http://blog.51cto.com/zero01/2171735

RabbitMQ

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-amqp -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

<version>2.0.4.RELEASE</version>

</dependency>

第一种方式 需要手动到rabbitmq创建队列myQueue

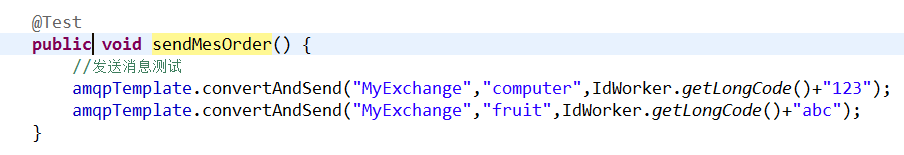
|  |
| --- |
| //接收消息  @Component  @Slf4j  **public** **class** MqReceiver {    @RabbitListener(queues = "myQueue")  **public** **void** process(String message){  Log.*info*("这是接收的消息" + message);  }  }  @Autowired  **private** AmqpTemplate amqpTemplate;  @Test  **public** **void** sendMes() {  //发送消息测试  amqpTemplate.convertAndSend("myQueue",IdWorker.*getLongCode*());  } |

第二种 自动创建queue和Exchange

|  |
| --- |
| @RabbitListener(bindings = @QueueBinding(  value=@Queue("myQueue"),  exchange = @Exchange("MyExchange")  ))  **public** **void** process(String message){  Log.*info*("这是接收的消息" + message);  } |

分析场景

分组的场景 订单服务只关心订单的消息 水果商只关心水果单的消息这个时候就要使用exchange分组







使用queue和key来分组消息

## Spring cloud Stream 对消息中间件的进一步封装

发送接收消息

第一步

|  |
| --- |
| **public** **interface** StreamClient {    @Input("myMessage")  SubscribableChannel input();    @Output("myMessage")  MessageChannel output();  } |

第二步

|  |
| --- |
| @Component  @EnableBinding(StreamClient.**class**)  @Slf4j  **public** **class** StreamReceiver {    //接收方  @StreamListener("myMessage")  **public** **void** process(Object message){  Log.*info*("StreamReceiver===" + message);  }    } |

第三部

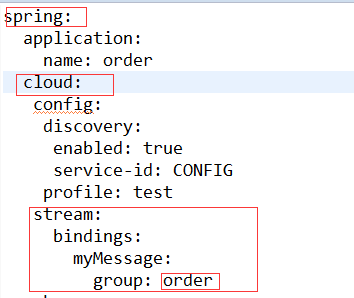
|  |
| --- |
| @RestController  **public** **class** SendMessageController {    @Autowired  **private** StreamClient streamClient;    @GetMapping("/send/{mes}")  **public** **void** sendMes(@PathVariable("mes")String mes){  streamClient.output().send(MessageBuilder.*withPayload*(mes).build());  }  } |

http://localhost:8085//send/faskjfgkjask



如果启动多个服务实例的话都会接收到消息

问题如果只想一个接收到消息的话怎么办 添加下面的配置 把一个服务化分到一个组里面不管多少个实例都只有一个可以接收到



org.springframework.amqp.rabbit.listener.exception.ListenerExecutionFailedException: Listener threw exception

**group:** order  
**content-type:** application/json

添加**content-type配置将mq中的消息转换为json格式原来是base64的**

**如何在接收到消息的时候告诉发送者，传统的就是mq发送一条消息过去**

**Stream只需要**

//接收方  
@StreamListener("myMessage")  
@SendTo("myMessage2") //发送给那个人  
public String Objprocess(order message){  
 Log.*info*("Objprocess===" + message);  
 return "我收到消息-并发送了一条"+ IdWorker.*getStringCode*();  
}  
  
//接收方  
@StreamListener("myMessage2") //我就是那个人  
public void Objprocess2(String message){  
 Log.*info*("Objprocess2收到了一条消息===" + message);  
}

Rabbitmq product服务发送到Order服务

/\*发送消息到Order服务\*/  
@GetMapping("/SendMesOrder")  
@ResponseBody  
public String sendToOrder(){  
 String code = IdWorker.*getStringCode*();  
 amqpTemplate.convertAndSend("myProductQueue", "我是订单服务过来的消息 :"+code);  
 return code;  
}

