

SpringCloud服务调用

1.Ribbon介绍

1.简介

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套**客户端负载均衡工具**，其主要功能是提供**客户端的软件负载均衡算法和服务调用**，Ribbon客户端组件提供一系列完善的配置项，如连接超时，重试等。简单地说，就是在配置文件中列出Load Balancer后面所有的机器，Ribbon会自动的帮助我们实现某种规则（如简单轮询，随机连接等）去连接这些机器。我们可以很容易的使用Ribbon来实现自定义的负载均衡算法。

2.功能

负载均衡（LB）：将用户的请求平摊的分配到多个服务商，从而达到系统的HA(High Available 高可用)，常见的负载均衡软件有Nginx,LVS,硬件 F5等。

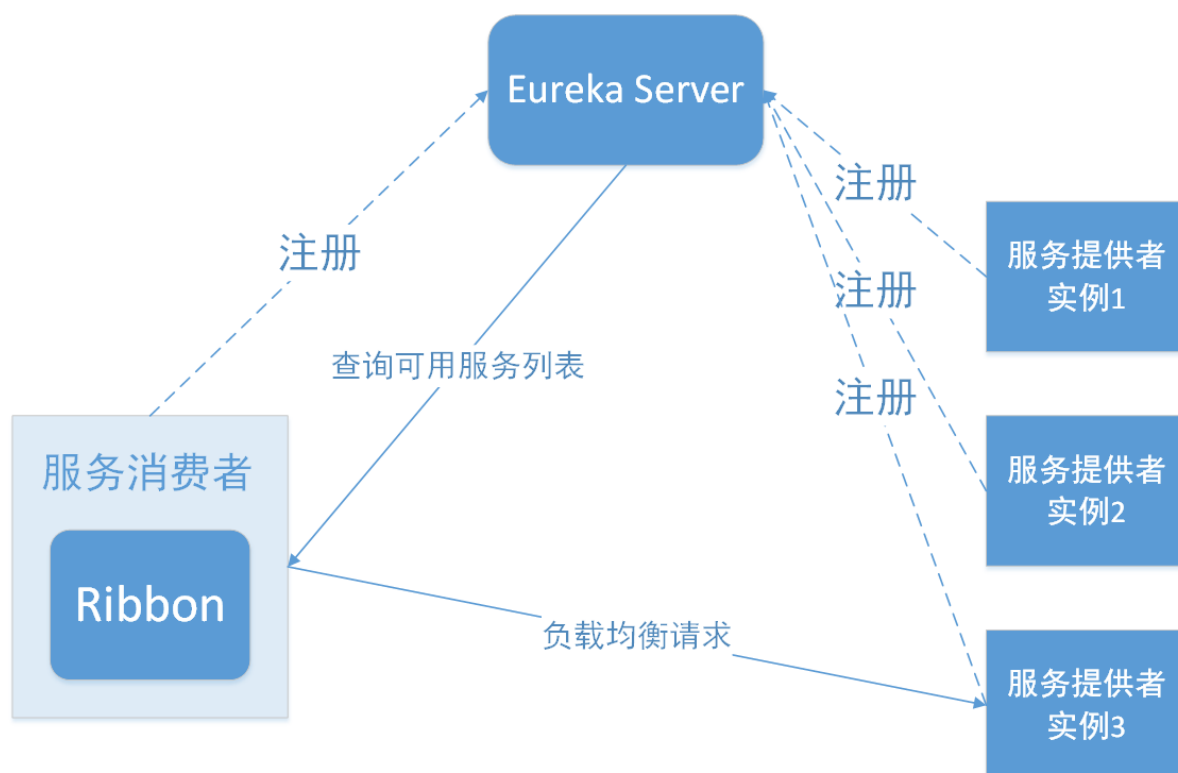
Ribbon本地负载均衡客户端和Nginx服务端负载均衡有什么区别？

Nginx是服务器负载均衡，客户端的所有请求都会交给Nginx,然后由Nginx实现转发请求。即负载均衡是由服务端实现的。服务器来实现负载均衡。（**集中式的LB（负载均衡）**：在服务的消费方和提供方之间使用独立的负载均衡设施，可以是硬件、软件，由该设施负责把访问请求通过某种策略转发至服务方）。

Ribbon是本地负载均衡，在调用微服务接口时，会在注册中心上获取注册信息服务列表，然后将其缓存到本地JVM，从而在本地实现RPC远程服务调用技术。（**进程内的LB(负载均衡)**：将负载均衡的逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己从这些地址中选择一个合适的服务器。Ribbon就属于进程内的负载均衡器，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址）。

