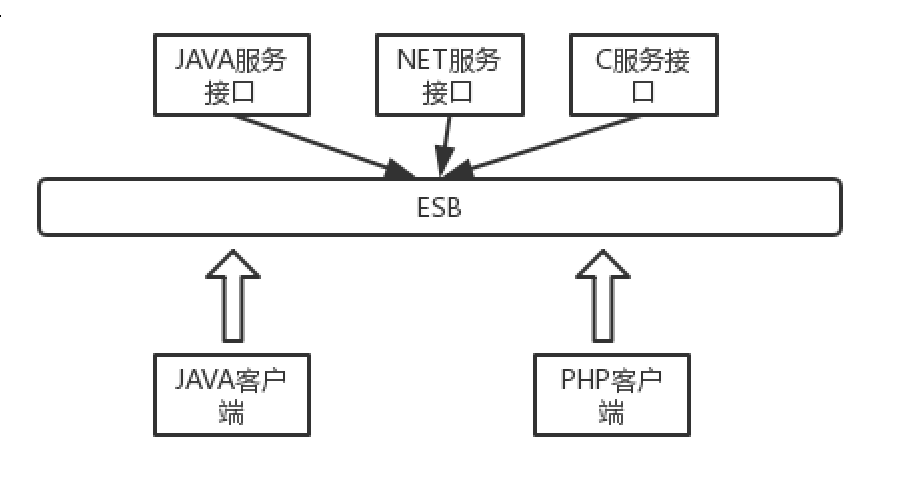
# RestFul简介

10几年前还是WebService的天下，但用过WebService的同学可能都知道，它使用的是WSDL文件描述的SOAP协议进行通讯了，这里重点不是讲WebService但在我们今天要讲到的RestFul却是基于WebService演进而来。

WebService的问题大概如下：

* 由于使用的SOAP协议，使用WSDL，这本质上是使用的XML进行内容通信，速度太慢，处理的效率太低
* 如果想使用本地接口的方式调用，要利用开发工具根据WSDL文件生成很多工具代码，接口任何变动都回导致工具代码重新生成，开发特别繁琐

在2005年的时候，又有一个新的概念产生，既SOA(面向服务架构)，他提出了一个服务总线(ESB)的概念。



这ESB类似于设计模式里面适配器模式，不管服务提供方是使用什么语音实现的，在他注册到消息总线里面，用过使用同样的一套规范，这样客户端调用就省事了很多。

所有的服务由服务总线通过管理，但这个消息总线其实只是一种思想，具体要要通过具体的RPC框架实现，说到RPC，其中最有代表意义的就是前面以及学习过的dubbo开发技术，他可以使用dubbo协议，或者RMI协议，在传输层是用netty。

除开耳熟能详的dubbo,成长比较快的是Rest协议，这其实就是因为JSON的广泛应用，如果说是XML技术成就了webservice，那么JSON就造就了Rest服务。

不过Rest还是一个未成型的标准，SpringCloud就真正讲Rest作为了RPC的实现技术，SpringCloud这技术以及成为了这行业的趋势，而且 SpringCloud 也依照于 SpringBoot

开发技术，可以实现项目的打包发布以及单独运行。

# Rest项目演练

## 项目搭建

既然是微服务，整个项目必然会参加成一个个微模块，

microcloud-api 模块，作为公共的信息导入配置模块；

microcloud-provider-product：作为服务提供者;

microcloud-consumer：作为微服务调用的客户端使用;

microcloud

新建立一个maven项目：microcloud

其中pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>enjoy</groupId>

<artifactId>springcloud</artifactId>

<packaging>pom</packaging>

<version>1.0-SNAPSHOT</version>

<modules>

<module>microcloudapi</module>

<module>microcloudproviderproduct</module>

<module>microcloudconsumer</module>

</modules>

<properties>

<jdk.version>1.8</jdk.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencyManagement>

<dependencies>

<dependency> <!-- 进行SpringCloud依赖包的导入处理 -->

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency> <!-- SpringCloud离不开SpringBoot，所以必须要配置此依赖包 -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>2.1.2.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.31</version>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.3.0</version>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<finalName>microcloud</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>${jdk.version}</source><!-- 源代码使用的开发版本 -->

<target>${jdk.version}</target><!-- 需要生成的目标class文件的编译版本 -->

</configuration>

</plugin>

</plugins>

</build>

</project>

注意：spingcloud中针对依赖包的版本并不像传统项目一样使用的是数字形式定义，反而是使用了一系列英国的地铁或者城市名字来定义，springcloud使用了springboot，其中对于的版本如下

Release Train Boot Version

Greenwich 2.1.x

Finchley 2.0.x

Edgware 1.5.x

Dalston 1.5.x

【microcloud-api】模块，建立一个公共模板，这模块的主要功能是提供公共处理的工具类，实体，接口等。

pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-api</artifactId>

<version>1.0.0</version>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

</project>

由于实体对象不管是服务提供放还是消费者都需要用到，实体对象先创建到api模块中，创建一个Product实体

package cn.enjoy.vo;

import java.io.Serializable;

public class Product implements Serializable {

private Long productId;

private String productName;

private String productDesc;

public String getProductDesc() {

return productDesc;

}

public void setProductDesc(String productDesc) {

this.productDesc = productDesc;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public Long getProductId() {

return productId;

}

public void setProductId(Long productId) {

this.productId = productId;

}

@Override

public String toString() {

return "Product{" +

"productId=" + productId +

", productName='" + productName + '\'' +

", productDesc='" + productDesc + '\'' +

'}';

}

}

## 服务提供方

【microcloud-provider-product】创建一个Product Rest提供者的项目模块,这个模块对应的数据库脚本如下

CREATE DATABASE springcloud CHARACTER SET UTF8 ;

USE springcloud ;

CREATE TABLE product (

prodcutId BIGINT AUTO\_INCREMENT ,

productName VARCHAR(50) ,

productDesc VARCHAR(50) ,

CONSTRAINT pk\_prodcut\_id PRIMARY KEY(prodcutId)

) ;

INSERT INTO product(productName,productDesc) VALUES ('java编程',database()) ;

INSERT INTO product(productName,productDesc) VALUES ('Springboot',database()) ;

INSERT INTO product(productName,productDesc) VALUES ('西游记',database()) ;

INSERT INTO product(productName,productDesc) VALUES ('水浒传',database()) ;

INSERT INTO product(productName,productDesc) VALUES ('西厢记',database()) ;

【microcloud-provider-product】模块继续使用mybaits对数据库进行操作，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<version>1.0.0</version>

<artifactId>microcloud-provider-product</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-provider-product】创建一个ProductMapper对数据库的操作接口，这个接口方法特别简单

package cn.enjoy.mapper;

import cn.enjoy.vo.Product;

import java.util.List;

public interface ProductMapper {

boolean create(Product product);

public Product findById(Long id);

public List<Product> findAll();

}

【microcloud-provider-product】新增修改application.yml文件，追加对mybatis以及数据库的支持

server:

port: 8080

mybatis:

mapper-locations: # 所有的mapper映射文件

- classpath:mapping/\*.xml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource # 配置当前要使用的数据源的操作类型

driver-class-name: com.mysql.cj.jdbc.Driver # 配置MySQL的驱动程序类

url: jdbc:mysql://localhost:3306/springcloud?serverTimezone=GMT%2B8 # 数据库连接地址

username: root # 数据库用户名

password: root1234% # 数据库连接密码

logging:

level:

cn.enjoy.mapper: debug

【microcloud-provider-product】创建修改 src/main/resources/mapping/ProductMapper.xml文件

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="cn.enjoy.mapper.ProductMapper">

<select id="findById" resultType="cn.enjoy.vo.Product" parameterType="long">

select productId,productName,productDesc from product WHERE productId=#{id} ;

</select>

<select id="findAll" resultType="cn.enjoy.vo.Product">

SELECT productId,productName,productDesc from product;

</select>

<insert id="create" parameterType="cn.enjoy.vo.Product">

INSERT INTO product(productName,productDesc) VALUES (#{productName},database()) ;

</insert>

</mapper>

【microcloud-provider-product】建立IProductService接口，并创建相关实现类

package cn.enjoy.service;

import cn.enjoy.vo.Product;

import java.util.List;

public interface IProductService {

Product get(long id);

boolean add(Product product);

List<Product> list();

}

package cn.enjoy.service.impl;

import cn.enjoy.mapper.ProductMapper;

import cn.enjoy.service.IProductService;

import cn.enjoy.vo.Product;

import org.springframework.stereotype.Service;

import javax.annotation.Resource;

import java.util.List;

@Service

public class ProductServiceImpl implements IProductService {

@Resource

private ProductMapper productMapper;

@Override

public Product get(long id) {

return productMapper.findById(id);

}

@Override

public boolean add(Product product) {

return productMapper.create(product);

}

@Override

public List<Product> list() {

return productMapper.findAll();

}

}

【microcloud-provider-product】 定义主程序类，并定义好mapper扫描包

package cn.enjoy;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

@MapperScan("cn.enjoy.mapper")

public class ProductApp{

public static void main(String[] args) {

SpringApplication.run(ProductApp.class,args);

}

}

【microcloud-provider-product】编写单元测试

package cn.enjoy;

import cn.enjoy.service.IProductService;

import cn.enjoy.vo.Product;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import javax.annotation.Resource;

@SpringBootTest(classes = ProductApp.class)

@RunWith(SpringRunner.class)

public class ProductServiceTest {

@Resource

private IProductService iProductService;

@Test

public void testGet() {

System.out.println(iProductService.get(1));

}

@Test

public void testAdd() {

Product dept = new Product() ;

dept.setProductName("lison-" + System.currentTimeMillis());

System.out.println(iProductService.add(dept));

}

@Test

public void testList() {

System.out.println(iProductService.list());

}

}

测试完成，确认所有服务的方法都正确

【microcloud-provider-product】建立ProductController建立一个Rest服务类

package cn.enjoy.controller;

import cn.enjoy.service.IProductService;

import cn.enjoy.vo.Product;

import org.springframework.web.bind.annotation.\*;

import javax.annotation.Resource;

@RestController

@RequestMapping("/prodcut")

public class ProductController {

@Resource

private IProductService iProductService;

@RequestMapping(value="/get/{id}")

public Object get(@PathVariable("id") long id) {

return this.iProductService.get(id) ;

}

@RequestMapping(value="/add")

public Object add(@RequestBody Product product) {

return this.iProductService.add(product) ;

}

@RequestMapping(value="/list")

public Object list() {

return this.iProductService.list() ;

}

}

浏览器访问：

调用get请求：[localhost:8080/prodcut/get/1](http://localhost:8080/prodcut/get/1)

调用list请求：localhost:8080/prodcut/list

## 服务消费方

创建一个maven新模块：【microcloud-consumer】

这个模块作为服务的消费方，调用前面的product服务

【microcloud-consumer】修改pom文件，pom文件内容如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-consumer</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-consumer】修改application.yml配置文件

server:

port: 80

【microcloud-consumer】创建Rest配置类，在这需要调用Rest服务，一般需要用到RestTemplate类对象

package cn.enjoy.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class RestConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

【microcloud-consumer】新建一个controller，负责使用RestTemplate调用远程的product服务

package cn.enjoy.controller;

import cn.enjoy.vo.Product;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.util.List;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

public static final String PRODUCT\_GET\_URL = "http://localhost:8080/prodcut/get/";

public static final String PRODUCT\_LIST\_URL="http://localhost:8080/prodcut/list/";

public static final String PRODUCT\_ADD\_URL = "http://localhost:8080/prodcut/add/";

@Resource

private RestTemplate restTemplate;

@RequestMapping("/product/get")

public Object getProduct(long id) {

Product product = restTemplate.getForObject(PRODUCT\_GET\_URL + id, Product.class);

return product;

}

@RequestMapping("/product/list")

public Object listProduct() {

List<Product> list = restTemplate.getForObject(PRODUCT\_LIST\_URL, List.class);

return list;

}

@RequestMapping("/product/add")

public Object addPorduct(Product product) {

Boolean result = restTemplate.postForObject(PRODUCT\_ADD\_URL, product, Boolean.class);

return result;

}

}

【microcloud-consumer】编写启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ConsumerApp {

public static void main(String[] args) {

SpringApplication.run(ConsumerApp.class,args);

}

}

调用测试

新增：<http://localhost/consumer/product/add?productName=lison>

列表查询：<http://localhost/consumer/product/list>

获得单个数据：<http://localhost/consumer/product/get?id=1>

# SpringSecurity

## 服务提供方配置安全验证

前面使用了RestTemplate进行远程接口调用，但要注意，这些Rest服务最终都可能暴露在公网的，任何人都可能调用，如果你的Rest服务属于一些私密信息，这样会导致信息的泄露。

如果想进行安全方面的处理，首先要在服务的提供方上进行处理。

【microcloud-provider-product】修改pom文件，追加 SpringSecurity 相关依赖信息

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

【microcloud-provider-product】修改application.yml配置文件，进行安全的用户名配置

spring:

security:

user:

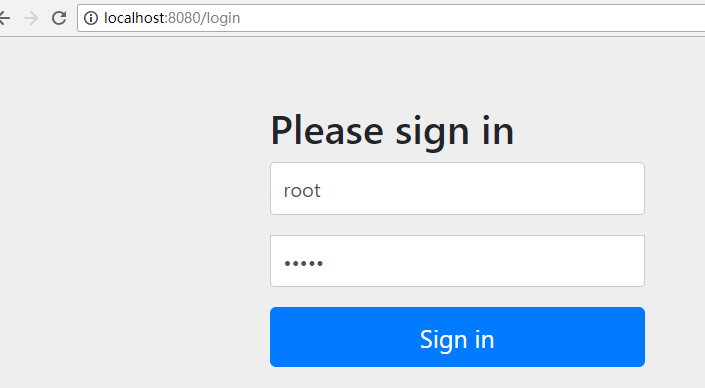
name: admin # 认证用户名

password: enjoy # 认证密码

roles:

- USER # 授权角色

在项目中访问rest接口，localhost:8080/prodcut/list，这个时候会要求先输入用户名以及密码才能允许访问



输入用户名密码就可以调用前面的接口了

## 服务消费方处理

服务提供方目前已经使用了密码验证，这个时候服务的消费方如果想直接访问就不可能了，这个时候一个以头的信息进行处理，然后使用Base64进行加密处理后才能得到正确的访问路径

【microcloud-consumer】 修改RestConfig配置类，在里面添加 HttpHeaders 的配置信息

package cn.enjoy.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpHeaders;

import org.springframework.web.client.RestTemplate;

import java.nio.charset.Charset;

import java.util.Base64;

@Configuration

public class RestConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public HttpHeaders getHeaders() { // 要进行一个Http头信息配置

HttpHeaders headers = new HttpHeaders(); // 定义一个HTTP的头信息

String auth = "admin:enjoy"; // 认证的原始信息

byte[] encodedAuth = Base64.getEncoder()

.encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

String authHeader = "Basic " + new String(encodedAuth);

headers.set("Authorization", authHeader);

return headers;

}

}

【microcloud-consumer】 修改ConsumerProductController，在进行服务端调用的时候加上这个头信息

package cn.enjoy.controller;

import cn.enjoy.vo.Product;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.util.List;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

public static final String PRODUCT\_GET\_URL = "http://localhost:8080/prodcut/get/";

public static final String PRODUCT\_LIST\_URL="http://localhost:8080/prodcut/list/";

public static final String PRODUCT\_ADD\_URL = "http://localhost:8080/prodcut/add/";

@Resource

private RestTemplate restTemplate;

@Resource

private HttpHeaders httpHeaders;

@RequestMapping("/product/get")

public Object getProduct(long id) {

Product product = restTemplate.exchange(PRODUCT\_GET\_URL + id,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), Product.class).getBody();

return product;

}

@RequestMapping("/product/list")

public Object listProduct() {

List<Product> list = restTemplate.exchange(PRODUCT\_LIST\_URL,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), List.class).getBody();

return list;

}

@RequestMapping("/product/add")

public Object addPorduct(Product product) {

Boolean result = restTemplate.exchange(PRODUCT\_ADD\_URL, HttpMethod.POST,new HttpEntity<Object>(product,httpHeaders), Boolean.class).getBody();

return result;

}

}

调用测试

新增：<http://localhost/consumer/product/add?productName=lison>

列表查询：<http://localhost/consumer/product/list>

获得单个数据：<http://localhost/consumer/product/get?id=1>

## 【microcloud-security】模块

现在服务提供方只有一个Product服务，但真实的项目开发中必然有多个服务提供方，绝大多数情况下，这些服务都会用到安全验证，而且密码也会一样，如果每个服务都单独维护，每次密码变动改动都会很大，所以应该单独建立一个安全验证的模块

创建一个microcloud-security模块，修改其pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<version>1.0.0</version>

<artifactId>microcloud-security</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

</project>

【springcloud】修改父工程pom文件，把相应的版本依赖加到里面

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

<version>1.0.0</version>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-security</artifactId>

<version>1.0.0</version>

</dependency>

【microcloud-security】建立一个统一的安全配置类，这个类负责用户以及密码相关的配置

package cn.enjoy.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration

@EnableWebSecurity

public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

public void configure(AuthenticationManagerBuilder auth)

throws Exception {

auth.inMemoryAuthentication().passwordEncoder(new BCryptPasswordEncoder()).withUser("root").password(new BCryptPasswordEncoder().encode("enjoy")).roles("USER").

and().withUser("admin").password(new BCryptPasswordEncoder().encode("enjoy")).roles("USER", "ADMIN");

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http.httpBasic().and().authorizeRequests().anyRequest()

.fullyAuthenticated();

http.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

}

}

【microcloud-provider-product】修改pom文件，删除spring-boot-starter-security的依赖信息，并加入自己定义的microcloud-security依赖

<!--<dependency>-->

<!--<groupId>org.springframework.boot</groupId>-->

<!--<artifactId>spring-boot-starter-security</artifactId>-->

<!--</dependency>-->

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-security</artifactId>

</dependency>

【microcloud-provider-product】修改application.yml，删除与安全相关的配置项。

~~# security:~~

~~# user:~~

~~# roles:~~

~~# - USER # 授权角色~~

~~# name: root~~

~~# password: enjoy~~

调用测试

新增：<http://localhost/consumer/product/add?productName=lison>

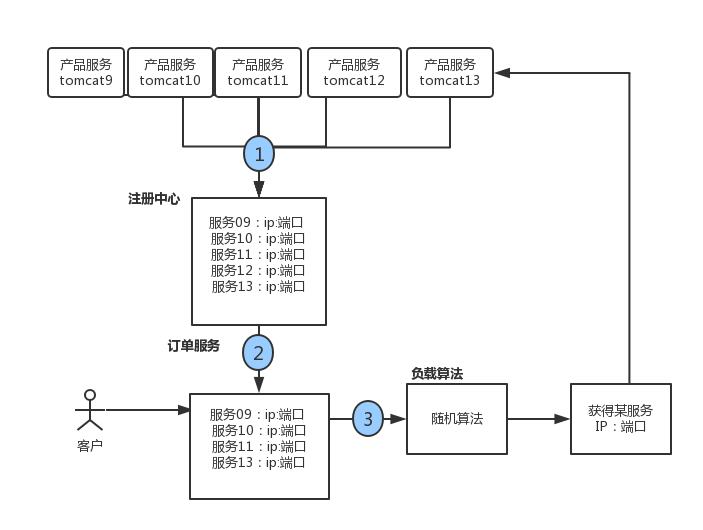
列表查询：<http://localhost/consumer/product/list>

获得单个数据：<http://localhost/consumer/product/get?id=1>

# Eureka服务注册与发现

在学习zookeeper的时候，重点就讲了服务注册与发现的流程，dubbo就是基于zookeeper来实现服务注册与发现的。

基于zookeeper的服务注册与发现大致流程如下



而在SpringCloud中，大量使用了Netflix的开源项目，其中Eureka就属于Netflix 提供的发现服务组件，所有的微服务都注册到Eureka中，它在其中扮演的就是注册中心的角色，后面所有的客户端直接从注册中心获取所需要的服务

## Eureka 服务端

新建一个microcloud-eureka模块，这模块做的事情非常简单，既启动Eureka的服务端，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-eureka</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

</dependencies>

</project>

注意：

如果是Edgware或之前的版本，用的是springboot 1.5.或者更低的版本，

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

【microcloud-eureka】修改application.yml文件，在里面配置eureka相关信息

server:

port: 7001

eureka:

instance: # eureak实例定义

hostname: localhost # 定义 Eureka 实例所在的主机名称

【microcloud-eureka】新增Eureka启动类，增加Eureka服务端注解

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class EurekaApp {

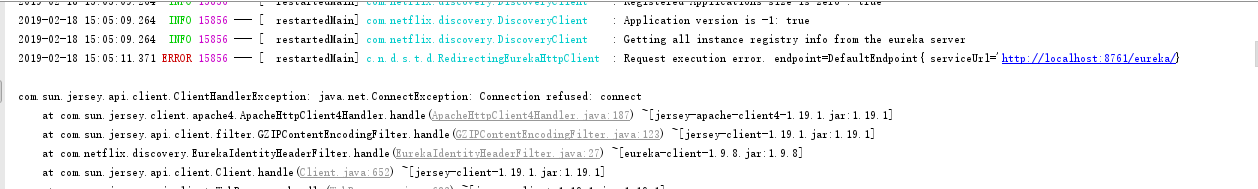
public static void main(String[] args) {

SpringApplication.run(EurekaApp.class,args);

}

}

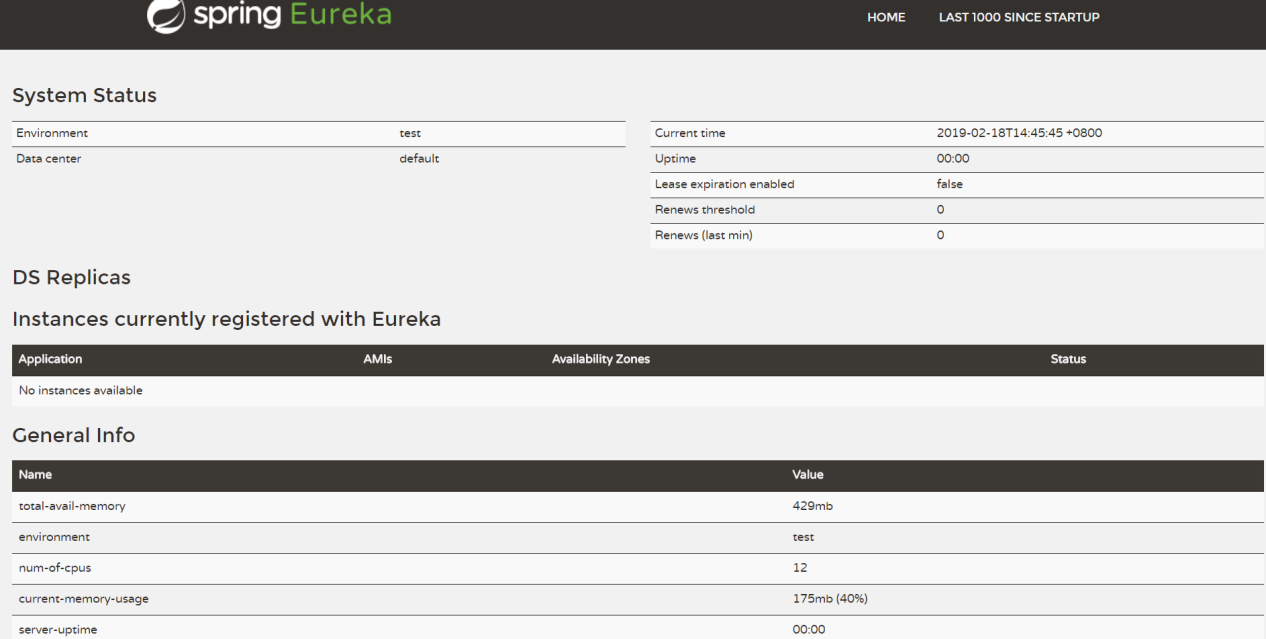
运行main方法



上面会有一些出错信息，先不用管，后面再来处理！

在浏览器上执行

<http://localhost:7001/>



## 服务提供方注册到Eureka

现在Eureka虽然有点小瑕疵，但现在已经能正常访问了，那么接下来就需要将用的微服务注册到Eureka服务当中，为后面客户端的使用做铺垫。

【microcloud-provider-product】修改pom文件，增加eureka客户端相关信息。

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

注意：如果是Edgware或之前的版本，用的是springboot 1.5.或者更低的版本

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

【microcloud-provider-product】修改application.yml配置文件，在者个文件中定义要注册的eureka服务的地址

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://localhost:7001/eureka

【microcloud-provider-product】修改启动类，在这个类上增加eureka客户端的注解信息

package cn.enjoy;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@MapperScan("cn.enjoy.mapper")

@EnableEurekaClient

public class ProductApp{

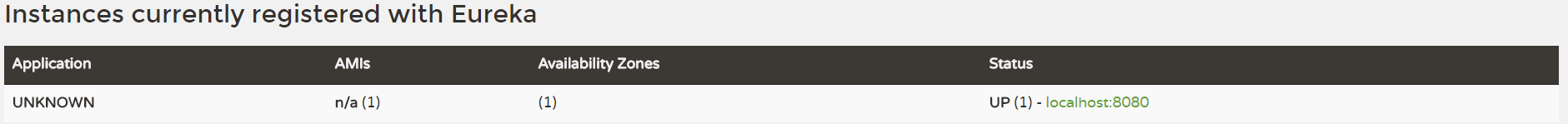
public static void main(String[] args) {

SpringApplication.run(ProductApp.class,args);

}

}

加上这注解后，启动



发现Application的名字是UNKNOWN，为此应该为这单独取一个名字

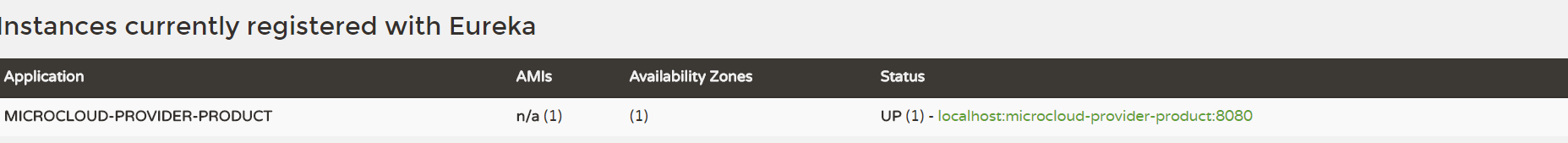
【microcloud-provider-product】修改application.yml配置文件，为这个微服务起一个名字

spring:

application:

name: microcloud-provider-product

重新启动后



现在虽然成功的实现了微服务注册，但是现在看下STATUS，这个时候名称还有点乱，我们现在开源自定义一个路径名称

【microcloud-provider-product】修改application.yml配置文件，追加主机名称的显示：

eureka:

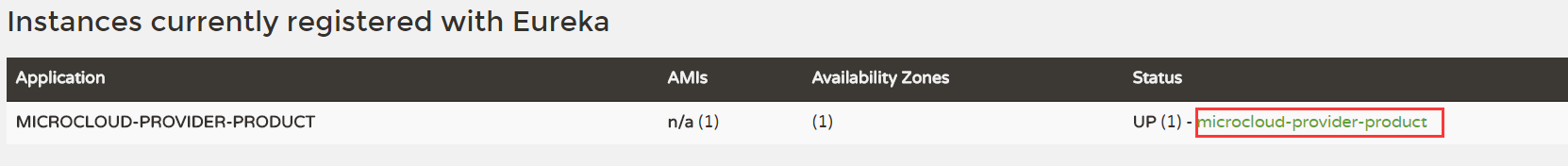
client: # 客户端进行Eureka注册的配置

service-url:

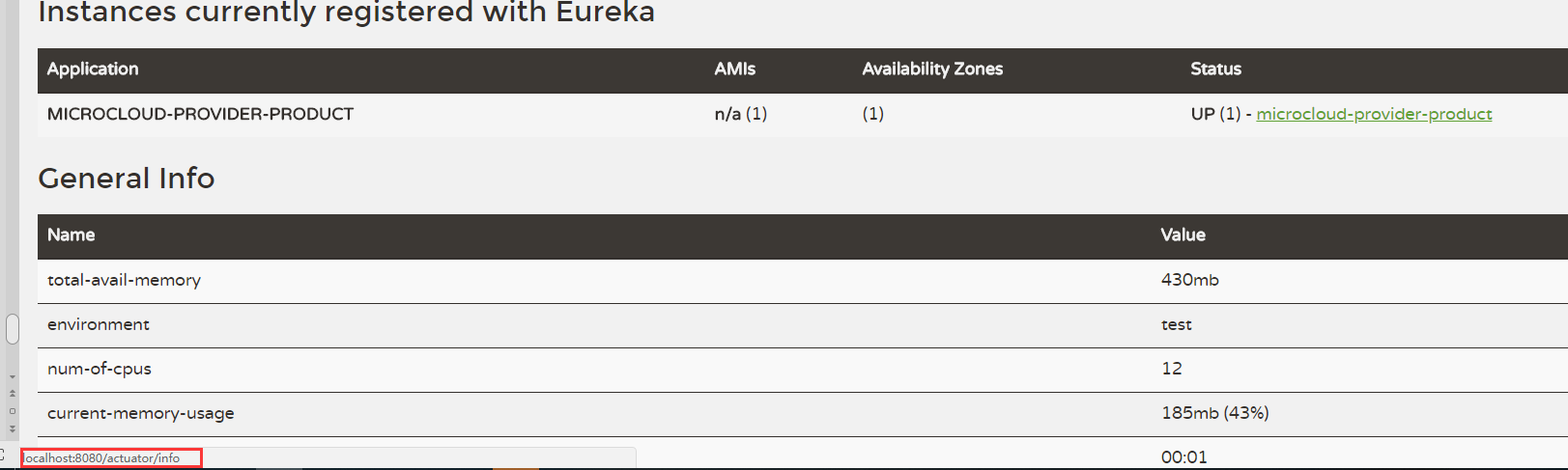
defaultZone: http://localhost:7001/eureka

instance:

instance-id: microcloud-provider-product



另外一般情况下，当鼠标点击查看的时候应该以IP作为链接项



【microcloud-provider-product】修改application.yml配置文件

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

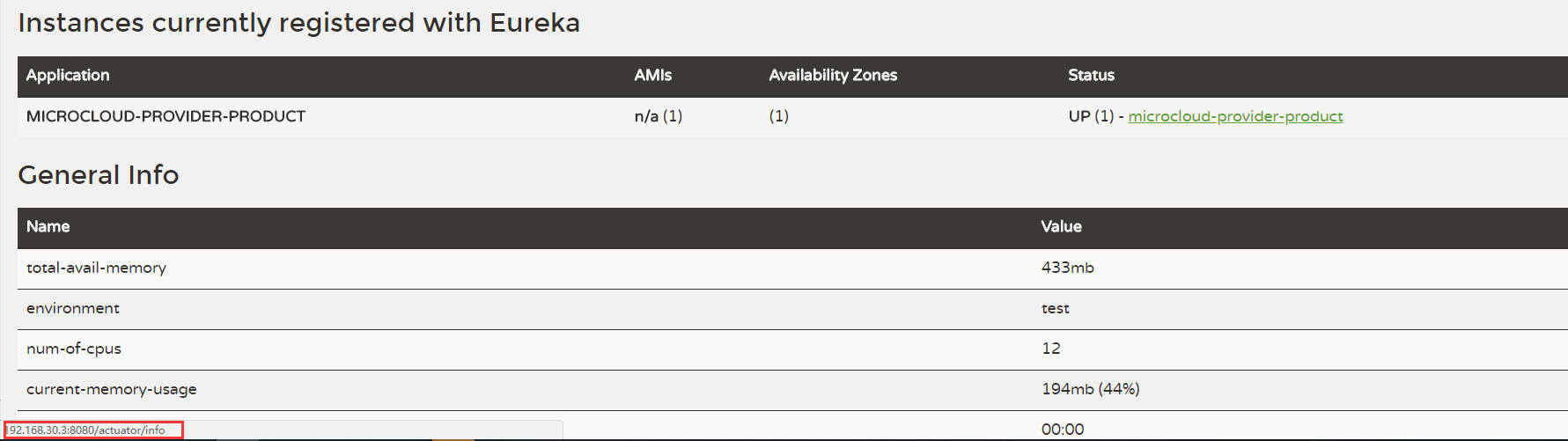
defaultZone: http://localhost:7001/eureka

instance:

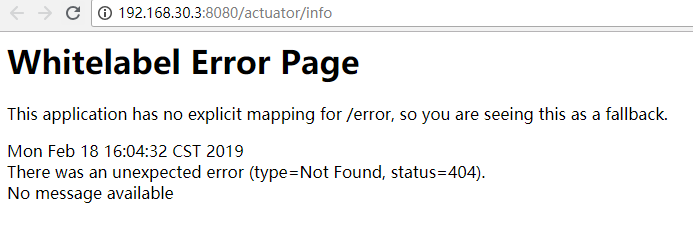
instance-id: microcloud-provider-product

prefer-ip-address: true

修改后地址栏变为



点击状态栏



【microcloud-provider-product】如果想看状态信息需要增加actuator模块，这一块的内容已经在讲springboot的时候讲过，修改pom文件，增加

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

【microcloud-provider-product】修改application.yml文件，追加info相关配置

info:

app.name: microcloud-provider-product

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

注意：由于在yml文件中使用了$，这个时候启动是会报错的，因此还需要一个maven-resources-plugin插件的支持

【microcloud】在父工程增加插件，修改pom文件

<build>

<finalName>microcloud</finalName>

<resources>

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<delimiters>

<delimiter>$</delimiter>

</delimiters>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>${jdk.version}</source><!-- 源代码使用的开发版本 -->

<target>${jdk.version}</target><!-- 需要生成的目标class文件的编译版本 -->

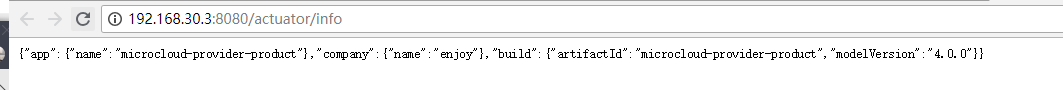
</configuration>

</plugin>

</plugins>

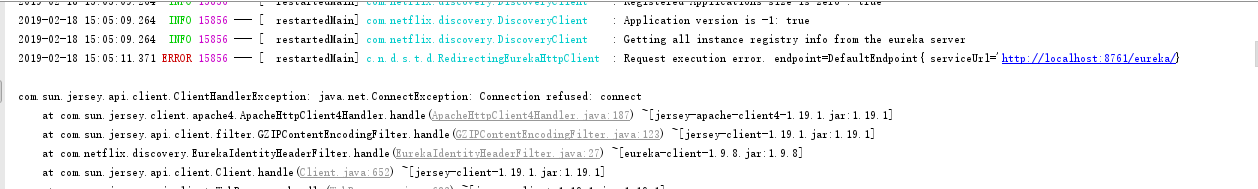
</build>

启动后：

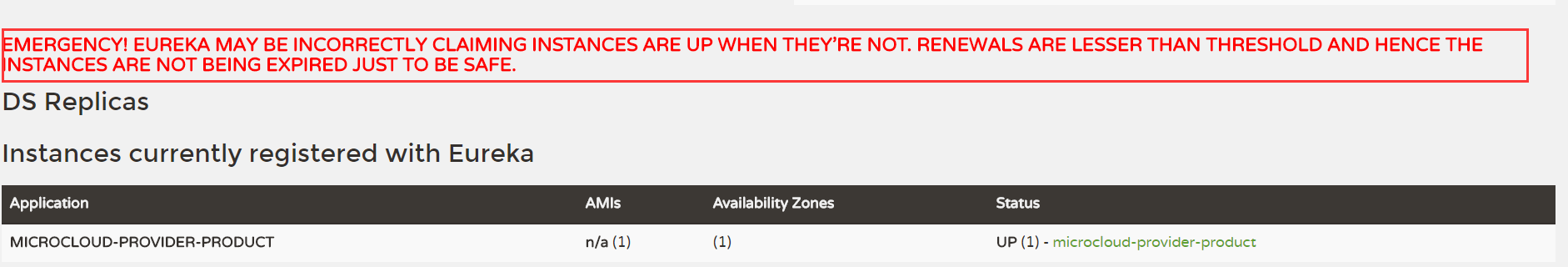


## 其他配置

在前面启动eureka中，会发现启动会报错，虽然这些错误不影响使用



另外在关闭【microcloud-provider-product】项目后，刷新eureka发现项目还在，隔一段时间后会发现



这其实就是触发了安全模式

【microcloud-eureka】设置服务的清理间隔时间，修改application.yml文件

server:

port: 7001

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

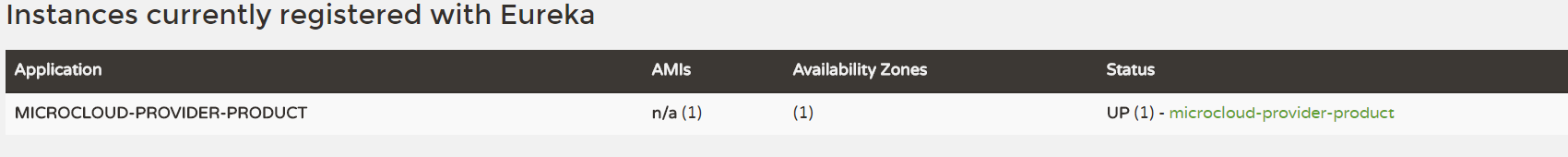
fetch-registry: false

register-with-eureka: false

instance: # eureak实例定义

hostname: localhost # 定义 Eureka 实例所在的主机名称

这个重新测试，服务提供方注册后，关闭服务发现服务实例依然还在。



【microcloud-provider-product】修改application.yml配置

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://localhost:7001/eureka

instance:

instance-id: microcloud-provider-product

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

由于所有的服务都注册到了 Eureka 之中

这样如果配置了“lease-expiration-duration-in-seconds”此选项，

表示距离上一次发送心跳之后等待下一次发送心跳的间隔时间，如果超过了此间隔时间，则认为该微服务已经宕机了。

【microcloud-provider-product】对于注册到 Eureka 上的服务，可以通过发现服务来获取一些服务信息，修改ProductController，增加一个方法

package cn.enjoy.controller;

import cn.enjoy.service.IProductService;

import cn.enjoy.vo.Product;

import org.springframework.cloud.client.discovery.DiscoveryClient;

import org.springframework.web.bind.annotation.\*;

import javax.annotation.Resource;

@RestController

@RequestMapping("/prodcut")

public class ProductController {

@Resource

private IProductService iProductService;

@Resource

private DiscoveryClient client ; // 进行Eureka的发现服务

@RequestMapping(value="/get/{id}")

public Object get(@PathVariable("id") long id) {

return this.iProductService.get(id) ;

}

@RequestMapping(value="/add")

public Object add(@RequestBody Product product) {

return this.iProductService.add(product) ;

}

@RequestMapping(value="/list")

public Object list() {

return this.iProductService.list() ;

}

@RequestMapping("/discover")

public Object discover() { // 直接返回发现服务信息

return this.client ;

}

}

【microcloud-provider-product】修改ProductApp， 在主程序中启用发现服务项

package cn.enjoy;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@MapperScan("cn.enjoy.mapper")

