### 零、微服务

传统的单体系统随着需求不断扩大，变得越来越臃肿，一个小的改造就可能导致其他功能受影响，维护成本越来越大。Martin Fowler首先提出了“微服务”的概念，微服务是系统架构上的一种设计风格，它的主旨是将一个原来独立的系统拆分成多个小型服务，这些小型服务可以在各自独立的进程中运行，服务间通过Http的RESTful API进行通信协作。每个服务内部是耦合度较高的业务功能集合，可以维护自己的数据存储、业务开发、测试以及独立部署。

挑战

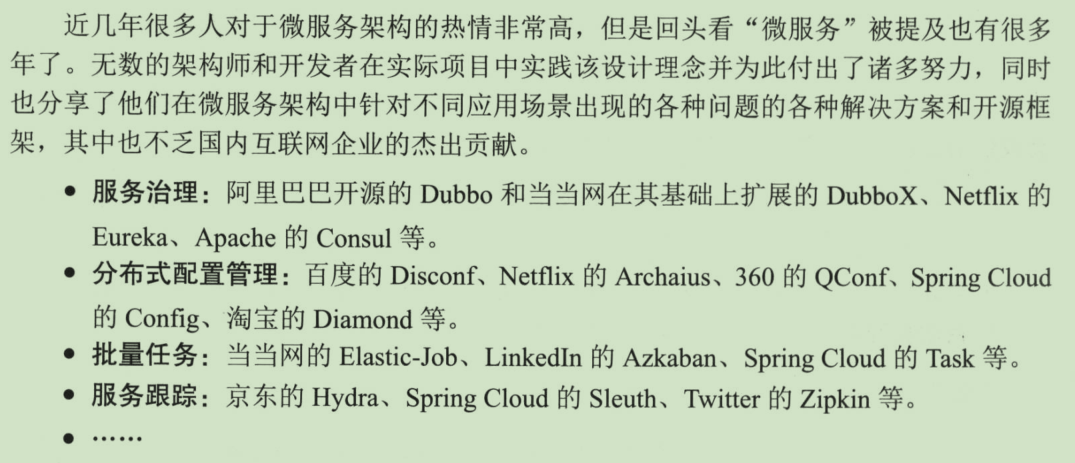
运维：部署需要更加自动化；

接口一致性：不同服务之间的调用关系，接口定义；

分布式的复杂性：要考虑网络延迟、分布式事务、异步消息；

微服务需具备特性

服务组件化



### 一、Spring Cloud介绍

Spring Cloud不像前面列举的框架，只是解决微服务中的某一个问题，而是提供了一个解决微服务机构的综合性解决框架，通过一个“全家桶”Spring Cloud可以让开发者快速实现出分布式系统中的通用功能，如配置管理，服务发现，断路器，智能路由等。因为分布式系统有很多样板模式。

Spring Cloud提供了以下几种机制：

* Distributed/versioned configuration分布式参数配置
* Service registration and discovery服务注册/发现
* Routing路由
* Service-to-service calls端对端调用
* Load balancing负载均衡
* Circuit Breakers断路器：将受保护的服务封装在一个可以监控故障的断路器对象中，当故障达到一定门限，断路器将跳闸（trip），所有后继调用将不会发往受保护的服务而由断路器对象之间返回错误。对于需要更长时间解决的故障问题，不断重试就没有太大意义了，可以使用断路器模式。([Netflix/hystrix](https://github.com/Netflix/hystrix))
* Distributed messaging分布式消息通信

版本(根据对应Boot版本，应该先对Spring Boot有所了解)，选择Release版本

Finchley == Spring Boot 2.0.x,  ~~1.5.x.~~

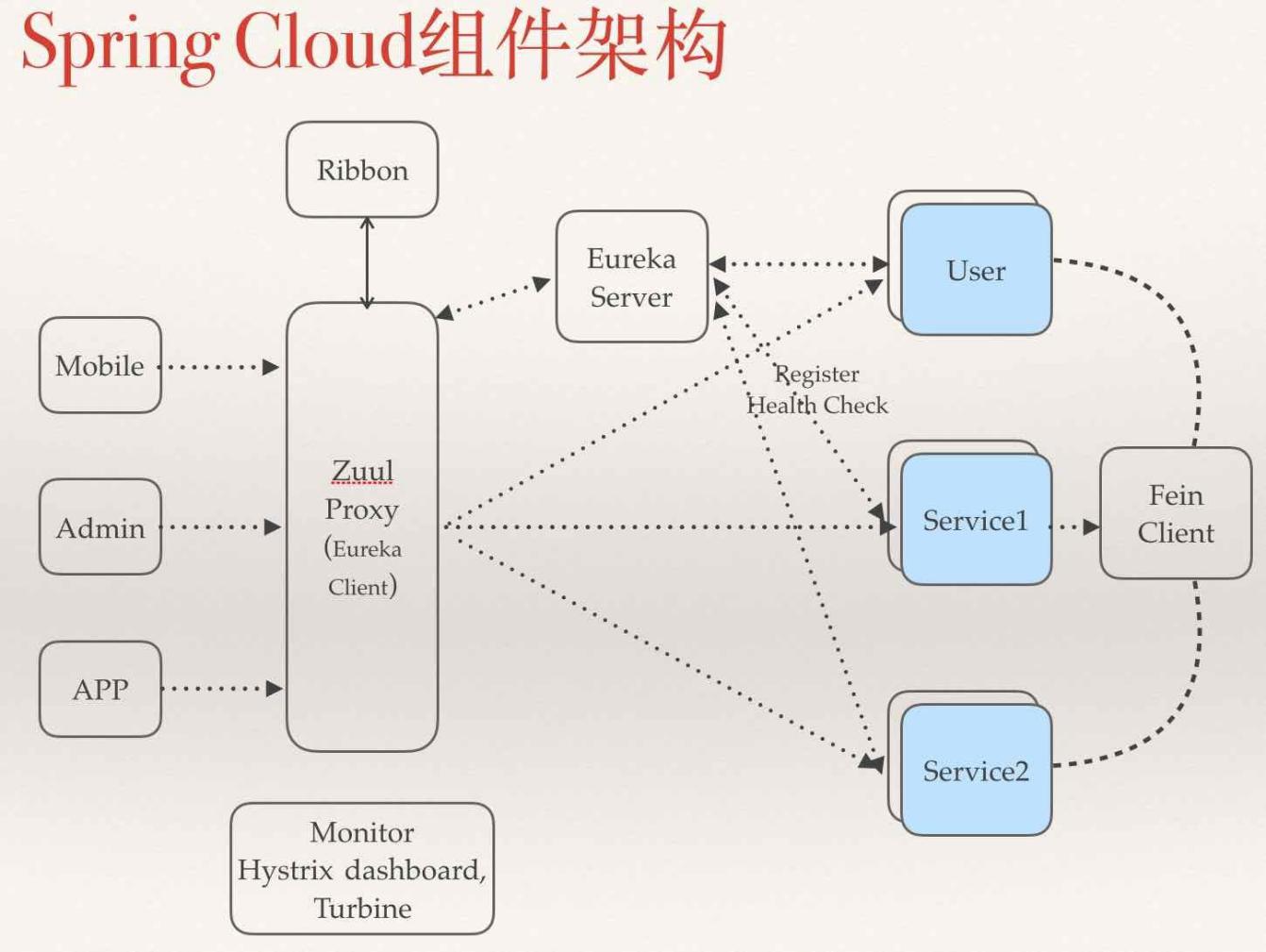
Dalston & Edgware == Spring Boot 1.5.x,  ~~2.0.x.~~，暂选Edgware

Camden == Spring Boot 1.4.x, /1.5.x.

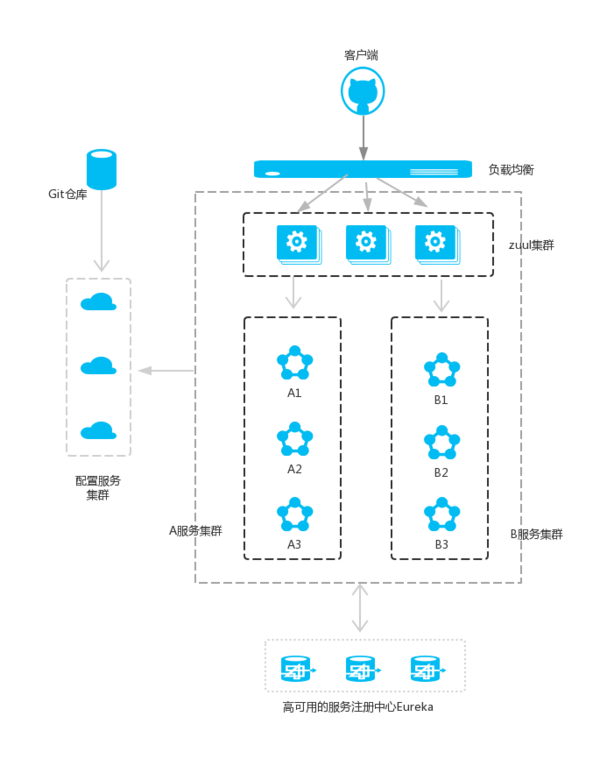
~~Brixton & Angel~~ (in July 2017)

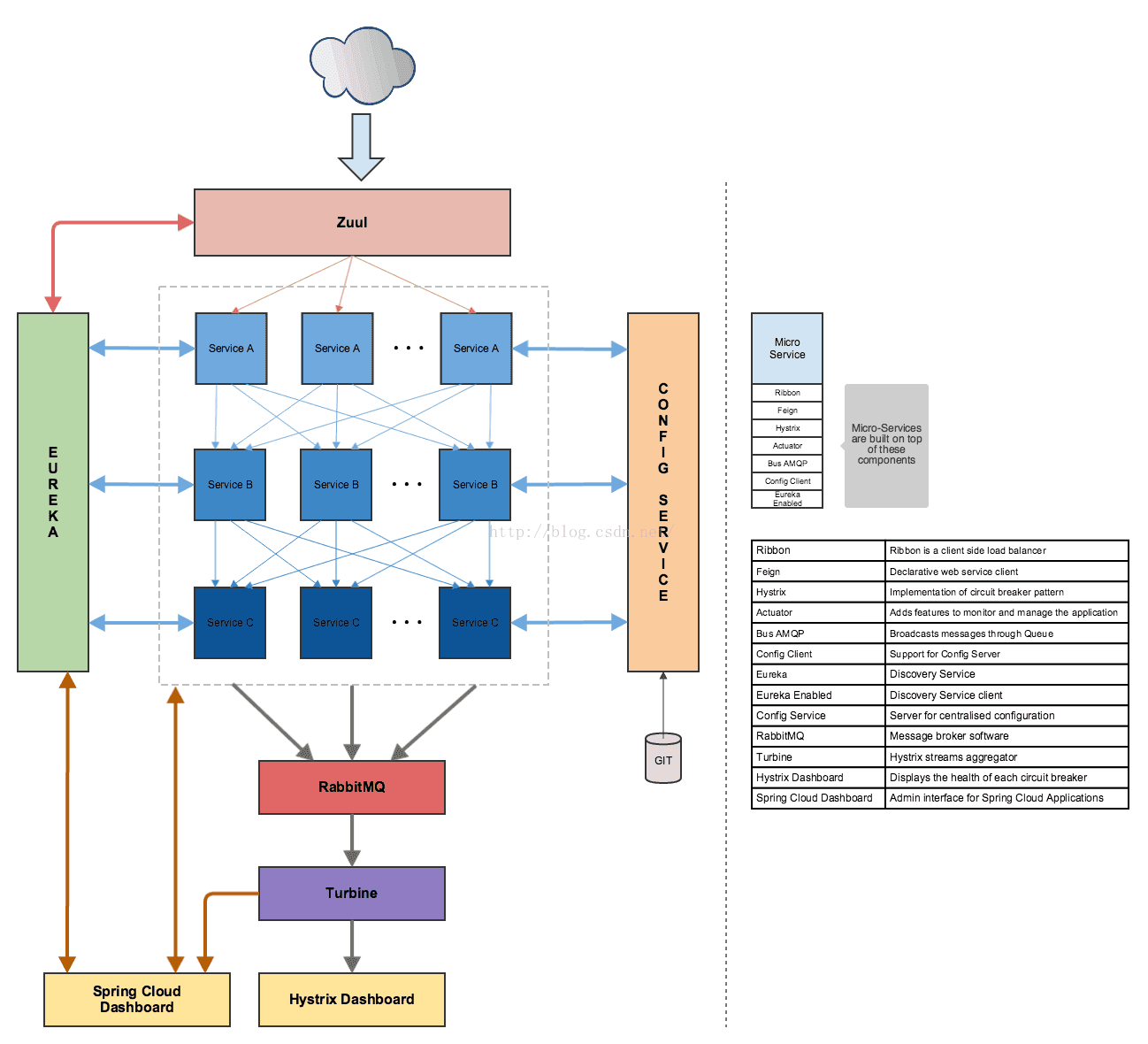
Brixton == Spring Boot 1.3.x, /1.4.x.,~~1.2.x~~

Angel == Spring Boot 1.2.x, ~~1.3.x~~.