Ribbon其实就是一个软负载均衡的客户端组件，他可以和其他所需请求的客户端结合使用，和eureka结合使用只是其中一个实例。说白了就是负载均衡+ RestTemplate

3.架构



Ribbon在工作时分为两步：

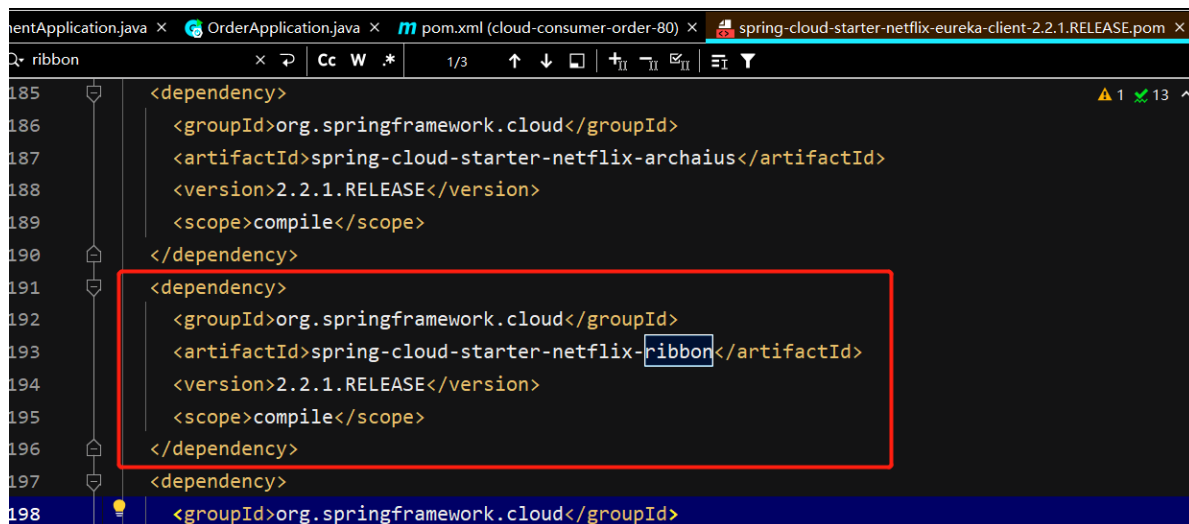
第一步：选择EurekaServer,它优先选择在同一个区域内负载较少的server

第二部：根据用户所制定的策略，在从server取到的服务注册列表选择一个地址，其中，Ribbon提供了多种策略，比如轮询、随机和根据响应时间加权

2.Ribbon的使用

1.引入

关于引入，其实新版的eureka-client中已经自己继承了ribbon，所以不用引入。有Eureka依赖即可



```
185 <dependency>
186     <groupId>org.springframework.cloud</groupId>
187     <artifactId>spring-cloud-starter-netflix-archaius</artifactId>
188     <version>2.2.1.RELEASE</version>
189     <scope>compile</scope>
190 </dependency>
191 <dependency>
192     <groupId>org.springframework.cloud</groupId>
193     <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
194     <version>2.2.1.RELEASE</version>
195     <scope>compile</scope>
196 </dependency>
197 <dependency>
198     <groupId>org.springframework.cloud</groupId>
```

3.再谈RestTemplate

1.getForObject方法和getForEntity方法

getForObject：返回对象为响应体中数据转化成的对象，基本上可以理解为JSON

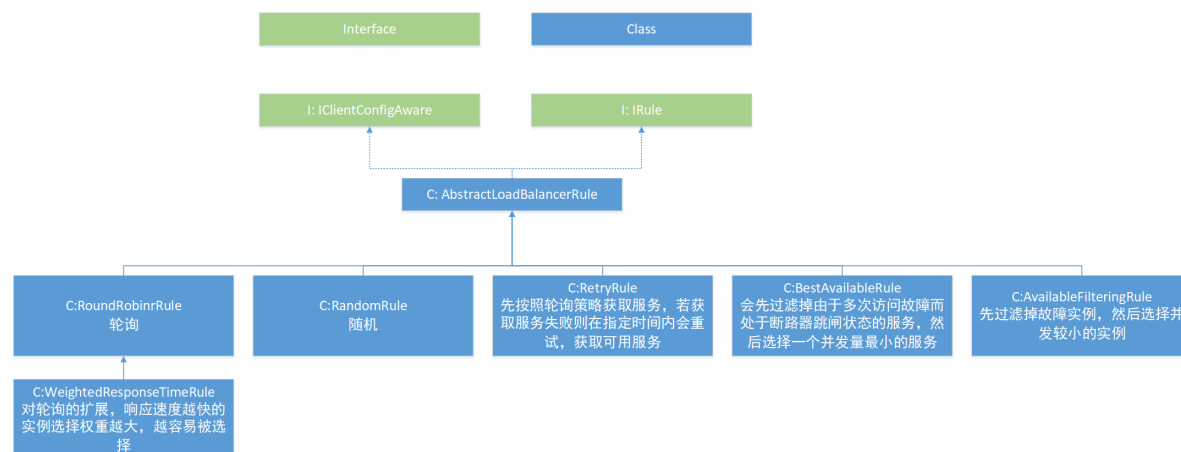
getForEntity：返回对象为ResponseEntity对象，包含了响应中一些重要的信息，如响应头、响应状态码、响应体等