@EnableEurekaClient

@EnableDiscoveryClient

public class ProductApp{

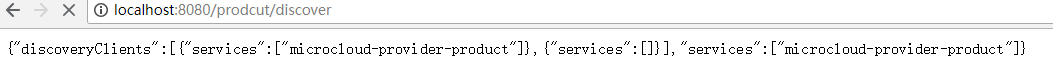
public static void main(String[] args) {

SpringApplication.run(ProductApp.class,args);

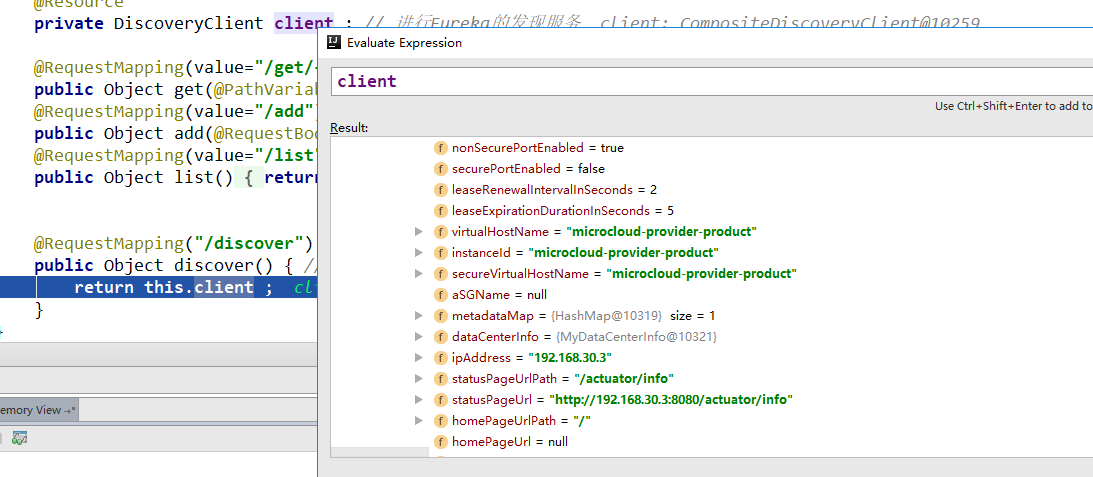
}

}

访问：localhost:8080/prodcut/discover



可以发现DiscoveryClient包含的很多信息



可以看到在eureka里面显示的信息都可以在这里获取得到。

## Eureka 安全机制

一般情况下Eureka 和服务的提供注册者都会在一个内网环境中，但免不了在某些项目中需要让其他外网的服务注册到Eureka，这个时候就有必要让Eureka增加一套安全认证机制了，让所有服务提供者通过安全认证后才能注册进来

【microcloud-eureka】修改pom文件，引入SpringSecurity的依赖包

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

【microcloud-eureka】 修改application.yml文件，增加用户、密码验证

server:

port: 7001

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

instance: # eureak实例定义

hostname: localhost # 定义 Eureka 实例所在的主机名称

spring:

security:

user:

name: admin

password: enjoy

【microcloud-provider-product】修改application.yml文件，增加验证信息

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

instance:

instance-id: microcloud-provider-product

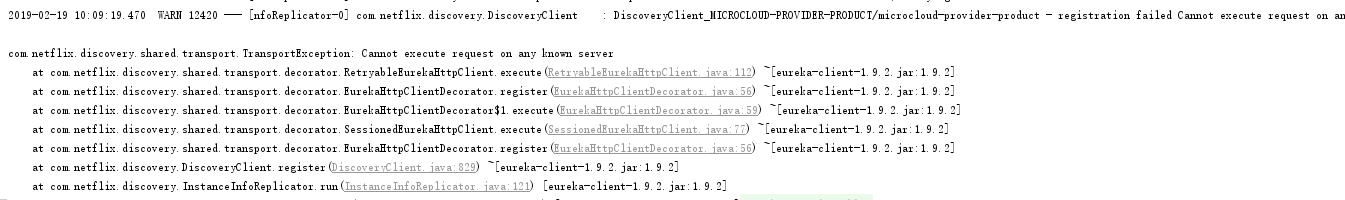
prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

重新启动

注意：如果是Edgware或之前的版本，做到这一步就行了，但使用现在版本，你会发现启动【microcloud-provider-product】后服务注册不上去



【microcloud-eureka】新增配置类EurekaSecurityConfig，重写configure方法，把csrf劫持关闭

package cn.enjoy;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration

@EnableWebSecurity

public class EurekaSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable();

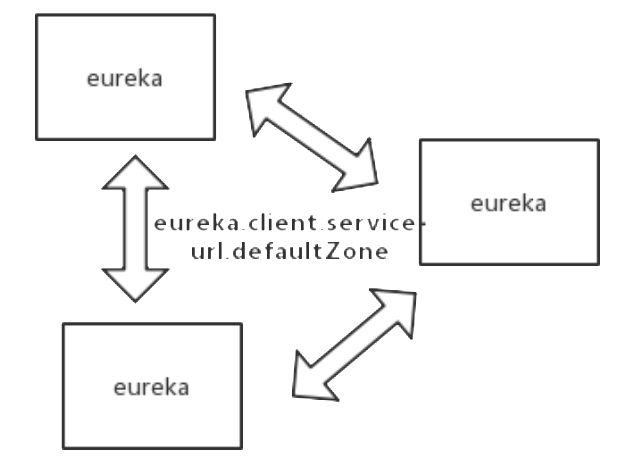
super.configure(http);

}

}

## HA 高可用

学习Zookepper服务注册的时候，注册中心是能实现高可用的，但现在的Eureka还是单节点的情况，如果Eureka出现了错误，将会导致整个集群无法继续使用，这个时候就需要考虑Eureka的高可用了。



现在需要3个eureka ，每个eureka都需要配置hostname,所有先修改hosts文件内容如下

127.0.0.1 eureka1

127.0.0.1 eureka2

127.0.0.1 eureka3

【microcloud-eureka】为了方便操作，讲microcloud-eureka项目复制两份，分别复制为【microcloud-eureka2】、 【microcloud-eureka2】

【microcloud-eureka】修改application.yml配置文件，修改端口以及注册位置

server:

port: 7001

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

#defaultZone: http://admin:enjoy@localhost:7001/eureka

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance: # eureak实例定义

hostname: eureka1 # 定义 Eureka 实例所在的主机名称

spring:

security:

user:

name: admin

password: enjoy

【microcloud-eureka2】修改application.yml配置文件

server:

port: 7002

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

#defaultZone: http://admin:enjoy@localhost:7001/eureka

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance: # eureak实例定义

hostname: eureka2 # 定义 Eureka 实例所在的主机名称

spring:

security:

user:

name: admin

password: enjoy

【microcloud-eureka3】修改application.yml配置文件

server:

port: 7003

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

#defaultZone: http://admin:enjoy@localhost:7001/eureka

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance: # eureak实例定义

hostname: eureka3 # 定义 Eureka 实例所在的主机名称

spring:

security:

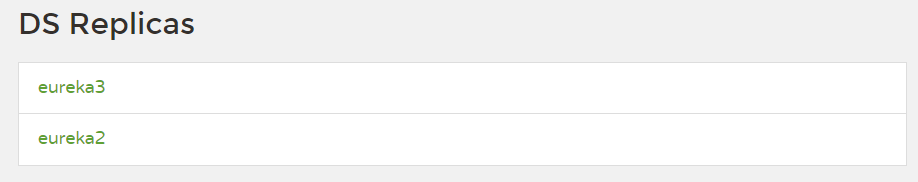
user:

name: admin

password: enjoy

启动eureka,eureka2,eureka3，进入服务的后台查看副本

登陆http://localhost:7001/



【microcloud-provider-product】修改application.yml配置文件，配置多台enreka的注册

server:

port: 8080

mybatis:

mapper-locations: # 所有的mapper映射文件

- classpath:mapping/\*.xml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource # 配置当前要使用的数据源的操作类型

driver-class-name: com.mysql.cj.jdbc.Driver # 配置MySQL的驱动程序类

url: jdbc:mysql://localhost:3306/springcloud?serverTimezone=GMT%2B8 # 数据库连接地址

username: root # 数据库用户名

password: root1234% # 数据库连接密码

application:

name: microcloud-provider-product

# security:

# user:

# roles:

# - USER # 授权角色

# name: root

# password: enjoy

logging:

level:

cn.enjoy.mapper: debug

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

#defaultZone: http://admin:enjoy@localhost:7001/eureka

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance:

instance-id: microcloud-provider-product

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

info:

app.name: microcloud-provider-product

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

## 打包发布

在真实项目中，需要讲Eureka发布到具体服务器上进行执行，打包部署其实和springboot里面讲的大同小异

和properties文件稍微有点不同，对于properties文件，不同的环境会有不同的配置文件比如application-dev.properties，application-test.properties,application-pro.properties等

但如果是yml文件，所有的的配置都再同一个yml文件中

【microcloud-eureka】修改application.yml文件

spring:

profiles:

active:

- dev-7001

---

server:

port: 7001

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance: # eureak实例定义

hostname: eureka1 # 定义 Eureka 实例所在的主机名称

spring:

profiles: dev-7001

security:

user:

name: admin

password: enjoy

application:

name: microcloud-eureka

---

server:

port: 7002

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance: # eureak实例定义

hostname: eureka2 # 定义 Eureka 实例所在的主机名称

spring:

profiles: dev-7002

security:

user:

name: admin

password: enjoy

application:

name: microcloud-eureka2

---

server:

port: 7003

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance: # eureak实例定义

hostname: eureka3 # 定义 Eureka 实例所在的主机名称

spring:

profiles: dev-7003

security:

user:

name: admin

password: enjoy

application:

name: microcloud-eureka3

【microcloud-eureka】添加一个打包插件，修改pom文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-eureka</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

</dependencies>

<build>

<finalName>eureka-server</finalName>

<plugins>

<plugin> <!-- 该插件的主要功能是进行项目的打包发布处理 -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration> <!-- 设置程序执行的主类 -->

<mainClass>cn.enjoy.EurekaApp</mainClass>

</configuration>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

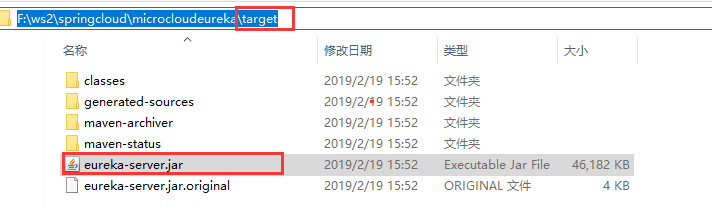
</project>

【microcloud-eureka】 在pom文件所在目录

mvn clean install package

接下来就可以在项目的编译目录发现

eureka-server.jar 文件



采用默认的方式执行 eureka-server.jar

那么此时将运行在 7001 端口上：java -jar eureka-server.jar

运行其它的两个 profile 配置：

· 运行“dev-7002”profile：java -jar eureka-server.jar --spring.profiles.active=dev-7002；

· 运行“dev-7003”profile：java -jar eureka-server.jar --spring.profiles.active=dev-7003

## 原理解析：

### EurekaClient

EurekaClient为了简化开发人员的工作量，将很多与EurekaServer交互的工作隐藏起来，自主完成。具体完成的工作如下

**应用启动阶段**

1. 读取与 Eureka Server交互的配置信息,封装成 EurekaClientConfig
2. 读取自身服务实例配置信息,封装成EurekalnstanceConfig
3. 从Eureka server拉取注册表信息并缓存到本地
4. 服务注册
5. 初始化发送心跳、缓存刷新(拉取注册表信息更新本地缓存)和按需注册(监控服务实例信息变化,决定是否重新发起注册,更新注册表中的服务实例元数据)定时任务

**应用执行阶段**

1. 定时发送心跳到Eureka Server中维持在注册表的租约
2. 定时从 Eureka Server中拉取注册表信息,更新本地注册表缓存
3. 监控应用自身信息变化,若发生变化,需要重新发起服务注册

**应用销毁阶段**

从 Eureka Server注销自身服务实例

#### 应用启动阶段与运行阶段

Eureka Client通过Starter的方式引人依赖, Spring Boot将会为项目使用以下的自动配置类

* **EurekaClientAutoConfiguration：**

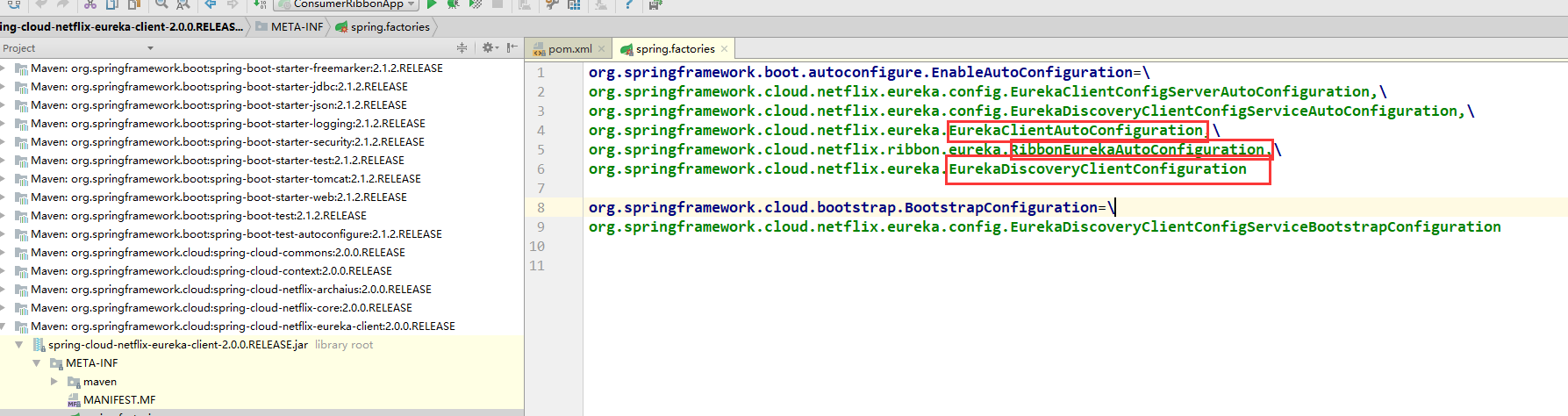
EurekeClient 自动配置类,负责Eureka关键Beans的配置和初始化,如AppplicationInfoManager和 EurekaClientConfig等。

* **RibbonEurekaAutoConfiguration:**

Ribbon负载均衡相关配置。

* **EurekaDiscoveryClientConfiguration:**

配置自动注册和应用的健康检查器。



读取应用自身配置

通过 **EurekaClientAutoConfiguration**配置类, Spring boot帮助 Eureka Client完成很多必要Bean的属性读取和配置

* EurekaClientConfig

封装 Eureka Client与 Eureka Server交互所需要的配置信息。 Spring Cloud为其提供了一个默认配置类的EurekaClientConfigBean,可以在配置文件中通过前缀 eureka.client属性名进行属性覆盖

* ApplicationInfoManager

作为应用信息管理器,管理服务实例的信息类 InstanceInfo和服务实例的配置信息类 EurekaInstanceConfig

* InstanceInfo

封装将被发送到 Eureka Server进行服务注册的服务实例元数据。它在Eurek Server的注册表中代表一个服务实例,其他服务实例可以通过 Instancelnfo了解该服

的相关信息从而发起服务请求

* EurekaInstanceConfig

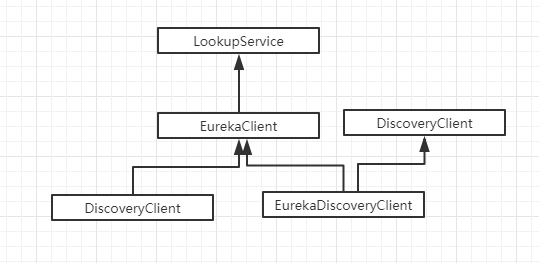
封装EurekaClient自身服务实例的配置信息,主要用于构建InstanceInfo通常这些信息在配置文件中的eureka.instance前缀下进行设置, SpringCloud通过EurekalnstanceConfigBean配置类提供了默认配置

* DiscoveryClient

Spring Cloud中定义用来服务发现的客户端接口

对于DiscoveryClient可以具体查看 EurekaDiscoveryClient，EurekaDiscoveryClient又借助EurekaClient来实现

另外在netflix包里面还有一个DiscoveryClient，按名字翻译其实就是服务发现客户端，他是整个EurekaClient的核心，是与EurekaServer进行交互的核心所在



运行服务

DiscoveryClient(ApplicationInfoManager applicationInfoManager, EurekaClientConfig config, AbstractDiscoveryClientOptionalArgs args,

Provider<BackupRegistry> backupRegistryProvider) {

if (args != null) {

this.healthCheckHandlerProvider = args.healthCheckHandlerProvider;

this.healthCheckCallbackProvider = args.healthCheckCallbackProvider;

this.eventListeners.addAll(args.getEventListeners());

this.preRegistrationHandler = args.preRegistrationHandler;

} else {

this.healthCheckCallbackProvider = null;

this.healthCheckHandlerProvider = null;

this.preRegistrationHandler = null;

}

this.applicationInfoManager = applicationInfoManager;

InstanceInfo myInfo = applicationInfoManager.getInfo();

clientConfig = config;

staticClientConfig = clientConfig;

transportConfig = config.getTransportConfig();

instanceInfo = myInfo;

if (myInfo != null) {

appPathIdentifier = instanceInfo.getAppName() + "/" + instanceInfo.getId();

} else {

logger.warn("Setting instanceInfo to a passed in null value");

}

//传入BackupRegistry（NotImplementedRegistryImpl）备份注册中心

this.backupRegistryProvider = backupRegistryProvider;

this.urlRandomizer = new EndpointUtils.InstanceInfoBasedUrlRandomizer(instanceInfo);

localRegionApps.set(new Applications());

fetchRegistryGeneration = new AtomicLong(0);

remoteRegionsToFetch = new AtomicReference<String>(clientConfig.fetchRegistryForRemoteRegions());

remoteRegionsRef = new AtomicReference<>(remoteRegionsToFetch.get() == null ? null : remoteRegionsToFetch.get().split(","));

//从eureka server拉起注册表信息 eureka.client.fetch-register

if (config.shouldFetchRegistry()) {

this.registryStalenessMonitor = new ThresholdLevelsMetric(this, METRIC\_REGISTRY\_PREFIX + "lastUpdateSec\_", new long[]{15L, 30L, 60L, 120L, 240L, 480L});

} else {

this.registryStalenessMonitor = ThresholdLevelsMetric.NO\_OP\_METRIC;

}

// 当前的客户端是否应该注册到erueka中 eureka.client.register-with-eureka

if (config.shouldRegisterWithEureka()) {

this.heartbeatStalenessMonitor = new ThresholdLevelsMetric(this, METRIC\_REGISTRATION\_PREFIX + "lastHeartbeatSec\_", new long[]{15L, 30L, 60L, 120L, 240L, 480L});

} else {

this.heartbeatStalenessMonitor = ThresholdLevelsMetric.NO\_OP\_METRIC;

}

logger.info("Initializing Eureka in region {}", clientConfig.getRegion());

//如果既不需要注册，也不需要拉去数据，直接返回,初始结束

if (!config.shouldRegisterWithEureka() && !config.shouldFetchRegistry()) {

logger.info("Client configured to neither register nor query for data.");

scheduler = null;

heartbeatExecutor = null;

cacheRefreshExecutor = null;

eurekaTransport = null;

instanceRegionChecker = new InstanceRegionChecker(new PropertyBasedAzToRegionMapper(config), clientConfig.getRegion());

// This is a bit of hack to allow for existing code using DiscoveryManager.getInstance()

// to work with DI'd DiscoveryClient

DiscoveryManager.getInstance().setDiscoveryClient(this);

DiscoveryManager.getInstance().setEurekaClientConfig(config);

initTimestampMs = System.currentTimeMillis();

logger.info("Discovery Client initialized at timestamp {} with initial instances count: {}",

initTimestampMs, this.getApplications().size());

return; // no need to setup up an network tasks and we are done

}

try {

//线程池大小为2，一个用户发送心跳，另外个缓存刷新

// default size of 2 - 1 each for heartbeat and cacheRefresh

scheduler = Executors.newScheduledThreadPool(2,

new ThreadFactoryBuilder()

.setNameFormat("DiscoveryClient-%d")

.setDaemon(true)

.build());

heartbeatExecutor = new ThreadPoolExecutor(

1, clientConfig.getHeartbeatExecutorThreadPoolSize(), 0, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>(),

new ThreadFactoryBuilder()

.setNameFormat("DiscoveryClient-HeartbeatExecutor-%d")

.setDaemon(true)

.build()

); // use direct handoff

cacheRefreshExecutor = new ThreadPoolExecutor(

1, clientConfig.getCacheRefreshExecutorThreadPoolSize(), 0, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>(),

new ThreadFactoryBuilder()

.setNameFormat("DiscoveryClient-CacheRefreshExecutor-%d")

.setDaemon(true)

.build()

); // use direct handoff

//初始化client与server交互的jersey客户端

eurekaTransport = new EurekaTransport();

scheduleServerEndpointTask(eurekaTransport, args);

AzToRegionMapper azToRegionMapper;

if (clientConfig.shouldUseDnsForFetchingServiceUrls()) {

azToRegionMapper = new DNSBasedAzToRegionMapper(clientConfig);

} else {

azToRegionMapper = new PropertyBasedAzToRegionMapper(clientConfig);

}

if (null != remoteRegionsToFetch.get()) {

azToRegionMapper.setRegionsToFetch(remoteRegionsToFetch.get().split(","));

}

instanceRegionChecker = new InstanceRegionChecker(azToRegionMapper, clientConfig.getRegion());

} catch (Throwable e) {

throw new RuntimeException("Failed to initialize DiscoveryClient!", e);

}

//拉取注册表的信息

if (clientConfig.shouldFetchRegistry() && !fetchRegistry(false)) {

fetchRegistryFromBackup();

}

//将服务实例进行注册

// call and execute the pre registration handler before all background tasks (inc registration) is started

if (this.preRegistrationHandler != null) {

this.preRegistrationHandler.beforeRegistration();

}

if (clientConfig.shouldRegisterWithEureka() && clientConfig.shouldEnforceRegistrationAtInit()) {

try {

if (!register() ) {

throw new IllegalStateException("Registration error at startup. Invalid server response.");

}

} catch (Throwable th) {

logger.error("Registration error at startup: {}", th.getMessage());

throw new IllegalStateException(th);

}

}

// finally, init the schedule tasks (e.g. cluster resolvers, heartbeat, instanceInfo replicator, fetch

//初始心跳定时任务，缓存刷新

initScheduledTasks();

try {

Monitors.registerObject(this);

} catch (Throwable e) {

logger.warn("Cannot register timers", e);

}

// This is a bit of hack to allow for existing code using DiscoveryManager.getInstance()

// to work with DI'd DiscoveryClient

DiscoveryManager.getInstance().setDiscoveryClient(this);

DiscoveryManager.getInstance().setEurekaClientConfig(config);

initTimestampMs = System.currentTimeMillis();

logger.info("Discovery Client initialized at timestamp {} with initial instances count: {}",

initTimestampMs, this.getApplications().size());

}

总结

DiscoveryClient构造函数做的事情

1. 相关配置赋值
2. 备份注册中心的初始化，实际什么事都没做
3. 拉取Server注册表中的信息
4. 注册前的预处理
5. 向Server注册自身
6. 初始心跳定时任务，缓存刷新等定时任务

（重要步骤）拉取Server注册表中的信息

if (clientConfig.shouldFetchRegistry() && !fetchRegistry(false))

//是否全量拉去

private boolean fetchRegistry(boolean forceFullRegistryFetch) {

Stopwatch tracer = FETCH\_REGISTRY\_TIMER.start();

try {

// If the delta is disabled or if it is the first time, get all

// applications

//如果增量拉取被禁止全量拉去

Applications applications = getApplications();

if (clientConfig.shouldDisableDelta()

|| (!Strings.isNullOrEmpty(clientConfig.getRegistryRefreshSingleVipAddress()))

|| forceFullRegistryFetch

|| (applications == null)

|| (applications.getRegisteredApplications().size() == 0)

|| (applications.getVersion() == -1)) //Client application does not have latest library supporting delta

{

//全量拉取

getAndStoreFullRegistry();

} else {

//增量拉取

getAndUpdateDelta(applications);

}

applications.setAppsHashCode(applications.getReconcileHashCode());

//打印注册表上所有服务实例信息

logTotalInstances();

} catch (Throwable e) {

logger.error(PREFIX + "{} - was unable to refresh its cache! status = {}", appPathIdentifier, e.getMessage(), e);

return false;

} finally {

if (tracer != null) {

tracer.stop();

}

}

onCacheRefreshed();

// Update remote status based on refreshed data held in the cache

updateInstanceRemoteStatus();

// registry was fetched successfully, so return true

return true;

}

全量拉取

一般只有在第一次拉去注册表信息的时候，全量拉取调用 getAndStoreFullRegistry()方法

private void getAndStoreFullRegistry() throws Throwable {

//拉取注册表的版本信息

long currentUpdateGeneration = fetchRegistryGeneration.get();

logger.info("Getting all instance registry info from the eureka server");

Applications apps = null;

EurekaHttpResponse<Applications> httpResponse = clientConfig.getRegistryRefreshSingleVipAddress() == null

? eurekaTransport.queryClient.getApplications(remoteRegionsRef.get())

: eurekaTransport.queryClient.getVip(clientConfig.getRegistryRefreshSingleVipAddress(), remoteRegionsRef.get());

//拉取成功

if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {

apps = httpResponse.getEntity();

}

logger.info("The response status is {}", httpResponse.getStatusCode());

}

增量拉取

getAndUpdateDelta(applications)增量拉取

private void getAndUpdateDelta(Applications applications) throws Throwable {

long currentUpdateGeneration = fetchRegistryGeneration.get();

//拉取信息

Applications delta = null;

EurekaHttpResponse<Applications> httpResponse = eurekaTransport.queryClient.getDelta(remoteRegionsRef.get());

if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {

delta = httpResponse.getEntity();

}

//如果拉取失败进行全量拉取

if (delta == null) {

logger.warn("The server does not allow the delta revision to be applied because it is not safe. "

+ "Hence got the full registry.");

getAndStoreFullRegistry();

} else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration, currentUpdateGeneration + 1)) {

logger.debug("Got delta update with apps hashcode {}", delta.getAppsHashCode());

String reconcileHashCode = "";

if (fetchRegistryUpdateLock.tryLock()) {

try {

//跟新本地缓存

updateDelta(delta);

reconcileHashCode = getReconcileHashCode(applications);

} finally {

fetchRegistryUpdateLock.unlock();

}

} else {

logger.warn("Cannot acquire update lock, aborting getAndUpdateDelta");

}

// There is a diff in number of instances for some reason

if (!reconcileHashCode.equals(delta.getAppsHashCode()) || clientConfig.shouldLogDeltaDiff()) {

reconcileAndLogDifference(delta, reconcileHashCode); // this makes a remoteCall

}

} else {

logger.warn("Not updating application delta as another thread is updating it already");

logger.debug("Ignoring delta update with apps hashcode {}, as another thread is updating it already", delta.getAppsHashCode());

}

}

（重要步骤）服务注册

if (clientConfig.shouldRegisterWithEureka() && clientConfig.shouldEnforceRegistrationAtInit()) {

try {

if (!register() ) {

throw new IllegalStateException("Registration error at startup. Invalid server response.");

}

} catch (Throwable th) {

logger.error("Registration error at startup: {}", th.getMessage());

throw new IllegalStateException(th);

}

}

register()方法负责服务的注册

boolean register() throws Throwable {

logger.info(PREFIX + "{}: registering service...", appPathIdentifier);

EurekaHttpResponse<Void> httpResponse;

try {

//把自身的实例发送给服务端

httpResponse = eurekaTransport.registrationClient.register(instanceInfo);

} catch (Exception e) {

logger.warn(PREFIX + "{} - registration failed {}", appPathIdentifier, e.getMessage(), e);

throw e;

}

return httpResponse.getStatusCode() == 204;

}

(重要步骤) 定时任务

initScheduledTasks()是负责定时任务的相关方法。

private void initScheduledTasks() {

if (clientConfig.shouldFetchRegistry()) {

// 拉取服务默认30秒，eureka.client.register-fetch-interval-seconds

int registryFetchIntervalSeconds = clientConfig.getRegistryFetchIntervalSeconds();

int expBackOffBound = clientConfig.getCacheRefreshExecutorExponentialBackOffBound();

scheduler.schedule(

new TimedSupervisorTask(

"cacheRefresh",

scheduler,

cacheRefreshExecutor,

registryFetchIntervalSeconds,

TimeUnit.SECONDS,

expBackOffBound,

new CacheRefreshThread()

),

registryFetchIntervalSeconds, TimeUnit.SECONDS);

}

if (clientConfig.shouldRegisterWithEureka()) {

int renewalIntervalInSecs = instanceInfo.getLeaseInfo().getRenewalIntervalInSecs();

int expBackOffBound = clientConfig.getHeartbeatExecutorExponentialBackOffBound();

logger.info("Starting heartbeat executor: " + "renew interval is: {}", renewalIntervalInSecs);

// 心跳服务，默认30秒

scheduler.schedule(

new TimedSupervisorTask(

"heartbeat",

scheduler,

heartbeatExecutor,

renewalIntervalInSecs,

TimeUnit.SECONDS,

expBackOffBound,

new HeartbeatThread()

),

renewalIntervalInSecs, TimeUnit.SECONDS);

// InstanceInfo replicator

instanceInfoReplicator = new InstanceInfoReplicator(

this,

instanceInfo,

clientConfig.getInstanceInfoReplicationIntervalSeconds(),

2); // burstSize

statusChangeListener = new ApplicationInfoManager.StatusChangeListener() {

@Override

public String getId() {

return "statusChangeListener";

}

@Override

public void notify(StatusChangeEvent statusChangeEvent) {

if (InstanceStatus.DOWN == statusChangeEvent.getStatus() ||

InstanceStatus.DOWN == statusChangeEvent.getPreviousStatus()) {

// log at warn level if DOWN was involved

logger.warn("Saw local status change event {}", statusChangeEvent);

} else {

logger.info("Saw local status change event {}", statusChangeEvent);

}

instanceInfoReplicator.onDemandUpdate();

}

};

if (clientConfig.shouldOnDemandUpdateStatusChange()) {

applicationInfoManager.registerStatusChangeListener(statusChangeListener);

}

instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationIntervalSeconds());

} else {

logger.info("Not registering with Eureka server per configuration");

}

}

#### 服务下线

com.netflix.discovery.DiscoveryClient#shutdown

@PreDestroy

@Override

public synchronized void shutdown() {

if (isShutdown.compareAndSet(false, true)) {

logger.info("Shutting down DiscoveryClient ...");

if (statusChangeListener != null && applicationInfoManager != null) {

//注销状态监听器

applicationInfoManager.unregisterStatusChangeListener(statusChangeListener.getId());

}

//取消定时任务

cancelScheduledTasks();

// If APPINFO was registered

if (applicationInfoManager != null

&& clientConfig.shouldRegisterWithEureka()

&& clientConfig.shouldUnregisterOnShutdown()) {

applicationInfoManager.setInstanceStatus(InstanceStatus.DOWN);

unregister();

}

//关闭与server连接的客户端

if (eurekaTransport != null) {

eurekaTransport.shutdown();

}

//关闭相关监控

heartbeatStalenessMonitor.shutdown();

registryStalenessMonitor.shutdown();

logger.info("Completed shut down of DiscoveryClient");

}

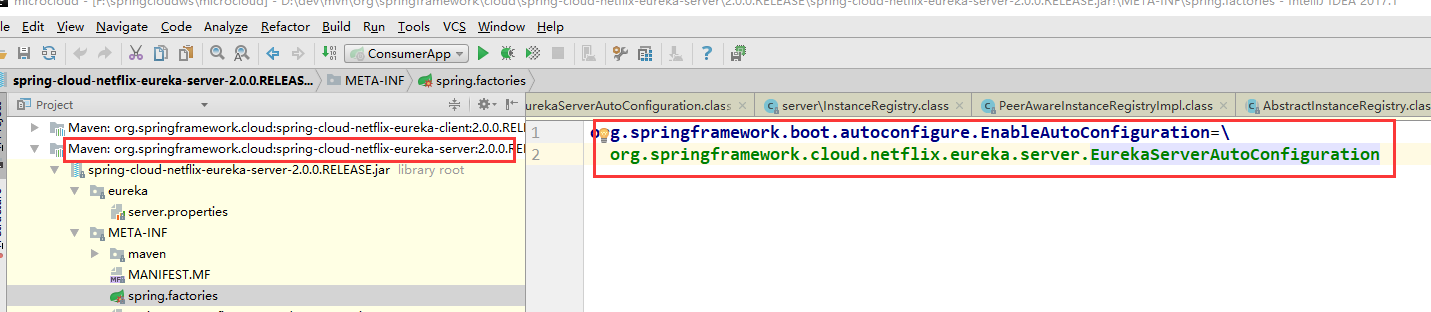
}

### EurekaServer

#### 总览

EurekaServer 是服务的注册中心，负责Eureka Client的相关信息注册，主要职责

* 服务注册
* 接受心跳服务
* 服务剔除
* 服务下线
* 集群同步



EurekaServerAutoConfiguration 是通过配置文件注册。

@Bean

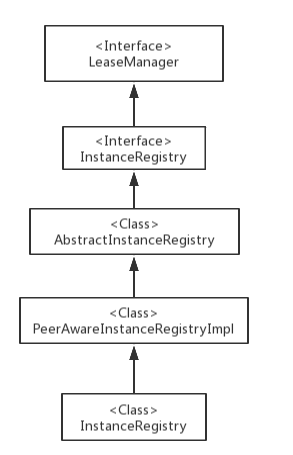
public PeerAwareInstanceRegistry peerAwareInstanceRegistry(ServerCodecs serverCodecs) {

this.eurekaClient.getApplications();

return new InstanceRegistry(this.eurekaServerConfig, this.eurekaClientConfig, serverCodecs, this.eurekaClient, this.instanceRegistryProperties.getExpectedNumberOfRenewsPerMin(), this.instanceRegistryProperties.getDefaultOpenForTrafficCount());

}

这里面有个InstanceRegistry就是重点需要关注的了



首先看下最上层的接口

public interface LeaseManager<T> {

//注册

void register(T var1, int var2, boolean var3);

//下线

boolean cancel(String var1, String var2, boolean var3);

//跟新

boolean renew(String var1, String var2, boolean var3);

//服务剔除

void evict();

}

PeerAwareInstanceRegistryImpl是一个子类的实现，在上面的基础上扩展对集群的同步操作，使Eureaka Server集群信息保持一致

#### 服务注册

com.netflix.eureka.registry.AbstractInstanceRegistry#register 这方法是负责服务的注册的。

public void register(InstanceInfo registrant, int leaseDuration, boolean isReplication) {

try {

//获取读锁

this.read.lock();

// gMap 其实可以发现，这里注册中心其实是个ConcurrentHashMap

Map<String, Lease<InstanceInfo>> gMap = (Map)this.registry.get(registrant.getAppName());

EurekaMonitors.REGISTER.increment(isReplication);

if(gMap == null) {

ConcurrentHashMap<String, Lease<InstanceInfo>> gNewMap = new ConcurrentHashMap();

//key 为appName,如果存在，返回存在的值，否则添加，返回null

gMap = (Map)this.registry.putIfAbsent(registrant.getAppName(), gNewMap);

if(gMap == null) {

gMap = gNewMap;

}

}

//根据instanceId获取实例的租约

Lease<InstanceInfo> existingLease = (Lease)((Map)gMap).get(registrant.getId());

if(existingLease != null && existingLease.getHolder() != null) {

Long existingLastDirtyTimestamp = ((InstanceInfo)existingLease.getHolder()).getLastDirtyTimestamp();

Long registrationLastDirtyTimestamp = registrant.getLastDirtyTimestamp();

//如果该实例的租约已经存在，比较最后的更新时间戳大小，取最大值的注册信息信息

if(existingLastDirtyTimestamp.longValue() > registrationLastDirtyTimestamp.longValue()) {

registrant = (InstanceInfo)existingLease.getHolder();

}

} else {

Object var6 = this.lock;

//如果租约不存在，注册一个新的实例

synchronized(this.lock) {

if(this.expectedNumberOfRenewsPerMin > 0) {

this.expectedNumberOfRenewsPerMin += 2;

this.numberOfRenewsPerMinThreshold = (int)((double)this.expectedNumberOfRenewsPerMin \* this.serverConfig.getRenewalPercentThreshold());

}

}

logger.debug("No previous lease information found; it is new registration");

}

//创建新的租约

Lease<InstanceInfo> lease = new Lease(registrant, leaseDuration);

if(existingLease != null) {

lease.setServiceUpTimestamp(existingLease.getServiceUpTimestamp());

}

//保存租约到map中

((Map)gMap).put(registrant.getId(), lease);

//获得最近注册队列

AbstractInstanceRegistry.CircularQueue var20 = this.recentRegisteredQueue;

synchronized(this.recentRegisteredQueue) {

this.recentRegisteredQueue.add(new Pair(Long.valueOf(System.currentTimeMillis()), registrant.getAppName() + "(" + registrant.getId() + ")"));

}

if(!InstanceStatus.UNKNOWN.equals(registrant.getOverriddenStatus())) {

logger.debug("Found overridden status {} for instance {}. Checking to see if needs to be add to the overrides", registrant.getOverriddenStatus(), registrant.getId());

if(!this.overriddenInstanceStatusMap.containsKey(registrant.getId())) {

logger.info("Not found overridden id {} and hence adding it", registrant.getId());

this.overriddenInstanceStatusMap.put(registrant.getId(), registrant.getOverriddenStatus());

}

}

InstanceStatus overriddenStatusFromMap = (InstanceStatus)this.overriddenInstanceStatusMap.get(registrant.getId());

if(overriddenStatusFromMap != null) {

logger.info("Storing overridden status {} from map", overriddenStatusFromMap);

registrant.setOverriddenStatus(overriddenStatusFromMap);

}

InstanceStatus overriddenInstanceStatus = this.getOverriddenInstanceStatus(registrant, existingLease, isReplication);

registrant.setStatusWithoutDirty(overriddenInstanceStatus);

if(InstanceStatus.UP.equals(registrant.getStatus())) {

lease.serviceUp();

}

registrant.setActionType(ActionType.ADDED);

this.recentlyChangedQueue.add(new AbstractInstanceRegistry.RecentlyChangedItem(lease));

registrant.setLastUpdatedTimestamp();

this.invalidateCache(registrant.getAppName(), registrant.getVIPAddress(), registrant.getSecureVipAddress());

} finally {

//释放锁

this.read.unlock();

}

}

#### 接受心跳服务

在Eureka Client完成服务的注册后，需要定时向Eureka Server发送心跳请求（默认30s）,维持自己在EurekaServer的租约有效性

public boolean renew(String appName, String id, boolean isReplication) {

EurekaMonitors.RENEW.increment(isReplication);

//根据appName获取服务集群租约集合

Map<String, Lease<InstanceInfo>> gMap = (Map)this.registry.get(appName);

Lease<InstanceInfo> leaseToRenew = null;

if(gMap != null) {

leaseToRenew = (Lease)gMap.get(id);