在微服务架构中，需要几个基础的服务治理组件，包括**服务注册与发现Eureka**、**服务消费Feign**、**负载均衡Ribbon**、**断路器Hystrix**、**智能路由Zuul**、**配置管理Config**等，由这几个基础组件相互协作，共同组建了一个简单的微服务系统。

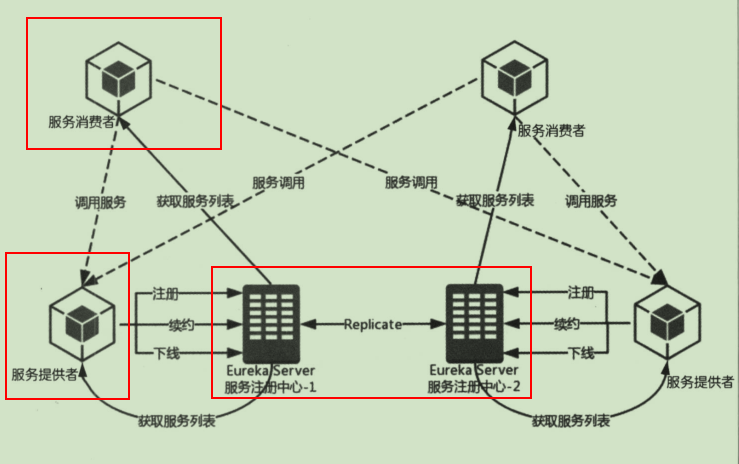




### 二、服务注册（Eureka）

#### 1 简介

服务治理是微服务架构中最核心和基础的模块，使用Spring Cloud Netflix的Eureka 实现服务注册和发现模块。Eureka服务治理三个要素：服务注册中心、服务提供者、服务消费者



#### 2 创建一个服务注册中心 eureka-server(:8761/eureka/)

一个Eureka-Server应该实现（）：

注册

下面以创建server为例子，详细说明创建过程：

创建完后的工程的pom.xml文件如下**(注意这里springboot-1.5.3,springcloud版本Edgware)**：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-eureka-server</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  **<version>1.5.3.RELEASE</version>**  </parent>  <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>Edgware.RELEASE</version>**  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

2.3 启动一个服务注册中心，只需要一个注解@EnableEurekaServer，这个注解需要在springboot工程的启动application类上加：

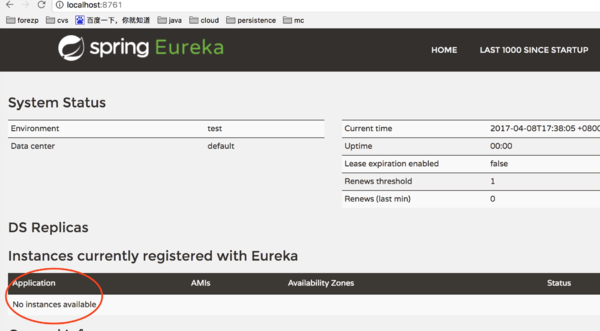
|  |
| --- |
| **@EnableEurekaServer**  @SpringBootApplication  public class EurekaserverApplication {  public static void main(String[] args) {  SpringApplication.run(EurekaserverApplication.class, args);  }  } |

eureka是一个高可用的组件，它没有后端缓存，每一个实例注册之后需要向注册中心发送心跳（因此可以在内存中完成），在默认情况下erureka server也是一个eureka client ,必须要指定一个 server。eureka-server的配置文件/src/main/resources/appication.yml：

|  |
| --- |
| server:  port: 8761  eureka:  instance:  hostname: localhost  client:  registerWithEureka: false  fetchRegistry: false  serviceUrl:  **defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/** |

通过eureka.client.registerWithEureka：false和fetchRegistry：false来表明自己是一个eureka server.

2.5 eureka server 是有界面的，启动工程,打开浏览器访问：   
[http://localhost:8761](http://localhost:8761/) ,界面如下：



#### 3 创建一个服务提供者 eureka-client(:8762/hi)

服务提供者作为一个Eureka-client需要实现：

注册，发送元数据rest

续约，定时（30s）发送心跳rest

获取ServiceUrl，定时（）查询rest

当client向server注册时，它会提供一些元数据，例如主机和端口，URL，主页等。Eureka server 从每个client实例接收心跳消息。 如果心跳超时，则通常将该实例从注册server中删除。

创建过程同server类似,创建完pom.xml如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-eureka-client</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <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>  <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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

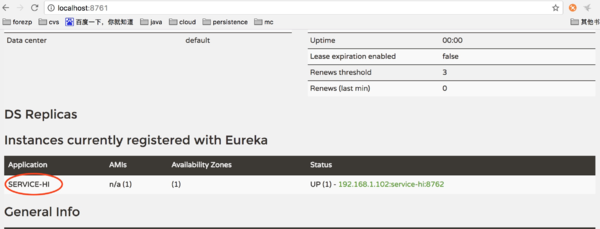
通过注解@EnableEurekaClient 表明自己是一个eurekaclient.

|  |
| --- |
| @SpringBootApplication  **@EnableEurekaClient**  @RestController  public class ServiceHiApplication {  public static void main(String[] args) {  SpringApplication.run(ServiceHiApplication.class, args);  }  @Value("${server.port}")  String port;  @RequestMapping("/hi")  public String home(@RequestParam String name) {  return "hi "+name+",i am from port:" +port;  }  } |

仅仅@EnableEurekaClient是不够的，还需要在配置文件中注明自己的服务注册中心的地址，application.yml配置文件如下：

|  |
| --- |
| eureka:  client:  serviceUrl:  **defaultZone: http://localhost:8761/eureka/**  server:  port: 8762  spring:  application:  name: service-hi |

需要指明**spring.application.name**,这个很重要，这在以后的服务与服务之间相互调用一般都是根据这个name 。   
启动工程，打开[http://localhost:8761](http://localhost:8761/) ，即eureka server 的网址：



你会发现一个服务已经注册在服务中了，服务名为SERVICE-HI ,端口为8762

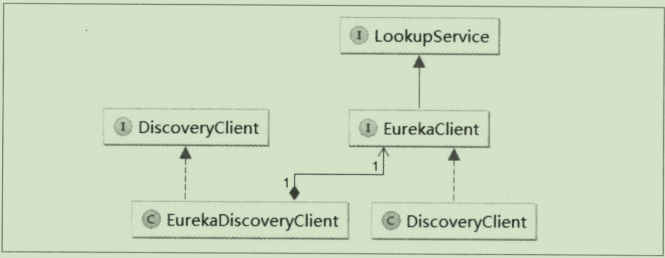
这时打开 http://localhost:8762/hi?name=abc，你会在浏览器上看到 :

hi abc,i am from port:8762

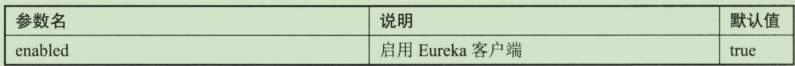
参考资料

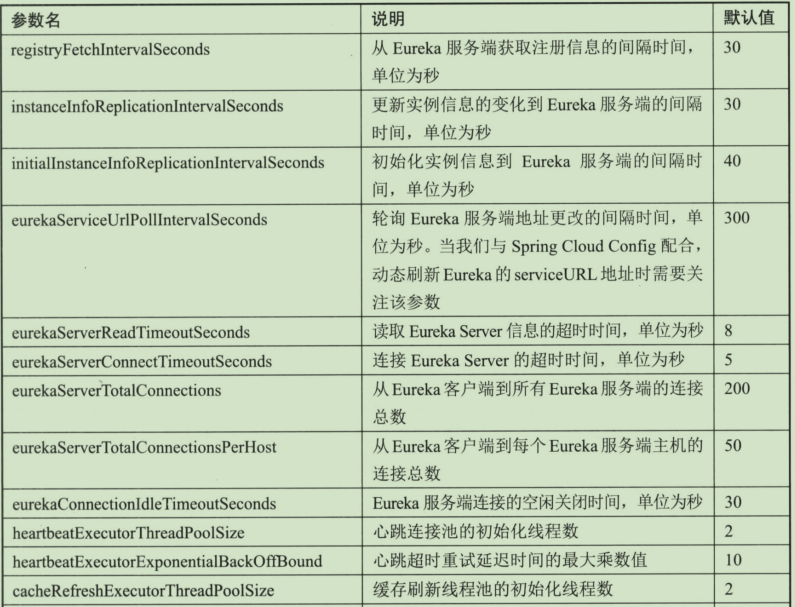
springcloud eureka server

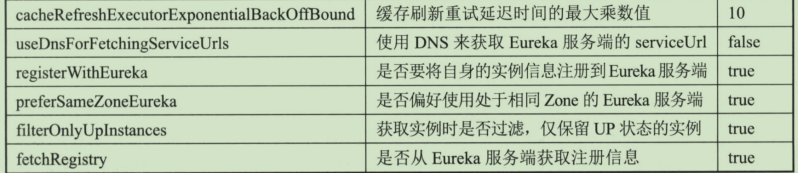
springcloud eureka client



eureka.client参数配置



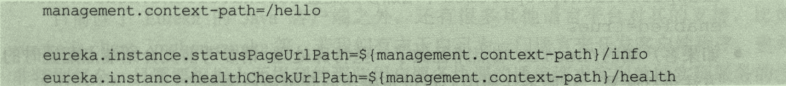


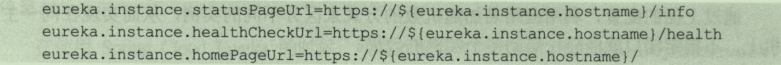


实例名配置（默认ip:服务名:主机名:端口）



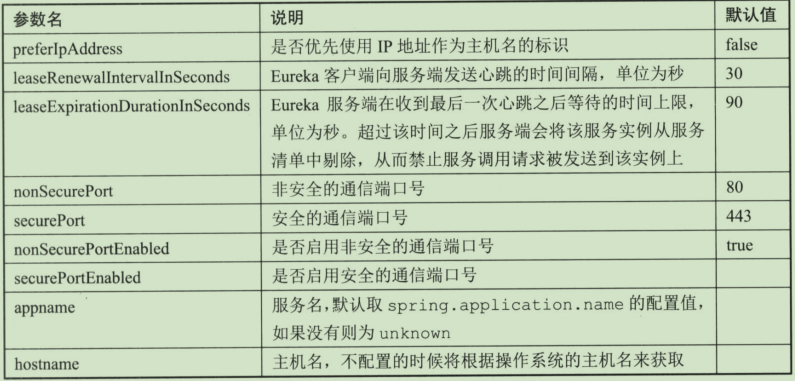
端点配置（actuator模块）定制满足条件



健康检测（使用actuator模块）

引入spring-boot-starter-actuator依赖，application.properties配置eureka.client.healthcheck.enabled=true，默认检测/health端点。

eureka.instance参数配置



自我保护模式背景：

首先Eureka注册中心需要了解的是Eureka各个节点都是平等的，没有ZK中角色的概念，即使N-1个节点挂掉也不会影响其他节点的正常运行。

默认情况下，如果Eureka Server在一定时间内（默认90秒）没有收到某个微服务实例的心跳，Eureka Server将会移除该实例。但是当网络分区故障发生时，微服务与Eureka Server之间无法正常通信，而微服务本身是正常运行的，此时不应该移除这个微服务，所以引入了自我保护机制。自我保护机制的工作机制是如果在15分钟内超过85%的客户端节点都没有正常的心跳，那么Eureka就会认为客户端与注册中心出现了网络故障，Eureka Server自动进入自我保护机制，此时会出现以下几种情况：