@Component  
@Slf4j  
public class ProductMqServerInfo {  
  
  
 @RabbitListener(queues = "myProductQueue")  
 public void process(String message){  
 Log.*info*("fruit这是接收的消息" + message);  
 }  
  
}

## Zuul 高可用 安全性 并发性 稳定性

核心过滤器

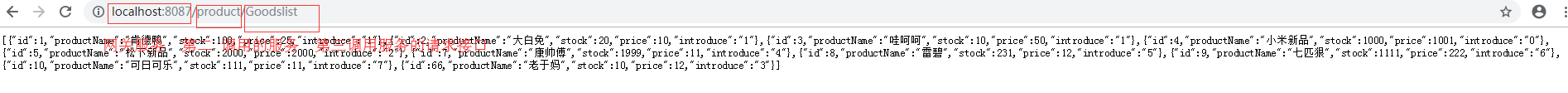
前置 后置

版本注意一致

**spring:  
 application:  
 name:** zuul  
 **cloud:  
 config:  
 discovery:  
 enabled:** true  
 **service-id:** CONFIG  
 **profile:** test  
**eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8761/eureka/  
**server:  
 port:** 9999

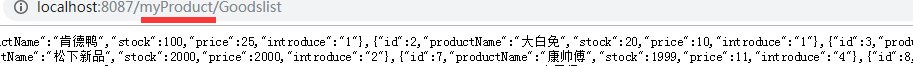
@SpringBootApplication  
@EnableZuulProxy  
public class ZuulApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ZuulApplication.class, args);  
 }  
}

通过网关访问product的list <http://localhost:8086/Goodslist>



**zuul:  
 routes:  
 myProduct:  
 path:** /myProduct/\*\*  
 **serviceId:** product

更改访问的服务名 一样可以访问



查看所有的路由规则



版本不同访问方式可能不同 具体查看日志

简洁写法

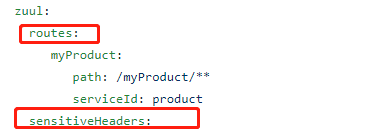
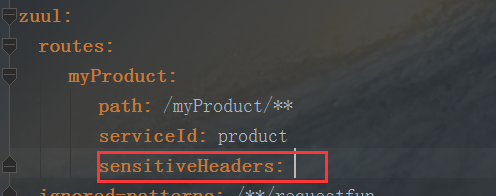
**zuul:  
 routes:  
 product:** /myProduct/\*\*

禁止路由访问product服务的requestfun这个接口 版本不同配置方式可能有差异

**zuul:  
 routes:  
 product:** /myProduct/\*\*  
 **ignored-patterns:** /\*\*/requestfun



Cookie无法传递的问题 将配置为空将不会过滤cookie

这个是对的这个是错误的XXXXXXX

动态路由

在启动类添加

@ConfigurationProperties("zuul")  
@RefreshScope  
public ZuulProperties zuulProperties(){  
 return new ZuulProperties();  
}

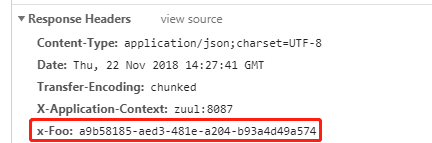
权限校验 token

访问路径必须添加token 也就是对前置路由的一个demo

|  |
| --- |
| package com.hoyan.filter;  import com.netflix.zuul.ZuulFilter; import com.netflix.zuul.context.RequestContext; import org.bouncycastle.util.Strings; import org.springframework.http.HttpStatus; import org.springframework.stereotype.Component; import org.springframework.util.StringUtils;  import javax.servlet.http.HttpServletRequest;  import static org.springframework.cloud.netflix.zuul.filters.support.FilterConstants.*PRE\_DECORATION\_FILTER\_ORDER*; import static org.springframework.cloud.netflix.zuul.filters.support.FilterConstants.*PRE\_TYPE*;   */\*\*  \* Created by 20160709 on 2018/11/22.  \*/* @Component public class TokenFilter extends ZuulFilter {  @Override  public String filterType() {  return *PRE\_TYPE*;//放在前置过滤器的位置生效  }   @Override  public int filterOrder() {  //PRE\_DECORATION\_FILTER\_ORDER这个过滤器的前面   return *PRE\_DECORATION\_FILTER\_ORDER* - 1;  }   @Override  public boolean shouldFilter() {  return true;  }   @Override  public Object run() {  RequestContext requestContext = RequestContext.*getCurrentContext*();  HttpServletRequest request = requestContext.getRequest();  //这里从url参数获取 也可以在cookie header中获取  String token = request.getParameter("token");  if(StringUtils.*isEmpty*(token)){  /\*表示没有权限\*/  requestContext.setSendZuulResponse(false);  //401错误  requestContext.setResponseStatusCode(HttpStatus.*UNAUTHORIZED*.value());  }  return null;  } } |