2.postForObject和PostForEntity

区别与get类似

4.Ribbon默认自带的负载规则，核心组件IRule

1.继承图

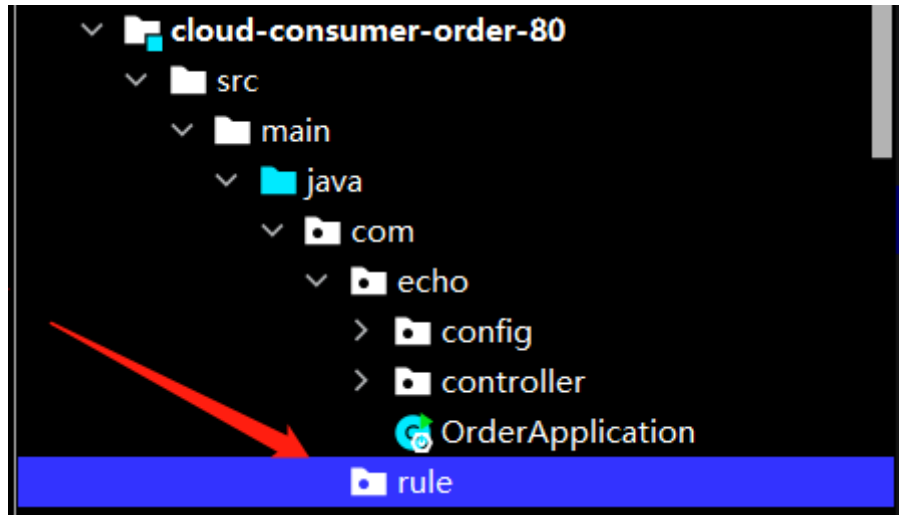


5.Ribbon负载规则的替换

1.修改Order80服务消费方

2.注意点

Ribbon官方明确提出了，自定义的轮询替换规则，不能放在@ComponentScan注解能扫描的包下，在springboot中，由于主启动类添加了@ComponentScan注解，因此，不可以将其放到主启动类所在包及其子包下，需要重新建一个包，来专门存放Ribbon的负载均衡配置



3.在rule包下新建一个MyselfRule的类，用来进行规则的定制

```
@Configuration
public class MyselfRule {

    @Bean
    public IRule myRule(){
        return new RandomRule();//返回一个随机的规则
    }
}
```

4.在主启动类上新增一个注解

```
@SpringBootApplication
//标识为Eureka客户端
@EnableEurekaClient
//该注解标识，在访问CLOUD-PAYMENT-SERVICE服务时，使用MyselfRule.class中定义的负载均衡规则
@RibbonClient(name = "CLOUD-PAYMENT-SERVICE",configuration = MyselfRule.class)
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class,args);
    }
}
```

6.Ribbon默认负载轮询算法原理

1.注释掉@RibbonClient注解，恢复成原来的轮询算法

2.负载均衡算法：rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标，每次服务重启后rest接口计数从1开始

3.例子：

```
List instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
```

如：List[0] instances = 127.0.0.1:8002

List[1] instances = 127.0.0.1:8001

8001 + 8002组合成为集群，它们共计两台机器，集群总数为2，按照轮询算法原理：

总请求数为1时： $1 \% 2 = 1$ 对应下表位置为1，访问127.0.0.1:8001

总请求数为2时： $2 \% 2 = 0$ 对应下表位置为0，访问127.0.0.1:8002

以此类推.....