}

//如果租约不存在，直接返回false

if(leaseToRenew == null) {

EurekaMonitors.RENEW\_NOT\_FOUND.increment(isReplication);

logger.warn("DS: Registry: lease doesn't exist, registering resource: {} - {}", appName, id);

return false;

} else {

InstanceInfo instanceInfo = (InstanceInfo)leaseToRenew.getHolder();

if(instanceInfo != null) {

//得到服务的最终状态

InstanceStatus overriddenInstanceStatus = this.getOverriddenInstanceStatus(instanceInfo, leaseToRenew, isReplication);

if(overriddenInstanceStatus == InstanceStatus.UNKNOWN) {

//如果状态为UNKNOWN，取消续约

logger.info("Instance status UNKNOWN possibly due to deleted override for instance {}; re-register required", instanceInfo.getId());

EurekaMonitors.RENEW\_NOT\_FOUND.increment(isReplication);

return false;

}

if(!instanceInfo.getStatus().equals(overriddenInstanceStatus)) {

logger.info("The instance status {} is different from overridden instance status {} for instance {}. Hence setting the status to overridden status", new Object[]{instanceInfo.getStatus().name(), instanceInfo.getOverriddenStatus().name(), instanceInfo.getId()});

instanceInfo.setStatusWithoutDirty(overriddenInstanceStatus);

}

}

this.renewsLastMin.increment();

//跟新续约有效时间

leaseToRenew.renew();

return true;

}

}

#### 服务剔除

如果Eureka Client在注册后，由于服务的崩溃或网络异常导致既没有续约，也没有下线，那么服务就处于不可知的状态，需要剔除这些服务

com.netflix.eureka.registry.AbstractInstanceRegistry#evict(long)

这是个定时任务调用的方法

com.netflix.eureka.registry.AbstractInstanceRegistry#postInit中使用

AbstractInstanceRegistry.EvictionTask 负责调用(默认60s)

public void evict(long additionalLeaseMs) {

logger.debug("Running the evict task");

//如果自我保护状态，不允许剔除服务

if(!this.isLeaseExpirationEnabled()) {

logger.debug("DS: lease expiration is currently disabled.");

} else {

List<Lease<InstanceInfo>> expiredLeases = new ArrayList();

//遍历注册表registry，获取所有过期的租约

Iterator var4 = this.registry.entrySet().iterator();

while(true) {

Map leaseMap;

do {

if(!var4.hasNext()) {

//获取注册表租约总数

int registrySize = (int)this.getLocalRegistrySize();

int registrySizeThreshold = (int)((double)registrySize \* this.serverConfig.getRenewalPercentThreshold());

//计算最多允许剔除的阈值

int evictionLimit = registrySize - registrySizeThreshold;

//两者中取小的值，为本常剔除的数量

int toEvict = Math.min(expiredLeases.size(), evictionLimit);

if(toEvict > 0) {

logger.info("Evicting {} items (expired={}, evictionLimit={})", new Object[]{Integer.valueOf(toEvict), Integer.valueOf(expiredLeases.size()), Integer.valueOf(evictionLimit)});

Random random = new Random(System.currentTimeMillis());

//逐个剔除

for(int i = 0; i < toEvict; ++i) {

int next = i + random.nextInt(expiredLeases.size() - i);

Collections.swap(expiredLeases, i, next);

Lease<InstanceInfo> lease = (Lease)expiredLeases.get(i);

String appName = ((InstanceInfo)lease.getHolder()).getAppName();

String id = ((InstanceInfo)lease.getHolder()).getId();

EurekaMonitors.EXPIRED.increment();

logger.warn("DS: Registry: expired lease for {}/{}", appName, id);

//剔除

this.internalCancel(appName, id, false);

}

}

return;

}

Entry<String, Map<String, Lease<InstanceInfo>>> groupEntry = (Entry)var4.next();

leaseMap = (Map)groupEntry.getValue();

} while(leaseMap == null);

Iterator var7 = leaseMap.entrySet().iterator();

while(var7.hasNext()) {

Entry<String, Lease<InstanceInfo>> leaseEntry = (Entry)var7.next();

Lease<InstanceInfo> lease = (Lease)leaseEntry.getValue();

if(lease.isExpired(additionalLeaseMs) && lease.getHolder() != null) {

expiredLeases.add(lease);

}

}

}

}

}

#### 服务下线

EurekaClient在应用销毁时候，会向Eureka Server发送下线请求

对于服务端的服务下线，其主要代码对应在

com.netflix.eureka.registry.AbstractInstanceRegistry#cancel

public boolean cancel(String appName, String id, boolean isReplication) {

return this.internalCancel(appName, id, isReplication);

}

protected boolean internalCancel(String appName, String id, boolean isReplication) {

boolean var10;

try {

//读锁，防止被其他线程进行修改

this.read.lock();

EurekaMonitors.CANCEL.increment(isReplication);

//根据appName获取服务实例集群

Map<String, Lease<InstanceInfo>> gMap = (Map)this.registry.get(appName);

Lease<InstanceInfo> leaseToCancel = null;

//移除服务实例租约

if(gMap != null) {

leaseToCancel = (Lease)gMap.remove(id);

}

AbstractInstanceRegistry.CircularQueue var6 = this.recentCanceledQueue;

synchronized(this.recentCanceledQueue) {

this.recentCanceledQueue.add(new Pair(Long.valueOf(System.currentTimeMillis()), appName + "(" + id + ")"));

}

InstanceStatus instanceStatus = (InstanceStatus)this.overriddenInstanceStatusMap.remove(id);

if(instanceStatus != null) {

logger.debug("Removed instance id {} from the overridden map which has value {}", id, instanceStatus.name());

}

//租约不存在，返回false

if(leaseToCancel == null) {

EurekaMonitors.CANCEL\_NOT\_FOUND.increment(isReplication);

logger.warn("DS: Registry: cancel failed because Lease is not registered for: {}/{}", appName, id);

boolean var17 = false;

return var17;

}

//设置租约的下线时间

leaseToCancel.cancel();

InstanceInfo instanceInfo = (InstanceInfo)leaseToCancel.getHolder();

String vip = null;

String svip = null;

if(instanceInfo != null) {

instanceInfo.setActionType(ActionType.DELETED);

this.recentlyChangedQueue.add(new AbstractInstanceRegistry.RecentlyChangedItem(leaseToCancel));

instanceInfo.setLastUpdatedTimestamp();

vip = instanceInfo.getVIPAddress();

svip = instanceInfo.getSecureVipAddress();

}

//设置缓存过期

this.invalidateCache(appName, vip, svip);

logger.info("Cancelled instance {}/{} (replication={})", new Object[]{appName, id, Boolean.valueOf(isReplication)});

var10 = true;

} finally {

//释放锁

this.read.unlock();

}

return var10;

}

#### 集群同步

如果Eureka Server是通过集群方式进行部署，为了为维护整个集群中注册表数据一致性所以集群同步也是非常重要得事情。

集群同步分为两部分

1. EurekaServer在启动过程中从他的peer节点中拉取注册表信息，并讲这些服务实例注册到本地注册表中；
2. 另一部分是eureka server每次对本地注册表进行操作时，同时会讲操作同步到他的peer节点中，达到数据一致；

Eureka Server初始化本地注册表信息

在eureka server启动过程中，会从它的peer节点中拉取注册表来初始化本地注册表，这部分主要通过

com.netflix.eureka.registry.PeerAwareInstanceRegistryImpl#syncUp

他从可能存在的peer节点中，拉取peer节点中的注册表信息，并将其中的服务实例的信息注册到本地注册表中。

public int syncUp() {

// Copy entire entry from neighboring DS node

int count = 0;

for (int i = 0; ((i < serverConfig.getRegistrySyncRetries()) && (count == 0)); i++) {

if (i > 0) {

try {

//根据配置休停下

Thread.sleep(serverConfig.getRegistrySyncRetryWaitMs());

} catch (InterruptedException e) {

logger.warn("Interrupted during registry transfer..");

break;

}

}

//获取所有的服务实例

Applications apps = eurekaClient.getApplications();

for (Application app : apps.getRegisteredApplications()) {

for (InstanceInfo instance : app.getInstances()) {

try {

//判断是否可以注册

if (isRegisterable(instance)) {

//注册到自身的注册表中

register(instance, instance.getLeaseInfo().getDurationInSecs(), true);

count++;

}

} catch (Throwable t) {

logger.error("During DS init copy", t);

}

}

}

}

return count;

}

通过这一步保证了eureka启动时的数据一致性

Eureka Server之间注册表信息同步复制

为了保证Eureka Server集群运行时候注册表的信息一致性，每个eureka server在对本地注册表进行管理操作时，会将相应的信息同步到peer节点中。

com.netflix.eureka.registry.PeerAwareInstanceRegistryImpl#cancel

com.netflix.eureka.registry.PeerAwareInstanceRegistryImpl#register

com.netflix.eureka.registry.PeerAwareInstanceRegistryImpl#renew

等方法中，都回调用方法

com.netflix.eureka.registry.PeerAwareInstanceRegistryImpl#replicateToPeers

private void replicateToPeers(Action action, String appName, String id,

InstanceInfo info /\* optional \*/,

InstanceStatus newStatus /\* optional \*/, boolean isReplication) {

Stopwatch tracer = action.getTimer().start();

try {

if (isReplication) {

numberOfReplicationsLastMin.increment();

}

// If it is a replication already, do not replicate again as this will create a poison replication

if (peerEurekaNodes == Collections.EMPTY\_LIST || isReplication) {

return;

}

//向peer集群中的每一个peer进行同步

for (final PeerEurekaNode node : peerEurekaNodes.getPeerEurekaNodes()) {

// If the url represents this host, do not replicate to yourself.

if (peerEurekaNodes.isThisMyUrl(node.getServiceUrl())) {

continue;

}

//根据action调用不同的同步请求

replicateInstanceActionsToPeers(action, appName, id, info, newStatus, node);

}

} finally {

tracer.stop();

}

}

# Ribbon负载均衡

现在服务提供方已经可以通过Eureka进行注册了，但对于服务的消费者，目前并没有处理，对于服务的消费方，也应该连接上eureka，进行服务的获取，这个时候就应该使用Ribbon这个组件了。

ribbon对应的pom文件如下

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

注意：如果是Edgware或之前的版本，ribbon文件如下

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency

在用到负载均衡之前，先要到eureka中获取相关的服务，所以我们这一块依然需要用到eureka，但eureka中已经内置集成了ribbon，所以在pom文件中，并不需要显示引入ribbon的依赖

## Ribbon基本使用

【microcloud-consumer】 修改pom文件，增加eureka的支持

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-consumer</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-consumer】 修改RestConfig配置类，在获取RestTemplate对象的时候加入Ribbon的配置信息

package cn.enjoy.config;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpHeaders;

import org.springframework.web.client.RestTemplate;

import java.nio.charset.Charset;

import java.util.Base64;

@Configuration

public class RestConfig {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public HttpHeaders getHeaders() { // 要进行一个Http头信息配置

HttpHeaders headers = new HttpHeaders(); // 定义一个HTTP的头信息

String auth = "root:enjoy"; // 认证的原始信息

byte[] encodedAuth = Base64.getEncoder()

.encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

String authHeader = "Basic " + new String(encodedAuth);

headers.set("Authorization", authHeader);

return headers;

}

}

【microcloud-consumer】修改application.yml文件，增加Eureka服务注册相关信息

server:

port: 80

eureka:

client:

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

【microcloud-consumer】修改项目启动类，增加Eureka客户端的配置注解

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class ConsumerApp {

public static void main(String[] args) {

SpringApplication.run(ConsumerApp.class,args);

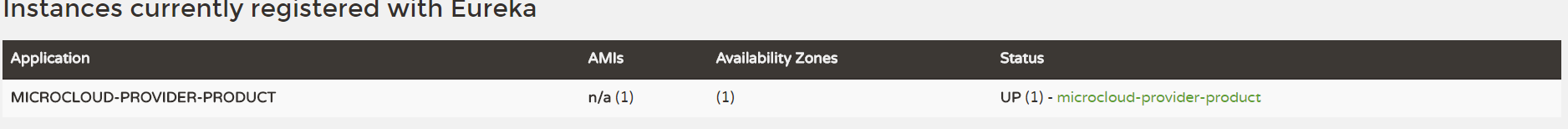
}

}

【microcloud-consumer】 修改ConsumerProductController控制器

现在在eureka中注册的服务名称都是大写字母：

**MICROCLOUD-PROVIDER-PRODUCT**



package cn.enjoy.controller;

import cn.enjoy.vo.Product;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.util.List;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

public static final String PRODUCT\_GET\_URL = "http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/get/";

public static final String PRODUCT\_LIST\_URL="http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/list/";

public static final String PRODUCT\_ADD\_URL = "http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/add/";

@Resource

private RestTemplate restTemplate;

@Resource

private HttpHeaders httpHeaders;

@RequestMapping("/product/get")

public Object getProduct(long id) {

Product product = restTemplate.exchange(PRODUCT\_GET\_URL + id,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), Product.class).getBody();

return product;

}

@RequestMapping("/product/list")

public Object listProduct() {

List<Product> list = restTemplate.exchange(PRODUCT\_LIST\_URL,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), List.class).getBody();

return list;

}

@RequestMapping("/product/add")

public Object addPorduct(Product product) {

Boolean result = restTemplate.exchange(PRODUCT\_ADD\_URL, HttpMethod.POST,new HttpEntity<Object>(product,httpHeaders), Boolean.class).getBody();

return result;

}

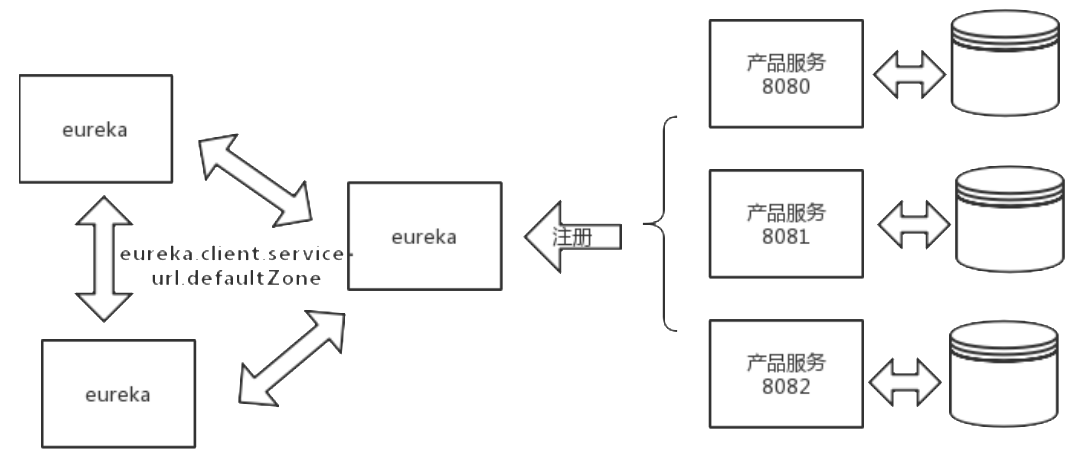
}

访问地址：<http://localhost/consumer/product/list>

这个时候Ribbon与Eureka已经整合成功

## Ribbon负载均衡的实现

通过上面的代码发现我们用到了一个注解@LoadBalanced，根据这名字大概就能知道Ribbon是可以实现负载均衡的。



【microcloud-provider-product】 复制两份

分别为【microcloud-provider-product2】与【microcloud-provider-product3】

【springcloud数据库】复制两份

分别为【springcloud2数据库】【springcloud3数据库】 里面分别执行spingcloud数据库的脚本

【microcloud-provider-product2】修改application.yml文件如下

server:

port: 8081

mybatis:

mapper-locations: # 所有的mapper映射文件

- classpath:mapping/\*.xml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource # 配置当前要使用的数据源的操作类型

driver-class-name: com.mysql.cj.jdbc.Driver # 配置MySQL的驱动程序类

url: jdbc:mysql://localhost:3306/springcloud2?serverTimezone=GMT%2B8 # 数据库连接地址

username: root # 数据库用户名

password: root1234% # 数据库连接密码

application:

name: microcloud-provider-product

# security:

# user:

# roles:

# - USER # 授权角色

# name: root

# password: enjoy

logging:

level:

cn.enjoy.mapper: debug

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

#defaultZone: http://admin:enjoy@localhost:7001/eureka

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance:

instance-id: microcloud-provider-product2

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

info:

app.name: microcloud-provider-product2

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

【microcloud-provider-product3】修改application.yml文件如下

server:

port: 8082

mybatis:

mapper-locations: # 所有的mapper映射文件

- classpath:mapping/\*.xml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource # 配置当前要使用的数据源的操作类型

driver-class-name: com.mysql.cj.jdbc.Driver # 配置MySQL的驱动程序类

url: jdbc:mysql://localhost:3306/springcloud3?serverTimezone=GMT%2B8 # 数据库连接地址

username: root # 数据库用户名

password: root1234% # 数据库连接密码

application:

name: microcloud-provider-product

# security:

# user:

# roles:

# - USER # 授权角色

# name: root

# password: enjoy

logging:

level:

cn.enjoy.mapper: debug

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

#defaultZone: http://admin:enjoy@localhost:7001/eureka

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance:

instance-id: microcloud-provider-product3

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

info:

app.name: microcloud-provider-product3

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

分别启动3个服务提供方，访问

[http://localhost:8080/prodcut/get/1](http://localhost:8082/prodcut/get/1)

[http://localhost:8081/prodcut/get/1](http://localhost:8082/prodcut/get/1)

<http://localhost:8082/prodcut/get/1>

确认3个服务是能正确提供访问的

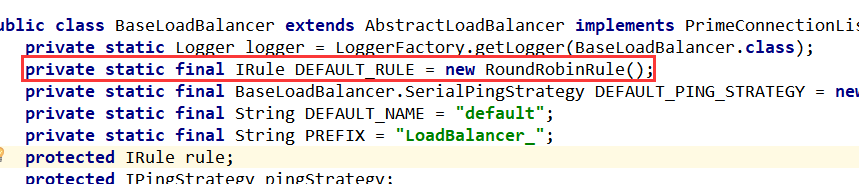
【microcloud-consumer】启动

访问：<http://localhost/consumer/product/list>

现在发现每一次获取数据都是通过不同的微服务获得的，所以现在同一个消费端就可以通过 Ribbon 实现了负载均衡配置处理。

## 自定义Ribbon路由

前面已经使用Ribbon实现了路由，通过测试，也不难发现默认Ribbon使用的路由策略是轮询，可以看下源代码BaseLoadBalancer



### 全局路由配置

这种负载均衡的策略其实也是可以由用户来修改的，如果想要去修改，可以使用自定义的LoadBalance

【microcloud-consumer】 修改RestConfig

package cn.enjoy.config;

import com.netflix.loadbalancer.IRule;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpHeaders;

import org.springframework.web.client.RestTemplate;

import java.nio.charset.Charset;

import java.util.Base64;

@Configuration

public class RestConfig {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public HttpHeaders getHeaders() { // 要进行一个Http头信息配置

HttpHeaders headers = new HttpHeaders(); // 定义一个HTTP的头信息

String auth = "root:enjoy"; // 认证的原始信息

byte[] encodedAuth = Base64.getEncoder()

.encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

String authHeader = "Basic " + new String(encodedAuth);

headers.set("Authorization", authHeader);

return headers;

}

@Bean

public IRule ribbonRule() { // 其中IRule就是所有规则的标准

return new com.netflix.loadbalancer.RandomRule(); // 随机的访问策略

}

}

这个时候重启测试发现，默认的路由规则已经变成了随机

### 单独设置某个Ribbon的路由

有时候，某个消费者可能需要访问多个多个服务提供方，而希望每个服务提供方提供的路由规则并不相同，这个时候就不能让Spring扫描到IRULE，需要通过@RibbonClient 来指定服务于配置的关系

【microcloud-consumer】 修改RestConfig，删除IRULE

package cn.enjoy.config;

import com.netflix.loadbalancer.IRule;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpHeaders;

import org.springframework.web.client.RestTemplate;

import java.nio.charset.Charset;

import java.util.Base64;

@Configuration

public class RestConfig {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public HttpHeaders getHeaders() { // 要进行一个Http头信息配置

HttpHeaders headers = new HttpHeaders(); // 定义一个HTTP的头信息

String auth = "root:enjoy"; // 认证的原始信息

byte[] encodedAuth = Base64.getEncoder()

.encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

String authHeader = "Basic " + new String(encodedAuth);

headers.set("Authorization", authHeader);

return headers;

}

~~@Bean~~

~~public IRule ribbonRule() { // 其中IRule就是所有规则的标准~~

~~return new com.netflix.loadbalancer.RandomRule(); // 随机的访问策略~~

~~}~~

}

【microcloud-consumer】新增一个路由规则的配置类，注意这个类不应该放到SpringCloud扫描不到的位置，否则又回变成全局的IRULE，所以这个时候应该单独使用一个新的包，着个包和启动并不在同一个包下

package cn.xiangxue.config;

import com.netflix.loadbalancer.IRule;

import org.springframework.context.annotation.Bean;

public class RibbonConfig {

@Bean

public IRule ribbonRule() { // 其中IRule就是所有规则的标准

return new com.netflix.loadbalancer.RandomRule(); // 随机的访问策略

}

}

【microcloud-consumer】 修改启动类，使用@RibbonClient指定配置类

package cn.enjoy;

import cn.xiangxue.config.RibbonConfig;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.ribbon.RibbonClient;

import org.springframework.cloud.netflix.ribbon.RibbonClients;

@SpringBootApplication

@EnableEurekaClient

@RibbonClient(name ="MICROCLOUD-PROVIDER-PRODUCT" ,configuration = RibbonConfig.class)

public class ConsumerApp {

public static void main(String[] args) {

SpringApplication.run(ConsumerApp.class,args);

}

}

这里的name 只服务的名称，如果需要有多个服务提供方，这个时候可以使用@RibbonClients进行配置

### 服务提供方的信息获取

在服务的消费方，也是可以获取到服务提供方的具体信息

【microcloud-consumer】修改ConsumerProductController

package cn.enjoy.controller;

import cn.enjoy.vo.Product;

import org.springframework.cloud.client.ServiceInstance;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.util.List;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

public static final String PRODUCT\_GET\_URL = "http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/get/";

public static final String PRODUCT\_LIST\_URL="http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/list/";

public static final String PRODUCT\_ADD\_URL = "http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/add/";

@Resource

private RestTemplate restTemplate;

@Resource

private HttpHeaders httpHeaders;

@Resource

private LoadBalancerClient loadBalancerClient;

@RequestMapping("/product/get")

public Object getProduct(long id) {

Product product = restTemplate.exchange(PRODUCT\_GET\_URL + id,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), Product.class).getBody();

return product;

}

@RequestMapping("/product/list")

public Object listProduct() {

ServiceInstance serviceInstance = this.loadBalancerClient.choose("MICROCLOUD-PROVIDER-PRODUCT") ;

System.out.println(

"【\*\*\* ServiceInstance \*\*\*】host = " + serviceInstance.getHost()

+ "、port = " + serviceInstance.getPort()

+ "、serviceId = " + serviceInstance.getServiceId());

List<Product> list = restTemplate.exchange(PRODUCT\_LIST\_URL,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), List.class).getBody();

return list;

}

@RequestMapping("/product/add")

public Object addPorduct(Product product) {

Boolean result = restTemplate.exchange(PRODUCT\_ADD\_URL, HttpMethod.POST,new HttpEntity<Object>(product,httpHeaders), Boolean.class).getBody();

return result;

}

}

## 脱离Eureka使用Ribbon

之前所用Ribbon都是从Eureka中获取服务并通过@LoadBalanced来实现负载均衡的，其实Ribbon也可以脱离Eureka来使用

复制【microcloud-consumer】 成一个新的模块【microcloud-consumer-ribbon】

【microcloud-consumer-ribbon】 修改pom文件，删除eureka的依赖添加ribbon的依赖

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-consumer-ribbon</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<!--<dependency>-->

<!--<groupId>org.springframework.cloud</groupId>-->

<!--<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>-->

<!--</dependency>-->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-consumer-ribbon】 修改application.yml配置文件

server:

port: 80

ribbon:

eureka:

enabled: false

MICROCLOUD-PROVIDER-PRODUCT:

ribbon:

listOfServers: http://localhost:8080,http://localhost:8081,http://localhost:8082

【microcloud-consumer-ribbon】 修改 RestConfig，删除@LoadBalanced注解

package cn.enjoy.config;

import com.netflix.loadbalancer.IRule;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpHeaders;

import org.springframework.web.client.RestTemplate;

import java.nio.charset.Charset;

import java.util.Base64;

@Configuration

public class RestConfig {

@Bean

//@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public HttpHeaders getHeaders() { // 要进行一个Http头信息配置

HttpHeaders headers = new HttpHeaders(); // 定义一个HTTP的头信息

String auth = "root:enjoy"; // 认证的原始信息

byte[] encodedAuth = Base64.getEncoder()

.encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

String authHeader = "Basic " + new String(encodedAuth);

headers.set("Authorization", authHeader);

return headers;

}

}

【microcloud-consumer-ribbon】修改ConsumerProductController，修改服务的调用URI

package cn.enjoy.controller;

import cn.enjoy.vo.Product;

import cn.xiangxue.config.RibbonConfig;

import org.springframework.cloud.client.ServiceInstance;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;

import org.springframework.cloud.netflix.ribbon.RibbonClient;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.net.URI;

import java.util.List;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

public static final String PRODUCT\_TOPIC = "MICROCLOUD-PROVIDER-PRODUCT";

@Resource

private RestTemplate restTemplate;

@Resource

private HttpHeaders httpHeaders;

@Resource

private LoadBalancerClient loadBalancerClient;

@RequestMapping("/product/list")

public Object listProduct() {

ServiceInstance serviceInstance = this.loadBalancerClient.choose(PRODUCT\_TOPIC) ;

System.out.println(

"【\*\*\* ServiceInstance \*\*\*】host = " + serviceInstance.getHost()

+ "、port = " + serviceInstance.getPort()

+ "、serviceId = " + serviceInstance.getServiceId());

URI uri = URI.create(String.format("http://%s:%s/prodcut/list/" ,

serviceInstance.getHost(), serviceInstance.getPort()));

List<Product> list = restTemplate.exchange(uri,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), List.class).getBody();

return list;

}

}

【microcloud-consumer-ribbon】启动

访问：<http://localhost/consumer/product/list>

## 原理解析

### @RibbonClient注解

@RibbonClient注解可以实现Ribbon客户端，ribbon需要设置客户端的名称，以及相关的路由配置类

@Configuration

@Import(RibbonClientConfigurationRegistrar.class)

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface RibbonClient {

String value() default "";

String name() default "";

Class<?>[] configuration() default {};

}

value和name是等价了，用于设置客户端的实例名称，而configuration用于指定配置类

接下来还导入了个RibbonClientConfigurationRegistrar，他实现了ImportBeanDefinitionRegistrar，他也不用多说，是spring的工具接口，用于spring动态注册BeanDefinition的接口，在这是用于注册Ribbon所需的BeanDefinition（比如Ribbon客户端实例）

org.springframework.cloud.netflix.ribbon.RibbonClientConfigurationRegistrar#registerBeanDefinitions

@Override

public void registerBeanDefinitions(AnnotationMetadata metadata,

BeanDefinitionRegistry registry) {

//获取@RibbonClient的参数，获取clientName后进行configuraction注册

Map<String, Object> client = metadata.getAnnotationAttributes(

RibbonClient.class.getName(), true);

//获取ribbonclient的value或者name属性

String name = getClientName(client);

if (name != null) {

//注册

registerClientConfiguration(registry, name, client.get("configuration"));

}

}

这里注册会调用

private void registerClientConfiguration(BeanDefinitionRegistry registry,

Object name, Object configuration) {

BeanDefinitionBuilder builder = BeanDefinitionBuilder

.genericBeanDefinition(RibbonClientSpecification.class);

builder.addConstructorArgValue(name);

builder.addConstructorArgValue(configuration);

registry.registerBeanDefinition(name + ".RibbonClientSpecification",

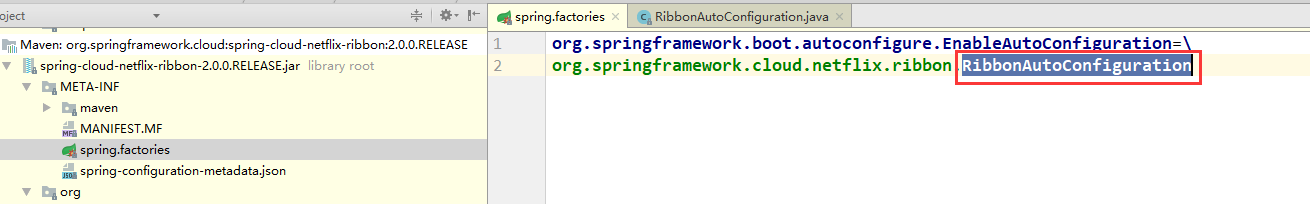
builder.getBeanDefinition());

}

registerClientConfiguration会注册一个RibbonClientSpecification的bean,名称是ribbon的名称加上.RibbonClientSpecification

RibbonClientSpecification 实现了 NamedContextFactory.Specification，是提供给SpringClientFactory使用的，他用于初始化ribbon的相关实例使用

SpringClientFactory在哪？



org.springframework.cloud.netflix.ribbon.RibbonAutoConfiguration

@Bean

public SpringClientFactory springClientFactory() {

SpringClientFactory factory = new SpringClientFactory();

factory.setConfigurations(this.configurations);

return factory;

}

@Bean

@ConditionalOnMissingBean(LoadBalancerClient.class)

public LoadBalancerClient loadBalancerClient() {

return new RibbonLoadBalancerClient(springClientFactory());

}

那么最重要的就回到了LoadBalancerClient ，他是Ribbon项目最核心的类

org.springframework.cloud.client.loadbalancer.LoadBalancerClient

public interface LoadBalancerClient extends ServiceInstanceChooser {

//从servericeId 所代表的服务列表中选择一个服务器来发送网络请求

<T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException;

<T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T> request) throws IOException;

//构建网络请求URI

URI reconstructURI(ServiceInstance instance, URI original);

}

### LoadBalancerClient

LoadBalancerClient 还继承了一个接口ServiceInstanceChooser

org.springframework.cloud.client.loadbalancer.ServiceInstanceChooser

public interface ServiceInstanceChooser {

//根据serviceId从服务器列表中选择一个ServiceInstance

ServiceInstance choose(String serviceId);

}

org.springframework.cloud.netflix.ribbon.RibbonLoadBalancerClient 是LoadBalancerClient 的实现类

@Override

public <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException {

//每次发送请求都回获取一个ILoadBalancer ，会涉及负载均衡（IRULS），服务器列表集群（ServerList） 和检验服务是否存活（IPing）等细节实现

ILoadBalancer loadBalancer = getLoadBalancer(serviceId);

Server server = getServer(loadBalancer);

if (server == null) {

throw new IllegalStateException("No instances available for " + serviceId);

}

RibbonServer ribbonServer = new RibbonServer(serviceId, server, isSecure(server,

serviceId), serverIntrospector(serviceId).getMetadata(server));

return execute(serviceId, ribbonServer, request);

}

另外

org.springframework.cloud.netflix.ribbon.RibbonLoadBalancerClient#getServer(com.netflix.loadbalancer.ILoadBalancer)

protected Server getServer(ILoadBalancer loadBalancer) {

if (loadBalancer == null) {

return null;

}

return loadBalancer.chooseServer("default");

}

这方法直接调用了ILoadBalancer 的chooseServer方法来使用负载君和策略，从已知的服务列表中选出一个服务器实例

来接下来重点就到了ILoadBalancer 这接口了

他定义负载君和操作的接口，由前面说过的SpringClientFactory获得而来。

而SpringClientFactory又是再RibbonAutoConfiguration定义

ILoadBalancer

com.netflix.loadbalancer.ZoneAwareLoadBalancer 是ILoadBalancer 的具体实现

看下他构造方法

public ZoneAwareLoadBalancer(IClientConfig clientConfig, IRule rule, IPing ping, ServerList<T> serverList, ServerListFilter<T> filter) {

super(clientConfig, rule, ping, serverList, filter);

}

**IClientConfig**

client的配置类，具体指的DefaultClientConfigImpl

**IRule**

负载均衡的策略类,默认轮询 RoundRobinRule

**IPing**

服务可用性检查，默认DummyPing

**ServerList**

服务列表获取，ConfigurationBasedServerList

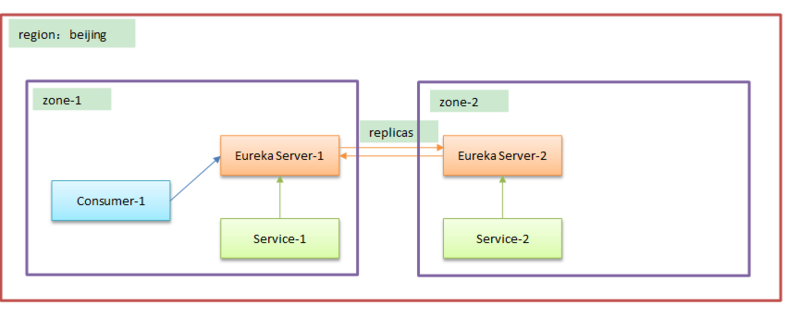
**ServerListFilter**

服务列表过滤 ZonePreferenceServerListFilter

ZoneAwareLoadBalancer其中一个重要的方法就是

com.netflix.loadbalancer.ZoneAwareLoadBalancer#chooseServer

扩展阅读：<https://www.cnblogs.com/junjiang3/p/9061867.html>

dd

@Override

public Server chooseServer(Object key) {

//如果就一个zone，直接返回

if (!ENABLED.get() || getLoadBalancerStats().getAvailableZones().size() <= 1) {

logger.debug("Zone aware logic disabled or there is only one zone");

return super.chooseServer(key);

}

Server server = null;

try {

//获取当前有关负载均衡的服务器状态集合

LoadBalancerStats lbStats = getLoadBalancerStats();

Map<String, ZoneSnapshot> zoneSnapshot = ZoneAvoidanceRule.createSnapshot(lbStats);

logger.debug("Zone snapshots: {}", zoneSnapshot);

//获取平均负载阈值

if (triggeringLoad == null) {

triggeringLoad = DynamicPropertyFactory.getInstance().getDoubleProperty(

"ZoneAwareNIWSDiscoveryLoadBalancer." + this.getName() + ".triggeringLoadPerServerThreshold", 0.2d);

}

//获取平均实例故障率阈值

if (triggeringBlackoutPercentage == null) {

triggeringBlackoutPercentage = DynamicPropertyFactory.getInstance().getDoubleProperty(

"ZoneAwareNIWSDiscoveryLoadBalancer." + this.getName() + ".avoidZoneWithBlackoutPercetage", 0.99999d);

}

//根据两个阈值来获取所有可用的服务列表

Set<String> availableZones = ZoneAvoidanceRule.getAvailableZones(zoneSnapshot, triggeringLoad.get(), triggeringBlackoutPercentage.get());

logger.debug("Available zones: {}", availableZones);

if (availableZones != null && availableZones.size() < zoneSnapshot.keySet().size()) {

//随机从可用服务区列表中选择一个服务器

String zone = ZoneAvoidanceRule.randomChooseZone(zoneSnapshot, availableZones);

logger.debug("Zone chosen: {}", zone);

if (zone != null) {

//得到zoone对应的BaseLoadBalancer

BaseLoadBalancer zoneLoadBalancer = getLoadBalancer(zone);

server = zoneLoadBalancer.chooseServer(key);

}

}

} catch (Exception e) {

logger.error("Error choosing server using zone aware logic for load balancer={}", name, e);

}

if (server != null) {

return server;

} else {

logger.debug("Zone avoidance logic is not invoked.");

return super.chooseServer(key);

}

}

com.netflix.loadbalancer.BaseLoadBalancer#chooseServer

public Server chooseServer(Object key) {

if (counter == null) {

counter = createCounter();

}

counter.increment();

if (rule == null) {

return null;

} else {

try {

//根据具体的路由算法获取服务

return rule.choose(key);

} catch (Exception e) {

logger.warn("LoadBalancer [{}]: Error choosing server for key {}", name, key, e);

return null;

}

}

}

### IRULE负载均衡的实现

具体服务的的负载还是由IRULE实现，前面在入门部分也讲过怎么更换具体的路由实现方法

IRULE具体在RibbonClientConfiguration进行配置，IRule的接口choose 方法负责选区一个具体的服务。

@Bean

@ConditionalOnMissingBean

public IRule ribbonRule(IClientConfig config) {

if (this.propertiesFactory.isSet(IRule.class, name)) {

return this.propertiesFactory.get(IRule.class, config, name);

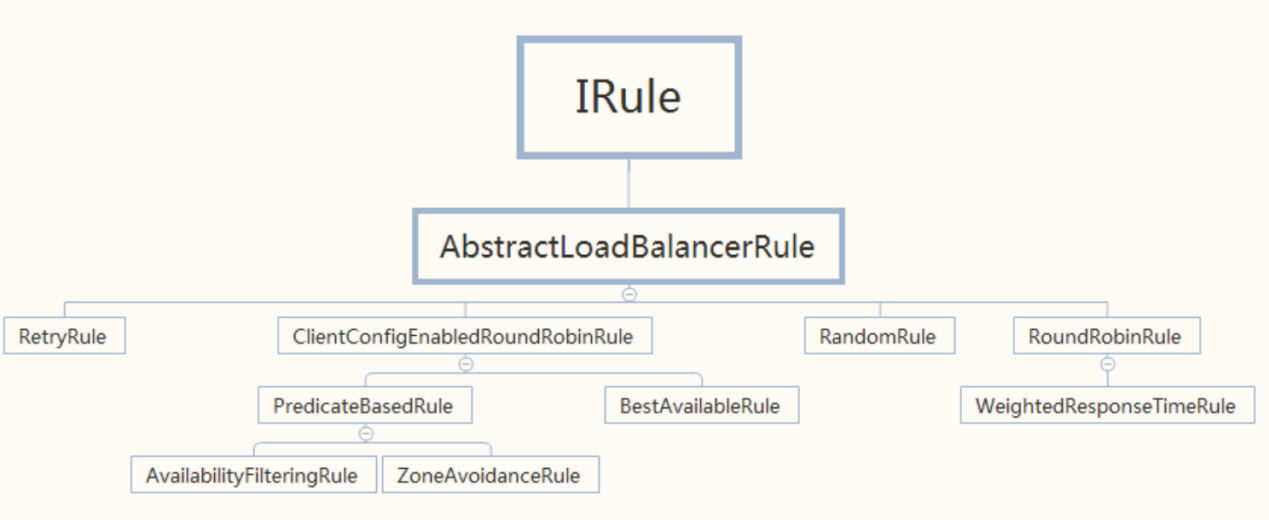
}

ZoneAvoidanceRule rule = new ZoneAvoidanceRule();

rule.initWithNiwsConfig(config);

return rule;