对返回结果处理

@Component  
public class addRespHeaderFilter extends ZuulFilter {  
 @Override  
 public String filterType() {  
 return *POST\_TYPE*;  
 }  
  
 @Override  
 public int filterOrder() {  
 return *SEND\_ERROR\_FILTER\_ORDER* -1;  
 }  
  
 @Override  
 public boolean shouldFilter() {  
 return true;  
 }  
  
 @Override  
 public Object run() {  
 RequestContext requestContext = RequestContext.*getCurrentContext*();  
 HttpServletResponse response = requestContext.getResponse();  
 response.setHeader("x-Foo", UUID.*randomUUID*().toString());  
 return null;  
 }  
}

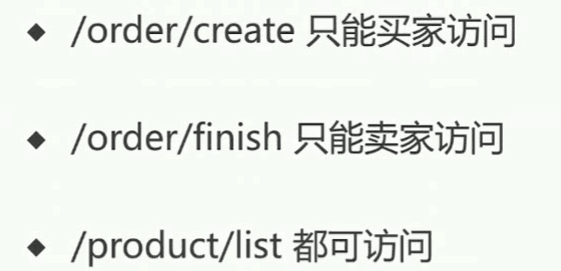


限流

|  |
| --- |
| @Component public class RateFilter extends ZuulFilter {  private static final RateLimiter *RATE\_LIMITER* = RateLimiter.*create*(50);   @Override  public String filterType() {  return *PRE\_TYPE*;  }   @Override  public int filterOrder() {  /\*限流要放在最前面 优先级最高\*/  return *SERVLET\_DETECTION\_FILTER\_ORDER* -1;  }   @Override  public boolean shouldFilter() {  return true;  }   @Override  public Object run() {  if(!*RATE\_LIMITER*.tryAcquire()){  /\*没有令牌就抛出异常\*/  throw new RateException();  }  return null;  } } |



问题实现



/\*createOrder创建订单只能买家访问\*/  
 @RequestMapping("/create")  
 @ResponseBody  
 public String createOrder(){  
 return "买家访问的接口-------------";  
 }  
  
/\*finishOrder完结订单只能卖家访问\*/  
 @RequestMapping("/finish")  
 @ResponseBody  
 public String finishOrder(){  
 return "卖家访问的接口==============";  
 }

TokenFilter

注释掉





通过zuul访问user服务的登录接口成功

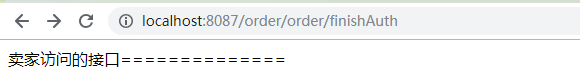
[http://localhost:8087/user//login/buyer/123](http://localhost:8087/user/login/buyer/123)

[http://localhost:8087/user//login/seller/abc](http://localhost:8087/user/login/seller/abc)

注意cookie的传递问题

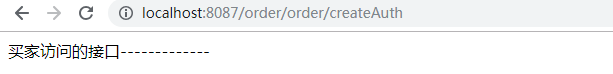


<http://localhost:8087/order/order/finishAuth>

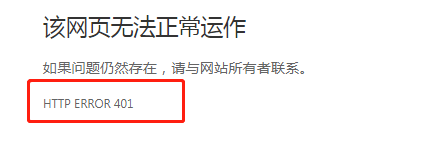




<http://localhost:8087/order/order/createAuth>



权限不满足返回401



分析 这样写存在代码不好维护的问题，可以将不同角色的拦截单独添加一个filter 如下



Zuul跨域

public class CorsConfig {  
  
 @Bean  
 public CorsFilter corsFilter(){  
 final UrlBasedCorsConfigurationSource source =new UrlBasedCorsConfigurationSource();  
 final CorsConfiguration config =new CorsConfiguration();  
 config.setAllowCredentials(true);/\*支持cookie跨域\*/  
 config.setAllowedOrigins(Arrays.*asList*("\*"));//原始域 http:www.a.com  
 config.setAllowedHeaders(Arrays.*asList*("\*"));//头部信息  
 config.setAllowedMethods(Arrays.*asList*("\*"));//方法 GET POST  
 config.setMaxAge(300L);//这个时间段相同的方法不进行检测  
  
 source.registerCorsConfiguration("/\*\*",config);/\*所有域名\*/  
 return new CorsFilter(source);  
 }  
  