7.RoundRuleRibbon源码分析

```
//  
// Source code recreated from a .class file by IntelliJ IDEA  
// (powered by FernFlower decompiler)  
//  
  
package com.netflix.loadbalancer;  
  
import com.netflix.client.config.IClientConfig;  
import java.util.List;  
import java.util.concurrent.atomic.AtomicInteger;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class RoundRobinRule extends AbstractLoadBalancerRule {  
    private AtomicInteger nextServerCyclicCounter;  
    private static final boolean AVAILABLE_ONLY_SERVERS = true;  
    private static final boolean ALL_SERVERS = false;  
    private static Logger log = LoggerFactory.getLogger(RoundRobinRule.class);  
  
    public RoundRobinRule() {  
        this.nextServerCyclicCounter = new AtomicInteger(0);  
    }  
  
    public RoundRobinRule(ILoadBalancer lb) {  
        this();  
        this.setLoadBalancer(lb);  
    }  
  
    public Server choose(ILoadBalancer lb, Object key) {  
        if (lb == null) {  
            log.warn("no load balancer");  
            return null;  
        } else {  
            Server server = null;  
            int count = 0;  
  
            while(true) {  
                if (server == null && count++ < 10) {  
                    List<Server> reachableServers = lb.getReachableServers();  
                    List<Server> allServers = lb.getAllServers();  
                    int upCount = reachableServers.size();  
                    int serverCount = allServers.size();  
                    if (upCount != 0 && serverCount != 0) {
```

```

        int nextServerIndex =
this.incrementAndGetModulo(serverCount);
        server = (Server)allServers.get(nextServerIndex);
        if (server == null) {
            Thread.yield();
        } else {
            if (server.isAlive() && server.isReadyToServe()) {
                return server;
            }

            server = null;
        }
        continue;
    }

    log.warn("No up servers available from load balancer: " +
1b);

    return null;
}

    if (count >= 10) {
        log.warn("No available alive servers after 10 tries from
load balancer: " + 1b);
    }

    return server;
}
}
}

private int incrementAndGetModulo(int modulo) {
    int current;
    int next;
    do {
        current = this.nextServerCyclicCounter.get();
        next = (current + 1) % modulo;
    } while(!this.nextServerCyclicCounter.compareAndSet(current, next));

    return next;
}

public Server choose(Object key) {
    return this.choose(this.getLoadBalancer(), key);
}

public void initWithNiwsConfig(IClientConfig clientConfig) {
}
}

```

8.手写负载均衡轮询算法

1.8001和8002添加一个获取端口号的接口

```

@GetMapping("/payment/lb")
public String getPaymentLB(){
    return serverPort;
}

```

2. 注释掉服务消费方80的@LoadBalance注解，从而使我们的自定义负载均衡生效

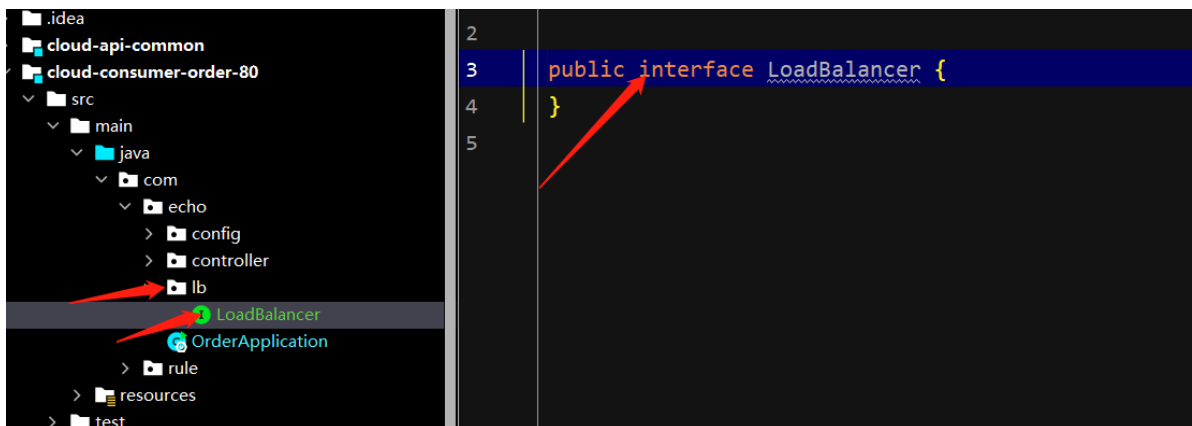
```

@Configuration
public class ApplicationContextConfig {

    @Bean
    // @LoadBalanced //负载均衡，根据服务名，找到不同的主机名，这个注解是开启Ribbon的负载均衡功能的注解
    public RestTemplate getRestTemplate(){return new RestTemplate();}
}

```

3. 新建一个自己的负载均衡的接口



```

public interface LoadBalancer {
    // 将所有的服务实例，装到List中
    ServiceInstance instances(List<ServiceInstance> serviceInstances);
}

