# **基于微服务架构的服务治理**

## 一、微服务与服务治理相关技术分析

## 二、功能实现

本次实验实现了商品微服务，模拟商家在销售中对自身中对自身库存进行查询、得到指定商品的库存数目以及罗列出目前仓库中所有商品的库存、进货后对商品的库存进行更改。同时，为保证服务的健壮性和安全性，将服务进行划分，主要实现了以下功能:

**1. 服务提供与注册（Eureka Server）**

**2. 服务发现与使用（Eureka Client）**

**3. 负载均衡（Ribbon和Feign）**

**4. 服务监控（Dashboard）**

**5. 网关服务（Zuul）**

**6. 服务过滤（Zuul Filter）**

**7. 服务熔断（Hystrix）**

在我们的微服务架构中，一个请求需要调用多个服务是很常见的。当一个服务处于阻塞状态时，若有大量的请求涌入，容器的线程资源会被消耗完毕，导致服务瘫痪。而服务之间的依赖性会导致故障传播，造成连锁反应，对整个微服务系统造成灾难性的严重后果，即“雪崩”效应。

从源头我们无法杜绝雪崩的发生，但是雪崩的根本原因来自于服务之间的强依赖，所以我们可以提前评估，做好熔断、降级、限流。这一部分我们介绍我们的商品微服务中熔断的实现。

1. Hystrix的实现：

* 配置依赖：在product\_provider中加入hystrix的相关依赖

1. **<dependency>**
2. **<groupId>**org.springframework.cloud**</groupId>**
3. **<artifactId>**spring-cloud-starter-netflix-hystrix**</artifactId>**
4. **<version>**1.4.6.RELEASE**</version>**
5. **</dependency>**

* 增加熔断类

1. @MapperScan("com.chenpeng.springcloud.mapper")
2. @EnableEurekaClient
3. @EnableDiscoveryClient
4. @SpringCloudApplication
5. **public** **class** Provider\_product\_hystrix\_8004 {
6. **public** **static** **void** main(String[] args) {
7. SpringApplication.run(Provider\_product\_hystrix\_8004.**class**, args);
8. }
9. }

* 在ProductController类中配置熔断的业务逻辑

1. @RestController
2. @RequestMapping("/product")
3. **public** **class** ProductController {
4. ......
5. @RequestMapping(value="/list")
6. @HystrixCommand(fallbackMethod = "listFallBack")
7. **public** Object list() {
8. **return** **this**.productService.list() ;
9. }
10. //触发熔断方法
11. **public** Object listFallBack(){
12. Product product = **new** Product();
13. product.setProductId(-1L);
14. product.setProductName("服务熔断");
15. product.setProductDesc("不好意思 服务现在用不了啦 先打回游戏 等会再来吧");
16. **return** product;
17. }
18. ......
19. }
20. 运行测试

* 当product\_provider服务正常时，浏览器正常访问：

（加入正常调用服务时的界面）

* 当服务熔断时，显示如下信息：

（加入服务熔断时的界面）

**8. 服务降级（Feign Circuit）**

服务降级和服务熔断非常类似，都是为了系统的稳定性，防止因为个别微服务的不可用而拖死整个系统服务；在表现上都是让用户感知，该服务暂时不可用请稍后再试；粒度上，都是服务级别的粒度。但降级和熔断存在区别：

* 服务熔断一般是某个服务挂掉了引起的，一般是下游服务，而服务降级一般是从整体的负荷考虑，主动降级；
* 熔断其实是一个框架级的处理，每个微服务都需要，没有层次之分，而降级一般需要对业务有层级之分，一般是从最外围服务开始。

1. 服务降级的实现

* 添加Feign支持

1. **<dependency>**
2. **<groupId>**org.springframework.cloud**</groupId>**
3. **<artifactId>**spring-cloud-starter-openfeign**</artifactId>**
4. **<version>**2.1.0.RELEASE**</version>**
5. **</dependency>**