1. Eureka Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。
2. Eureka Server仍然能够接受新服务的注册和查询请求，但是不会同步到其他节点上，保证当前节点仍然可用。
3. 当网络稳定时，当前Eureka Server新的注册信息会同步到其他节点中。

因此，Eureka Server可用很好的应对因网络故障导致部分节点失联的情况，而不像ZK那样，如果有一半不可用的情况会导致整个集群不可用而变成瘫痪。

Eureka的自我保护机制，通过eureka.server.enable-self-preservation来true打开/false禁用自我保护机制，默认是打开状态，建议生产环境打开此配置。

开发环境配置：

开发环境中如果要实现服务失效能自动移除，只需要修改一下配置：

1. 注册中心关闭自我保护机制，修改检查失效服务的时间。

eureka:

server:

enable-self-preservation:false

eviction-interval-timer-in-ms: 3000

1. 微服务修改减短服务心跳的时间。

#默认90秒

lease-expiration-duration-in-seconds: 10

#默认30秒

lease-renewal-interval-in-seconds: 3

以上配置建议在生产环境使用默认的时间配置。

### 三、服务消费者（ribbon）

在微服务架构中，业务都会被拆分成一个独立的服务，服务与服务的通讯是基于http restful的。Spring cloud有两种服务调用方式，一种是ribbon+restTemplate，另一种是feign（集成了ribbon）。

#### 1 ribbon简介

ribbon是一个客户端负载均衡（区别于服务端软硬件负载均衡F5/Nginx），**每个客户端维护自己要访问的服务端清单**，这些清单来自服务注册中心（eureka-server），ribbon 已经默认实现了这些配置bean：

IClientConfig ribbonClientConfig: DefaultClientConfigImpl

IRule ribbonRule: **ZoneAvoidanceRule（负载均衡规则）（轮询，随机，最佳，重试，过滤）**

IPing ribbonPing: NoOpPing（检测实例健康状态）

ServerList ribbonServerList: ConfigurationBasedServerList

ServerListFilter ribbonServerListFilter: ZonePreferenceServerListFilter

ILoadBalancer ribbonLoadBalancer: **ZoneAwareLoadBalancer（负载均衡器）看源码**

#### 2 准备工作

基于上面的工程，启动eureka-server 工程；启动service-hi工程，它的端口为8762；将service-hi的配置文件的端口改为8763,并启动，这时你会发现：service-hi在eureka-server注册了2个实例，这就相当于一个小的集群。访问localhost:8761如图所示：



#### 3 建一个服务消费者(service-ribbon:8764/hi)

重新新建一个spring-boot工程，取名为：service-ribbon;   
在它的pom.xml文件分别引入起步依赖spring-cloud-starter-eureka、spring-cloud-starter-ribbon、spring-boot-starter-web，代码如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-ribbon-client</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-eureka</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-ribbon</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在工程的配置文件指定服务的注册中心地址为<http://localhost:8761/eureka/>，程序名称为 service-ribbon，程序端口为8764。配置文件application.yml如下：

|  |
| --- |
| eureka:  client:  serviceUrl:  **defaultZone: http://localhost:8761/eureka/**  server:  port: 8764  spring:  application:  name: service-ribbon |

在工程的启动类中,通过@EnableDiscoveryClient向服务中心注册；并且向程序的ioc注入一个bean: restTemplate;并通过@LoadBalanced注解表明这个restRemplate开启负载均衡的功能。

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

写一个测试类HelloService，通过之前注入ioc容器的restTemplate来消费service-hi服务的“/hi”接口，在这里我们直接用的程序名替代了具体的url地址，在ribbon中它会根据服务名来选择具体的服务实例，根据服务实例在请求的时候会用具体的url替换掉服务名，代码如下：

|  |
| --- |
| @Service  public class HelloService {  @Autowired  **RestTemplate restTemplate;**  public String hiService(String name) {  //restTemplate.getForEntity(url,class,arg...):ResponseEntity  return restTemplate.getForObject(**"http://SERVICE-HI/hi?name="**+name,String.class);  }  GET请求  restTemplate.getForEntity(url,class,arg...):ResponseEntity  restTemplate.getForObject(url,class,arg...):T;  POST请求  restTemplate.postForEntity();  restTemplate.postForObject();  restTemplate.post  PUT请求  restTemplate.put()  DELETE请求  restTemplate.delete()  } |

写一个controller，在controller中用调用HelloService 的方法，代码如下：

|  |
| --- |
| @RestController  public class HelloControler {  @Autowired  HelloService helloService;  @RequestMapping(value = "/hi")  public String hi(@RequestParam String name){  return helloService.hiService(name);  }  } |

在浏览器上多次访问http://localhost:8764/hi?name=abc，浏览器交替显示：

hi abc,i am from port:8762

hi abc,i am from port:8763

这说明当我们通过调用restTemplate.getForObject(“[http://SERVICE-HI/hi?name=](http://service-hi/hi?name=" \t "http://blog.csdn.net/forezp/article/details/_blank)“+name,String.class)方法时，已经做了负载均衡，访问了不同的端口的服务实例。

此时的架构



一个服务注册中心，eureka server,端口为8761

service-hi工程跑了两个实例，端口分别为8762,8763，分别向服务注册中心注册

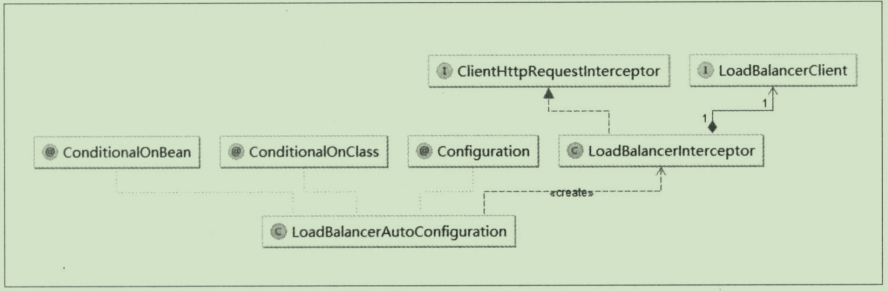
sercvice-ribbon端口为8764,向服务注册中心注册

当sercvice-ribbon通过restTemplate调用service-hi的hi接口时，因为用ribbon进行了负载均衡，会轮流的调用service-hi：8762和8763 两个端口的hi接口；

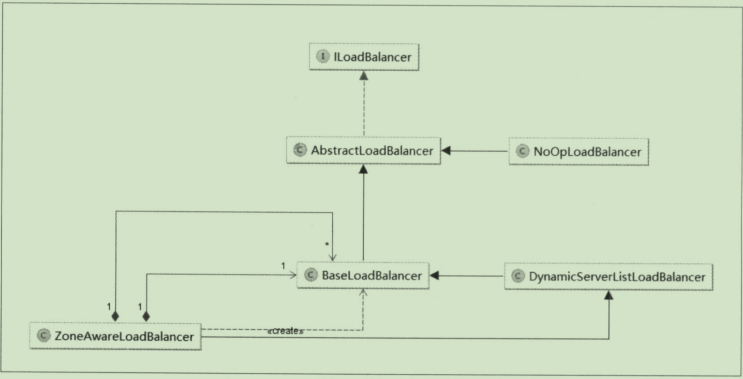
参考资料

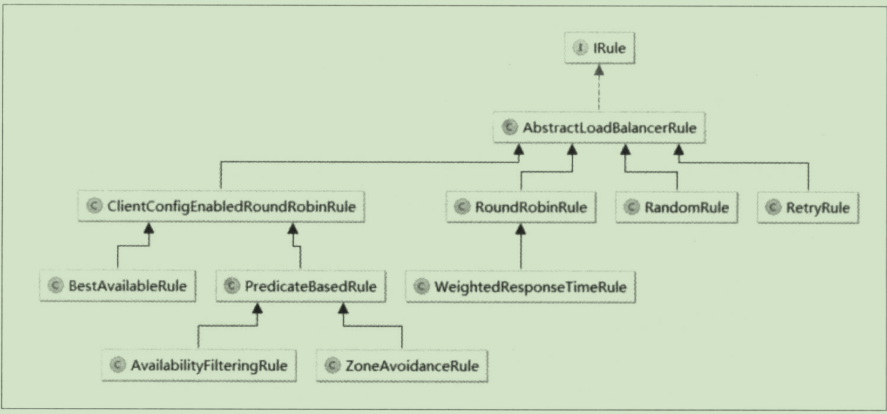
[spring-cloud-ribbon](http://projects.spring.io/spring-cloud/spring-cloud.html#spring-cloud-ribbon)

[springcloud ribbon with eureka](http://blog.csdn.net/liaokailin/article/details/51469834)



通过@LoadBalanced注解给restTemplate加上拦截器，当有url请求，拦截器根据url的服务名host由ILoadBalancer（默认ZoneAwareLoadBalancer）负载均衡算法从清单选择具体的实例，组成实际的请求URL（host:port）





### 四、服务消费者（Feign）

#### 1 简介

Feign是一个声明式的伪Http客户端，它使得写Http客户端变得更简单。使用Feign，只需要创建一个接口并注解。它具有可插拔的注解特性，可使用Feign 注解和JAX-RS注解。Feign支持可插拔的编码器和解码器。Feign默认集成了Ribbon，并和Eureka结合，默认实现了负载均衡的效果。

#### 2 准备工作

启动上面的eureka-server，端口为8761; 启动service-hi 两次，端口分别为8762 、8763.

#### 3 创建一个feign的服务(service-feign:8765/hi)

新建一个spring-boot工程，取名为serice-feign，在它的pom文件引入Feign的依赖spring-cloud-starter-feign、Eureka的依赖spring-cloud-starter-eureka、Web的依赖spring-boot-starter-web，代码如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-feign-client</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-eureka</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-feign</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在工程的配置文件application.yml文件，指定程序名为service-feign，端口号为8765，服务注册地址为<http://localhost:8761/eureka/> ，代码如下：

|  |
| --- |
| eureka:  client:  serviceUrl:  **defaultZone: http://localhost:8761/eureka/**  server:  port: 8765  spring:  application:  name: service-feign |

在程序的启动类ServiceFeignApplication ，加上@EnableFeignClients注解开启Feign的功能：

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

定义一个feign接口，通过@ FeignClient（“服务名”），来指定调用哪个服务。比如在代码中调用了service-hi服务的“/hi”接口，代码如下：

|  |
| --- |
| **@FeignClient(value = "service-hi")**  public interface SchedualServiceHi {  **@RequestMapping(value = "/hi",**method = RequestMethod.GET)  String sayHiFromClientOne(@RequestParam(value = "name") String name);  } |

在Web层的controller层，对外暴露一个”/hi”的API接口，通过上面定义的Feign客户端SchedualServiceHi 来消费服务。代码如下：

|  |
| --- |
| @RestController  public class HiController {  @Autowired  SchedualServiceHi schedualServiceHi;  @RequestMapping(value = "/hi",method = RequestMethod.GET)  public String sayHi(@RequestParam String name){  return schedualServiceHi.sayHiFromClientOne(name);  }  } |

启动程序，多次访问[http://localhost:8765/hi?name=a](http://localhost:8765/hi?name=forezp)bc,浏览器交替显示：

hi abc,i am from port:8762

hi abc,i am from port:8763

参考资料

[spring-cloud-feign](http://projects.spring.io/spring-cloud/spring-cloud.html#spring-cloud-feign)

### 五、断路器（Hystrix）

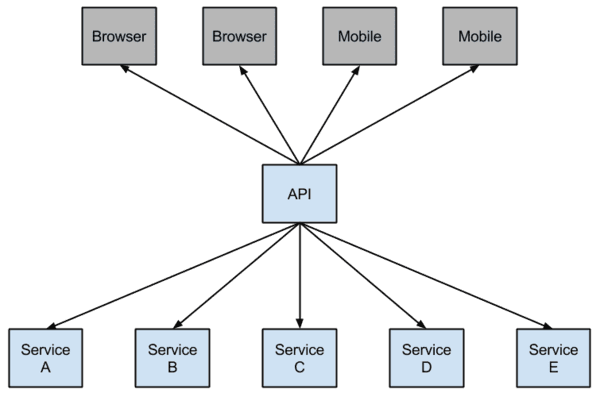
为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。为了解决这个问题，业界提出了**断路器模型**。

#### 1 断路器简介

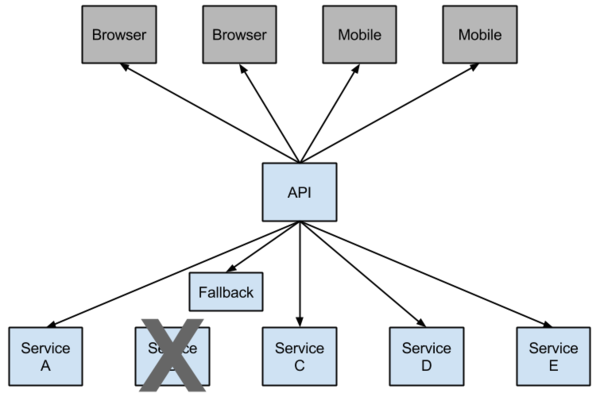
Netflix has created a library called Hystrix that implements the circuit breaker pattern. In a microservice architecture it is common to have multiple layers of service calls.

