# 第一章 spring cloud 简介

## 一、介绍

Spring Cloud是一个基于Spring Boot实现的云应用开发工具，它为基于JVM的云应用开发中涉及的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。Spring Cloud包含了多个子项目（针对分布式系统中涉及的多个不同开源产品），比如：Spring Cloud Config、Spring Cloud Netflix、Spring Cloud0 CloudFoundry、Spring Cloud AWS、Spring Cloud Security、Spring Cloud Commons、Spring Cloud Zookeeper、Spring Cloud CLI等项目。

## 二、微服务架构

“微服务架构”在这几年非常的火热，以至于关于微服务架构相关的开源产品被反复的提及（比如：netflix、dubbo），Spring Cloud也因Spring社区的强大知名度和影响力也被广大架构师与开发者备受关注。那么什么是“微服务架构”呢？简单的说，微服务架构就是将一个完整的应用从数据存储开始垂直拆分成多个不同的服务，每个服务都能独立部署、独立维护、独立扩展，服务与服务间通过诸如RESTful API的方式互相调用。对于“微服务架构”，大家在互联网可以搜索到很多相关的介绍和研究文章来进行学习和了解。

## 三、服务治理

在简单介绍了Spring Cloud和微服务架构之后，下面回归本文的主旨内容，如何使用Spring Cloud来实现服务治理。由于Spring Cloud为服务治理做了一层抽象接口，所以在Spring Cloud应用中可以支持多种不同的服务治理框架，比如：Netflix Eureka、Consul、Zookeeper。在Spring Cloud服务治理抽象层的作用下，我们可以无缝地切换服务治理实现，并且不影响任何其他的服务注册、服务发现、服务调用等逻辑。所以，下面我们通过介绍两种服务治理的实现来体会Spring Cloud这一层抽象所带来的好处。

## 四、与Dubbo对比

SpringCloud整合了一套较为完整的微服务解决方案框架，而Dubbo只是解决了微服务的几个方面的问题。

|  |  |  |
| --- | --- | --- |
| content | Dubbo | SpringCloud |
| 服务注册中心 | zookeeper | Spring Cloud Netflix Eureka |
| 服务调用方式 | RPC | REST API |
| 服务网关 | 无 | Spring Cloud Netflix Zuul |
| 断路器 | 不完善 | Spring Cloud Netflix Hystrix |
| 分布式配置 | 无 | Spring Cloud Config |
| 服务跟踪 | 无 | Spring Cloud Sleuth |
| 消息总线 | 无 | Spring Cloud Bus |
| 数据流 | 无 | Spring Cloud Stream |
| 批量任务 | 无 | Spring Cloud Task |

当然，虽然dubbo没有提供很多解决方案，但他也可以整合第三方的项目来实现。

# 第二章 服务的注册与发现

## 一、Spring Cloud Eureka

首先，我们来尝试使用Spring Cloud Eureka来实现服务治理。

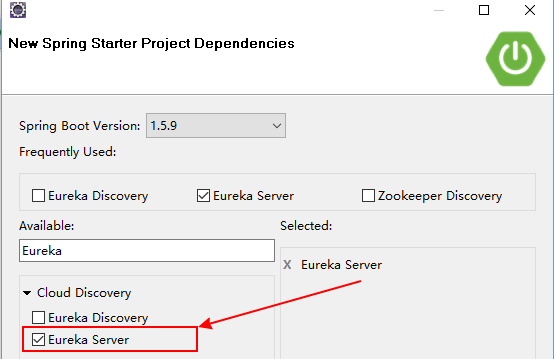
Spring Cloud Eureka是Spring Cloud Netflix项目下的服务治理模块。而Spring Cloud Netflix项目是Spring Cloud的子项目之一，主要内容是对Netflix公司一系列开源产品的包装，它为Spring Boot应用提供了自配置的Netflix OSS整合。通过一些简单的注解，开发者就可以快速的在应用中配置一下常用模块并构建庞大的分布式系统。它主要提供的模块包括：服务发现（Eureka），断路器（Hystrix），智能路由（Zuul），客户端负载均衡（Ribbon）等。

下面，就来具体看看如何使用Spring Cloud Eureka实现服务治理。

## 二、创建服务注册中心

1.创建一个spring boot项目

项目名称是eureka-server,选择插件的时候勾中Eureka-server，即可。



pom.xml的内容如下。注意spring-cloud.version的版本是Edgware.RELEASE。

|  |
| --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <spring-cloud.version>Edgware.RELEASE</spring-cloud.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

2.配置

在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需要在application.properties配置文件中增加如下信息：

|  |
| --- |
| spring.application.name=eureka-server  server.port=8761  eureka.client.service-url.defaultZone=http://127.0.0.1:${server.port}/eureka/  #eureka.instance.hostname=127.0.0.1  eureka.client.register-with-eureka=false  eureka.client.fetch-registry=false |

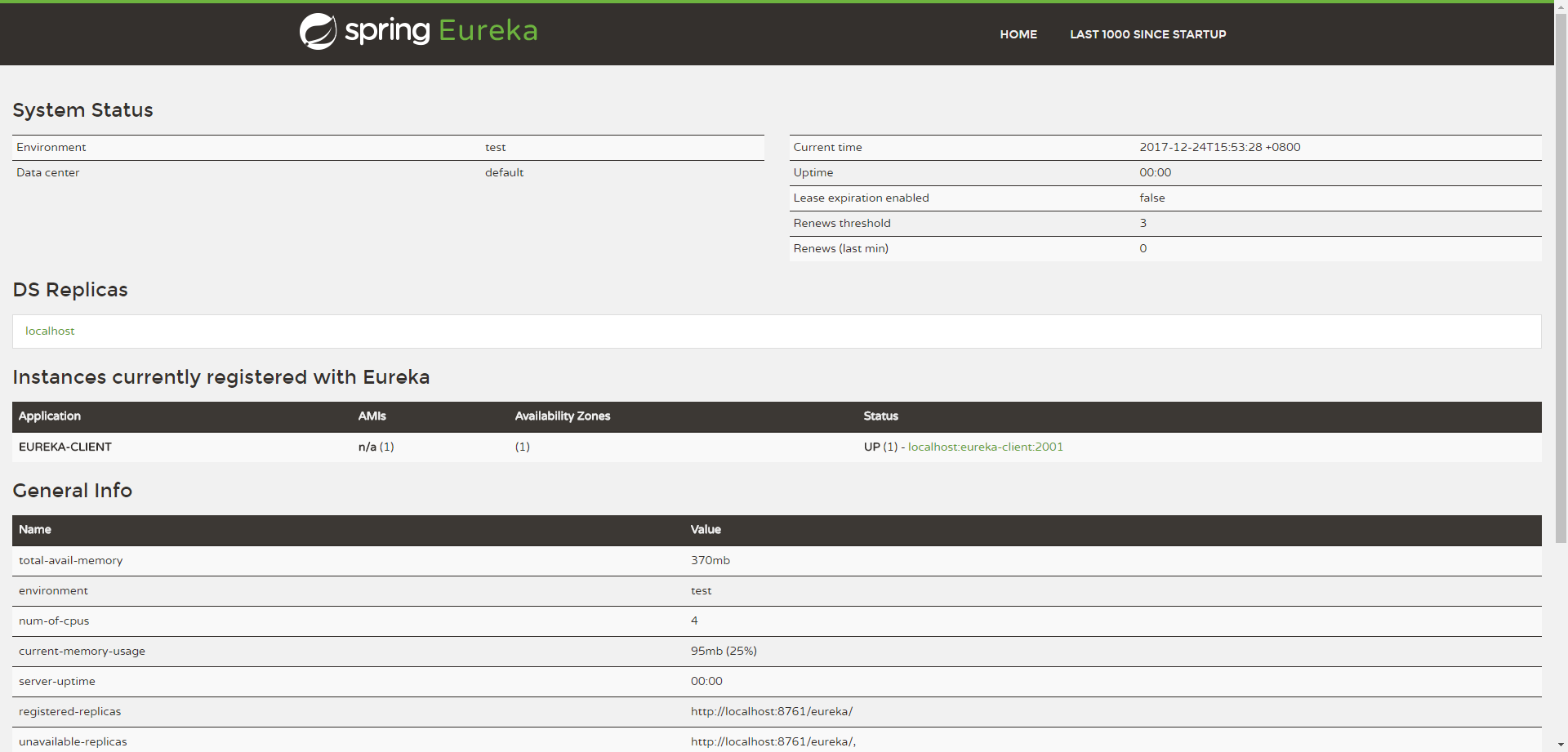
为了与后续要进行注册的服务区分，这里将服务注册中心的端口通过server.port属性设置为1001。

3.创建启动

通过@EnableEurekaServer注解启动一个服务注册中心提供给其他应用进行对话。这一步非常的简单，只需要在一个普通的Spring Boot应用中添加这个注解就能开启此功能，比如下面的例子：

|  |
| --- |
| @EnableEurekaServer  @SpringBootApplication  public class EurekaServerApplication {  public static void main(String[] args) {  SpringApplication.run(EurekaServerApplication.class, args);  }  } |

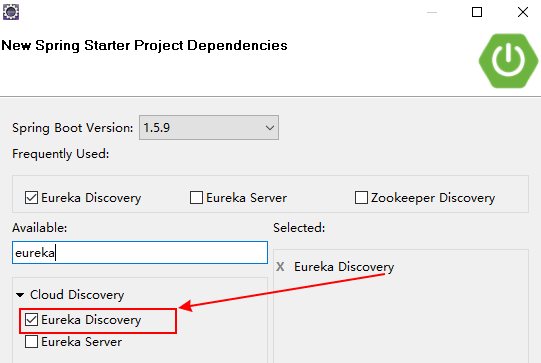
启动工程后，访问：http://localhost:8761/，可以看到下面的页面，其中还没有发现任何服务。



## 三、创建服务提供方

下面我们创建提供服务的客户端，并向服务注册中心注册自己。本文我们主要介绍服务的注册与发现，所以我们不妨在服务提供方中尝试着提供一个接口来获取当前所有的服务信息。

1. 创建一个spring boot项目，项目名称为：eureka-simple-client。选择Eureka Discovery，以及web组件。



|  |
| --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <spring-cloud.version>Edgware.RELEASE</spring-cloud.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

2. 配置

通过spring.application.name属性，我们可以指定微服务的名称后续在调用的时候只需要使用该名称就可以进行服务的访问。eureka.client.service-url.defaultZone属性对应服务注册中心的配置内容，指定服务注册中心的位置。为了在本机上测试区分服务提供方和服务注册中心，使用server.port属性设置不同的端口。

|  |
| --- |
| spring.application.name=eureka-simple-client  server.port=2001  eureka.client.service-url.defaultZone=http://127.0.0.1:8761/eureka/ |

3. 创建一个restFul服务接口

DiscoveryClient类就是服务的监管类

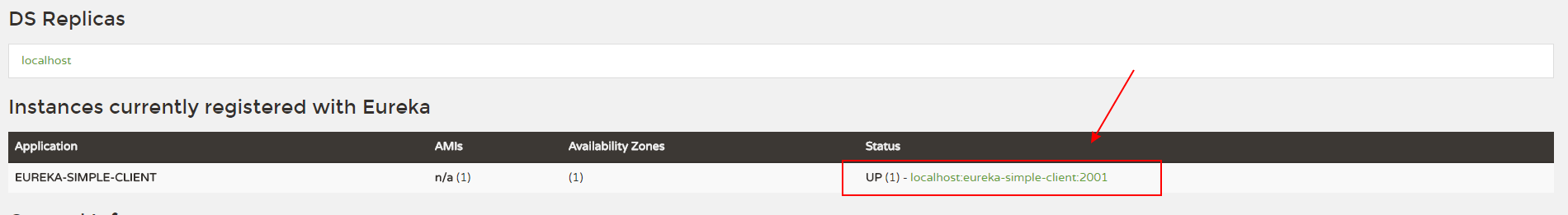
|  |
| --- |
| @RestController  public class DcController {  @Autowired  private DiscoveryClient discoveryClient;  @GetMapping("/dc")  public List<String> dc(){  List<String> services = discoveryClient.getServices();  System.out.println(services);  return services;  }  } |

