1. Feign
2. 概述

之前使用Ribbon的负载均衡功能，简化了远程调用时的代码，但是每次调用都需要写基本相同的代码，代码重复性高。Feign可以把Rest的请求进行隐藏，伪装成类似Controller一样，我们不需要拼接url，这些工作都可以让Feign来完成。

1. Feign实例
2. 在服务消费方添加依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>

1. 开启Feign功能

在服务消费方的引导类上添加注解@EnableFeignClients，用于开启Feign功能。

1. 创建Feign客户端

创建一个Feign客户端接口，在该接口上添加注解@FeignClient(“user-service”)，该注解声明了当前的类是一个Feign客户端，并且指定了访问的服务名是user-service。Feign客户端就是为了拼接URL，接口内的每个方法都是一个请求方法，在每个方法上添加注解访问的URL，如@GetMapping(“/user/{id}”)，动态代理会自动生成该接口的实现类，将该接口的服务名和请求方法的URL拼接起来，组成最终访问提供服务的URL.。

package com.springboot.client;  
  
import com.springboot.pojo.User;  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
  
@FeignClient("user-service")  
public interface ConsumerFeignClient {  
  
 @GetMapping("/user/{id}")  
 User queryById(@PathVariable Long id);  
}

1. 创建处理器，注入Feign客户端

创建一个处理器Controller，注入Feign客户端，使用该客户端接口的方法queryById，动态代理会自动创建该接口的实现类。那么该方法实现的就是访问客户端拼接成的服务地址，返回服务数据。

@RestController  
@RequestMapping("feign")  
public class ConsumerFeignController {  
  
 @Autowired  
 private ConsumerFeignClient feignClient;  
  
 @GetMapping("/{id}")  
 public User queryById(@PathVariable Long id){  
 return feignClient.queryById(id);  
 }  
}

1. 测试

启动服务提供方，Eureka Server，服务消费方，通过Feign的Controller设定的地址访问其中的方法，就可以获取相应的服务数据。

1. Feign的负载均衡和熔断
2. Feign配置Ribbon

Feign中已经集成了Ribbon依赖和自动配置，因此不需要额外引入依赖，也不需要再注册RestTemplate对象。Feign内置的Ribbon默认设置了请求超时时长，默认是1000，我们可以通过手动配置来修改：

ribbon:

ReadTimeout: 2000 #读取超时时长

ConnectionTimeout: 1000 #建立连接的超时时长

Ribbon内部有重试机制，超时后会自动重新发起请求，如果想修改，如下所示：

ribbon:

ConnectionTimeout: 1000 #连接超时时长

ReadTimeout: 2000 #数据通信超时时长

MaxAutoRetries: 0 #当前服务器的最大重试次数

MaxAutoRetriesNextServer: 0

OkToRetryOnAllOperations: false #是否对所有的请求方式都重试

1. Feign配置Hystrix
2. 开启Hystrix

同样的，Feign中已经内置了Hystrix，直接通过配置来开启Hystrix，如下所示：

feign:

hsytrix:

enabled: true

1. 创建服务降级的类

创建一个在服务降级时调用的类，该类中的每个方法都与Feign客户端接口中的方法对应，因此直接实现Feign客户端接口，重写每个方法，编写访问服务失败时的操作。注意，要在该类之上添加注解@Component，让Spring知道这是一个组件。

@Component

public class FeignClientFallback implements ConsumerFeignClient {  
 @Override  
 public User queryById(Long id) {  
 User user = new User();  
 user.setId(id);  
 user.setName("error");  
 return user;  
 }  
}

1. 修改Feign客户端的@FeignClient注解

修改Feign客户端的@FeignClient注解，添加fallback参数，绑定服务降级类到Feign客户端中。

@FeignClient(value="user-service", fallback = FeignClientFallback.class)