}



BestAvailableRule :选择最小请求数的服务器

RoundRobinRule：轮询

ClientConfigEnabledRoundRobinRule: 使用RoundRobinRule选择服务器

RetryRule： 根据选的轮询的方式重试

WeightedResponseTimeRule： 根据响应时间去计算一个权重weight ，weight越低，被选择的可能性就越低

ZoneAvoidanceRule： 根据server的zone区域和可用性来轮询选择。

#### ZoneAvoidanceRule 的具体实现

ZoneAvoidanceRule是默认的IRule实例，他使用PredicateBasedRule来根据服务区的运行状况和服务器的可用性来选择服务器

它的父类是PredicateBasedRule

com.netflix.loadbalancer.PredicateBasedRule

具体依次做了以下事情

1. 先使用ILoadBalancer 获取服务器列表
2. 使用AbstractServerPredicate进行服务器过滤
3. 最后轮询从剩余的服务器列表中选择最终的服务器

public abstract class PredicateBasedRule extends ClientConfigEnabledRoundRobinRule {

public abstract AbstractServerPredicate getPredicate();

@Override

public Server choose(Object key) {

ILoadBalancer lb = getLoadBalancer();

Optional<Server> server = getPredicate().chooseRoundRobinAfterFiltering(lb.getAllServers(), key);

if (server.isPresent()) {

return server.get();

} else {

return null;

}

}

}

com.netflix.loadbalancer.PredicateBasedRule#getPredicate 又是一个抽象的实现

具体实现

com.netflix.loadbalancer.ZoneAvoidanceRule#getPredicate

public ZoneAvoidanceRule() {

super();

ZoneAvoidancePredicate zonePredicate = new ZoneAvoidancePredicate(this);

AvailabilityPredicate availabilityPredicate = new AvailabilityPredicate(this);

compositePredicate = createCompositePredicate(zonePredicate, availabilityPredicate);

}

//将两个Predicate组合成一个CompositePredicate

private CompositePredicate createCompositePredicate(ZoneAvoidancePredicate p1, AvailabilityPredicate p2) {

return CompositePredicate.withPredicates(p1, p2)

.addFallbackPredicate(p2)

.addFallbackPredicate(AbstractServerPredicate.alwaysTrue())

.build();

}

ZoneAvoidancePredicate ：判断一个服务器的运行状况是否可用，去除不可用服务器的所有服务器

AvailabilityPredicate ：用于过滤连接数过多的服务器

在来看下chooseRoundRobinAfterFiltering方法，前面已经说过了它是过滤的方法，然后AvailabilityPredicate 里面并没有这方法，他直接继承了他的父类

com.netflix.loadbalancer.AbstractServerPredicate#chooseRoundRobinAfterFiltering(java.util.List<com.netflix.loadbalancer.Server>)

public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers) {

//过滤服务器列表

List<Server> eligible = getEligibleServers(servers);

if (eligible.size() == 0) {

return Optional.absent();

}

//（i+1）%n 轮询选择一个服务实例

return Optional.of(eligible.get(incrementAndGetModulo(eligible.size())));

}

com.netflix.loadbalancer.AbstractServerPredicate#getEligibleServers(java.util.List<com.netflix.loadbalancer.Server>, java.lang.Object)

由于前面loadBalancerKey直接传入的null, 方法getEligibleServers会使用serverOnlyPredicate来依次过滤

public List<Server> getEligibleServers(List<Server> servers, Object loadBalancerKey) {

if (loadBalancerKey == null) {

return ImmutableList.copyOf(Iterables.filter(servers, this.getServerOnlyPredicate()));

} else {

List<Server> results = Lists.newArrayList();

for (Server server: servers) {

if (this.apply(new PredicateKey(loadBalancerKey, server))) {

results.add(server);

}

}

return results;

}

}

serverOnlyPredicate 则会调用apply方法，并将Server 对象分装PredicateKey当作参数传入

private final Predicate<Server> serverOnlyPredicate = new Predicate<Server>() {

@Override

public boolean apply(@Nullable Server input) {

return AbstractServerPredicate.this.apply(new PredicateKey(input));

}

};

AbstractServerPredicate并没有实现apply方法，具体的实现又回到了子类CompositePredicate的apply方法

会依次调用ZoneAvoidancePredicate与AvailabilityPredicate的apply方法

public class ZoneSnapshot {

//实例数

final int instanceCount;

//平均负载

final double loadPerServer;

//断路器端口数量

final int circuitTrippedCount;

//活动请求数量

final int activeRequestsCount;

}

com.netflix.loadbalancer.ZoneAvoidancePredicate#apply

@Override

public boolean apply(@Nullable PredicateKey input) {

if (!ENABLED.get()) {

return true;

}

String serverZone = input.getServer().getZone();

if (serverZone == null) {

//如果服务器没有zone的相关信息，直接返回

return true;

}

LoadBalancerStats lbStats = getLBStats();

//LoadBalancerStats 存储每个服务器节点的执行特征和运行记录，这些信息可供动态负责均衡使用

if (lbStats == null) {

//如果没有服务器的记录，直接返回

return true;

}

if (lbStats.getAvailableZones().size() <= 1) {

//如果根本就一个服务器，直接返回

return true;

}

//PredicateKey 封装了Server的信息，判断下服务器区的记录是否用当前区的信息

Map<String, ZoneSnapshot> zoneSnapshot = ZoneAvoidanceRule.createSnapshot(lbStats);

//如果没有直接返回

if (!zoneSnapshot.keySet().contains(serverZone)) {

// The server zone is unknown to the load balancer, do not filter it out

return true;

}

logger.debug("Zone snapshots: {}", zoneSnapshot);

//获取可用的服务器列表

Set<String> availableZones = ZoneAvoidanceRule.getAvailableZones(zoneSnapshot, triggeringLoad.get(), triggeringBlackoutPercentage.get());

logger.debug("Available zones: {}", availableZones);

//判断当前服务器是否在可用的服务器列表中

if (availableZones != null) {

return availableZones.contains(input.getServer().getZone());

} else {

return false;

}

}

com.netflix.loadbalancer.ZoneAvoidanceRule#getAvailableZones(java.util.Map<java.lang.String,com.netflix.loadbalancer.ZoneSnapshot>, double, double)

最后落到这方法上

这方法是用来筛选服务区列表的

public static Set<String> getAvailableZones(

Map<String, ZoneSnapshot> snapshot, double triggeringLoad,

double triggeringBlackoutPercentage) {

if (snapshot.isEmpty()) {

return null;

}

Set<String> availableZones = new HashSet<String>(snapshot.keySet());

if (availableZones.size() == 1) {

return availableZones;

}

Set<String> worstZones = new HashSet<String>();

double maxLoadPerServer = 0;

boolean limitedZoneAvailability = false;

//遍历所有的服务区

for (Map.Entry<String, ZoneSnapshot> zoneEntry : snapshot.entrySet()) {

String zone = zoneEntry.getKey();

ZoneSnapshot zoneSnapshot = zoneEntry.getValue();

//获取服务器中的服务实例数

int instanceCount = zoneSnapshot.getInstanceCount();

if (instanceCount == 0) {

//如果服务器中没有服务实例，那么移除该服务区

availableZones.remove(zone);

limitedZoneAvailability = true;

} else {

double loadPerServer = zoneSnapshot.getLoadPerServer();

//服务区的实例平均负载小于0，或者实例故障率（断路器端口次数/实例数）大于等于阈值（默认0.99999），则去掉该服务区

if (((double) zoneSnapshot.getCircuitTrippedCount())

/ instanceCount >= triggeringBlackoutPercentage

|| loadPerServer < 0) {

availableZones.remove(zone);

limitedZoneAvailability = true;

} else {

//如果该服务区的平均负载和最大负载的差小于一定的两，则将该服务器加入到最坏服务区集合

if (Math.abs(loadPerServer - maxLoadPerServer) < 0.000001d) {

// they are the same considering double calculation

// round error

worstZones.add(zone);

} else if (loadPerServer > maxLoadPerServer) {

//否则，如果该zone的平均负载还大于最大负载

maxLoadPerServer = loadPerServer;

worstZones.clear();

worstZones.add(zone);

}

}

}

}

//如果最大的平均负载小于设定的阈值则直接返回

if (maxLoadPerServer < triggeringLoad && !limitedZoneAvailability) {

// zone override is not needed here

return availableZones;

}

//否则，从最好的服务器集合里面随机挑选一个

String zoneToAvoid = randomChooseZone(snapshot, worstZones);

if (zoneToAvoid != null) {

availableZones.remove(zoneToAvoid);

}

return availableZones;

}

CompositePredicate的apply方法

会依次调用ZoneAvoidancePredicate与AvailabilityPredicate的apply方法

那接下来就是AvailabilityPredicate的apply方法了

@Override

public boolean apply(@Nullable PredicateKey input) {

LoadBalancerStats stats = getLBStats();

if (stats == null) {

return true;

}

//获得关于该服务器的记录

return !shouldSkipServer(stats.getSingleServerStat(input.getServer()));

}

private boolean shouldSkipServer(ServerStats stats) {

//如果该服务器的断路器已经打开，或者他的连接数大于设定的阈值，那么就需要将服务器过滤掉

if ((CIRCUIT\_BREAKER\_FILTERING.get() && stats.isCircuitBreakerTripped())

|| stats.getActiveRequestsCount() >= activeConnectionsLimit.get()) {

return true;

}

return false;

}

# Feign接口服务

前面已经学习了Ribbon,从Eureka获取服务的实例在通过RestTemplate调用，并转换成需要的对象

List<Product> list = restTemplate.exchange(PRODUCT\_LIST\_URL,HttpMethod.GET,new HttpEntity<Object>(httpHeaders), List.class).getBody();

可以发现所有的数据调用和转换都是由用户直接来完成的，我们可能不想直接访问Rest接口，如果转换回来的直接是对象而不需要直接使用RestTemplate进行转换就好了，这个时候就需要使用Feign了

## Feign基本使用

复制【microcloud-consumer】 成一个新的模块【microcloud-consumer-feign】

【microcloud-consumer-feign】修改pom文件，增加对feign的支持

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-consumer-feign</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

</dependencies>

</project>

注意：这里又有版本问题，如果是Edgware或之前的版本，

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

feign的操作其实需要ribbon的支持

【microcloud-service】，新建立一个microcloud-service模块，这个模块专门定义客户端的调用接口

【microcloud-service】，修改pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-service</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-service】如果要通过Feign进行远程调用，依然需要安全服务提供方的认证问题，不过在feign里面已经集成了这块功能

package cn.enjoy.feign;

import feign.auth.BasicAuthRequestInterceptor;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FeignClientConfig {

@Bean

public BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {

return new BasicAuthRequestInterceptor("admin", "enjoy");

}

}

【microcloud-service】 新建一个IProductClientService接口

package cn.enjoy.service;

import cn.enjoy.feign.FeignClientConfig;

import cn.enjoy.vo.Product;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@FeignClient(name = "MICROCLOUD-PROVIDER-PRODUCT",configuration = FeignClientConfig.class)

public interface IProductClientService {

@RequestMapping("/prodcut/get/{id}")

public Product getProduct(@PathVariable("id")long id);

@RequestMapping("/prodcut/list")

public List<Product> listProduct() ;

@RequestMapping("/prodcut/add")

public boolean addPorduct(Product product) ;

}

【microcloud-consumer-feign】 修改pom文件，引入microcloud-service 包

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-service</artifactId>

</dependency>

【microcloud-consumer-feign】 由于microcloud-service里面已经做了安全验证，并且后面并不直接使用RestTemplate ,删除RestConfig.java类

//package cn.enjoy.config;

//

//

//import org.springframework.cloud.client.loadbalancer.LoadBalanced;

//import org.springframework.context.annotation.Bean;

//import org.springframework.context.annotation.Configuration;

//import org.springframework.http.HttpHeaders;

//import org.springframework.web.client.RestTemplate;

//

//import java.nio.charset.Charset;

//import java.util.Base64;

//

//@Configuration

//public class RestConfig {

//

// @Bean

// @LoadBalanced

// public RestTemplate restTemplate() {

// return new RestTemplate();

// }

//

// @Bean

// public HttpHeaders getHeaders() { // 要进行一个Http头信息配置

// HttpHeaders headers = new HttpHeaders(); // 定义一个HTTP的头信息

// String auth = "root:enjoy"; // 认证的原始信息

// byte[] encodedAuth = Base64.getEncoder()

// .encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

// String authHeader = "Basic " + new String(encodedAuth);

// headers.set("Authorization", authHeader);

// return headers;

// }

//

//

//}

【microcloud-consumer-feign】 修改ConsumerProductController，这个时候直接使用microcloud-service定义的服务就可以了

package cn.enjoy.controller;

import cn.enjoy.service.IProductClientService;

import cn.enjoy.vo.Product;

import org.springframework.cloud.client.ServiceInstance;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.util.List;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

@Resource

private IProductClientService iProductClientService;

@RequestMapping("/product/get")

public Object getProduct(long id) {

return iProductClientService.getProduct(id);

}

@RequestMapping("/product/list")

public Object listProduct() {

return iProductClientService.listProduct();

}

@RequestMapping("/product/add")

public Object addPorduct(Product product) {

return iProductClientService.addPorduct(product);

}

}

可见，这个时候ConsumerProductController的代码已经简洁了不少

【microcloud-consumer-feign】修改程序主类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients("cn.enjoy.service")

public class ConsumerFeignApp {

public static void main(String[] args) {

SpringApplication.run(ConsumerFeignApp.class,args);

}

}

启动测试:

<http://localhost/consumer/product/list>

可以发现Feign在调用接口的时候自带负载均衡，这也不奇怪，因为Fegin里面内置就使用的Ribbon

可以做个测试，看下是否真的如此

【microcloud-consumer-feign】修改程序主类

package cn.enjoy;

import cn.xiangxue.config.RibbonConfig;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.ribbon.RibbonClient;

import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients("cn.enjoy.service")

@RibbonClient(name ="MICROCLOUD-PROVIDER-PRODUCT" ,configuration = RibbonConfig.class)

public class ConsumerFeignApp {

public static void main(String[] args) {

SpringApplication.run(ConsumerFeignApp.class,args);

}

}

启动测试:

<http://localhost/consumer/product/list>

可以发现，现在的路由规则以及变成了随机访问

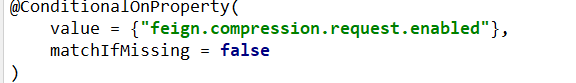
## 其他配置

### 数据压缩

前面我们已经知道Feign之中最核心的作用就是将Rest服务的信息转化为接口，这其中还有其他的一些地方应该要考虑，比如：数据的压缩

Rest协议更多的传输的是文本，JSON或者XML，如果用户发送的请求很大，这个时候有必要对数据进行压缩处理，好在feign本身就提供了压缩的支持

FeignContentGzipEncodingAutoConfiguration 可以先看下这个类



虽然Feign支持压缩，但默认是不开启的

再看下FeignClientEncodingProperties，可以根据这里面的属性进行相关压缩的配置

【microcloud-consumer-feign】 修改application.yml配置文件

feign:

compression:

request:

enabled: true

mime-types: # 可以被压缩的类型

- text/xml

- application/xml

- application/json

min-request-size: 2048 # 超过2048的字节进行压缩

### 日志配置

[在构建@FeignClient注解修饰的服务客户端时，会为一个客户端都创建一个feign.Logger实例，可以利用日志来分析Feign的请求细节，不过默认](mailto:在构建@FeignClient注解修饰的服务客户端时，会为一个客户端都创建一个feign.Logger实例，可以利用日志来分析Feign的请求细节，不过默认) Feign的日志是不开启的。

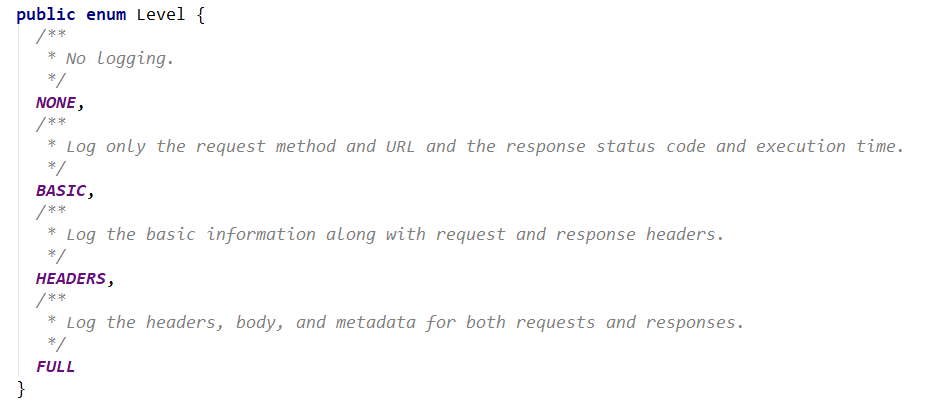
【microcloud-consumer-feign】 修改 application.yml配置文件，增加日志信息

logging:

level:

cn.enjoy.service: DEBUG

只添加上面配置还无法实现对DEBUG日志的输出，以因为Feign客户端默认的logger.level对象定义为none级别，所以不会记录feign调用过程中的信息



【microcloud-service】修改FeignClientConfig，开启日志输出

package cn.enjoy.feign;

import feign.Logger;

import feign.auth.BasicAuthRequestInterceptor;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FeignClientConfig {

@Bean

public Logger.Level getFeignLoggerLevel() {

return feign.Logger.Level.FULL ;

}

@Bean

public BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {

return new BasicAuthRequestInterceptor("admin", "enjoy");

}

}

访问：localhost/consumer/product/list

2019-02-21 10:34:54.015 INFO 424 --- [p-nio-80-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'

2019-02-21 10:34:54.015 INFO 424 --- [p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'

2019-02-21 10:34:54.021 INFO 424 --- [p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 6 ms

2019-02-21 10:34:54.050 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] ---> GET http://MICROCLOUD-PROVIDER-PRODUCT/prodcut/list HTTP/1.1

2019-02-21 10:34:54.050 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] Authorization: Basic YWRtaW46ZW5qb3k=

2019-02-21 10:34:54.050 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] ---> END HTTP (0-byte body)

2019-02-21 10:34:54.493 INFO 424 --- [p-nio-80-exec-1] c.n.u.concurrent.ShutdownEnabledTimer : Shutdown hook installed for: NFLoadBalancer-PingTimer-MICROCLOUD-PROVIDER-PRODUCT

2019-02-21 10:34:54.518 INFO 424 --- [p-nio-80-exec-1] c.netflix.loadbalancer.BaseLoadBalancer : Client: MICROCLOUD-PROVIDER-PRODUCT instantiated a LoadBalancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=MICROCLOUD-PROVIDER-PRODUCT,current list of Servers=[],Load balancer stats=Zone stats: {},Server stats: []}ServerList:null

2019-02-21 10:34:54.524 INFO 424 --- [p-nio-80-exec-1] c.n.l.DynamicServerListLoadBalancer : Using serverListUpdater PollingServerListUpdater

2019-02-21 10:34:54.590 INFO 424 --- [p-nio-80-exec-1] c.n.l.DynamicServerListLoadBalancer : DynamicServerListLoadBalancer for client MICROCLOUD-PROVIDER-PRODUCT initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=MICROCLOUD-PROVIDER-PRODUCT,current list of Servers=[192.168.30.3:8081, 192.168.30.3:8082, 192.168.30.3:8080],Load balancer stats=Zone stats: {defaultzone=[Zone:defaultzone; Instance count:3; Active connections count: 0; Circuit breaker tripped count: 0; Active connections per server: 0.0;]

},Server stats: [[Server:192.168.30.3:8080; Zone:defaultZone; Total Requests:0; Successive connection failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00 CST 1970; First connection made: Thu Jan 01 08:00:00 CST 1970; Active Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]

, [Server:192.168.30.3:8081; Zone:defaultZone; Total Requests:0; Successive connection failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00 CST 1970; First connection made: Thu Jan 01 08:00:00 CST 1970; Active Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]

, [Server:192.168.30.3:8082; Zone:defaultZone; Total Requests:0; Successive connection failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00 CST 1970; First connection made: Thu Jan 01 08:00:00 CST 1970; Active Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]

]}ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList@53c53064

2019-02-21 10:34:55.568 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] <--- HTTP/1.1 200 (1517ms)

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] cache-control: no-cache, no-store, max-age=0, must-revalidate

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] content-type: application/json;charset=UTF-8

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] date: Thu, 21 Feb 2019 02:34:55 GMT

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] expires: 0

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] pragma: no-cache

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] transfer-encoding: chunked

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] x-content-type-options: nosniff

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] x-frame-options: DENY

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] x-xss-protection: 1; mode=block

2019-02-21 10:34:55.569 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct]

2019-02-21 10:34:55.571 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] [{"productId":1,"productName":"java编程","productDesc":"springcloud"},{"productId":2,"productName":"Springboot","productDesc":"springcloud"},{"productId":3,"productName":"西游记","productDesc":"springcloud"},{"productId":4,"productName":"水浒传","productDesc":"springcloud"},{"productId":5,"productName":"西厢记","productDesc":"springcloud"}]

2019-02-21 10:34:55.571 DEBUG 424 --- [p-nio-80-exec-1] cn.enjoy.service.IProductClientService : [IProductClientService#listProduct] <--- END HTTP (353-byte body)

* 当使用 Feign 要通过接口的方法访问 Rest 服务的时候会根据设置的服务类型发出请求，这个请求是发送给 Eureka
* 随后由于配置了授权处理，所以继续发送授权信息（“Authorization”）

其实在外面使用RestTemplate的时候也是这么做的，可以对应日志的加密内容和直接访问其实是一样的。

* 在进行服务调用的时候 Feign 融合了 Ribbon 技术，所以也支持有负载均衡的处理

Feign = RestTempate + HttpHeader + Ribbon + Eureka 综合体，使用feign大大增加了代码的灵活程度

### 原理解析

在@EnableFeignClients标签中，import了一个FeignClientsRegistrar类

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@Documented

@Import(FeignClientsRegistrar.class)

public @interface EnableFeignClients {

String[] value() default {};

String[] basePackages() default {};

Class<?>[] basePackageClasses() default {};

Class<?>[] defaultConfiguration() default {};

Class<?>[] clients() default {};

}

org.springframework.cloud.openfeign.FeignClientsRegistrar#registerBeanDefinitions 方法不多说，它会再AbstractApplicationContext#refresh()，这个不熟悉的同学好好去看下spring的源代码

@Override

public void registerBeanDefinitions(AnnotationMetadata metadata,

BeanDefinitionRegistry registry) {

//从EnableFeignClients的属性值来构建Feign的Configuration

registerDefaultConfiguration(metadata, registry);

//扫描package,注册被@FeignClient修饰的接口类Bean的信息

registerFeignClients(metadata, registry);

}

#### 准备工作

方法registerDefaultConfiguration

org.springframework.cloud.openfeign.FeignClientsRegistrar#registerDefaultConfiguration

private void registerDefaultConfiguration(AnnotationMetadata metadata,

BeanDefinitionRegistry registry) {

//获取到metadata中关于EnableFeignClients的属性值键值对。

Map<String, Object> defaultAttrs = metadata

.getAnnotationAttributes(EnableFeignClients.class.getName(), true);

//如果配置了defaultConfiguration 进行配置,如果没有使用默认的configuration

if (defaultAttrs != null && defaultAttrs.containsKey("defaultConfiguration")) {

String name;

if (metadata.hasEnclosingClass()) {

name = "default." + metadata.getEnclosingClassName();

}

else {

name = "default." + metadata.getClassName();

}

//进行注册

registerClientConfiguration(registry, name,

defaultAttrs.get("defaultConfiguration"));

}

}

org.springframework.cloud.openfeign.FeignClientsRegistrar#registerClientConfiguration

接下来进行注册

private void registerClientConfiguration(BeanDefinitionRegistry registry, Object name,

Object configuration) {

//使用BeanDefinitionBuilder 来生成BeanDefinition,并把它进行注册

BeanDefinitionBuilder builder = BeanDefinitionBuilder

.genericBeanDefinition(FeignClientSpecification.class);

builder.addConstructorArgValue(name);

builder.addConstructorArgValue(configuration);

registry.registerBeanDefinition(

name + "." + FeignClientSpecification.class.getSimpleName(),

builder.getBeanDefinition());

}

BeanDefinitionRegistry 是pring框架用于动态注册BeanDefinition信息的接口，调用registerBeanDefinition方法可以将BeanDefinition注册到Spring容器中，其中name属性就是注册的BeanDefinition的名称

在这里它注册了一个FeignClientSpecification的对象

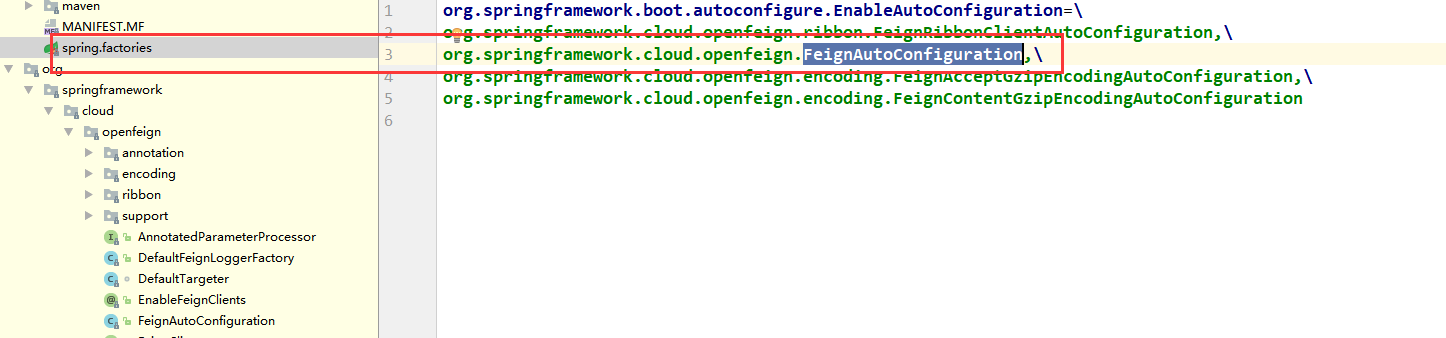
FeignClientSpecification实现了NamedContextFactory.Specification接口，它是Feign实例化的重要一环，在上面的方法中，它持有自定义配置的组件实例，Springcloud使用NamedContextFactory创建一些列的运行上下文（ApplicationContext）来让对应的Specification在这些上下文中创建实例对象

NamedContextFactory有3个功能

1. 创建AnnotationConfigApplicationContext上下文
2. 在上下文中创建并获取bean实例
3. 当上下文销毁时清除其中的feign实例

NamedContextFactory有个非常重要的子类FeignContext，用于存储各种OpenFeign的组件实例

FeignContext是哪里构建的呢？



org.springframework.cloud.openfeign.FeignAutoConfiguration#feignContext

@Autowired(required = false)

private List<FeignClientSpecification> configurations = new ArrayList<>();

@Bean

public FeignContext feignContext() {

FeignContext context = new FeignContext();

context.setConfigurations(this.configurations);

return context;

}

public FeignContext() {

//将默认的FeignClientsConfiguration作为参数传递给构造函数

super(FeignClientsConfiguration.class, "feign", "feign.client.name");

}

FeignContext 创建的时候会将之前FeignClientSpecification通过setConfigurations设置给context上下文

org.springframework.cloud.context.named.NamedContextFactory#createContext

FeignContext 的父类NamedContextFactory的createContext方法会将创建AnnotationConfigApplicationContext实例，这实例将作为当前上下文的子上下文，用于关联feign组件的不同实例

protected AnnotationConfigApplicationContext createContext(String name) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

//获取name所对应的configuration,如果有就注册到子context中

if (this.configurations.containsKey(name)) {

for (Class<?> configuration : this.configurations.get(name)

.getConfiguration()) {

context.register(configuration);

}

}

//注册default的Configuration,也就是 FeignClientsRegistrar类中registerDefaultConfiguration方法中注册的Configuration

for (Map.Entry<String, C> entry : this.configurations.entrySet()) {

if (entry.getKey().startsWith("default.")) {

for (Class<?> configuration : entry.getValue().getConfiguration()) {

context.register(configuration);

}

}

}

//注册PropertyPlaceholderAutoConfiguration

context.register(PropertyPlaceholderAutoConfiguration.class,

this.defaultConfigType);

//设置Environment的propertySources属性源

context.getEnvironment().getPropertySources().addFirst(new MapPropertySource(

this.propertySourceName,

Collections.<String, Object> singletonMap(this.propertyName, name)));

if (this.parent != null) {

// Uses Environment from parent as well as beans

context.setParent(this.parent);

}

context.setDisplayName(generateDisplayName(name));

context.refresh();

return context;

}

由于NamedContextFactory实现了 DisposableBean，所以当实例消亡的时候会调用

@Override

public void destroy() {

Collection<AnnotationConfigApplicationContext> values = this.contexts.values();

for (AnnotationConfigApplicationContext context : values) {

context.close();

}

this.contexts.clear();

}

总结一下：

NamedContextFactory会创建出AnnotationConfigApplicationContext实例,并以name作为唯一标识,然后每个AnnotationConfigApplicationContext实例都会注册部分配置类,从面可以给出一系列的基于配置类生成的组件实例,这样就可以基于name来管理一系列的组件实例,为不同的 FeignClient准备不同配置组件实例

方法registerFeignClients

FeignClientsRegistrar做的第二件事就是扫描指定包下的类文件，注册@FeignClient修饰的接口类信息

public void registerFeignClients(AnnotationMetadata metadata,

BeanDefinitionRegistry registry) {

//生成自定义的ClassPathScanningCandidateComponentProvider

ClassPathScanningCandidateComponentProvider scanner = getScanner();

scanner.setResourceLoader(this.resourceLoader);

Set<String> basePackages;

//获取EnableFeignClients所有属性的键值对

Map<String, Object> attrs = metadata

.getAnnotationAttributes(EnableFeignClients.class.getName());

//依照AnnotationTypeFilter 来进行过滤，只会扫描出EnableFeignClients修饰的类

AnnotationTypeFilter annotationTypeFilter = new AnnotationTypeFilter(

FeignClient.class);

final Class<?>[] clients = attrs == null ? null

: (Class<?>[]) attrs.get("clients");

//如果没有配置clients属性，那么就要扫描basePackages

if (clients == null || clients.length == 0) {

scanner.addIncludeFilter(annotationTypeFilter);

basePackages = getBasePackages(metadata);

}

else {

final Set<String> clientClasses = new HashSet<>();

basePackages = new HashSet<>();

//遍历上诉过程中获取basePackages 的列表

for (String basePackage : basePackages) {

//获取basePackage中所有的BeanDefinition

Set<BeanDefinition> candidateComponents = scanner

.findCandidateComponents(basePackage);

for (BeanDefinition candidateComponent : candidateComponents) {

if (candidateComponent instanceof AnnotatedBeanDefinition) {

// verify annotated class is an interface

AnnotatedBeanDefinition beanDefinition = (AnnotatedBeanDefinition) candidateComponent;

AnnotationMetadata annotationMetadata = beanDefinition.getMetadata();

Assert.isTrue(annotationMetadata.isInterface(),

"@FeignClient can only be specified on an interface");

//从这些BeanDefinition中获取FeignClient的属性值

Map<String, Object> attributes = annotationMetadata

.getAnnotationAttributes(

FeignClient.class.getCanonicalName());

String name = getClientName(attributes);

//对单独的某个FeignClient的configuration进行配置

registerClientConfiguration(registry, name,

attributes.get("configuration"));

//注册

registerFeignClient(registry, annotationMetadata, attributes);

}

}

}

}

private void registerFeignClient(BeanDefinitionRegistry registry,

AnnotationMetadata annotationMetadata, Map<String, Object> attributes) {

String className = annotationMetadata.getClassName();

BeanDefinitionBuilder definition = BeanDefinitionBuilder

.genericBeanDefinition(FeignClientFactoryBean.class);

validate(attributes);

definition.addPropertyValue("url", getUrl(attributes));

definition.addPropertyValue("path", getPath(attributes));

String name = getName(attributes);

definition.addPropertyValue("name", name);

definition.addPropertyValue("type", className);

definition.addPropertyValue("decode404", attributes.get("decode404"));

definition.addPropertyValue("fallback", attributes.get("fallback"));

definition.addPropertyValue("fallbackFactory", attributes.get("fallbackFactory"));

definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE\_BY\_TYPE);

String alias = name + "FeignClient";

AbstractBeanDefinition beanDefinition = definition.getBeanDefinition();

boolean primary = (Boolean)attributes.get("primary"); // has a default, won't be null

beanDefinition.setPrimary(primary);

String qualifier = getQualifier(attributes);

if (StringUtils.hasText(qualifier)) {

alias = qualifier;

}

BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDefinition, className,

new String[] { alias });

BeanDefinitionReaderUtils.registerBeanDefinition(holder, registry);

}

总结FeignClientsRegistrar方法会依据@EnableFeignClients的属性获取要扫描的包路径信息，然后获取这些包下被@FeignClient注解修饰的接口的BeanDefinition

FeignClientFactoryBean 是工厂类，Spring容器通过调用它的getObject方法来获取对应的bean实例，被@FeignClient修饰的接口都是通过FeignClientFactoryBean 的getObject方法来进行实例化的。

org.springframework.cloud.openfeign.FeignClientFactoryBean#getObject

@Override

public Object getObject() throws Exception {

FeignContext context = applicationContext.getBean(FeignContext.class);

Feign.Builder builder = feign(context);

if (StringUtils.hasText(this.url) && !this.url.startsWith("http")) {

this.url = "http://" + this.url;

}

String url = this.url + cleanPath();

//调用FeignContext的getInstance方法获取Client对象

Client client = getOptional(context, Client.class);

if (client != null) {

if (client instanceof LoadBalancerFeignClient) {

client = ((LoadBalancerFeignClient)client).getDelegate();

}

builder.client(client);

}

Targeter targeter = get(context, Targeter.class);

return targeter.target(this, builder, context, new HardCodedTarget<>(

this.type, this.name, url));

}

public <T> T getInstance(String name, Class<T> type) {

AnnotationConfigApplicationContext context = getContext(name);

if (BeanFactoryUtils.beanNamesForTypeIncludingAncestors(context,

type).length > 0) {

//从对应的context获取bean实例

return context.getBean(type);

}

return null;

}

Targeter 是一个接口，它的target方法会生成对应实例对象，它有两个实现类，分表为DefaultTargeter和HystrixTargeter ,fegign使用HystrixTargeter 这一层抽象来封装关于Hystrix的实现，DefaultTargeter的实现如下所示，只是调用了Feign.Builder的target方法

class DefaultTargeter implements Targeter {

@Override

public <T> T target(FeignClientFactoryBean factory, Feign.Builder feign, FeignContext context,

Target.HardCodedTarget<T> target) {

return feign.target(target);

}

}

Feign.Builder负责生成被@FeignClient修饰的FeignClient接口类实例，它通过JAVA反射机制构建

public <T> T target(Target<T> target) {

return build().newInstance(target);

}

public Feign build() {

SynchronousMethodHandler.Factory synchronousMethodHandlerFactory =

new SynchronousMethodHandler.Factory(client, retryer, requestInterceptors, logger,

logLevel, decode404);

ParseHandlersByName handlersByName =

new ParseHandlersByName(contract, options, encoder, decoder,

errorDecoder, synchronousMethodHandlerFactory);

return new ReflectiveFeign(handlersByName, invocationHandlerFactory);

}

}

feign.ReflectiveFeign#newInstance

这方法主要做两件事、

1. 扫描FeignClient接口类的所有函数，生成对应的Handler
2. 使用Proxy生成FeignClient的实例对象

public <T> T newInstance(Target<T> target) {

//ParseHandlersByName.appy方法填充信息

Map<String, MethodHandler> nameToHandler = targetToHandlersByName.apply(target);

Map<Method, MethodHandler> methodToHandler = new LinkedHashMap<Method, MethodHandler>();

List<DefaultMethodHandler> defaultMethodHandlers = new LinkedList<DefaultMethodHandler>();

for (Method method : target.type().getMethods()) {

if (method.getDeclaringClass() == Object.class) {

continue;

} else if(Util.isDefault(method)) {

//为每一个默认方法生成一个DefaultMethodHandler

DefaultMethodHandler handler = new DefaultMethodHandler(method);

defaultMethodHandlers.add(handler);

methodToHandler.put(method, handler);

} else {

methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type(), method)));

}

}

//生成java InvocationHandler 是个动态代理的类

InvocationHandler handler = factory.create(target, methodToHandler);

T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(), new Class<?>[]{target.type()}, handler);

for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {

defaultMethodHandler.bindTo(proxy);

}

return proxy;

}

扫描FeignClient接口类的所有函数，生成对应的Handler

在扫描FeignClient接口类所有函数生成对应的Handle过程中，feign会生成调用该函数时发送网络请求的模板，也就是RequestTemplate实例,RequestTemplate中包含了发送网络请求的URL和产生填充信息，@RequestMapping,@RequestVariable等注解信息也会包含到RequestTemplate中，这一具体过程就是ParseHandlersByName#apply来实现的。

feign.ReflectiveFeign.ParseHandlersByName#apply

public Map<String, MethodHandler> apply(Target key) {

//获取type中所有的方法信息，会根据注解生成每个方法的RequestTemplate

List<MethodMetadata> metadata = contract.parseAndValidatateMetadata(key.type());

Map<String, MethodHandler> result = new LinkedHashMap<String, MethodHandler>();

for (MethodMetadata md : metadata) {

BuildTemplateByResolvingArgs buildTemplate;

if (!md.formParams().isEmpty() && md.template().bodyTemplate() == null) {

buildTemplate = new BuildFormEncodedTemplateFromArgs(md, encoder);

} else if (md.bodyIndex() != null) {

buildTemplate = new BuildEncodedTemplateFromArgs(md, encoder);

} else {

buildTemplate = new BuildTemplateByResolvingArgs(md);

}

result.put(md.configKey(),

factory.create(key, md, buildTemplate, options, decoder, errorDecoder));

}

return result;

}

}

Contract的默认实现是org.springframework.cloud.openfeign.support.SpringMvcContract

其基类为Contract.BaseContract

org.springframework.cloud.openfeign.support.SpringMvcContract#parseAndValidateMetadata

方法会解析与HTTP请求相关的所有函数的基本信息和注解信息

@Override

public MethodMetadata parseAndValidateMetadata(Class<?> targetType, Method method) {

this.processedMethods.put(Feign.configKey(targetType, method), method);

//调用父类BaseContract的函数

MethodMetadata md = super.parseAndValidateMetadata(targetType, method);

RequestMapping classAnnotation = findMergedAnnotation(targetType,

RequestMapping.class);

//处理RequestMapping注解

if (classAnnotation != null) {

// produces - use from class annotation only if method has not specified this

if (!md.template().headers().containsKey(ACCEPT)) {

parseProduces(md, method, classAnnotation);

}

// consumes -- use from class annotation only if method has not specified this

if (!md.template().headers().containsKey(CONTENT\_TYPE)) {

parseConsumes(md, method, classAnnotation);

}

// headers -- class annotation is inherited to methods, always write these if

// present

parseHeaders(md, method, classAnnotation);

}

return md;

}

feign.Contract.BaseContract#parseAndValidateMetadata 父类的parseAndValidateMetadata 方法会依次解析接口类中的注解，方法中的注解，和各种参数注解，并将这些注解包含的信息封装到MethodMetadata 对象中，然后返回。

protected MethodMetadata parseAndValidateMetadata(Class<?> targetType, Method method) {

MethodMetadata data = new MethodMetadata();

//函数的返回值

data.returnType(Types.resolve(targetType, targetType, method.getGenericReturnType()));

data.configKey(Feign.configKey(targetType, method));

//获取并处理修饰class的注解信息

if(targetType.getInterfaces().length == 1) {

processAnnotationOnClass(data, targetType.getInterfaces()[0]);

}

//调用子类processAnnotationOnClass的实现

processAnnotationOnClass(data, targetType);

//处理修饰method的注解信息

for (Annotation methodAnnotation : method.getAnnotations()) {

processAnnotationOnMethod(data, methodAnnotation, method);

}

checkState(data.template().method() != null,

"Method %s not annotated with HTTP method type (ex. GET, POST)",

method.getName());

//方法参数的类型

Class<?>[] parameterTypes = method.getParameterTypes();

Type[] genericParameterTypes = method.getGenericParameterTypes();

//方法参数的注解

Annotation[][] parameterAnnotations = method.getParameterAnnotations();

int count = parameterAnnotations.length;

//依次处理各个函数参数的注解

for (int i = 0; i < count; i++) {

boolean isHttpAnnotation = false;

if (parameterAnnotations[i] != null) {

isHttpAnnotation = processAnnotationsOnParameter(data, parameterAnnotations[i], i);

}

if (parameterTypes[i] == URI.class) {

data.urlIndex(i);

} else if (!isHttpAnnotation) {

checkState(data.formParams().isEmpty(),

"Body parameters cannot be used with form parameters.");

checkState(data.bodyIndex() == null, "Method has too many Body parameters: %s", method);

data.bodyIndex(i);

data.bodyType(Types.resolve(targetType, targetType, genericParameterTypes[i]));

}

}

if (data.headerMapIndex() != null) {

checkMapString("HeaderMap", parameterTypes[data.headerMapIndex()], genericParameterTypes[data.headerMapIndex()]);

}

if (data.queryMapIndex() != null) {

checkMapString("QueryMap", parameterTypes[data.queryMapIndex()], genericParameterTypes[data.queryMapIndex()]);

}

return data;

}

使用Proxy生成FeignClient的实例对象

feign.ReflectiveFeign#newInstance

InvocationHandler handler = factory.create(target, methodToHandler);

中另外一件事情是

OpenFeign使用Proxy.newProxyInstance方法来创建FeignClient接口类的实例，然后讲InvocationHandle绑定到接口实例上，用于处理接口类函数调用

feign.InvocationHandlerFactory.Default

static final class Default implements InvocationHandlerFactory {

@Override

public InvocationHandler create(Target target, Map<Method, MethodHandler> dispatch) {

return new ReflectiveFeign.FeignInvocationHandler(target, dispatch);

}

}

Default 实现了InvocationHandlerFactory 接口，create方法其实返回ReflectiveFeign.FeignInvocationHandler的实例

ReflectiveFeign.FeignInvocationHandler.invoke方法

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if ("equals".equals(method.getName())) {

try {

Object

otherHandler =

args.length > 0 && args[0] != null ? Proxy.getInvocationHandler(args[0]) : null;

return equals(otherHandler);

} catch (IllegalArgumentException e) {

return false;

}

} else if ("hashCode".equals(method.getName())) {

return hashCode();

} else if ("toString".equals(method.getName())) {

return toString();

}

return dispatch.get(method).invoke(args);

}

Invoke方法会根据函数名称来调用不同的MethodHandle实例invoke方法

网络请求

接下来就直接使用FeignClient接口类的实例，调用它的函数来发送网络请求，

发送网络请求可以分为3个阶段

1. 是将函数实际参数添加到RequestTemplate中
2. 调用Target生成具体的Request对象
3. 调用Client来发送网络请求，将Response转化为对象返回

feign.SynchronousMethodHandler#invoke

@Override

public Object invoke(Object[] argv) throws Throwable {

//根据函数参数创建RequestTemplate

RequestTemplate template = buildTemplateFromArgs.create(argv);

Retryer retryer = this.retryer.clone();

while (true) {

try {

return executeAndDecode(template);

} catch (RetryableException e) {

retryer.continueOrPropagate(e);

if (logLevel != Logger.Level.NONE) {

logger.logRetry(metadata.configKey(), logLevel);

}

continue;

}

}

}

@Override

public RequestTemplate create(Object[] argv) {

RequestTemplate mutable = new RequestTemplate(metadata.template());

//设置url

if (metadata.urlIndex() != null) {

int urlIndex = metadata.urlIndex();

checkArgument(argv[urlIndex] != null, "URI parameter %s was null", urlIndex);

mutable.insert(0, String.valueOf(argv[urlIndex]));

}

Map<String, Object> varBuilder = new LinkedHashMap<String, Object>();

//遍历MethodMetadata中关于参数的所以以及对应名称的配置信息

for (Entry<Integer, Collection<String>> entry : metadata.indexToName().entrySet()) {

int i = entry.getKey();

//entry.getKey就是参数的索引

Object value = argv[entry.getKey()];

if (value != null) { // Null values are skipped.

