1 微服务下网关与智能路由的应用

1.1 Gateway服务网关概述

1. Gateway简介

Spring Cloud Gateway是Spring官方基于Spring 5.0，Spring Boot 2.0和Reactor等技术开发的网关

Spring Cloud Gateway作为Spring Cloud生态系中的网关，目标是替代Netflix ZUUL

> `Reactor`模式是一种典型的事件驱动的编程模型

Gateway解决了哪些问题

Gateway主要解决了路由转发、限流、熔断、权限功能。

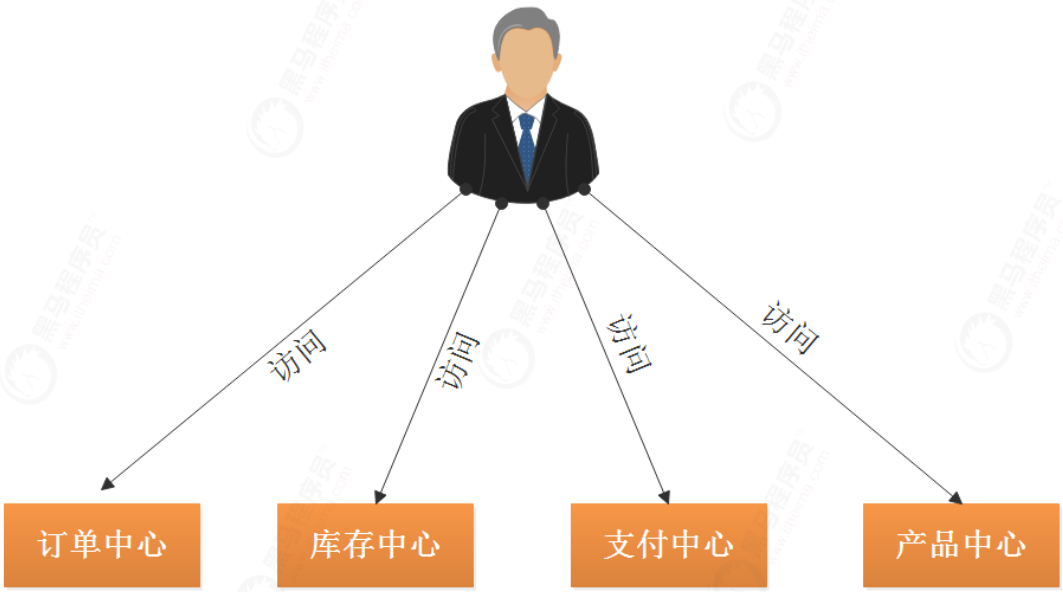
Gateway构建方式

- 代码实现(不推荐)
- 配置文件实现（推荐)

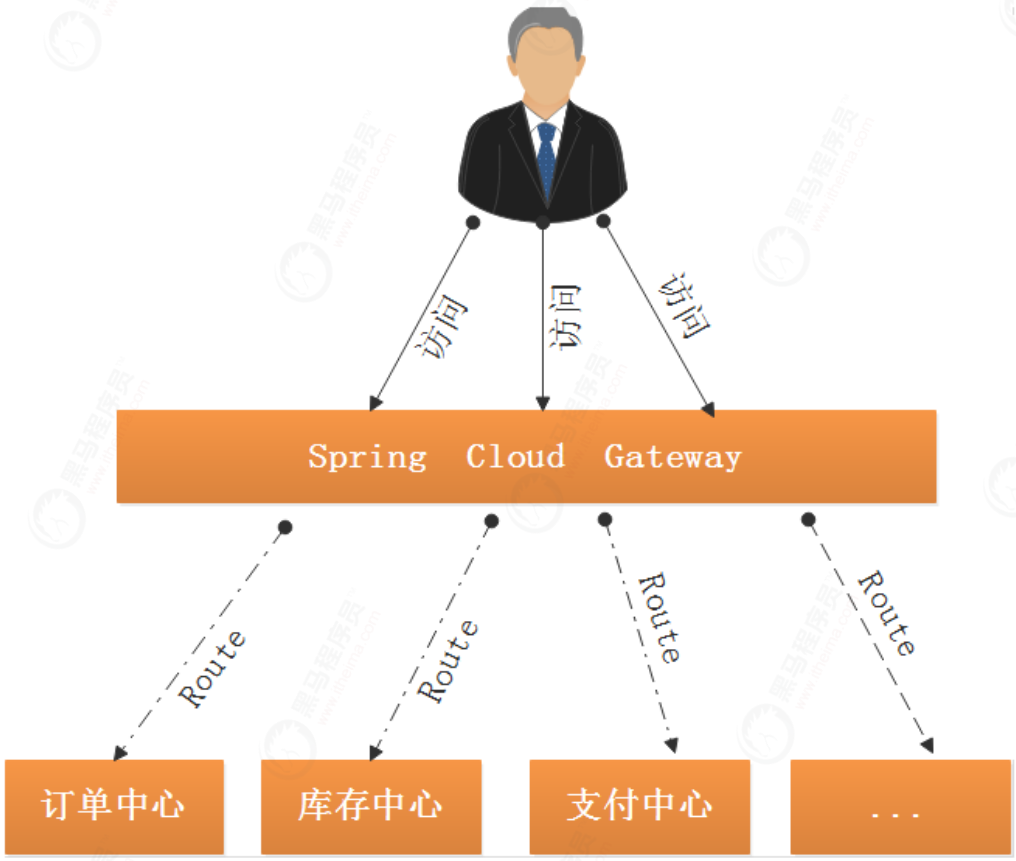
2. Gateway工作流程

思考：

不使用网关，直接访问各个微服务可以吗？



使用网关



1.2 Gateway核心概念介绍

1、核心概念

- Route （路由）

Route 是网关的基础元素，由 ID、目标 URI、断言、过滤器组成。

当请求到达网关时，由 Gateway Handler Mapping 通过断言进行路由匹配（Mapping），当断言为真时，匹配到路由。

- Predicate（断言）

Predicate 是 Java 8 中提供的一个函数。

它允许开发人员匹配来自 HTTP 的请求，简单来说它就是匹配条件。

- Filter （过滤器）

过滤器作为网关的其中一个重要功能，就是实现请求的鉴权。

Gateway自带过滤器有很多，常见自带过滤器有：

过滤器名称	说明
AddRequestHeader	对匹配上的请求加上Header
AddRequestParameters	对匹配上的请求路由
AddResponseHeader	对从网关返回的响应添加Header
StripPrefix	对匹配上的请求路径去除前缀
PrefixPath	对匹配上的请求路径添加前缀

网关使用场景：

- 请求鉴权：如果没有访问权限，直接进行拦截
- 异常处理：记录异常日志
- 服务调用时长统计

2. 过滤器配置

Gateway有两种过滤器

- 局部过滤器：只作用在当前配置的路由上。
- 全局过滤器：作用在所有路由上。

配置全局过滤器步骤(此处配置AddResponseHeader，具体参照上面的表格)：

1）修改application.yml文件

```
spring:
  cloud:
    gateway:
      # 配置全局默认过滤器
       filters:
       # 往响应过滤器中加入信息
         - AddResponseHeader=Hi,itheima
```

2) 浏览器输入http://localhost:8081/demo/hi，F12查看响应头信息

1.3 Gateway断言工厂介绍

1. After路由断言

After 路由可配置一个UTC时间格式的时间参数，当请求进来的当前时间在路由断言工厂之后会成功匹配，才交给 route去处理。

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: http://httpbin.org:80/get
        predicates:
        - After=2020-11-1T17:42:47.789-07:00[Asia/Shanghai]
```

2. Between路由断言

Between路由断言工厂接受两个参数，datetime1和datetime2。

匹配发生在datetime1与datetime2之间请求。

```
spring:
  cloud:
    gateway:
      routes:
      - id: between_route
        uri: http://httpbin.org/get
        predicates:
        - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2019-09-21T17:42:47.789-07:00[America/Denver]
```

3. Header路由断言

Header由断言工厂接受两个参数，头名称和正则表达式。

给定名称的头匹配，该值与正则表达式匹配。

```
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: http://httpbin.org/get
        predicates:
        - Header=X-Request-Id, \d+
```

4. Cookie路由断言

Cookie路由断言工厂接受两个参数：【Cookie=cookie名, cookie值的正则表达式规则】

cookie配置成功则进行路由

```
spring:
  cloud:
    gateway:
      routes:
      - id: cookie_route
        uri: http://httpbin.org:80/get
        predicates:
        - Cookie=username,[0-9]
```

5. Host 路由断言

Host路由断言工厂接受一个参数：需要一个参数即hostname。它可以使用. * 等去匹配host。

这个参数会匹配请求头中的host的值，一致，则请求正确转发。

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: http://httpbin.org:80/get
        predicates:
        - Host=**.hadron.cn
```

6. Method 路由断言

Method路由断言工厂接受一个参数：HTTP方法来匹配。

```
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: http://httpbin.org:80/post
        predicates:
        - Method=POST
```

7. Path 路由断言

Path路由断言工厂接受一个参数：采用Spring PathMatcher 模式

```
spring:
  cloud:
    gateway:
      routes:
      - id: path_route
        uri: lb://service-order
        predicates:
        - Path=/order/{segment}
```

8. Query路由断言

Query路由断言工厂接受两个参数：配置说明：【Query=参数名，参数值】

如果请求的参数包含name=zhangsan则成功路由

```
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: http://httpbin.org:80/get
        predicates:
        - Query=name, zhangsan
```

