# 案例1：使用Feign替换原有RestTemplate的调用方式

在服务调用者工程（消费）创建接口（添加注解）

（效果）Feign = RestTemplate+Ribbon+Hystrix

服务消费者工程（页面静态化微服务）中引入Feign依赖（或者父类工程）

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

服务消费者工程（静态化微服务）启动类使用注解@EnableFeignClients添加Feign支持

​  
@SpringBootApplication  
@EnableDiscoveryClient  // 开启服务发现  
@EnableFeignClients   // 开启Feign  
public class PageApplication {  
​  
   public static void main(String[] args) {  
       SpringApplication.run(PageApplication.class,args);  
  }  
    
}

注意：此时去掉Hystrix熔断的支持注解@EnableCircuitBreaker即可包括引入的依赖，因为Feign会自动引入

在消费者微服务中创建Feign接口：

@FeignClient(name = "demo-service-product")  
public interface ProductFeign {  
​  
   /\*\*  
    \* 通过id获取商品信息  
    \* @param id  
    \* @return  
    \*/  
   @RequestMapping("/product/query/{id}")  
   public Products query(@PathVariable Integer id);  
​  
   /\*\*  
    \* 获得端口号  
    \* @return  
    \*/  
   @RequestMapping("/server/query")  
   public String findServerPort();  
​  
}

注意：

1）@FeignClient注解的name属性用于指定要调用的服务提供者名称，和服务提供者yml文件中spring.application.name保持一致

2）接口中的接口方法，就好比是远程服务提供者Controller中的Hander方法（只不过如同本地调用了），那么在进行参数绑定的时，可以使用@PathVariable、@RequestParam、@RequestHeader等，这也是OpenFeign对SpringMVC注解的支持，但是需要注意value必须设置，否则会抛出异常

3) @FeignClient(name = "demo-service-product")，name在消费者微服务中只能出现一次。（升级Spring Boot 2.1.0 Spring Cloud Greenwich.M1 版本后，在2个Feign接口类内定义相同的名字， @FeignClient(name = 相同的名字 就会出现报错，在之前的版本不会提示报错），所以最好将调用一个微服务的信息都定义在一个Feign接口中。

改造PageController中原有的调用方式：

@RestController  
@RequestMapping("/page")  
public class PageController {  
   @Autowired  
   private ProductFeign productFeign;  
   @RequestMapping("/getData/{id}")  
   public Products findDataById(@PathVariable Integer id) {  
       return productFeign.query(id);  
  }  
   @RequestMapping("/getPort")  
   public String getProductServerPort() {  
       return productFeign.findServerPort();  
  }  
}

# 案例2：创建网关微服务并配置路由规则

使用网关对静态化微服务进行代理（添加在它的上游，相当于隐藏了具体微服务的信息，对外暴露的是网关）

使用网关对静态化微服务进行代理（添加在它的上游，相当于隐藏了具体微服务的信息，对外暴露的是网关）

创建工程demo-cloud-gateway-server导入依赖

GateWay不需要使用web模块，它引入的是WebFlux（类似于SpringMVC）

<?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>com.demo</groupId>  
   <artifactId>demo-cloud-gateway</artifactId>  
   <version>1.0-SNAPSHOT</version>  
​  
​  
   <!--spring boot 父启动器依赖-->  
   <parent>  
       <groupId>org.springframework.boot</groupId>  
       <artifactId>spring-boot-starter-parent</artifactId>  
       <version>2.1.6.RELEASE</version>  
   </parent>  
​  
   <dependencies>  
       <dependency>  
           <groupId>org.springframework.cloud</groupId>  
           <artifactId>spring-cloud-commons</artifactId>  
       </dependency>  
       <dependency>  
           <groupId>org.springframework.cloud</groupId>  
           <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
       </dependency>  
       <!--GateWay 网关-->  
       <dependency>  
           <groupId>org.springframework.cloud</groupId>  
           <artifactId>spring-cloud-starter-gateway</artifactId>  
       </dependency>  
       <!--引入webflux-->  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-webflux</artifactId>  
       </dependency>  
       <!--日志依赖-->  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-logging</artifactId>  
       </dependency>  
       <!--测试依赖-->  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-test</artifactId>  
           <scope>test</scope>  
       </dependency>  
       <!--lombok工具-->  
       <dependency>  
           <groupId>org.projectlombok</groupId>  
           <artifactId>lombok</artifactId>  
           <version>1.18.4</version>  
           <scope>provided</scope>  
       </dependency>  
​  
       <!--引入Jaxb，开始-->  
       <dependency>  
           <groupId>com.sun.xml.bind</groupId>  
           <artifactId>jaxb-core</artifactId>  
           <version>2.2.11</version>  
       </dependency>  
       <dependency>  
           <groupId>javax.xml.bind</groupId>  
           <artifactId>jaxb-api</artifactId>  
       </dependency>  
       <dependency>  
           <groupId>com.sun.xml.bind</groupId>  
           <artifactId>jaxb-impl</artifactId>  
           <version>2.2.11</version>  
       </dependency>  
       <dependency>  
           <groupId>org.glassfish.jaxb</groupId>  
           <artifactId>jaxb-runtime</artifactId>  
           <version>2.2.10-b140310.1920</version>  
       </dependency>  
       <dependency>  
           <groupId>javax.activation</groupId>  
           <artifactId>activation</artifactId>  
           <version>1.1.1</version>  
       </dependency>  
       <!--引入Jaxb，结束-->  
​  
       <!-- Actuator可以帮助你监控和管理Spring Boot应用-->  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-actuator</artifactId>  
       </dependency>  
       <!--热部署-->  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-devtools</artifactId>  
           <optional>true</optional>  
       </dependency>  
​  
       <!--链路追踪-->  
       <dependency>  
           <groupId>org.springframework.cloud</groupId>  
           <artifactId>spring-cloud-starter-sleuth</artifactId>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.springframework.cloud</groupId>  
           <artifactId>spring-cloud-starter-zipkin</artifactId>  
       </dependency>  
​  
   </dependencies>  
​  
   <dependencyManagement>  
       <!--spring cloud依赖版本管理-->  
       <dependencies>  
           <dependency>  
               <groupId>org.springframework.cloud</groupId>  
               <artifactId>spring-cloud-dependencies</artifactId>  
               <version>Greenwich.RELEASE</version>  
               <type>pom</type>  
               <scope>import</scope>  
           </dependency>  
       </dependencies>  
   </dependencyManagement>  
​  
   <build>  
       <plugins>  
           <!--编译插件-->  
           <plugin>  
               <groupId>org.apache.maven.plugins</groupId>  
               <artifactId>maven-compiler-plugin</artifactId>  
               <configuration>  
                   <source>11</source>  
                   <target>11</target>  
                   <encoding>utf-8</encoding>  
               </configuration>  
           </plugin>  
           <!--打包插件-->  
           <plugin>  
               <groupId>org.springframework.boot</groupId>  
               <artifactId>spring-boot-maven-plugin</artifactId>  
           </plugin>  
       </plugins>  
   </build>  
​  
</project>