```

4. 新建一个类来实现我们定义的负载均衡接口

```

@Component
public class MyLoadBalancerImpl implements LoadBalancer{

    private AtomicInteger atomicInteger = new AtomicInteger(0);
    public final int getAncIncrement(){
        int current;
        int next;
        // 用自旋锁取下一次访问的值
        do {
            current = this.atomicInteger.get();
            next = current >= Integer.MAX_VALUE ? 0 : current + 1;
        } while (!this.atomicInteger.compareAndSet(current, next));
        System.out.println("*****第几次访问，次数next*****" + next);
        return next;
    }

    @Override
    public ServiceInstance instances(List<ServiceInstance> serviceInstances) {
        int index = getAncIncrement() % serviceInstances.size();
        return serviceInstances.get(index);
    }
}

```

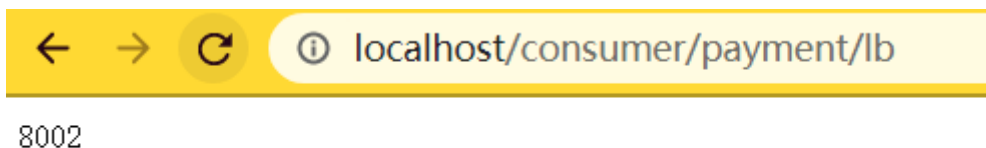
```
}
```

5.修改80的Controller,引入我们自己的负载均衡

```
@Autowired
private LoadBalancer loadBalancer;
@Autowired
private DiscoveryClient discoveryClient;

@GetMapping("/consumer/payment/lb")
public String getPaymentLB(){
    //1.通过DiscoveryClient来获取所有CLOUD-PAYMENT-SERVICE的实例
    List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-
PAYMENT-SERVICE");
    if(instances == null || instances.size() <= 0){
        //服务无效
        return null;
    }
    //2.将所有的实例传递给我们定义的负载均衡算法，拿到经过负载均衡的实例
    ServiceInstance serviceInstance = loadBalancer.instances(instances);
    URI uri = serviceInstance.getUri();
    return restTemplate.getForObject(uri + "/payment/lb",String.class);
}
```

6.进行测试，访问localhost/consumer/payment/lb



多次测试可以发现负载均衡成功

9.OpenFeign简介

1.简介

Feign是一个声明式WebService客户端，使用Feign能让编写Web Service客户端更加简单，**它的使用方法是定义一个服务接口，然后上面添加注解**。Feign也支持可插拔式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持Spring MVC标准注解和HttpMessageConverts。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

2.功能

Feign旨在使编写Java Http客户端更加容易。前面在使用Ribbon + RestTemplate时，利用RestTemplate对http请求进行了封装，形成了一套模板化的调用方法。但是在实际开发过程中，由于对服务依赖的调用可能不止一处，**往往一个接口会被多处调用，所以通常会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用**。所以，Feign在此基础上做了进一步的封装，由它来帮我们定义和实现依赖服务接口的定义。在Feign的实现下，**我们只需创建一个接口并使用注解的方式来配置它（以前是Dao接口上面标注Mapper注解，现在是一个微服务接口上面标注一个Feign注解即可）**，即可完成对服务提供方的接口绑定，简化了使用Spring Cloud Ribbon时，自动封装服务调用客户端的开发量。

3.拓展

Feign集成了Ribbon，利用Ribbon维护了Payment的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，**通过Feign只需定义服务绑定接口，且以声明式的方法**，优雅且简单的实现了服务调用

4. Feign和OpenFeign的区别

Feign	OpenFeign
Feign是Spring Cloud组件中的一个轻量级RESTful的HTTP服务客户端 Feign内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。Feign的使用方式是：使用Feign的注解定义接口，调用这个接口，就可以调用服务注册中心的服务	OpenFeign是Spring Cloud 在Feign的基础上支持了SpringMVC的注解，如@RequesMapping等等。OpenFeign的@FeignClient可以解析SpringMVC的@RequestMapping注解下的接口，并通过动态代理的方式产生实现类，实现类中做负载均衡并调用其他服务。
<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-feign</artifactId> </dependency></pre>	<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-openfeign</artifactId> </dependency></pre>

10. OpenFeign服务调用

1. 接口+注解

微服务调用接口 + @FeignClient

2. 新建cloud-consumer-feign-order-80工程

pom

```
<?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>SpringCloudStudyNew</artifactId>
    <groupId>com.echo</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>cloud-consumer-feign-order-80</artifactId>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
  </properties>
  <dependencies>
    <!--      引入openfeign,它天生整合了ribbon-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <!--      引入EurekaClient-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <!--      引入自定义的公共模块-->
    <dependency>
      <groupId>com.echo</groupId>
      <artifactId>cloud-api-common</artifactId>
      <version>${project.version}</version>
    </dependency>
```



```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--      springboot热部署-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行
周期需要其参与。与compile相比，跳过编译而已，-->
  <scope>runtime</scope>
  <!--      <optional>true</optional>表示两个项目之间依赖不传递；不设
置optional或者optional是false，表示传递依赖。-->
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
</dependencies>
</project>