4.创建启动

最后在应用主类中通过加上@EnableDiscoveryClient注解，该注解能激活Eureka中的DiscoveryClient实现，这样才能实现Controller中对服务信息的输出。

|  |
| --- |
| @EnableDiscoveryClient  @SpringBootApplication  public class EurekaSimpleClientApplication {  public static void main(String[] args) {  SpringApplication.run(EurekaSimpleClientApplication.class, args);  }  } |

启动工程后，访问：http://localhost:8761/，可以看到下面的页面



访问<http://localhost:2001/dc>，返回：["eureka-simple-client"]

## 四、Consul

除了Eureka之外，还有Consul和zookeeper作为注册中心。

Consul是一个分布式高可用的系统，它包含多个组件，但是作为一个整体，在微服务架构中为我们的基础设施提供服务发现和服务配置的工具。它包含了下面几个特性：

1. 服务发现
2. 健康检查
3. Key/Value存储
4. 多数据中心

官网：<https://www.consul.io/>

1. 下载安装consul

consul作为第三方工具被安装后用于服务监管

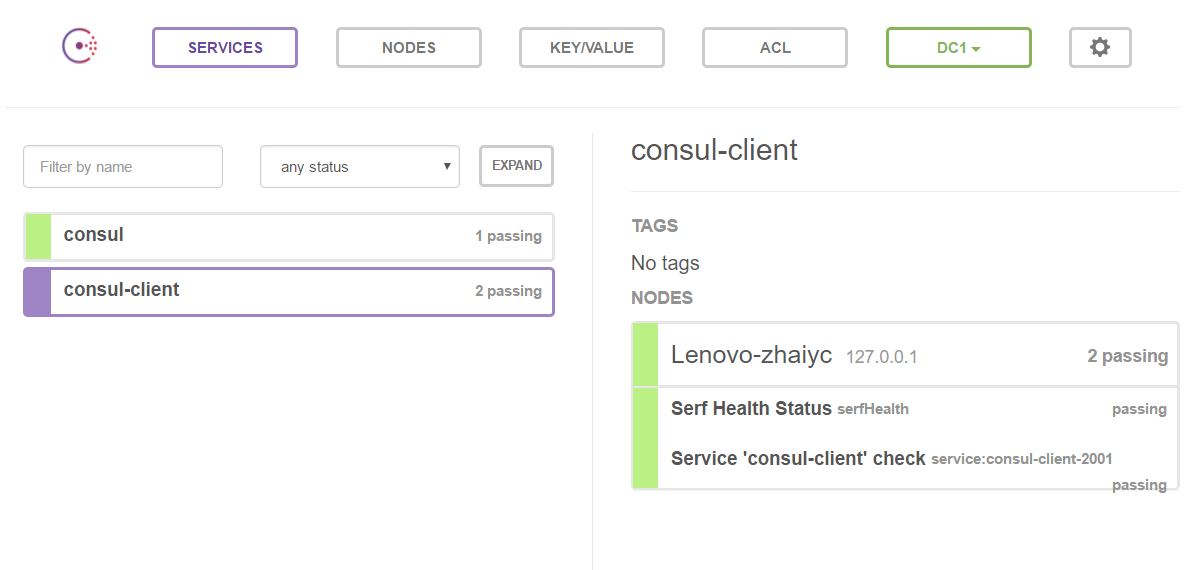
2. 创建一个服务提供者，项目与上面的三一样。只是将eureka-client换成consul-discovery。

3. 配置

|  |
| --- |
| spring.application.name=eureka-server  server.port=1001  spring.cloud.consul.host=localhost  spring.cloud.consul.port=8500 |

4. 创建启动与三一样。

启动后如下界面。



# 第三章 服务消费

## 一、简单使用消费者--LoadBalancerClient

在Spring Cloud Commons中提供了大量的与服务治理相关的抽象接口，包括DiscoveryClient、这里我们即将介绍的LoadBalancerClient等。对于这些接口的定义我们在上一篇介绍服务注册与发现时已经说过，Spring Cloud做这一层抽象，很好的解耦了服务治理体系，使得我们可以轻易的替换不同的服务治理设施。

从LoadBalancerClient接口的命名中，我们就知道这是一个负载均衡客户端的抽象定义，下面我们就看看如何使用Spring Cloud提供的负载均衡器客户端接口来实现服务的消费。

1. 创建一个消费者工程

项目名称为eureka-consumer。项目依赖和eureka-client一样。

2.配置

消费者需要注册到注册中心用来调用服务。

|  |
| --- |
| spring.application.name=eureka-simple-consumer  server.port=8080  eureka.client.service-url.defaultZone=http://127.0.0.1:8761/eureka/ |

3.创建启动类

初始化RestTemplate，用来真正发起REST请求。@EnableDiscoveryClient注解用来将当前应用加入到服务治理体系中。

|  |
| --- |
| @EnableDiscoveryClient  @SpringBootApplication  public class EurekaSimpleConsumerApplication {  @Bean  public RestTemplate restTemplate() {  return new RestTemplate();  }    public static void main(String[] args) {  SpringApplication.run(EurekaSimpleConsumerApplication.class, args);  }  } |

4.创建消费者

创建一个接口用来消费eureka-simple-client提供的接口

|  |
| --- |
| @RestController  public class DcController {  @Autowired  LoadBalancerClient loadBalancerClient;    @Autowired  RestTemplate restTemplate;  @GetMapping("/consumer")  public String dc() {  ServiceInstance serviceInstance = loadBalancerClient.choose("eureka-simple-client");  String url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/dc";  System.out.println(url);  return restTemplate.getForObject(url, String.class);  }  } |

可以看到这里，我们注入了LoadBalancerClient和RestTemplate，并在/consumer接口的实现中，先通过loadBalancerClient的choose函数来负载均衡的选出一个eureka-client的服务实例，这个服务实例的基本信息存储在ServiceInstance中，然后通过这些对象中的信息拼接出访问/dc接口的详细地址，最后再利用RestTemplate对象实现对服务提供者接口的调用。

在完成了上面你的代码编写之后，读者可以将eureka-server、eureka-client、eureka-consumer都启动起来，然后访问http://localhost:2101/consumer ，来跟踪观察eureka-consumer服务是如何消费eureka-client服务的/dc接口的。

## 二、便捷使用消费者--Ribbon（推荐）

我们已经学会如何通过LoadBalancerClient接口来获取某个服务的具体实例，并根据实例信息来发起服务接口消费请求。但是这样的做法需要我们手工的去编写服务选取、链接拼接等繁琐的工作，对于开发人员来说非常的不友好。所以，下来我们看看Spring Cloud中针对客户端负载均衡的工具包：Spring Cloud Ribbon。

1.Spring Cloud Ribbon

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端负载均衡的工具。它是一个基于HTTP和TCP的客户端负载均衡器。它可以通过在客户端中配置ribbonServerList来设置服务端列表去轮询访问以达到均衡负载的作用。

当Ribbon与Eureka联合使用时，ribbonServerList会被DiscoveryEnabledNIWSServerList重写，扩展成从Eureka注册中心中获取服务实例列表。同时它也会用NIWSDiscoveryPing来取代IPing，它将职责委托给Eureka来确定服务端是否已经启动。

而当Ribbon与Consul联合使用时，ribbonServerList会被ConsulServerList来扩展成从Consul获取服务实例列表。同时由ConsulPing来作为IPing接口的实现。

我们在使用Spring Cloud Ribbon的时候，不论是与Eureka还是Consul结合，都会在引入Spring Cloud Eureka或Spring Cloud Consul依赖的时候通过自动化配置来加载上述所说的配置内容，所以我们可以快速在Spring Cloud中实现服务间调用的负载均衡。

2.加入依赖

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

3.修改启动类

为RestTemplate增加@LoadBalanced注解

|  |
| --- |
| @EnableDiscoveryClient  @SpringBootApplication  public class EurekaSimpleConsumerApplication {  @Bean  @LoadBalanced  public RestTemplate restTemplate() {  return new RestTemplate();  }    public static void main(String[] args) {  SpringApplication.run(EurekaSimpleConsumerApplication.class, args);  }  } |

4.修改消费者

去掉原来通过LoadBalancerClient选取实例和拼接URL的步骤，直接通过RestTemplate发起请求

|  |
| --- |
| @RestController  public class DcController2 {    @Autowired  RestTemplate restTemplate;  @GetMapping("/demo")  public String demo() {  return restTemplate.getForObject("http://eureka-simple-client/dc", String.class);  }  } |

除了去掉了原来与LoadBalancerClient相关的逻辑之外，对于RestTemplate的使用，我们的第一个url参数有一些特别。这里请求的host位置并没有使用一个具体的IP地址和端口的形式，而是采用了服务名的方式组成。那么这样的请求为什么可以调用成功呢？因为Spring Cloud Ribbon有一个拦截器，它能够在这里进行实际调用的时候，自动的去选取服务实例，并将实际要请求的IP地址和端口替换这里的服务名，从而完成服务接口的调用。

在完成了上面你的代码编写之后，读者可以将eureka-server、eureka-client、eureka-consumer-ribbon都启动起来，然后访问http://localhost:2101/consumer ，来跟踪观察eureka-consumer-ribbon服务是如何消费eureka-client服务的/dc接口的，并且也可以通过启动多个eureka-client服务来观察其负载均衡的效果。

## 三、生产环境的消费者--Feign（推荐）

Spring Cloud Feign是一套基于Netflix Feign实现的声明式服务调用客户端。它使得编写Web服务客户端变得更加简单。我们只需要通过创建接口并用注解来配置它既可完成对Web服务接口的绑定。它具备可插拔的注解支持，包括Feign注解、JAX-RS注解。它也支持可插拔的编码器和解码器。Spring Cloud Feign还扩展了对Spring MVC注解的支持，同时还整合了Ribbon和Eureka来提供均衡负载的HTTP客户端实现。

2.加入依赖

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

3.修改启动类

通过@EnableFeignClients注解开启扫描Spring Cloud Feign客户端的功能

|  |
| --- |
| @EnableFeignClients  @EnableDiscoveryClient  @SpringBootApplication  public class EurekaSimpleConsumerApplication {    public static void main(String[] args) {  SpringApplication.run(EurekaSimpleConsumerApplication.class, args);  }  } |

4.创建一个接口的定义

创建一个Feign的客户端接口定义。使用@FeignClient注解来指定这个接口所要调用的服务名称，接口中定义的各个函数使用Spring MVC的注解就可以来绑定服务提供方的REST接口，比如下面就是绑定eureka-client服务的/dc接口的例子

|  |
| --- |
| @FeignClient("eureka-simple-client")  public interface DcClient {    @GetMapping("/dc")  String dc();  } |

5.修改消费者

去掉原来通过LoadBalancerClient选取实例和拼接URL的步骤，直接通过RestTemplate发起请求

|  |
| --- |
| @RestController  public class DcController {  @Autowired  DcClient dcClient;  @GetMapping("/consumer")  public String dc() {  return dcClient.dc();  }    } |

通过Spring Cloud Feign来实现服务调用的方式更加简单了，通过@FeignClient定义的接口来统一的生命我们需要依赖的微服务接口。而在具体使用的时候就跟调用本地方法一点的进行调用即可。由于Feign是基于Ribbon实现的，所以它自带了客户端负载均衡功能，也可以通过Ribbon的IRule进行策略扩展。另外，Feign还整合的Hystrix来实现服务的容错保护，在Dalston版本中，Feign的Hystrix默认是关闭的。待后文介绍Hystrix带领大家入门之后，我们再结合介绍Feign中的Hystrix以及配置方式。