//indexToExpander保存着将各种类型参数的值转化为String类型的Expander转换器

if (indexToExpander.containsKey(i)) {

//将Value值为String

value = expandElements(indexToExpander.get(i), value);

}

for (String name : entry.getValue()) {

varBuilder.put(name, value);

}

}

}

RequestTemplate template = resolve(argv, mutable, varBuilder);

//设置queryMap参数

if (metadata.queryMapIndex() != null) {

// add query map parameters after initial resolve so that they take

// precedence over any predefined values

template = addQueryMapQueryParameters((Map<String, Object>) argv[metadata.queryMapIndex()], template);

}

//设置headersMap参数

if (metadata.headerMapIndex() != null) {

template = addHeaderMapHeaders((Map<String, Object>) argv[metadata.headerMapIndex()], template);

}

return template;

}

feign.RequestTemplate#resolve(java.util.Map<java.lang.String,?>, java.util.Map<java.lang.String,java.lang.Boolean>)

RequestTemplate resolve(Map<String, ?> unencoded, Map<String, Boolean> alreadyEncoded) {

//替换query数值，将{queryVariable} 替换成实际的值

replaceQueryValues(unencoded, alreadyEncoded);

Map<String, String> encoded = new LinkedHashMap<String, String>();

for (Entry<String, ?> entry : unencoded.entrySet()) {

//把所有参数进行编码

final String key = entry.getKey();

final Object objectValue = entry.getValue();

String encodedValue = encodeValueIfNotEncoded(key, objectValue, alreadyEncoded);

encoded.put(key, encodedValue);

}

//编码URL

String resolvedUrl = expand(url.toString(), encoded).replace("+", "%20");

if (decodeSlash) {

resolvedUrl = resolvedUrl.replace("%2F", "/");

}

url = new StringBuilder(resolvedUrl);

//把头部信息进行传串行化

Map<String, Collection<String>> resolvedHeaders = new LinkedHashMap<String, Collection<String>>();

for (String field : headers.keySet()) {

Collection<String> resolvedValues = new ArrayList<String>();

for (String value : valuesOrEmpty(headers, field)) {

String resolved = expand(value, unencoded);

resolvedValues.add(resolved);

}

resolvedHeaders.put(field, resolvedValues);

}

headers.clear();

headers.putAll(resolvedHeaders);

//处理body信息

if (bodyTemplate != null) {

body(urlDecode(expand(bodyTemplate, encoded)));

}

return this;

}

feign.SynchronousMethodHandler#invoke

executeAndDecode()方法会根据RequestTemplate生成Request对象，然后交给Client实例发送网络请求，最后返回对应的函数返回类型的实例。

这里的client为

org.springframework.cloud.openfeign.ribbon.LoadBalancerFeignClient

Object executeAndDecode(RequestTemplate template) throws Throwable {

//根据RequestTemplate生成Request

Request request = targetRequest(template);

if (logLevel != Logger.Level.NONE) {

logger.logRequest(metadata.configKey(), logLevel, request);

}

Response response;

long start = System.nanoTime();

try {

//client发送网络请求

response = client.execute(request, options);

// ensure the request is set. TODO: remove in Feign 10

response.toBuilder().request(request).build();

} catch (IOException e) {

if (logLevel != Logger.Level.NONE) {

logger.logIOException(metadata.configKey(), logLevel, e, elapsedTime(start));

}

throw errorExecuting(request, e);

}

long elapsedTime = TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - start);

boolean shouldClose = true;

try {

if (logLevel != Logger.Level.NONE) {

response =

logger.logAndRebufferResponse(metadata.configKey(), logLevel, response, elapsedTime);

// ensure the request is set. TODO: remove in Feign 10

response.toBuilder().request(request).build();

}

//如果是response返回类型就可以直接返回

if (Response.class == metadata.returnType()) {

if (response.body() == null) {

return response;

}

//设置b**ody**

if (response.body().length() == null ||

response.body().length() > MAX\_RESPONSE\_BUFFER\_SIZE) {

shouldClose = false;

return response;

}

// Ensure the response body is disconnected

byte[] bodyData = Util.toByteArray(response.body().asInputStream());

return response.toBuilder().body(bodyData).build();

}

if (response.status() >= 200 && response.status() < 300) {

if (void.class == metadata.returnType()) {

return null;

} else {

return decode(response);

}

} else if (decode404 && response.status() == 404 && void.class != metadata.returnType()) {

return decode(response);

} else {

throw errorDecoder.decode(metadata.configKey(), response);

}

} catch (IOException e) {

if (logLevel != Logger.Level.NONE) {

logger.logIOException(metadata.configKey(), logLevel, e, elapsedTime);

}

throw errorReading(request, response, e);

} finally {

if (shouldClose) {

ensureClosed(response.body());

}

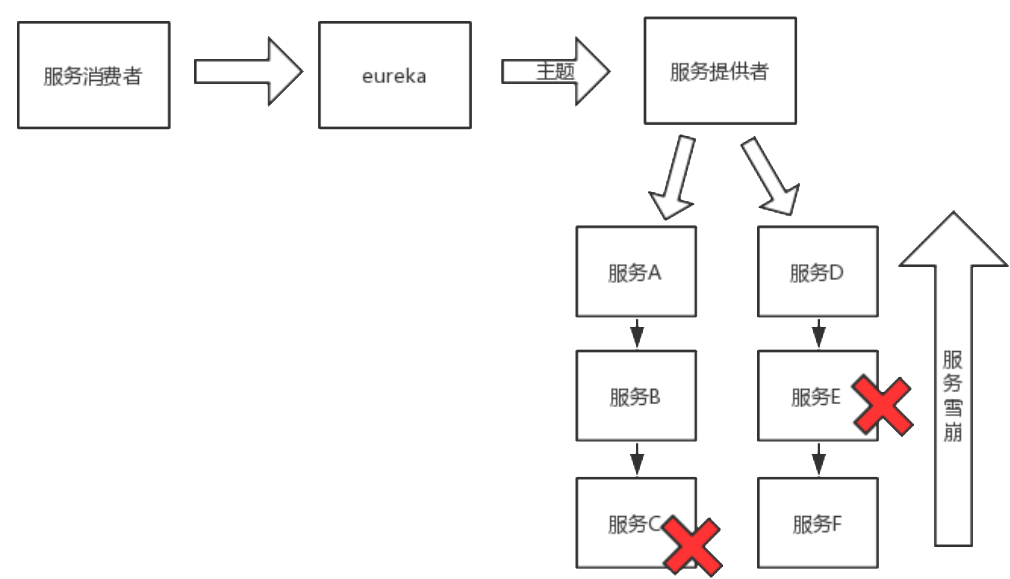
}

}

# Hystrix 熔断机制

在分布式环境下，微服务之间不可避免的发生互相调用的情况，但是没有一个系统是能保证自身绝对正确的，在服务的调用过程中，很可能面临服务失败的问题，因此需要一个公共组件能够在服务通过网络请求访问其他微服务时，能对服务失效情况下有很强的容错能力，对微服务提供保护和监控。

Hystrix是netflix的一个开源项目，他能够在依赖服务失效的情况下，通过隔离系统依赖的方式，防止服务的级联失败（服务的雪崩）



对于服务的熔断机制，其实需要考虑两种情况

1. 服务提供方存活，但调用接口报错
2. 服务提供方本身就出问题了

## 服务提供方报错

其实这种情况类似于异常捕获机制，当出现异常，返回一个通用的接口报文

【microcloud-provider-product】 复制一份成为【microcloud-provider-product-hystrix】

【microcloud-provider-product-hystrix】修改pom文件，增加 Hystrix依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

【microcloud-provider-product-hystrix】 修改ProductController

package cn.enjoy.controller;

import cn.enjoy.service.IProductService;

import cn.enjoy.vo.Product;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

import org.springframework.cloud.client.discovery.DiscoveryClient;

import org.springframework.web.bind.annotation.\*;

import javax.annotation.Resource;

@RestController

@RequestMapping("/prodcut")

public class ProductController {

@Resource

private IProductService iProductService;

@Resource

private DiscoveryClient client ; // 进行Eureka的发现服务

@RequestMapping(value="/get/{id}")

@HystrixCommand(fallbackMethod = "getFallback")

public Object get(@PathVariable("id") long id) {

Product product = this.iProductService.get(id);

if(product == null) {

throw new RuntimeException("该产品已下架！") ;

}

return product;

}

public Object getFallback(@PathVariable("id") long id){

Product product = new Product();

product.setProductName("HystrixName");

product.setProductDesc("HystrixDesc");

product.setProductId(0L);

return product;

}

@RequestMapping(value="/add")

public Object add(@RequestBody Product product) {

return this.iProductService.add(product) ;

}

@RequestMapping(value="/list")

public Object list() {

return this.iProductService.list() ;

}

@RequestMapping("/discover")

public Object discover() { // 直接返回发现服务信息

return this.client ;

}

}

一旦 get()方法上抛出了错误的信息，那么就认为该服务有问题

会默认使用“@HystrixCommand”注解之中配置好的fallbackMethod 调用类中的指定方法，返回相应数据

【microcloud-provider-product-hystrix】修改启动类，增加对熔断的支持

package cn.enjoy;

import org.mybatis.spring.annotation.MapperScan;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@MapperScan("cn.enjoy.mapper")

@EnableEurekaClient

@EnableDiscoveryClient

@EnableCircuitBreaker

public class ProductHystrixApp {

public static void main(String[] args) {

SpringApplication.run(ProductHystrixApp.class,args);

}

}

测试：localhost:8080/prodcut/get/100 访问

## 服务器失连

在某些情况下，服务提供方并没有失效，但可能由于网络原因，服务的消费方并不能调用到服务接口，在这种情况下，直接在服务的提供方提供熔断机制依然还是不够的，这方面的处理需要在服务的消费方进行服务的回退（服务的降级）处理

**服务的熔断：熔断**指的是当服务的提供方不可使用的时候，程序不会出现异常，而会出现本地的操作调用，服务的熔断是在服务消费方实现的，在断网情况下服务提供方的任何处理都是没有意义的。

【microcloud-service】新增一个IProductClientService的失败调用(降级处理)

package cn.enjoy.service.fallback;

import cn.enjoy.service.IProductClientService;

import cn.enjoy.vo.Product;

import feign.hystrix.FallbackFactory;

import org.springframework.stereotype.Component;

import java.util.List;

@Component

public class IProductClientServiceFallbackFactory implements FallbackFactory<IProductClientService> {

@Override

public IProductClientService create(Throwable throwable) {

return new IProductClientService() {

@Override

public Product getProduct(long id) {

Product product = new Product();

product.setProductId(999999L);

product.setProductName("feign-hystrixName");

product.setProductDesc("feign-hystrixDesc");

return product;

}

@Override

public List<Product> listProduct() {

return null;

}

@Override

public boolean addPorduct(Product product) {

return false;

}

};

}

}

【microcloud-service】 修改IProductClientService，增加fallback配置

package cn.enjoy.service;

import cn.enjoy.feign.FeignClientConfig;

import cn.enjoy.service.fallback.IProductClientServiceFallbackFactory;

import cn.enjoy.vo.Product;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@FeignClient(name = "MICROCLOUD-PROVIDER-PRODUCT",configuration = FeignClientConfig.class,

fallbackFactory = IProductClientServiceFallbackFactory.class)

public interface IProductClientService {

@RequestMapping("/prodcut/get/{id}")

public Product getProduct(@PathVariable("id")long id);

@RequestMapping("/prodcut/list")

public List<Product> listProduct() ;

@RequestMapping("/prodcut/add")

public boolean addPorduct(Product product) ;

}

【microcloud-consumer-feign】 复制一份成为【microcloud-consumer-hystrix】模块

【microcloud-consumer-hystrix】 修改application.yml配置文件，启用hystrix配置

feign:

hystrix:

enabled: true

compression:

request:

enabled: true

mime-types: # 可以被压缩的类型

- text/xml

- application/xml

- application/json

min-request-size: 2048 # 超过2048的字节进行压缩

启动，服务提供者

访问：<http://localhost/consumer/product/get?id=1，能正常访问>

关闭，服务提供者

访问：[http://localhost/consumer/product/get?id=1，也能正常访问](http://localhost/consumer/product/get?id=1，能正常访问)

## HystrixDashboard

在hystrix里面提供一个Dashboard（仪表盘）的功能，他是一种监控的功能，可以利用它来进行整体服务的监控

新建一个模块【microcloud-consumer-hystrix-dashboard】

【microcloud-consumer-hystrix-dashboard】pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-consumer-hystrix-dashboard</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-provider-product-hystrix】 pom文件确保里面有健康检查模块

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

【microcloud-consumer-hystrix-dashboard】 修改application.yml配置文件

server:

port: 9001

【microcloud-consumer-hystrix-dashboard】 创建一个启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

@SpringBootApplication

@EnableHystrixDashboard

public class HystrixDashboardApp {

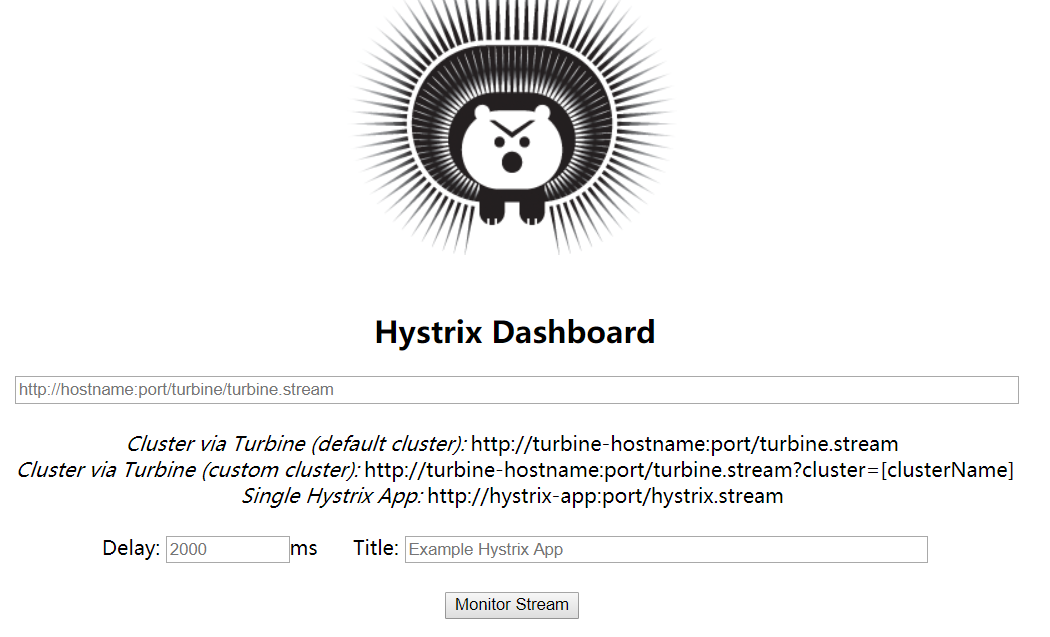
public static void main(String[] args) {

SpringApplication.run(HystrixDashboardApp.class,args);

}

}

启动运行：<http://localhost:9001/hystrix>



【microcloud-provider-product-hystrix】 修改applcation.yml文件

management:

endpoints:

web:

exposure:

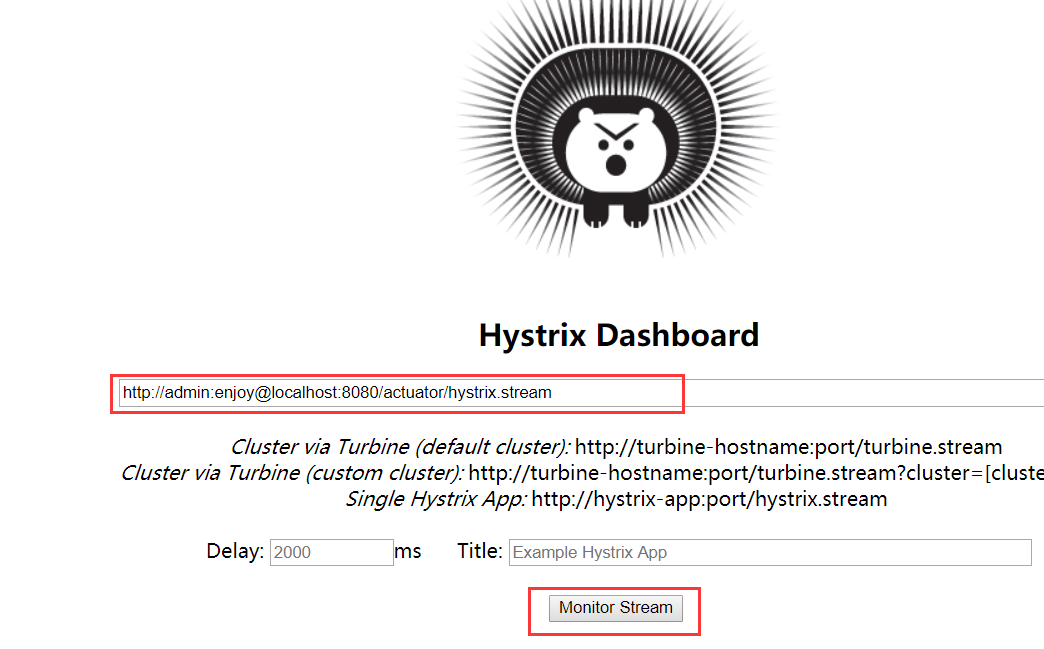
include: '\*'

【microcloud-provider-product-hystrix】启动

访问：localhost:8080/actuator/hystrix.stream

<http://localhost:9001/hystrix> 填写信息如下

<http://admin:enjoy@localhost:8080/actuator/hystrix.stream>



这个时候对localhost:8080的访问都可以被监控到

## Turbine

HystrixDashboard 前面已经知道了，它的主要功能是可以对某一项微服务进行监控，但真实情况下，不可能只对某一个服务进行监控，更多的是对很多服务进行一个整体的监控，这个时候就需要使用到turbine来完成了。

为了演示监控多个服务模块，这个时候新建一个模块【microcloud-provider-user-hystrix】,为简单起见，这个模块并不连接数据库，也不做安全控制。

【microcloud-provider-user-hystrix】pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-provider-user-hystrix</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</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-netflix-hystrix</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-api】新增一个VO类：Users

package cn.enjoy.vo;

import java.io.Serializable;

public class Users implements Serializable {

private String name;

private int age;

private String sex;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getSex() {

return sex;

}

public void setSex(String sex) {

this.sex = sex;

}

}

【microcloud-provider-user-hystrix】 新建一个UserController

package cn.enjoy.controller;

import cn.enjoy.vo.Users;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/users")

public class UserController {

@RequestMapping("/get/{name}")

@HystrixCommand

public Object get(@PathVariable("name")String name) {

Users users = new Users();

users.setName(name);

users.setAge(18);

users.setSex("F");

return users;

}

}

【microcloud-provider-user-hystrix】新增启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableCircuitBreaker

@EnableEurekaClient

public class UsersApp {

public static void main(String[] args) {

SpringApplication.run(UsersApp.class,args);

}

}

【microcloud-provider-user-hystrix】修改application.yml配置文件

server:

port: 8090

spring:

application:

name: microcloud-provider-users

logging:

level:

cn.enjoy.mapper: debug

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

instance:

instance-id: microcloud-provider-users

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

info:

app.name: microcloud-provider-users

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

management:

endpoints:

web:

exposure:

include: '\*'

启动后：

访问地址：http://localhost:8090/users/get/enjoy

hystrix监控地址：<http://localhost:8090/actuator/hystrix.stream>

前面准备工作完成后，如果想要实现 turbine 的配置，准备一个turbine模块

新增【microcloud-consumer-turbine】模块，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-consumer-turbine</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-turbine</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-consumer-turbine】修改application.yml配置文件

server:

port: 9101

eureka:

client:

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

turbine:

app-config: MICROCLOUD-PROVIDER-PRODUCT,MICROCLOUD-PROVIDER-USERS

cluster-name-expression: new String("default")

可以发现对于turbine，其实是从eureka配置在app-config中服务，然后进行监控

【microcloud-consumer-turbine】 新建一个启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.turbine.EnableTurbine;

@SpringBootApplication

@EnableTurbine

public class TurbineApp {

public static void main(String[] args) {

SpringApplication.run(TurbineApp.class,args);

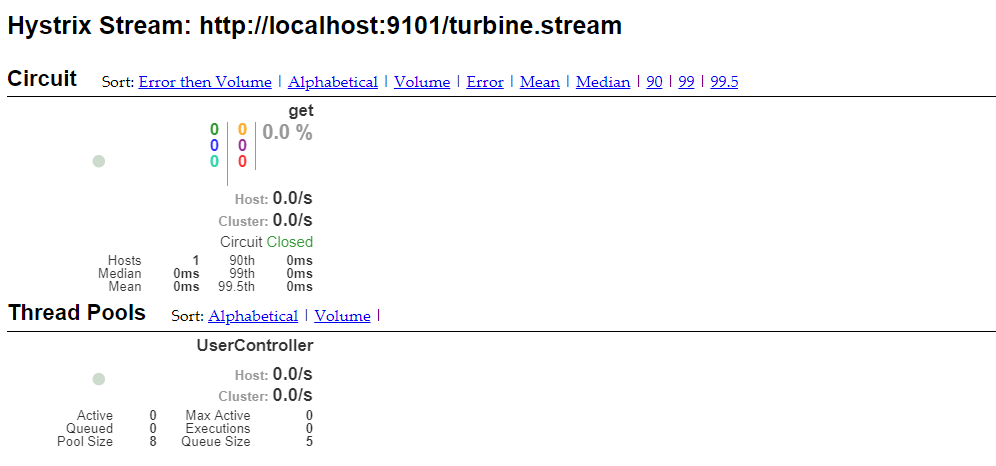
}

}

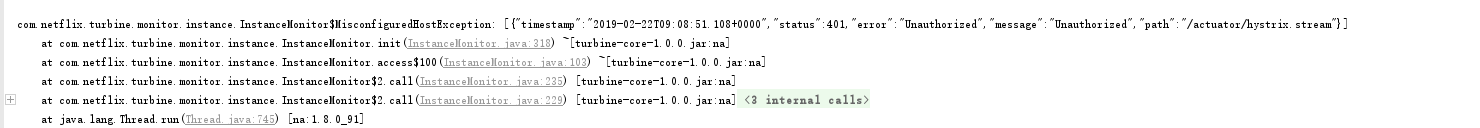
turbine监控地址：

启动Dashboard: <http://localhost:9001/hystrix>

在Dashboard里面填上 turbine监控地址



发现目前turbine只监控了UserController的信息，看下turbine后台发现报错



其实原因也很简单，User服务并不需要用户验证，所以能正常访问，但对于Product服务，配置了用户名密码的，turbine肯定无法访问

【microcloud-security】如果现在需要turbine进行加密服务的访问，那么只能折衷处理，让访问/actuator/hystrix.stream与/turbine.stream这两个地址的时候不需要用户密码验证

【microcloud-security】 修改WebSecurityConfiguration

package cn.enjoy.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.builders.WebSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration

@EnableWebSecurity

public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

@Override

public void configure(AuthenticationManagerBuilder auth)

throws Exception {

auth.inMemoryAuthentication().passwordEncoder(new BCryptPasswordEncoder()).withUser("root").password(new BCryptPasswordEncoder().encode("enjoy")).roles("USER").

and().withUser("admin").password(new BCryptPasswordEncoder().encode("enjoy")).roles("USER", "ADMIN");

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http.httpBasic().and().authorizeRequests().anyRequest()

.fullyAuthenticated();

http.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

}

@Override

public void configure(WebSecurity web) throws Exception {

web.ignoring().antMatchers("/actuator/hystrix.stream","/turbine.stream") ;

}

}

turbine监控地址：<http://localhost:9101/turbine.stream>

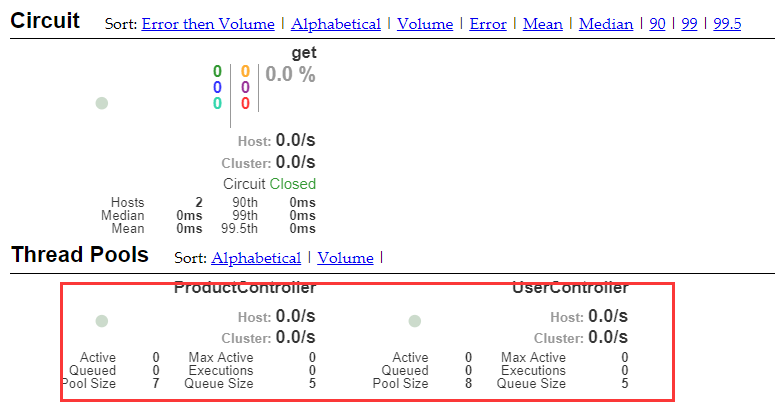
启动Dashboard: <http://localhost:9001/hystrix>

在Dashboard里面填上 turbine监控地址

刷新：

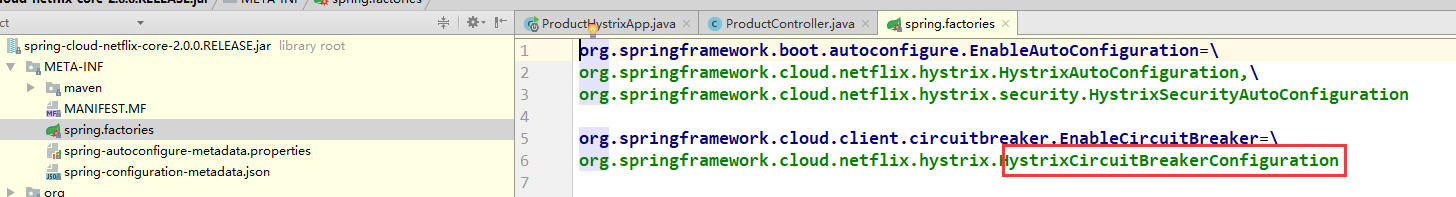
<http://localhost:8080/prodcut/get/1>

<http://localhost:8090/users/get/1>



## 原理解析：

### Hystrix 的AOP



@Bean

//注入一个AOP的切面

public HystrixCommandAspect hystrixCommandAspect() {

return new HystrixCommandAspect();

}

HystrixCommandAspect 用于处理被注解 @HystrixCommand 标记的方法。通过名字可以知道，Hystrix 基于 AOP 机制实现，对目标方法做了代理，然后实现了自己一系列功能特性。

@Pointcut("@annotation(com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand)")

public void hystrixCommandAnnotationPointcut() {

}

methodsAnnotatedWithHystrixCommand用来执行目标方法

@Around("hystrixCommandAnnotationPointcut() || hystrixCollapserAnnotationPointcut()")

public Object methodsAnnotatedWithHystrixCommand(ProceedingJoinPoint joinPoint) throws Throwable {

//根据切点获取Method（被@HystrixCommand标记方法）

Method method = AopUtils.getMethodFromTarget(joinPoint);

Validate.notNull(method, "failed to get method from joinPoint: %s", new Object[]{joinPoint});

if(method.isAnnotationPresent(HystrixCommand.class) && method.isAnnotationPresent(HystrixCollapser.class)) {

throw new IllegalStateException("method cannot be annotated with HystrixCommand and HystrixCollapser annotations at the same time");

} else {

HystrixCommandAspect.MetaHolderFactory metaHolderFactory = (HystrixCommandAspect.MetaHolderFactory)META\_HOLDER\_FACTORY\_MAP.get(HystrixCommandAspect.HystrixPointcutType.of(method));

MetaHolder metaHolder = metaHolderFactory.create(joinPoint);

// 准备各种材料后，创建HystrixInvokable

HystrixInvokable invokable = HystrixCommandFactory.getInstance().create(metaHolder);

ExecutionType executionType = metaHolder.isCollapserAnnotationPresent()?metaHolder.getCollapserExecutionType():metaHolder.getExecutionType();

try {

Object result;

if(!metaHolder.isObservable()) {

// 利用工具CommandExecutor来执行

result = CommandExecutor.execute(invokable, executionType, metaHolder);

} else {

result = this.executeObservable(invokable, executionType, metaHolder);

}

return result;

} catch (HystrixBadRequestException var9) {

throw (Throwable)(var9.getCause() != null?var9.getCause():var9);

} catch (HystrixRuntimeException var10) {

throw this.hystrixRuntimeExceptionToThrowable(metaHolder, var10);

}

}

}

HystrixInvokable 只是一个空接口，没有任何方法，只是用来标记具备可执行的能力。

#### 创建 HystrixInvokable

那 HystrixInvokable 又是如何创建的？它具体的实现类又是什么？先看看 HystrixCommandFactory.getInstance().create() 的代码。

public HystrixInvokable create(MetaHolder metaHolder) {

Object executable;

if(metaHolder.isCollapserAnnotationPresent()) {

executable = new CommandCollapser(metaHolder);

} else if(metaHolder.isObservable()) {

executable = new GenericObservableCommand(HystrixCommandBuilderFactory.getInstance().create(metaHolder));

} else {

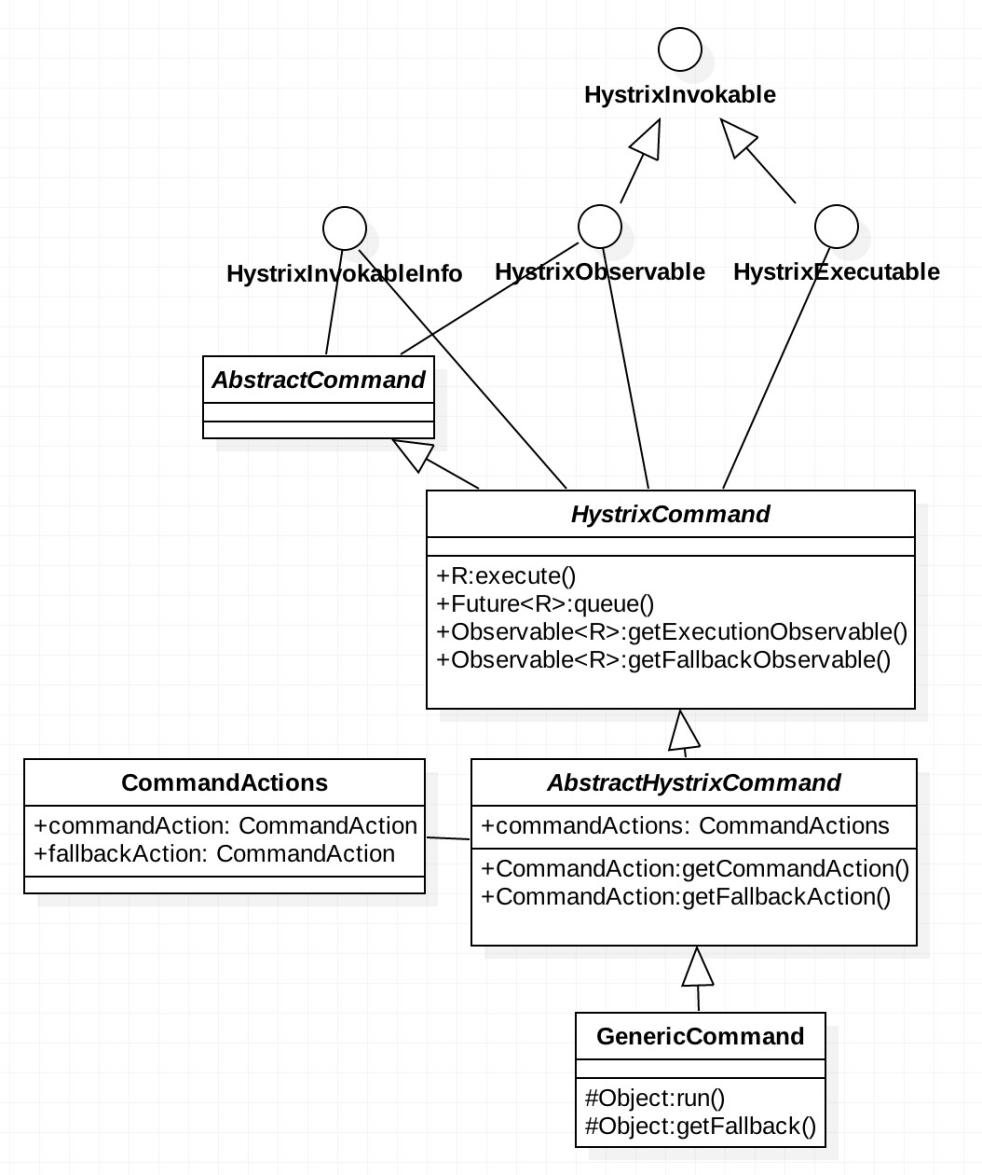
executable = new GenericCommand(HystrixCommandBuilderFactory.getInstance().create(metaHolder));

}

return (HystrixInvokable)executable;

}

这里可以看到实现类是 GenericCommand



三个抽象父类 AbstractHystrixCommand、HystrixCommand、AbstractCommand 帮助 GenericCommand 做了不少公共的事情，而 负责执行具体的方法和fallback时的方法。

com.netflix.hystrix.contrib.javanica.command.GenericCommand

// 执行具体的方法

protected Object run() throws Exception {

LOGGER.debug("execute command: {}", this.getCommandKey().name());

return this.process(new AbstractHystrixCommand<Object>.Action() {

Object execute() {

return GenericCommand.this.getCommandAction().execute(GenericCommand.this.getExecutionType());

}

});

}

// 执行fallback方法

protected Object getFallback() {

final CommandAction commandAction = this.getFallbackAction();

if(commandAction != null) {

try {

return this.process(new AbstractHystrixCommand<Object>.Action() {

Object execute() {

MetaHolder metaHolder = commandAction.getMetaHolder();

Object[] args = CommonUtils.createArgsForFallback(metaHolder, GenericCommand.this.getExecutionException());

return commandAction.executeWithArgs(metaHolder.getFallbackExecutionType(), args);

}

});

} catch (Throwable var3) {

LOGGER.error(FallbackErrorMessageBuilder.create().append(commandAction, var3).build());

throw new FallbackInvocationException(ExceptionUtils.unwrapCause(var3));

}

} else {

return super.getFallback();

}

}

#### 利用工具CommandExecutor来执行

public static Object execute(HystrixInvokable invokable, ExecutionType executionType, MetaHolder metaHolder) throws RuntimeException {

Validate.notNull(invokable);

Validate.notNull(metaHolder);

switch (executionType) {

//以同步方式

case SYNCHRONOUS: {

// 转为 HystrixExecutable 并执行

return castToExecutable(invokable, executionType).execute();

}

//以异步方式

case ASYNCHRONOUS: {

HystrixExecutable executable = castToExecutable(invokable, executionType);

if (metaHolder.hasFallbackMethodCommand()

