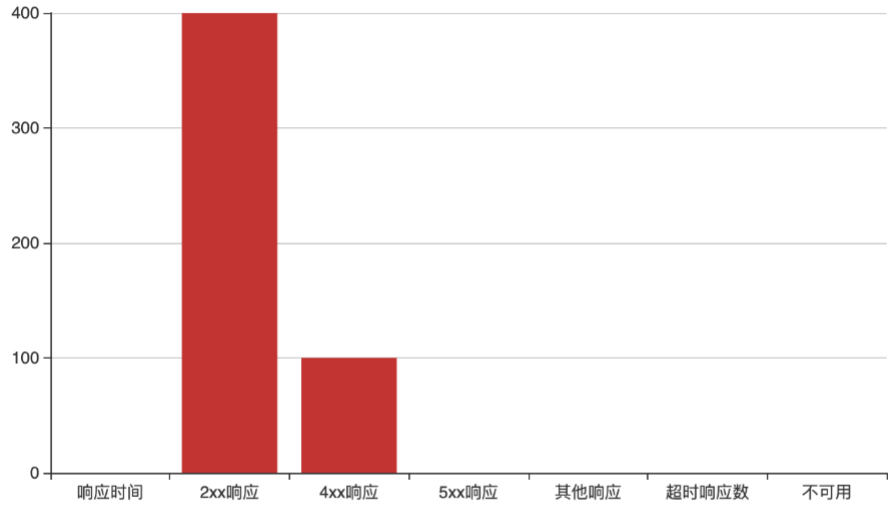
```
<dependency>
  <groupId>com.ksyun.kmse</groupId>
  <artifactId>spring-cloud-kmse-starter-prometheus</artifactId>
  <version>${version}</version>
</dependency>
```

服务统计 KMSE 支持以被调视角展示服务指标的统计信息。用户可以通过统计信息了解服务指标的变化情况。

1. 登录微服务管理平台，在左侧导航栏中选择 **观测** >**服务统计**。
2. 在服务统计查询中，设置查询条件，单击**查询**。
   - **服务**：单击下拉框，在下拉框中选择服务。
   - **服务版本**：单击下拉框，在下拉框中选择服务版本。
   - **日期**：日期可选择最近三天。
3. 单击操作列的**查看日环比/周同比**，可以查看各指标日环比/周同比情况。
   - 日环比表示查询日 T 和之前一天 T - 1 的指标数据对比。

日环比 / 周同比

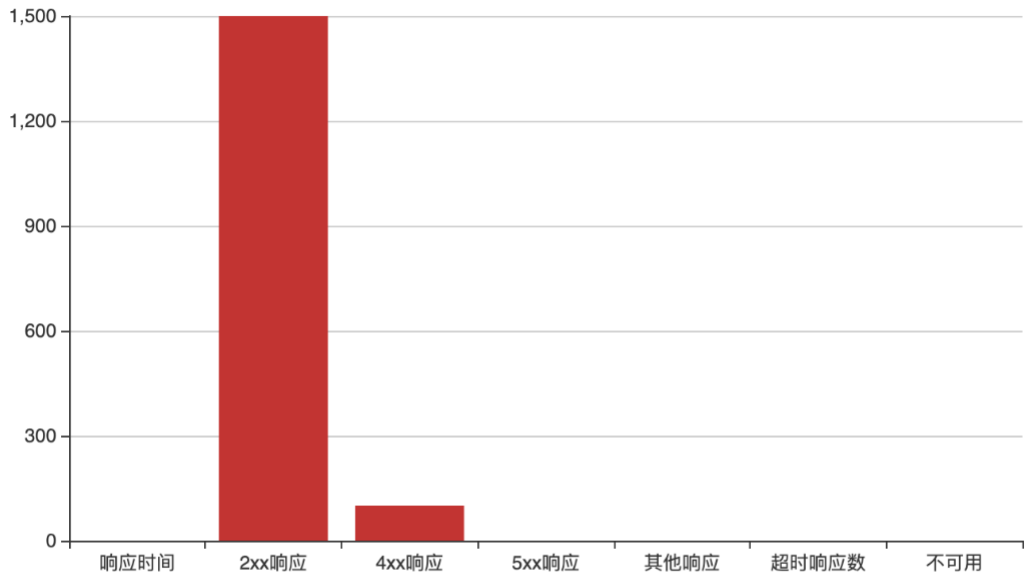
日环比 (%)