在完成了上面你的代码编写之后，读者可以将eureka-server、eureka-client、eureka-consumer-feign都启动起来，然后访问http://localhost:8080/consumer ，来跟踪观察eureka-consumer-feign服务是如何消费eureka-client服务的/dc接口的，并且也可以通过启动多个eureka-client服务来观察其负载均衡的效果。

# 第四章 分布式配置中心

Spring Cloud Config是Spring Cloud团队创建的一个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持，它分为服务端与客户端两个部分。其中服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置仓库并为客户端提供获取配置信息、加密/解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。Spring Cloud Config实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于Spring构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。由于Spring Cloud Config实现的配置中心默认采用Git来存储配置信息，所以使用Spring Cloud Config构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过Git客户端工具来方便的管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如：SVN仓库、本地化文件系统。

以下是Git为案例说明，其他方式参考官网

## 一. 准备git

创建一个git仓库。比如：<https://gitee.com/zx19890628/spring-cloud-config-server/>

创建三个个配置文件

|  |
| --- |
| config.propertie  **a**=default  **b**=default |

|  |
| --- |
| config-dev.propertie  **a**=dev  **b**=dev |

|  |
| --- |
| config-test.propertie  **a**=test  **b**=test |

## 二.创建配置中心

1.maven依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency> |

2.创建启动类，添加@EnableConfigServer,开启Spring Cloud Config的服务端功能。

|  |
| --- |
| @EnableConfigServer  @SpringBootApplication  public class ConfigServerApplication {  public static void main(String[] args) {  SpringApplication.run(ConfigServerApplication.class, args);  }  } |

3. 添加配置

|  |
| --- |
| spring.application.name=config-server  spring.cloud.config.server.git.uri=https://gitee.com/zx19890628/spring-cloud-config-server/  #spring.cloud.config.server.git.username=username  #spring.cloud.config.server.git.password=password  server.port=1001 |

如果我们的Git仓库需要权限访问，那么可以通过配置下面的两个属性来实现；

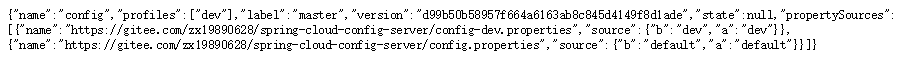
spring.cloud.config.server.git.username：访问Git仓库的用户名

spring.cloud.config.server.git.password：访问Git仓库的用户密码

4.启动项目

浏览器访问：<http://localhost:1001/config/dev/master>

出现以下内容，即为正常



完成了这些准备工作之后，我们就可以通过浏览器、POSTMAN或CURL等工具直接来访问到我们的配置内容了。访问配置信息的URL与配置文件的映射关系如下：

1. /{application}/{profile}[/{label}]
2. /{application}-{profile}.yml
3. /{label}/{application}-{profile}.yml
4. /{application}-{profile}.properties
5. /{label}/{application}-{profile}.properties

上面的url会映射{application}-{profile}.properties对应的配置文件，其中{label}对应Git上不同的分支，默认为master。我们可以尝试构造不同的url来访问不同的配置内容，比如，要访问master分支，config-client应用的dev环境，就可以访问这个url：http://localhost:1201/config-client/dev/master

## 三、创建客户端

1. maven依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config</artifactId>  </dependency> |

2. 创建启动类

默认即可

3.创建配置

这里的配置有两个。一个是本项目自己使用的application.properties，另一个是bootstrap.properties

bootstrap是引用配置中心的配置。

|  |
| --- |
| spring.cloud.config.uri=http://127.0.0.1:1001  spring.cloud.config.name=config  spring.cloud.config.profile=dev  spring.cloud.config.label=master |

上述配置参数与Git中存储的配置文件中各个部分的对应关系如下：

1. spring.application.name：对应配置文件规则中的{application}部分
2. spring.cloud.config.profile：对应配置文件规则中的{profile}部分
3. spring.cloud.config.label：对应配置文件规则中的{label}部分
4. spring.cloud.config.uri：配置中心config-server的地址

4. 创建调用配置的类

|  |
| --- |
| @Component  public class Config {  @Value("${a}")  private String a;  @Value("${b}")  private String b;  } |

|  |
| --- |
| @RestController  public class DemoController {  @Autowired  private Config config;  @GetMapping("/get")  public String get(){  return config.toString();  }  } |

5.启动并访问项目

<http://localhost:8080/get>

返回字符串：Config [a=dev, b=dev]

# 第五章 服务容错保护

## 一、Hystrix服务降级

在微服务架构中，我们将系统拆分成了一个个的服务单元，各单元应用间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会出现因等待出现故障的依赖方响应而形成任务积压，线程资源无法释放，最终导致自身服务的瘫痪，进一步甚至出现故障的蔓延最终导致整个系统的瘫痪。如果这样的架构存在如此严重的隐患，那么相较传统架构就更加的不稳定。为了解决这样的问题，因此产生了断路器等一系列的服务保护机制。

针对上述问题，在Spring Cloud Hystrix中实现了线程隔离、断路器等一系列的服务保护功能。它也是基于Netflix的开源框架 Hystrix实现的，该框架目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix具备了服务降级、服务熔断、线程隔离、请求缓存、请求合并以及服务监控等强大功能。

1.准备服务

eureka-server工程：服务注册中心，端口：8671

eureka-client工程：服务提供者，两个实例启动端口分别为2001、2002

使用之前的eureka-consumer-ribbon项目，修改名称为eureka-consumer-ribbon-hystrix

2.引入依赖

pom.xml的dependencies节点中引入spring-cloud-starter-hystrix依赖：

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

3.创建启动类

在应用主类中使用@EnableCircuitBreaker或@EnableHystrix注解开启Hystrix的使用：

@EnableCircuitBreaker

@EnableDiscoveryClient

@SpringBootApplication

public class Application {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

注意：这里我们还可以使用Spring Cloud应用中的@SpringCloudApplication注解来修饰应用主类，该注解的具体定义如下所示。我们可以看到该注解中包含了上我们所引用的三个注解，这也意味着一个Spring Cloud标准应用应包含服务发现以及断路器。

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootApplication

@EnableDiscoveryClient

@EnableCircuitBreaker

public @interface SpringCloudApplication {

}

4.改造服务消费方式

新增ConsumerService类，然后将在Controller中的逻辑迁移过去。最后，在为具体执行逻辑的函数上增加@HystrixCommand注解来指定服务降级方法，比如：

@RestController

public class DcController {

@Autowired

ConsumerService consumerService;

@GetMapping("/consumer")

public String dc() {

return consumerService.consumer();

}

class ConsumerService {

@Autowired

RestTemplate restTemplate;

@HystrixCommand(fallbackMethod = "fallback")

public String consumer() {

return restTemplate.getForObject("http://eureka-client/dc", String.class);

}

public String fallback() {

return "fallback";

}

}

}

下面我们来验证一下上面Hystrix带来的一些基础功能。我们先把涉及的服务都启动起来，然后访问localhost:2101/consumer，此时可以获取正常的返回，比如：Services: [eureka-consumer-ribbon-hystrix, eureka-client]。

为了触发服务降级逻辑，我们可以将服务提供者eureka-client的逻辑加一些延迟，比如：

@GetMapping("/dc")

public String dc() throws InterruptedException {

Thread.sleep(5000L);

String services = "Services: " + discoveryClient.getServices();

System.out.println(services);

return services;

}

重启eureka-client之后，再尝试访问localhost:2101/consumer，此时我们将获得的返回结果为：fallback。我们从eureka-client的控制台中，可以看到服务提供方输出了原本要返回的结果，但是由于返回前延迟了5秒，而服务消费方触发了服务请求超时异常，服务消费者就通过HystrixCommand注解中指定的降级逻辑进行执行，因此该请求的结果返回了fallback。这样的机制，对自身服务起到了基础的保护，同时还为异常情况提供了自动的服务降级切换机制。

## 二、Hystrix依赖隔离

上面的服务降级实现方式非常简单，同时对于降级逻辑还能实现一些更加复杂的级联降级等策略。之前对于使用Hystrix来实现服务容错保护时，除了服务降级之外，还有线程隔离、断路器等功能。

1.依赖隔离

“舱壁模式”对于熟悉Docker的读者一定不陌生，Docker通过“舱壁模式”实现进程的隔离，使得容器与容器之间不会互相影响。而Hystrix则使用该模式实现线程池的隔离，它会为每一个Hystrix命令创建一个独立的线程池，这样就算某个在Hystrix命令包装下的依赖服务出现延迟过高的情况，也只是对该依赖服务的调用产生影响，而不会拖慢其他的服务。

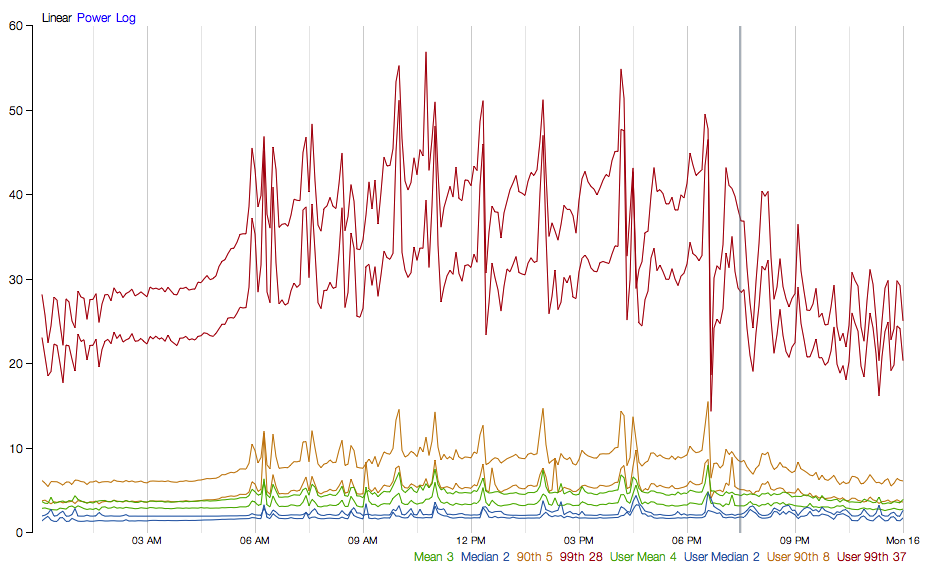
通过对依赖服务的线程池隔离实现，可以带来如下优势：

* 应用自身得到完全的保护，不会受不可控的依赖服务影响。即便给依赖服务分配的线程池被填满，也不会影响应用自身的额其余部分。
* 可以有效的降低接入新服务的风险。如果新服务接入后运行不稳定或存在问题，完全不会影响到应用其他的请求。
* 当依赖的服务从失效恢复正常后，它的线程池会被清理并且能够马上恢复健康的服务，相比之下容器级别的清理恢复速度要慢得多。
* 当依赖的服务出现配置错误的时候，线程池会快速的反应出此问题（通过失败次数、延迟、超时、拒绝等指标的增加情况）。同时，我们可以在不影响应用功能的情况下通过实时的动态属性刷新（后续会通过Spring Cloud Config与Spring Cloud Bus的联合使用来介绍）来处理它。
* 当依赖的服务因实现机制调整等原因造成其性能出现很大变化的时候，此时线程池的监控指标信息会反映出这样的变化。同时，我们也可以通过实时动态刷新自身应用对依赖服务的阈值进行调整以适应依赖方的改变。
* 除了上面通过线程池隔离服务发挥的优点之外，每个专有线程池都提供了内置的并发实现，可以利用它为同步的依赖服务构建异步的访问。

