# SpringCloudConfig

Spring Cloud Config为分布式系统中的外部化配置提供服务器和客户端支持，达到配置集中化管理，结合spring cloud bus可以实现动态更新属性配置。

Spring cloud config 分为两部分 server 和 client

* config-server 配置服务端，服务管理配置信息
* config-client 客户端，客户端调用server端暴露接口获取配置信息

## server服务端

### pom依赖

parent

|  |
| --- |
| <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.tk</groupId>  <artifactId>config-server</artifactId>  <packaging>jar</packaging>  <version>0.0.1-SNAPSHOT</version>    <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  </parent>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.8</java.version>  </properties>    <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter</artifactId>  </dependency>    <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency>    </dependencies>    </project> |

### 配置文件

**bootstrap.properties**

bootstrap.propertie优先加载，一般存放不会改变的配置，访问git的地址必须配置在bootstrap中。

Git地址，git仓库中存储配置文件

|  |
| --- |
| spring.cloud.config.server.git.uri: https://github.com/mouwengang/config.git |

**application.yml**

|  |
| --- |
| spring:  application:  name: configserver  server:  port: 8888 |

### Application启动类

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

运行 application,configserver就可以启动。

### 相关注解

@SpringBootApplication：springboot定义的简化注解，用于代替@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan这三个注解。

@EnableConfigServer：springcloud定义的注解，标识该项目作为configserver服务端

### 数据加密

**先决条件：**要使用加密和解密功能，您需要在JVM中安装全功能JCE（默认情况下不存在）。您可以从Oracle下载并按照安装说明进行安装（实质上将JRE lib / security目录中的2个策略文件替换为您下载的策略文件）。

Java 8 JCE的地址是：<http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>

其他Java版本的JCE的下载地址及安装在Spring Cloud 文档中有提及，详见：<http://cloud.spring.io/spring-cloud-static/Camden.SR2/#_cloud_native_applications>

#### 对称加密

在application.yml文件中加入以下参数：

encrypt.key: foo

1.加密

启动configserver后执行下列命令加密，返回加密后的密文

$ curl localhost:8888/encrypt -d mysecret

682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

2.解密

反向也可以通过命令/decrypt解密，返回mysecret。

curl localhost:8888/decrypt –d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda

3.如果远程属性源包含加密内容，则需要以{cipher}开头，服务端将通过HTTP发送到客户端之前解密。

例如

*application.yml*

spring:

datasource:

username: dbuser

password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'

.properties文件中的加密值不能用引号括起来，否则该值不会被解密：

*application.properties*

spring.datasource.username：dbuser

spring.datasource.password：{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ

#### 非对称加密

1. 创建一个密钥库

命令：keytool -genkeypair -alias mytestkey -keyalg RSA -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" -keypass changeme -keystore server.jks -storepass letmein

执行该命令后生成一个server.jks文件

-dname  组织信息   
-alias  别名为  
-keyalg   
     DSA RSA                    DSA或RSA算法(当使用-genkeypair参数)   
     DES DESede AES      DES或DESede或AES算法(当使用-genseckey参数)   
-keypass   这个证书的私钥密码

-keystore  证书库的名称

-storepass       证书库的访问密码

2.将server.jks文件放在类路径classpath下，然后在配置服务器的application.yml中添加以下配置

encrypt:

keyStore:

location: classpath:/server.jks

password: letmein #证书库访问密码

alias: mytestkey

secret: changeme #私钥密码

加密和解密方式和对称加密相同。

#### 客户端解密

前面的配置是只在configserver服务端配置了encrypt.\*属性，获取到远程仓库的密文解密后发送给客户端，有时，我们希望客户端在本地解密配置，而不是在服务器中进行。在这种情况下，我们仍然可以拥有/ encrypt和/ decrypt端点（需要提供encrypt.\* 配置以定位密钥），但服务端需要使用显式关闭对传出属性的解密spring.cloud.config.server.encrypt.enabled=false，同时要在configclient客户端配置encrypt.\*属性定位密钥。

## Client客户端

### Pom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.tk</groupId>  <artifactId>config-client</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>    <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>    <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>    </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

### 配置文件

访问配置服务器的spring.cloud.config.uri必须放在bootstrap中。

Boostrap.yml

|  |
| --- |
| spring.application.name=configclient  server.port=7002  #configserver访问路径  spring.cloud.config.uri: http://localhost:8888 |