}

### 雪崩效应Hystrix

服务降级 优先核心服务

依赖隔离

服务熔断

监控

Product服务定义接口

@PostMapping("/queryProducts")  
@ResponseBody  
public List<Product> findQueryByIds(@RequestBody List<Integer> productIds){  
 System.*out*.println("------------product----------");  
 return productService.findbyProductIdIn(productIds);  
}

Order通过restTemplate来调用接口

@RestController  
public class HystrixController {  
  
 @GetMapping("/getProductInfoList")  
 public String getProductInfoList(){  
 RestTemplate restTemplate =new RestTemplate();  
 return restTemplate.postForObject("http://127.0.0.1:8086/queryProducts", Arrays.*asList*(1,5,8),String.class);  
 }  
  
}

当两个服务正常启动，可以调用的时候。断开product将会导致服务不可用，也不会出现任何提示500错误

添加依赖

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

启动类添加

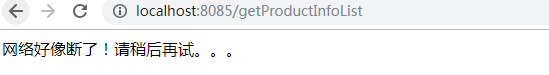
@EnableCircuitBreaker

@SpringCloudApplication

这个注解相当于另外三个注解



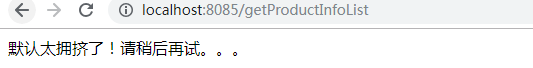
修改之后测试order服务将会出现相应提示，



调用服务不可用或者直接抛出异常将会调用fallback

如果多个方法需要降级的话可以在类中

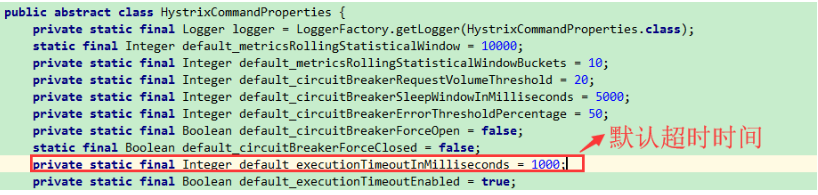




问题:如果连接超时的话

如product服务休眠两秒的话。Order服务将会一直是默认太拥挤的提示

由于客户端请求服务端方法时，服务端方法响应超过1秒将会触发降级，所以我们可以配置Hystrix默认的超时配置



@HystrixCommand(commandProperties = {  
 @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value = "3000")  
})  
@GetMapping("/getProductInfoList")  
public String getProductInfoList() {  
 RestTemplate restTemplate = new RestTemplate();  
 return restTemplate.postForObject("http://127.0.0.1:8086/queryProducts", Arrays.*asList*(1, 5, 8), String.class);

## 服务熔断

配置 ： <https://www.cnblogs.com/zhangwanhua/p/7742703.html>

意思为当某一个接口的出错概率达到一定的级别如50% 这项服务将会被熔断（理解为线路断了） 不被访问

隔一定时间 系统将会把线路连接变成半熔断状态 访问一次 如果好了。就提供服务。如果还是不行将继续熔断

自我保护

　hystrix.command.[commandkey].circuitBreaker.enabled 是否开启熔断机制，默认为true。

　　hystrix.command.[commandkey].circuitBreaker.forceOpen 强制开启熔断，默认为false。

　　hystrix.command.[commandkey].circuitBreaker.forceClosed 强制关闭熔断，默认为false。

　　hystrix.command.[commandkey].circuitBreaker.sleepWindowInMilliseconds  熔断窗口时间，默认为5s。

　　hystrix.command.[commandkey].circuitBreaker.requestVolumeThreshold 当在配置时间窗口内达到此数量后的失败，进行短路。默认20个

　　hystrix.command.[commandkey].circuitBreaker.errorThresholdPercentage 出错百分比阈值，当达到此阈值后，开始短路。默认50%