总之，通过对依赖服务实现线程池隔离，让我们的应用更加健壮，不会因为个别依赖服务出现问题而引起非相关服务的异常。同时，也使得我们的应用变得更加灵活，可以在不停止服务的情况下，配合动态配置刷新实现性能配置上的调整。

虽然线程池隔离的方案带了如此多的好处，但是很多使用者可能会担心为每一个依赖服务都分配一个线程池是否会过多地增加系统的负载和开销。对于这一点，使用者不用过于担心，因为这些顾虑也是大部分工程师们会考虑到的，Netflix在设计Hystrix的时候，认为线程池上的开销相对于隔离所带来的好处是无法比拟的。同时，Netflix也针对线程池的开销做了相关的测试，以证明和打消Hystrix实现对性能影响的顾虑。

下图是Netflix Hystrix官方提供的一个Hystrix命令的性能监控，该命令以每秒60个请求的速度（QPS）向一个单服务实例进行访问，该服务实例每秒运行的线程数峰值为350个。



从图中的统计我们可以看到，使用线程池隔离与不使用线程池隔离的耗时差异如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 比较情况 | 未使用线程池隔离 | 使用了线程池隔离 | 耗时差距 |
| 中位数 | 2ms | 2ms | 2ms |
| 90百分位 | 5ms | 8ms | 3ms |
| 99百分位 | 28ms | 37ms | 9ms |

在99%的情况下，使用线程池隔离的延迟有9ms，对于大多数需求来说这样的消耗是微乎其微的，更何况为系统在稳定性和灵活性上所带来的巨大提升。虽然对于大部分的请求我们可以忽略线程池的额外开销，而对于小部分延迟本身就非常小的请求（可能只需要1ms），那么9ms的延迟开销还是非常昂贵的。实际上Hystrix也为此设计了另外的一个解决方案：信号量。

Hystrix中除了使用线程池之外，还可以使用信号量来控制单个依赖服务的并发度，信号量的开销要远比线程池的开销小得多，但是它不能设置超时和实现异步访问。所以，只有在依赖服务是足够可靠的情况下才使用信号量。在HystrixCommand和HystrixObservableCommand中2处支持信号量的使用：

* 命令执行：如果隔离策略参数execution.isolation.strategy设置为SEMAPHORE，Hystrix会使用信号量替代线程池来控制依赖服务的并发控制。
* 降级逻辑：当Hystrix尝试降级逻辑时候，它会在调用线程中使用信号量。

信号量的默认值为10，我们也可以通过动态刷新配置的方式来控制并发线程的数量。对于信号量大小的估算方法与线程池并发度的估算类似。仅访问内存数据的请求一般耗时在1ms以内，性能可以达到5000rps，这样级别的请求我们可以将信号量设置为1或者2，我们可以按此标准并根据实际请求耗时来设置信号量。

2.使用依赖隔离

说了那么多依赖隔离的好处，那么我们如何使用Hystrix来实现依赖隔离呢？其实定义服务降级的时候，已经自动的实现了依赖隔离。

在使用了@HystrixCommand来将某个函数包装成了Hystrix命令，这里除了定义服务降级之外，Hystrix框架就会自动的为这个函数实现调用的隔离。所以，依赖隔离、服务降级在使用时候都是一体化实现的，这样利用Hystrix来实现服务容错保护在编程模型上就非常方便的，并且考虑更为全面。除了依赖隔离、服务降级之外，还有一个重要元素：断路器，这三个重要利器构成了Hystrix实现服务容错保护的强力组合拳。

## 三、Hystrix断路器

断路器模式源于Martin Fowler的Circuit Breaker一文。“断路器”本身是一种开关装置，用于在电路上保护线路过载，当线路中有电器发生短路时，“断路器”能够及时的切断故障电路，防止发生过载、发热、甚至起火等严重后果。

在分布式架构中，断路器模式的作用也是类似的，当某个服务单元发生故障（类似用电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），直接切断原来的主逻辑调用。但是，在Hystrix中的断路器除了切断主逻辑的功能之外，还有更复杂的逻辑，下面我们来看看它更为深层次的处理逻辑。

以服务降级为例子，来解释断路器的工作原理。当我们把服务提供者eureka-client中加入了模拟的时间延迟之后，在服务消费端的服务降级逻辑因为hystrix命令调用依赖服务超时，触发了降级逻辑，但是即使这样，受限于Hystrix超时时间的问题，我们的调用依然很有可能产生堆积。

这个时候断路器就会发挥作用，那么断路器是在什么情况下开始起作用呢？这里涉及到断路器的三个重要参数：快照时间窗、请求总数下限、错误百分比下限。这个参数的作用分别是：

* 快照时间窗：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的10秒。
* 请求总数下限：在快照时间窗内，必须满足请求总数下限才有资格根据熔断。默认为20，意味着在10秒内，如果该hystrix命令的调用此时不足20次，即时所有的请求都超时或其他原因失败，断路器都不会打开。
* 错误百分比下限：当请求总数在快照时间窗内超过了下限，比如发生了30次调用，如果在这30次调用中，有16次发生了超时异常，也就是超过50%的错误百分比，在默认设定50%下限情况下，这时候就会将断路器打开。

那么当断路器打开之后会发生什么呢？我们先来说说断路器未打开之前，对于之前那个示例的情况就是每个请求都会在当hystrix超时之后返回fallback，每个请求时间延迟就是近似hystrix的超时时间，如果设置为5秒，那么每个请求就都要延迟5秒才会返回。当熔断器在10秒内发现请求总数超过20，并且错误百分比超过50%，这个时候熔断器打开。打开之后，再有请求调用的时候，将不会调用主逻辑，而是直接调用降级逻辑，这个时候就不会等待5秒之后才返回fallback。通过断路器，实现了自动地发现错误并将降级逻辑切换为主逻辑，减少响应延迟的效果。

在断路器打开之后，处理逻辑并没有结束，我们的降级逻辑已经被成了主逻辑，那么原来的主逻辑要如何恢复呢？对于这一问题，hystrix也为我们实现了自动恢复功能。当断路器打开，对主逻辑进行熔断之后，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的成为主逻辑，当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的主逻辑上，如果此次请求正常返回，那么断路器将继续闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时。

通过上面的一系列机制，hystrix的断路器实现了对依赖资源故障的端口、对降级策略的自动切换以及对主逻辑的自动恢复机制。这使得我们的微服务在依赖外部服务或资源的时候得到了非常好的保护，同时对于一些具备降级逻辑的业务需求可以实现自动化的切换与恢复，相比于设置开关由监控和运维来进行切换的传统实现方式显得更为智能和高效。

## 四、Hystrix监控面板

我们提到断路器是根据一段时间窗内的请求情况来判断并操作断路器的打开和关闭状态的。而这些请求情况的指标信息都是HystrixCommand和HystrixObservableCommand实例在执行过程中记录的重要度量信息，它们除了Hystrix断路器实现中使用之外，对于系统运维也有非常大的帮助。这些指标信息会以“滚动时间窗”与“桶”结合的方式进行汇总，并在内存中驻留一段时间，以供内部或外部进行查询使用，Hystrix Dashboard就是这些指标内容的消费者之一。

下面我们基于之前的示例来结合Hystrix Dashboard实现Hystrix指标数据的可视化面板，这里我们将用到下之前实现的几个应用，包括：

* eureka-server：服务注册中心
* eureka-client：服务提供者
* eureka-consumer-ribbon-hystrix：使用ribbon和hystrix实现的服务消费者

由于eureka-consumer-ribbon-hystrix项目中的/consumer接口实现使用了@HystrixCommand修饰，所以这个接口的调用情况会被Hystrix记录下来，以用来给断路器和Hystrix Dashboard使用。断路器我们在上一篇中已经介绍过了，下面我们来具体说说Hystrix Dashboard的构建。

1.创建项目hystrix-dashboard

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

<dependency>

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

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

</dependency>

2.创建启动类

@EnableHystrixDashboard，启用Hystrix Dashboard功能

@EnableHystrixDashboard

@SpringCloudApplication

public class HystrixDashboardApplication {

public static void main(String[] args) {

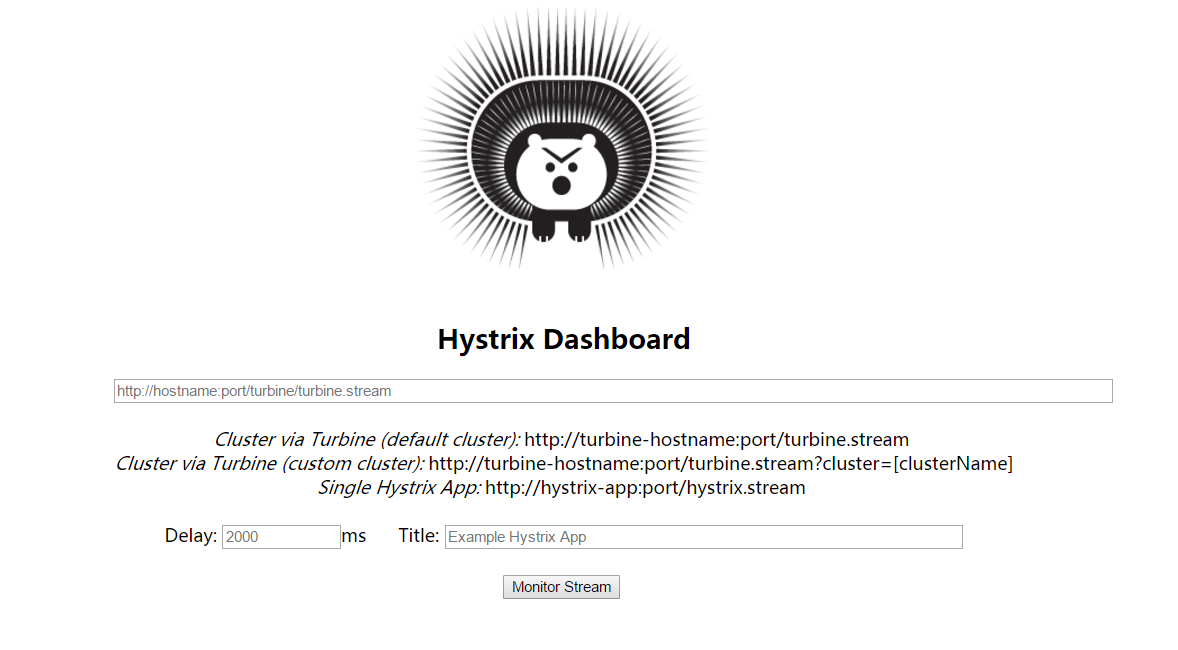
SpringApplication.run(HystrixDashboardApplication.class, args);

}

}

3.启动并访问（配置端口为1301）

http://localhost:1301/hystrix



这是Hystrix Dashboard的监控首页，该页面中并没有具体的监控信息。从页面的文字内容中我们可以知道，Hystrix Dashboard共支持三种不同的监控方式，依次为：

* 默认的集群监控：通过URLhttp://turbine-hostname:port/turbine.stream开启，实现对默认集群的监控。
* 指定的集群监控：通过URLhttp://turbine-hostname:port/turbine.stream?cluster=[clusterName]开启，实现对clusterName集群的监控。
* 单体应用的监控：通过URLhttp://hystrix-app:port/hystrix.stream开启，实现对具体某个服务实例的监控。

前两者都对集群的监控，需要整合Turbine才能实现，这部分内容我们将在下一篇中做详细介绍。在本节中，我们主要实现对单个服务实例的监控，所以这里我们先来实现单个服务实例的监控。