### Application启动类

|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  @RestController  @RefreshScope  **public** **class** ConfigClientApplication {  @Value("${from}")  **private** String from;    @Value("${myname}")  **private** String myname;  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ConfigClientApplication.**class**, args);  }  @RequestMapping("/get")  **public** String get(){  **return** from+"===="+myname;  }    } |

运行application，引用了spring-cloud-starter-config依赖，配置了spring.cloud.config.uri，启动时自动访问configserver，加载git上的配置。

### 加载配置的规则

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

/{application}-{profile}.yml

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

/{application}-{profile}.properties

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

{application}映射到客户端的“spring.application.name”;

{profile}映射到客户端上的“spring.profiles.active”（逗号分隔列表）

{label}这是一个服务器端功能标记一个“版本化”的配置文件集，默认为master。

例如：该客户端spring.application.name是configclient，git仓库中有configclient-default.yml配置文件，项目启动时就会去自动加载该配置文件。

如果git仓库中有application.peoperties或application.yml文件，所有依赖集中配置管理的项目都会加载。

### 动态修改配置

通过config server加载的配置可以实现动态修改. 如上，ConfigClientApplication中的“from”属性是从git仓库中的配置文件加载的，项目启动后已经成功加载，此时，修改配置文件中“from”的值，不重启client端也可以做到修改client中的from属性，前提是给“from”所在类上加一个注解**@RefreshScope**,这个注解加了之后以**POST**形式访问“**http://ip:port/refresh**”(注:这里的ip和port是指需要被刷新的client端的ip和port)就可以刷新“from”的值，原理是重新创建一个代理bean，引用该bean的对象都指向新的代理bean。

使用/refresh功能需要在项目中添加spring-boot-starter-actuator依赖

## configserver配置参数

### git配置

spring.cloud.config.server.git.uri:链接到git远程仓库的地址

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

spring.cloud.config.server.git.password:连接git仓库用户名密码

spring.cloud.config.server.git.searchPaths:搜索配置文件的路径，默认只搜索根路径下。如果该属性配置为foo,bar\*,代表将会在仓库根目录和“foo /”子目录以及名称以“bar”开头的任何子目录中搜索配置文件。

spring.cloud.config.server.git.basedir:将远程仓库克隆到本地服务器的路径

spring.cloud.config.server.git.force-pull:如果本地克隆的配置文件经过修改有了脏数据，当此参数是true时强制从远程存储库中获取，默认值为false。

### svn配置

以上这些配置是使用的git仓库存储配置文件，如果使用的是svn，需要注意以下两点:

1. 激活subversion(config server内部定义的一个profile)
2. 把上面提到的配置中的git改为svn。

### 加密配置

spring.cloud.config.server.encrypt.enabled:远程仓库的配置信息发送给客户端前是否解密，默认为true.

spring.cloud.config.server.default-label:可以自己配置，没有配置时，系统根据仓库类型git或svn分别取默认值master和trunk

### 本地存储配置

需要设置spring.profiles.active=native,configserver会默认从应用的src/main/resource目录下加载配置文件，也可以通过设置spring.cloud.config.server.native.searchLocations=file:F:/properties/属性来指定配置文件所在路径。

## configclient配置参数

spring.cloud.config.uri:连接configserver地址，默认是”<http://localhost：8888>”

spring.cloud.config.profile:对应加载配置文件规则中的｛profile｝，默认default

spring.cloud.config.label:对应配置文件分支名，git默认是master

spring.cloud.config.failFast:在某些情况下，客户端无法连接到配置服务器，则希望服务启动失败，此时可以设置该值为true,默认为false。

# SpringCloudEureka

Eureka是Netflix服务发现服务器和客户端。服务器可以配置和部署为高可用性，每个服务器将已注册服务的状态复制到其他服务器。eureka是提供服务注册，为每个注册服务提供知晓其他服务状态的服务。

## Server服务端

### Pom依赖

|  |
| --- |
| <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"*>  <groupId>com.tk</groupId>  <modelVersion>4.0.0</modelVersion>  <artifactId>eureka-server</artifactId>  <version>0.0.1-SNAPSHOT</version>    <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  </parent>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>    </dependencies>    </project> |

### 配置文件

**application.yml**

**standalone模式**

|  |
| --- |
| #单机模式  server:  port: 8761  spring:  application:  name: eureka-server  profiles: standalone  eureka:  instance:  hostname: localhost  #心跳检测时间（秒），默认30 生产环境建议默认值  leaseRenewalIntervalInSeconds: 5  client:  registerWithEureka: **true**  fetchRegistry: **true**  serviceUrl:  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ |

**集群模式**

Hostname不能相同，本机测试时需要修改host文件

|  |
| --- |
| #集群模式eureka-1  server:  port: 8761  spring:  application:  name: eureka-server  profiles: eureka-1  eureka:  instance:  hostname: eureka-1  #心跳检测时间（秒），默认30 生产环境建议默认值  leaseRenewalIntervalInSeconds: 5  #服务失效时间。默认是90秒  leaseExpirationDurationInSeconds: 10  client:  #注册到eureka服务中  registerWithEureka: **true**  #从eureka服务中获取列表信息  fetchRegistry: **true**  serviceUrl:  #注册到eureka-1，2服务上  defaultZone: http://eureka-1:8761/eureka/,http://eureka-2:8762/eureka/  ---  #集群模式eureka-2  server:  port: 8762  spring:  application:  name: eureka-server  profiles: eureka-2  eureka:  instance:  hostname: eureka-2  #心跳检测时间（秒），默认30 生产环境建议默认值  leaseRenewalIntervalInSeconds: 5  leaseExpirationDurationInSeconds: 10  client:  registerWithEureka: **true**  fetchRegistry: **true**  serviceUrl:  #注册到eureka-1，2服务上  defaultZone: http://eureka-1:8761/eureka/,http://eureka-2:8762/eureka/ |

### Application启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaServer  **public** **class** ApplicationEurekServer {    **public** **static** **void** main(String[] args) {  SpringApplication sa = **new** SpringApplication(ApplicationEurekServer.**class**);  // sa.setAdditionalProfiles("eureka-1");//集群模式 服务1  // sa.setAdditionalProfiles("eureka-2");//集群模式服务2  sa.setAdditionalProfiles("standalone");//单机模式  sa.run(args);  }  } |

运行application.java后访问localhost：8761查看注册的服务

### 相关注解

* @EnableEurekaServer: 该注解表明应用为eureka服务，有可以联合多个服务作为集群，对外提供服务注册以及发现功能

## Client客户端

**以configserver为例**

### pom依赖

Pom中新增依赖：

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

### application启动类

Application.java类上添加注解：

@EnableDiscoveryClient或@EnableEurekaClient

### 配置文件

Application.yml中添加注册到eureka的配置

|  |
| --- |
| eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/ |

运行application.java后，访问localhost:8761，可以看见configserver已经注册到了eureka上。

### 相关注解

* @EnableEurekaClient和 @EnableDiscoveryClient：该注解表明应用既作为eureka实例又为eureka client 可以发现注册的服务，任选一个即可。

## Eureka配置信息

*eureka.dashboard.path:eureka:*页面展示路径，默认是“/”.

*eureka.server.registry-sync-retries:*注册服务时重试次数，默认0

*eureka.server.number-of-replication-retries:*集群消息复制重试次数

*eureka.client.fetch-registry:*客户端是否应从eureka服务器提取eureka注册表信息，默认true

*eureka.client.register-with-eureka*: 是否应向eureka服务器注册其信息以供其他人发现，默认true

*eureka.client.registry-fetch-interval-seconds*: 从eureka服务器获取注册表信息的频率（以秒为单位）

*eureka.instance.hostname:*主机名成

*eureka.instance.lease-renewalIntervalInSeconds:* eureka客户端需要向eureka服务器发送心跳以指示其仍然活动的频率（以秒为单位）。如果在leaseExpirationDurationInSeconds中指定的时间段内没有收到心跳，eureka服务器将从其视图中删除该实例，从而禁止该实例的流量。

*eureka.instance.lease-expiration-duration-in-seconds:* eureka服务器在收到最后一次心跳之后等待的时间（以秒为单位），然后它才能从其视图中删除此实例，并禁止该实例的流量。 将此值设置过长可能意味着即使实例不存在，流量也可以路由到实例。将此值设置过小可能意味着，实例可能由于临时网络故障而被取出流量。此值设置为至少高于leaseRenewalIntervalInSeconds中指定的值*eureka.instance.preferIpAddress:*设为true，应用程序注册服务时使用的是IP地址，不是主机名，默认为false

## 实践总结

当eurekaserver集群是两台时，如果eurekaclient的defaultZone只指定一台，eurekaserver集群正常时不会出现问题，集群内部数据会自动同步；但当指定的那台eurekaserver服务down机后，eurekaserver集群的另一台服务不能接收到心跳信息，导致该eurekaclient会被移除；所以eurekaclient的defaultZone指定多个可以防止此现象发生。