&& ExecutionType.ASYNCHRONOUS == metaHolder.getFallbackExecutionType()) {

return new FutureDecorator(executable.queue());

}

return executable.queue();

}

case OBSERVABLE: {

HystrixObservable observable = castToObservable(invokable);

return ObservableExecutionMode.EAGER == metaHolder.getObservableExecutionMode() ? observable.observe() : observable.toObservable();

}

default:

throw new RuntimeException("unsupported execution type: " + executionType);

}

}

com.netflix.hystrix.HystrixCommand#execute

public R execute() {

try {

return queue().get();

} catch (Exception e) {

throw Exceptions.sneakyThrow(decomposeException(e));

}

}

public Future<R> queue() {

final Future<R> delegate = toObservable().toBlocking().toFuture();

//利用 JUC 的 Future 来异步执行

final Future<R> f = new Future<R>() {.... };

if (f.isDone()) {

try {

//执行正常的业务方法

f.get();

return f;

} catch(Exception e) {

//异常处理

}

}

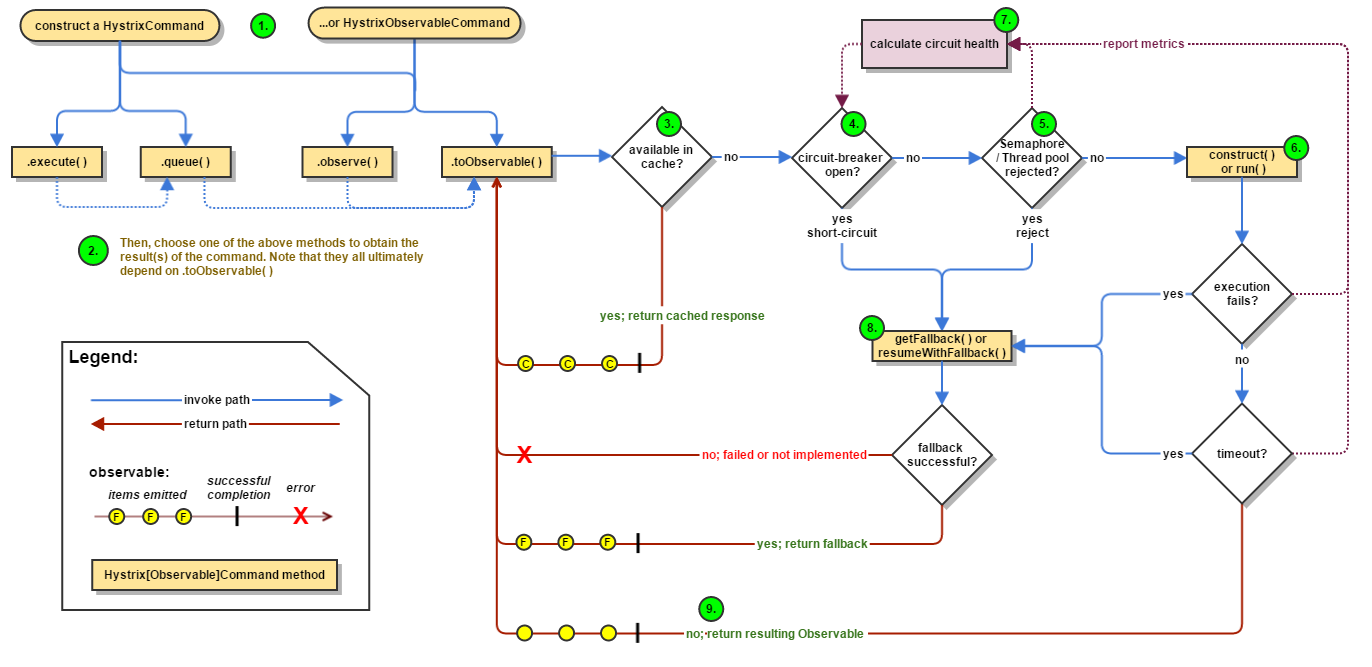
return f;

}

GenericCommand.run()

其实上述过程算是总览，可以知道Hystrix是通过AOP来实现的。

### 工作原理



1.构建命令

@HystrixCommand

2.执行命令

@com.netflix.hystrix.HystrixCommand#execute

3.检查缓存

如果启用了 Hystrix Cache，任务执行前将先判断是否有相同命令执行的缓存。如果有则直接返回缓存的结果；如果没有缓存的结果，但启动了缓存，将缓存本次执行结果以供后续使用。

4.检查断路器是否打开

断路器(circuit-breaker)和保险丝类似，保险丝在发生危险时将会烧断以保护电路，而断路器可以在达到我们设定的阀值时触发短路(比如请求失败率达到50%)，拒绝执行任何请求。

如果断路器被打开，Hystrix 将不会执行命令，直接进入Fallback处理逻辑

5.检查线程池/信号量情况

Hystrix 隔离方式有线程池隔离和信号量隔离。当使用Hystrix线程池时，Hystrix 默认为每个依赖服务分配10个线程，当10个线程都繁忙时，将拒绝执行命令。信号量同理

6.执行具体的任务

HystrixCommand.run() 来运行用户真正的任务

7.计算链路健康情况

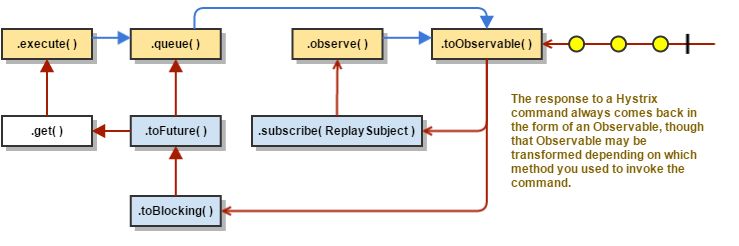
每次开始执行command、结束执行command以及发生异常等情况时，都会记录执行情况，例如：成功、失败、拒绝以及超时等情况，会定期处理这些数据，再根据设定的条件来判断是否开启断路器。

8.命令失败时执行 Fallback 逻辑

在命令失败时执行用户指定的 Fallback 逻辑。上图中的断路、线程池拒绝、信号量拒绝、执行执行、执行超时都会进入 Fallback 处理。

9.返回执行结果

原始结果将以Observable形式返回，在返回给用户之前，会根据调用方式的不同做一些处理。



com.netflix.hystrix.AbstractCommand#toObservable

public Observable<R> toObservable() {

final AbstractCommand<R> \_cmd = this;

// 命令执行结束后的清理者

final Action0 terminateCommandCleanup = new Action0() {...};

// 取消订阅时处理者

final Action0 unsubscribeCommandCleanup = new Action0() {...};

// Hystrix 核心逻辑: 断路器、隔离

final Func0<Observable<R>> applyHystrixSemantics = new Func0<Observable<R>>() {...};

// 发射数据(OnNext表示发射数据)时的Hook

final Func1<R, R> wrapWithAllOnNextHooks = new Func1<R, R>() {...};

// 命令执行完成的Hook

final Action0 fireOnCompletedHook = new Action0() {...};

// 通过Observable.defer()创建一个Observable

return Observable.defer(new Func0<Observable<R>>() {

@Override

public Observable<R> call() {

final boolean requestCacheEnabled = isRequestCachingEnabled();

final String cacheKey = getCacheKey();

// 首先尝试从请求缓存中获取结果

if (requestCacheEnabled) {

HystrixCommandResponseFromCache<R> fromCache = (HystrixCommandResponseFromCache<R>) requestCache.get(cacheKey);

if (fromCache != null) {

isResponseFromCache = true;

return handleRequestCacheHitAndEmitValues(fromCache, \_cmd);

}

}

// 使用上面的Func0：applyHystrixSemantics 来创建Observable

Observable<R> hystrixObservable =

Observable.defer(applyHystrixSemantics)

.map(wrapWithAllOnNextHooks);

Observable<R> afterCache;

// 如果启用请求缓存，将Observable包装成HystrixCachedObservable并进行相关处理

if (requestCacheEnabled && cacheKey != null) {

HystrixCachedObservable<R> toCache = HystrixCachedObservable.from(hystrixObservable, \_cmd);

...

} else {

afterCache = hystrixObservable;

}

// 返回Observable

return afterCache

.doOnTerminate(terminateCommandCleanup)

.doOnUnsubscribe(unsubscribeCommandCleanup)

.doOnCompleted(fireOnCompletedHook);

}

});

}

解释下Action0、Func1这种对象。Action、Func和Runnable、Callable类似，是一个可以被执行的实体。Action没有返回值，Action0…ActionN表示有0..N个参数，Action0就表示没有参数；Func有返值，0..N一样表示参数。

下面用核心的 applyHystrixSemantics 来阐述一下。

// applyHystrixSemantics 是一个Func0(理解为执行实体或处理者),表示没有参数,返回值是Observable。

final Func0<Observable<R>> applyHystrixSemantics = new Func0<Observable<R>>() {

@Override

public Observable<R> call() {

// 如果未订阅,返回一个"哑炮" Observable, 即一个不会发射任何数据的Observable

if (commandState.get().equals(CommandState.UNSUBSCRIBED)) {

return Observable.never();

}

// 调用applyHystrixSemantics()来创建Observable

return applyHystrixSemantics(\_cmd);

}

};

toObservable 到底做了什么？

其实就是主备大量的处理中（观察者），实际使用时是最后的Observable.defer(new Func0>(){…}

Observable.defer

defer译为延迟，表示演讲者会等有观众来时才开始分享。Observable.defer() 就是说：必须有观察者订阅时，Observable 才开始发射数据。而defer()的参数是个Func0，是一个会返回Observable的执行实体。下面看看defer()：

return Observable.defer(new Func0<Observable<R>>() {

@Override

public Observable<R> call() {

// 再一次使用Observable.defer()技能，这次用的是applyHystrixSemantics这个Func0

Observable<R> hystrixObservable =

Observable.defer(applyHystrixSemantics)

.map(wrapWithAllOnNextHooks);

... // 此处忽略了请求缓存处理,上面已有提及

Observable<R> afterCache;

...

// 为Observable绑定几个特定事件的处理者,这都是上门创建的Action0

return afterCache

.doOnTerminate(terminateCommandCleanup)

.doOnUnsubscribe(unsubscribeCommandCleanup)

.doOnCompleted(fireOnCompletedHook);

}

});

private Observable<R> applyHystrixSemantics(final AbstractCommand<R> \_cmd) {

// 源码中有很多executionHook、eventNotifier的操作，这是Hystrix拓展性的一种体现。这里面啥事也没做，留了个口子，开发人员可以拓展

executionHook.onStart(\_cmd);

// 判断断路器是否开启

if (circuitBreaker.attemptExecution()) {

// 获取执行信号

final TryableSemaphore executionSemaphore = getExecutionSemaphore();

final AtomicBoolean semaphoreHasBeenReleased = new AtomicBoolean(false);

final Action0 singleSemaphoreRelease = new Action0() {...};

final Action1<Throwable> markExceptionThrown = new Action1<Throwable>() {...};

// 判断是否信号量拒绝

if (executionSemaphore.tryAcquire()) {

try {

// 处理隔离策略和Fallback策略

return executeCommandAndObserve(\_cmd)

.doOnError(markExceptionThrown)

.doOnTerminate(singleSemaphoreRelease)

.doOnUnsubscribe(singleSemaphoreRelease);

} catch (RuntimeException e) {

return Observable.error(e);

}

} else {

return handleSemaphoreRejectionViaFallback();

}

}

// 开启了断路器,执行Fallback

else {

return handleShortCircuitViaFallback();

}

}

private Observable<R> executeCommandAndObserve(final AbstractCommand<R> \_cmd) {

final HystrixRequestContext currentRequestContext = HystrixRequestContext.getContextForCurrentThread();

final Action1<R> markEmits = new Action1<R>() {...};

final Action0 markOnCompleted = new Action0() {...};

// 利用Func1获取处理Fallback的 Observable

final Func1<Throwable, Observable<R>> handleFallback = new Func1<Throwable, Observable<R>>() {

@Override

public Observable<R> call(Throwable t) {

circuitBreaker.markNonSuccess();

Exception e = getExceptionFromThrowable(t);

executionResult = executionResult.setExecutionException(e);

// 拒绝处理

if (e instanceof RejectedExecutionException) {

return handleThreadPoolRejectionViaFallback(e);

// 超时处理

} else if (t instanceof HystrixTimeoutException) {

return handleTimeoutViaFallback();

} else if (t instanceof HystrixBadRequestException) {

return handleBadRequestByEmittingError(e);

} else {

...

return handleFailureViaFallback(e);

}

}

};

final Action1<Notification<? super R>> setRequestContext ...

Observable<R> execution;

// 利用特定的隔离策略来处理

if (properties.executionTimeoutEnabled().get()) {

execution = executeCommandWithSpecifiedIsolation(\_cmd)

.lift(new HystrixObservableTimeoutOperator<R>(\_cmd));

} else {

execution = executeCommandWithSpecifiedIsolation(\_cmd);

}

return execution.doOnNext(markEmits)

.doOnCompleted(markOnCompleted)

// 绑定Fallback的处理者

.onErrorResumeNext(handleFallback)

.doOnEach(setRequestContext);

private Observable<R> executeCommandWithSpecifiedIsolation(final AbstractCommand<R> \_cmd) {

// 线程池隔离

if (properties.executionIsolationStrategy().get() == ExecutionIsolationStrategy.THREAD) {

// 再次使用 Observable.defer(), 通过执行Func0来得到Observable

return Observable.defer(new Func0<Observable<R>>() {

@Override

public Observable<R> call() {

// 收集metric信息

metrics.markCommandStart(commandKey, threadPoolKey, ExecutionIsolationStrategy.THREAD);

...

try {

... // 获取包裹实际Task的Observable

return getUserExecutionObservable(\_cmd);

} catch (Throwable ex) {

return Observable.error(ex);

}

...

}

// 绑定各种处理者

}).doOnTerminate(new Action0() {...})

.doOnUnsubscribe(new Action0() {...})

// 绑定超时处理者

.subscribeOn(threadPool.getScheduler(new Func0<Boolean>() {

@Override

public Boolean call() {

return properties.executionIsolationThreadInterruptOnTimeout().get() && \_cmd.isCommandTimedOut.get() == TimedOutStatus.TIMED\_OUT;

}

}));

}

// 信号量隔离，和线程池大同小异

else {

return Observable.defer(new Func0<Observable<R>>() {...}

}

}

这里出现了两种隔离方式

Thread Pools(线程池)

将各依赖服务的访问交由独立的线程池来处理，会为每个依赖服务创建一个线程池。

虽然可以起到很好的隔离作用，但也增加了计算开

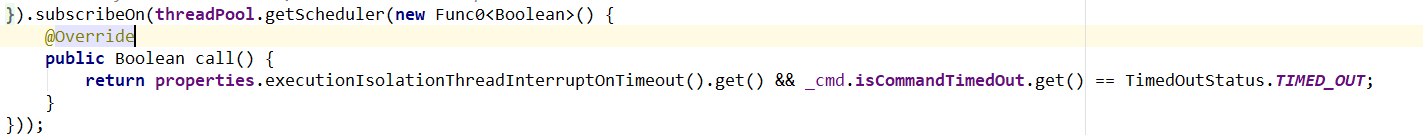
Semaphores(信号量)

通过为各依赖服务设置信号量(或计数器)来限制并发调用，相当于对各依赖服务做限流。信号量模式下任务由当前线程直接处理，不涉及到线程切换，自然也就没有超时控制。

线程池模式核心代码是这句：

subscribeOn(threadPool.getScheduler(new Func0<Boolean>()...

使用Scheduler来处理当前任务



.subscribeOn(threadPool.getScheduler(new Func0<Boolean>() {

@Override

public Boolean call() {

return properties.executionIsolationThreadInterruptOnTimeout().get() && \_cmd.isCommandTimedOut.get() == TimedOutStatus.TIMED\_OUT;

}

}));

threadPool 为HystrixThreadPool类型

com.netflix.hystrix.AbstractCommand#initThreadPool 里面初始化的

private static HystrixThreadPool initThreadPool(HystrixThreadPool fromConstructor, HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter threadPoolPropertiesDefaults) {

if (fromConstructor == null) {

// 每个threadPoolKey会维护一个HystrixThreadPool

return HystrixThreadPool.Factory.getInstance(threadPoolKey, threadPoolPropertiesDefaults);

} else {

return fromConstructor;

}

}

static class Factory {

final static ConcurrentHashMap<String, HystrixThreadPool> threadPools = new ConcurrentHashMap<String, HystrixThreadPool>();

static HystrixThreadPool getInstance(HystrixThreadPoolKey threadPoolKey, HystrixThreadPoolProperties.Setter propertiesBuilder) {

String key = threadPoolKey.name();

// this should find it for all but the first time

HystrixThreadPool previouslyCached = threadPools.get(key);

if (previouslyCached != null) {

return previouslyCached;

}

// if we get here this is the first time so we need to initialize

synchronized (HystrixThreadPool.class) {

if (!threadPools.containsKey(key)) {

threadPools.put(key, new HystrixThreadPoolDefault(threadPoolKey, propertiesBuilder));

}

}

return threadPools.get(key);

}

com.netflix.hystrix.AbstractCommand#applyHystrixSemantics

// 获取信号量

final TryableSemaphore executionSemaphore = getExecutionSemaphore();

com.netflix.hystrix.AbstractCommand#getExecutionSemaphore

protected TryableSemaphore getExecutionSemaphore() {

if (properties.executionIsolationStrategy().get() == ExecutionIsolationStrategy.SEMAPHORE) {

if (executionSemaphoreOverride == null) {

TryableSemaphore \_s = executionSemaphorePerCircuit.get(commandKey.name());

if (\_s == null) {

// we didn't find one cache so setup

executionSemaphorePerCircuit.putIfAbsent(commandKey.name(), **new** TryableSemaphoreActual(properties.executionIsolationSemaphoreMaxConcurrentRequests()));

// assign whatever got set (this or another thread)

return executionSemaphorePerCircuit.get(commandKey.name());

} else {

return \_s;

}

} else {

return executionSemaphoreOverride;

}

} else {

// return NoOp implementation since we're not using SEMAPHORE isolation

return TryableSemaphoreNoOp.DEFAULT;

}

}

只有在隔离策略为SEMAPHORE时，才会创建TryableSemaphoreActual，否则返回一个什么也不做的TryableSemaphoreNoOp(tryAcquire()将永远返回true)。

static class TryableSemaphoreActual implements TryableSemaphore {

protected final HystrixProperty<Integer> numberOfPermits;

private final AtomicInteger count = new AtomicInteger(0);

public TryableSemaphoreActual(HystrixProperty<Integer> numberOfPermits) {

// 每个HystrixCommandKey默认信号量数量，默认10

this.numberOfPermits = numberOfPermits;

}

@Override

public boolean tryAcquire() {

int currentCount = count.incrementAndGet();

// 如果信号量超过设定的信号量，则启动信号量拒绝

if (currentCount > numberOfPermits.get()) {

count.decrementAndGet();

return false;

} else {

return true;

}

}

@Override

public void release() {

count.decrementAndGet();

}

@Override

public int getNumberOfPermitsUsed() {

return count.get();

}

}

# Zuul路由

前面所有的微服务都是通过Eureka找到的，但是在很多开发中为了规范微服务的使用，提供有一个处理控制器Zuul

Zuul其实是一个API网关，类似于设计模式里面的Facade门面模式，他的存在就像是整个微服务的门面，所有的外部客户端访问都需要经过它来进行调度与过滤

## 基本使用

新建立一个模块【microcloud-zuul-gateway】

【microcloud-zuul-gateway】 的pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-zuul-gateway</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-zuul-gateway】修改application.yml文件

server:

port: 9501

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

register-with-eureka: false

spring:

application:

name: microcloud-zuul-gateway

【microcloud-zuul-gateway】 创建启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

import org.springframework.web.bind.annotation.RequestMapping;

@SpringBootApplication

@EnableZuulProxy

public class ZuulApp {

public static void main(String[] args) {

SpringApplication.run(ZuulApp.class,args);

}

}

启动：



发现启动报错，其实这是因为zuul目前对springboot2.1.2支持并不好，这也是zuul最近一直被人诟病的地方。

【springcloud】修改父工程pom文件，降低springboot版本为2.0.7.RELEASE

<dependency> <!-- SpringCloud离不开SpringBoot，所以必须要配置此依赖包 -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>2.0.7.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

重新启动ZuulApp

正常访问用户服务：<http://localhost:8090/users/get/1>

使用zuul代理访问用户服务：http://localhost:9501/microcloud-provider-users/users/get/1

## Zuul配置路由

前面以及简单的使用了zuul，但你会发现访问地址还必须知道程序的名称，如果不知道这个名称是无法访问的，但如果让用户知道了这名称，那么使用zuul就是去它的实际意义的，我们可以通过名称直接调用

既然是使用代理，那么代理的功能就是不能让用户看到真实的操作，屏蔽真实的调用地址，这个时候就需要自己增加zuul的路由规则配置了。

【microcloud-zuul-gateway】修改application.yml配置文件，增加路由配置

zuul:

routes:

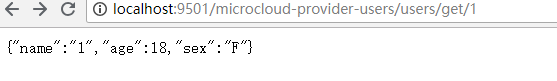
microcloud-provider-users: /users-proxy/\*\*

这个时候就可以通过/users-proxy 来访问microcloud-provider-users服务

http://localhost:9501/users-proxy/users/get/1

但是还会发现，虽然现在以及开启了路由访问的支持，但依然通过应用程序的名称还是能访问

<http://localhost:9501/microcloud-provider-users/users/get/1>



【microcloud-zuul-gateway】修改application.yml文件，忽略掉用户服务的名称

zuul:

routes:

microcloud-provider-users: /users-proxy/\*\*

ignored-services:

microcloud-provider-users

做完后，就可以进行代理的安全使用，但真实情况下，一般会有很多微服务，如果完全按照上面的配置方式会非常的麻烦，所有最加到的做法是可以采用一个通配符“\*”的模式来统一完成。

【microcloud-zuul-gateway】修改application.yml文件

zuul:

routes:

microcloud-provider-users: /users-proxy/\*\*

ignored-services:

"\*"

除开上面这一种访问模式以外，在zuul中还有另外一种配置方式

【microcloud-zuul-gateway】修改application.yml文件

zuul:

routes:

users.path: /users-proxy/\*\*

users.serviceId: microcloud-provider-users

ignored-services:

"\*"

其中在配置文件中出现的users其实是一个逻辑名称，这个名称主要作用是将path与serviceId绑定在一起

【microcloud-zuul-gateway】如果说不想通过eureka进行访问，对于zuul来说也是可以实现的，但是在真实的开发环境中，基本不会使用

zuul:

routes:

users:

path: /users-proxy/\*\*

serviceId: microcloud-provider-users

users2:

path: /users2-proxy/\*\*

url: http://localhost:8090/

ignored-services:

"\*"

访问：<http://localhost:9501/users2-proxy/users/get/1>

【microcloud-zuul-gateway】 设置公共前缀

zuul:

routes:

users:

path: /users-proxy/\*\*

serviceId: microcloud-provider-users

users2:

path: /users2-proxy/\*\*

url: http://localhost:8090/

ignored-services:

"\*"

prefix: /enjoy-api

一旦设置了公共前缀，所以的访问路径都要在前面加上前缀

<http://localhost:9501/enjoy-api/users-proxy/users/get/1>

<http://localhost:9501/enjoy-api/users2-proxy/users/get/1>

## zuul 过滤访问

其实zuul的功能本质上就是一个代理操作，类似于nginx，但是在真实的使用中，所有的微服务一点都有增加的认证信息，那么就必须在其访问之前追加认证的头部操作，这样的功能需要通过zuul的过去操作完成。

【microcloud-zuul-gateway】 修改application.yml配置，增加产品微服务

zuul:

routes:

users:

path: /users-proxy/\*\*

serviceId: microcloud-provider-users

users2:

path: /users2-proxy/\*\*

url: http://localhost:8090/

product:

path: /product-proxy/\*\*

serviceId: microcloud-provider-product

ignored-services:

"\*"

prefix: /enjoy-api

这样直接访问：http://localhost:9501/enjoy-api/product-proxy/prodcut/get/1

这样是访问不到的

【microcloud-zuul-gateway】追加过滤处理

package cn.enjoy.filter;

import com.netflix.zuul.ZuulFilter;

import com.netflix.zuul.context.RequestContext;

import com.netflix.zuul.exception.ZuulException;

import org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

import java.nio.charset.Charset;

import java.util.Base64;

public class AuthorizedRequestFilter extends ZuulFilter{

@Override

public String filterType() {

return FilterConstants.PRE\_TYPE;

}

@Override

public int filterOrder() {

return 0;

}

@Override

public boolean shouldFilter() {

return true;

}

@Override

public Object run() throws ZuulException {

RequestContext currentContext = RequestContext.getCurrentContext() ; // 获取当前请求的上下文

String auth = "admin:enjoy"; // 认证的原始信息

byte[] encodedAuth = Base64.getEncoder()

.encode(auth.getBytes(Charset.forName("US-ASCII"))); // 进行一个加密的处理

String authHeader = "Basic " + new String(encodedAuth);

currentContext.addZuulRequestHeader("Authorization", authHeader);

return null;

}

}

其中filterType为过滤的类型

在进行Zuul过滤的时候可以设置其过滤执行的位置，那么此时有如下几种类型：

* pre：在请求发出之前执行过滤，如果要进行访问，肯定在请求前设置头信息
* route：在进行路由请求的时候被调用；
* post：在路由之后发送请求信息的时候被调用；
* error：出现错误之后进行调用

【microcloud-zuul-gateway】建立一个配置程序类

package cn.enjoy.config;

import cn.enjoy.filter.AuthorizedRequestFilter;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class ZuulConfig {

@Bean

public AuthorizedRequestFilter getAuthorizedRequestFilter() {

return new AuthorizedRequestFilter() ;

}

}

这个时候访问：

<http://localhost:9501/enjoy-api/product-proxy/prodcut/get/1>

<http://localhost:9501/enjoy-api/users-proxy/users/get/1>

这两个服务都能正常访问了。

## Zuul安全访问

作为所有接口的统一门面，zuul也是可以进行加密访问的

【microcloud-zuul-gateway】修改pom文件，增加安全访问模块

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

【microcloud-zuul-gateway】修改application.yml配置文件，增加用户配置

spring:

application:

name: microcloud-zuul-gateway

security:

user:

name: admin

password: enjoy

再访问<http://localhost:9501/enjoy-api/users-proxy/users/get/1>

这个时候就需要输入用户名密码了

## Feign访问Zuul

前面学习feign的时候确实已经知道,他其实是去eureka中获取服务地址的，如果想使用feign来访问zuul，首先就应该让zuul注册到eureka中

【microcloud-zuul-gateway】 修改application.yml文件

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://admin:enjoy@eureka1:7001/eureka,http://admin:enjoy@eureka2:7002/eureka,http://admin:enjoy@eureka3:7003/eureka

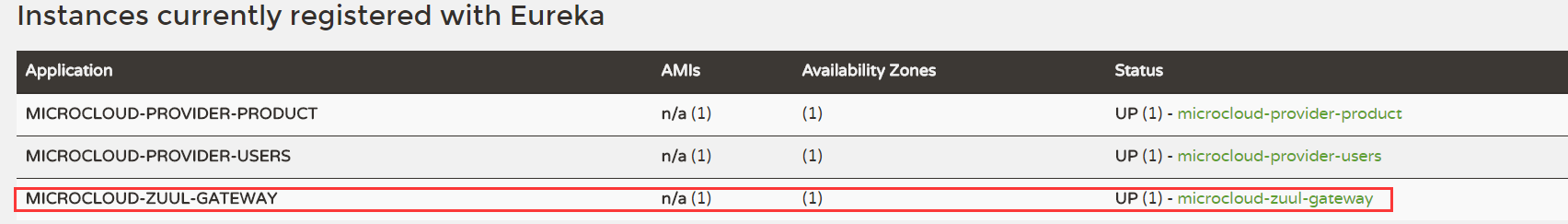
instance:

instance-id: microcloud-zuul-gateway

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）



【microcloud-service】现在所有的服务要通过zuul的代理进行访问，新增接口

package cn.enjoy.service;

import cn.enjoy.feign.FeignClientConfig;

import cn.enjoy.service.fallback.IProductClientServiceFallbackFactory;

import cn.enjoy.service.fallback.IZUUlClientServiceallbackFactory;

import cn.enjoy.vo.Product;

import cn.enjoy.vo.Users;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@FeignClient(name = "MICROCLOUD-ZUUL-GATEWAY",configuration = FeignClientConfig.class,

fallbackFactory = IZUUlClientServiceallbackFactory.class)

public interface IZUUlClientService {

@RequestMapping("/enjoy-api/product-proxy/prodcut/get/{id}")

public Product getProduct(@PathVariable("id")long id);

@RequestMapping("/enjoy-api/product-proxy/prodcut/list")

public List<Product> listProduct() ;

@RequestMapping("/enjoy-api/product-proxy/prodcut/add")

public boolean addPorduct(Product product) ;

@RequestMapping("/enjoy-api/users-proxy/users/get/{name}")

public Users getUsers(@PathVariable("name")String name);

}

新增IZUUlClientServiceallbackFactory，在Zuul由于出现网络问题失去联系后进行容错处理

package cn.enjoy.service.fallback;

import cn.enjoy.service.IProductClientService;

import cn.enjoy.service.IZUUlClientService;

import cn.enjoy.vo.Product;

import cn.enjoy.vo.Users;

import feign.hystrix.FallbackFactory;

import org.springframework.stereotype.Component;

import java.util.List;

@Component

public class IZUUlClientServiceallbackFactory implements FallbackFactory<IZUUlClientService> {

@Override

public IZUUlClientService create(Throwable throwable) {

return new IZUUlClientService() {

@Override

public Product getProduct(long id) {

Product product = new Product();

product.setProductId(999999L);

product.setProductName("feign-zuulName");

product.setProductDesc("feign-zuulDesc");

return product;

}

@Override

public List<Product> listProduct() {

return null;

}

@Override

public boolean addPorduct(Product product) {

return false;

}

@Override

public Users getUsers(String name) {

Users user = new Users();

user.setSex("F");

user.setAge(17);

user.setName("zuul-fllback："+name);

return user;

}

};

}

}

【microcloud-consumer-hystrix】 修改ConsumerProductController，增加一个新的方法，访问接口

package cn.enjoy.controller;

import cn.enjoy.service.IProductClientService;

import cn.enjoy.service.IZUUlClientService;

import cn.enjoy.vo.Product;

import org.springframework.cloud.client.ServiceInstance;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpMethod;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import javax.annotation.Resource;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

@RestController

@RequestMapping("/consumer")

public class ConsumerProductController {

@Resource

private IProductClientService iProductClientService;

@Resource

private IZUUlClientService izuUlClientService;

@RequestMapping("/product/get")

public Object getProduct(long id) {

return iProductClientService.getProduct(id);

}

@RequestMapping("/product/list")

public Object listProduct() {

return iProductClientService.listProduct();

}

@RequestMapping("/product/add")

public Object addPorduct(Product product) {

return iProductClientService.addPorduct(product);

}

@RequestMapping("/product/getProductAndUser")

public Object getProductAndUser(long id) {

Map<String,Object> result = new HashMap();

result.put("product",izuUlClientService.getProduct(id));

result.put("user",izuUlClientService.getUsers(id+""));

return result;

}

}

依次启动eureka，user服务，product服务，zuul服务，customerhystrix服务

在地址栏输入：

<http://localhost/consumer/product/getProductAndUser?id=1>

关闭zuul服务

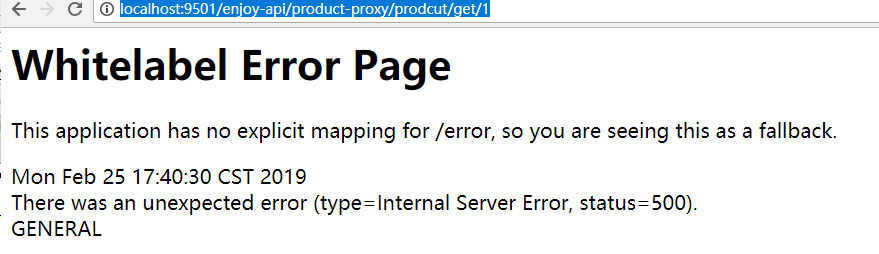
<http://localhost/consumer/product/getProductAndUser?id=1>

发现服务降级已经开启

## Zuul熔断

zuul是一个代理服务，但如果被代理的服务突然断了，这个时候zuul上面会有出错信息，例如，停止product服务

访问：<http://localhost:9501/enjoy-api/product-proxy/prodcut/get/1>



现在服务的调用方已经做了处理，不会出现这样的错误信息，但一般来说，对于zuul本身代理方，也应该进行zuul的降级处理

修改【microcloud-zuul-gateway】建立fallback回退处理类

package cn.enjoy.fallback;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.client.ClientHttpResponse;

import org.springframework.stereotype.Component;

import org.springframework.cloud.netflix.zuul.filters.route.FallbackProvider;

import java.io.ByteArrayInputStream;

import java.io.IOException;

import java.io.InputStream;

@Component

public class ProviderFallback implements FallbackProvider {

@Override

public String getRoute() {

return "\*";

}

@Override

public ClientHttpResponse fallbackResponse(String route, Throwable cause) {

return new ClientHttpResponse() {

@Override

public HttpHeaders getHeaders() {

HttpHeaders headers = new HttpHeaders() ;

headers.set("Content-Type", "text/html; charset=UTF-8");

return headers;

}

@Override

public InputStream getBody() throws IOException {

// 响应体

return new ByteArrayInputStream("产品微服务不可用，请稍后再试。".getBytes());

}

@Override

public HttpStatus getStatusCode() throws IOException {

return HttpStatus.BAD\_REQUEST;

}

@Override

public int getRawStatusCode() throws IOException {

return HttpStatus.BAD\_REQUEST.value();

}

@Override

public String getStatusText() throws IOException {

return HttpStatus.BAD\_REQUEST.getReasonPhrase();

}

@Override

public void close() {

}

};

}

}

访问：<http://localhost:9501/enjoy-api/product-proxy/prodcut/get/1>

getRoute：方法可以返回服务的ID，比如‘microcloud-provider-product’，如果需要匹配全部适应 “\*”

## 原理解析

1.引入依赖，在启动类中添加@EnableZuulProxy，声明这是一个Zuul代理。

2.注册到Eureka Server，启动服务，访问这个端口，url中带上要请求的服务名。

//开启断路器(hystrix)

@EnableCircuitBreaker

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(ZuulProxyMarkerConfiguration.class)

public @interface EnableZuulProxy {

}

org.springframework.cloud.netflix.zuul.ZuulProxyMarkerConfiguration

//让ZuulProxyMarkerConfiguration 配置生效

@Configuration

public class ZuulProxyMarkerConfiguration {

//创建Marker 实例，让ZuulProxyMarkerConfiguration 配置生效

@Bean

public Marker zuulProxyMarkerBean() {

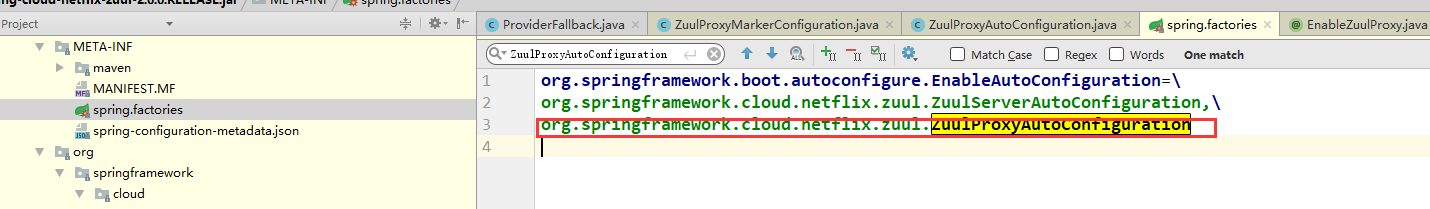
return new Marker();

}

class Marker {

}

}



org.springframework.cloud.netflix.zuul.ZuulProxyAutoConfiguration的父类是

org.springframework.cloud.netflix.zuul.ZuulServerAutoConfiguration

org.springframework.boot.web.servlet.ServletRegistrationBean

//注册一个zuulServlet

@Bean

@ConditionalOnMissingBean(name = "zuulServlet")

@ConditionalOnProperty(name = "zuul.use-filter", havingValue = "false", matchIfMissing = true)

public ServletRegistrationBean zuulServlet() {

ServletRegistrationBean<ZuulServlet> servlet = new ServletRegistrationBean<>(

new ZuulServlet(), this.zuulProperties.getServletPattern());

// The whole point of exposing this servlet is to provide a route that doesn't

// buffer requests.

servlet.addInitParameter("buffer-requests", "false");

return servlet;

}

//注册一个zuulServletFilter

@Bean

@ConditionalOnMissingBean(name = "zuulServletFilter")

@ConditionalOnProperty(name = "zuul.use-filter", havingValue = "true", matchIfMissing = false)

public FilterRegistrationBean zuulServletFilter() {

final FilterRegistrationBean<ZuulServletFilter> filterRegistration = new FilterRegistrationBean<>();

filterRegistration.setUrlPatterns(

Collections.singleton(this.zuulProperties.getServletPattern()));

filterRegistration.setFilter(new ZuulServletFilter());

filterRegistration.setOrder(Ordered.LOWEST\_PRECEDENCE);

// The whole point of exposing this servlet is to provide a route that doesn't

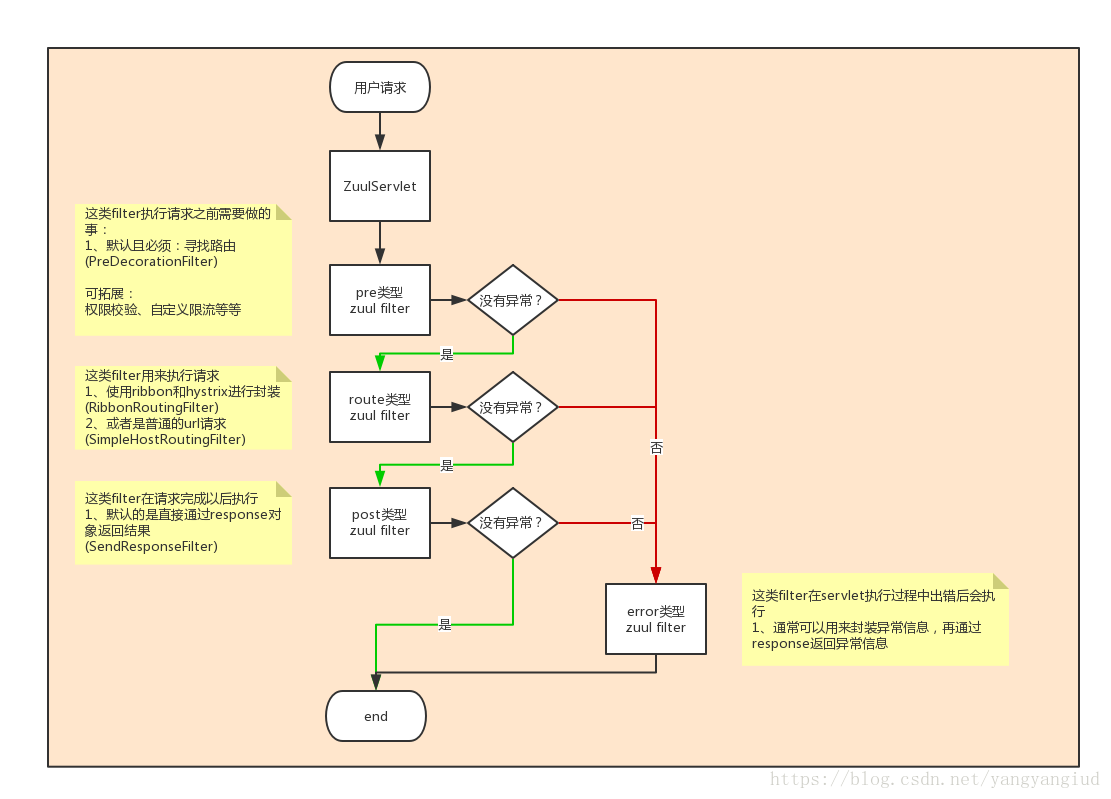
// buffer requests.