```

application.yml

```

server:
  port: 80
eureka:
  client:
    register-with-eureka: false #不把它注册到eureka
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka #集群版

```

主启动

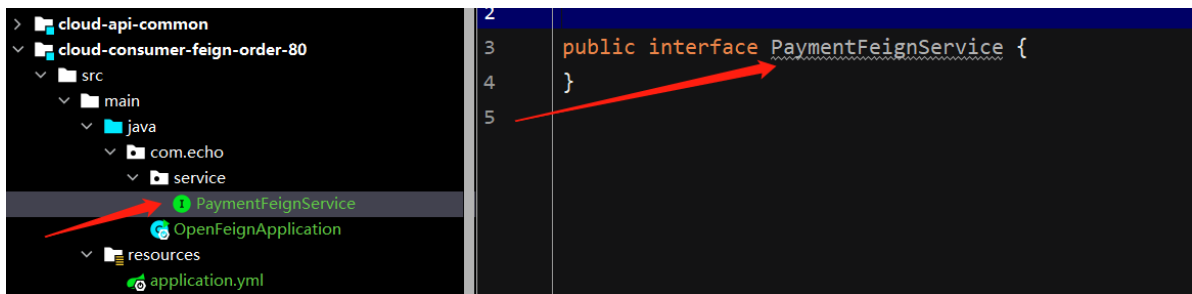
```

@SpringBootApplication
@EnableFeignClients //开启Feign
public class OpenFeignApplication {
  public static void main(String[] args) {
    SpringApplication.run(OpenFeignApplication.class,args);
  }
}

```

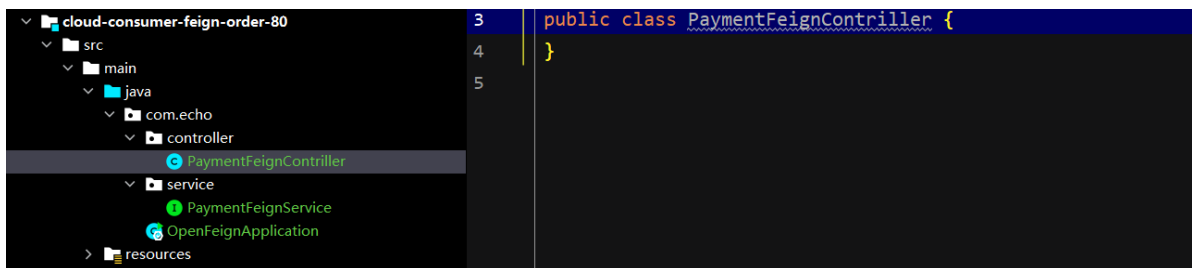
3.业务类

新建一个service包，并创建一个接口，用来远程调用



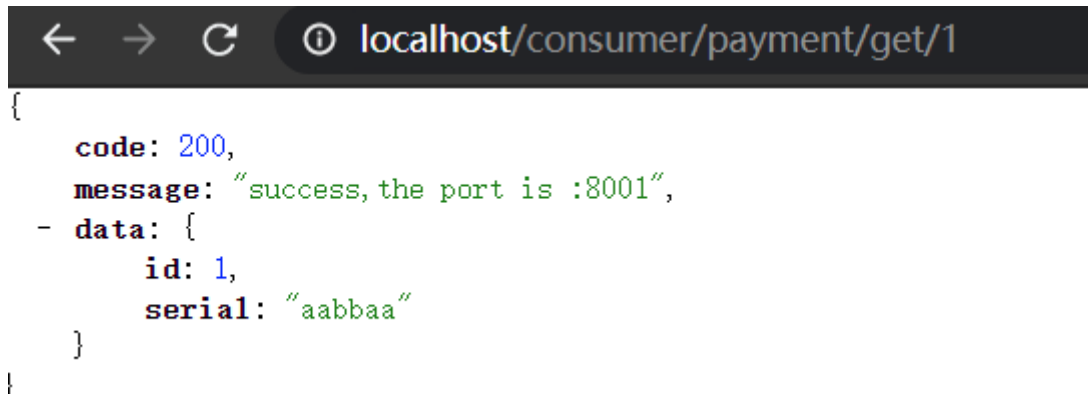
```
@Component  
@FeignClient(value = "CLOUD-PAYMENT-SERVICE") //调用的服务名称  
public interface PaymentFeignService {  
    @GetMapping(value = "/payment/get/{id}")  
    public CommonResult getPaymentById(@PathVariable("id")Long id);  
}
```