1. 测试

设定Hystrix的超时时间为2秒，同时在Feign的Controller中设置休眠两秒，制造异常，发现服务降级成功。

1. Feign配置压缩

Spring Cloud Feign能够对请求和响应进行GZIP压缩，以减少通信过程中的性能损耗。直接通过配置feign中的参数即可开启压缩功能，如下所示：

feign:

compression:

request:

enabled: true #请求压缩

mime-types: text/html,application/xml,application.json #压缩的数据类型

mim-request-size: 2048 #设置触发压缩的大小下限

response:

enabled: true #响应压缩

1. Feign配置日志级别

一般的包的日志级别可以通过在配置文件中设置 logging.level.xx=debug,就可以为xx这个包设定日志级别为debug。然而对于被@FeignClient注解修改过的Feign客户端接口来说，当它被代理时，会创建一个新的Feign.Logger实例，我们需要额外指定这个日志的级别。需要创建一个Feign的配置类，用于创建新的Logger实例。

1. 修改包的日志级别

logging:

level:

com.springboot: debug

1. 添加Feign的配置类

添加Feign的配置类，定义日志级别。注意，要在该配置类上添加@Configuration表示这是一个配置类。

package com.springboot.config;  
  
import feign.Logger;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class FeignConfig {  
  
 @Bean  
 Logger.Level feignLoggerLevel(){  
 return Logger.Level.*FULL*;  
 }  
}

Feign支持的日志级别有4种：

1. NONE： 不记录任何日志信息，默认
2. BASIC： 只记录请求的方法，URL以及响应状态码和执行时间
3. HEADERS：在BASIC的基础上，添加了请求和响应的头信息
4. FULL：记录所有请求和响应的明细，包括头信息，请求体，元数据
5. 在Feign的客户端中指定配置类

这个日志的配置类是为Feign客户端接口配置的，因此需要在Feign的客户端接口中通过注解@FeignClient指定该配置类。

@FeignClient(value="user-service", fallback = FeignClientFallback.class, configuration = FeignConfig.class)

1. 测试

开启服务消费方，正常访问后，日志就会打印在控制台中。

1. Spring Cloud Gateway网关
2. 概述

Spring Cloud Gateway是Spring自己开发的网关服务，基于Filter链提供网关基本功能：安全，监控，限流，能够为微服务架构提供简单，有效且统一的API路由管理。

1. 原理

Spring Cloud Gateway组件的核心是过滤器，通过这些过滤器可以将客户端发送的请求路由到对应的微服务。Spring Cloud Gateway是设置在微服务前端的防火墙和代理器，能够隐藏微服务节点的ip端口信息，从而加强安全保护。一切来自客户端的请求或者是服务内部调用，只要是对服务的请求都可以经过网关，然后通过网关来验证，路由。

1. 概念
2. 路由(route)

路由信息由一个ID，一个目的URL，一组断言工厂，一组Filter组成。如果路由断言为真，说明请求URL和配置路由匹配。

1. 断言（Predicate）

断言函数允许开发者自定义匹配来气Http Request中的任何信息。

1. 过滤器（Filter）

Filter分为两种，一种是Gateway Filter, 一种是Global Filter，过滤器会对请求和响应进行修改处理。

1. Gateway项目实战
2. 需求分析

需求是将包含有/user的URL请求路由到<http://localhost:8080/user/id>中

1. 创建工程

创建一个Maven工程springcloud-gateway。

1. 添加依赖

需要添加两个依赖，一个是gateway的，还有一个是Eureka client的，因为gateway本身也是一个服务，需要注册到Eureka Server中。

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 <version></version>  
 </dependency>  
</dependencies>

1. 创建引导类即配置文件

在引导类上添加@EnableDiscoveryClient注解表示能够发现Eureka Server，并注册到Eureka Server中。

package com.gateway;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
  
@SpringBootApplication  
@EnableDiscoveryClient  
public class GatewayApp {  
 public static void main(String[] args) {  
 SpringApplication.*run*(GatewayApp.class, args);  
 }  
}

设置application.yml，基础的设置不用过多说明，主要是网关gateway的配置。在gateway的routes下有多个路由，每个路由表示一种路由方式。id表示路由的id，可以任意编写；uri表示要代理的服务地址，即路由到的服务地址；predicates表示断言，可以有多个断言，断言即判断请求是否满足断言的条件，如Path路径中含有/user时就进行路由。

server:  
 port: 8084  
spring:  
 application:  
 name: api-gateway  
 cloud:  
 gateway:  
 routes:  
 *#路由id* - id: user-service-route  
 *#代理的服务地址* uri: http://127.0.0.1:8080  
 *#路由断言* predicates:  
 - Path=/user/\*\*  
eureka:  
 client:  
 service-url:  
 defaultZone: http://127.0.0.1:8082/eureka  
 instance:  
 prefer-ip-address: true

