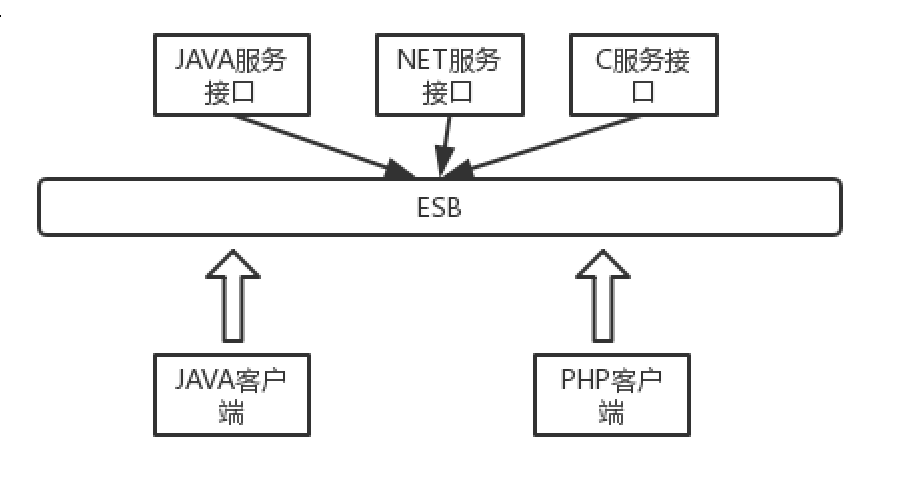
# RestFul简介

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:

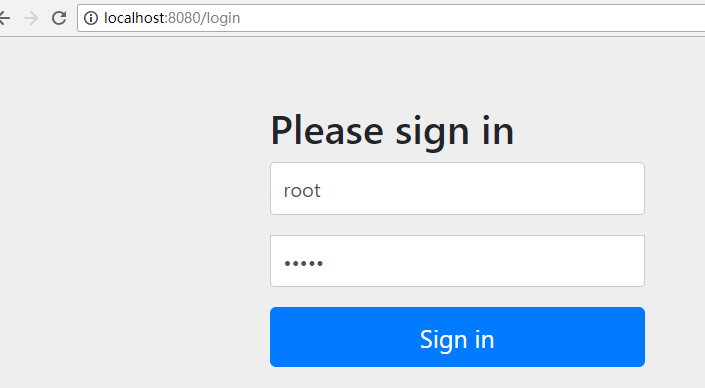
name: admin # 认证用户名

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>

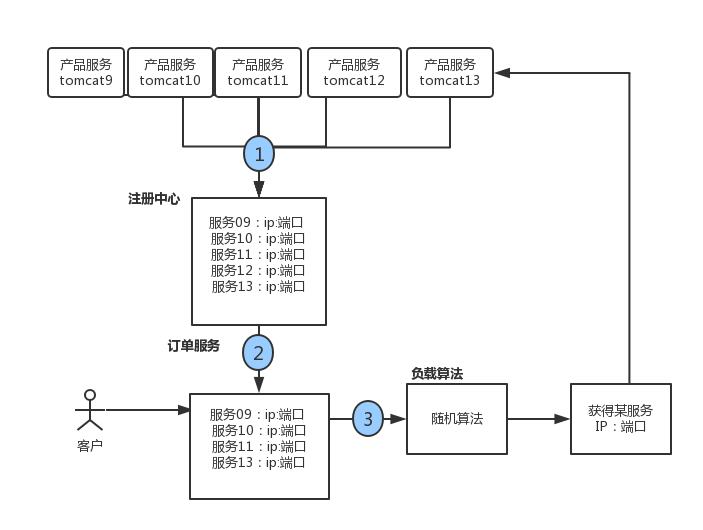
列表查询：<http://localhost/consumer/product/list>

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 {

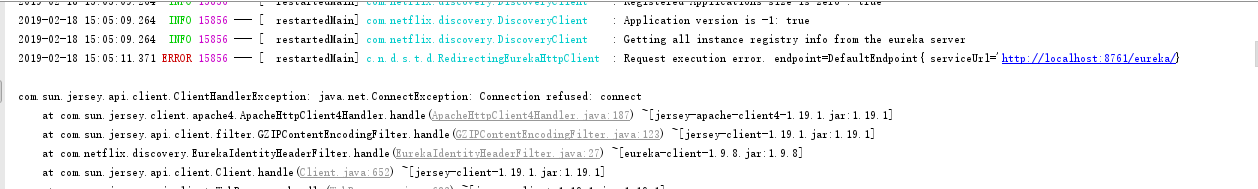
public static void main(String[] args) {

SpringApplication.run(EurekaApp.class,args);