9. RemoteAddr路由断言

支持通过设置某个 ip 区间号段的请求才会路由

```
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: http://httpbin.org:80/get
        predicates:
        - RemoteAddr=10.17.36.1/24
```

组合使用

```
spring:
  cloud:
    gateway:
      routes:
        - id: blog
          uri: http://httpbin.org:80/get
          predicates:
            - Query=author, hengboy
            - Method=GET
            - Cookie=hengboy, yuqiyu
            - Header=X-Request-Id, \d+
            - RemoteAddr=192.168.1.56/24
```

`Gateway`内部提供了很多种灵活的路由转发规则，在同一个路由内存在多个`Predicate`时，同时满足规则后，请求才会被路由转发。

1.4 Gateway网关快速入门

1.4.1 Gateway智能路由

本章节场景：

我们将建立三个Module，父项目是itheima-service-gateway，里面没有任何代码，它用来管理我们的聚合工程。

```
    <modules>
        <module>service-order</module>
        <module>service-payment</module>
        <module>service-gateway</module>
    </modules>
```

具体场景：

将service-order、service-payment、service-gateway注册到Nacos，通过service-gateway网关服务调用

1.创建Gateway模块

创建service-gateway模块，端口8081，作为服务网关

2. 引入起步依赖

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
        </dependency>
```

3. 修改application.yml文件

```yml
server:
  port: 8081

spring:
  application:
    name: service-gateway
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
    gateway:
      default-filters:
      #设置全局的默认过滤器（请求头、响应头、header等，但不会覆盖局部过滤器，并存）
```

```yaml
        - AddResponseHeader=czbk,welcome
      routes:
      - id: service-order
        uri: lb://service-order
        predicates:
        - Path=/czbk/v1/order/**
        filters:
        - AddResponseHeader=czbk,order
        - StripPrefix=1
      - id: service-payment
        uri: lb://service-payment
        predicates:
        - Path=/czbk/v1/pay/**
        filters:
        - AddResponseHeader=czbk,payment
        - StripPrefix=1
```

> tips
>
> AddResponseHeader=czbk,order
>
> 写成
>
> AddResponseHeader=czbk-order
>
> 会报错

4. 启动入口类

```java
package com.itheima;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class ServiceGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServiceGatewayApplication.class, args);
    }


}
```

5. 启动

分别启动service-order、service-payment、service-gateway注册到Nacos

6. 访问

1) 不使用网关访问service-order服务

```
http://localhost:8762/v1/order/create
```

如下图

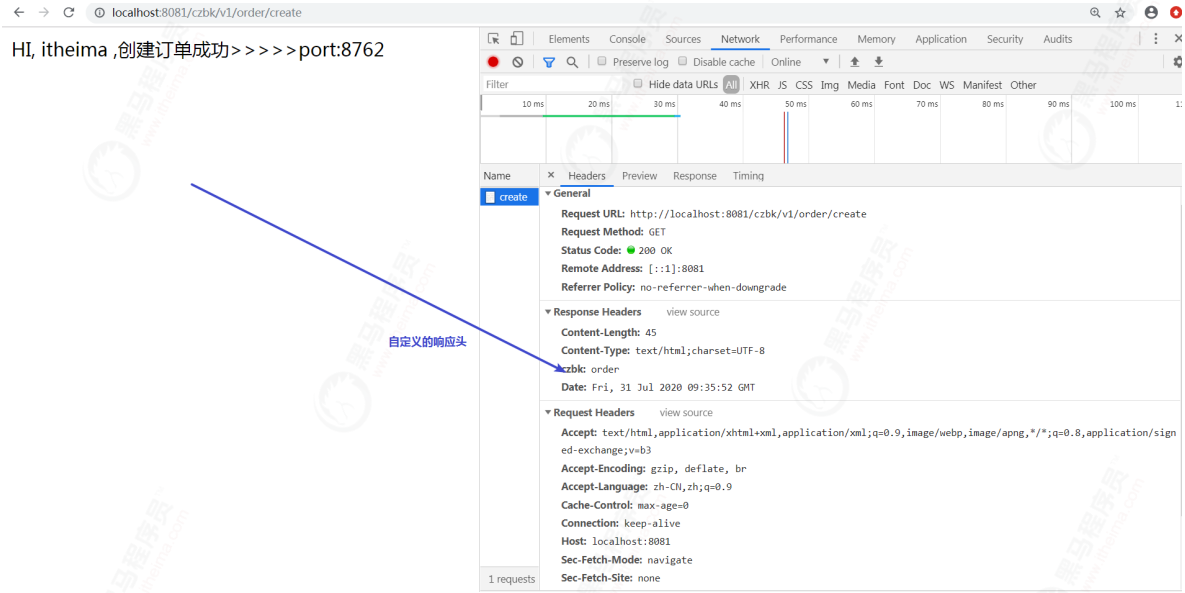
HI, itheima ,创建订单成功>>>>>port:8762

2) 使用网关访问service-order服务

```
http://localhost:8081/czbk/v1/order/create
```

浏览器输入如下图

HI, itheima ,创建订单成功>>>>>port:8762



由此可见，我们使用http://localhost:8081/czbk/v1/order/create访问网关，也同样输出了HI, itheima ,创建订单成功>>>>>port:8762

那是因为我们的网关服务service-gateway配置文件application.yml文件中配置了路由转发功能；核心的参数为

```
uri: lb://service-order  - Path=/czbk/v1/order/**
```

url为指向注册中心的服务，主要是从我们的注册中心获取服务，注册中心可以是多种类型Consul或者Zookeeper

Path为转发路径

在上面的配置中，配置了一个Path 的predict,将以/czbk/v1/order/**开头的请求都会转发到uri: lb://service-order 的地址上， lb://service-order即service-order服务的负载均衡地址，并用StripPrefix的filter 在转发之前将/czbk去掉

支付模块，网关访问

```
http://localhost:8081/czbk/v1/pay/payment
```

1.4.2 Gateway过滤器

spring cloud Gateway网关过滤器类型主要分：全局和局部过滤器

1、全局过滤器

- **GlobalFilter**：全局过滤器，对所有的路由均起作用无需任何配置文件配置就可以起到过滤器的作用
- 全局默认过滤器

过滤器名称	说明
AddRequestHeader	对匹配上的请求加上Header
AddRequestParameters	对匹配上的请求路由
AddResponseHeader	对从网关返回的响应添加Header
StripPrefix	对匹配上的请求路径去除前缀
PrefixPath	对匹配上的请求路径添加前缀

配置方式

```
      default-filters:
      #设置全局的默认过滤器（请求头、响应头、header等，但不会覆盖局部过滤器，并存）
        - AddResponseHeader=czbk,welcome
```

2、局部过滤器

GatewayFilter：只对指定的路由起作用

一种是直接 实现GatewayFilter, Ordered接口，另一种是 继承AbstractGatewayFilterFactory类

可以通过下面的方式配置

注意：

下面的 - RequestTime=true的RequestTime

对应它的类名RequestTimeGatewayFilterFactory

```
      routes:   #路由
      - id: service-order
        uri: lb://SERVICE-order
        predicates:
          - Path=/order/**
        filters:
          - RequestTime=true
```

全局过滤器代码

```
 package com.itheima.filter;


import org.apache.commons.lang.StringUtils;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Configuration
public class AuthTokenFilter implements GlobalFilter, Ordered {
    private final static Log log = LogFactory.getLog(AuthTokenFilter.class);
    private static final String TOKEN = "token";
```

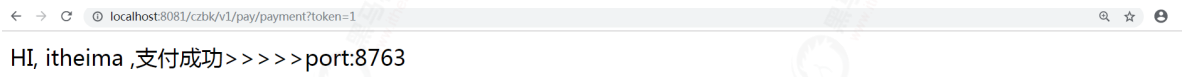
```
    /*
     * @Description: 小----->高
     * @Method: getOrder
     * @Param: []
     * @Update:
     * @since: 1.0.0
     * @Return: int
     *
     */
    @Override
    public int getOrder() {
        return 0;
    }


    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        String token = exchange.getRequest().getQueryParams().getFirst(TOKEN);
        if (StringUtils.isEmpty(token)) {
            log.error("非法访问>>>>>>>>");
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();

        }
        return chain.filter(exchange);


    }

}
```

不加token访问

```
http://localhost:8081/czbk/v1/order/create
http://localhost:8081/czbk/v1/pay/payment
```

加入token

```
http://localhost:8081/czbk/v1/pay/payment?token=1
http://localhost:8081/czbk/v1/order/create?token=1
```



> 总结
>
> 过滤器分类
>
> 1、全局过滤器；
>
> 全局自定义过滤器：实现接口GlobalFilter；不用配置
>
> 全局默认过滤器：默认使用default-filters设置全局默认过滤器
>
> 2、局部过滤器；使用下面的方式指定；注意命名格式； RequestTime=true的RequestTime需要切割
>
> filters:
> - RequestTime=true
>
> 3.自定义全局过滤器，只需要配置bean即可，不需要任何配置即可生效

2 基于消息总线服务聚合与广播实现

```
引言
在学习消息总线前
我们先回顾一下前面学习的内容，分布式配置中心解决方案spring cloud  confg
在那个章节，我们遗留下来一个问题
当我们的配置（提交到github的配置）之后，
进行修改，如果下面的微服务想要拿到最新的值
我们就需要通过actuator/refresh进行刷新（每个客户端都要刷新，
100个微服务，刷新100次）才可以获取到最新的值