1. 测试

通过访问网关的端口8084，并且在路径中添加/user/7，符合路由的断言，那么该请求就会被路由到8080的服务地址，并且路径中的/user/7同样被添加到访问服务的请求URL中，访问成功。

1. 面向服务的路由
2. 概述

当同一个服务有多个实例的话，需要使用动态路由来从网关路由到相应的服务地址，这就需要修改之前设置的网关的uri，使用服务的服务名称来动态路由。当uri使用的协议为lb时，网关将使用LoadBalncerClient把user-service通过eureka解析为实际的主机和端口，并进行Ribbon负载均衡。

1. 修改网关配置

修改网关的配置文件，将路由的uri修改为 lb://服务名称 这种格式。

uri: lb://user-service

1. 路由前缀处理
2. 概述

用户发送请求到网关时，不一定是正确的请求地址，可能会有所偏差，这时就需要我们来为用户修正请求地址了。

1. 添加前缀

通过配置路由的过滤器PrefixPath，实现映射路径中地址的添加。添加如下参数：

Spring:

cloud:

gateway:

routes:

* id: user-service-route

uri: lb://user-service

predicates:

-Path=/\*\*

filters:

-PrefixPath=/user

上面的配置中，断言的路径是所有路径，它将所有的路径都放进来，但是在过滤器中添加了路径的前缀 /user，即为所有进来的请求路径中都添加了/user 前缀，如请求为localhost:8084/7，进来以后就变为了localhost:8084/user/7。

1. 去除前缀

同理，通过配置StripPrefix，可以将映射路径中地址的前缀去除。StipPrefix的值为要去掉前缀的个数，以”/”为分割线。如请求路径为/api/user/\*\*，那么当StripPrefix为1时，就去掉/api，当StipPrefix为2时，就去掉/api/user。

Spring:

cloud:

gateway:

routes:

-id: user-service-route

uri: lb://user-service

predicates:

-Path=/api/user/\*\*

filters:

-StripPrefix=1

1. 过滤器
2. 概述

过滤器为网关实现请求的鉴权功能，Gateway自带了几十个过滤器，能够为用户的请求修改很多参数。如：AddRequestHeader能够对匹配上的请求添加Header，AddRequestParematers能够为匹配上的请求路由添加参数，AddResponseHeader能够为从网关上返回的响应添加Header。

1. AddResponseHeader过滤器示例

在网关的配置文件中添加过滤器的配置，本次添加的是全局的默认过滤器，对所有路由都有效，如下所示：

spring:

cloud:

gateway:

routes:

…

default-filters:

-AddResponseHeader=X-Response-Foo, Bar

过滤器的值为两个，第一个是要添加的参数名，第二个是该参数的值。

1. 过滤器类型
2. 局部过滤器

通过spring.cloud.gateway.routes.filters 配置在具体的路由下面，只作用在当前路由上，自带的过滤器都可以配置或者自定义的过滤器。像我们上面配置spring.cloud.gateway.default-filters可以对所有的路由生效，这是一种全局的过滤器，但是这些过滤器的实现都是要实现GatewayFilterFactory接口的。

1. 全局过滤器

全局过滤器不需要再配置文件中配置，作用在所有的路由中，实现GlobalFilter接口即可。

1. 过滤器执行生命周期

在过滤器执行的前后可以执行一些操作，即通过过滤器的GatewayFilterChain执行filter方法的前后来实现。

如，请求鉴权时，在执行filter方法之前，进行鉴定访问权限，执行过滤器后，如果没有权限，就返回空。同样的，在异常处理时，执行filter方法后，记录异常并返回异常。