--摘自官网

Netflix开源了Hystrix组件，实现了断路器模式，SpringCloud对这一组件进行了整合。 在微服务架构中，一个请求需要调用多个服务是非常常见的，如下图：



较底层的服务如果出现故障，会导致连锁故障。当对特定的服务的调用的不可用达到一个阀值（Hystric 是5秒20次） 断路器将会被打开。



断路打开后，可用避免连锁故障，fallback方法可以直接返回一个固定值。

#### 2 准备工作

启动上面eureka-server 工程；启动service-hi工程，它的端口为8762。

#### 3 在ribbon使用断路器(service-ribbon:8764/hi)

改造serice-ribbon 工程的代码，首先在pox.xml文件中加入spring-cloud-starter-hystrix的依赖：

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

在程序的启动类ServiceRibbonApplication 加@EnableHystrix注解开启Hystrix：

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

改造HelloService类，在hiService方法上加上@HystrixCommand注解。该注解对该方法创建了熔断器的功能，并指定了fallbackMethod熔断方法，熔断方法直接返回了一个字符串，字符串为”hi,”+name+”,sorry,error!”，代码如下：

|  |
| --- |
| @Service  public class HelloService {  @Autowired  RestTemplate restTemplate;  **@HystrixCommand(fallbackMethod = "hiError")**  public String hiService(String name) {  return restTemplate.getForObject("http://SERVICE-HI/hi?name="+name,String.class);  }  public String **hiError**(String name) {  return "hi,"+name+",sorry,error!";  }  } |

启动：service-ribbon 工程，当我们访问[http://localhost:8764/hi?name=a](http://localhost:8764/hi?name=forezp)bc,浏览器显示：

hi abc,i am from port:8762

此时关闭 service-hi 工程，当我们再访问[http://localhost:8764/hi?name=a](http://localhost:8764/hi?name=forezp)bc，浏览器会显示：

hi ,abc,orry,error!

这就说明当 service-hi 工程不可用的时候，service-ribbon调用 service-hi的API接口时，会执行快速失败，直接返回一组字符串，而不是等待响应超时，这很好的控制了容器的线程阻塞。

#### 4 在Feign中使用断路器(servcie-feign:8765/hi)

Feign是自带断路器的，在D版本的Spring Cloud中，它没有默认打开。需要在配置文件中配置打开它，在配置文件/src/main/resources/application.properties加以下代码：

|  |
| --- |
| **feign.hystrix.enabled=true** |

基于service-feign工程进行改造，只需要在FeignClient的SchedualServiceHi接口的注解中加上fallback的指定类就行了：

|  |
| --- |
| @FeignClient(value = "service-hi",**fallback = SchedualServiceHiHystric.class**)  public interface SchedualServiceHi {  @RequestMapping(value = "/hi",method = RequestMethod.GET)  String sayHiFromClientOne(@RequestParam(value = "name") String name);  } |

SchedualServiceHiHystric需要实现SchedualServiceHi 接口，并注入到Ioc容器中，代码如下：

|  |
| --- |
| @Component  public class SchedualServiceHiHystric **implements SchedualServiceHi** {  @Override  public String sayHiFromClientOne(String name) {  return "sorry "+name;  }  } |

启动四servcie-feign工程，浏览器打开<http://localhost:8765/hi?name=forezp>,注意此时service-hi工程没有启动，网页显示：

sorry abc

打开service-hi工程，再次访问，浏览器显示：

>

hi abc,i am from port:8762

这证明断路器起到作用了。

PS:在Spring Cloud中，Feign和Ribbon在整合了Hystrix后，可能会出现首次调用失败的问题。这是因为Hystrix默认的超时时间是1秒，如果超过这个时间未响应会进入fallback代码。而首次请求往往比较慢（因为Spring的懒加载机制，要实例化一些类），这个响应就有可能大于1秒钟。

解决方法有三种：

1. hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 5000

该配置是hystrix的超时时间改成5秒

1. hystrix.command.default.execution.timeout.enabled: false

该配置用于禁用hystrix的超时时间

1. feign.hystrix.enable: false

该配置用于禁用feign的hystrix，除非在一些特殊场景，不然不推荐使用。

#### 5 Hystrix仪表盘(service-ribbon:8764/hystrix)

基于service-ribbon 改造，Feign的改造和这一样。

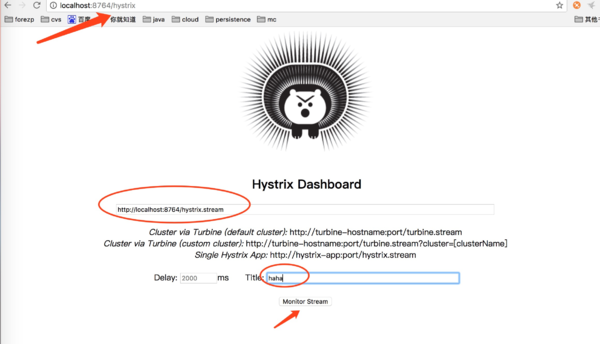
首选在pom.xml引入spring-cloud-starter-hystrix-dashboard的依赖：

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  **<artifactId>spring-boot-starter-actuator</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>**  </dependency> |

在主程序启动类中加入@EnableHystrixDashboard注解，开启hystrixDashboard：

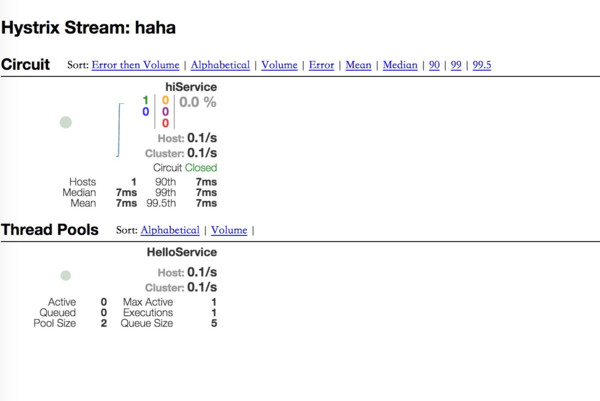
|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  @EnableHystrix  **@EnableHystrixDashboard**  public class ServiceRibbonApplication {  public static void main(String[] args) {  SpringApplication.run(ServiceRibbonApplication.class, args);  }  @Bean  @LoadBalanced  RestTemplate restTemplate()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

打开浏览器：访问<http://localhost:8764/hystrix>,界面如下，输入<http://localhost:8764/hystrix>.stream，title随便输



点击monitor stream，进入下一个界面，访问：[http://localhost:8764/hi?name=a](http://localhost:8764/hi?name=forezp)bc

此时会出现监控界面：



参考资料

[circuit\_breaker\_hystrix](http://projects.spring.io/spring-cloud/spring-cloud.html#_circuit_breaker_hystrix_clients)

[feign-hystrix](http://projects.spring.io/spring-cloud/spring-cloud.html#spring-cloud-feign-hystrix)

[hystrix\_dashboard](http://projects.spring.io/spring-cloud/spring-cloud.html#_circuit_breaker_hystrix_dashboard)

### 六、路由器-API网关（Zuul）

#### 1 Zuul简介

Zuul的主要功能是路由转发和过滤器，相当于Facade模式，外部客户端访问经过它来进行调度和过滤。路由功能是微服务的一部分，比如/api/user转发到到user服务，/api/shop转发到到shop服务。zuul默认和Ribbon结合实现了负载均衡的功能。

zuul有以下功能：

Authentication

Insights

Stress Testing

Canary Testing

Dynamic Routing

Service Migration

Load Shedding

Security

Static Response handling

Active/Active traffic management

#### 2 准备工作

在原有的工程上，创建一个新的工程。

#### 3 创建service-zuul工程( service-zuul:8769/api-\*)

其pom.xml文件如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-zuul-gateway</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-eureka</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-zuul</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在其入口applicaton类加上注解@EnableZuulProxy，开启zuul的功能：

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

加上配置文件application.yml加上以下的配置代码：

|  |
| --- |
| eureka:  client:  serviceUrl:  **defaultZone: http://localhost:8761/eureka/**  server:  port: 8769  spring:  application:  name: service-zuul  zuul:  **routes:**  api-a:  path: /api-a/\*\*  serviceId: service-ribbon  api-b:  path: /api-b/\*\*  serviceId: service-feign |

首先指定服务注册中心的地址为<http://localhost:8761/eureka/>，服务的端口为8769，服务名为service-zuul；以/api-a/ 开头的请求都转发给service-ribbon服务；以/api-b/开头的请求都转发给service-feign服务；

依次运行这五个工程;打开浏览器访问：[http://localhost:8769/api-a/hi?name=a](http://localhost:8769/api-a/hi?name=forezp)bc ;浏览器显示：

hi abc,i am from port:8762

打开浏览器访问：<http://localhost:8769/api-b/hi?name=forezp> ;浏览器显示：

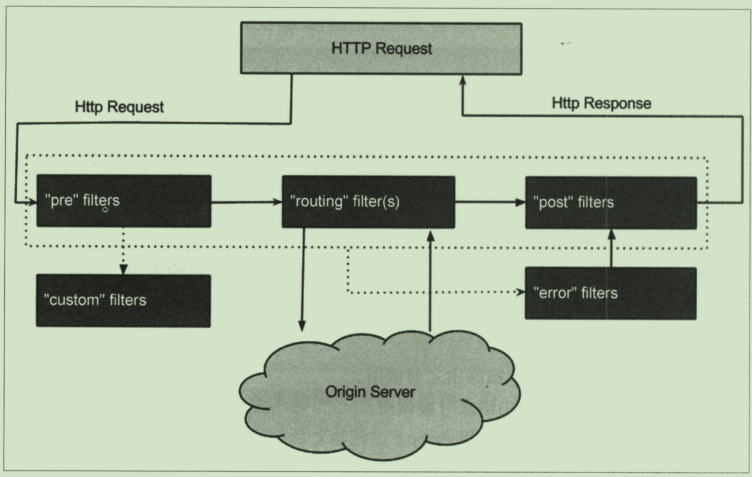
hi abc,i am from port:8762

这说明zuul起到了路由的作用

#### 4 服务过滤

zuul不仅只是路由，并且还能过滤，做一些安全验证。

请求的生命周期



继续改造工程；通过继承ZuulFilter

|  |
| --- |
| @Component  public class MyZuulFilter extends **ZuulFilter**{  private static Logger log = LoggerFactory.getLogger(MyFilter.class);  @Override  public String filterType() {//标志过滤器的类型  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(String.format("%s >>> %s", request.getMethod(),  request.getRequestURL().toString()));  Object accessToken = request.getParameter("token");  if(accessToken == null) {  log.warn("token is empty");  ctx.setSendZuulResponse(**false**);  ctx.setResponseStatusCode(401);  try {  ctx.getResponse().getWriter().write("token is empty");  }catch (Exception e){}  //ctx.setResponseBody(body);  return null;  }  log.info("ok");  return null;  }  } |

filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：

pre：路由之前

routing：路由之时（外部请求转发到相应服务实例，并返回结果后）

post： 路由之后

error：发送错误调用（还会经过post）

filterOrder：过滤的顺序

shouldFilter：这里可以写逻辑判断，是否要过滤，本文true,永远过滤。

run：过滤器的具体逻辑。可用很复杂，包括查sql，nosql去判断该请求到底有没有权限访问。

这时访问：[http://localhost:8769/api-a/hi?name=a](http://localhost:8769/api-a/hi?name=forezp)bc ；网页显示：

token is empty

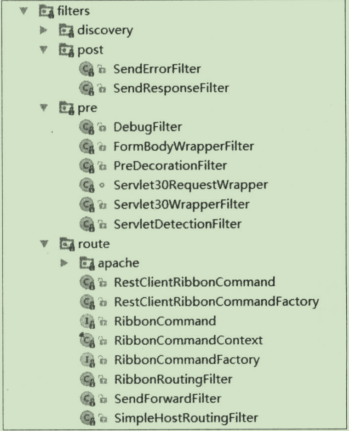
访问 [http://localhost:8769/api-a/hi?name=abc&token=22](http://localhost:8769/api-a/hi?name=forezp&token=22) ；   
网页显示：

hi abc,i am from port:8762

参考资料：

[router\_and\_filter\_zuul](http://projects.spring.io/spring-cloud/spring-cloud.html#_router_and_filter_zuul)

默认的过滤器



### 七、服务配置（Config）

#### 1 简介

在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件spring cloud config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在spring cloud config 组件中，分两个角色，一是config server，二是config client。

#### 2 构建Config Server(config-server:8888)

创建一个spring-boot项目，取名为config-server,其pom.xml:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-config-server</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-config-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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在程序的入口Application类加上@EnableConfigServer注解开启配置服务器的功能，代码如下：

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

需要在程序的配置文件application.properties文件配置以下：

|  |
| --- |
| spring.application.name=config-server  server.port=8888  **spring.cloud.config.server.git.uri=https://github.com/forezp/SpringcloudConfig/**  spring.cloud.config.server.git.searchPaths=respo  spring.cloud.config.label=master  spring.cloud.config.server.git.username=your username  spring.cloud.config.server.git.password=your password |

spring.cloud.config.server.git.uri：配置git仓库地址

spring.cloud.config.server.git.searchPaths：配置仓库路径

spring.cloud.config.label：配置仓库的分支

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

如果Git仓库为公开仓库，可以不填写用户名和密码，如果是私有仓库需要填写，本例子是公开仓库，放心使用。

远程仓库<https://github.com/forezp/SpringcloudConfig/> 中有个文件config-client-dev.properties文件中有一个属性：

foo = foo version 3

启动程序：访问<http://localhost:8888/foo/dev>

{"name":"foo","profiles":["dev"],"label":"master",

"version":"792ffc77c03f4b138d28e89b576900ac5e01a44b","state":null,"propertySources":[]}

证明配置服务中心可以从远程程序获取配置信息。

http请求地址和资源文件映射如下:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

#### 3 构建Config Client(config-client:8881)

重新创建一个springboot项目，取名为config-client,其pom文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-config-client</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-config</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

其配置文件bootstrap.properties：

|  |
| --- |
| spring.application.name=config-client  spring.cloud.config.label=master  spring.cloud.config.profile=dev  **spring.cloud.config.uri= http://localhost:8888/**  server.port=8881 |

spring.cloud.config.label 指明远程仓库的分支

spring.cloud.config.profile

dev开发环境配置文件

test测试环境

pro正式环境

spring.cloud.config.uri= <http://localhost:8888/> 指明配置服务中心的网址。

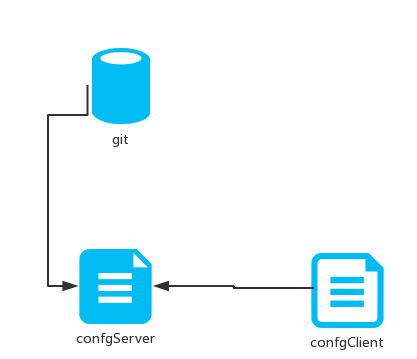
程序的入口类，写一个API接口“/hi”，返回从配置中心读取的foo变量的值，代码如下：

|  |
| --- |
| @SpringBootApplication  @RestController  public class ConfigClientApplication {  public static void main(String[] args) {  SpringApplication.run(ConfigClientApplication.class, args);  }  **@Value("${foo}")**  String foo;  @RequestMapping(value = "/hi")  public String hi(){  return foo;  }  } |

打开网址访问：<http://localhost:8881/hi>，网页显示：

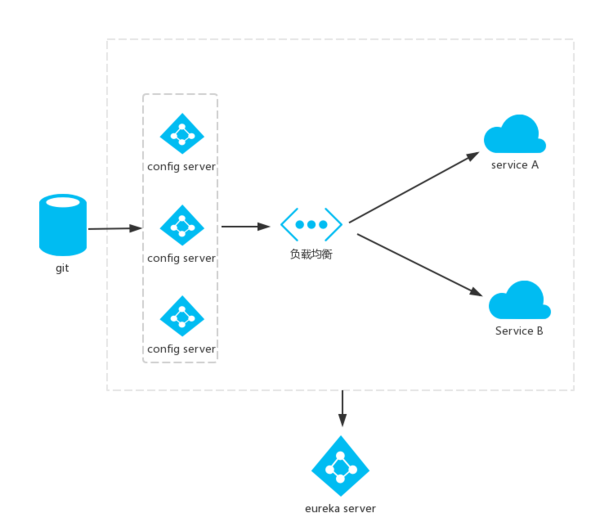
foo version 3

这就说明，config-client从config-server获取了foo的属性，而config-server是从git仓库读取的,如图：



#### 4 高可用的分布式配置中心

当服务实例很多时，都从配置中心读取文件，这时可以考虑将配置中心做成一个微服务，将其集群化，从而达到高可用，架构图如下：



##### 4.1 改造config-server

在其pom.xml文件加上EurekaClient的依赖spring-cloud-starter-eureka，代码如下:

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-eureka</artifactId>**  </dependency>  </dependencies> |

配置文件application.yml，指定**服务注册**地址为[http://localhost:8761/eureka/](http://localhost:8889/eureka/)，其他配置同上一篇文章，完整的配置如下：

|  |
| --- |
| spring.application.name=config-server  server.port=8888  spring.cloud.config.server.git.uri=https://github.com/forezp/SpringcloudConfig/  spring.cloud.config.server.git.searchPaths=respo  spring.cloud.config.label=master  spring.cloud.config.server.git.username= your username  spring.cloud.config.server.git.password= your password  **eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/** |

最后需要在程序的启动类Application加上**@EnableEurekaClient**的注解。

##### 4.2 改造config-client

将其注册到微服务注册中心，作为Eureka客户端，需要pom文件加上依赖

spring-cloud-starter-eureka，代码如下：

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-eureka</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency> |

配置文件bootstrap.properties，注意是bootstrap。加上服务注册地址为[http://localhost:8761/eureka/](http://localhost:8889/eureka/)

|  |
| --- |
| spring.application.name=config-client  spring.cloud.config.label=master  spring.cloud.config.profile=dev  #spring.cloud.config.uri= http://localhost:8888/  **eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/**  **spring.cloud.config.discovery.enabled=true**  **spring.cloud.config.discovery.serviceId=config-server**  server.port=8881 |

spring.cloud.config.discovery.enabled 是从配置中心读取文件。

spring.cloud.config.discovery.serviceId 配置中心的servieId，即服务名。

这时发现，在读取配置文件不再写ip地址，而是服务名，这时如果配置服务部署多份，通过负载均衡，从而高可用。

依次启动eureka-servr,config-server,config-client   
访问网址：[http://localhost:8761/](http://localhost:8889/)



访问<http://localhost:8881/hi>，浏览器显示：

foo version 3

### 八、高可用的服务注册中心

前面介绍了服务注册与发现，其中服务注册中心Eureka Server，是一个实例，当成千上万个服务向它注册的时候，它的负载是非常高的，这在生产环境上是不太合适的，下面介绍怎么将Eureka Server集群化。

#### 1 准备工作

Eureka can be made even more resilient and available by running multiple instances and asking them to register with each other. In fact, this is the default behaviour, so all you need to do to make it work is add a valid serviceUrl to a peer, e.g.

--摘自官网

Eureka通过运行多个实例，使其更具有高可用性。事实上，这是它默认的属性，你需要做的就是给对等的实例一个合法的关联serviceurl。

#### 2 改造工作

在eureka-server工程中resources文件夹下，配置文件application.yml:

|  |
| --- |
| ---  server:  port: 8761  spring:  profiles: peer1  application:  name: eureka-server  eureka:  instance:  hostname: peer1  client:  serviceUrl:  **defaultZone: http://peer2:8769/eureka/**  ---  server:  port: 8769  spring:  profiles: peer2  application:  name: eureka-server  eureka:  instance:  hostname: peer2  client:  serviceUrl:  **defaultZone: http://peer1:8761/eureka/** |

这时eureka-server就已经改造完毕。

需要改变etc/hosts，linux系统通过vim /etc/hosts ,加上：

127.0.0.1 peer1

127.0.0.1 peer2

windows电脑，在c:/windows/systems/drivers/etc/hosts 修改。

(也可以不手动改host，通过eureka.instance.preferIpAddress=true设置使用ip进行服务注册)

这时客户端需要改造下service-hi:

|  |
| --- |
| eureka:  client:  serviceUrl:  defaultZone: **http://peer1:8761/eureka/,http://peer2:8769/eureka/**  server:  port: 8762  spring:  application:  name: service-hi |

#### 3 启动工程

启动eureka-server，指定启动的profile：

java -jar eureka-server-0.0.1-SNAPSHOT.jar **--spring.profiles.active=peer1**

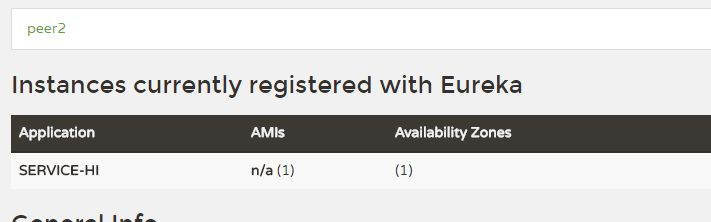
java -jar eureka-server-0.0.1-SNAPSHOT.jar **--spring.profiles.active=peer2**

>

启动service-hi:

java -jar service-hi-0.0.1-SNAPSHOT.jar

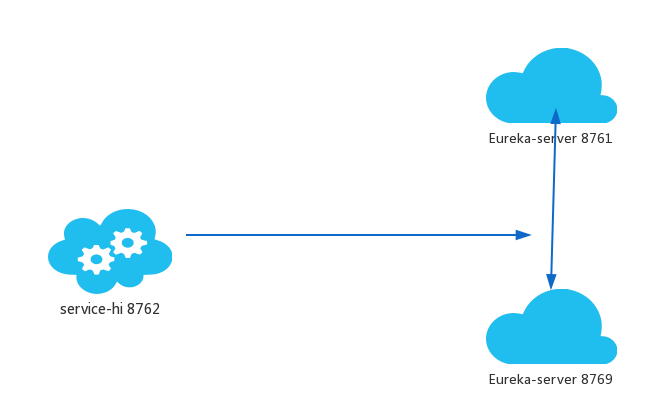
访问：localhost:8761,如图：



你会发现注册了service-hi，并且有个peer2节点，同理访问localhost:8760你会发现有个peer1节点。

client只向8761注册，但是你打开8769，你也会发现，8769也有 client的注册信息。

此时的架构图：



Eureka-eserver peer1:8761,Eureka-eserver peer2:8769相互感应，当有服务注册时，两个Eureka-eserver是对等的，它们都存有相同的信息，这就是通过服务器的冗余来增加可靠性，当有一台服务器宕机了，服务并不会终止，因为另一台服务存有相同的数据

### 九、服务链路追踪（Sleuth）

#### 1 简介

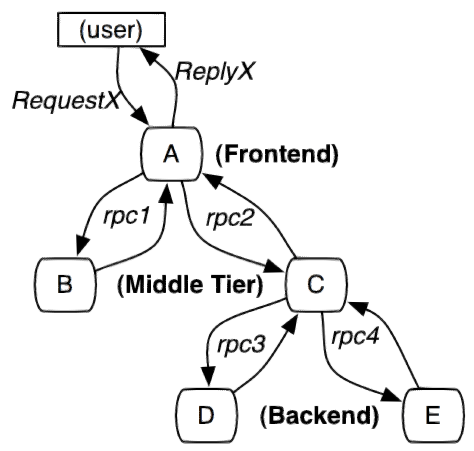
Add sleuth to the classpath of a Spring Boot application (see below for Maven and Gradle examples), and you will see the correlation data being collected in logs, as long as you are logging requests.