4.创建一个项目

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

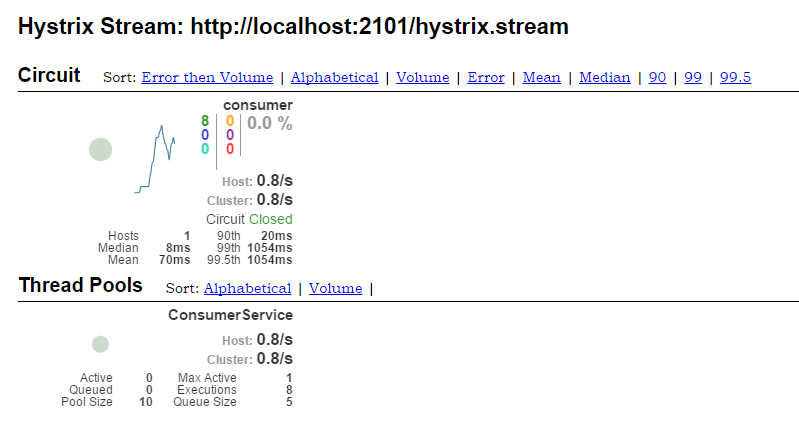
<groupId>org.springframework.boot</groupId>

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

</dependency>

启动类中增加确保在服务实例的主类中已经使用@EnableCircuitBreaker或@EnableHystrix注解，开启了断路器功能。配置端口为2101

到这里已经完成了所有的配置，我们可以在Hystrix Dashboard的首页输入http://localhost:2101/hystrix.stream，已启动对“eureka-consumer-ribbon-hystrix”的监控，点击“Monitor Stream”按钮，此时我们可以看到如下页面：



在对该页面介绍前，我们先看看在首页中我们还没有介绍的两外两个参数：

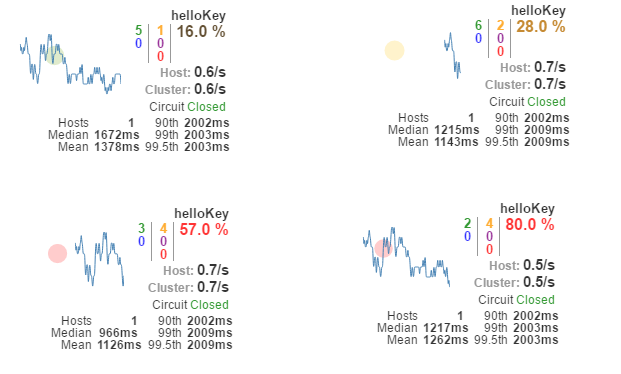
* Delay：该参数用来控制服务器上轮询监控信息的延迟时间，默认为2000毫秒，我们可以通过配置该属性来降低客户端的网络和CPU消耗。
* Title：该参数对应了上图头部标题Hystrix Stream之后的内容，默认会使用具体监控实例的URL，我们可以通过配置该信息来展示更合适的标题。

回到监控页面，我们来详细说说其中各元素的具体含义：

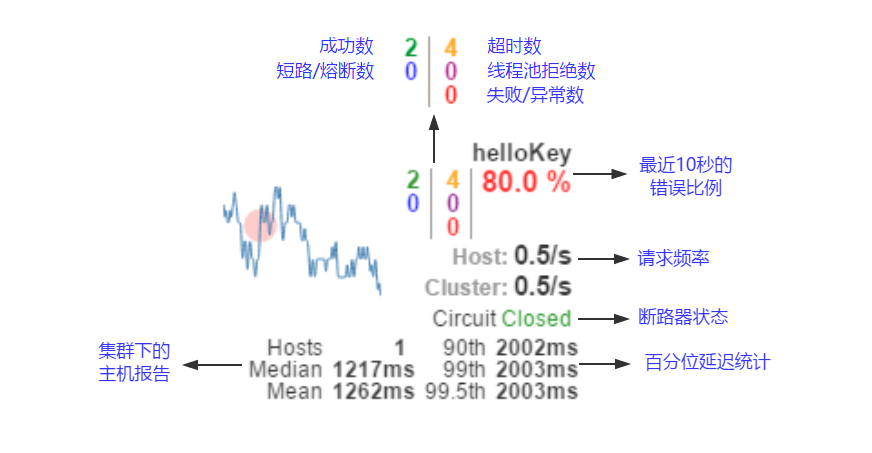
我们可以在监控信息的左上部分找到两个重要的图形信息：一个实心圆和一条曲线。

实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，如下图所示，它的健康度从绿色、黄色、橙色、红色递减。该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，我们就可以在大量的实例中快速的发现故障实例和高压力实例。

曲线：用来记录2分钟内流量的相对变化，我们可以通过它来观察到流量的上升和下降趋势



其他一些数量指标如下图所示

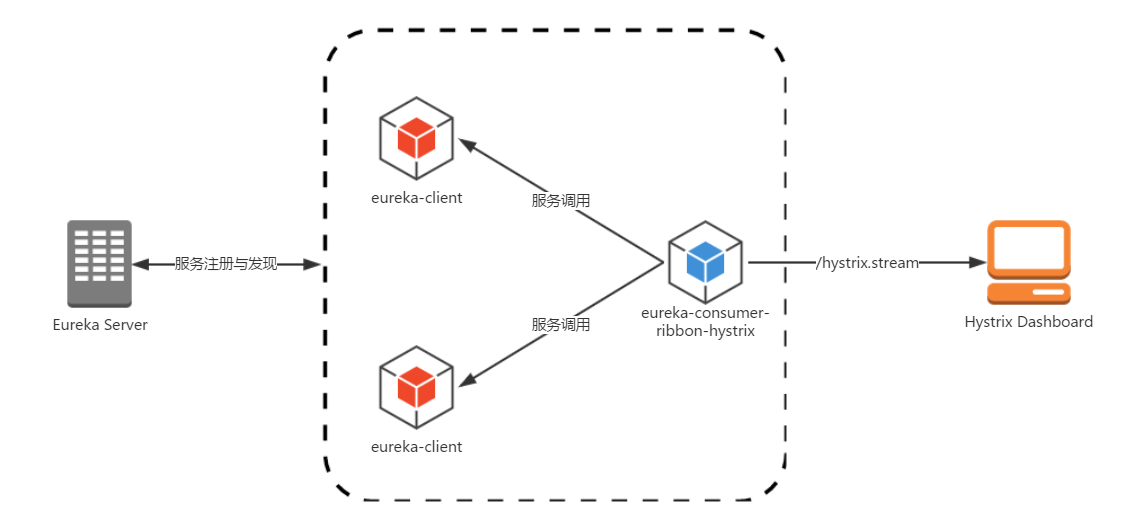


## 五、Hystrix监控数据聚合

Hystrix Dashboard是用来展示Hystrix用于熔断的各项度量指标。通过Hystrix Dashboard，我们可以方便的查看服务实例的综合情况，比如：服务调用次数、服务调用延迟等。但是仅通过Hystrix Dashboard我们只能实现对服务当个实例的数据展现，在生产环境我们的服务是肯定需要做高可用的，那么对于多实例的情况，我们就需要将这些度量指标数据进行聚合。下面，在本篇中，我们就来介绍一下另外一个工具：Turbine。

1.回顾Hystrix Dashboard

服务的架构如下



其中，我们构建的内容包括：

* eureka-server：服务注册中心
* eureka-client：服务提供者
* eureka-consumer-ribbon-hystrix：使用ribbon和hystrix实现的服务消费者
* hystrix-dashboard：用于展示eureka-consumer-ribbon-hystrix服务的Hystrix数据

我们将在上述架构基础上，引入Turbine来对服务的Hystrix数据进行聚合展示。这里我们将分别介绍两种聚合方式。

### HTTP收集聚合

创建项目命名为：turbine

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine</artifactId>

</dependency>

<dependency>

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

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

</dependency>

启动类使用@EnableTurbine注解开启Turbine

|  |
| --- |
| @Configuration @EnableAutoConfiguration @EnableTurbine @EnableDiscoveryClient public class TurbineApplication {  public static void main(String[] args) {  SpringApplication.run(TurbineApplication.class, args);  } } |

配置如下：

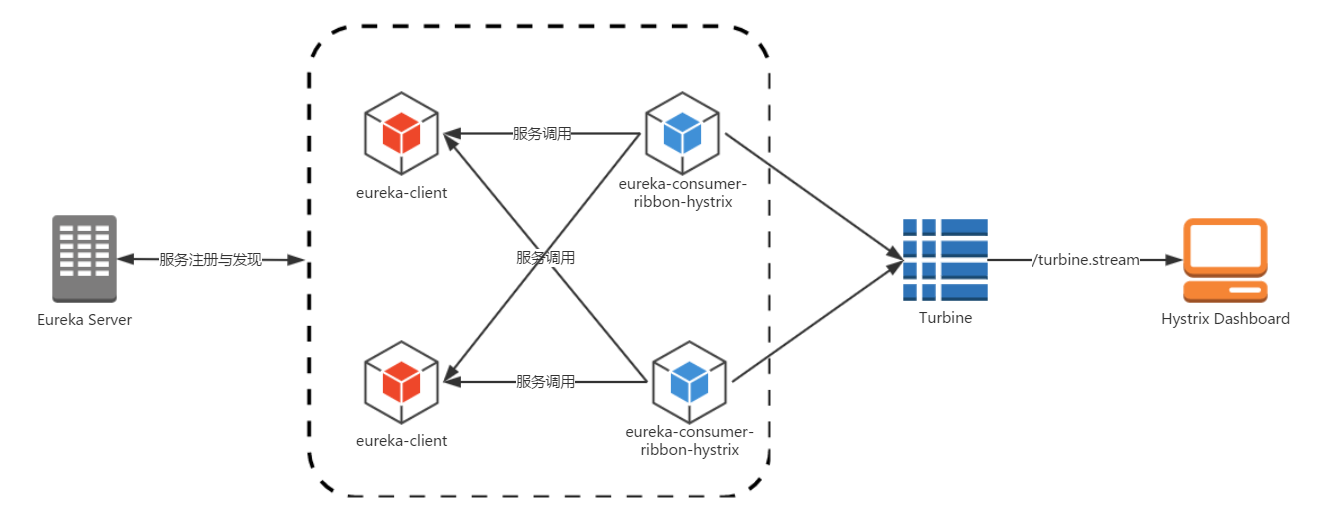
|  |
| --- |
| spring.application.name=turbine  server.port=8989 management.port=8990  eureka.client.serviceUrl.defaultZone=http://localhost:1001/eureka/  turbine.app-config=eureka-consumer-ribbon-hystrix turbine.cluster-name-expression="default" turbine.combine-host-port=true |

参数说明：

* turbine.app-config参数指定了需要收集监控信息的服务名；
* turbine.cluster-name-expression 参数指定了集群名称为default，当我们服务数量非常多的时候，可以启动多个Turbine服务来构建不同的聚合集群，而该参数可以用来区分这些不同的聚合集群，同时该参数值可以在Hystrix仪表盘中用来定位不同的聚合集群，只需要在Hystrix Stream的URL中通过cluster参数来指定；
* turbine.combine-host-port参数设置为true，可以让同一主机上的服务通过主机名与端口号的组合来进行区分，默认情况下会以host来区分不同的服务，这会使得在本地调试的时候，本机上的不同服务聚合成一个服务来统计。