在分布式场景中，如果我们的服务非常多，几十个、甚至上百个都是有可能的，
如果我们每修改一个配置文件，都要手工去一个一个去刷新，
这样工作量会非常非常大，在分布式场景中显然这种方式是不可取的。

那么，有没有一种方式，刷新一次，所有的微服务全部获取到最新的值呢？

答案是有的
就是通过我们本章学习的消息总线来实现（RabbltMQ+BUS）
```

```
实现一次改变、所有微服务都可以刷新
```

2.1 Bus消息总线概述

1. Bus消息总线简介

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。

关键的思想就是，消息总线可以为微服务做监控，也可以实现应用程序之间相通信。

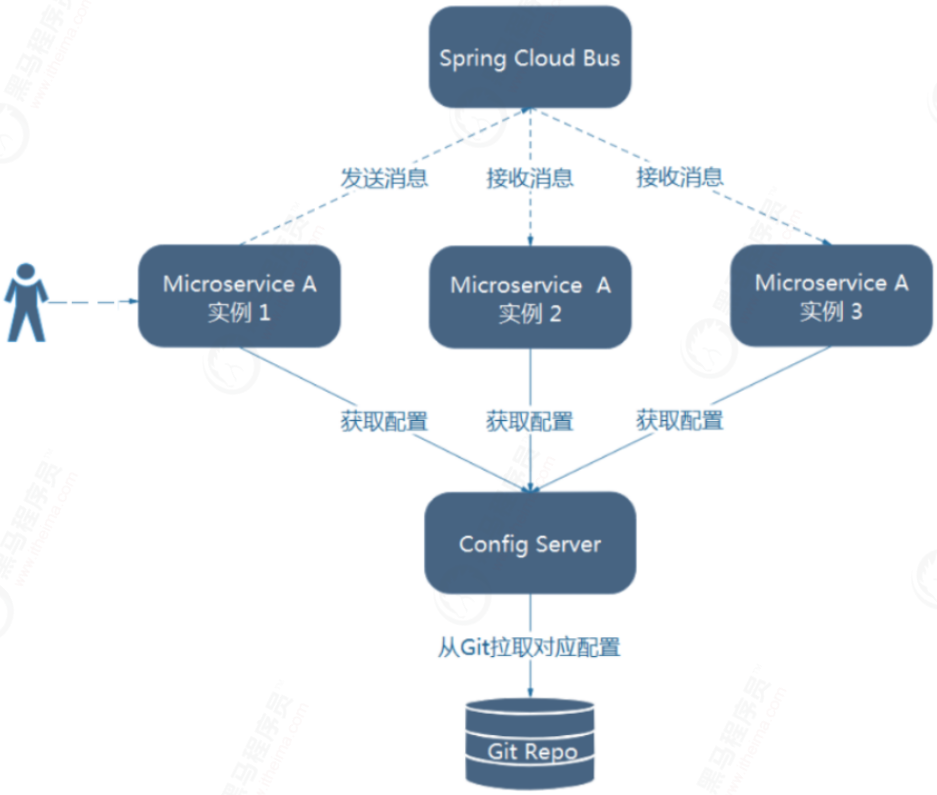
Spring Cloud Bus 可选的消息代理组建包括RabbitMQ / Kafka

2.2 Bus消息总线工作原理

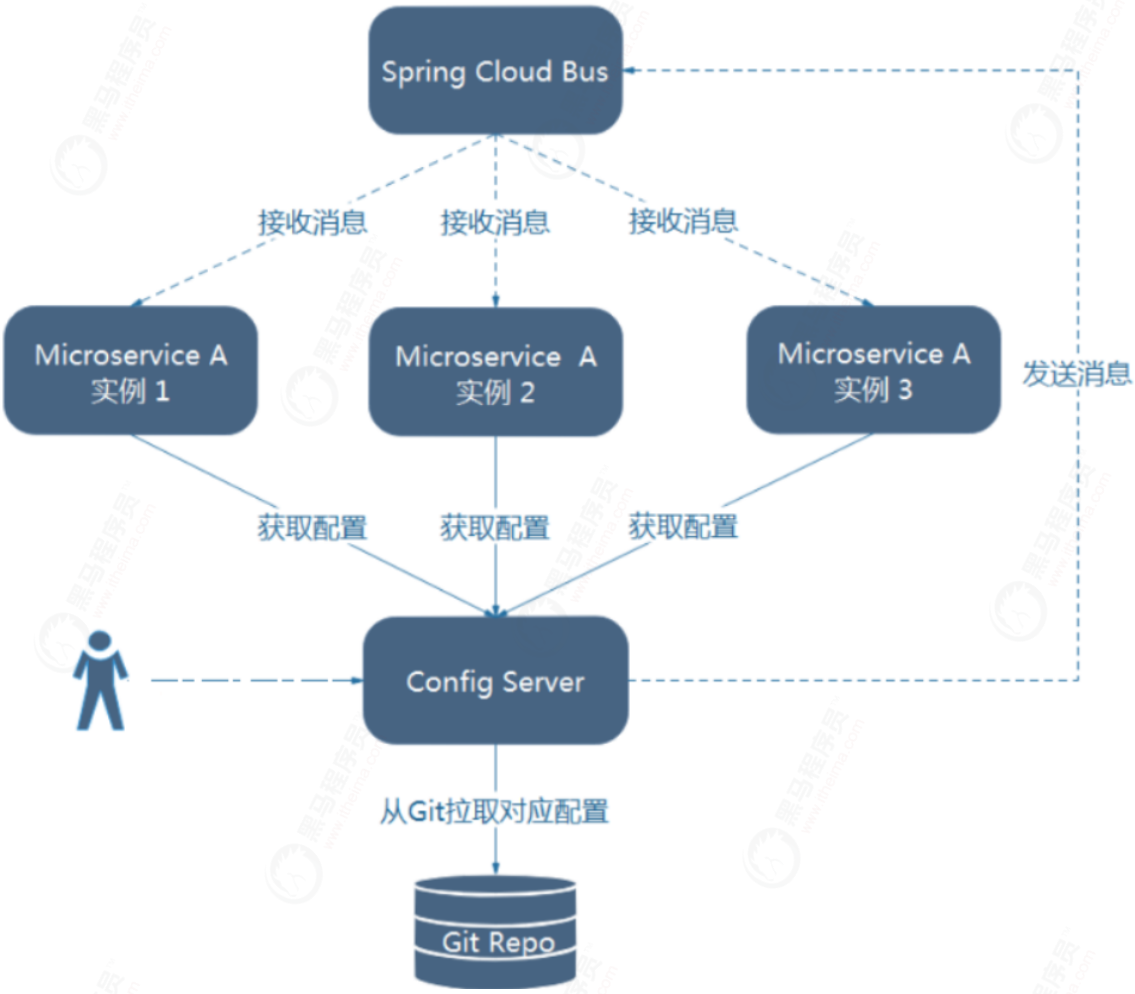
基于Bus总线分布式配置中心刷新模式（两种）

Spring cloud bus通过轻量消息代理连接各个分布的节点，用在广播状态的变化（例如配置变化）或者其他的消息指令，本质是利用了MQ的广播机制在分布式的系统中传播消息，目前常用的有Kafka和RabbitMQ。下面说下使用bus后的两种架构，来看看两者之间的不同：

1) 客户端刷新：


Spring Cloud Bus
发送消息
接收消息
接收消息
Microservice A
实例 1
Microservice A
实例 2
Microservice A
实例 3
获取配置
获取配置
获取配置
Config Server
从Git拉取对应配置
Git Repo

2) 服务端刷新



将在快速入门中详细介绍下这两种方式的代码实现。

2.3 Bus聚合与广播快速入门

2.3.1 基础环境介绍与构建

模块介绍

```
<modules>
    <module>config-client</module>
    <module>config-server</module>
    <module>config-client-git</module>
    <module>config-client-git-backup</module>
    <module>config-server-git</module>
</modules>
```

本章节场景：

由于在配置中心章节无法解决远程仓库文件发生改变，N个微服务同时获取最新值

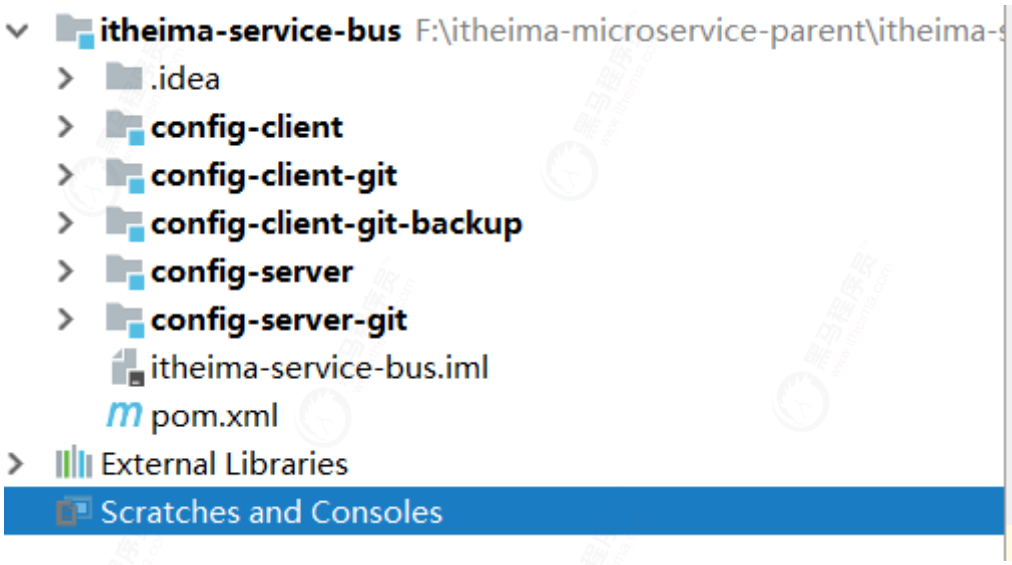
所以当前章节将借助RabbitMQ+Spring Cloud Bus实现

首先启动

config-client-git配置中心客户端。用于远程仓库码云读取
config-client-git-backup配置中心客户端。用于远程仓库码云读取
config-server-git配置中心客户端。用于配置远程仓库码云

然后去码云远程仓库修改配置文件

然后执行【客户端刷新】或者【服务端刷新】，这样的话config-client-git-backup、config-client-git都可以获取到最新的值，而不是每个微服务都要执行一次，即使有1000个微服务，执行【客户端刷新】或者【服务端刷新】，这个100微服务都可以刷新。



1. 引入mq起步依赖

修改之前的**config-client-git**、config-client-git-backup、config-server-git**的pom文件，增加mq起步依赖

之所以修改上面的两个工程，主要实现客户端或者服务端刷新

spring-cloud-starter-bus-amqp消息总线启动器（包含了rabbit，spring-cloud-starter-stream-rabbit）