-- [摘自官网](https://github.com/spring-cloud/spring-cloud-sleuth)

Spring Cloud Sleuth 主要功能就是在分布式系统中提供追踪解决方案，并且兼容支持了 zipkin，你只需要在pom文件中引入相应的依赖即可。

#### 2 服务追踪分析

微服务架构上通过业务来划分服务的，通过REST调用，对外暴露的一个接口，可能需要很多个服务协同才能完成这个接口功能，如果链路上任何一个服务出现问题或者网络超时，都会形成导致接口调用失败。随着业务的不断扩张，服务之间互相调用会越来越复杂。



#### 3 术语

Span：基本工作单元，例如，在一个新建的span中发送一个RPC等同于发送一个回应请求给RPC，span通过一个64位ID唯一标识，trace以另一个64位ID表示，span还有其他数据信息，比如摘要、时间戳事件、关键值注释(tags)、span的ID、以及进度ID(通常是IP地址)   
span在不断的启动和停止，同时记录了时间信息，当你创建了一个span，你必须在未来的某个时刻停止它。

Trace：一系列spans组成的一个树状结构，例如，如果你正在跑一个分布式大数据工程，你可能需要创建一个trace。

Annotation：用来及时记录一个事件的存在，一些核心annotations用来定义一个请求的开始和结束

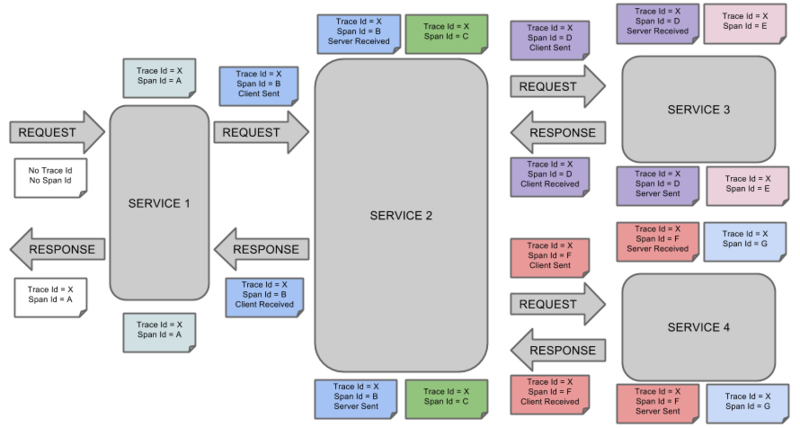
cs - Client Sent -客户端发起一个请求，这个annotion描述了这个span的开始

sr - Server Received -服务端获得请求并准备开始处理它，如果将其sr减去cs时间戳便可得到网络延迟

ss - Server Sent -注解表明请求处理的完成(当请求返回客户端)，如果ss减去sr时间戳便可得到服务端需要的处理请求时间

cr - Client Received -表明span的结束，客户端成功接收到服务端的回复，如果cr减去cs时间戳便可得到客户端从服务端获取回复的所有所需时间   
将Span和Trace在一个系统中使用Zipkin注解的过程图形化：

将Span和Trace在一个系统中使用Zipkin注解的过程图形化：



#### 4 构建工程

基本知识讲解完毕，下面我们来实战，本文的案例主要有三个工程组成:

一个server-zipkin,它的主要作用使用ZipkinServer 的功能，收集调用数据，并展示；

一个service-hi,对外暴露hi接口；

一个service-miya,对外暴露miya接口；

这两个service可以相互调用；并且只有调用了，server-zipkin才会收集数据的，这就是为什么叫服务追踪了。

##### 4.1 构建server-zipkin(:9411)

建一个spring-boot工程取名为server-zipkin，在其pom引入依赖：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-sleuth-zipkin</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>io.zipkin.java</groupId>  **<artifactId>zipkin-server</artifactId>**  </dependency>  <dependency>  <groupId>io.zipkin.java</groupId>  **<artifactId>zipkin-autoconfigure-ui</artifactId>**  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在其程序入口类, 加上注解@EnableZipkinServer，开启ZipkinServer的功能：

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

在配置文件application.yml指定服务端口为：

server.port=9411

##### 4.2 创建service-hi(:8988)

在其pom引入起步依赖spring-cloud-starter-zipkin，代码如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-sleuth-zipkin</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  **<artifactId>spring-cloud-starter-zipkin</artifactId>**  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</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>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在其配置文件application.yml指定zipkin server的地址，通过配置“spring.zipkin.base-url”指定：

|  |
| --- |
| server:  port: 8988  spring:  zipkin:  **baseUrl: http://localhost:9411**  spring:  application:  name: service-hi |

通过引入spring-cloud-starter-zipkin依赖和设置spring.zipkin.base-url就可以了。

对外暴露接口：

|  |
| --- |
| @SpringBootApplication  @RestController  public class ServiceHiApplication {  public static void main(String[] args) {  SpringApplication.run(ServiceHiApplication.class, args);  }  private static final Logger LOG = Logger.getLogger(ServiceHiApplication.class.getName());  @Autowired  private RestTemplate restTemplate;  @Bean  public RestTemplate getRestTemplate(){  return new RestTemplate();  }  @RequestMapping("/hi")  public String callHome(){  LOG.log(Level.INFO, "calling trace service-hi ");  return restTemplate.getForObject(**"http://localhost:8989/miya"**, String.class);  }  @RequestMapping("/info")  public String info(){  LOG.log(Level.INFO, "calling trace service-hi ");  return "i'm service-hi";  }  @Bean  public AlwaysSampler defaultSampler(){  return new AlwaysSampler();  }  } |

##### 4.3 创建service-miya

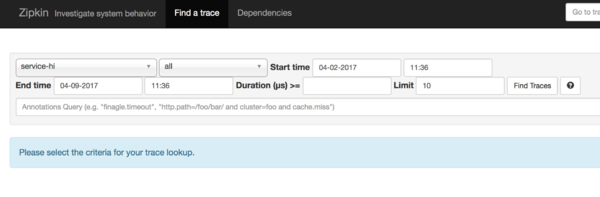
创建过程同service-hi，引入相同的依赖，配置下spring.zipkin.base-url。

对外暴露接口：

|  |
| --- |
| @SpringBootApplication  @RestController  public class ServiceMiyaApplication {  public static void main(String[] args) {  SpringApplication.run(ServiceMiyaApplication.class, args);  }  private static final Logger LOG = Logger.getLogger(ServiceMiyaApplication.class.getName());  @RequestMapping("/hi")  public String home(){  LOG.log(Level.INFO, "hi is being called");  return "hi i'm miya!";  }  @RequestMapping("/miya")  public String info(){  LOG.log(Level.INFO, "info is being called");  return restTemplate.getForObject(**"http://localhost:8988/info"**,String.class);  }  @Autowired  private RestTemplate restTemplate;  @Bean  public RestTemplate getRestTemplate(){  return new RestTemplate();  }  } |

##### 4.4 启动工程，演示追踪

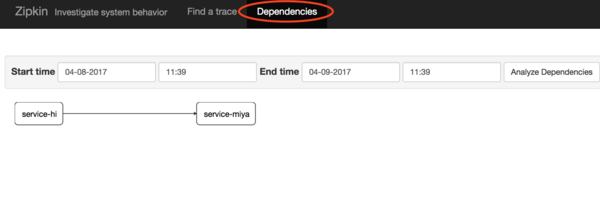
依次启动上面的三个工程，打开浏览器访问：<http://localhost:9411/>，会出现以下界面：



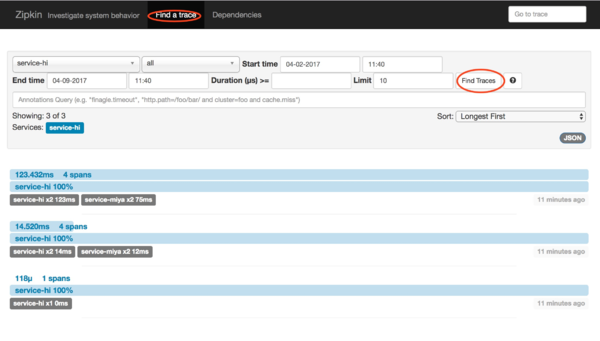
访问：<http://localhost:8989/miya>，浏览器出现：

i’m service-hi

再打开<http://localhost:9411/>的界面，点击Dependencies,可以发现服务的依赖关系：



点击find traces,可以看到具体服务相互调用的数据：



参考资料

[spring-cloud-sleuth](https://github.com/spring-cloud/spring-cloud-sleuth)利用Zipkin对Spring Cloud应用进行服务追踪分析

Spring Cloud Sleuth使用简介

### 十、消息总线（Bus）

#### 1 简介

Spring Cloud Bus 将分布式的节点用轻量的消息代理连接起来。它可以用于广播配置文件的更改或者服务之间的通讯，也可以用于监控。本文要讲述的是用Spring Cloud Bus实现通知微服务架构的配置文件的更改。

#### 2准备工作

本文还是基于上一篇文章来实现。按照官方文档，我们只需要在配置文件中配置 spring-cloud-starter-bus-amqp ；我们还需要装rabbitMq。

#### 3改造config-client

在pom文件加上起步依赖spring-cloud-starter-bus-amqp，完整的配置文件如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>sc-config-client</artifactId>  <version>0.1.0</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.retry</groupId>  <artifactId>spring-retry</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <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>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Edgware.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <java.version>1.8</java.version>  </properties>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

在配置文件application.properties中加上RabbitMq的配置，包括RabbitMq的地址、端口，用户名、密码，代码如下：

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

如果rabbitmq有用户名密码，输入即可。

依次启动eureka-server、config-server,启动两个config-client，端口为：8881、8882。

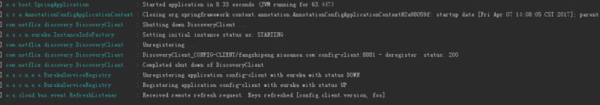
访问<http://localhost:8881/hi> 或者<http://localhost:8882/hi> 浏览器显示：

foo version 3

这时我们去[代码仓库](https://github.com/forezp/SpringcloudConfig/blob/master/respo/config-client-dev.properties)将foo的值改为“foo version 4”，即改变配置文件foo的值。如果是传统的做法，需要重启服务，才能达到配置文件的更新。此时，我们只需要发送post请求：<http://localhost:8881/bus/refresh>，你会发现config-client会重新读取配置文件