}

}

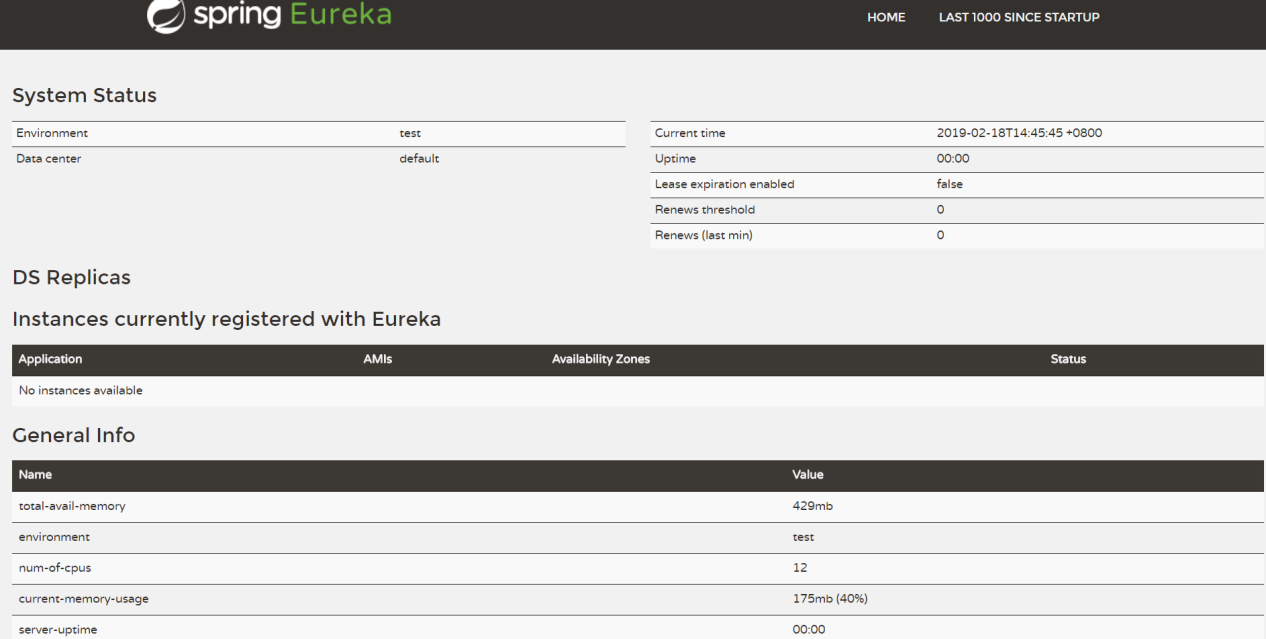
运行main方法



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{

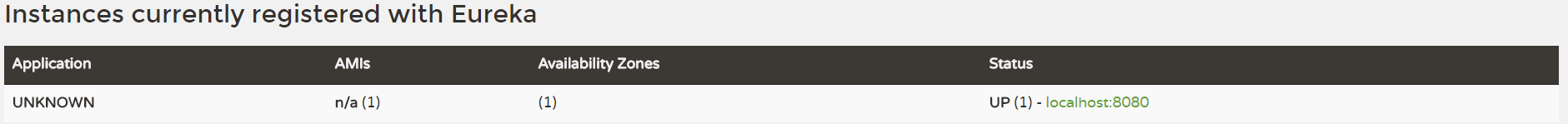
public static void main(String[] args) {

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

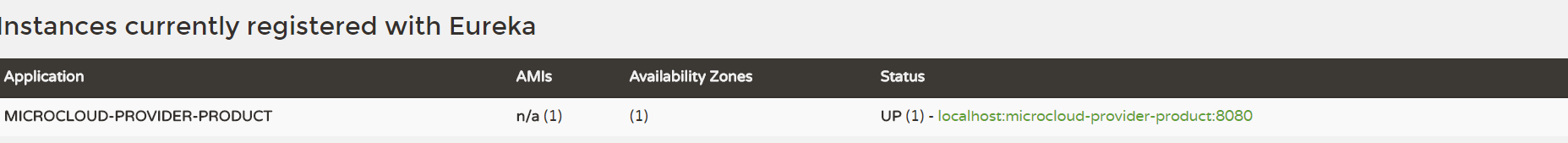
【microcloud-provider-product】修改application.yml配置文件，为这个微服务起一个名字

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:

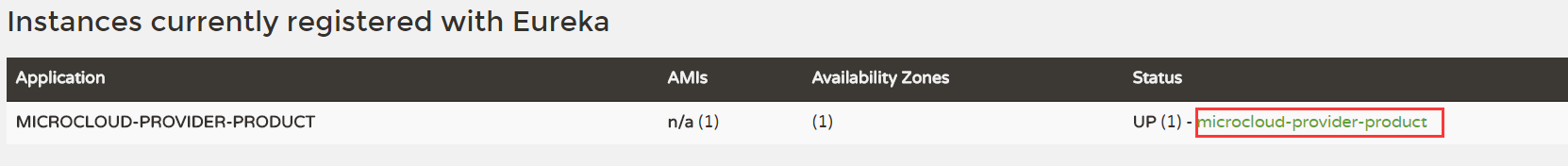
client: # 客户端进行Eureka注册的配置

service-url:

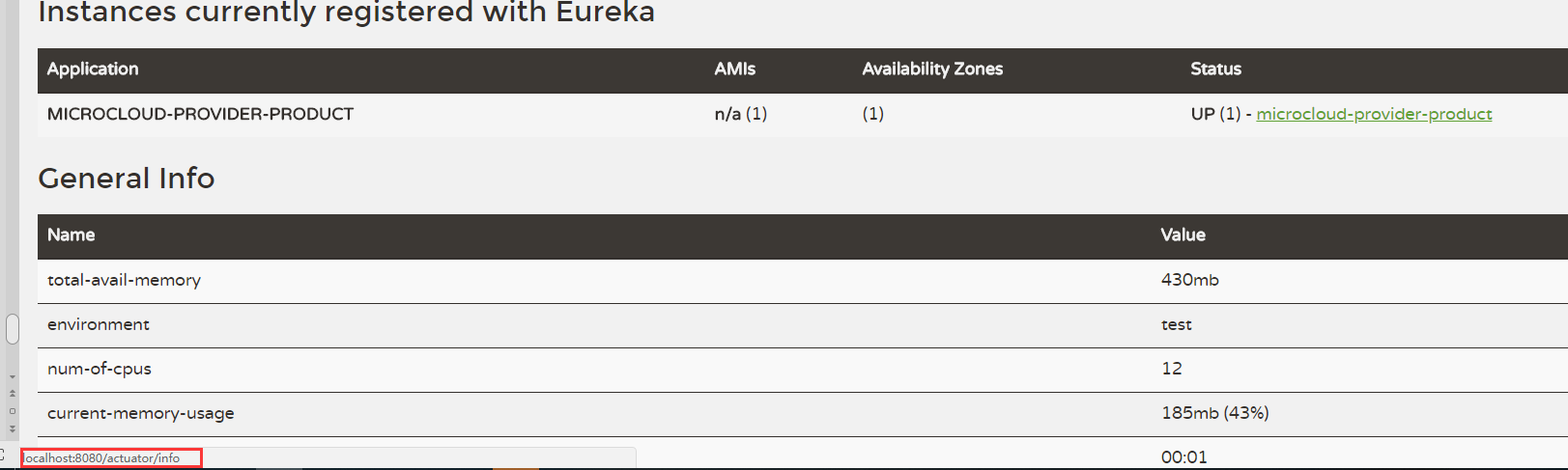
defaultZone: http://localhost:7001/eureka

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:

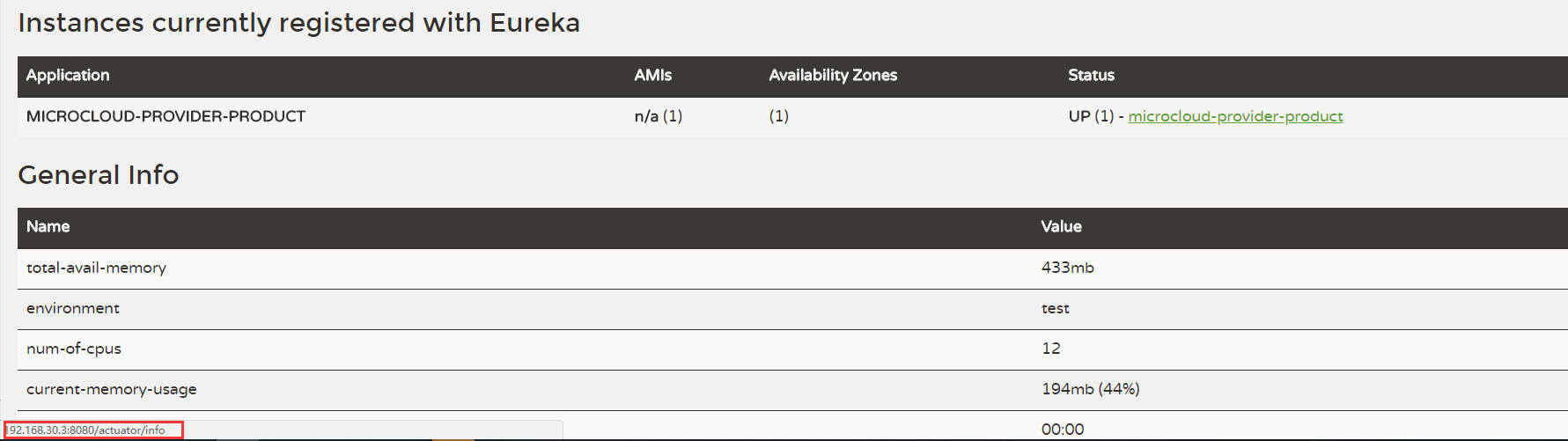
defaultZone: http://localhost:7001/eureka

instance:

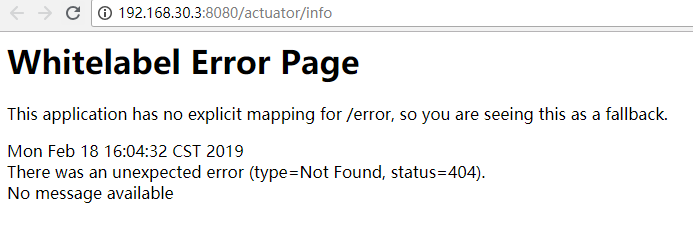
instance-id: microcloud-provider-product

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 -->

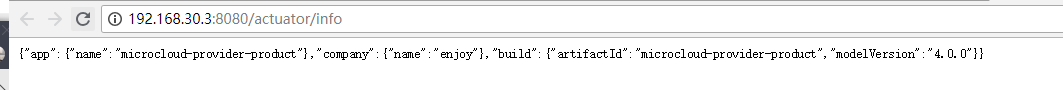
</configuration>

</plugin>

</plugins>

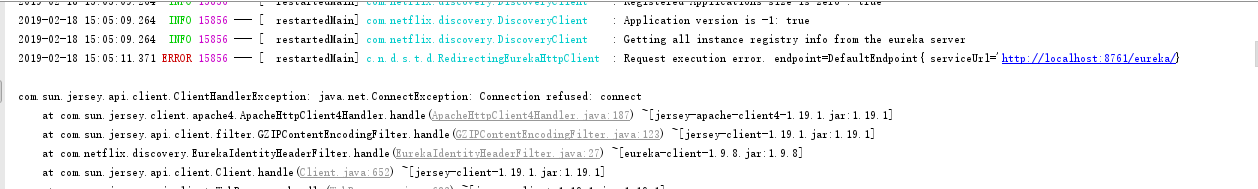
</build>

启动后：

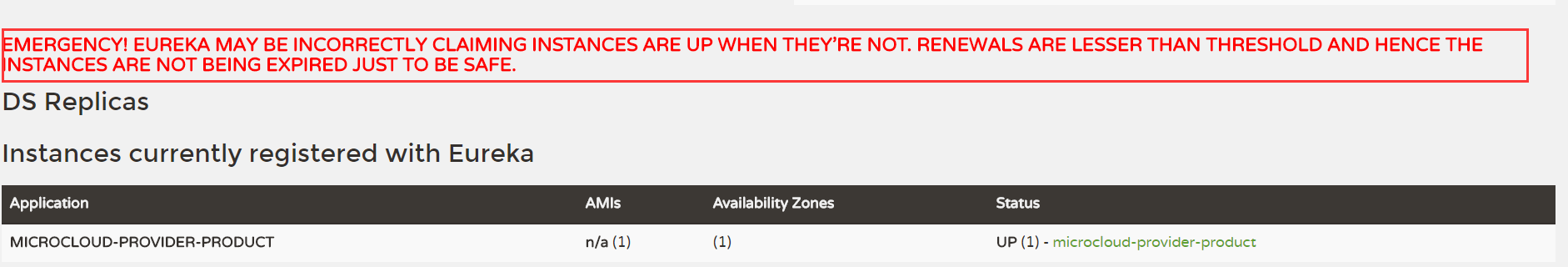


## 其他配置

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:

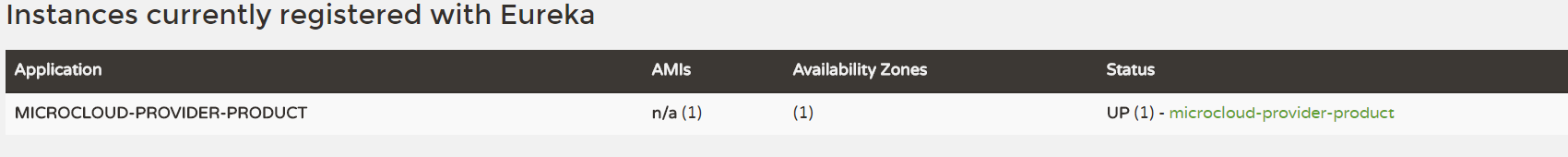
fetch-registry: false

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{

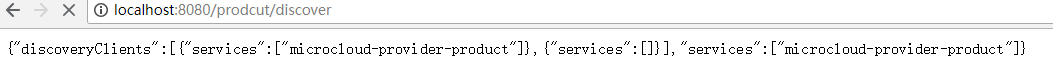
public static void main(String[] args) {