## 问题

提供服务的集群注册到eurekaserver后，服务调用者以客户端负载均衡调用，当集群服务中的一台服务机器down机后，eurekaserver需要一段时间才会移除该服务，这段时间调用者可能会访问到该服务，导致访问异常。

经测试，Feign通过Ribbon的可以完成retry功能，需要修改配置，后面Feign+Ribbon客户端负载均衡详细说明。

# REST客户端Feign

Feign是一个声明式的WebService客户端。使用Feign能让编写WebService客户端更加简单，它的使用方法是定义一个接口，然后在接口上添加注解，同时也支持JAX-RS标准的注解。Feign也支持可插拔式的编码器和解码器。SpringCloud对Feign进行了封装，使其支持SpringMVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

## 服务端

注册到eureka的服务端提供服务，Feign来调用。

服务端代码：

|  |
| --- |
| @RestController  **public** **class** MyController {    @Value("${server.port}")  **private** String port;    @RequestMapping(value="/hello/{name}")  **public** String sayHello(@PathVariable String name){    **return** port+": Hello "+name;  }  } |

注册到eureka的服务名成是spring-server，配置如下：

#该服务注册到eureka

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

#应用名称

spring:

application:

name: spring-server

## feign客户端调用

### pom依赖

|  |
| --- |
| <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.tk</groupId>  <artifactId>eureka-client-feign</artifactId>  <version>0.0.1-SNAPSHOT</version>    <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  </parent>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>    <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.8</java.version>  </properties>    <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>  <!--暴露各种指标-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  </dependencies>  </project> |

### 配置文件

|  |
| --- |
| spring:  application:  name: feign  server:  port: 8090  eureka:  client:  serviceUrl:  defaultZone: http://eureka-1:8761/eureka/ |

|  |
| --- |
|  |

### SpringServerClient

|  |
| --- |
| @FeignClient("spring-server")  **public** **interface** SpringServerClient{    @RequestMapping(method = RequestMethod.***GET***, value = "/hello/{name}")  String getInfo(@PathVariable("name") String name);  } |

创建SpringServerClient接口，添加@FeignClient注解，该接口则以rest形式调用在eureka上注册名称是“spring-server”的服务,该值可以使用占位符${feignClient}，@RequestMapping中的value表示具体访问路径，如果访问路径中有参数，必须通过注解@PathVariable指定。

### Application启动类

|  |
| --- |
| @EnableEurekaClient//注册到eureka服务器  @EnableFeignClients//开启加载@FeignClient注解的功能  @SpringBootApplication  **public** **class** ApplicationFeign {    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ApplicationFeign.**class**, args);  }  } |

### 提供服务的controller

|  |
| --- |
| @RestController  **public** **class** Controller {    @Autowired  **private** SpringServerClient springServerClient;    @RequestMapping("/index/{name}")  **public** String getMessage(@PathVariable("name") String name){  **return** springServerClient.getMessage(name);  }  } |

可以注解注入上面的SpringServerClient，直接调用。

执行该application.java，启动后访问localhost:8090/index/Beijing,返回Hello Beijing。

### 相关注解

@FeignClient：表示是一个Feign客户端，需要指定name属性

@EnableFeignClients:开启Feign自动配置，扫描@FeignClient

## Feign Logging

为每个创建的Feign客户端创建一个记录器。默认情况下，记录器的名称是用于创建Feign客户端的接口的完整类名。 Feign记录只响应DEBUG级别。

例如：

application.yml

|  |
| --- |
| #feignClient的记录配置，logging.level后面是FeignClient的全路径名，值只支持DEBUG  logging.level.com.tk.client.SpringServerClient: DEBUG |

您可以为每个客户端配置的Logger.Level对象告诉Feign记录多少,级别有以下几种：

* NONE，无记录（DEFAULT）。
* BASIC，仅记录请求方法和URL以及响应状态代码和执行时间。
* HEADERS，记录基本信息以及请求和响应头。
* FULL，记录请求和响应的头，主体和元数据。

示例：

|  |
| --- |
| @Configuration  public class Configuration {  @Bean  Logger.Level feignLoggerLevel() {  return Logger.Level.FULL;  }  } |

访问时的日志中如下：

|  |
| --- |
| [SpringServerClient#getMessage] <--- HTTP/1.1 200 (624ms)  [SpringServerClient#getMessage] content-length: 18  [SpringServerClient#getMessage] content-type: text/plain;charset=UTF-8  [SpringServerClient#getMessage] date: Wed, 01 Mar 2017 06:49:14 GMT [SpringServerClient#getMessage] x-application-context: spring-server:server-2:8082  [SpringServerClient#getMessage] 8082: Hello bejing [SpringServerClient#getMessage] <--- END HTTP (18-byte body) |