IMG_256

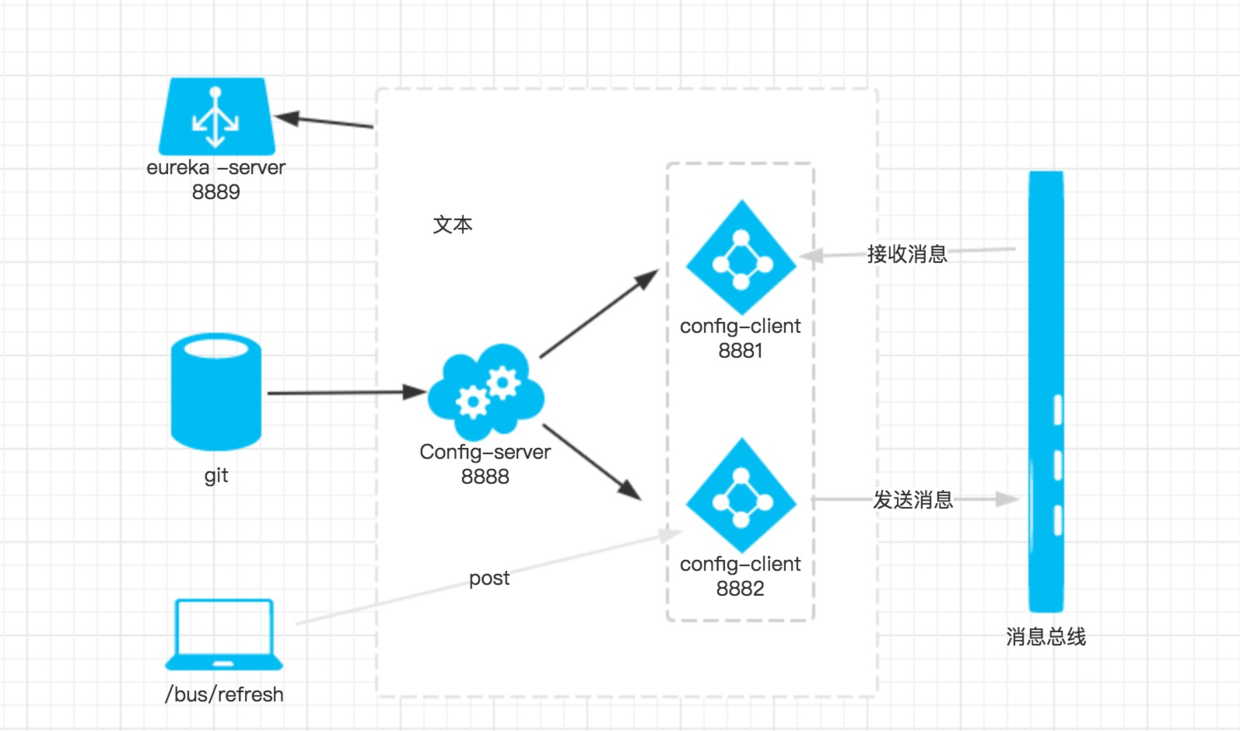
重新读取配置文件：



这时我们再访问<http://localhost:8881/hi> 或者<http://localhost:8882/hi> 浏览器显示：

foo version 4

另外，/bus/refresh接口可以指定服务，即使用”destination”参数，比如 “/bus/refresh?destination=customers:\*\*” 即刷新服务名为customers的所有服务，不管ip。

此时的架构图：   


当git文件更改的时候，通过pc端用post 向端口为8882的config-client发送请求/bus/refresh/；此时8882端口会发送一个消息，由消息总线向其他服务传递，从而使整个微服务集群都达到更新配置文件。

#### 4其他扩展

可以用作自定义的Message Broker,只需要spring-cloud-starter-bus-amqp, 然后再配置文件写上配置即可，同上。

Tracing Bus Events：   
需要设置：spring.cloud.bus.trace.enabled=true，如果那样做的话，那么Spring Boot TraceRepository（如果存在）将显示每个服务实例发送的所有事件和所有的ack,比如：（来自官网）

{

"timestamp": "2015-11-26T10:24:44.411+0000",

"info": {

"signal": "spring.cloud.bus.ack",

"type": "RefreshRemoteApplicationEvent",

"id": "c4d374b7-58ea-4928-a312-31984def293b",

"origin": "stores:8081",

"destination": "\*:\*\*"

}

},

{

"timestamp": "2015-11-26T10:24:41.864+0000",

"info": {

"signal": "spring.cloud.bus.sent",

"type": "RefreshRemoteApplicationEvent",

"id": "c4d374b7-58ea-4928-a312-31984def293b",

"origin": "customers:9000",

"destination": "\*:\*\*"

}

},

{

"timestamp": "2015-11-26T10:24:41.862+0000",

"info": {

"signal": "spring.cloud.bus.ack",

"type": "RefreshRemoteApplicationEvent",

"id": "c4d374b7-58ea-4928-a312-31984def293b",

"origin": "customers:9000",

"destination": "\*:\*\*"

}

}

参考资料

spring\_cloud\_bus

## 十一、国内使用公司

五.国内使用情况  
 1. 中国联通子公司  
2. 上海米么金服  
3.指点无限（北京）科技有限公司  
4. 易保软件 目前在定制开发中  
5. 广州简法网络  
6.深圳睿云智合科技有限公司  
7.持续交付产品基于Spring Cloud研发 [http://www.wise2c.com](http://www.wise2c.com/)  
8.猪八戒网  
9上海云首科技有限公司  
**10华为**  
 整合netty进来用rpc 包括nerflix那套东西 需要注意的是sleuth traceid的传递需要自己写。tps在物理机上能突破20w  
 10.东软  
11.南京云帐房网络科技有限公司  
12.四众互联(北京)网络科技有限公司  
13.深圳摩令技术科技有限公司  
14.广州万表网  
15.视觉中国  
16.上海秦苍信息科技有限公司-买单侠  
17.爱油科技(大连)有限公司  
18.爱油科技基于SpringCloud的微服务实践  
**19.广发银行**  
20.卖货郎(<http://www.51mhl.com/>）  
21.拍拍贷  
22.甘肃电信  
23.新浪商品部  
24.春秋航空  
25.冰鉴科技  
26.万达网络科技集团-共享商业平台-共享供应链中心  
27.网易乐得技术团队([http://tech.lede.co](http://tech.lede.co/)m)  
28.饿了么某技术团队  
29.高阳捷迅信息科技–话费中心业务平台–凭证查询及收单系统  
30.中国电信综合办公业务  
31.上海拍拍信  
**32. 众安保险**

## 十二、现有的项目与Spring Cloud交互

#### 1 独立使用Feign进行http 调用

现有的项目并不是spring cloud项目，使用者关注的重点仅仅是简化http调用代码的编写。

如果采用httpclient或者okhttp这样相对较重的框架，对初学者来说编码量与学习曲线都会是一个挑战，而使用spring中RestTemplate，又没有配置化的解决方案，这里我们独立使用Feign。

##### 1.1 maven依赖

|  |
| --- |
| <dependency>  <groupId>com.netflix.feign</groupId>  <artifactId>feign-core</artifactId>  <version>8.18.0</version>  </dependency> |

自定义接口

import feign.Param;

import feign.RequestLine;

public interface RemoteService {

@RequestLine("GET /users/list?name={name}")

String getOwner(@Param(value = "name") String name);

}

通过@RequestLine指定HTTP协议及URL地址

配置类

RemoteService service = Feign.builder()

.options(new Options(1000, 3500))

.retryer(new Retryer.Default(5000, 5000, 3))

.target(RemoteService.class, **"http://127.0.0.1:8762/hi"**);

options方法指定连接超时时长及响应超时时长，retryer方法指定重试策略,target方法绑定接口与服务端地址。返回类型为绑定的接口类型。

##### 1.2调用

String result = service.getOwner("scott");

与调用本地方法相同的方式调用feign包装的接口，直接获取远程服务提供的返回值。

附：服务生产者

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

@Controller

@RequestMapping(value="users")

public class UserController {

@RequestMapping(value="/list",method={RequestMethod.GET,RequestMethod.POST,RequestMethod.PUT})

@ResponseBody

public String list(@RequestParam String name) throws InterruptedException{

return name.toUpperCase();

}

}

##### 1.3更进一步

在项目中，服务消费端与生产端之间交换的数据往往是一或多个对象，feign同样提供基于json的对象转换工具，方便我们直接以对象形式交互。

业务接口

public interface RemoteService {

@Headers({"Content-Type: **application/json**","Accept: **application/json**"})

@RequestLine("POST /users/list")

User getOwner(User user);

}

加入@Headers注解，指定Content-Type为json

配置

RemoteService service = Feign.builder()

**.encoder(new JacksonEncoder())**

**.decoder(new JacksonDecoder())**

.options(new Options(1000, 3500))

.retryer(new Retryer.Default(5000, 5000, 3))

.target(RemoteService.class, "http://127.0.0.1:8085");

encoder指定对象编码方式，decoder指定对象解码方式。这里用的是基于Jackson的编、解码方式，需要在pom.xml中添加Jackson的依赖

|  |
| --- |
| <dependency>  <groupId>com.netflix.feign</groupId>  <artifactId>feign-jackson</artifactId>  <version>8.18.0</version>  </dependency> |

调用

User result = service.getOwner(u);

附：服务生产者

@Controller

@RequestMapping(value="users")

public class UserController {

@RequestMapping(value="/list",method={RequestMethod.GET,RequestMethod.POST,RequestMethod.PUT})

@ResponseBody

public User list(@RequestBody User user) throws InterruptedException{

System.out.println(user.getUsername());

user.setId(100L);

user.setUsername(user.getUsername().toUpperCase());

return user;

}

}

唯一的变化就是使用了@RequestBody来接收json格式的数据。

作者：SamHxm  
链接：https://www.jianshu.com/p/3d597e9d2d67  
來源：简书  
https://github.com/OpenFeign/feign

#### 2 使用Feign+Ribbon实现负载均衡

在项目开发中，除了考虑正常的调用之外，负载均衡和故障转移也是关注的重点，这也是feign + ribbon的优势所在。

##### 2.1 maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.netflix.feign</groupId>  <artifactId>feign-core</artifactId>  <version>8.18.0</version>  </dependency>  <dependency>  <groupId>com.netflix.feign</groupId>  <artifactId>feign-jackson</artifactId>  <version>8.18.0</version>  </dependency>  <dependency>  <groupId>com.netflix.feign</groupId>  <artifactId>feign-ribbon</artifactId>  <version>8.18.0</version>  </dependency>  </dependencies> |

其中feign-core和feign-ribbon是必须的，如果需要在服务消费端和服务生产端之间进行对象交互，建议使用feign-jackson

服务消费端接口

|  |
| --- |
| public interface RemoteService {  **@Headers({"Content-Type: application/json","Accept: application/json"})**  @RequestLine("GET /users/list")  User getOwner(User user);  } |

RemoteService接口中定义了一个名为getUser的方法，该方法的参数与返回类型都是User(标准POJO),定义如下：

|  |
| --- |
| public class User {  private Long id;    private String username;  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getUsername() {  return username;  }  public void setUsername(String username) {  this.username = username;  }  } |

##### 2.2配置及运行

|  |
| --- |
| import com.netflix.config.ConfigurationManager;  import feign.Feign;  import feign.jackson.JacksonDecoder;  import feign.jackson.JacksonEncoder;  import feign.ribbon.RibbonClient;  public class AppRun {  public static void main(String[] args) throws Exception {  ConfigurationManager.loadPropertiesFromResources("sample-client.properties");  User param = new User();  param.setUsername("scott");  RemoteService service = Feign.builder().client(**RibbonClient**.create())  .encoder(new JacksonEncoder())  .decoder(new JacksonDecoder())  .target(RemoteService.class, **"http://sample-client/gradle-web"**);  for (int i = 1; i <= 10; i++) {  User result = service.getOwner(param);  System.out.println(result.getId() + "," + result.getUsername());  }  }  } |

首先利用com.netflix.config.ConfigurationManager读取配置文件sample-client.properties,该文件位于src/main/resources下。

然后声明了一个User类型的对象param，该对象将作为参数被发送至服务生产端。

重点在于通过RibbonClient.create()使得Feign对象获得了Ribbon的特性。之后通过encoder,decoder设置编码器与解码器，并通过target方法将之前定义的接口RemoteService与一个URL地址http://sample-client/gradle-web进行了绑定。