SpringApplication.run(ProductApp.class,args);

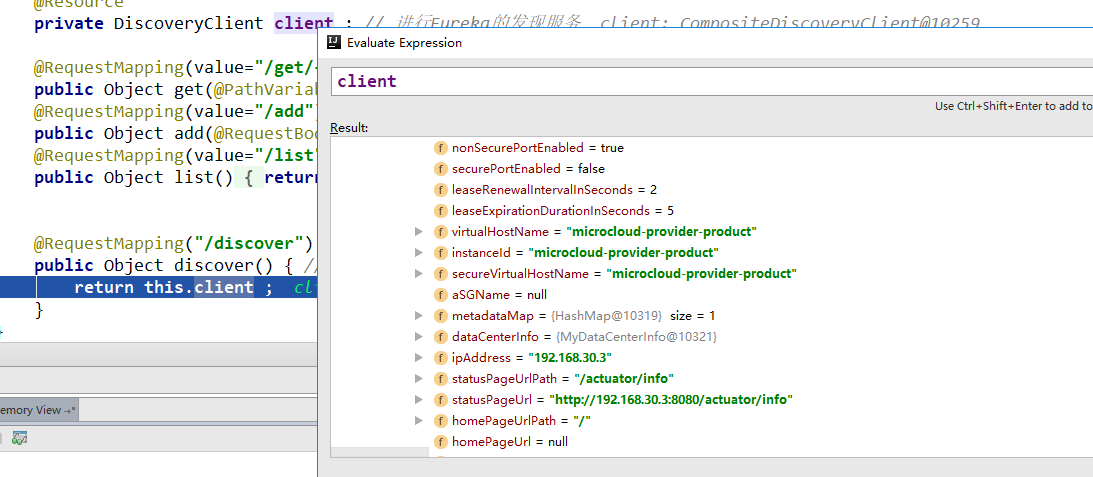
}

}

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

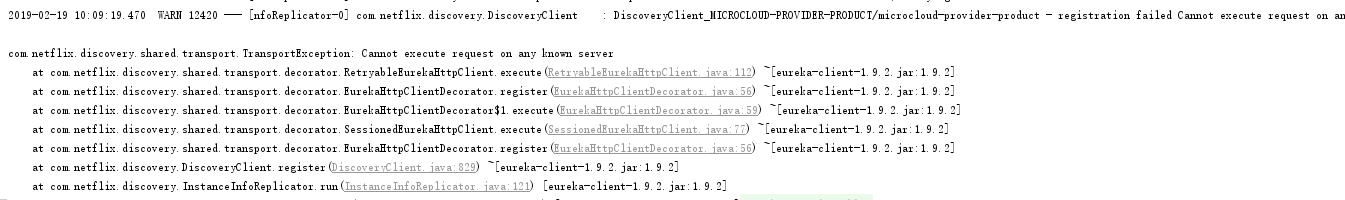
prefer-ip-address: true

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();

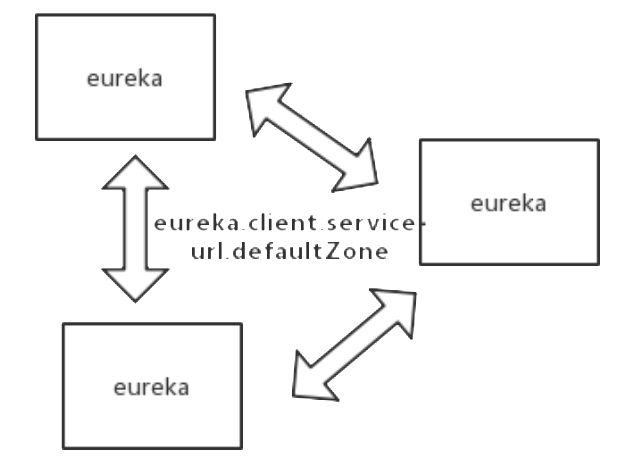
super.configure(http);

}

}

## 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:

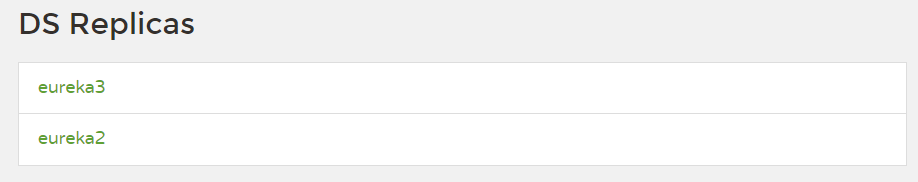
user:

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>

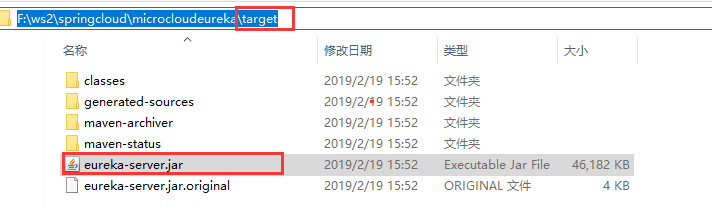
</project>

【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：**

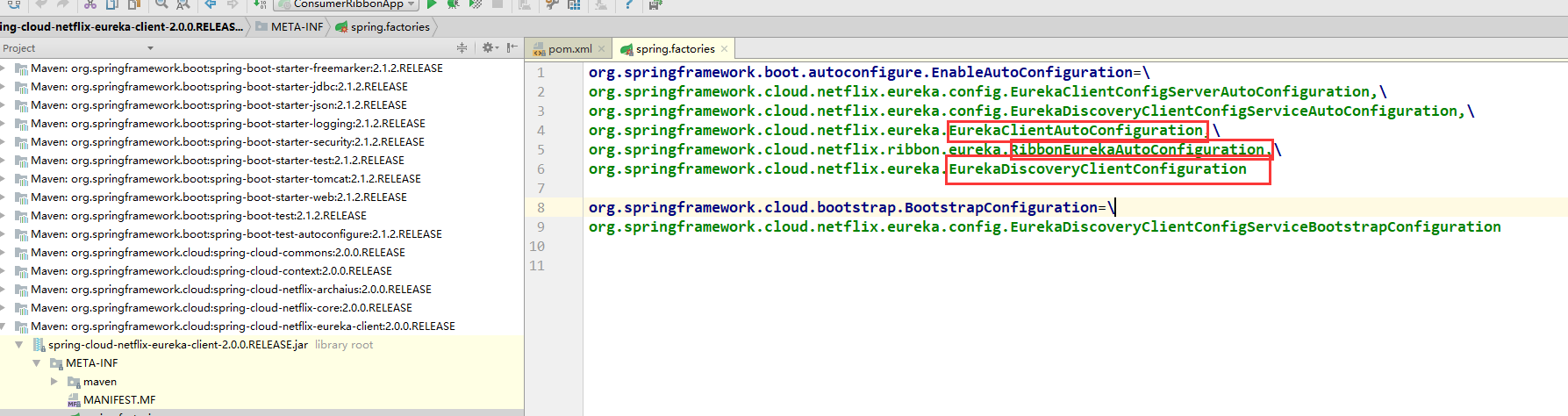
EurekeClient 自动配置类,负责Eureka关键Beans的配置和初始化,如AppplicationInfoManager和 EurekaClientConfig等。

* **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

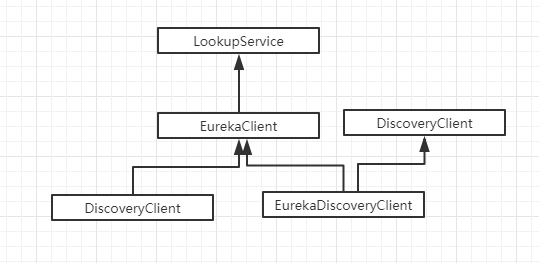
封装EurekaClient自身服务实例的配置信息,主要用于构建InstanceInfo通常这些信息在配置文件中的eureka.instance前缀下进行设置, SpringCloud通过EurekalnstanceConfigBean配置类提供了默认配置

* 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");