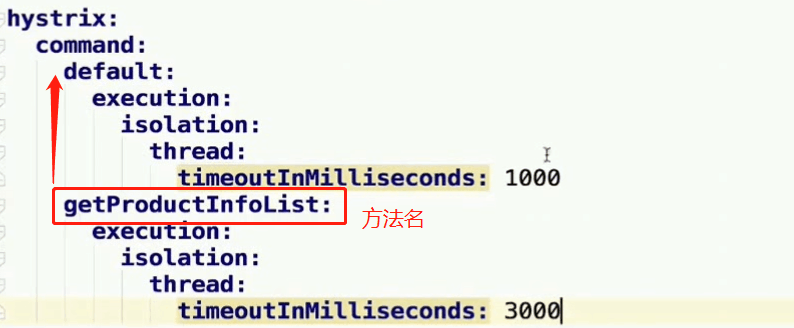


写入配置的方式 注解不可删除 @HystrixCommand

超时配置

**hystrix:  
 command:  
 default:  
 execution:  
 isolation:  
 thread:  
 timeoutInMilliseconds:** 3000

单独为某一个方法设置超时时间



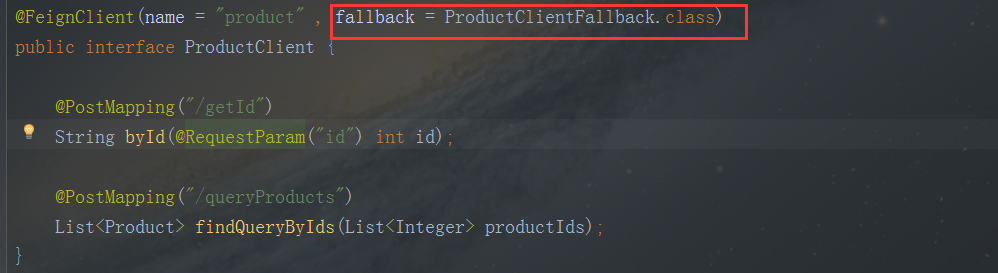
## Feign搭配hystrix

配置

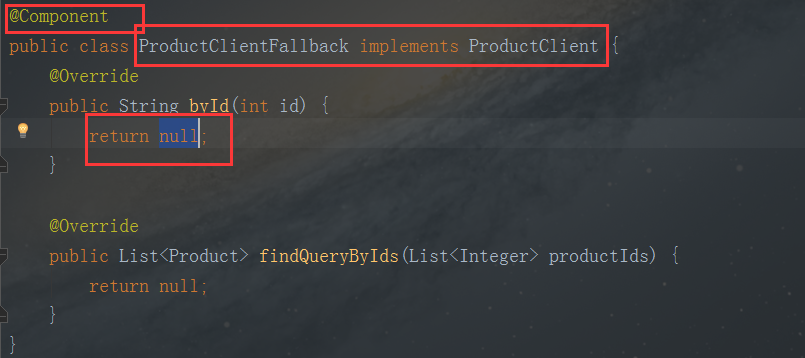
**feign:  
 hystrix:  
 enabled:** true

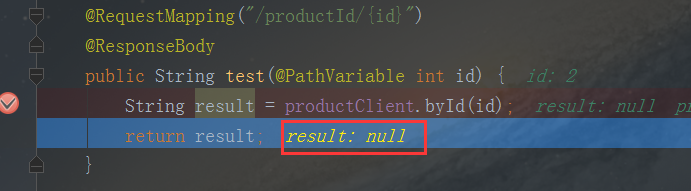
Zuul 8084 Order 8085 product 8086 zuul 8087 user 8088