```
      <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-bus-amqp</artifactId>
        </dependency>
      <dependency>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. 修改bootstrap.yml

修改之前的**config-client-git**的bootstrap.yml文件，加入

```
    bus:
      enabled: true #开启spring  cloud bus 服务
      trace:
        enabled: true #必须设置
```

```
  rabbitmq:
    host: localhost  #配置RabbitMQ的IP地址
    port: 5672 #配置RabbitMQ的端口
    username: guest #配置RabbitMQ的用户名
    password: guest #配置RabbitMQ的密码
management:
  endpoints:
    web:
      exposure:
        include: '*' #开启刷新端点       # bus-refresh 必须设置，才能在postman中访问
bus-refresh PATH
```

汇总如下（backup自行配置）

```
spring:
  application:
    name: config-client-git
  cloud:
    bus:
   #true 开启spring  cloud bus 服务
      enabled: true
      trace:
        enabled: true #true 必须设置
    config:
      uri: http://localhost:8769
      #spring.cloud.config.fail-fast 表示如果没有读取成功，则执行快速失败
      fail-fast: true
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
  profiles:
  #表示读取dev环境的配置文件，对应git的config-client-git-dev.yml
    active: dev
  rabbitmq:
    host: localhost  #配置RabbitMQ的IP地址
    port: 5672 #配置RabbitMQ的端口
    username: guest #配置RabbitMQ的用户名
    password: guest #配置RabbitMQ的密码
server:
  port: 8762
management:
  endpoints:
    web:
      exposure:
        include: '*' #开启刷新端点       # bus-refresh 必须设置，才能在postman中访问
bus-refresh PATH
```

3. 修改之前的config-server-git的application.yml文件，加入

```
server:
  port: 8769
```

```
# 从git 仓库读取配置文件的配置项
# GIT目录已经上传config-client-dev.properties文件 该文件没有指定端口 只配置了itheima变量
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139:8848
    config:
      server:
        git:
          #下面的git为码云地址
          uri: https://gitee.com/code80341157/itheima-repository.git
          #配置搜索远程仓库的文件夹地址
          searchPaths: itheima
          #公有仓库不需要用户名和密码，私有仓库是需要的
          username:
          password:
          #为git仓库的分支名，本例从master读取
      label: master
  application:
    name: config-server-git
  rabbitmq:
    host: localhost  #配置RabbitMQ的IP地址
    port: 5672 #配置RabbitMQ的端口
    username: guest #配置RabbitMQ的用户名
    password: guest #配置RabbitMQ的密码
    #publisher-confirms: true
    #virtual-host: /
management:
  endpoints:
    web:
      exposure:
        include: '*' #开启刷新端点       # bus-refresh 必须设置，才能在postman中访问
bus-refresh PATH
```

4. 读取配置文件

在ConfigClientApplication类中写一个API接口，读取配置文件itheima变量，并通过API返回

```
@SpringBootApplication
@RestController
@EnableEurekaClient
@RefreshScope
public class ConfigClientApplication {

    @Value("${itheima}")
    String itheima;

    public static void main(String[] args) {
        SpringApplication.run(ConfigClientApplication.class, args);
    }

    @RequestMapping(value = "/itheima")
    public String hi() {
```

```
        return itheima;
    }
}
```

5. 启动

启动config-server-git、config-client-git、config-client-git-backup,如下图：

RabbitMQ地址

RabbitMQ安装完成后，在开始菜单右键选择【RabbitMQ Service - start】，**然后以管理员身份运行**

（常见问题）无法访问？

```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.7\sbin
```

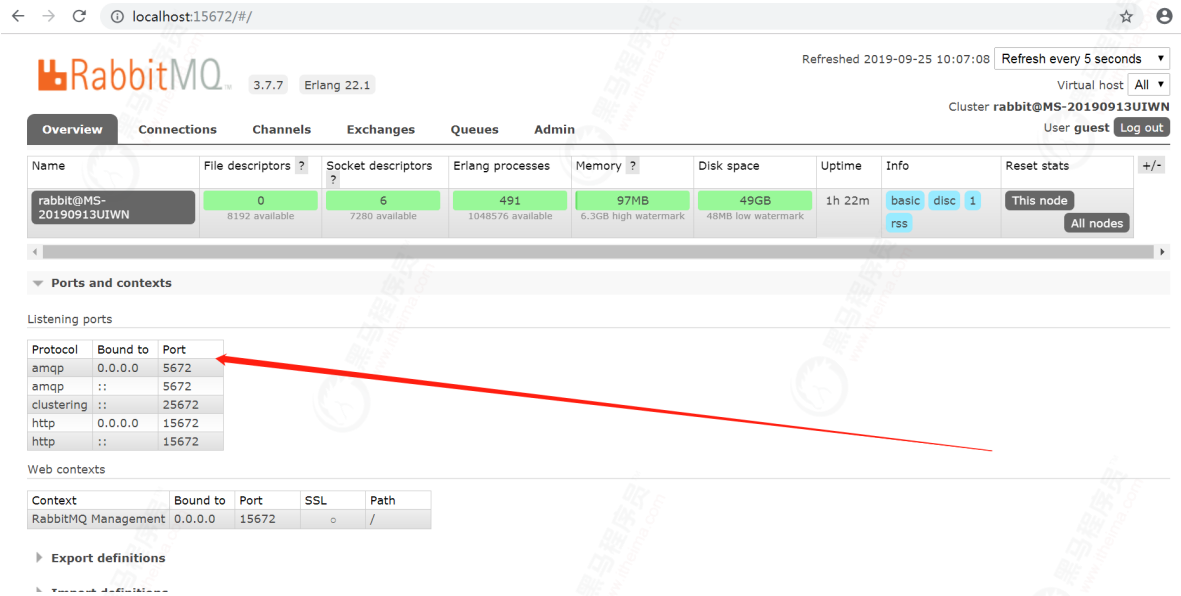
在当前目录打开CMD，执行下面的命令

```
# 开启RabbitMQ节点
rabbitmqctl start_app
# 开启RabbitMQ管理模块的插件，并配置到RabbitMQ节点上
rabbitmq-plugins enable rabbitmq_management
# 关闭RabbitMQ节点
rabbitmqctl stop
```

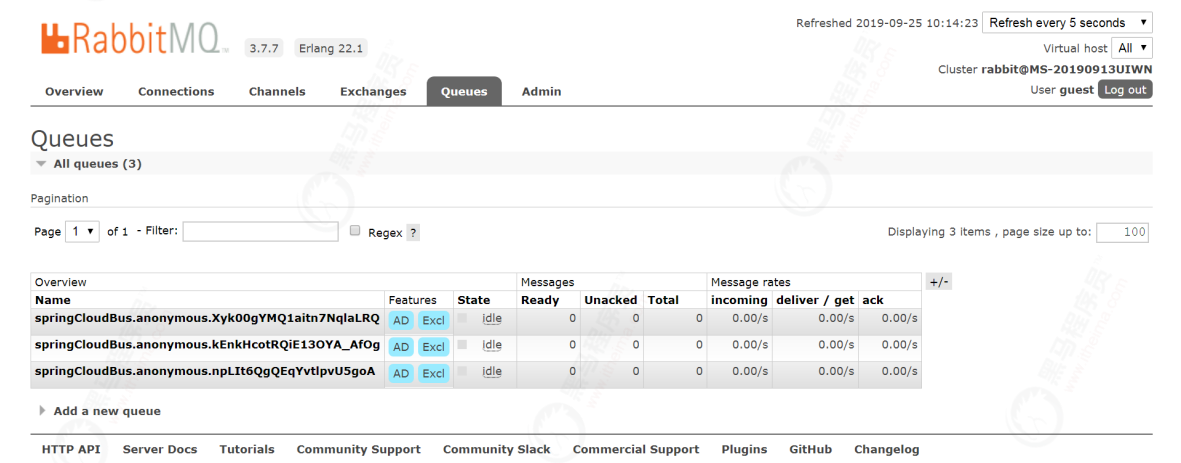
此时，RabbitMQ管理模块的插件已经配置到RabbitMQ节点上。
再次回到浏览器中访问 http://localhost:15672 即可成功打开RabbitMQ管理界面

```
http://localhost:15672/
```