日期	响应时间	2xx 响应	4xx 响应	5xx 响应	其他响应	超时响应数	不可用
2020-05-23	0.547ms	3	0	0	0	0	0
2020-05-24	0.547ms	15	1	0	0	0	0
日环比	0.00%	400.00%	100.00%	0.00%	0.00%	0.00%	0.00%

- 周同比表示查询日 T 和查询日前7天 T - 7 的指标数据对比

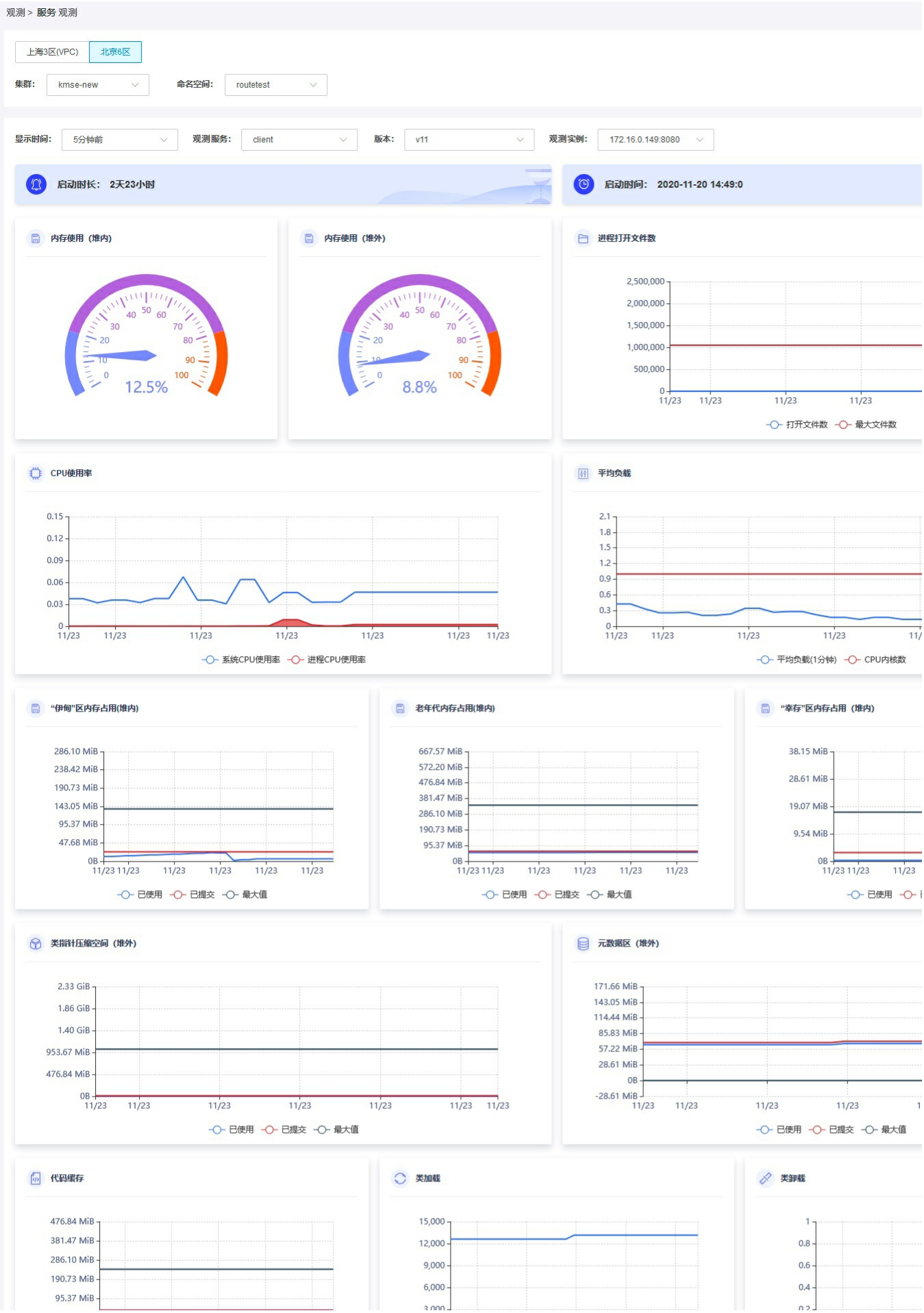
周同比 (%)



日期	响应时间	2xx 响应	4xx 响应	5xx 响应	其他响应	超时响应数	不可用
2020-05-18	0.547ms	0	0	0	0	0	0
2020-05-24	0.547ms	15	1	0	0	0	0
周同比	0.00%	1500.00%	100.00%	0.00%	0.00%	0.00%	0.00%