## Feign支持Hystrix

1.Hystrix是容错管理工具，对超时、异常等服务调用快速进行响应，但访问异常时可以执行指定的逻辑。

添加依赖

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

### Fallback属性

2.修改SpringServerClient,添加fallback属性，如下，其中ServerClientFallBack.class是接口SpringServerClient的实现类，当Feign调用spring-server服务失败时，自动处理ServerClientFallBack.java中的实现逻辑。

|  |
| --- |
| @FeignClient(name="spring-server",fallback=ServerClientFallBack.**class**)  **public** **interface** SpringServerClient {  @RequestMapping(method = RequestMethod.***GET***, value = "/hello/{name}")  String getInfo(@PathVariable("name") String name);  } |

3.ServerClientFallBack.java

|  |
| --- |
| @Component  **public** **class** ServerClientFallBack **implements** ServerClient{  **public** String getInfo(String name) {  **return** "get info fail !";  }  } |

注意该实现类必须实例化bean，需要加注解@Component。

启动后访问localhost:8090/index/Beijing，正常情况返回Hello Beijing，当configserver异常时，执行fallback返回“get info fail！”。

### Fallbackfactory属性

有时希望异常时返回异常信息，可以使用@FeignClient的fallbackfactory属性。

1. 修改SpringServerClient.java，给@FeignClient添加fallbackfactory属性

|  |
| --- |
| @FeignClient(name="spring-server",fallbackFactory=ServerClientFallBackFactory.**class**)  **public** **interface** SpringServerClient {  @RequestMapping(method = RequestMethod.***GET***, value = "/hello/{name}")  String getInfo(@PathVariable("name") String name);  } |

1. ServerClientFallBackFactory.java

|  |
| --- |
| @Component  **public** **class** ServerClientFallBackFactory **implements** FallbackFactory<ServerClient>{  @Override  **public** ServerClient create(Throwable cause) {  **return** **new** ServerClientFallBack(){  @Override  **public** String getInfo(String name) {  **return** "get info fail !"+cause.getMessage();  }  };  }  } |

ServerClientFallBackFactory.java需要实现FallbackFactory<T>接口,T是对应加了@FeignClient注解的接口，重写create方法，可以从参数Throwable中获取异常信息。

启动后，若访问异常时，执行fallbackfactory中的逻辑。

若fallback和fallbackfacty属性都存在，首先执行fallback，fallbackfacty无效。

## Feign+ribbon客户端负载均衡

Feign客户端默认集成了ribbon进行负载均衡和失败后retry功能，目前因为Feign默认集成了hystrix，hystrix的默认响应时间是1秒，需要修改，否则访问集群中的一台服务器失败还没重新请求别的机器时hystrix就已经响应了fallbcak。

### 服务端

三台注册到eureka的服务提供服务，提供不同的profile，spring.applcation.name都是spring-server，相当于集群模式。

配置文件：

|  |
| --- |
| #该服务注册到eureka  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  #应用名称  spring:  application:  name: spring-server  ---  #实例server-1  spring:  profiles: server-1  server:  #端口  port: 8081  eureka:  instance:  #ip映射名称，需要添加到hosts文件  hostname: server-1  #MyController注入的信息  word: message from server-1  ---  #实例server-2  spring:  profiles: server-2  server:  port: 8082  eureka:  instance:  hostname: server-2  word: message from server-2    ---  #实例server-3  spring:  profiles: server-3  server:  port: 8083  eureka:  instance:  hostname: server-3  word: message from server-3 |

Controller类

|  |
| --- |
| @RestController  **public** **class** MyController {  @RequestMapping("/getWord")  **public** String getWord(){  **return** word;  }  } |

Application.java类通过指定不同的profile启动三台实例。

|  |
| --- |
| @EnableEurekaClient  @SpringBootApplication  **public** **class** ApplicationServer {  **public** **static** **void** main(String[] args) {  SpringApplication app = **new** SpringApplication(ApplicationServer.**class**);  // app.setAdditionalProfiles("server-1");  app.setAdditionalProfiles("server-2");  // app.setAdditionalProfiles("server-3");  app.run(args);  }  } |

### 客户端

#### pom依赖

|  |
| --- |
| <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.tk</groupId>  <artifactId>eureka-client-feign</artifactId>  <version>0.0.1-SNAPSHOT</version>    <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  </parent>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>    <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.8</java.version>  </properties>    <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>  <!--暴露各种指标-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  </dependencies>  </project> |