以用户名/密码（guest / guest）登录RabbitMQ之后的界面如下



5672是rabbitmq 默认TCP监听端口，箭头方向指的是使用当前端口监听我们的configserver



上图是RabbitMQ的web页面，为我们生成的Queues队列

config-server-git、config-client-git、config-client-git-backup对应的三个队列

6. 设置码云

1) 浏览器输入码云远程git仓库地址

```
https://gitee.com/code80341157/itheima-repository/tree/master/itheima
```

如下图

将config-client-git-dev.yml里面itheima的值修改成itheima config server Version 5.0.1，如下图

然后点击保存即可。

2.3.2 消息广播之Client刷新

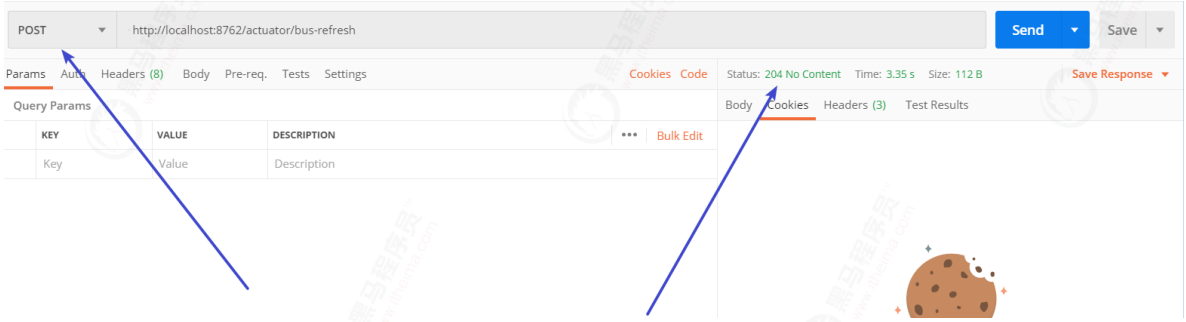
1) 浏览器输入

```
http://localhost:8762/itheima

和

http://localhost:8763/itheima
```

此时发现里面的值还不是最新的，还是之前最早的itheima config server Version 4.0.0

2) 通过postman工具(或者curl)发送post请求到：

```
http://localhost:8762/actuator/bus-refresh
```




```
C:\Users\Administrator>curl -X POST  http://localhost:8762/actuator/bus-refresh

C:\Users\Administrator>_
```

3 然后访问

发现两个微服务都拿到了最新的配置

```
http://localhost:8262/itheima
和
http://localhost:8763/itheima
```

此时发现，我们只是在cmd curl 中执行了curl -X POST http://localhost:8762/actuator/bus-refresh

操作，并没有执行端口号为8763的curl操作，而且8762和8763都可以获取到了最新的值itheima config server Version 5.0.1

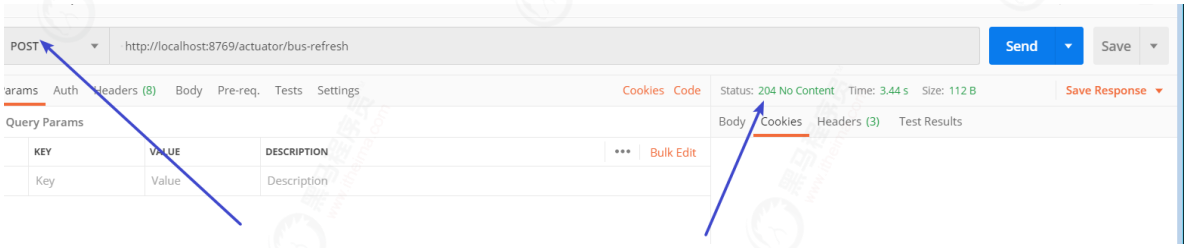
这样就能很好的给我们解决在大规模的分布式场景中，比如有500个微服务，这个500个微服务都读取了key为itheima的配置，如果配置文件发生了变化，我们只需要通过bus+rabbitmq的方式执行一次，这500个微服务都可以获取到最新的值，而不是手工执行500次操作，大大提高开发和维护的效率

2.3.3 消息广播之Server刷新

只post服务端，所有微服务都可以拿到最新配置

```
http://localhost:8769/actuator/bus-refresh
```

如下图：




```
C:\Users\Administrator>curl -X POST http://localhost:8769/actuator/bus-refresh

C:\Users\Administrator>_
```

服务端刷新、客户端刷新我们都介绍完了，在实际应用中大家可以采用自己喜欢的方式

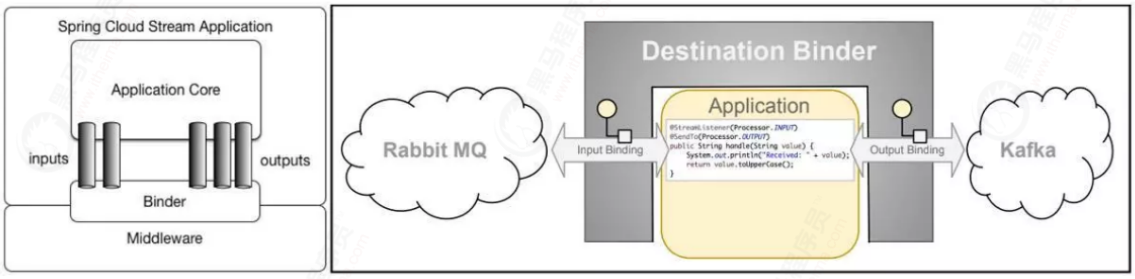
3 微服务下消息复杂性应对之道

3.1 Stream消息驱动概述

Spring Cloud Stream是一个框架，用于构建连接到共享消息传递系统的高度可伸缩的事件驱动微服务。

该框架提供了一个灵活的编程模型，它建立在已经建立和熟悉的Spring习惯用法和最佳实践之上，包括对持久发布/子语义、消费者组和有状态分区的支持

3.2 Stream工作原理



编程模型

- **Destination Binders（目的地绑定器）**： 负责与外部消息系统集成交互的组件**（可以理解成绑定的消息中间件）**
- **Destination Bindings（目的地绑定）**： 在外部消息系统和应用的生产者和消费者之间的桥梁**（可以理解成交换机或者topic）**
- **OUT：输出通道**
- **INPUT：输入通道**

3.3 Stream快速入门

本章节场景：

我们将建立三个个Module，父项目是itheima-service-stream，里面没有任何代码，它用来管理我们的聚合工程。

本章节将采取RabbitMQ+Stream实现消息的发送和消息的接收以及如何处理重复消费

```
<modules>
    <module>stream-consumer-rabbitmq</module>
    <module>stream-producer-rabbitmq</module>
    <module>stream-consumer-rabbitmq-backup</module>
</modules>
```

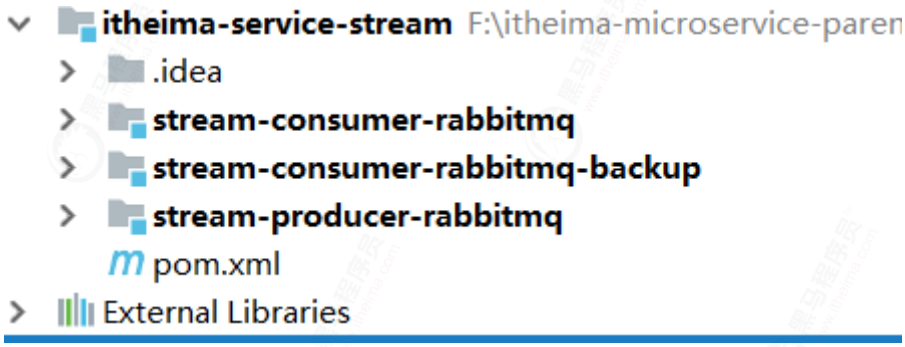
具体场景

stream-producer-rabbitmq生产消息，将消息发送到RabbitMQ

stream-consumer-rabbitmq去RabbitMQ上面消费消息

为了验证重复消费，我们又加了个Module 模块stream-consumer-rabbitmq-backup去验证重复消费

本章节代码 工程结构如下图：



3.3.1 创建生产者

创建模块stream-producer-rabbitmq（消息生产者），作为Stream的生产者，主要作用是应用程序通过管道向RabbitMQ发送消息

1. 引入起步依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

`Stream`依赖RabbitMQ、绑定器，所以需要引入spring-cloud-starter-stream-rabbit起步依赖

2. 修改application.yml文件

```
#客户端连接的地址
spring.rabbitmq.addresses=amqp://127.0.0.1:5672
#rabbit用户名
spring.rabbitmq.username=guest
#rabbit密码
spring.rabbitmq.password=guest
#目的地绑定器,这个producerChannel表示输入,destination：指定了消息获取的目的地，对应于MQ就是 exchange，这里的exchange就是itheimaBind
spring.cloud.stream.bindings.producerChannel.destination =itheimaBind
#默认情况下同一个队列的只能被同一个group的消费者消费
#spring.cloud.stream.bindings.producerChannel.group = itheimaBind-group
server.port=8080
spring.cloud.nacos.discovery.server-addr=192.168.23.139:8848
spring.application.name=stream-producer-rabbitmq
```