将product服务关闭 调用相关product服务相关接口触发降级



创建类

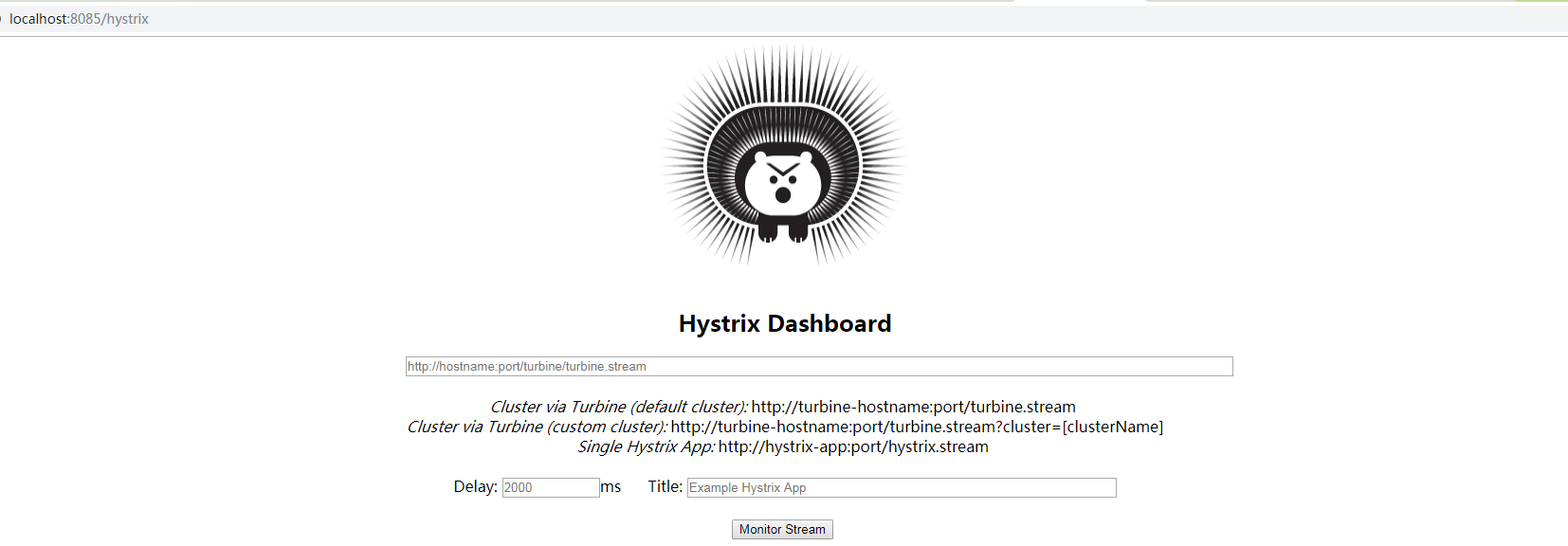


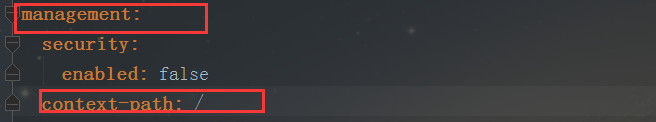


可视化界面

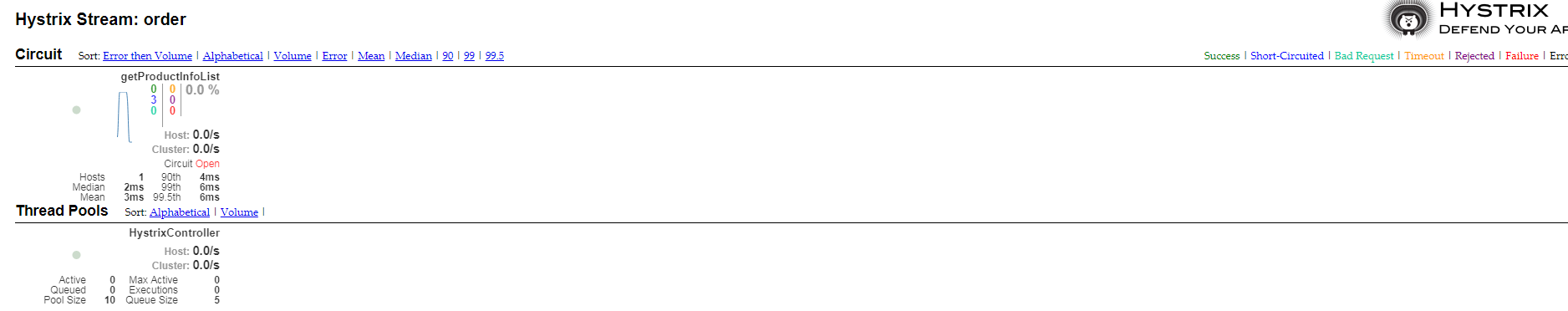
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>  
</dependency>