1. 自定义局部过滤器
2. 需求分析

编写并配置一个自定义的局部过滤器，该过滤器可以通过配置文件中的参数名称获取请求的参数值。如将请求<http://locahost:8084/user/8?name=xx中的>参数name的值获取并且输出到控制台。注意，自定义过滤器的名称是由固定的后缀GatewayFilterFactory的，前面可以添加自定义的名称，如在前面加上MyParam，则自定义过滤器的名称为MyParamGatewayFilterFactory，在配置时只需要配置MyParam即可。

1. 配置文件

在网关的配置中添加自定义的过滤器，如下所示：

spring:  
 cloud:

gateway:

routes:

-id:user-service-route

uri: lb://user-service

predicates:

-Path=/api/user/\*\*

filters:

-StripPrefix=1

-MyParam=name

1. 创建自定义过滤器

（1）创建配置类

创建自定义的过滤器MyParamGatewayFilterFactory，过滤器都要继承一个抽象类AbstractGatewayFilterFactory<>，需要传入一个泛型，我们在配置中配置了参数name，但是无法得知name的值的类型。因此需要创建一个配置类，动态地获取参数name的值，并将该类传到泛型的位置。

public class MyParamGatewayFilterFactory extends AbstractGatewayFilterFactory<MyParamGatewayFilterFactory.Config>{

public static class Config{

private String param;

public String getParam(){  
 return param;

}

public void setParam(String param){

this.param=param;

}

}

}

（2）添加构造函数

public MyParamGatewayFilterFactory(){

super(Config.class);

}

（3）添加shortcutFieldOrder函数

该函数的作用是将配置类中的参数添加到列表中，这样才能够从配置文件中读取数据。注意PARAM-NAME的值要和配置类中的参数名一致，都为param。

static final String PARAM\_NAME = “param”;

public List<String> shortcutFieldOrder(){

return Arrays.asList(PARAM\_NAME);

}

(4)重写apply方法

apply方法是过滤器的逻辑所在，如何过滤是该方法编写的内容。传入配置类作为参数，通过获取请求Request，判断请求参数中是否包含了我们需要的参数，如果有，就打印该参数。注意，此时在配置类中的config.param指的就是我们在网关的配置中填入的name参数。

@Override  
public GatewayFilter apply(Config config) {  
 return (exchange, chain) -> {  
 ServerHttpRequest request = exchange.getRequest();  
 if(request.getQueryParams().containsKey(config.param)){  
 request.getQueryParams().get(config.param).forEach(value-> System.*out*.printf("局部过滤器 ： %s = %s", config.param, value ));  
 }  
 return chain.filter(exchange);  
 };  
}

1. 测试

正常地通过网关进行访问，当我们在访问的url后面添加name参数时，如localhost:8084/7?name=xx控制台就会打印name参数的值xx了。

1. 自定义全局过滤器
2. 需求分析

定义一个全局过滤器，检查请求地址是否携带token参数，若token参数存在则放行；如果token参数不存在则设置返回的状态码为：未授权不再继续执行。

1. 创建全局过滤器

全局过滤器需要继承GlobalFilter类，如果想让过滤器的执行有序的话，还要继承Ordered类。重写filter方法与getOrder方法，filter方法中编写的是过滤器的过滤逻辑，判断获取请求的参数中是否有token参数，如果有则继续执行，如果没有则停止执行。getOrder方法是用于过滤器的排序，返回的值越小就越先执行。

public class MyGlobalFilter implements GlobalFilter, Ordered {  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 System.*out*.println("去哪聚过滤器： ");  
 String token = exchange.getRequest().getQueryParams().getFirst("token");  
 if(StringUtils.*isBlank*(token)){  
 //设置响应状态码为未授权  
 exchange.getResponse().setStatusCode(HttpStatus.*UNAUTHORIZED*);  
 return exchange.getResponse().setComplete();  
 }  
 return chain.filter(exchange);  
 }  
  
 @Override  
 public int getOrder() {  
 //值越小过滤器越先执行  
 return 1;  
 }  
}

1. Gateway的跨域问题

跨域问题是在JS的请求中，如果访问的地址与当前服务器的域名，ip或者端口号不一致则成为跨域请求，若不能跨域则无法获取到请求地址的返回结果。因此，我们需要在网关中解决跨域问题，直接在网关中配置即可，配置有哪些服务是可以跨域请求的。配置如下：