4.控制层



```
@RestController  
@Slf4j  
public class PaymentFeignController {  
    @Autowired  
    private PaymentFeignService paymentFeignService;  
  
    @GetMapping("/consumer/payment/get/{id}")  
    public CommonResult getPaymentById(@PathVariable("id")Long id){  
        return paymentFeignService.getPaymentById(id);  
    }  
}
```

5.测试



经过测试也可以发现，OpenFeign也自带负载均衡

11.OpenFeign超时控制

1.超时

服务提供者完成某项服务需要三秒钟，而服务消费者只能等两秒钟，这样就存在了时间差。出现了超时现象。

2.服务提供方8001和8002，故意提供一个超时的服务

在服务提供方8001和8002里面提供一个服务超时的方法。

```
public String paymentFeignTimeout(){
    try{
        TimeUnit.SECONDS.sleep(3);
    }
    catch (InterruptedException e){
        e.printStackTrace();
    }
    return serverPort;
}
```

3.在feign服务消费方的controller和service中添加服务调用接口

service

```
@GetMapping("/payment/feign/timeout")
public String paymentFeignTimeout();
```

controller

```
@GetMapping("/consumer/payment/feign/timeout")
public String paymentFeignTimeout(){
    //openfeign的底层默认是ribbon，客户端默认等待1秒钟，但是我们服务提供方是3秒钟
    return paymentFeignService.paymentFeignTimeout();
}
```

4.报错

默认Feign客户端只等待一秒钟，但是服务端处理需要超过1秒钟，导致Feign客户端不想等待了，直接返回报错，为避免这样的情况，有时需要设置Feign客户端的超时控制

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat May 08 14:56:27 CST 2021

There was an unexpected error (type=Internal Server Error, status=500).

Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout

feign.RetryableException: Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout

```
at feign.FeignException.errorExecuting(FeignException.java:213)
at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:115)
at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:80)
at feign.ReflectiveFeign$FeignInvocationHandler.invoke(ReflectiveFeign.java:103)
at com.sun.proxy.$Proxy156.paymentFeignTimeout(Unknown Source)
at com.echo.controller.PaymentFeignController.paymentFeignTimeout(PaymentFeignController.java:26)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:190)
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:138)
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:106)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:888)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:793)
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1040)
```

5.设置feign客户端超时时间（OpenFeign默认支持ribbon）

```

server:
  port: 80
eureka:
  client:
    register-with-eureka: false #不把它注册到eureka
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka #集群版
# 设置Feign客户端超时时间(OpenFeign默认支持Ribbon)
ribbon:
  # 指的是建立连接所用时间,适用于网络状况正常的情况下,两端连接所需要的时间
  ReadTimeout: 5000
  # 指的是建立连接后从服务器读取到可用资源所用的时间
  ConnectTimeout: 5000

```

12.OpenFeign日志打印功能

1.是什么?

Feign提供了日志打印功能,我们可以通过配置来调整日志级别,从而了解Feign中Http请求的细节。说白了就是对Feign接口的调用情况进行监控和输出

2.日志级别

NONE:默认的,不显示任何日志

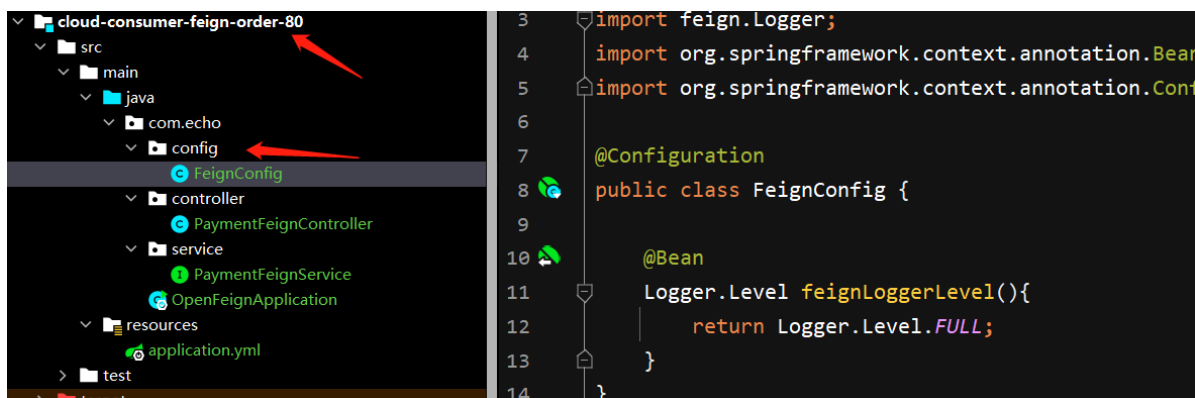
BASIC:仅记录请求方法、URL、响应状态码以及执行时间

HEADERS:除了BASIC中定义的信息之外,还有请求和响应的头信息

FULL:除了HEADERS中定义的信息之外,还有请求和响应的正文和元数据。

3.配置日志bean

创建一个config包,然后写一个配置类



```

@Configuration
public class FeignConfig {

    @Bean
    Logger.Level feignLoggerLevel(){
        return Logger.Level.FULL; //开启一个详细日志
    }
}