服务观测 KMSE 支持以图表形式展示观测实例的统计信息。用户可以通过统计信息了解观测实例的指标变化情况。

1. 登录微服务管理平台，在左侧导航栏中选择**观测** > **服务观测**。
2. 在服务观测查询中，设置查询条件，单击**查询**。
   - 显示时间：单击下拉框，在下拉框中选择显示时间范围，如 5 分钟前、1 小时前。
   - 观测服务：单击下拉框，在下拉框中选择服务
   - 版本：单击下拉框，在下拉框中选择服务版本。
   - 观测实例：单击下拉框，在下拉框中选择观测实例。



告警

操作场景
KMSE 微服务管理平台提供告警查询功能，主要包括告警策略、告警历史两部分。为用户提供简单便捷的告警服务。目前支持按照 GC、系统负载、请求错误率等指标进行告警。

前提条件
向工程中添加依赖，在 pom.xml 中添加以下依赖：

```
<dependency>
  <groupId>com.ksyun.kmse</groupId>
  <artifactId>spring-cloud-kmse-starter-prometheus</artifactId>
  <version>${version}</version>
</dependency>
```

告警策略

1. 登录微服务管理平台，在左侧导航栏中选择【告警-服务告警】，切换至”告警策略“标签页。
2. 点击“新建告警策略”，进入新建策略页面。

3. 填写”告警范围”、“设置告警规则”、“告警级别”、“告警通知”，单击【提交】。
4. 列表中会自动显示已提交的告警策略。 **告警历史**
   告警触发后，告警历史列表中会自动显示告警历史记录。
   登录微服务管理平台，在左侧导航栏中选择【告警-服务告警】，切换至”告警历史“标签页，查看发送状态。

Dubbo应用接入

操作场景
KMSE作为服务注册中心，通过依赖 jar 包的方式接入Dubbo服务。本例介绍如何通过Spring Boot的方式接入，其他方式请参阅Dubbo官网。
操作步骤

1. 添加依赖 根据业务使用的对应的 Dubbo 版本 SDK 如下：
   maven依赖

```
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo-spring-boot-starter</artifactId>
  <version>${dubbo-version}</version>
</dependency>
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo-registry-consul</artifactId>
  <version>${dubbo-version}</version>
</dependency>
<dependency>
  <groupId>com.orbitz.consul</groupId>
  <artifactId>consul-client</artifactId>
  <version>${consul-client-version}</version>
</dependency>
```

2. 编辑相关配置 第一步，编辑application.yaml

```
dubbo:
  application:
    name: hello-world-app
  registry:
    protocol: consul
    address: consul-kmse-system-consul-client.kmse-system #本地调试时请改为本地地址
    port: 8500
  protocol:
    name: dubbo
    port: 20880
```

第二步，编写启动类

```
@EnableDubbo
@SpringBootApplication
public class DubboProviderApplication{
   public static void main(String[] args) throws IOException {
      SpringApplication.run(DubboProviderApplication.class, args);
   }
}
```

第三步，编写接口和实现类

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

@DubboService(interfaceClass = DemoService.class)
@Component
public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) {
```

```
        return "Hello " + name;
    }
}
```

3. Spring Boot 打包 通过 spring-boot-maven-plugin 构建一个包含所有依赖的 jar 包（FatJar），执行命令mvn clean package。
4. 启动服务

```
// 日志显示以下内容表示启动成功
2020-08-17 14:52:21.528  INFO 29568 --- [pool-1-thread-1] .b.c.e.AwaitingNonWebApplicationListener :  [Dubbo] Current Spring Boot Application is await...
```

端云联调

操作场景
在进行应用开发时，本地应用需要和云端的应用进行测试联调，而用户无 VPN的场景下，可以轻松通过本方案，帮助您轻松实现端云联调。
前提条件
购买代理服务器 1.在金山云购买一台云服务器，配置该云服务器网络，与微服务集群处于同一VPC。 2.如果需要集群内部服务调用本地服务，需要修改代理服务器的ssh配置。

```
vim /etc/ssh/sshd_config
```

修改ssh相关配置

```
GatewayPorts clientspecified
```

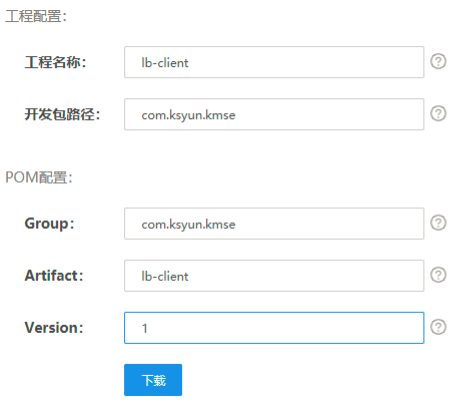
重启ssh服务