}

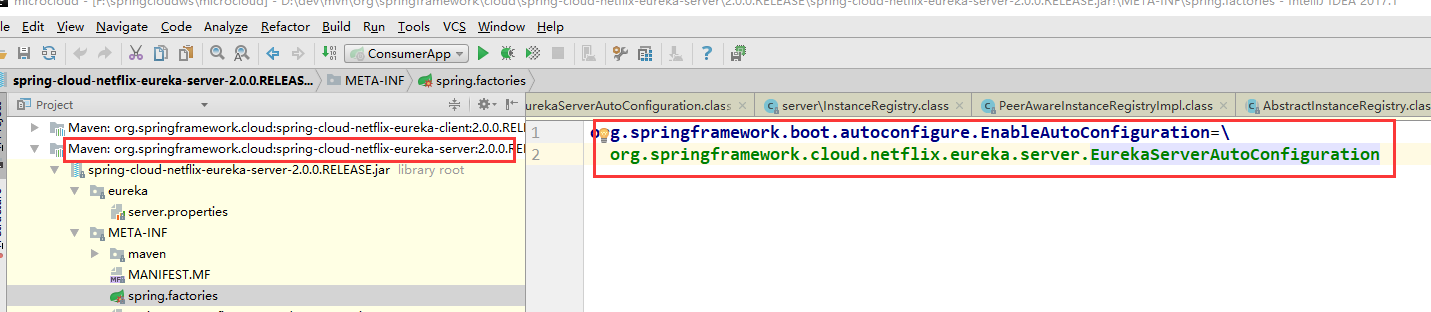
}

### 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

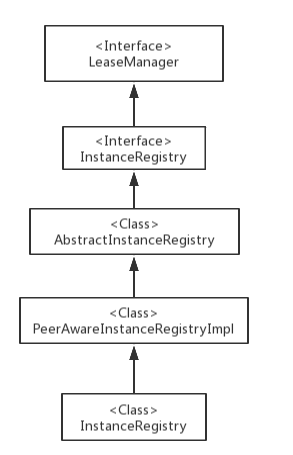
public PeerAwareInstanceRegistry peerAwareInstanceRegistry(ServerCodecs serverCodecs) {

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);

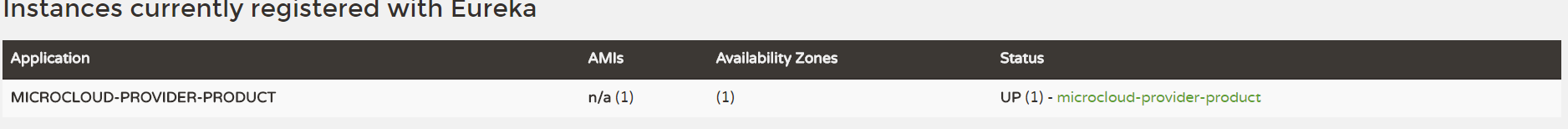
}

}

【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;

}

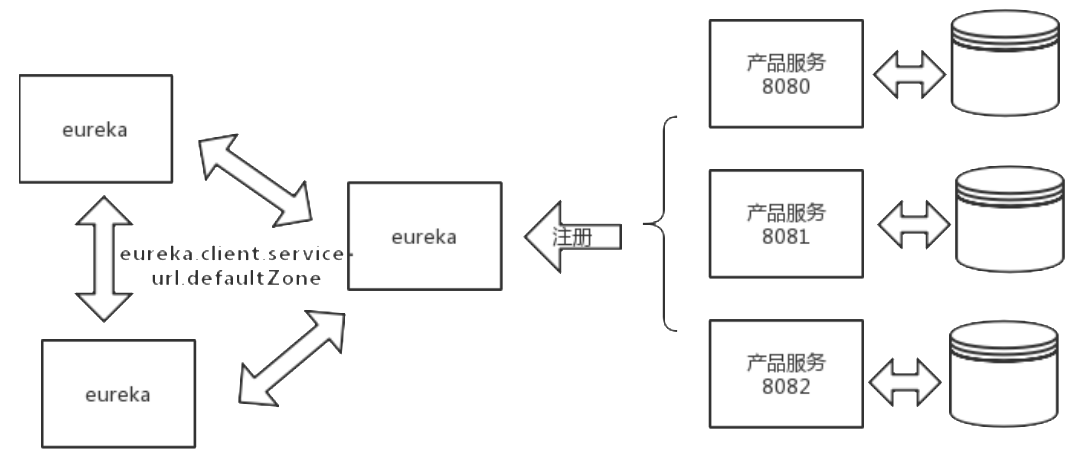
}

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

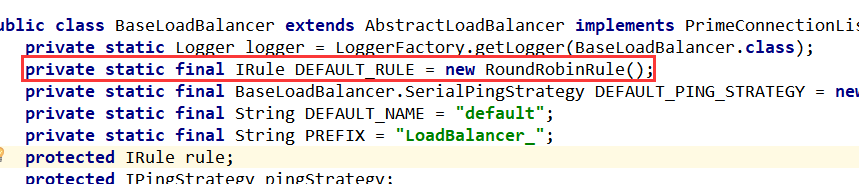
【microcloud-consumer】启动

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",

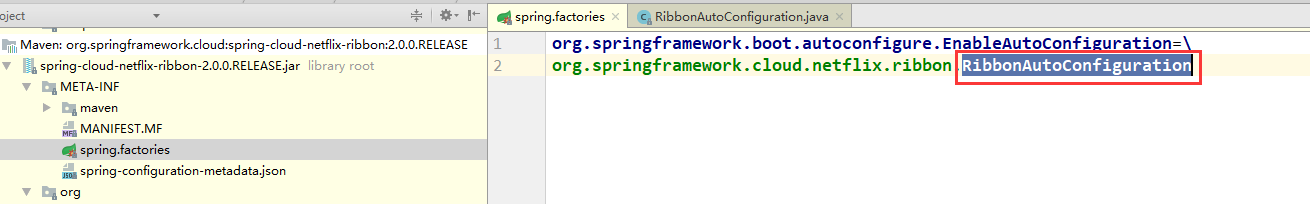
builder.getBeanDefinition());