注意：不要引入starter-web模块，需要引入web-flux

application.yml 配置文件内容

server:  
port: 9300  
eureka:  
client:  
  serviceUrl: # eureka server的路径  
    defaultZone: http://demoCloudEurekaServerA:9200/eureka,http://demoCloudEurekaServerB:9201/eureka  
instance:  
  prefer-ip-address: true  
  instance-id: ${spring.cloud.client.ip-address}:${spring.application.name}:${server.port}:@project.version@  
spring:  
application:  
  name: demo-cloud-gateway  
 #网关的配置  
cloud:  
  gateway:  
    routes: #配置路由  
      - id: service-page-router  
        uri: http://127.0.0.1:9100  
        predicates: #当断言成功后，交给某一个微服务处理时使用的是转发  
          - Path=/page/\*\*  
      - id: service-product-router  
        uri: http://127.0.0.1:9000  
        predicates:  
          - Path=/product/\*\*  
        filters:  
           # http://127.0.0.1:9300/product/service/port --> /service/port --> 商品微服务  
          - StripPrefix=1  #去掉uri中的第一部分，所以就要求我们通过网关访问的时候，把uri的第一部分设置为product，从uri的第二部分开始才是真正的uri  
​

启动类

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

# 案例3：自定义全局过滤器实现IP访问限制（黑白名单）

请求过来时，判断发送请求的客户端的ip，如果在黑名单中，拒绝访问。自定义GateWay全局过滤器时，我们实现Global Filter接口即可，通过全局过滤器可以实现黑白名单、限流等功能。

package com.demo.gateway;  
​  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.cloud.gateway.filter.GatewayFilterChain;  
import org.springframework.cloud.gateway.filter.GlobalFilter;  
import org.springframework.core.Ordered;  
import org.springframework.core.io.buffer.DataBuffer;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.server.reactive.ServerHttpRequest;  
import org.springframework.http.server.reactive.ServerHttpResponse;  
import org.springframework.stereotype.Component;  
import org.springframework.web.server.ServerWebExchange;  
import reactor.core.publisher.Mono;  
​  
import java.util.ArrayList;  
import java.util.List;  
​  
/\*\*  
\* 定义全局过滤器，会对所有路由生效  
\*/  
@Slf4j  
@Component  // 让容器扫描到，等同于注册了  
public class BlackListFilter implements GlobalFilter, Ordered {  
​  
   // 模拟黑名单（实际可以去数据库或者redis中查询）  
   private static List<String> blackList = new ArrayList<>();  
​  
   static {  
       blackList.add("0:0:0:0:0:0:0:1");  // 模拟本机地址  
       blackList.add("127.0.0.1");  
  }  
​  
   /\*\*  
    \* 过滤器核心方法  
    \* @param exchange 封装了request和response对象的上下文  
    \* @param chain 网关过滤器链（包含全局过滤器和单路由过滤器）  
    \* @return  
    \*/  
   @Override  
   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
       System.out.println("....BlackListFilter....");  
       // 思路：获取客户端ip，判断是否在黑名单中，在的话就拒绝访问，不在的话就放行  
       // 从上下文中取出request和response对象  
       ServerHttpRequest request = exchange.getRequest();  
       ServerHttpResponse response = exchange.getResponse();  
​  
       // 从request对象中获取客户端ip  
       String clientIp = request.getRemoteAddress().getHostString();  
       // 拿着clientIp去黑名单中查询，存在的话就决绝访问  
       if(blackList.contains(clientIp)) {  
           // 决绝访问，返回  
           response.setStatusCode(HttpStatus.UNAUTHORIZED); // 状态码  
           log.info("=====>IP:" + clientIp + " 在黑名单中，将被拒绝访问！");  
           String data = "Request be denied!";  
           DataBuffer wrap = response.bufferFactory().wrap(data.getBytes());  
           return response.writeWith(Mono.just(wrap));  
      }  
​  
       // 合法请求，放行，执行后续的过滤器  
       return chain.filter(exchange);  
  }  
​  
​  
   /\*\*  
    \* 返回值表示当前过滤器的顺序(优先级)，数值越小，优先级越高  
    \* @return  
    \*/  
   @Override  
   public int getOrder() {  
       return 0;  
  }  
}