```
sudo systemctl restart sshd
```

使用限制

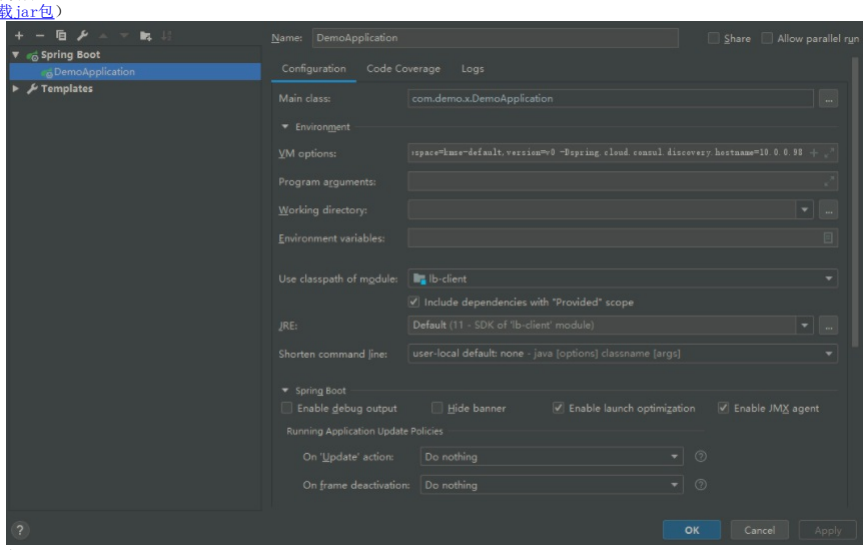
- 代理服务器必须和微服务集群处于同一VPC。
- 不支持监控、日志、链路追踪联调。

操作步骤
1.新建本地应用



微服务平台下载新的项目的agent包（点击下载jar包）

2.下载联调



3.配置jvm启动参数

- JVM启动参数配置如下:

```
-javaagent:"D:\kmse-debug-agent-1.0-SNAPSHOT-jar-with-dependencies.jar=ssh_host=代理服务器公网ip&&user=用户名&&pass=密码&&local_port=本地服务端口&&remote_ip=代理服务器内网ip"
-Dspring.cloud.consul.discovery.tags=namespace=命名空间,version=服务版本
-Dspring.cloud.consul.discovery.hostname=代理服务器内网ip
-Dspring.cloud.consul.host=consul server的ip
```

- 示例:

```
-javaagent:"D:\kmse-debug-agent\kmse-debug-agent\target\kmse-debug-agent-1.0-SNAPSHOT-jar-with-dependencies.jar=ssh_host=120.92.109.190&&user=root&&pass=Wy140408&&local_port=8081&&remote_ip=10.0.0.98"
-Dspring.cloud.consul.discovery.tags=namespace=kmse-default,version=v0
-Dspring.cloud.consul.discovery.hostname=10.0.0.98
-Dspring.cloud.consul.host=http://172.16.2.9
```

参数	描述	必填
javaagent	kmse-debug-agent.jar地址	必填
ssh_host	代理服务器公网ip	必填
user	ssh登录代理服务器用户名	必填
pass	ssh登录代理服务器密码	必填
local_port	本地服务端口	必填
remote_ip	代理服务器内网ip	选填(集群内部服务不调用本地时可以忽略)
namespace	服务所在命名空间	必填
version	服务版本	必填
spring.cloud.consul.discovery.hostname	代理服务器内网ip	必填
spring.cloud.consul.host	consul server pod的ip	必填
spring.cloud.consul.discovery.tags	服务的命名空间和版本(参考示例格式)	必填

- 配置详情：

4. 启动服务联调验证

本地测试能成功访问线上集群。

```
PS C:\Users\Administrator> curl http://127.0.0.1:8081/server/x/2


StatusCode        : 200
StatusDescription :
Content           : v6#server-v6-ff54589d-jjhgl
RawContent        : HTTP/1.1 200
                    Content-Length: 27
                    Content-Type: text/plain;charset=UTF-8
                    Date: Mon, 12 Oct 2020 02:55:45 GMT

                    v6#server-v6-ff54589d-jjhgl
Forms             : {}
Headers           : {[Content-Length, 27], [Content-Type, text/plain;charset=UTF-8], [Date, Mon, 12 Oct 2020 02:55:45 G
                    MT]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 27
```