3. 创建通道绑定（Output）

消息从发布者传递到队列的整个过程是通过通道完成的。

因此，我们创建一个MessageChannelOutputBinding接口，其中包含我们的消息通道 `producerChannel`：

```
package com.itheima.steam.rabbit;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

/**
* @Class: MessageProducer 生产者自定义通道
* @Package com.itheima.steam.rabbit
* @Description:
* @Company: http://www.itheima.com/
*/

public interface MessageProducer {

    @Output("producerChannel")
    MessageChannel  producerChannel();
}
```

4. 消息推送（Output）

创建rest接口，将消息发送到RabbitMQ

```
package com.itheima.steam.rabbit;

import org.springframework.messaging.Message;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;

/**
 * @Class: ProducerController
```

```
 * @Package com.itheima.steam.rabbit
 * @Description: 生产消息
 * @Company: http://www.itheima.com/
 */
@RestController
public class ProducerController {
    @Resource
    private MessageProducer messageProducer;

    @GetMapping("publish/{name}")
    public void publishMessage(@PathVariable String name) {
        String message = "Hello" + name;
        Message<String> msg = MessageBuilder.withPayload(message).build();
        messageProducer.producerChannel().send(msg);
    }
}
```

5. 生产者入口

生产者的入口程序

```
package com.itheima;

import com.itheima.steam.rabbit.MessageProducer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.stream.annotation.EnableBinding;

@EnableBinding(MessageProducer.class)
@SpringBootApplication
@EnableDiscoveryClient
public class ProducerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class, args);
    }
}
```

3.3.2 创建消费者

创建模块stream-consumer-rabbitmq，作为Stream的消费者，Stream将通过管道订阅RabbitMQ上面的数据

1. 引入起步依赖

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
        </dependency>
```

2. 修改application.yml文件

```
#客户端连接的地址，有多个的时候使用逗号分隔，该地址可以是IP与Port的结合
spring.rabbitmq.addresses=amqp://127.0.0.1:5672
#rabbit用户名
spring.rabbitmq.username=guest
#rabbit密码
spring.rabbitmq.password=guest
#目的地绑定器（基础信息生产）
spring.cloud.stream.bindings.producerChannel.destination=itheimaBind
#默认情况下同一个队列的只能被同一个group的消费者消费
spring.cloud.stream.bindings.producerChannel.group=itheimaBind-group
server.port=9090
spring.cloud.nacos.discovery.server-addr=192.168.23.139:8848
spring.application.name=stream-consumer-rabbitmq
```

3. 创建通道绑定（Input）

需要监听之前创建的通道producerChannel为它创建一个绑定

与生产者绑定的两个非常明显区别。因为我们正在消费消息，所以我们使用`SubscribableChannel`和`@Input`注解连接到producerChannel

```
package com.itheima.stream.rabbit;

import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

/**
* @Class: MessageConsumer
* @Package com.itheima.stream.rabbit
* @Description: 消费者
* @Company: http://www.itheima.com/
*/

public interface MessageConsumer {
    @Input("producerChannel")
    SubscribableChannel  producerChannel();
}
```

4. 消息消费

利用元注解StreamListener参数为MessageChannelInputBinding，用SubscribableChannel和@Input注解连接到producerChannel（RabbitMQ）,在RabbitMQ上进行消费

```
package com.itheima.stream.rabbit;

import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;

/**
 * 添加@EnableBinding启用了MessageChannelInputBinding
 */
@EnableBinding(MessageChannelInputBinding.class)
public class MessageConsumerListener {
    /**
     * StreamListener可以接收处理流的事件
```

```
     * 从mq消费消息，然后消息输出
     *
     * @param msg
     */
    @StreamListener(target = MessageChannelInputBinding.GREETING)
    public void processHelloChannelGreeting(String msg) {
        System.out.println("Received: "+msg);
    }
}
```

5. 消费者入口

消息消费的入口程序

```
package com.itheima.stream.rabbit;

import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;

/**
* @Class: MessageConsumerListener
* @Package com.itheima.stream.rabbit
* @Description:监听、绑定
* @Company: http://www.itheima.com/
*/
@EnableBinding(MessageConsumer.class)
public class MessageConsumerListener {


    @StreamListener(target ="producerChannel" )
    public  void  getMessage(String msg){
        System.out.println(">>>>>>>>>>>>>>>>>>>>"+msg);
    }
}
```

6. 启动

注意：stream-consumer-rabbitmq-backup工程：复制stream-consumer-rabbitmq，修改下application.yml里面的server.port为其他端口即可，创建过程不再赘述

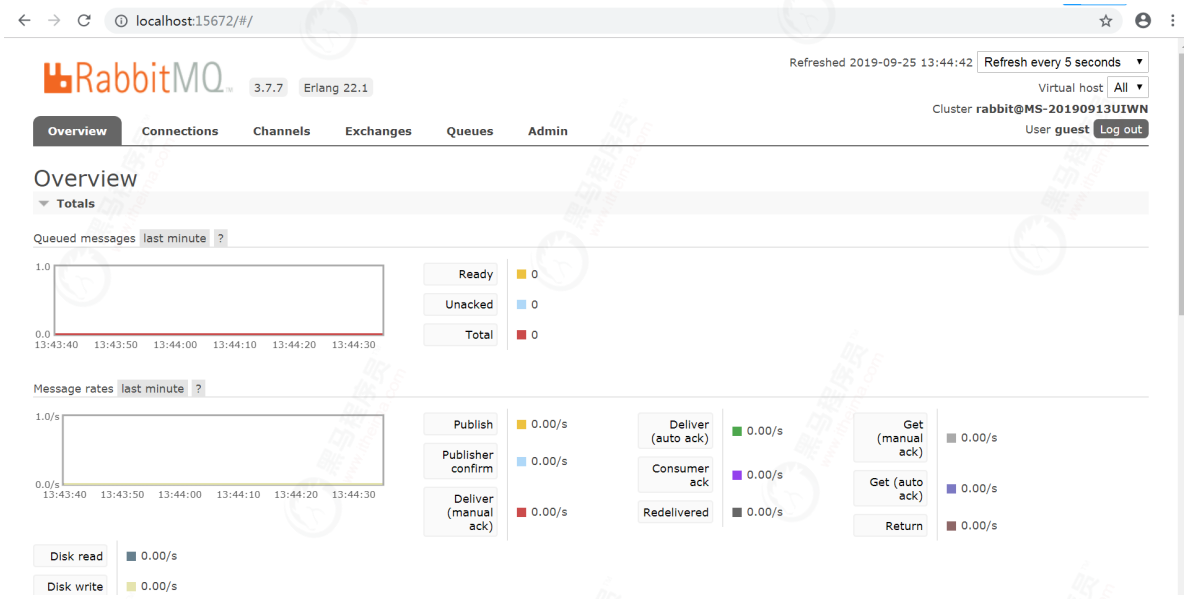
启动stream-producer-rabbitmq、stream-consumer-rabbitmq、stream-consumer-rabbitmq-backup、RabbitMQ

7. 访问

1）访问RabbitMQ浏览器输入

```
http://localhost:15672
```

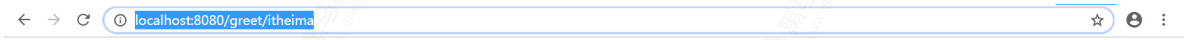
确保可以正常访问，如下图：



2）访问生产者 `REST` 端点生产消息（将消息发送到RabbitMQ），浏览器输入

```
http://localhost:8080/publish/itheima
```

生产者向RabbitMQ交换器、队列发送消息 【itheima】



查看IDEA的效果

当我们点击生产者`REST`端点生产消息时，我们看到两个消费者(stream-consumer-rabbitmq、stream-consumer-rabbitmq-backup)都收到了消息，这可能是我们在一些用例中想要的。但是，如果我们只想让一个消费者消费一条消息呢?为此，我们需要在`application.properties`中创建一个消费者组。增加消费者的配置文件:

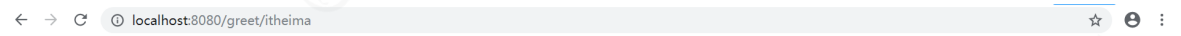
```
#默认情况下同一个队列的只能被同一个group的消费者消费
spring.cloud.stream.bindings.producerChannel.group=itheimaBind-group
#需要注意上述配置中producerChannel是在代码中@Output和@Input中传入的名字
```

现在，我们在重启stream-consumer-rabbitmq、stream-consumer-rabbitmq-backup工程

接着访问生产者`REST`端点生产消息，浏览器输入