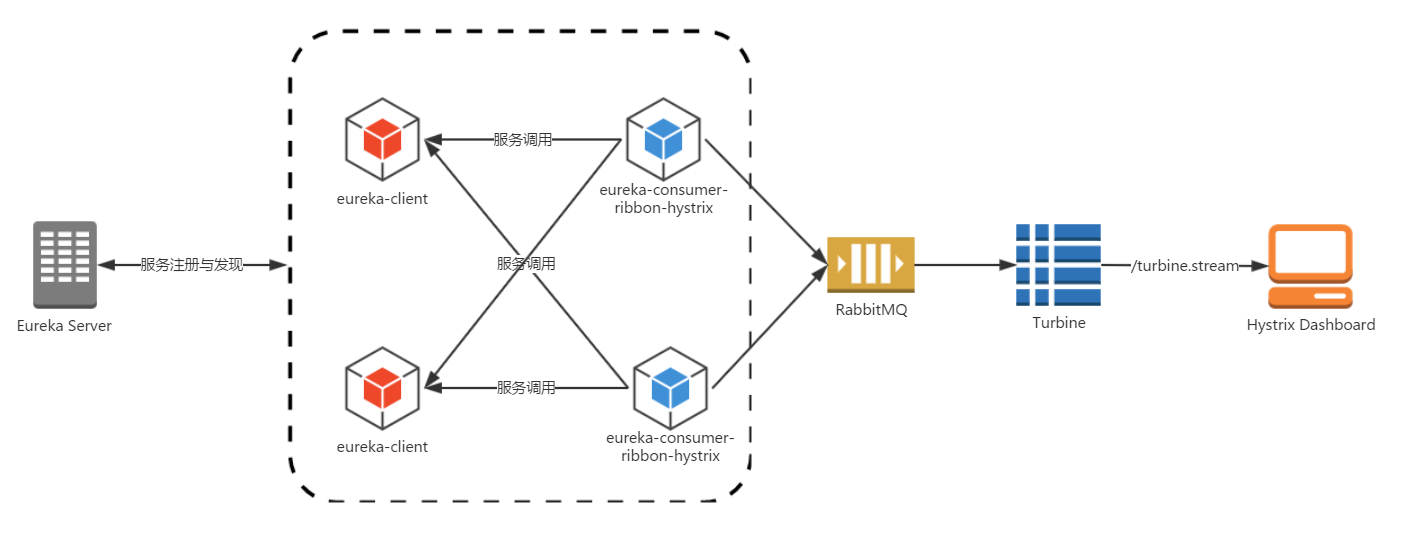
在完成了上面的内容构建之后，我们来体验一下Turbine对集群的监控能力。分别启动eureka-server、eureka-client、eureka-consumer-ribbon-hystrix、turbine以及hystrix-dashboard。访问Hystrix Dashboard，并开启对http://localhost:8989/turbine.stream`的监控，这时候，我们将看到针对服务`eureka-consumer-ribbon-hystrix`的聚合监控数据。

而此时的架构如下图所示：



### 消息代理收集聚合

Spring Cloud在封装Turbine的时候，还实现了基于消息代理的收集实现。所以，我们可以将所有需要收集的监控信息都输出到消息代理中，然后Turbine服务再从消息代理中异步的获取这些监控信息，最后将这些监控信息聚合并输出到Hystrix Dashboard中。通过引入消息代理，我们的Turbine和Hystrix Dashoard实现的监控架构可以改成如下图所示的结构：



创建项目命名为：turbine-amqp

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine-amqp</artifactId>

</dependency>

<dependency>

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

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

</dependency>

启动类使用@EnableTurbineStream注解开启Turbine Stream

|  |
| --- |
| @Configuration @EnableAutoConfiguration @EnableTurbineStream @EnableDiscoveryClient public class TurbineApplication {  public static void main(String[] args) {  SpringApplication.run(TurbineApplication.class, args);  } } |

配置如下：

|  |
| --- |
| spring.application.name=turbine-amqp  server.port=8989 management.port=8990  eureka.client.serviceUrl.defaultZone=http://localhost:1001/eureka/ |

修改消费者

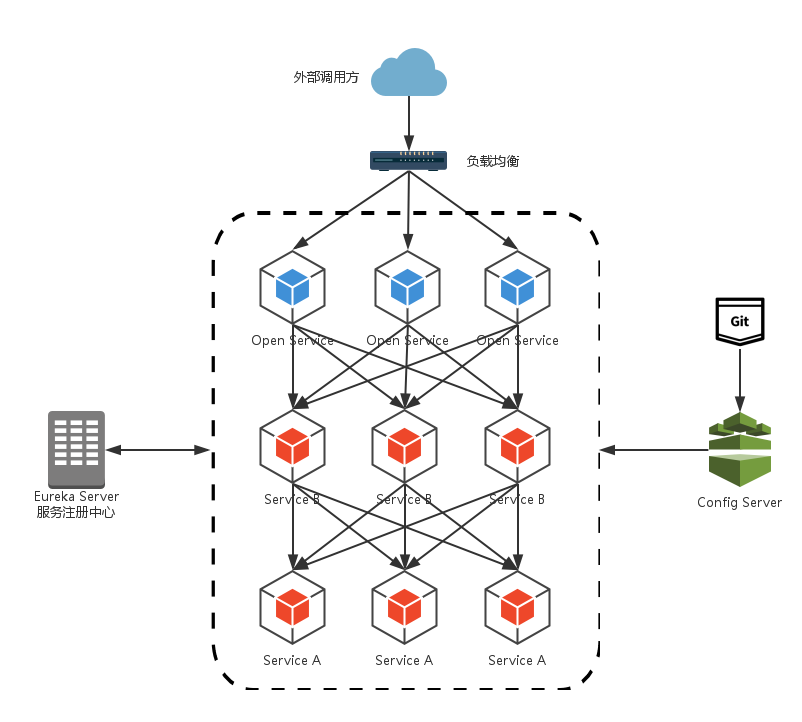
对于Turbine的配置已经完成了，下面我们需要对服务消费者eureka-consumer-ribbon-hystrix做一些修改，使其监控信息能够输出到RabbitMQ上。这个修改也非常简单，只需要在pom.xml中增加对spring-cloud-netflix-hystrix-amqp依赖，具体如下：

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-netflix-hystrix-amqp</artifactId> </dependency> |

在完成了上面的配置之后，我们可以继续之前的所有项目（除turbine以外），并通过Hystrix Dashboard开启对http://localhost:8989/turbine.stream的监控，我们可以获得如之前实现的同样效果，只是这里我们的监控信息收集时是通过了消息代理异步实现的。

# 第六章 服务网关

通过以上5章内容我们已经可以构建一个简略的（不够完善）微服务架构了。比如下图所示：



我们使用Spring Cloud Netflix中的Eureka实现了服务注册中心以及服务注册与发现；而服务间通过Ribbon或Feign实现服务的消费以及均衡负载；通过Spring Cloud Config实现了应用多环境的外部化配置以及版本管理。为了使得服务集群更为健壮，使用Hystrix的融断机制来避免在微服务架构中个别服务出现异常时引起的故障蔓延。

在该架构中，我们的服务集群包含：内部服务Service A和Service B，他们都会注册与订阅服务至Eureka Server，而Open Service是一个对外的服务，通过均衡负载公开至服务调用方。本文我们把焦点聚集在对外服务这块，这样的实现是否合理，或者是否有更好的实现方式呢？

先来说说这样架构需要做的一些事儿以及存在的不足：

* 首先，破坏了服务无状态特点。为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。
* 其次，无法直接复用既有接口。当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。

面对类似上面的问题，我们要如何解决呢？下面进入本文的正题：服务网关！

为了解决上面这些问题，我们需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器，它就是本文将来介绍的：服务网关。

服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix中的Zuul就担任了这样的一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。

下面我们通过实例例子来使用一下Zuul来作为服务的路由功能。

## 一、简单使用

1. 准备

使用前面使用的eureka-client和eureka-consumer作为生产者和消费者

2. 创建网关服务

项目名：api-gateway

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency> </dependencies> |

3. 创建启动类

使用@EnableZuulProxy注解开启Zuul的功能

|  |
| --- |
| @EnableZuulProxy @SpringCloudApplication public class Application {  public static void main(String[] args) {  new SpringApplicationBuilder(Application.class).web(true).run(args);  } } |

4. 配置

|  |
| --- |
| spring:  application:  name: api-gateway server:  port: 1101 eureka:  client:  serviceUrl:  defaultZone: http://eureka.didispace.com/eureka/ |

1. 启动并访问

到这里，一个基于Spring Cloud Zuul服务网关就已经构建完毕。启动该应用，一个默认的服务网关就构建完毕了。由于Spring Cloud Zuul在整合了Eureka之后，具备默认的服务路由功能，即：当我们这里构建的api-gateway应用启动并注册到eureka之后，服务网关会发现上面我们启动的两个服务eureka-client和eureka-consumer，这时候Zuul就会创建两个路由规则。每个路由规则都包含两部分，一部分是外部请求的匹配规则，另一部分是路由的服务ID。针对当前示例的情况，Zuul会创建下面的两个路由规则：

转发到eureka-client服务的请求规则为：/eureka-client/\*\*

转发到eureka-consumer服务的请求规则为：/eureka-consumer/\*\*

最后，我们可以通过访问1101端口的服务网关来验证上述路由的正确性：

比如访问：http://localhost:1101/eureka-client/dc ，该请求将最终被路由到eureka-client的/dc接口上。

## 二、路由配置

### 1.传统路由配置

所谓的传统路由配置方式就是在不依赖于服务发现机制的情况下，通过在配置文件中具体指定每个路由表达式与服务实例的映射关系来实现API网关对外部请求的路由。

没有Eureka和Consul的服务治理框架帮助的时候，我们需要根据服务实例的数量采用不同方式的配置来实现路由规则：

单实例配置：通过一组zuul.routes.<route>.path与zuul.routes.<route>.url参数对的方式配置，比如：

|  |
| --- |
| zuul.routes.user-service.path=/user-service/\*\* zuul.routes.user-service.url=http://localhost:8080/ |

该配置实现了对符合/user-service/\*\*规则的请求路径转发到http://localhost:8080/地址的路由规则，比如，当有一个请求http://localhost:1101/user-service/hello被发送到API网关上，由于/user-service/hello能够被上述配置的path规则匹配，所以API网关会转发请求到http://localhost:8080/hello地址。

多实例配置：通过一组zuul.routes.<route>.path与zuul.routes.<route>.serviceId参数对的方式配置，比如：

|  |
| --- |
| zuul.routes.user-service.path=/user-service/\*\* zuul.routes.user-service.serviceId=user-service ribbon.eureka.enabled=false user-service.ribbon.listOfServers=http://localhost:8080/,http://localhost:8081/ |

该配置实现了对符合/user-service/\*\*规则的请求路径转发到http://localhost:8080/和http://localhost:8081/两个实例地址的路由规则。它的配置方式与服务路由的配置方式一样，都采用了zuul.routes.<route>.path与zuul.routes.<route>.serviceId参数对的映射方式，只是这里的serviceId是由用户手工命名的服务名称，配合<serviceId>.ribbon.listOfServers参数实现服务与实例的维护。由于存在多个实例，API网关在进行路由转发时需要实现负载均衡策略，于是这里还需要Spring Cloud Ribbon的配合。由于在Spring Cloud Zuul中自带了对Ribbon的依赖，所以我们只需要做一些配置即可，比如上面示例中关于Ribbon的各个配置，它们的具体作用如下：

* ribbon.eureka.enabled：由于zuul.routes.<route>.serviceId指定的是服务名称，默认情况下Ribbon会根据服务发现机制来获取配置服务名对应的实例清单。但是，该示例并没有整合类似Eureka之类的服务治理框架，所以需要将该参数设置为false，不然配置的serviceId是获取不到对应实例清单的。
* user-service.ribbon.listOfServers：该参数内容与zuul.routes.<route>.serviceId的配置相对应，开头的user-service对应了serviceId的值，这两个参数的配置相当于在该应用内部手工维护了服务与实例的对应关系。
* 不论是单实例还是多实例的配置方式，我们都需要为每一对映射关系指定一个名称，也就是上面配置中的<route>，每一个<route>就对应了一条路由规则。每条路由规则都需要通过path属性来定义一个用来匹配客户端请求的路径表达式，并通过url或serviceId属性来指定请求表达式映射具体实例地址或服务名。