现在来看sample-client.properties中的配置项

|  |
| --- |
| sample-client.ribbon.MaxAutoRetries=1  sample-client.ribbon.MaxAutoRetriesNextServer=1  sample-client.ribbon.OkToRetryOnAllOperations=true  sample-client.ribbon.ServerListRefreshInterval=2000  sample-client.ribbon.ConnectTimeout=3000  sample-client.ribbon.ReadTimeout=3000  **sample-client.ribbon.listOfServers=127.0.0.1:8080,127.0.0.1:8085**  sample-client.ribbon.EnablePrimeConnections=false |

所有的key都以**sample-client**开头，表明这些配置项作用于名为sample-client的服务。其实就是与之前绑定RemoteService接口的URL地址的schema相对应。

重点看sample-client.ribbon.listOfServers配置项，该配置项指定了服务生产端的真实地址。

之前与RemoteService接口绑定的URL地址是http://sample-client/gradle-web，在调用时会被替换为http://127.0.0.1:8080/gradle-web或http://127.0.0.1:8085/gradle-web,再与接口中@RequestLine指定的地址进行拼接，得到最终请求地址。本例中最终请求地址为http://127.0.0.1:8080/gradle-web/users/list或http://127.0.0.1:8085/gradle-web/users/list

由于使用的ribbon，所以feign不再需要配置超时时长，重试策略。ribbon提供了更为完善的策略实现。

本例中，服务生产端是一个简单的spring mvc，实现如下:

|  |
| --- |
| @RestController  @RequestMapping(value="users")  public class UserController {  @RequestMapping(value="/list",method={RequestMethod.GET,RequestMethod.POST,RequestMethod.PUT})  public User list(@RequestBody User user) throws InterruptedException{  HttpServletRequest request =  ((ServletRequestAttributes)RequestContextHolder.getRequestAttributes()).getRequest();  user.setId(new Long(request.getLocalPort()));  user.setUsername(user.getUsername().toUpperCase());  return user;  }  } |

简单的将容器运行的端口号设为id并将username改为大写字母后返回。

将服务生产端部署到2个容器中，分别在8080和8085端口运行。

运行服务消费端的main方法，观察输出。

8085,SCOTT

8080,SCOTT

8085,SCOTT

8080,SCOTT

8085,SCOTT

8080,SCOTT

8085,SCOTT

8080,SCOTT

8085,SCOTT

8080,SCOTT

可以发现，feign交替的访问2个服务端。

如果关闭其中一个服务端，再次运行客户端会发现通过访问了存活的服务端，客户端仍然能够运行，这就是ribbon提供负载均衡与故障转移的最简单实现。

故障转移是通过sample-client.properties中的配置项进行配置。

具体参数含义请查阅官方文档。

##### 2.3负载均衡的策略设置

|  |
| --- |
| import com.netflix.client.ClientFactory;  import com.netflix.client.config.IClientConfig;  import com.netflix.config.ConfigurationManager;  import com.netflix.loadbalancer.ILoadBalancer;  import com.netflix.loadbalancer.RandomRule;  import com.netflix.loadbalancer.ZoneAwareLoadBalancer;  import feign.Feign;  import feign.jackson.JacksonDecoder;  import feign.jackson.JacksonEncoder;  import feign.ribbon.LBClient;  import feign.ribbon.LBClientFactory;  import feign.ribbon.RibbonClient;  public class AppRun {  public static void main(String[] args) throws Exception {  ConfigurationManager.loadPropertiesFromResources("sample-client.properties");  User param = new User();  param.setUsername("scott");  RibbonClient client = RibbonClient.builder().lbClientFactory(new LBClientFactory() {  @Override  public LBClient create(String clientName) {  IClientConfig config = ClientFactory.getNamedConfig(clientName);  ILoadBalancer lb = ClientFactory.getNamedLoadBalancer(clientName);  ZoneAwareLoadBalancer zb = (ZoneAwareLoadBalancer) lb;  **zb.setRule(new RandomRule());//随机策略**  return LBClient.create(lb, config);  }  }).build();  RemoteService service = Feign.builder().client(client)  .encoder(new JacksonEncoder())  .decoder(new JacksonDecoder())  .target(RemoteService.class, "http://sample-client/gradle-web");  for (int i = 1; i <= 10; i++) {  User result = service.getOwner(param);  System.out.println(result.getId() + "," + result.getUsername());  }  }  } |

不再使用RibbonClient.create()来创建默认的RibbonClient，而是通过RibbonClient.builder()获得feign.ribbon.Builder,进而设置LBClientFactory的实现来定制LBClient，在创建LBClient的过程中即可指定负载策略的具体实现。

参考

[http://www.idouba.net/netflix-source-ribbon-rule/?utm\_source=tuicool&utm\_medium=referral](https://link.jianshu.com?t=http://www.idouba.net/netflix-source-ribbon-rule/?utm_source=tuicool&utm_medium=referral)

这里使用的指定方式并不推荐，实际项目中应该创建LBClient的过程中指定ILoadBalancer实现，而不应该创建完毕后再强转类型来进行修改。

确认2个服务端正常启动，运行客户端，观察输出。

8080,SCOTT

8080,SCOTT

8080,SCOTT

8080,SCOTT

8085,SCOTT

8080,SCOTT

8085,SCOTT

8080,SCOTT

8085,SCOTT

8085,SCOTT

现在不再是交替访问服务端，而是随机访问。证明更换负载均衡策略有效。

作者：SamHxm  
链接：https://www.jianshu.com/p/37e915bea7c8  
來源：简书

#### 3 传统spring应用注入Feign对象

上面的demo都是在main方法中进行feign对象的组装，而实际的java项目大部分都会用到spring IOC容器进行对象管理。

spring环境有很多类型，大致可分为:

* 传统的，基于xml或Java config的spring应用
* spring boot应用
* spring cloud应用

##### 3.1在spring cloud中

feign，ribbon等netflix产品已经与spring进行了集成。

##### 3.2在spring boot中

使用@Bean对远程调用接口定义即可。例如:

|  |
| --- |
| import java.io.IOException;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import com.netflix.client.ClientFactory;  import com.netflix.client.config.IClientConfig;  import com.netflix.config.ConfigurationManager;  import com.netflix.loadbalancer.ILoadBalancer;  import com.netflix.loadbalancer.IRule;  import com.netflix.loadbalancer.RandomRule;  import com.netflix.loadbalancer.ZoneAvoidanceRule;  import com.netflix.loadbalancer.ZoneAwareLoadBalancer;  import com.sam.demo.service.RemoteService;  import feign.Feign;  import feign.jackson.JacksonDecoder;  import feign.jackson.JacksonEncoder;  import feign.ribbon.LBClient;  import feign.ribbon.LBClientFactory;  import feign.ribbon.RibbonClient;  @Configuration  public class Config {  public Config() {  try {  ConfigurationManager.loadPropertiesFromResources("sample-client.properties");  } catch (IOException e) {  e.printStackTrace();  }  }  @Bean  public RemoteService remoteService() {  RibbonClient client = RibbonClient.builder().lbClientFactory(new LBClientFactory() {  @Override  public LBClient create(String clientName) {  IClientConfig config = ClientFactory.getNamedConfig(clientName);  ILoadBalancer lb = ClientFactory.getNamedLoadBalancer(clientName);  ZoneAwareLoadBalancer zb = (ZoneAwareLoadBalancer) lb;  zb.setRule(zoneAvoidanceRule());  return LBClient.create(lb, config);  }  }).build();  RemoteService service = Feign.builder().client(client)  .encoder(new JacksonEncoder())  .decoder(new JacksonDecoder())  .target(RemoteService.class, "http://sample-client/gradle-web");  return service;  }  /\*\*  \* Ribbon负载均衡策略实现  \* 使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，  \* 剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。  \* @return  \*/  private IRule zoneAvoidanceRule() {  return new ZoneAvoidanceRule();  }    /\*\*  \* Ribbon负载均衡策略实现  \* 随机选择一个server。  \* @return  \*/  private IRule randomRule() {  return new RandomRule();  }  } |

RemoteService就是自定义的远程调用接口。

|  |
| --- |
| import com.sam.demo.entity.User;  import feign.Headers;  import feign.RequestLine;  public interface RemoteService {    @Headers({"Content-Type: application/json","Accept: application/json"})  @RequestLine("POST /users/list")  public User getOwner(User user);  } |

这里顺便用到了Ribbon的另一种负载均衡策略实现ZoneAvoidanceRule,并定义了RandomRule负载均衡策略实现，如果有多个远程调用接口，可以根据业务情况，选择不同的负载均衡策略实现。

需要注意的是，在组装远程接口的实现之前，需要读取配置文件，这里将读取动作简单的放到了构造方法中。实际项目中应与其他读取配置文件的代码封装在一起。

在需要使用远程接口的地方，注入该接口即可，例如:

|  |
| --- |
| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.sam.demo.entity.User;  import com.sam.demo.service.RemoteService;  @RestController  public class RemoteController {    **@Autowired**  **private RemoteService remoteService;**    @RequestMapping(value="/remote",method={RequestMethod.GET})  public String list() {    User user = new User();  user.setUsername("scott");  user = remoteService.getOwner(user);    return user.getUsername();  }  } |

将RemoteService当作普通的bean注入即可，使用上也完全与本地调用方式相同。

##### 3.3传统spring应用中

处理思路与spring boot相同，仅仅的操作方式发生改变。

|  |
| --- |
| <bean id="remoteService"  class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">  <property name="targetClass">  **<value>com.sam.demo.config.Config</value>**  </property>  <property name="targetMethod">  **<value>remoteService</value>**  </property>  </bean> |

同样是将方法返回值定义为bean。下面是com.sam.demo.config.Config的实现，与spring boot的方式几乎一致。只不过为了方便，改为静态方法。

|  |
| --- |
| import java.io.IOException;  import com.netflix.client.ClientFactory;import com.netflix.client.config.IClientConfig;  import com.netflix.config.ConfigurationManager;  import com.netflix.loadbalancer.ILoadBalancer;  import com.netflix.loadbalancer.IRule;  import com.netflix.loadbalancer.RandomRule;  import com.netflix.loadbalancer.ZoneAvoidanceRule;  import com.netflix.loadbalancer.ZoneAwareLoadBalancer;  import com.sam.demo.service.RemoteService;  import feign.Feign;  import feign.jackson.JacksonDecoder;  import feign.jackson.JacksonEncoder;  import feign.ribbon.LBClient;  import feign.ribbon.LBClientFactory;  import feign.ribbon.RibbonClient;  public class Config {    static {  try {  ConfigurationManager.loadPropertiesFromResources("sample-client.properties");  } catch (IOException e) {  e.printStackTrace();  }  }  public static RemoteService remoteService() {  RibbonClient client = RibbonClient.builder().lbClientFactory(new LBClientFactory() {  @Override  public LBClient create(String clientName) {  IClientConfig config = ClientFactory.getNamedConfig(clientName);  ILoadBalancer lb = ClientFactory.getNamedLoadBalancer(clientName);  ZoneAwareLoadBalancer zb = (ZoneAwareLoadBalancer) lb;  zb.setRule(zoneAvoidanceRule());  return LBClient.create(lb, config);  }  }).build();  RemoteService service = Feign.builder().client(client)  .encoder(new JacksonEncoder())  .decoder(new JacksonDecoder())  .target(RemoteService.class, "http://sample-client/gradle-web");  return service;  }  /\*\*  \* Ribbon负载均衡策略实现  \* 使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，  \* 剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。  \* @return  \*/  private static IRule zoneAvoidanceRule() {  return new ZoneAvoidanceRule();  }    /\*\*  \* Ribbon负载均衡策略实现  \* 随机选择一个server。  \* @return  \*/  private static IRule randomRule() {  return new RandomRule();  }  } |

在需要调用远程接口的类中，注入该接口即可。

|  |
| --- |
| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.sam.demo.entity.User;  import com.sam.demo.service.RemoteService;  @RestController  public class RemoteController {    **@Autowired**  **private RemoteService remoteService;**    @RequestMapping(value="/remote",method={RequestMethod.GET})  public String list() {    User user = new User();  user.setUsername("scott");  user = remoteService.getOwner(user);    return user.getUsername();  }  } |

作者：SamHxm  
链接：https://www.jianshu.com/p/7bb96e2dc944  
來源：简书