```
http://localhost:8080/publish/itheima
```

生产者向RabbitMQ交换器、队列发送消息 【itheima】

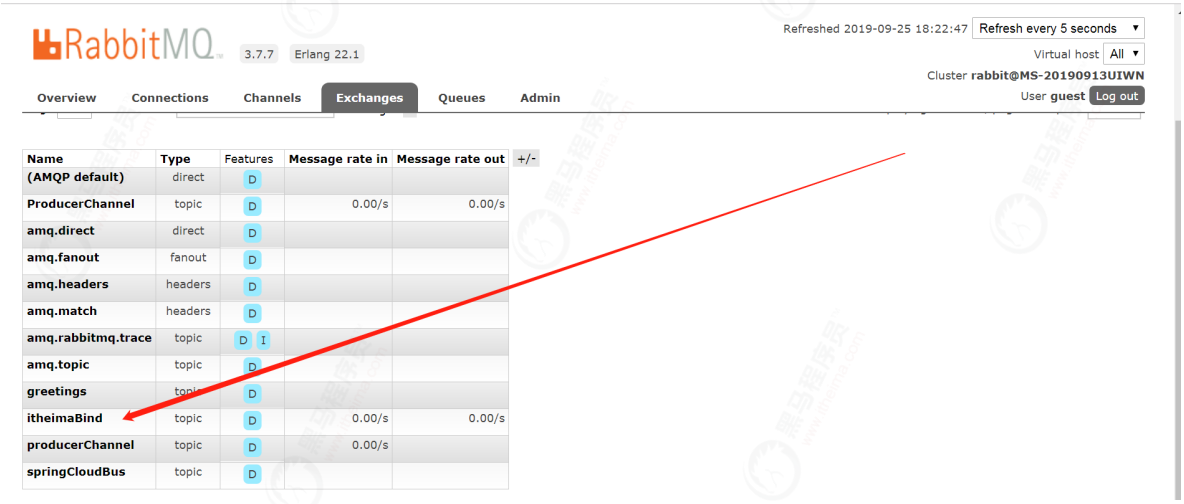


可以再次查看效果

此时我们发现两个消费者（stream-consumer-rabbitmq、stream-consumer-rabbitmq-backup）只有一个

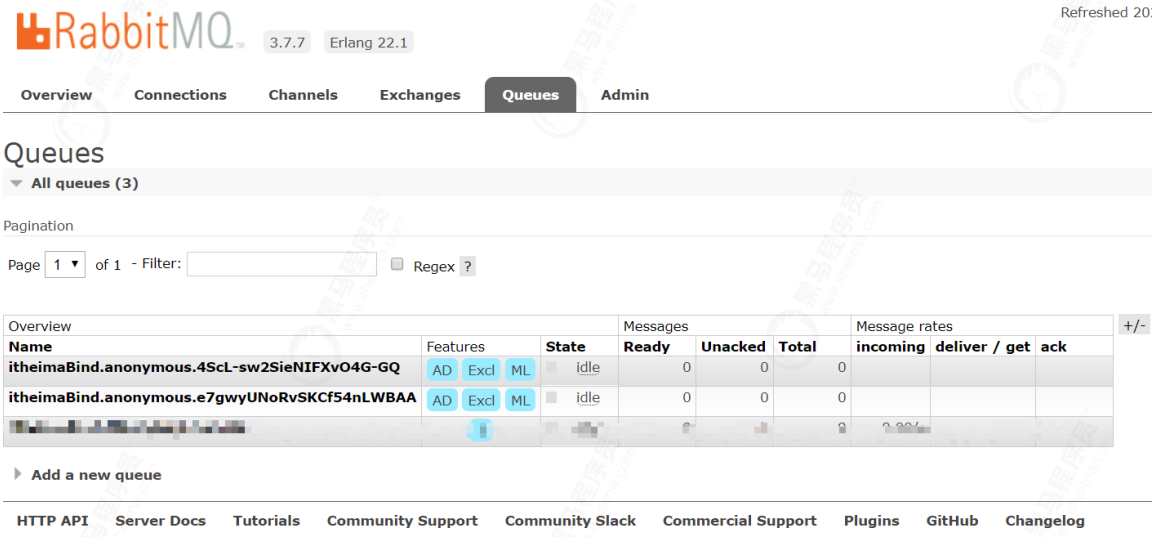
消费者（stream-consumer-rabbitmq，端口为9090）消费了消息，**成功解决了重复消费问题**

查看RabbitMQ web页面交换器（Exchanges）itheimaBind，如下图:



上图的交换器itheima是我们生产者消费者对应的目的绑定器

查看RabbitMQ web页面交换器（Queues）itheimaBind，如下图:



上图RabbitMQ对应的Queues队列，对应我们消费者；订阅队列

重复消费总结：

当我们指定了某个绑定所指向的消费组之后，往当前主题发送的消息在每个订阅消费组中，

只会有一个订阅者接收和消费，从而实现了对消息的负载均衡。

只所以之前会出现重复消费的问题，是由于默认情况下，任何订阅都会产生一个匿名消费组，

所以每个订阅实例都会有自己的消费组，从而当有消息发送的时候，就形成了广播的模式

4 微服务下分布式链路追踪器

4.1 Sleuth链路追踪简介

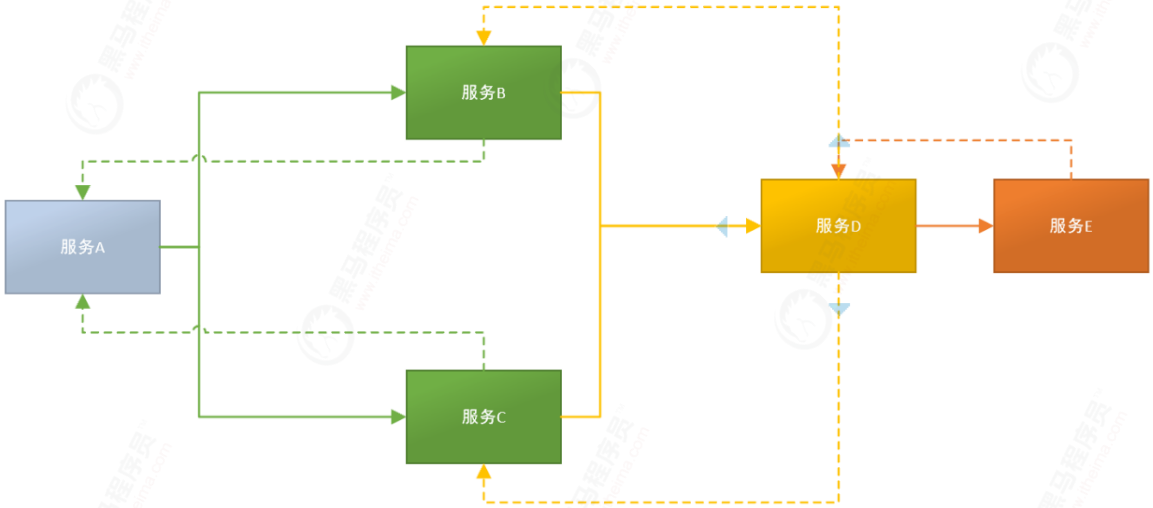
Sleuth概念

Spring-Cloud-Sleuth是Spring Cloud的组成部分之一

为SpringCloud应用实现了一种分布式追踪解决方案，其兼容了Zipkin, HTrace和log-based追踪

目前，链路追踪组件有Google的Dapper，Twitter 的Zipkin，以及阿里的Eagleeye （鹰眼）等，它们都是非常优秀的链路追踪开源组件。

2. 为什么需要Sleuth



3. Sleuth解决了什么问题

提供链路追踪	**通过sleuth可以很清楚的看出一个请求经过了哪些服务，可以方便的理清服务局的调用关系**
性能分析	通过sleuth可以很方便的看出每个采集请求的耗时，分析出哪些服务调用比较耗时，当服务调用的耗时随着请求量的增大而增大时，也可以对服务的扩容提供一定的提醒作用
数据分析优化链路	对于频繁地调用一个服务，或者并行地调用等，可以针对业务做一些优化措施
可视化	对于程序未捕获的异常，可以在zipkpin界面上看到

4.2 Zipkin链路数据收集

Zipkin概念

Zipkin 是一个开放源代码分布式的跟踪系统，它基于 Google Dapper 实现；由Twitter公司开源，它致力于收集服务的定时数据，

每个服务向zipkin报告计时链路数据，以解决微服务架构中的延迟问题。

主要功能：数据的收集、存储、查找和展现。

优势：

Zipkin 会根据调用关系通过Zipkin UI生成依赖关系图，显示多少跟踪请求通过每个服务，该系统让开发者可通过一个 Web 前端轻松的收集和分析数据，例如用户每次请求服务的处理时间等，可方便的监测系统中存在的瓶颈。

为什么要使用Zipkin

因为sleuth对于分布式链路的跟踪仅仅是一些数据的记录， 这些数据人为来读取和处理难免会太麻烦，所以，我们一般会把数据上交给Zipkin Server 来统一处理、展现和分析

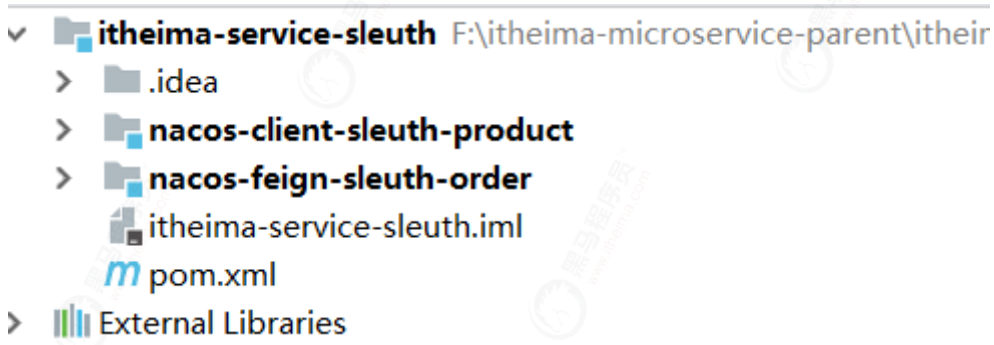
> Zipkin Server分为安装版和构建版
>
> 安装版：下载 Zipkin Server可执行jar
>
> 　　java -jar运行 无需代码构建 （F版本后）
>
> 构建版：引入Zipkin Server起步依赖，入口类开启EnableZipkinServer

4.3 Sleuth快速入门

代码讲解：