### 2.服务路由配置

服务路由我们在上一篇中也已经有过基础的介绍和体验，Spring Cloud Zuul通过与Spring Cloud Eureka的整合，实现了对服务实例的自动化维护，所以在使用服务路由配置的时候，我们不需要向传统路由配置方式那样为serviceId去指定具体的服务实例地址，只需要通过一组zuul.routes.<route>.path与zuul.routes.<route>.serviceId参数对的方式配置即可。

比如下面的示例，它实现了对符合/user-service/\*\*规则的请求路径转发到名为user-service的服务实例上去的路由规则。其中<route>可以指定为任意的路由名称。

|  |
| --- |
| zuul.routes.user-service.path=/user-service/\*\* zuul.routes.user-service.serviceId=user-service |

对于面向服务的路由配置，除了使用path与serviceId映射的配置方式之外，还有一种更简洁的配置方式：zuul.routes.<serviceId>=<path>，其中<serviceId>用来指定路由的具体服务名，<path>用来配置匹配的请求表达式。比如下面的例子，它的路由规则等价于上面通过path与serviceId组合使用的配置方式。

|  |
| --- |
| zuul.routes.user-service=/user-service/\*\* |

传统路由的映射方式比较直观且容易理解，API网关直接根据请求的URL路径找到最匹配的path表达式，直接转发给该表达式对应的url或对应serviceId下配置的实例地址，以实现外部请求的路由。那么当采用path与serviceId以服务路由方式实现时候，没有配置任何实例地址的情况下，外部请求经过API网关的时候，它是如何被解析并转发到服务具体实例的呢？

在Spring Cloud Netflix中，Zuul巧妙的整合了Eureka来实现面向服务的路由。实际上，我们可以直接将API网关也看做是Eureka服务治理下的一个普通微服务应用。它除了会将自己注册到Eureka服务注册中心上之外，也会从注册中心获取所有服务以及它们的实例清单。所以，在Eureka的帮助下，API网关服务本身就已经维护了系统中所有serviceId与实例地址的映射关系。当有外部请求到达API网关的时候，根据请求的URL路径找到最佳匹配的path规则，API网关就可以知道要将该请求路由到哪个具体的serviceId上去。由于在API网关中已经知道serviceId对应服务实例的地址清单，那么只需要通过Ribbon的负载均衡策略，直接在这些清单中选择一个具体的实例进行转发就能完成路由工作了。

## 三、过滤器

### 1.过滤器的作用

通过上面所述的两篇我们，我们已经能够实现请求的路由功能，所以我们的微服务应用提供的接口就可以通过统一的API网关入口被客户端访问到了。但是，每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都需要有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而，目前的服务路由并没有限制权限这样的功能，所有请求都会被毫无保留地转发到具体的应用并返回结果，为了实现对客户端请求的安全校验和权限控制，最简单和粗暴的方法就是为每个微服务应用都实现一套用于校验签名和鉴别权限的过滤器或拦截器。不过，这样的做法并不可取，它会增加日后的系统维护难度，因为同一个系统中的各种校验逻辑很多情况下都是大致相同或类似的，这样的实现方式会使得相似的校验逻辑代码被分散到了各个微服务中去，冗余代码的出现是我们不希望看到的。所以，比较好的做法是将这些校验逻辑剥离出去，构建出一个独立的鉴权服务。在完成了剥离之后，有不少开发者会直接在微服务应用中通过调用鉴权服务来实现校验，但是这样的做法仅仅只是解决了鉴权逻辑的分离，并没有在本质上将这部分不属于业余的逻辑拆分出原有的微服务应用，冗余的拦截器或过滤器依然会存在。

对于这样的问题，更好的做法是通过前置的网关服务来完成这些非业务性质的校验。由于网关服务的加入，外部客户端访问我们的系统已经有了统一入口，既然这些校验与具体业务无关，那何不在请求到达的时候就完成校验和过滤，而不是转发后再过滤而导致更长的请求延迟。同时，通过在网关中完成校验和过滤，微服务应用端就可以去除各种复杂的过滤器和拦截器了，这使得微服务应用的接口开发和测试复杂度也得到了相应的降低。

为了在API网关中实现对客户端请求的校验，我们将需要使用到Spring Cloud Zuul的另外一个核心功能：过滤器。

Zuul允许开发者在API网关上通过定义过滤器来实现对请求的拦截与过滤，实现的方法非常简单，我们只需要继承ZuulFilter抽象类并实现它定义的四个抽象函数就可以完成对请求的拦截和过滤了。

### 2.过滤器的实现

比如下面的代码，我们定义了一个简单的Zuul过滤器，它实现了在请求被路由之前检查HttpServletRequest中是否有accessToken参数，若有就进行路由，若没有就拒绝访问，返回401 Unauthorized错误。

|  |
| --- |
| public class AccessFilter extends ZuulFilter {   private static Logger log = LoggerFactory.getLogger(AccessFilter.class);   @Override  public String filterType() {  return "pre";  }   @Override  public int filterOrder() {  return 0;  }   @Override  public boolean shouldFilter() {  return true;  }   @Override  public Object run() {  RequestContext ctx = RequestContext.getCurrentContext();  HttpServletRequest request = ctx.getRequest();   log.info("send {} request to {}", request.getMethod(), request.getRequestURL().toString());   Object accessToken = request.getParameter("accessToken");  if(accessToken == null) {  log.warn("access token is empty");  ctx.setSendZuulResponse(false);  ctx.setResponseStatusCode(401);  return null;  }  log.info("access token ok");  return null;  } } |

在上面实现的过滤器代码中，我们通过继承ZuulFilter抽象类并重写了下面的四个方法来实现自定义的过滤器。这四个方法分别定义了：

* filterType：过滤器的类型，它决定过滤器在请求的哪个生命周期中执行。这里定义为pre，代表会在请求被路由之前执行。
* filterOrder：过滤器的执行顺序。当请求在一个阶段中存在多个过滤器时，需要根据该方法返回的值来依次执行。
* shouldFilter：判断该过滤器是否需要被执行。这里我们直接返回了true，因此该过滤器对所有请求都会生效。实际运用中我们可以利用该函数来指定过滤器的有效范围。
* run：过滤器的具体逻辑。这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回的错误码，当然我们也可以进一步优化我们的返回，比如，通过ctx.setResponseBody(body)对返回body内容进行编辑等。

在实现了自定义过滤器之后，它并不会直接生效，我们还需要为其创建具体的Bean才能启动该过滤器，比如，在应用主类中增加如下内容：

|  |
| --- |
| @EnableZuulProxy @SpringCloudApplication public class Application {   public static void main(String[] args) {  new SpringApplicationBuilder(Application.class).web(true).run(args);  }   @Bean  public AccessFilter accessFilter() {  return new AccessFilter();  } } |

在对api-gateway服务完成了上面的改造之后，我们可以重新启动它，并发起下面的请求，对上面定义的过滤器做一个验证：

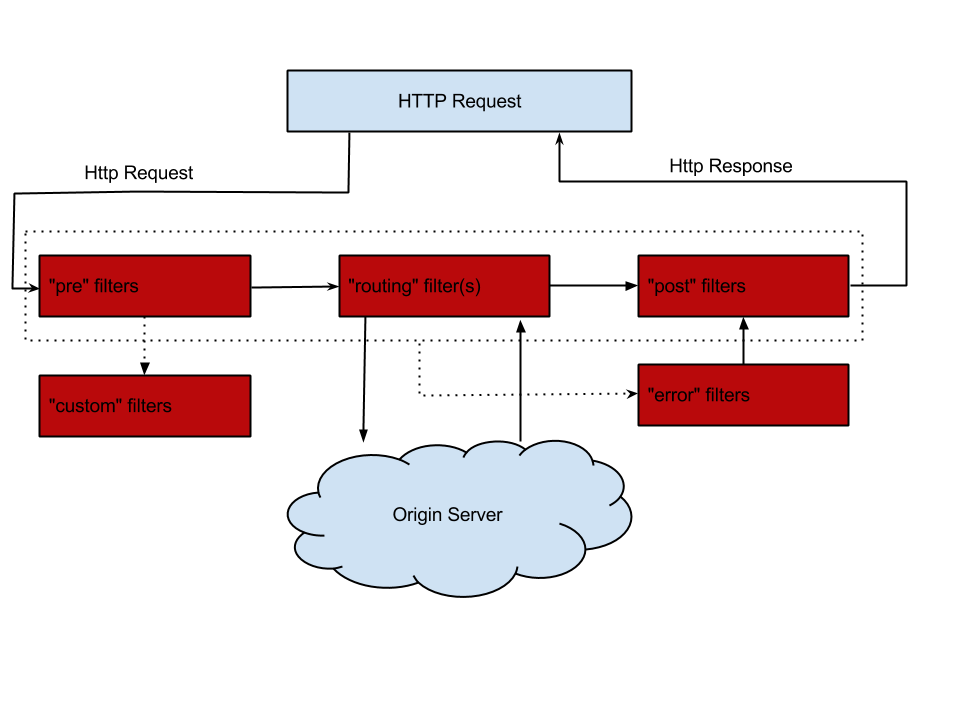
http://localhost:1101/api-a/hello：返回401错误

http://localhost:1101/api-a/hello&accessToken=token：正确路由到hello-service的/hello接口，并返回Hello World

到这里，对于Spring Cloud Zuul过滤器的基本功能就以介绍完毕。读者可以根据自己的需要在服务网关上定义一些与业务无关的通用逻辑实现对请求的过滤和拦截，比如：签名校验、权限校验、请求限流等功能。

### 3.生命周期

对于其他一些过滤类型，这里就不一一展开了，根据之前对filterType生命周期介绍，可以参考下图去理解，并根据自己的需要在不同的生命周期中去实现不同类型的过滤器。



最后，总结一下为什么服务网关是微服务架构的重要部分，是我们必须要去做的原因：

不仅仅实现了路由功能来屏蔽诸多服务细节，更实现了服务级别、均衡负载的路由。

实现了接口权限校验与微服务业务逻辑的解耦。通过服务网关中的过滤器，在各生命周期中去校验请求的内容，将原本在对外服务层做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理。

实现了断路器，不会因为具体微服务的故障而导致服务网关的阻塞，依然可以对外服务。

# 第七章 消息总线

在之前的Spring Cloud Config的介绍中，我们还留了一个悬念：如何实现对配置信息的实时更新。虽然，我们已经能够通过/refresh接口和Git仓库的Web Hook来实现Git仓库中的内容修改触发应用程序的属性更新。但是，若所有触发操作均需要我们手工去维护Web Hook中的应用位置的话，这随着系统的不断扩张，会变的越来越难以维护，而消息代理中间件是解决该问题最为合适的方案。是否还记得我们在介绍消息代理中的特点时有提到过这样一个功能：消息代理中间件可以将消息路由到一个或多个目的地。利用这个功能，我们就能完美的解决该问题，下面我们来说说Spring Cloud Bus中的具体实现方案。

## 一、RabbitMQ实现。

1. 使用上面已经创建好的项目。

config-server-eureka和config-client-eureka

在config-client-eureka，提供一个/from接口，它会获取config-repo/didispace-dev.properties中的from属性返回

2. 并加入依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId> </dependency> |

3. 配置

|  |
| --- |
| spring.rabbitmq.host=localhost spring.rabbitmq.port=5672 spring.rabbitmq.username=springcloud spring.rabbitmq.password=123456 |