filterRegistration.addInitParameter("buffer-requests", "false");

return filterRegistration;

}

Zuulfilter 有四种 pre(前置),route(路由定位)，post(执行后)，error（出错），执行流程如上图，执行代码如下面



public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {

try {

this.init((HttpServletRequest)servletRequest, (HttpServletResponse)servletResponse);

RequestContext context = RequestContext.getCurrentContext();

context.setZuulEngineRan();

try {

//执行前置过滤器

this.preRoute();

} catch (ZuulException var12) {

//实现前置过滤器出错，则执行出错过滤器

this.error(var12);

//执行后置过滤

this.postRoute();

return;

}

try {

this.route();

} catch (ZuulException var13) {

this.error(var13);

this.postRoute();

return;

}

try {

this.postRoute();

} catch (ZuulException var11) {

this.error(var11);

}

} catch (Throwable var14) {

this.error(new ZuulException(var14, 500, "UNHANDLED\_EXCEPTION\_" + var14.getClass().getName()));

} finally {

RequestContext.getCurrentContext().unset();

}

}

不管是哪一种过滤器，最后都会落到

com.netflix.zuul.FilterProcessor#runFilters

public Object runFilters(String sType) throws Throwable {

if(RequestContext.getCurrentContext().debugRouting()) {

Debug.addRoutingDebug("Invoking {" + sType + "} type filters");

}

boolean bResult = false;

//据filter类型，获取排过序的filter

List<ZuulFilter> list = FilterLoader.getInstance().getFiltersByType(sType);

if(list != null) {

for(int i = 0; i < list.size(); ++i) {

//处理filter执行

ZuulFilter zuulFilter = (ZuulFilter)list.get(i);

Object result = this.processZuulFilter(zuulFilter);

if(result != null && result instanceof Boolean) {

bResult |= ((Boolean)result).booleanValue();

}

}

}

return Boolean.valueOf(bResult);

}

public Object processZuulFilter(ZuulFilter filter) throws ZuulException {

RequestContext ctx = RequestContext.getCurrentContext();

boolean bDebug = ctx.debugRouting();

String metricPrefix = "zuul.filter-";

long execTime = 0L;

String filterName = "";

try {

long ltime = System.currentTimeMillis();

filterName = filter.getClass().getSimpleName();

RequestContext copy = null;

Object o = null;

Throwable t = null;

if(bDebug) {

Debug.addRoutingDebug("Filter " + filter.filterType() + " " + filter.filterOrder() + " " + filterName);

copy = ctx.copy();

}

//真正执行ZuulFilter内存方法

ZuulFilterResult result = filter.runFilter();

ExecutionStatus s = result.getStatus();

execTime = System.currentTimeMillis() - ltime;

switch(null.$SwitchMap$com$netflix$zuul$ExecutionStatus[s.ordinal()]) {

case 1:

t = result.getException();

ctx.addFilterExecutionSummary(filterName, ExecutionStatus.FAILED.name(), execTime);

break;

case 2:

o = result.getResult();

ctx.addFilterExecutionSummary(filterName, ExecutionStatus.SUCCESS.name(), execTime);

if(bDebug) {

Debug.addRoutingDebug("Filter {" + filterName + " TYPE:" + filter.filterType() + " ORDER:" + filter.filterOrder() + "} Execution time = " + execTime + "ms");

Debug.compareContextState(filterName, copy);

}

}

if(t != null) {

throw t;

} else {

this.usageNotifier.notify(filter, s);

return o;

}

} catch (Throwable var15) {

if(bDebug) {

Debug.addRoutingDebug("Running Filter failed " + filterName + " type:" + filter.filterType() + " order:" + filter.filterOrder() + " " + var15.getMessage());

}

this.usageNotifier.notify(filter, ExecutionStatus.FAILED);

if(var15 instanceof ZuulException) {

throw (ZuulException)var15;

} else {

ZuulException ex = new ZuulException(var15, "Filter threw Exception", 500, filter.filterType() + ":" + filterName);

ctx.addFilterExecutionSummary(filterName, ExecutionStatus.FAILED.name(), execTime);

throw ex;

}

}

}

com.netflix.zuul.ZuulFilter#runFilter

public ZuulFilterResult runFilter() {

ZuulFilterResult zr = new ZuulFilterResult();

if (!isFilterDisabled()) {

if (shouldFilter()) {

Tracer t = TracerFactory.instance().startMicroTracer("ZUUL::" + this.getClass().getSimpleName());

try {

//执行，run方法就是具体filter 实现的run方法

Object res = run();

zr = new ZuulFilterResult(res, ExecutionStatus.SUCCESS);

} catch (Throwable e) {

t.setName("ZUUL::" + this.getClass().getSimpleName() + " failed");

zr = new ZuulFilterResult(ExecutionStatus.FAILED);

zr.setException(e);

} finally {

t.stopAndLog();

}

} else {

zr = new ZuulFilterResult(ExecutionStatus.SKIPPED);

}

}

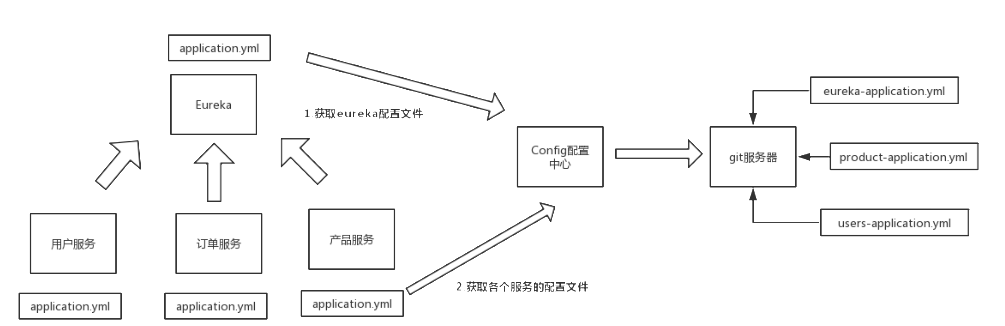
return zr;

}

# SpringCloudConfig分布式配置中心

讲解Zookeeper的时候其实实现过分布式的配置中心，springcloudconfig的核心作用其实就是在于对配置进行管理

虽然springcloud使用springboot进行开发，节省了大量的配置文件，但每个服务依然有自己的application.yml配置文件，而且每个服务一般都有负载均衡，所以，这么依赖对于配置文件的统一管理就非常有必要了。



上图是springcloudconfig总体结构图。

左边这一块我们很熟悉，最开始有个eureka，它通过配置文件application.yml启动，在这个配置文件里面会指定端口，实例名，注册地址等

对于服务提供商来说，它也需要把相关信息写到application.yml文件中，比如数据库配置，端口，项目名称等，其中最重要的就就是要指定eureka的具体位置

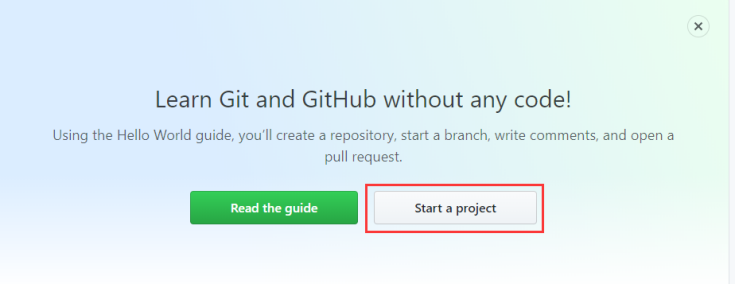
这是前面反复说过的，但现在是基于cloudConfig的配置中心，最开始启动eureka的时候，eureka的具体配置就不是写死在eureka的application.yml文件中了，这个时候也会有application.yml（bootstrap.yml）配置文件，只是这里的配置指定的时候config的配置中心，在eureka启动的时候读取【配置中心】的配置，并启动

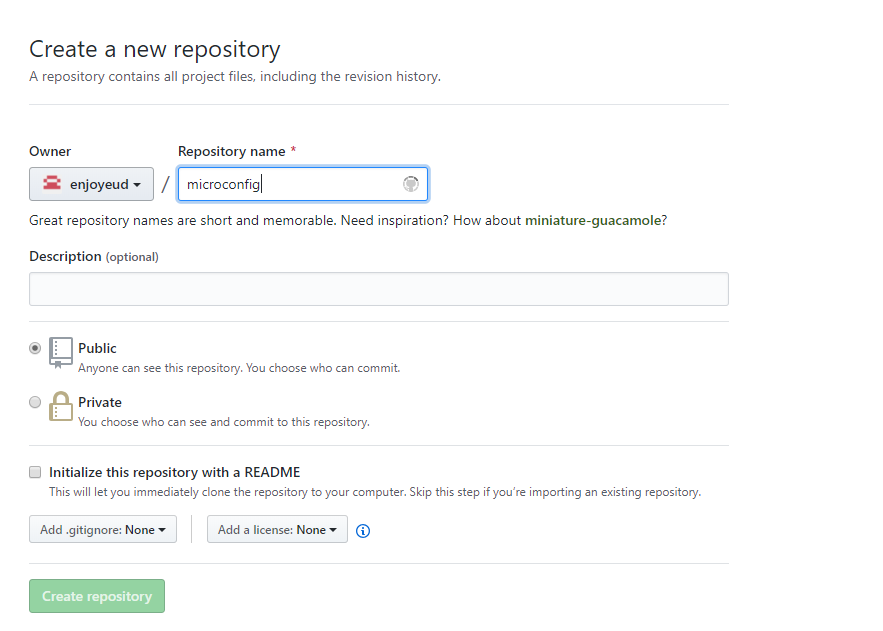
对于服务提供商也是同样的道理，以产品服务为例，它的application.yml文件也不在指定具体的配置，真实需要访问的数据库，端口等信息也是在启动的时候直接从【配置中心】读取。

所以说config配置中心在其中占据非常重要的位置，但config里面的配置从哪来呢？其实是从git服务器里面来的，开发者需要把相关的配置上传到git服务器，这里的git服务器可以自己搭建，也可以直接用github，后面项目为了方便就直接使用github了。

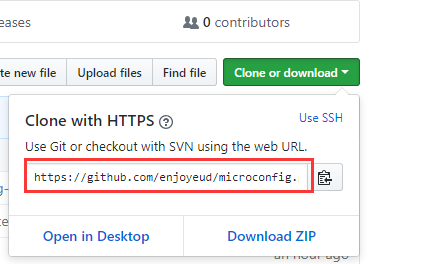
## 准备github账号

<https://github.com/> 注册就不说了

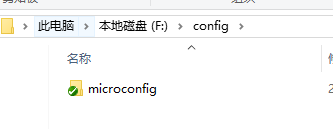




获得git的地址



输入用户名密码：把新建的仓库信息下载倒本地



这在里我放倒了f盘的config目录中。

## 配置中心搭建

有了git服务器之后，接下来就要准备配置中心了

【microcloud-config】 新建一个配置中心的服务提供模块，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-config</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-security</artifactId>

</dependency>

</dependencies>

</project>

引入springcloudserver的同时，这分布式配置中心也不是谁都能访问的，所以增加了安全验证模块。

应该还记得这时候的用户名密码为：admin/enjoy

【microcloud-config】 新增application.yml配置文件，增加git连接配置信息

server:

port: 7101

spring:

application:

name: microcloud-config

cloud:

config:

server:

git:

uri: https://github.com/enjoyeud/microconfig.git

【microcloud-config】 新增启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableConfigServer

public class ConfigApp {

public static void main(String[] args) {

SpringApplication.run(ConfigApp.class,args);

}

}

这些工作准备完成后，为了测试【microcloud-config】是能正确运行的，还需要上传个配置文件到github

【GITHUB】 上传一个application.yml

spring:

profiles:

active:

- dev

---

spring:

profiles: dev

application:

name: microconfig-test-dev

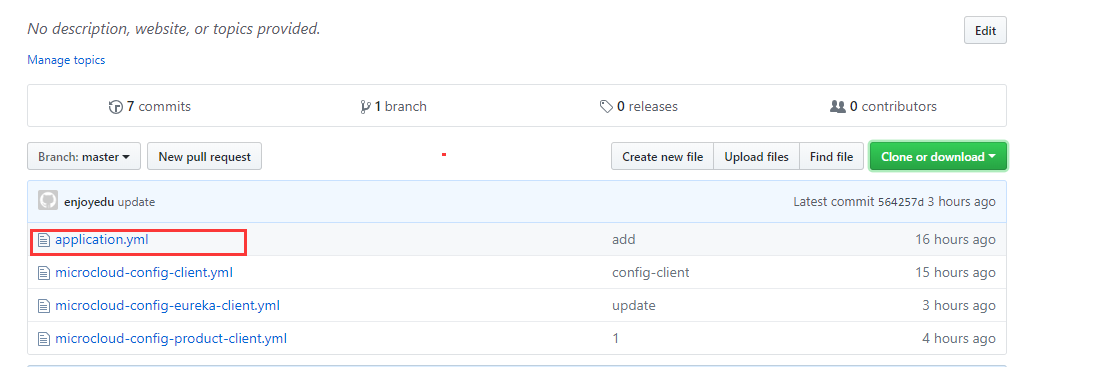
---

spring:

profiles: default

application:

name: microconfig-test-default



准备好配置文件后启动【microcloud-config】

|  |  |  |
| --- | --- | --- |
| NO | 访问形式 | 访问路径 |
| 1 | /{application}-{profile}.yml | <http://localhost:7101/application-dev.yml>  <http://localhost:7101/application-default.yml>  <http://localhost:7101/application-beta.yml> 不存在 |
| 2 | /{application}/{profile}[/{label}] | <http://localhost:7101/application/dev/master>  http://localhost:7101/application/default/master |
| 3 | /{label}/{application}-{profile}.yml | <http://localhost:7101/master/application-default.yml>  http://localhost:7101/master/application-dev.yml |

## 简单的客户端

现在已经成功的搭建好了配置中心，但这个时候如果只通过url的访问形式其实没什么太多的意义，最终还是需要把github相关信息加载到客户端上进行访问

新增加一个【microcloud-config-client】模块，这模块讲读取github里面的信息，也不做其他的事情，只是显示一下。

新建【microcloud-config-client】，pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-config-client</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

</project>

【github】上传一个新的配置文件microcloud-config-client.yml，后面就通过程序来读取这个配置

spring:

profiles:

active:

- dev

---

server:

port: 8201

spring:

profiles: dev

application:

name: microconfig-test-client

eureka:

client:

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

---

server:

port: 8102

spring:

profiles: beta

application:

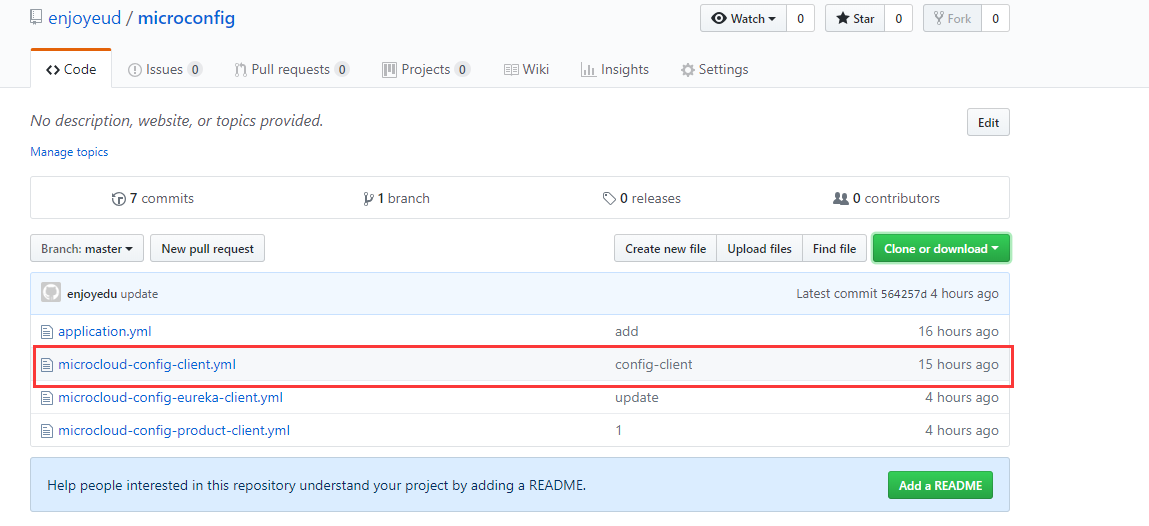
name: microconfig-test-client

eureka:

client:

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka



【microcloud-config-client】新建bootstrap.yml文件，这文件读取配置中心的配置

spring:

cloud:

config:

name: microcloud-config-client # 定义要读取的资源文件的名称

profile: dev # 定义profile的 名称

label: master # 定义配置文件所在的分支

uri: http://localhost:7101 # SpringCloudConfig的服务地址

username: admin # 连接的用户名

password: enjoy # 连接的密码

可能有些人奇怪，为什么不直接把相关信息写道application.yml文件之中，其实这是一种规范

* “application.yml”：对应的是用户级的资源配置项；
* “bootstrap.yml”：对应的是系统级的资源配置，其优先级更高

【microcloud-config-client】新建application.yml文件，这文件只是简单的配置一个应用名称

spring:

application:

name: microcloud-config-client # 编写应用的名称

【microcloud-config-client】新建一个controller,这个controller显示从服务器下载到的配置文件

package cn.enjoy.controller;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class ConfigClientController {

@Value("${spring.application.name}")

private String applicationName;

@Value("${eureka.client.serviceUrl.defaultZone}")

private String eurekaServers;

@RequestMapping("/config")

public String getConfig() {

return "ApplicationName = " + this.applicationName + "、EurekaServers = "

+ this.eurekaServers;

}

}

【microcloud-config-client】 新建一个启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ConfigClientApp {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApp.class,args);

}

}

启动配置中心，访问，确认配置通过url是能正常访问的。

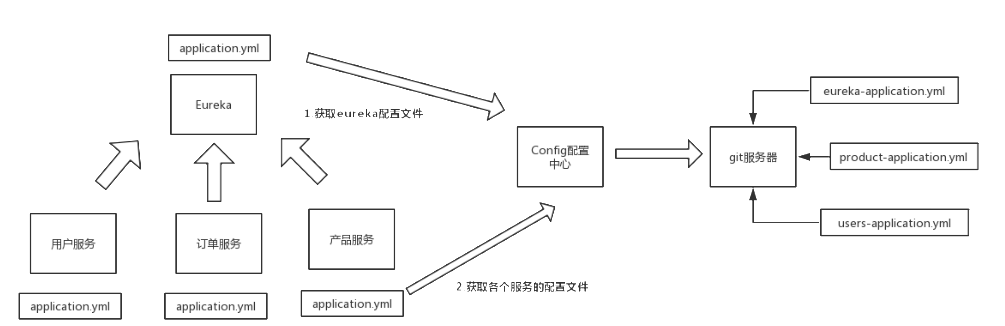
<http://localhost:7101/microcloud-config-client-beta.yml>

启动【microcloud-config-client】发现tomcat启动了，占用的端口就是dev的8201，访问

<http://localhost:8201/config>

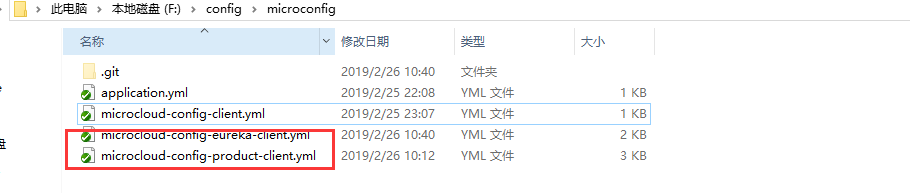
这个时候简单的客户端已经搭建完成。

## Eureka与服务提供商读取配置



有了上面这些基础，接下来就可以完成这个图的功能了，这里依然简化一下，只考虑product产品服务与eureka

eureka与product服务的配置信息要求去配置中心获取，所以在正式部署项目之前，先准备两个配置，上传到github之中



microcloud-config-eureka-client.yml，这个是eureka的配置文件，这里就没有考虑eureka的高可用了

spring:

profiles:

active:

- dev

---

server:

port: 7001

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

instance: # eureak实例定义

hostname: localhost # 定义 Eureka 实例所在的主机名称

spring:

profiles: dev

security:

user:

name: admin

password: enjoy

application:

name: microcloud-config-eureka-client

---

server:

port: 7002

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@localhost:7002/eureka

instance: # eureak实例定义

hostname: localhost # 定义 Eureka 实例所在的主机名称

spring:

profiles: beta

security:

user:

name: admin

password: enjoy

application:

name: microcloud-config-eureka-client

microcloud-config-product-client.yml，这个事对于产品服务这个服务提供商提供者的配置文件

spring:

profiles:

active:

- dev

---

server:

port: 8080

mybatis:

mapper-locations: # 所有的mapper映射文件

- classpath:mapping/\*.xml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource # 配置当前要使用的数据源的操作类型

driver-class-name: com.mysql.jdbc.Driver # 配置MySQL的驱动程序类

url: jdbc:mysql://localhost:3306/springcloud?serverTimezone=GMT%2B8 # 数据库连接地址

username: root # 数据库用户名

password: root1234% # 数据库连接密码

application:

name: microcloud-config-product-client

profiles: dev

logging:

level:

cn.enjoy.mapper: debug

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

instance:

instance-id: microcloud-config-product-client

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

info:

app.name: microcloud-provider-product

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

management:

endpoints:

web:

exposure:

include: '\*'

---

server:

port: 8081

mybatis:

mapper-locations: # 所有的mapper映射文件

- classpath:mapping/\*.xml

spring:

datasource:

type: com.alibaba.druid.pool.DruidDataSource # 配置当前要使用的数据源的操作类型

driver-class-name: com.mysql.jdbc.Driver # 配置MySQL的驱动程序类

url: jdbc:mysql://localhost:3306/springcloud?serverTimezone=GMT%2B8 # 数据库连接地址

username: root # 数据库用户名

password: root1234% # 数据库连接密码

application:

name: microcloud-config-product-client

profiles: beta

logging:

level:

cn.enjoy.mapper: debug

eureka:

client: # 客户端进行Eureka注册的配置

service-url:

defaultZone: http://admin:enjoy@localhost:7002/eureka

instance:

instance-id: microcloud-config-product-client

prefer-ip-address: true

lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）

lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）

info:

app.name: microcloud-config-product-client

company.name: enjoy

build.artifactId: $project.artifactId$

build.modelVersion: $project.modelVersion$

management:

endpoints:

web:

exposure:

include: '\*'

有了这两个配置文件，接下来就可以搭建eureka服务和product服务了

### eureka配置

复制【microcloud-eureka】一份，修改成为【microcloud-config-eureka-client】，

【microcloud-config-eureka-client】 修改pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-config-eureka-client</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-config-eureka-client】创建bootstrap.yml文件，读取配置中心eureka的配置

spring:

cloud:

config:

uri: http://localhost:7101

name: microcloud-config-eureka-client

profile: beta

label: master

username: admin

password: enjoy

【microcloud-config-eureka-client】 修改application.yml，删除不需要的配置

spring:

application:

name: microcloud-config-eureka-client

由于使用的是beta，它里面指定的eureka的端口是7002

重启后访问：localhost:7002

### product服务配置

复制【microcloud-provider-product】项目为【microcloud-config-product-client】

【microcloud-config-product-client】pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<version>1.0.0</version>

<artifactId>microcloud-config-product-client</artifactId>

<dependencies>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<!--<dependency>-->

<!--<groupId>org.springframework.boot</groupId>-->

<!--<artifactId>spring-boot-starter-security</artifactId>-->

<!--</dependency>-->

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</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-config</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-config-product-client】新增bootstrap.yml 文件，配置如下

spring:

cloud:

config:

uri: http://localhost:7101

name: microcloud-config-product-client

profile: beta

label: master

username: admin

password: enjoy

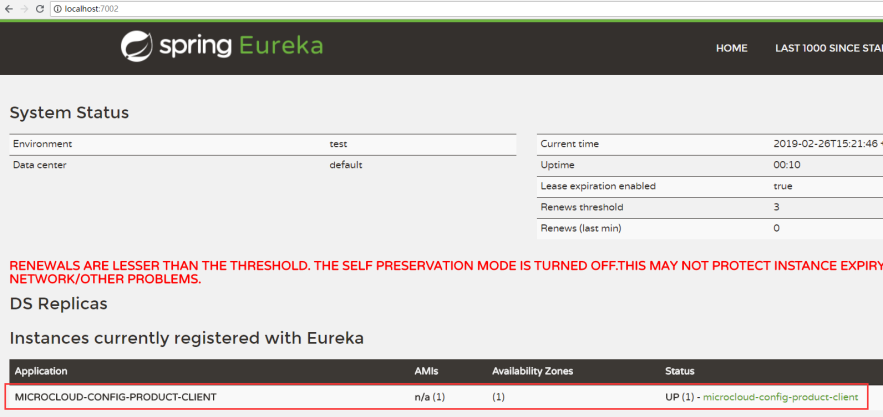
【microcloud-config-product-client】 修改application.yml文件

spring:

application:

name: microcloud-config-product-client

启动product服务，访问eureka，现在产品服务已经添加上去了。



## Config配置中心高可用

现在不管是erueka还是服务提供者都是基于SpringCloudConfig获取配置文件的，这个时候配置中心就至关重要了，但在真实的项目环境中，难免SpringCloudConfig会出现各种问题，这个时候就需要考虑config的高可用机制了。

其实解决方式也很简单，把SpringCloudConfig注册到Eureka就搞定了，这个时候用户访问的时候不是直接从配置中心获取配置，而是通过eureka中获取配置中心的地址，再从配置中心获取具体服务的参数就行。

复制【microcloud-eureka】一份，修改成为【microcloud-ha-config-eureka】，这个eureka不注册具体的业务服务，只是负责config配置中心的负载均衡使用

【microcloud-ha-config-eureka】pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-ha-config-eureka</artifactId>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-ha-config-eureka】 修改application.yml文件

server:

port: 7301

eureka:

server:

eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）

enable-self-preservation: false #设置为false表示关闭保护模式

client:

fetch-registry: false

register-with-eureka: false

service-url:

defaultZone: http://admin:enjoy@localhost:7301/eureka

instance: # eureak实例定义

hostname: localhost # 定义 Eureka 实例所在的主机名称

spring:

security:

user:

name: admin

password: enjoy

application:

name: microcloud-ha-config-eureka

启动类如下

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class HaConfigEurekaApp {

public static void main(String[] args) {

SpringApplication.run(HaConfigEurekaApp.class,args);

}

}

【microcloud-config】再复制两份，总共3个配置中心，分别为【microcloud-config2】【microcloud-config3】

【microcloud-config】【microcloud-config2】【microcloud-config3】 修改pom文件

增加eureka的支持

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

【microcloud-config】【microcloud-config2】【microcloud-config3】修改application.yml文件，增加eureka的注册地址

server:

port: 7101

spring:

application:

name: microcloud-config

cloud:

config:

server:

git:

uri: https://github.com/enjoyeud/microconfig.git

eureka:

client:

service-url:

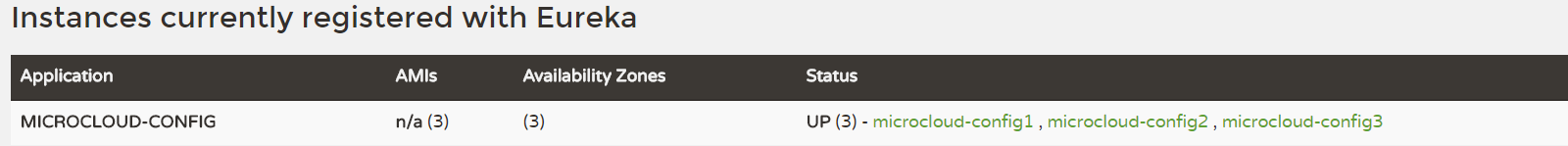
defaultZone: http://admin:enjoy@localhost:7301/eureka

instance:

prefer-ip-address: true # 在地址栏上使用IP地址进行显示

instance-id: microcloud-config1

启动eureka并启动三个配置中心后



【microcloud-config-client】 修改pom文件，增加eureka的支持

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-config-client</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</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-netflix-eureka-client</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-config-client】 修改bootstrap.yml文件，增加eureka相关配置

spring:

cloud:

config:

name: microcloud-config-client # 定义要读取的资源文件的名称

profile: dev # 定义profile的 名称

label: master # 定义配置文件所在的分支

#uri: http://localhost:7101 # SpringCloudConfig的服务地址

username: admin # 连接的用户名

password: enjoy # 连接的密码

discovery:

enabled: true # 通过配置中心加载配置文件

service-id: MICROCLOUD-CONFIG # 在eureka之中注册的服务ID

eureka:

client:

service-url:

defaultZone: http://admin:enjoy@localhost:7301/eureka

访问：<http://localhost:8201/config>

## 自动刷新

在整个SpringCloudConfig设计之中，我们已经实现了配置的统一管理，但其实还有一个问题，就是自动刷新。

尝试修改一下【github】 microcloud-config-client.yml文件

spring:

profiles:

active:

- dev

---

server:

port: 8201

spring:

profiles: dev

application:

name: microconfig-test-client2

eureka:

client:

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

---

server:

port: 8102

spring:

profiles: beta

application:

name: microconfig-test-client2

eureka:

client:

service-url:

defaultZone: http://admin:enjoy@localhost:7001/eureka

这里的修改非常简单，只是修改了下应用名称，提交后

访问：<http://localhost:8201/config>

发现配置并没有修改，一直要重启【microcloud-config-client】后才会发现配置已经修改成功，其实这对大多数应用没有什么问题，如果你定时要关注这个小问题也是有办法处理的，在springcloud里面可以借助消息总线SpringCloudBus解决这问题。

ESB（企业服务总线）在一开始讲springcloud的时候就讲过，在springcloud中就是使用springcloudbus解决这问题的

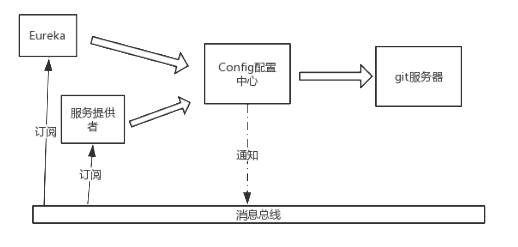
基于总线的设计其实是借鉴了硬件的思想，编写的一个程序都需要用到CPU，内存，硬盘等资源，每一个硬件资源的接口都不同，所以需要借助驱动程序，这个驱动程序就类似于设计模式里面的适配器模式。通过这种设计，举例来说，在程序运行中可能需要用到CPU的资源，但由于不同的CPU通过驱动程序已经对硬件完全兼容了。

虽然这样一来CPU，内存，硬盘对系统是可以识别了，但这些硬件资源一般都不是由程序自己去访问，而是由系统的内核来处理，由操作系统来统一调度各种硬件资源。

这其实就是消息总线的一种体现，操作系统就是这个总线，在架构中，消息中线是一个中枢系统。

springcloudbus是基于SpringCloudStream的，SpringCloudStream的作用其实也是一种适配器模式的体现，消息中间件由很多，比如activemq，rabbitmq ，kafka，不同的消息中间件都会由使用上的差异，而SpringCloudStream就是为了屏蔽各种消息中间件的差异而存在的，具体的SpringCloudStream我们后面会单独拿来说

再来看个图



与之前的架构不一样的地方在于增加了消息总线，消息总线连接了config配置中心和各个配置中心的消费方，当配置提交到github的时候，可以借助/bus/refresh刷新，config配置中心再将变更的消息通知到其他的客户端

【github】 修改配置microcloud-config-client.yml

spring:

profiles:

active:

- dev

---

server:

port: 8201

spring:

profiles: dev

application:

name: microconfig-test-client2

eureka:

client:

serviceUrl:

defaultZone: http://admin:enjoy@localhost:7301/eureka

register-with-eureka: false

info:

app.name: microcloud-config-client-dev

company.name: enjoy

---

server:

port: 8102

spring:

profiles: beta

application:

name: microconfig-test-client2

eureka:

client:

serviceUrl:

defaultZone: http://admin:enjoy@localhost:7301/eureka

register-with-eureka: false

info:

app.name: microcloud-config-client-dev

company.name: enjoy

启动rabbitmq



登陆查看 <http://localhost:15672>

### 准备bus配置中心

新建立一个模块【microcloud-config-bus】，这模块是配置中心的升级版，作用也是配置中心。

【microcloud-config-bus】 修改pom文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-config-bus</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</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-netflix-eureka-client</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-config-bus】 修改application.yml文件，配置上git目录，一样连接上eureka,要和消息中间件通讯，所以RabbitMQ的连接信息也配置上

server:

port: 7201

spring:

cloud:

config:

server:

git:

uri: https://github.com/enjoyeud/microconfig.git

bus:

trace:

enabled: true

rabbitmq:

host: localhost

port: 5672 # RabbitMQ的监听端口

username: enjoy # 用户名

password: 5428325 # 密码

application:

name: microcloud-config-bus

eureka:

client:

serviceUrl:

defaultZone: http://admin:enjoy@localhost:7301/eureka

instance:

prefer-ip-address: true # 在地址栏上使用IP地址进行显示

instance-id: microcloud-config-bus

management:

endpoints:

web:

exposure:

include: "\*"

新增启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableConfigServer

@EnableEurekaClient

public class ConfigBusApp {

public static void main(String[] args) {

SpringApplication.run(ConfigBusApp.class,args);

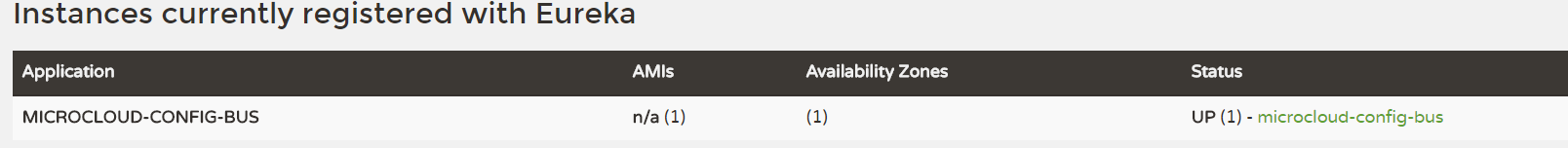
}

}

先启动eureka，再启动ConfigBusApp 之后访问：

[http://localhost:7301/](http://localhost:7301/，发现新的注册中心已经注册上去了。)

[发现新的注册中心已经注册上去了。](http://localhost:7301/，发现新的注册中心已经注册上去了。)



### 准备新的客户端

新建立microcloud-config-bus-client模块，这模块是注册中心的客户端，从注册中心获取数据，职责和【microcloud-config-client】一样，可用基于他拷贝修改，只是增加bus相关的功能。

【microcloud-config-bus-client】 修改pom文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-config-bus-client</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</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-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-config-bus-client】bootstrap.yml 增加rabbitmq相关信息，另外的配置和前面一样，需要从eureka找到注册中心，也需要找具体配置文件信息

spring:

cloud:

config:

name: microcloud-config-client # 定义要读取的资源文件的名称

profile: dev # 定义profile的 名称

label: master # 定义配置文件所在的分支

#uri: http://localhost:7101 # SpringCloudConfig的服务地址

username: admin # 连接的用户名

password: enjoy # 连接的密码

discovery:

enabled: true

service-id: MICROCLOUD-CONFIG-BUS

rabbitmq:

host: localhost

port: 5672 # RabbitMQ的监听端口

username: enjoy # 用户名

password: 5428325 # 密码

eureka:

client:

serviceUrl:

defaultZone: http://admin:enjoy@localhost:7301/eureka

register-with-eureka: false

修改application.yml文件

spring:

application:

name: microcloud-config-client # 编写应用的名称

【microcloud-config-bus-client】 建立一个配置文件的映射类，这类是为了演示使用，里面的属性和github的属性一一对应，同时增加@RefreshScope，代表这个类是可用基于rabbitmq自动刷新的。

package cn.enjoy.config;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.cloud.context.config.annotation.RefreshScope;

import org.springframework.stereotype.Component;

@Component

@RefreshScope

public class InfoConfig {

@Value("${info.app.name}")

private String appName ;

@Value("${info.company.name}")

private String companyName ;

public String getAppName() {

return appName;

}

public void setAppName(String appName) {

this.appName = appName;

}

public String getCompanyName() {

return companyName;

}

public void setCompanyName(String companyName) {

this.companyName = companyName;

}

@Override

public String toString() {

return "InfoConfig{" +

"appName='" + appName + '\'' +

", companyName='" + companyName + '\'' +

'}';

}

}

【microcloud-config-bus-client】修改ConfigClientController

package cn.enjoy.controller;

import cn.enjoy.config.InfoConfig;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.cloud.context.config.annotation.RefreshScope;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;

@RestController

public class ConfigClientController {

@Value("${spring.application.name}")

private String applicationName;

@Value("${eureka.client.serviceUrl.defaultZone}")

private String eurekaServers;

@Resource

private InfoConfig infoConfig;

@RequestMapping("/config")

public String getConfig() {

return "ApplicationName = " + this.applicationName + "、EurekaServers = "

+ this.eurekaServers+"、infos = " +infoConfig.toString();

}

}

新增启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

public class ConfigClientBusApp {

public static void main(String[] args) {

SpringApplication.run(ConfigClientBusApp.class,args);

}

}

启动后访问

<http://localhost:8201/config，这时候已经能获得配置中心的数据>

### 测试自动刷新

其实这里的自动刷新只能说是半自动的。

【github】microcloud-config-client.yml，随便修改里面的内容，提交

spring:

profiles:

active:

- dev

---

server:

port: 8201

spring:

profiles: dev

application:

name: microconfig-test-client2

eureka:

client:

serviceUrl:

defaultZone: http://admin:enjoy@localhost:7301/eureka

register-with-eureka: false

info:

app.name: microcloud-config-client-dev

company.name: enjoy

---

server:

port: 8102

spring:

profiles: beta

application:

name: microconfig-test-client2

eureka:

client:

serviceUrl:

defaultZone: http://admin:enjoy@localhost:7301/eureka

register-with-eureka: false

info:

app.name: microcloud-config-client-beta

company.name: enjoybeta

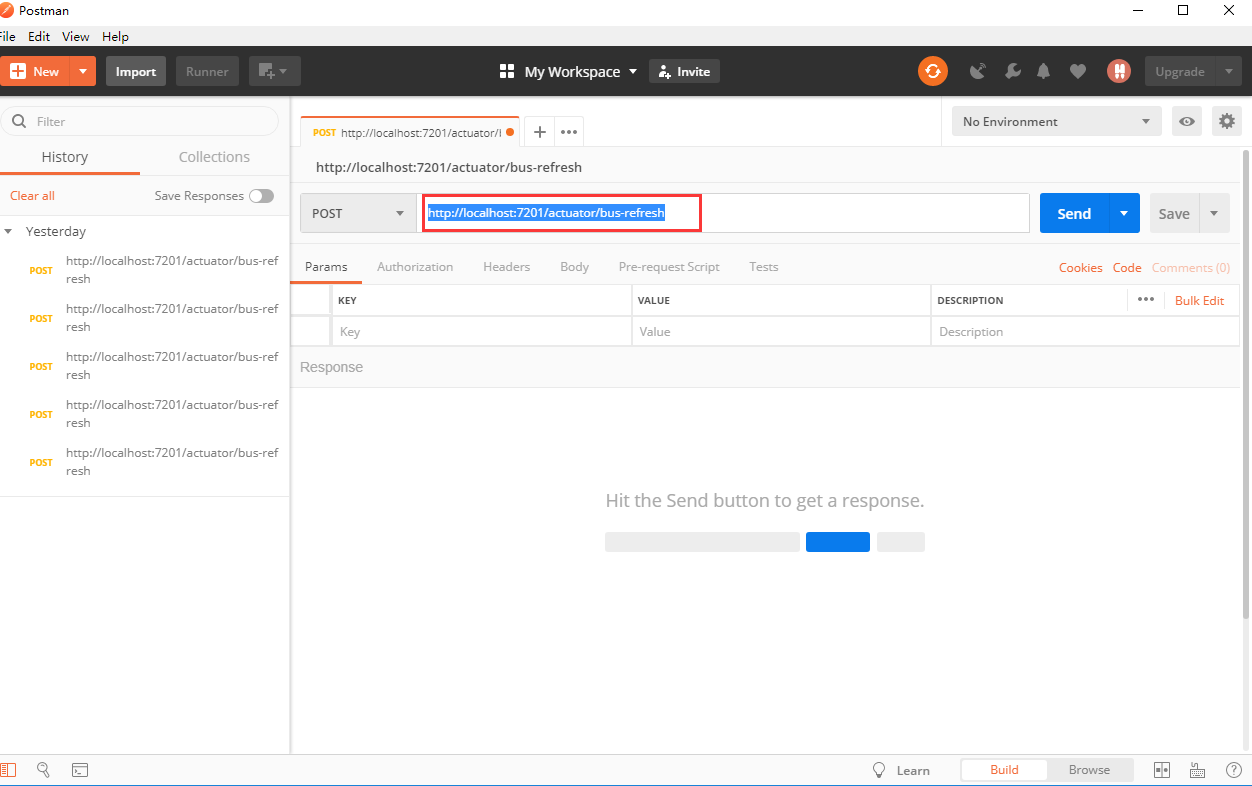
【microcloud-config-bus-client】 刷新客户端

[http://localhost:8201/config](http://localhost:8201/config，这个时候信息并没自动刷新，数据还是以前的，这是以为对应消息中间件来说，还需要给他发个消息，代表数据已经更新了。)

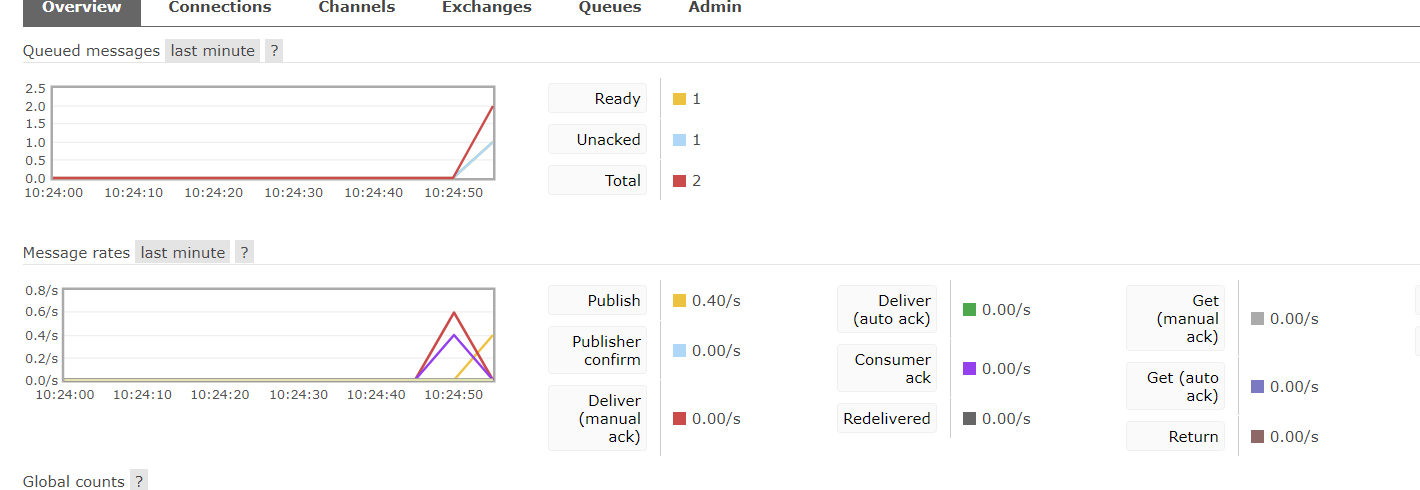
[这个时候信息并没自动刷新，数据还是以前的，这是以为对应消息中间件来说，还需要给他发个消息，代表数据已经更新了。](http://localhost:8201/config，这个时候信息并没自动刷新，数据还是以前的，这是以为对应消息中间件来说，还需要给他发个消息，代表数据已经更新了。)

【microcloud-config-bus】使用postman发生一条post刷新的指令

http://localhost:7201/actuator/bus-refresh



访问http://localhost:15672，发现消息队列里面已经有消息传递了。



【microcloud-config-bus-client】 刷新客户端

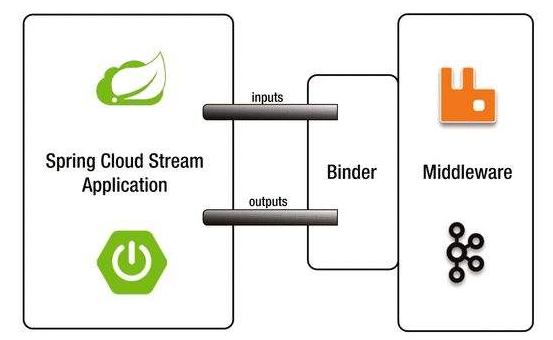
<http://localhost:8201/config> 发现数据已经跟新

# SpringCloudStream 消息驱动

SpringCloudStream看名字就知道他和消息队列相关，但它又不是消息队列，准确来说它类似于硬件里面的驱动程序，也就是前面说的适配器模式的体现

在系统开发里面难免用到消息队列，但各个的消息队列又有所区别，SpringCloudStream的作用就是屏蔽各种消息队列的区别，对消息队列的API进行进一步的抽象，使得在springcloud里面能更加方便的集成各种消息系统

首先来看SpringCloudStream的组成



这个图很容易看懂，没有什么复杂的概念

不管是生产者还是消费者，并不会直接和消息中间件打交道，在springcloudstream中抽象了已成binder（绑定层），有了这一层，使用者并不关心具体的消息中间件配置了，由访问层真正的和消息队列进行通信。

## 创建消息生产者

【microcloud-stream-provider】创建一个新的模块，这模块负责生产一个消息

【microcloud-stream-provider】 pom文件如下，映入springcloudstream的相关组件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-stream-provider</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

</project>

【microcloud-stream-provider】 修改application.yml文件

server:

port: 8401

spring:

cloud:

stream:

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

output: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

application:

name: microcloud-stream-provider

【microcloud-stream-provider】 定义一个消息发送接口

package cn.enjoy.service;

import cn.enjoy.vo.Product;

public interface IMessageProvider {

void send(Product product);

}

【microcloud-stream-provider】 定义接口的实现类

package cn.enjoy.service.impl;

import cn.enjoy.service.IMessageProvider;

import cn.enjoy.vo.Product;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.messaging.Source;

import org.springframework.messaging.MessageChannel;

import org.springframework.messaging.support.MessageBuilder;

import javax.annotation.Resource;

@EnableBinding(Source.class)

public class MessageProviderImpl implements IMessageProvider{

@Resource

private MessageChannel output; // 消息的发送管道

@Override

public void send(Product product) {

output.send(MessageBuilder.withPayload(product).build());

}

}

【microcloud-stream-provider】 定义启动类主程序

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class StreamProviderApp {

public static void main(String[] args) {

SpringApplication.run(StreamProviderApp.class,args);

}

}

【microcloud-stream-provider】 编写测试类

package cn.enjoy.test;

import cn.enjoy.StreamProviderApp;

import cn.enjoy.service.IMessageProvider;

import cn.enjoy.vo.Product;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import javax.annotation.Resource;

@SpringBootTest(classes = StreamProviderApp.class)

@RunWith(SpringRunner.class)

public class TestMessageProvider {

@Resource

private IMessageProvider messageProvider;

@Test

public void testSend() {

Product product = new Product();

product.setProductId(1L);

product.setProductName("messageName");

product.setProductDesc("desc");

messageProvider.send(product);

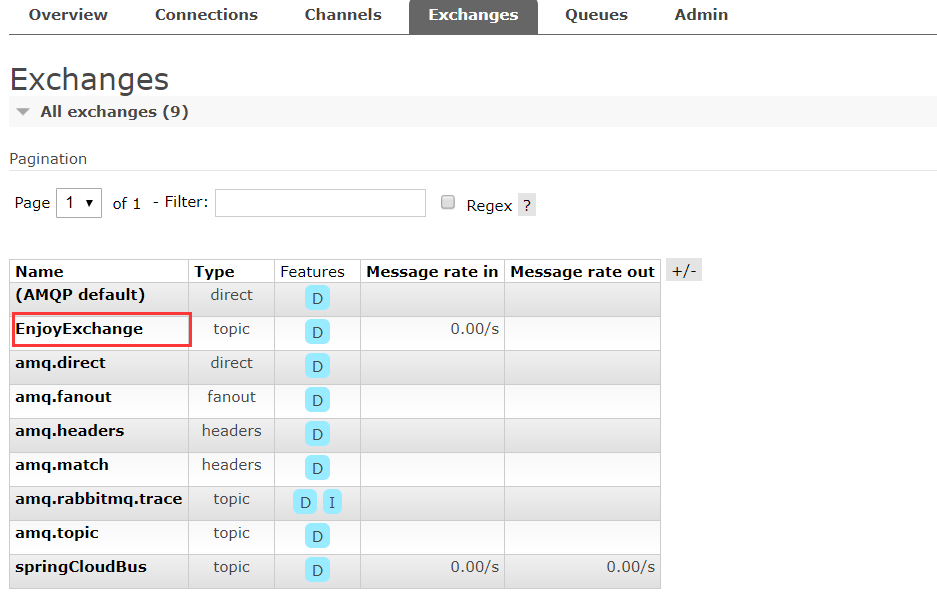
}

}

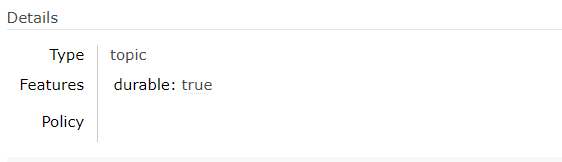
运行测试方法

RabbitMq:http[://localhost:15672/](http://localhost:15672/)

已经产生了个



对于RabbitMq来说，在这默认的Exchange的类型是Topic



## 创建消息消费者

【microcloud-stream-consumer】 新建模块

【microcloud-stream-consumer】修改pom文件如下

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-stream-consumer</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

<dependency>

<groupId>enjoy</groupId>

<artifactId>microcloud-api</artifactId>

</dependency>

</dependencies>

</project>

【microcloud-stream-consumer】 修改application.yml配置文件

server:

port: 8402

spring:

cloud:

stream:

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

input: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

application:

name: microcloud-stream-consumer

【microcloud-stream-consumer】 定义一个消息的监听

package cn.enjoy.listener;

import cn.enjoy.vo.Product;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.cloud.stream.messaging.Sink;

import org.springframework.messaging.Message;

import org.springframework.stereotype.Component;

@Component

@EnableBinding(Sink.class)

public class MessageListener {

@StreamListener(Sink.INPUT)

public void input(Message<Product> message) {

System.err.println("【\*\*\* 消息接收 \*\*\*】" + message.getPayload());

}

}

【microcloud-stream-consumer】 创建启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class StreamConsumerApp {

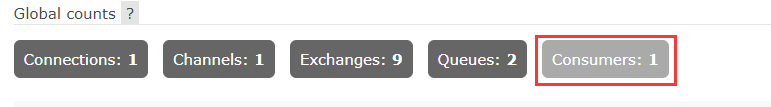
public static void main(String[] args) {

SpringApplication.run(StreamConsumerApp.class,args);

}

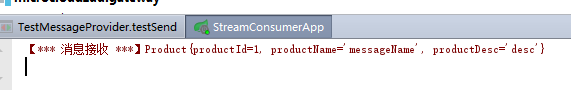
}

启动后，发现rabbitmq里面已经有了一个消费者

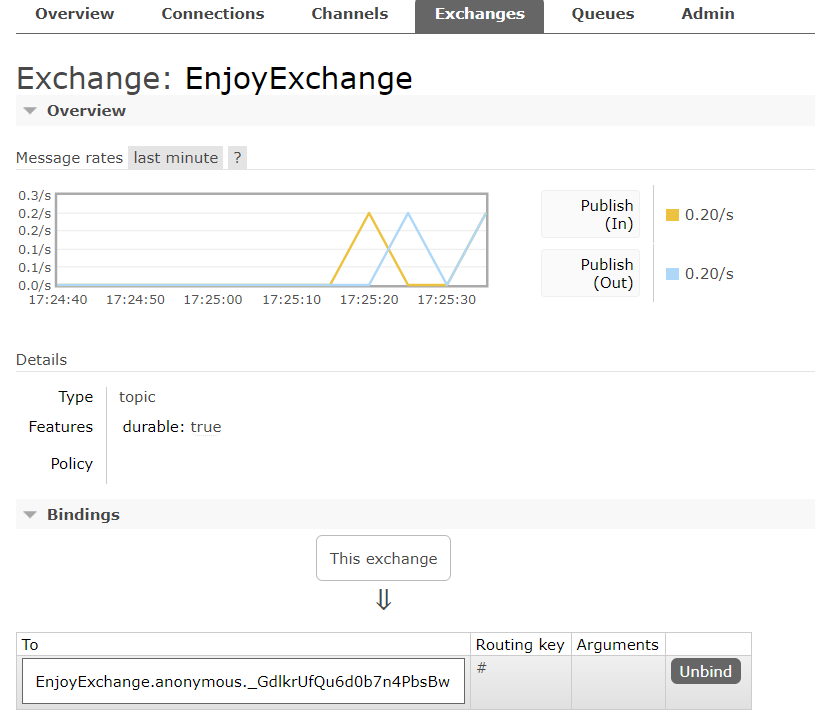


【microcloud-stream-provider】 运行测试类

【microcloud-stream-consumer】 已经收到消息



【rabbitmq】在消息队列中，也检测到了这匿名队列，可以发现Exchange默认的类型就是TOPIC，RoutingKey我们没有自定，默认的就是#，会给所有的消费者发消息



## 自定义消息通道

[前面](mailto:前面在创建服务提供者的时候用到了@EnableBinding(Source.class))

[在创建消息生产者的时候用到了@EnableBinding(Source.class)](mailto:前面在创建服务提供者的时候用到了@EnableBinding(Source.class))

[在创建消息消费者的时候用到了@EnableBinding(Sink.class)](mailto:在创建消息消费者的时候用到了@EnableBinding(Sink.class))

可以查看下Source.class与Sink.class的源码

package org.springframework.cloud.stream.messaging;

import org.springframework.cloud.stream.annotation.Output;

import org.springframework.messaging.MessageChannel;

public interface Source {

String OUTPUT = "output";

@Output("output")

MessageChannel output();

}

package org.springframework.cloud.stream.messaging;

import org.springframework.cloud.stream.annotation.Input;

import org.springframework.messaging.SubscribableChannel;

public interface Sink {

String INPUT = "input";

@Input("input")

SubscribableChannel input();

}

其实这个input,output就是对应配置文件里面的内容。

其实用户也是可以自定义管道信息的

由于在【microcloud-stream-provider】 【microcloud-stream-consumer】 这两个模块中都需要用到自定义管道信息

【micocloud-api】修改pom文件，增加stream的支持

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

</dependencies>

【micocloud-api】 增加管道接口

package cn.enjoy.channel;

import org.springframework.cloud.stream.annotation.Input;

import org.springframework.cloud.stream.annotation.Output;

import org.springframework.messaging.MessageChannel;

import org.springframework.messaging.SubscribableChannel;

public interface DefaultProcess {

public static final String OUTPUT = "enjoy\_output"; // 输出通道名称

public static final String INPUT = "enjoy\_input"; // 输入通道名称

@Input(DefaultProcess.INPUT)

public SubscribableChannel input();

@Output(DefaultProcess.OUTPUT)

public MessageChannel output();

}

【microcloud-stream-provider】 修改application.yml配置文件

server:

port: 8401

spring:

cloud:

stream:

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

enjoy\_output: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

application:

name: microcloud-stream-provider

【microcloud-stream-consumer】 修改application.yml文件

server:

port: 8402

spring:

cloud:

stream:

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

enjoy\_input: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

application:

name: microcloud-stream-consumer

【microcloud-stream-provider】 修改MessageProviderImpl发送实现类

package cn.enjoy.service.impl;

import cn.enjoy.channel.DefaultProcess;

import cn.enjoy.service.IMessageProvider;

import cn.enjoy.vo.Product;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.messaging.Source;

import org.springframework.messaging.MessageChannel;

import org.springframework.messaging.support.MessageBuilder;

import javax.annotation.Resource;

@EnableBinding(DefaultProcess.class)

public class MessageProviderImpl implements IMessageProvider{

@Resource

@Qualifier("enjoy\_output")

private MessageChannel output; // 消息的发送管道

@Override

public void send(Product product) {

output.send(MessageBuilder.withPayload(product).build());

}

}

【microcloud-stream-consumer】 修改MessageListener

package cn.enjoy.listener;

import cn.enjoy.channel.DefaultProcess;

import cn.enjoy.vo.Product;

import org.springframework.cloud.stream.annotation.EnableBinding;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.cloud.stream.messaging.Sink;

import org.springframework.messaging.Message;

import org.springframework.stereotype.Component;

@Component

@EnableBinding(DefaultProcess.class)

public class MessageListener {

@StreamListener(DefaultProcess.INPUT)

public void input(Message<Product> message) {

System.err.println("【\*\*\* 消息接收 \*\*\*】" + message.getPayload());

}

}

修改之后就可以用自定义管道了。

## 分组（队列）

Group（分组）其实就是对应rabbitmq里面得队列，在前面的案例中，我们并没有指定group，产生的就是一个匿名队列。

如果启动了多个【microcloud-stream-consumer】接收者，但并没有指定group，，那么将会产生多个匿名的消息队列，导致多个接收者都会收到同一个消息，也就是说一个消息被重复消费了，这在某些业务场景来说是并不运行的。

这个时候就需要用到group分组了，对于不想重复消费个某消息的各个消费者必须属于同一个组。

【microcloud-stream-consumer】 修改application.yml

server:

port: 8403

spring:

cloud:

stream:

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

enjoy\_input: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

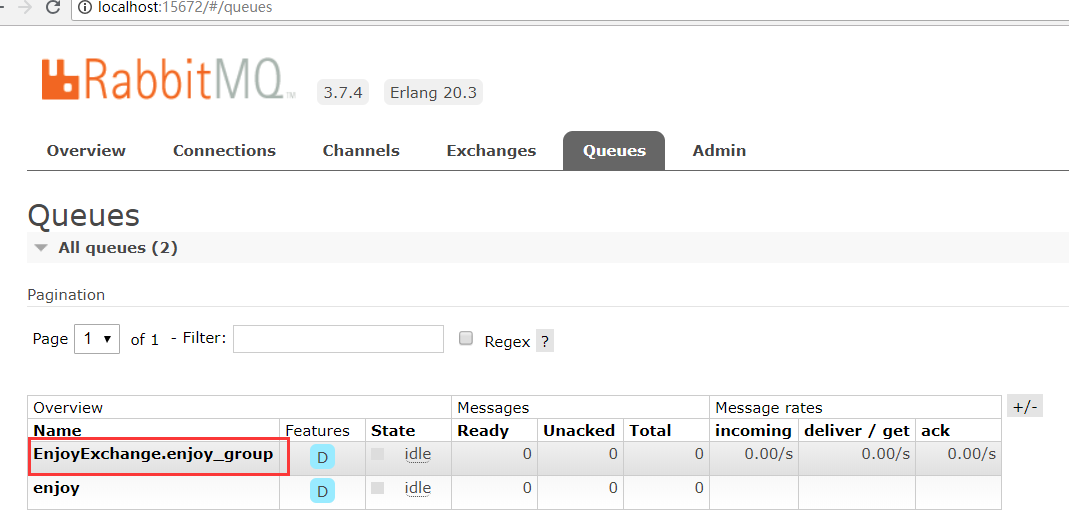
binder: defaultRabbit # 设置要绑定的消息服务的具体设置

group: enjoy\_group

application:

name: microcloud-stream-consumer

启动了多个【microcloud-stream-consumer】接收者，由于指定了group,在rabbitmq也就只会产生一个队列



这个时候服务生产者【microcloud-stream-provider】发送的任何消息都只会被同一个group的某一个消费者处理了。

## 设置 RoutingKey

RoutingKey其实是RabbitMq的概念，在RabbitMq里面其实有好几种Exchange，在SpringCloudStream里面默认就是使用的最通用的Topic

如果没有配置RoutingKey，它使用的RoutingKey其实就是#，既类似于fanout的广播类型

其实也可以使用RoutingKey来实现类似于direct类型，既然直连

【microcloud-stream-consumer】修改application.yml配置文件

server:

port: 8402

spring:

cloud:

stream:

rabbit:

bindings:

enjoy\_input:

consumer:

bindingRoutingKey: enjoyKey # 设置一个RoutingKey信息

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

enjoy\_input: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

group: enjoy\_group

application:

name: microcloud-stream-consumer

【microcloud-stream-provider】定义 RoutingKey 的表达式配置：

server:

port: 8401

spring:

cloud:

stream:

rabbit:

bindings:

enjoy\_output:

producer:

routingKeyExpression: '''enjoyKey'''

binders: # 在此处配置要绑定的rabbitmq的服务信息；

defaultRabbit: # 表示定义的名称，用于于binding整合

type: rabbit # 消息组件类型

environment: # 设置rabbitmq的相关的环境配置

spring:

rabbitmq:

addresses: localhost

port: 5672

username: enjoy

password: 5428325

virtual-host: /

bindings: # 服务的整合处理

enjoy\_output: # 这个名字是一个通道的名称，在分析具体源代码的时候会进行说明

destination: EnjoyExchange # 表示要使用的Exchange名称定义

content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设置“text/plain”

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

application:

name: microcloud-stream-provider

# SpringCloudSleuth 链路跟踪

在微服务的架构下，系统由大量服务组成，每个服务可能是由不同的团队开发，开发使用不同的语言，部署在几千台服务器上，并且横跨多个不同的数据中心，一次请求绝大多数情况会涉及多个服务，在系统发生故障的时候，想要快速定位和解决问题，就需要跟踪服务请求序列

SpringCloudSleuth使用的核心组件是Twitter推出的Zipkin监控组件，zipkin就是一个分布式的跟踪系统，用户收集服务的数据

## 基本使用

### 跟踪服务

【microcloud-sleuth】 新建立模块，这个模块用户跟踪用户的请求，把请求的链路进行展示

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-sleuth</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-server</artifactId>

<version>2.9.3</version>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-autoconfigure-ui</artifactId>

<version>2.9.3</version>

</dependency>

</dependencies>

</project>

【microcloud-sleuth】修改application.yml文件

server:

port: 8601

spring:

application:

name: microcloud-zipkin-server

management:

metrics:

web:

server:

auto-time-requests: false

【microcloud-sleuth】新建启动类

package cn.enjoy;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import zipkin.server.internal.EnableZipkinServer;

@SpringBootApplication

@EnableZipkinServer

public class SleuthApp {

public static void main(String[] args) {

SpringApplication.run(SleuthApp.class,args);

}

}

### 客户端配置

为了演示链路追踪，需要启动一些列服务



上图为显示的最终结果

1.【microcloud-eureka】，启动这个服务的目的是让product,users两服务注册到其中，后面需要zuul的调用

2.修改一下服务的pom文件，增加sleuth的支持

【microcloud-provider-user-hystrix】

【microcloud-provider-product-hystrix】

【microcloud-zuul-gateway】

【microcloud-consumer-hystrix】

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

有可能下载不到，如果抽风，用这个

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

<version>2.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

<version>2.0.0.RELEASE</version>

</dependency>

3.修改一下服务的application.yml文件，增加zipkin的配置

【microcloud-provider-user-hystrix】

【microcloud-provider-product-hystrix】

【microcloud-zuul-gateway】

【microcloud-consumer-hystrix】

spring:

zipkin:

base-url: http://localhost:8601 #所有的数据提交到此服务之中

sender:

type: web #提交的类型是web 服务

sleuth:

sampler:

probability: 1.0 # 定义抽样比率，默认为0.1

1. 依次启动

【microcloud-eureka】

【microcloud-provider-user-hystrix】

【microcloud-provider-product-hystrix】

【microcloud-zuul-gateway】

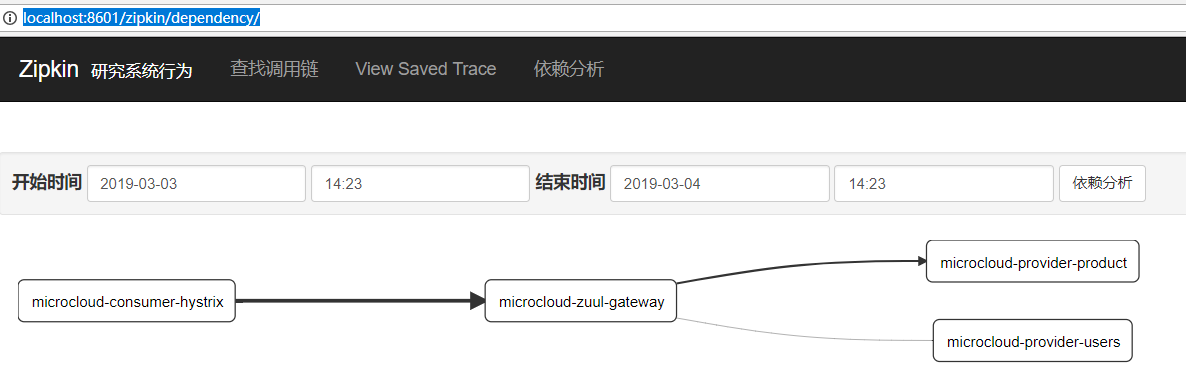
【microcloud-consumer-hystrix】

访问

<http://localhost/consumer/product/getProductAndUser?id=1>

访问跟踪服务：

http://localhost:8601/zipkin/dependency/



## 数据持久化

现在以及成功实现了一个 SpringCloudSleuth 的基本操作，但会发现，如果重新启动【microcloud-sleuth】服务，所有的链路跟踪数据都会丢失，那么这些数据应该存储到数据库里面的。

但又有另外一个问题，如果请并发量特别大，对于mysql来说可能会承受不了这么大的并发，为了解决这个问题，可以使用消息队列缓冲处理，最后才从mq中把数据存到mysql中。

1.【databases】 先在数据库里面创建zipkin的表

<https://github.com/openzipkin/zipkin/blob/master/zipkin-storage/mysql-v1/src/main/resources/mysql.sql>

DROP DATABASE IF EXISTS zipkin ;

CREATE DATABASE zipkin CHARACTER SET UTF8 ;

USE zipkin ;

CREATE TABLE IF NOT EXISTS zipkin\_spans (

`trace\_id\_high` BIGINT NOT NULL DEFAULT 0 COMMENT 'If non zero, this means the trace uses 128 bit traceIds instead of 64 bit',

`trace\_id` BIGINT NOT NULL,

`id` BIGINT NOT NULL,

`name` VARCHAR(255) NOT NULL,

`parent\_id` BIGINT,

`debug` BIT(1),

`start\_ts` BIGINT COMMENT 'Span.timestamp(): epoch micros used for endTs query and to implement TTL',

`duration` BIGINT COMMENT 'Span.duration(): micros used for minDuration and maxDuration query'

) ENGINE=InnoDB ROW\_FORMAT=COMPRESSED CHARACTER SET=utf8 COLLATE utf8\_general\_ci;

ALTER TABLE zipkin\_spans ADD UNIQUE KEY(`trace\_id\_high`, `trace\_id`, `id`) COMMENT 'ignore insert on duplicate';

ALTER TABLE zipkin\_spans ADD INDEX(`trace\_id\_high`, `trace\_id`, `id`) COMMENT 'for joining with zipkin\_annotations';

ALTER TABLE zipkin\_spans ADD INDEX(`trace\_id\_high`, `trace\_id`) COMMENT 'for getTracesByIds';

ALTER TABLE zipkin\_spans ADD INDEX(`name`) COMMENT 'for getTraces and getSpanNames';

ALTER TABLE zipkin\_spans ADD INDEX(`start\_ts`) COMMENT 'for getTraces ordering and range';

CREATE TABLE IF NOT EXISTS zipkin\_annotations (

`trace\_id\_high` BIGINT NOT NULL DEFAULT 0 COMMENT 'If non zero, this means the trace uses 128 bit traceIds instead of 64 bit',

`trace\_id` BIGINT NOT NULL COMMENT 'coincides with zipkin\_spans.trace\_id',

`span\_id` BIGINT NOT NULL COMMENT 'coincides with zipkin\_spans.id',

`a\_key` VARCHAR(255) NOT NULL COMMENT 'BinaryAnnotation.key or Annotation.value if type == -1',

`a\_value` BLOB COMMENT 'BinaryAnnotation.value(), which must be smaller than 64KB',

`a\_type` INT NOT NULL COMMENT 'BinaryAnnotation.type() or -1 if Annotation',

`a\_timestamp` BIGINT COMMENT 'Used to implement TTL; Annotation.timestamp or zipkin\_spans.timestamp',

`endpoint\_ipv4` INT COMMENT 'Null when Binary/Annotation.endpoint is null',

`endpoint\_ipv6` BINARY(16) COMMENT 'Null when Binary/Annotation.endpoint is null, or no IPv6 address',

`endpoint\_port` SMALLINT COMMENT 'Null when Binary/Annotation.endpoint is null',

`endpoint\_service\_name` VARCHAR(255) COMMENT 'Null when Binary/Annotation.endpoint is null'

) ENGINE=InnoDB ROW\_FORMAT=COMPRESSED CHARACTER SET=utf8 COLLATE utf8\_general\_ci;

ALTER TABLE zipkin\_annotations ADD UNIQUE KEY(`trace\_id\_high`, `trace\_id`, `span\_id`, `a\_key`, `a\_timestamp`) COMMENT 'Ignore insert on duplicate';

ALTER TABLE zipkin\_annotations ADD INDEX(`trace\_id\_high`, `trace\_id`, `span\_id`) COMMENT 'for joining with zipkin\_spans';

ALTER TABLE zipkin\_annotations ADD INDEX(`trace\_id\_high`, `trace\_id`) COMMENT 'for getTraces/ByIds';

ALTER TABLE zipkin\_annotations ADD INDEX(`endpoint\_service\_name`) COMMENT 'for getTraces and getServiceNames';

ALTER TABLE zipkin\_annotations ADD INDEX(`a\_type`) COMMENT 'for getTraces and autocomplete values';

ALTER TABLE zipkin\_annotations ADD INDEX(`a\_key`) COMMENT 'for getTraces and autocomplete values';

ALTER TABLE zipkin\_annotations ADD INDEX(`trace\_id`, `span\_id`, `a\_key`) COMMENT 'for dependencies job';

CREATE TABLE IF NOT EXISTS zipkin\_dependencies (

`day` DATE NOT NULL,

`parent` VARCHAR(255) NOT NULL,

`child` VARCHAR(255) NOT NULL,

`call\_count` BIGINT,

`error\_count` BIGINT

) ENGINE=InnoDB ROW\_FORMAT=COMPRESSED CHARACTER SET=utf8 COLLATE utf8\_general\_ci;

ALTER TABLE zipkin\_dependencies ADD UNIQUE KEY(`day`, `parent`, `child`);

2.【microcloud-sleuth】修改pom文件，增加数据库，消息队列相关的依赖

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<parent>

<artifactId>springcloud</artifactId>

<groupId>enjoy</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>microcloud-sleuth</artifactId>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency>

<groupId>io.zipkin.zipkin2</groupId>

<artifactId>zipkin</artifactId>

<version>2.9.3</version>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-server</artifactId>

<version>2.9.3</version>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-autoconfigure-ui</artifactId>

<version>2.9.3</version>

</dependency>

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-autoconfigure-collector-rabbitmq</artifactId>

<version>2.9.3</version>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-autoconfigure-storage-mysql</artifactId>

<version>2.9.4</version>

</dependency>

</dependencies>

</project>

3.【microcloud-sleuth】修改application.yml文件

server:

port: 8601

spring:

application:

name: microcloud-zipkin-server

management:

metrics:

web:

server:

auto-time-requests: false

zipkin:

collector:

sample-rate: 1.0

http:

enabled: false

rabbitmq:

addresses: localhost:5672

password: 5428325

queue: zipkin

username: enjoy

virtual-host: /

storage:

type: mysql

mysql:

host: localhost

port: 3306

username: root

password: root1234%

db: zipkin

1. 确认客户方都有rabbitmq的依赖，修改pom文件，增加相关依赖

【microcloud-provider-user-hystrix】

【microcloud-provider-product-hystrix】

【microcloud-zuul-gateway】

【microcloud-consumer-hystrix】

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

5.修改一下服务的application.yml文件,修改zipkin的配置

【microcloud-provider-user-hystrix】

【microcloud-provider-product-hystrix】

【microcloud-zuul-gateway】

【microcloud-consumer-hystrix】

spring:

zipkin:

base-url: http://localhost:8601 #所有的数据提交到此服务之中

sender:

type: rabbit #提交的类型是web 服务

sleuth:

sampler:

probability: 1.0 # 定义抽样比率，默认为0.1

依次启动

【microcloud-eureka】

【microcloud-provider-user-hystrix】

【microcloud-provider-product-hystrix】

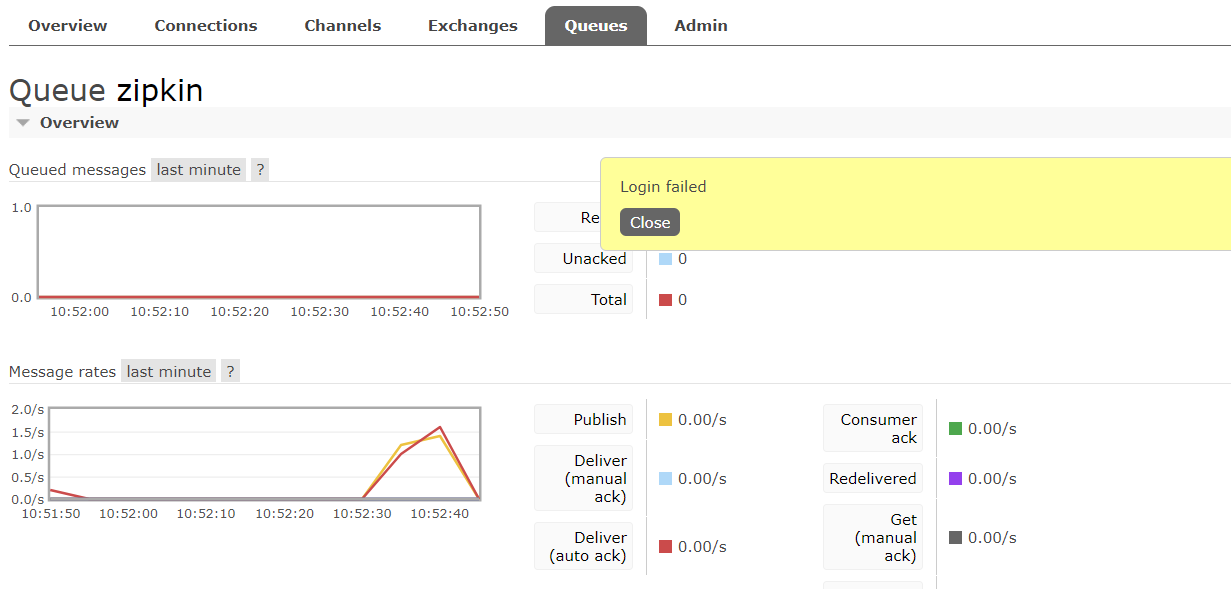
【microcloud-zuul-gateway】

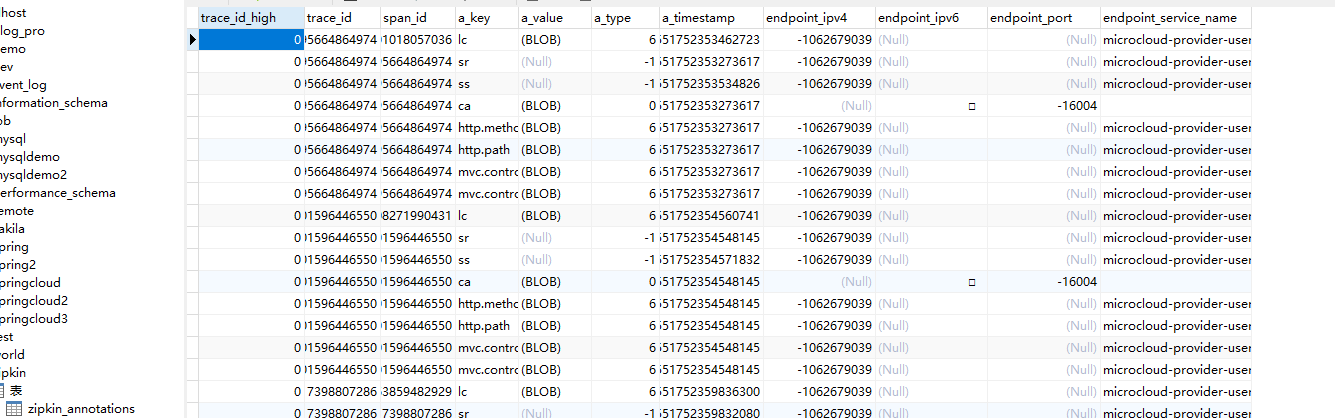
【microcloud-consumer-hystrix】

访问

<http://localhost/consumer/product/getProductAndUser?id=1>

http://localhost:15672/#/queues/%2F/zipkin，数据一节通过rabbitmq传输，并且数据已经到了数据库





具体配置参考：

https://github.com/openzipkin/zipkin/blob/master/zipkin-server/src/main/resources/zipkin-server-shared.yml