}

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

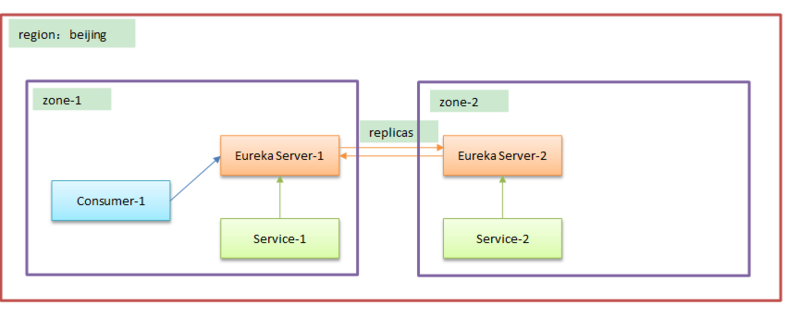
**ServerListFilter**

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);

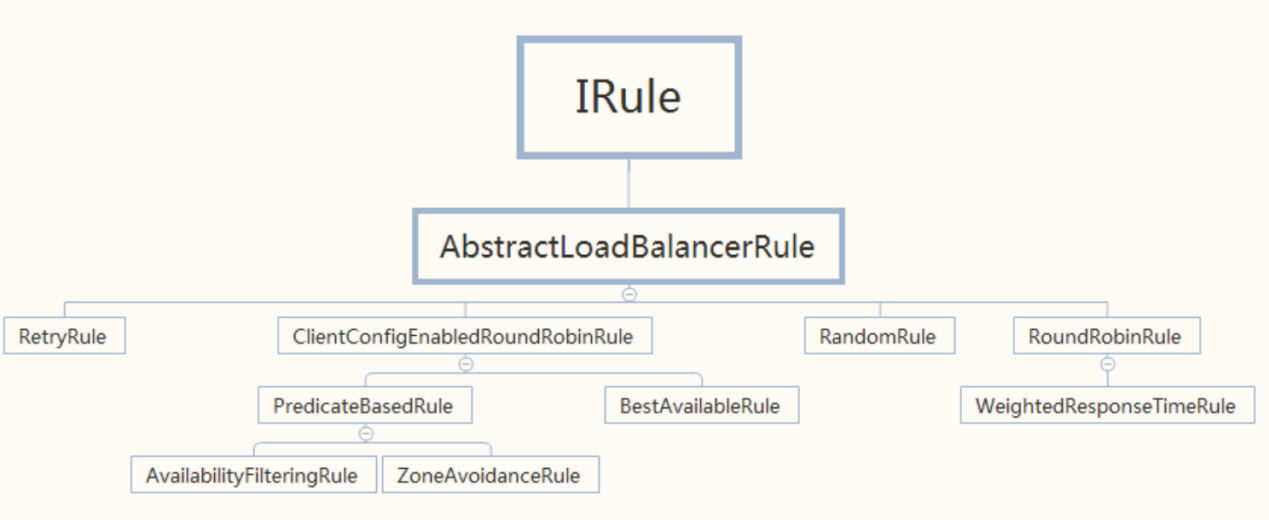
}

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

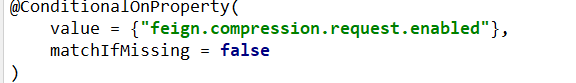
## 其他配置

### 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。

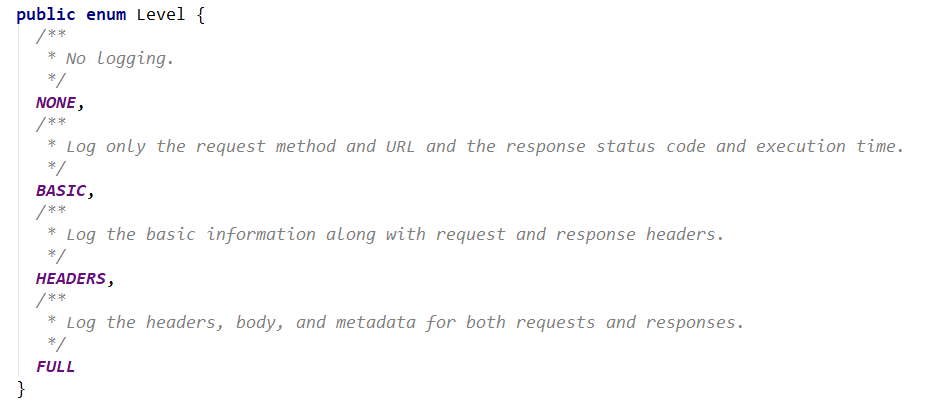
【microcloud-consumer-feign】 修改 application.yml配置文件，增加日志信息

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