* 增加Product\_feign\_consumer\_80类：

1. @SpringBootApplication
2. @EnableHystrixDashboard
3. @EnableFeignClients
4. @EnableCircuitBreaker
5. **public** **class** Product\_feign\_consumer\_80 {
6. **public** **static** **void** main(String[] args) {
7. SpringApplication.run(Product\_feign\_consumer\_80.**class**, args);
8. }
9. }

* 在ConsumerController类中添加降级方法

1. @RestController
2. @RequestMapping("/consumer")
3. **public** **class** ConsumerController {
4. @Autowired
5. **private** ProductService productService;
6. @RequestMapping("/product/get/{id}")
7. **public** Object getProduct(@PathVariable("id") **long** id) {
8. Product product = productService.getProduct(id);
9. **return** product;
10. }
11. @RequestMapping("/product/list")
12. **public** Object listProduct() {
13. List<Product> list = productService.listProduct();
14. **return** list;
15. }
16. @RequestMapping("/product/add")
17. **public** Object addPorduct(Product product) {
18. Boolean result = productService.addProduct(product);
19. **return** result;
20. }
21. }

* 申明需要调用的微服务

1. @FeignClient(name = "PRODUCT-PROVIDER", fallback = ProductFeignServiceCallBack.**class**)
2. **public** **interface** ProductService {
3. /\*
4. \*配置调用微服务的接口
5. \*/
6. @RequestMapping(value = "/product/get/{id}", method = RequestMethod.GET)
7. **public** Product getProduct(@PathVariable("id") **long** id);
9. @RequestMapping(value = "/product/list", method = RequestMethod.GET)
10. **public** List<Product> listProduct();
12. @RequestMapping(value = "/product/add", method = RequestMethod.POST)
13. **public** Boolean addProduct(Product product);
15. }

* 实现服务熔断的降级操作

1. @Component
2. **public** **class** ProductFeignServiceCallBack **implements** ProductService {
3. ......
4. **public** List<Product> listProduct() {
5. Product product = **new** Product();
6. product.setProductId(-1L);
7. product.setProductName("服务熔断的降级操作");
8. product.setProductDesc("不好意思啊 现在用不了 先打一把游侠放松下待会再来哦");
9. List<Product> list = **new** ArrayList<Product>();
10. list.add(product);
11. **return** list;
12. }
13. ......
14. 运行测试

服务正常运行时用户正常获取信息

（加入服务正常运行时的界面）

服务降级时用户看到的结果

（加入服务降级时访问的界面）

**9. 服务限流（Gateway Filter）**

在网关中实现限流，也可以达到保护系统的目的，可以认为是服务降级的一种。一般来说系统的吞吐量是可以被测算的，为了保证系统的稳固运行，一旦达到需要限制的阈值，就需要限制流量并采取少量措施以完成限制流量的目的。

局部过滤器（Gateway Filter）是针对单个路由的过滤器，可以对访问的URL过滤，进行切面处理。我们实现了如下三种方式的限流：

* 基于请求路径的限流
* 基于请求IP地址的限流
* 基于请求用户的限流

具体实现如下：

1. @Configuration
2. **public** **class** KeyResolverConfiguration {
3. //基于请求路径的限流
4. @Bean
5. **public** KeyResolver PathKeyResolver(){
6. **return** exchange -> Mono.just(
7. exchange.getRequest().getPath().toString()
8. );
9. }
10. //基于请求ip地址的限流
11. //    @Bean
12. **public** KeyResolver IpKeyResolver(){
13. **return** exchange -> Mono.just(
14. exchange.getRequest().getHeaders().getFirst("X-Forwarded-For")
15. );
16. }
17. //基于请求用户的限流
18. //    @Bean
19. **public** KeyResolver UserKeyResolver(){
20. **return** exchange -> Mono.just(
21. exchange.getRequest().getQueryParams().getFirst("user")
22. );
23. }
24. }