```

4.在yml中添加日志配置

```
logging:
  level:
    # feign日志, 以什么级别, 监视哪个接口(以debug方式监控PaymentFeignService接口)
    com.echo.service.PaymentFeignService: debug
```

5. 可以从控制台中看到详细日志

```
2021-05-08 15:15:53.217 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService      :
[PaymentFeignService#paymentFeignTimeout] ---> GET http://CLOUD-PAYMENT-
SERVICE/payment/feign/timeout HTTP/1.1
2021-05-08 15:15:53.217 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService      :
[PaymentFeignService#paymentFeignTimeout] ---> END HTTP (0-byte body)
2021-05-08 15:15:53.370 INFO 1524 --- [p-nio-80-exec-2]
c.netflix.config.ChainedDynamicProperty  : Flipping property: CLOUD-PAYMENT-
SERVICE.ribbon.ActiveConnectionsLimit to use NEXT property:
niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
2021-05-08 15:15:53.406 INFO 1524 --- [p-nio-80-exec-2]
c.n.u.concurrent.ShutdownEnabledTimer    : Shutdown hook installed for:
NFLoadBalancer-PingTimer-CLOUD-PAYMENT-SERVICE
2021-05-08 15:15:53.408 INFO 1524 --- [p-nio-80-exec-2]
c.netflix.loadbalancer.BaseLoadBalancer  : Client: CLOUD-PAYMENT-SERVICE
instantiated a LoadBalancer: DynamicServerListLoadBalancer:
{NFLoadBalancer:name=CLOUD-PAYMENT-SERVICE,current list of Servers=[],Load
balancer stats=Zone stats: {},Server stats: []}ServerList:null
2021-05-08 15:15:53.416 INFO 1524 --- [p-nio-80-exec-2]
c.n.l.DynamicServerListLoadBalancer      : Using serverListUpdater
PollingServerListUpdater
2021-05-08 15:15:53.448 INFO 1524 --- [p-nio-80-exec-2]
c.netflix.config.ChainedDynamicProperty  : Flipping property: CLOUD-PAYMENT-
SERVICE.ribbon.ActiveConnectionsLimit to use NEXT property:
niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
2021-05-08 15:15:53.450 INFO 1524 --- [p-nio-80-exec-2]
c.n.l.DynamicServerListLoadBalancer      : DynamicServerListLoadBalancer for
client CLOUD-PAYMENT-SERVICE initialized: DynamicServerListLoadBalancer:
{NFLoadBalancer:name=CLOUD-PAYMENT-SERVICE,current list of Servers=
[192.168.45.1:8002, 192.168.45.1:8001],Load balancer stats=Zone stats:
{defaultzone=[Zone:defaultzone; Instance count:2; Active connections count: 0;
Circuit breaker tripped count: 0; Active connections per server: 0.0;]
},Server stats: [[Server:192.168.45.1:8001; 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:192.168.45.1:8002; 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.DomainExtractingSer
verList@65c8c596
```

```
2021-05-08 15:15:54.423 INFO 1524 --- [erListUpdater-0]
c.netflix.config.ChainedDynamicProperty : Flipping property: CLOUD-PAYMENT-
SERVICE.ribbon.ActiveConnectionsLimit to use NEXT property:
niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
2021-05-08 15:15:56.536 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] <--- HTTP/1.1 200 (3318ms)
2021-05-08 15:15:56.537 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] connection: keep-alive
2021-05-08 15:15:56.537 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] content-length: 4
2021-05-08 15:15:56.537 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] content-type: text/plain;charset=UTF-8
2021-05-08 15:15:56.537 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] date: Sat, 08 May 2021 07:15:56 GMT
2021-05-08 15:15:56.537 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] keep-alive: timeout=60
2021-05-08 15:15:56.537 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout]
2021-05-08 15:15:56.539 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] 8002
2021-05-08 15:15:56.539 DEBUG 1524 --- [p-nio-80-exec-2]
com.echo.service.PaymentFeignService :
[PaymentFeignService#paymentFeignTimeout] <--- END HTTP (4-byte body)
```