Spring:

cloud:

gateway:

globalcors:  
 cors-configurations:  
 '[/\*\*]':  
 *#表示允许访问的服务器地址* allowedOrigins:  
 - "http://doc.spring.io"  
 allowedMethods:  
 - GET

allowedOrigins表示指定允许访问的服务器地址

allowedMethods表示允许的方法

‘[/\*\*]’表示对所有访问到网关服务器的请求地址，因此上面的配置表示可以允许来自<http://docs.spring.io的get>请求方法获取服务数据。

1. Gateway的高可用

在服务内部之间访问时，可以通过启动多个Gateway服务，形成集群自动负载均衡。在外部访问Gateway时，这些请求无法通过Eureka进行负载均衡，需要使用其他的服务网关来对Gateway进行代理，如Nginx。

1. Gateway与Feign的区别

Gateway用于外部请求对微服务的调用，Feign则是将当前微服务的部分服务接口暴露出来，主要用于各个微服务之间的服务调用。

1. Spring Cloud Config 分布式配置中心
2. 概述

在分布式系统中，由于服务数量众多，每个微服务项目中都有自己的配置文件，无法统一地管理所有的配置文件。这个问题可以通过分布式配置中心组件来解决，在Spring Cloud中提供了Spring Cloud Config组件，支持配置文件放在配置服务的本地，也可以放在远程Git仓库中，这样，我们直接使用Git来管理配置文件即可。

当然，这个配置中心组件也是一个微服务，需要注册在Eureka注册中心中。

1. Git配置管理
2. 创建gitee仓库

在码云平台中创建一个仓库。

1. 在仓库中创建配置文件

配置中心组件是为了在Git上管理配置文件的，因此，我们需要在仓库中创建配置文件。这个配置文件的命名是由规范的，由两部分组成，中间中“-“分隔开。前面一部分是应用名称，后面一部分是用于区分开发环境，测试环境，生产环境的标识。配置文件格式依然是yml和properties两种。如下所示：

user-dev.yml表示应用名为user，在开发环境下使用的配置文件。

将user-service服务的配置文件作为内容在仓库中创建新的配置文件user-dev.yml。

server:  
 port: ${port:8080}  
spring:  
 datasource:  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://localhost:3306/springcloud?autoReconnect=true&useSSL=false  
 username: root  
 password: 123  
 application:  
 name: user-service  
  
mybatis:  
 type-aliases-package: com.user.pojo  
eureka:  
 client:  
 service-url:  
 defaultZone: http://127.0.0.1:8082/eureka  
 instance:  
 ip-address: 127.0.0.1  
 prefer-ip-address: true  
 lease-expiration-duration-in-seconds: 5  
 lease-renewal-interval-in-seconds: 5

1. 创建配置中心工程
2. 配置pom.xml

添加eureka-client 注册中心客户端依赖，config-server配置中心依赖。

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-server</artifactId>  
 </dependency>  
</dependencies>

1. 添加引导类

创建引导类，并且添加注解@EnalbeConfigServer表示开启配置服务。

@SpringBootApplication  
@EnableConfigServer  
public class ConfigServerApp {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ConfigServerApp.class, args);  
 }  
}

1. 创建配置文件

在配置文件中添加端口，服务名称，gitee地址，以及注册到eureka服务器。

server:  
 port: 8085  
  
spring:  
 application:  
 name: config-server  
 cloud:  
 config:  
 server:  
 git:  
 uri: https://gitee.com/Mrwxxx/config-test.git  
  
eureka:  
 client:  
 service-url:  
 defaultZone: http://127.0.0.1:8082/eureka

1. 测试配置中心

访问配置中心的地址，它会直接连接到Gitee的仓库地址中，可以直接访问到仓库中创建的配置文件user-dev,yml。同时，在Gitee中修改配置文件也能够即时地更新到配置中心工程中来。

1. 其他工程获取配置中心中的配置
2. 需求分析

将用户微服务工程user-service的配置文件application.yml删除，改从配置中心config-server中获取。

1. 实现步骤
2. 在user-service工程中添加config的启动器依赖。

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
 <version>2.1.1.RELEASE</version>  
</dependency>