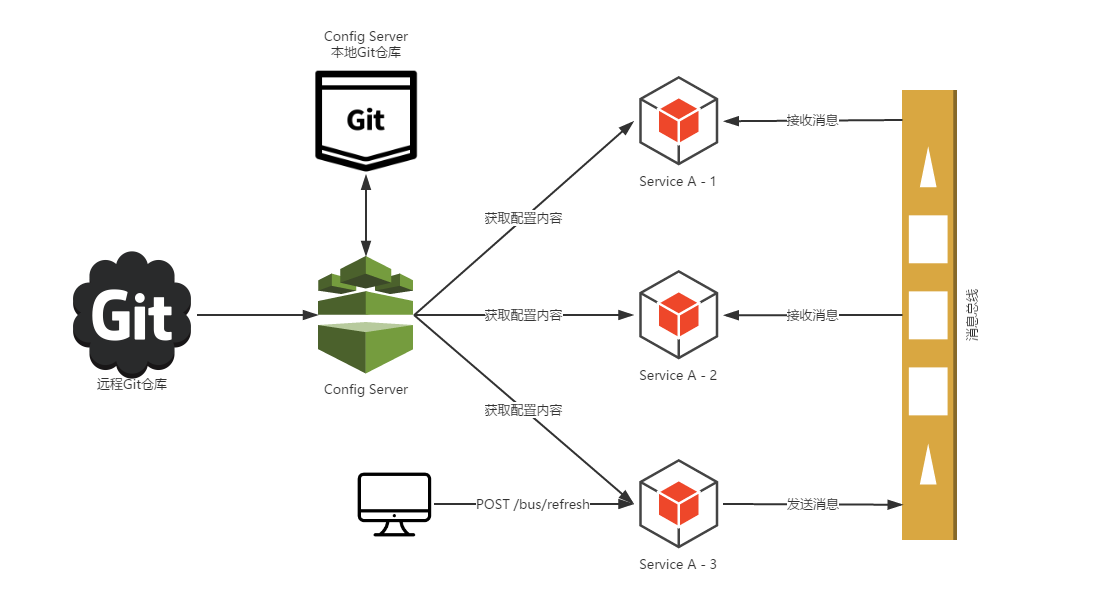
4. 启动config-server-eureka，再启动两个config-client-eureka（分别在不同的端口上，比如7002、7003），我们可以在config-client-eureka中的控制台中看到如下内容，在启动时候，客户端程序多了一个/bus/refresh请求。

* 先访问两个config-client-eureka的/from请求，会返回当前config-repo/didispace-dev.properties中的from属性。
* 接着，我们修改config-repo/didispace-dev.properties中的from属性值，并发送POST请求到其中的一个/bus/refresh。
* 最后，我们再分别访问启动的两个config-client-eureka的/from请求，此时这两个请求都会返回最新的config-repo/didispace-dev.properties中的from属性。

到这里，我们已经能够通过Spring Cloud Bus来实时更新总线上的属性配置了。

5. 原理

我们通过使用Spring Cloud Bus与Spring Cloud Config的整合，并以RabbitMQ作为消息代理，实现了应用配置的动态更新。

整个方案的架构如上图所示，其中包含了Git仓库、Config Server、以及微服务“Service A”的三个实例，这三个实例中都引入了Spring Cloud Bus，所以他们都连接到了RabbitMQ的消息总线上。

当我们将系统启动起来之后，“Service A”的三个实例会请求Config Server以获取配置信息，Config Server根据应用配置的规则从Git仓库中获取配置信息并返回。

此时，若我们需要修改“Service A”的属性。首先，通过Git管理工具去仓库中修改对应的属性值，但是这个修改并不会触发“Service A”实例的属性更新。我们向“Service A”的实例3发送POST请求，访问/bus/refresh接口。此时，“Service A”的实例3就会将刷新请求发送到消息总线中，该消息事件会被“Service A”的实例1和实例2从总线中获取到，并重新从Config Server中获取他们的配置信息，从而实现配置信息的动态更新。

而从Git仓库中配置的修改到发起/bus/refresh的POST请求这一步可以通过Git仓库的Web Hook来自动触发。由于所有连接到消息总线上的应用都会接受到更新请求，所以在Web Hook中就不需要维护所有节点内容来进行更新，从而解决了通过Web Hook来逐个进行刷新的问题。

1. 指定刷新范围

上面的例子中，我们通过向服务实例请求Spring Cloud Bus的/bus/refresh接口，从而触发总线上其他服务实例的/refresh。但是有些特殊场景下（比如：灰度发布），我们希望可以刷新微服务中某个具体实例的配置。

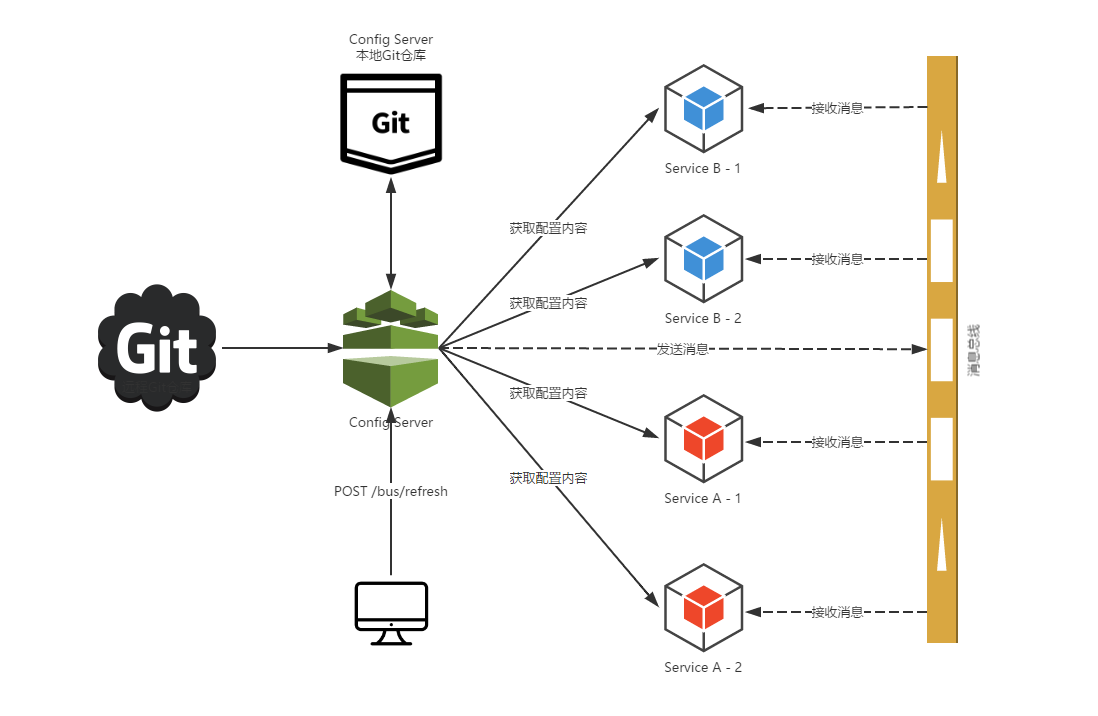
Spring Cloud Bus对这种场景也有很好的支持：/bus/refresh接口还提供了destination参数，用来定位具体要刷新的应用程序。比如，我们可以请求/bus/refresh?destination=customers:9000，此时总线上的各应用实例会根据destination属性的值来判断是否为自己的实例名，若符合才进行配置刷新，若不符合就忽略该消息。

destination参数除了可以定位具体的实例之外，还可以用来定位具体的服务。定位服务的原理是通过使用Spring的PathMatecher（路径匹配）来实现，比如：/bus/refresh?destination=customers:\*\*，该请求会触发customers服务的所有实例进行刷新。

7.架构优化

既然Spring Cloud Bus的/bus/refresh接口提供了针对服务和实例进行配置更新的参数，那么我们的架构也相应的可以做出一些调整。在之前的架构中，服务的配置更新需要通过向具体服务中的某个实例发送请求，再触发对整个服务集群的配置更新。虽然能实现功能，但是这样的结果是，我们指定的应用实例就会不同于集群中的其他应用实例，这样会增加集群内部的复杂度，不利于将来的运维工作，比如：我们需要对服务实例进行迁移，那么我们不得不修改Web Hook中的配置等。所以我们要尽可能的让服务集群中的各个节点是对等的。

因此，我们将之前的架构做了一些调整，如下图所示



我们主要做了这些改动：

* 在Config Server中也引入Spring Cloud Bus，将配置服务端也加入到消息总线中来。
* /bus/refresh请求不在发送到具体服务实例上，而是发送给Config Server，并通过destination参数来指定需要更新配置的服务或实例。

通过上面的改动，我们的服务实例就不需要再承担触发配置更新的职责。同时，对于Git的触发等配置都只需要针对Config Server即可，从而简化了集群上的一些维护工作。

## 二、Kafka实现

Kafka是一个由LinkedIn开发的分布式消息系统，它于2011年初开源，现在由著名的Apache基金会维护与开发。Kafka使用Scala实现，被用作LinkedIn的活动流和运营数据处理的管道，现在也被诸多互联网企业广泛地用作为数据流管道和消息系统。

Kafka是基于消息发布/订阅模式实现的消息系统，其主要设计目标如下：

* 消息持久化：以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间复杂度的访问性能。
* 高吞吐：在廉价的商用机器上也能支持单机每秒100K条以上的吞吐量
* 分布式：支持消息分区以及分布式消费，并保证分区内的消息顺序
* 跨平台：支持不同技术平台的客户端（如：Java、PHP、Python等）
* 实时性：支持实时数据处理和离线数据处理
* 伸缩性：支持水平扩展

Kafka中涉及的一些基本概念：

* Broker：Kafka集群包含一个或多个服务器，这些服务器被称为Broker。
* Topic：逻辑上同Rabbit的Queue队列相似，每条发布到Kafka集群的消息都必须有一个Topic。（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个Broker上，但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处）
* Partition：Partition是物理概念上的分区，为了提供系统吞吐率，在物理上每个Topic会分成一个或多个Partition，每个Partition对应一个文件夹（存储对应分区的消息内容和索引文件）。
* Producer：消息生产者，负责生产消息并发送到Kafka Broker。
* Consumer：消息消费者，向Kafka Broker读取消息并处理的客户端。
* Consumer Group：每个Consumer属于一个特定的组（可为每个Consumer指定属于一个组，若不指定则属于默认组），组可以用来实现一条消息被组内多个成员消费等功能。

1.使用rabblit的config-client-eureka项目，将spring-cloud-starter-bus-amqp替换成spring-cloud-starter-bus-kafka模块

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-kafka</artifactId> </dependency> |

2.启动

使用命令：kafka-topics --list --zookeeper localhost:2181。查看当前Kafka中的Topic，若已成功启动了config-server并配置正确，我们就可以在Kafka中看到已经多了一个名为springCloudBus的Topic。

3. Kafka配置

在上面的例子中，由于Kafka、ZooKeeper均运行于本地，所以我们没有在测试程序中通过配置信息来指定Kafka和ZooKeeper的配置信息，就完成了本地消息总线的试验。但是我们实际应用中，Kafka和ZooKeeper一般都会独立部署，所以在应用中都需要来为Kafka和ZooKeeper配置一些连接信息等。Kafka的整合与RabbitMQ不同，在Spring Boot 1.3.7中并没有直接提供的Starter模块，而是采用了Spring Cloud Stream的Kafka模块，所以对于Kafka的配置均采用了spring.cloud.stream.kafka的前缀，比如：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | 属性名 | 说明 | 默认值 | | --- | --- | --- | | | 属性名 | 说明 | 默认值 | | --- | --- | --- | | | 属性名 | 说明 | 默认值 | | --- | --- | --- | |
| spring.cloud.stream.kafka.binder.brokers | Kafka的服务端列表 | localhost |
| spring.cloud.stream.kafka.binder.defaultBrokerPort | Kafka服务端的默认端口，当brokers属性中没有配置端口信息时，就会使用这个默认端口 | 9092 |
| spring.cloud.stream.kafka.binder.zkNodes | Kafka服务端连接的ZooKeeper节点列表 | localhost |
| spring.cloud.stream.kafka.binder.defaultZkPort | ZooKeeper节点的默认端口，当zkNodes属性中没有配置端口信息时，就会使用这个默认端口 | 2181 |