```
    <modules>
        <module>nacos-client-sleuth-product</module>
        <module>nacos-feign-sleuth-order</module>
    </modules>
```

具体场景：



1. 创建Sleuth模块

创建工程nacos-feign-sleuth-order

2. 引入起步依赖

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

3. 修改application.yml

```
spring:
  application:
    name: nacos-feign-sleuth
  sleuth:
    web:
      client:
        enabled: true
```

```
    sampler:
      probability: 1.0 # 将采样比例设置为 1.0，也就是全部都需要。默认是 0.1
  zipkin:
    base-url: http://localhost:9411/ # 指定了 Zipkin 服务器的地址
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.23.139

server:
  port: 8765
```

应用程序在监测到Java依赖包中有sleuth和zipkin后，**会自动在RestTemplate（或feign）的调用过程中向HTTP请求注入追踪信息**，并向Zipkin Server发送这些信息

```
spring:
zipkin:
 base-url: http://localhost:9411
sleuth:
 sampler:
   percentage: 1.0
```

probability:将采样比例设置为 1.0，也就是全部都需要。默认是 0.1，由于分布式系统的请求量一般比较大，不可能把所有的请求链路进行收集整理，因此sleuth采用抽样收集的方式，设定一个抽样百分比。在开发阶段，我们一般设定百分比为100%也就是1

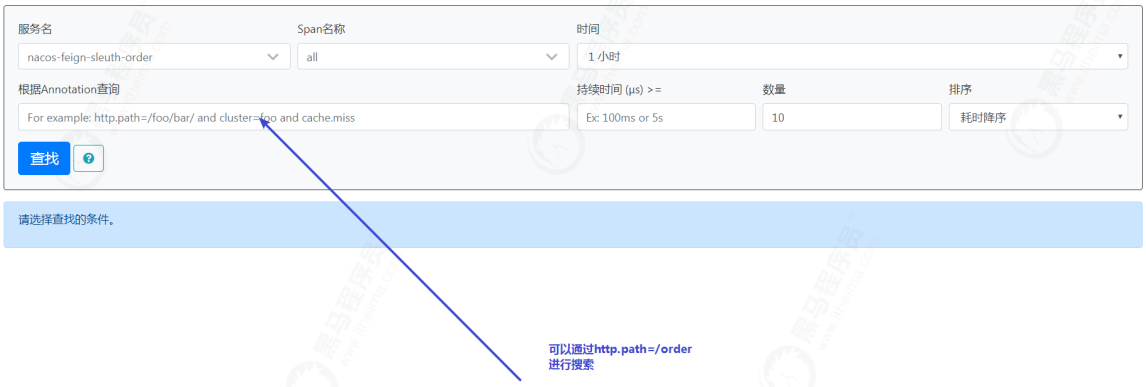
base-url:Zipkin 服务器的地址

4. 启动

启动 Zipkin Server

zipkin server是一个可执行jar包，浏览器输入，双击就可以运行

```
http://localhost:9411/zipkin/
```

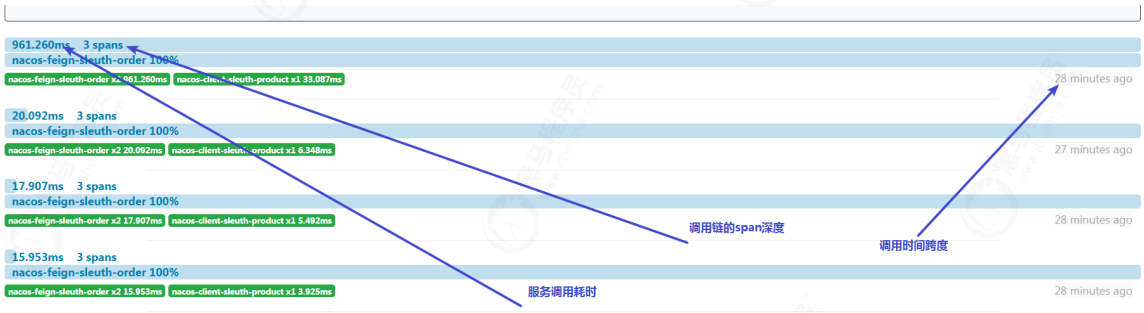


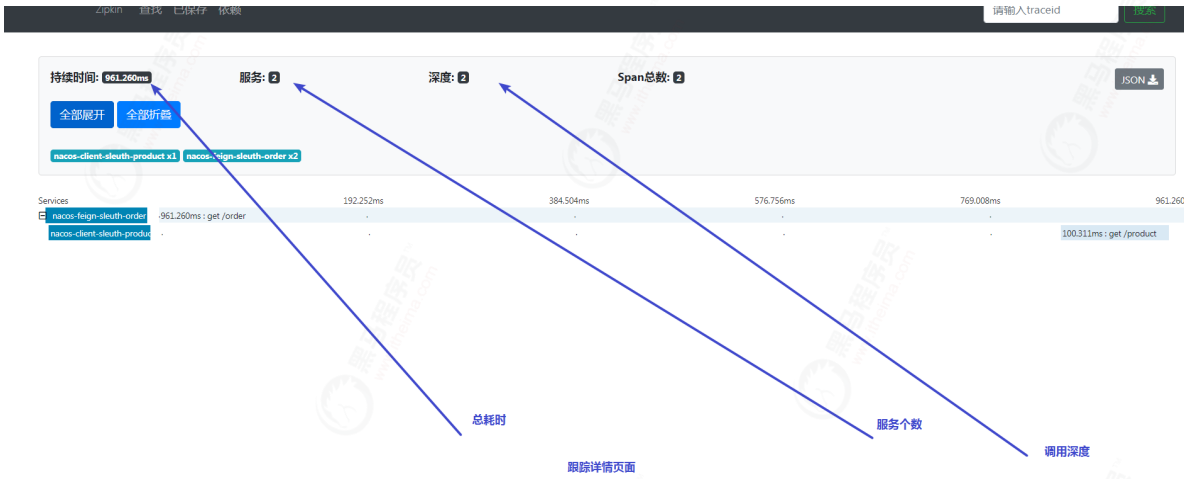
启动 nacos-client-sleuth-product、启动nacos-feign-sleuth-order，如下图：

4.4 Zipkin调用链路可视化

浏览器输入

```
http://localhost:8888/order
```





点击link出现了详细信息，如下图

Services: nacos-client-sleuth-product,nacos-feign-sleuth-order

Date Time	Relative Time	Annotation	Address
2020/8/11 下午5:21:46	834.059ms	Client Start	172.16.43.175 (nacos-feign-sleuth-order)
2020/8/11 下午5:21:47	901.372ms	Server Start	172.16.43.175 (nacos-client-sleuth-product)
2020/8/11 下午5:21:47	934.370ms	Client Finish	172.16.43.175 (nacos-feign-sleuth-order)
2020/8/11 下午5:21:47	934.459ms	Server Finish	172.16.43.175 (nacos-client-sleuth-product)

Key	Value
http.method	GET
http.path	/product
mvc.controller.class	ProductController
mvc.controller.method	product
Client Address	172.16.43.175:11450

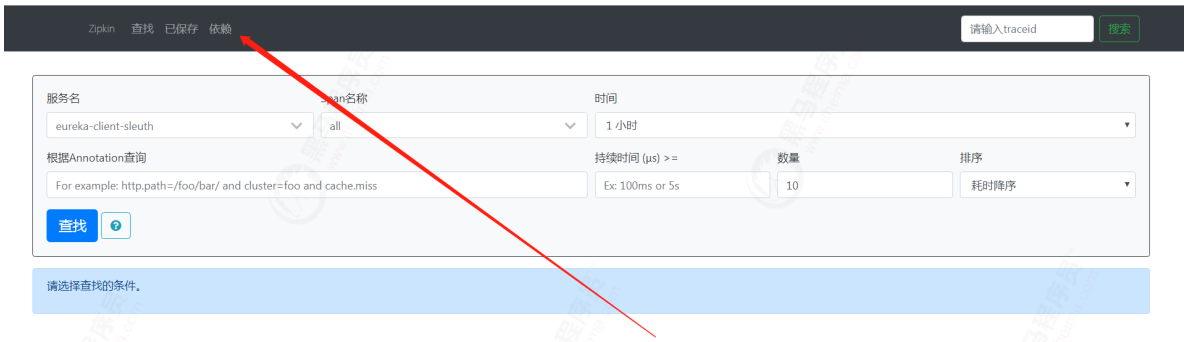
调用时间

相对时间

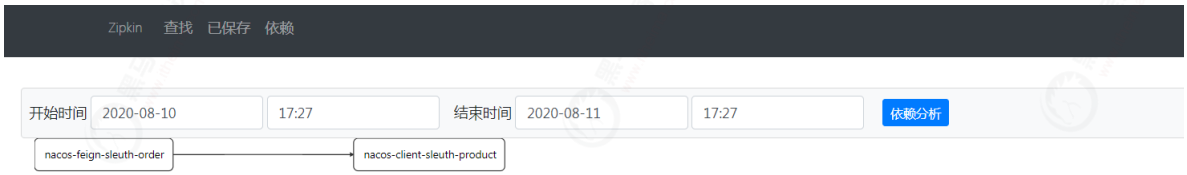
调用行为

展现ID

除了可以查看服务链路信息为，zipkin还可以查看服务依赖情况，点击【依赖】，如下图



点击【依赖】，展示依赖详情，如下图：



Key	Value
Number of calls	10
Number of errors	0

> Number of calls： 总的调用数（除去异常的）
>
> Number of errors：调用异常的次数