启动类添加@EnableHystrixDashboard





多次请求配置了熔断的方法



## 服务追踪Spring cloud sleuth

A服务调用b服务 b服务调用c服务 c服务调用D服务

耗时比较长 服务追踪差看那个服务耗时最长的利器

将order服务和product服务都添加依赖

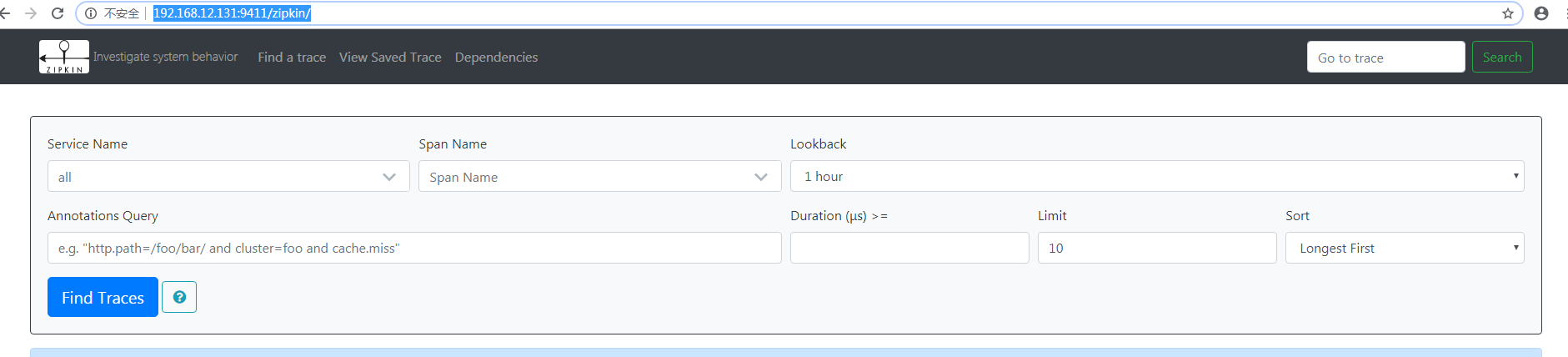
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-sleuth -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-sleuth</artifactId>  
 <version>1.3.0.RELEASE</version>  
</dependency>



Docker安装zipkin

docker run -d -p 9411:9411 openzipkin/zipkin

<http://192.168.12.131:9411/zipkin/>

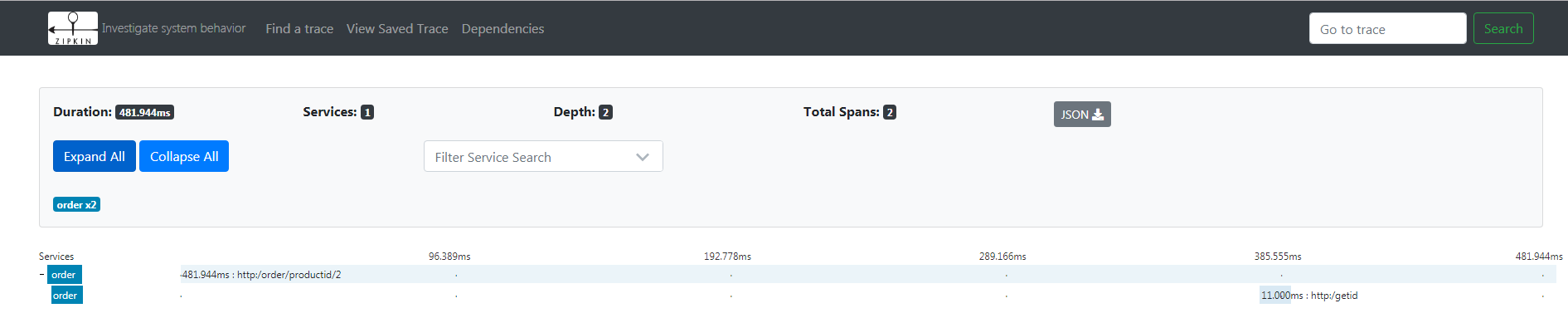


<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-sleuth-zipkin -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-sleuth-zipkin</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-zipkin</artifactId>  
</dependency>

**spring:**

**zipkin:  
 base-url:** http://192.168.12.131:9411  
**sleuth:  
 sampler:  
 percentage:** 1 抽样检测1为100%

请求： http://localhost:8085/order/productId/2



慢慢看耗时操作在那个服务