1. 修改配置文件

删除之前的application.yml文件，而添加一个bootstrap.yml，这个配置文件专用于配置固定的参数，动态的参数要设置在Gitee仓库中。

bootstrap.yml中需要配置Spring Cloud config配置中心的参数，如这个用户微服务工程所对应Gitee中的配置文件user-dev.yml，设置name参数为user，设置profile参数为dev，设置label参数为user-dev.yml在Gitee中所处的分支master，同时还有discovery中要开启发现配置中心，以及设置配置中心的服务名config-server。注意，别忘了将该服务注册到Eureka上面。

spring:  
 cloud:  
 config:  
 name: user  
 profile: dev  
 label: master  
 discovery:  
 enabled: true  
 service-id: config-server  
  
eureka:  
 client:  
 service-url:  
 defaultZone: http://127.0.0.1:8082/eureka

1. 测试

启动用户微服务，发现访问依旧是正常的，说明用户微服务能够直接从配置中心中拉取对应的配置文件，，测试成功！！！

1. Spring Cloud Bus服务总线
2. 概述

当我们在Gitee中修改配置的文件后，无法自动更新到项目之中，只有重启项目更新的配置才会生效。那么，如何在不重启微服务的情况下自动更新配置呢？这就需要使用到Spring Cloud Bus了。Spring Cloud Bus是用轻量的消息代理将分布式的节点连接起来，可以用于广播配置文件的更改或者服务的监控管理，也可以实现应用程序之间互相通信。Spring Cloud Bus可选的消息代理有RabbitMQ和Kafka。

1. 原理

配置中心从Git仓库中拉取最新的配置文件，缓存到本地中，各个微服务从本地配置缓存中获取配置文件，并且注册到注册中心Eureka之中。在使用了Spring Cloud Bus总线之后，配置中心中的配置检测到修改后，发送消息到RabbitMQ之中，各个微服务实时监控着RabbitMQ的消息情况，只要有配置文件修改的消息，则通过配置中心重新拉取新的配置文件，实现实时更新功能。

1. Spring Cloud Bus实例
2. 需求分析

启动RabbitMQ通过修改码云中的配置文件后发送Post请求实现及时更新用户微服务中的配置项。

1. 启动RabbitMQ

之前已经在Linux中安装过RabbitMQ，直接启动RabbitMQ。

1. 修改配置中心
2. 修改依赖

添加Spring Cloud Bus需要的依赖。如下所示：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-bus</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-stream-binder-rabbit</artifactId>  
</dependency>

1. 在配置中心中配置rabbitmq

配置中心需要向rabbitmq发送消息，因此需要在配置中心中配置rabbitmq的参数，如host,port,username,password。

spring:

rabbitmq:  
host: 192.168…  
port: 5672  
username: wxx  
password: wxx

1. 在配置中心中配置暴露触发消息总线的地址

配置该地址后就能通过访问该地址来触发消息总线，更新配置文件。

management:  
 endpoints:  
 web:  
 exposure:  
 include: bus-refresh

1. 配置服务提供者的依赖

服务提供者需要监听Rabbitmq，因此需要添加Spring Cloud Bus总线的依赖。

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-bus</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-stream-binder-rabbit</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

1. 在服务提供者中配置rabbitMQ

服务提供者需要监听rabbitMQ，因此需要在配置文件中配置rabbitMQ的参数。与之前的配置相同。

spring:

rabbitmq:  
host: 192.168…  
port: 5672  
username: wxx  
password: wxx

1. 修改服务提供者的Controller

服务提供者的哪个Controller需要实时更新，就在哪个Controller之上添加注解@RefreshScope，表示实时刷新配置。

1. 测试

注意，在启动各个工程时，配置中心要在其他服务之前启动，因为其他服务的配置文件在配置中心上。当我们在Gitee上修改配置文件后，直接访问服务提供者发现配置文件是没有更新的，通过RESTClient工具发送POST方式请求访问http:127.0.0.1:8085/actuator/bus-refresh ，8085是配置中心的端口号，actuator是固定的，bus-refresh是我们在配置中心添加的触发消息总线的地址。访问完该地址后，配置文件就会自动更新。

1. Spring Cloud 体系综合应用