#### 配置文件

|  |
| --- |
| spring:  application:  name: feign  #启用ribbon的retry功能，当第一次请求异常时，  cloud.loadbalancer.retry.enabled: **true**  server:  port: 8090  #将该服务注册到eureka服务器  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  #配置hystrix超时时间，默认是1s，如果太短，当请求的集群服务中一台服务挂掉后，  #不能等到retry功能启用就调用hystrix的rollback,实际大小需要根据retry次数设置  hystrix.command.default.execution:  #是否开启超时，默认true  # timeout.enabled: false  #超时时间，默认1000ms  isolation.thread.timeoutInMilliseconds: 5000 |

|  |
| --- |
|  |

#### SpringServerClient

|  |
| --- |
| @FeignClient("spring-server")  **public** **interface** SpringServerClient{    @RequestMapping(method = RequestMethod.***GET***, value = "/getWord")  String getWord();  } |

#### controller

|  |
| --- |
| @RestController  **public** **class** Controller {    @Autowired  **private** SpringServerClient springServerClient;    @RequestMapping("/ribbon/feign")  **public** String getWord(){  **return** springServerClient.getWord();  }} |

#### Application启动类

|  |
| --- |
| @EnableEurekaClient//注册到eureka服务器  @EnableFeignClients//开启加载@FeignClient注解的功能  @SpringBootApplication  **public** **class** ApplicationFeign {  @LoadBalanced  @Bean  **public** RestTemplate restTempalte(){  **return** **new** RestTemplate();  }    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ApplicationFeign.**class**, args);  }  } |

执行该application.java，启动后访问localhost:8090/ribbon/feign,返回spring-server一台服务中的信息(message from server-1），再次刷新访问会返回另一台服务的信息(message from server-2)，可以发现已经实现了客户端负载均衡。

关闭服务server-1，继续访问，由于eureka的注册信息更新需要一段时间，此时客户端还不知道server-1已经停止，所以还会请求server-1，此时会发现响应时间稍慢，是因为访问server-1时失败，feign会自动retry访问其他实例（如server-2），所以最终会返回正常的信息，而不是报服务找不到。

# SpringCloudBus

在SpringCloudConfig中，我们已经可以通过/refresh接口实现属性更新，但这只能更新具体实例中的配置，若随着系统的不断扩张，会变得越来越难以维护，而消息代理中间件可以很好的解决这一问题。

Bus目前针对rabbitmq和kafka这两个常用消息服务器开发了starter，使用时导入对应的starter，配置对应的连接信息即可。

先看个示例：

## ConfigServer服务端

### 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.tk</groupId>  <artifactId>config-server-kafka</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  </parent>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-kafka</artifactId>  </dependency>  <!-- <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency> -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</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-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

### 配置文件

**bootstrap.properties**

bootstrap.propertie优先加载，一般存放不会改变的配置

Git地址，git仓库中存储配置文件

|  |
| --- |
| spring.cloud.config.server.git.uri: https://github.com/mouwengang/config.git |

**application.properties**

|  |
| --- |
| spring.application.name=config-server-kafka  server.port=8888  #注册到eurekaserver配置  eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/  #kafka 配置  spring.cloud.stream.kafka.binder.zk-nodes=localhost:2181  spring.cloud.stream.kafka.binder.brokers=localhost:9092  #rabbitmq 配置  #spring.rabbitmq.host=localhost  #spring.rabbitmq.port=5672  #spring.rabbitmq.username=guest  #spring.rabbitmq.password=guest |

### Java代码

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

## ConfigClient客户端

### 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.tk</groupId>  <artifactId>config-client-kafka</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>config-server-kafka</name>  <description>Demo project for Spring Boot</description>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.3.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Camden.SR3</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-kafka</artifactId>  </dependency>  <!-- <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency> -->  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

### 配置文件

**bootstrap.properties**

|  |
| --- |
| spring.application.name=config-client-kafka  #连接configserver配置  spring.cloud.config.uri: http://localhost:8888  #注册到eurekaserver配置  eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/  #如果不能连接到configserver则快速失败  spring.cloud.config.failFast=true  #kafka  spring.cloud.stream.kafka.binder.zk-nodes=localhost:2181  spring.cloud.stream.kafka.binder.brokers=localhost:9092  #显示bus事件跟踪  spring.cloud.bus.trace.enabled=true  #rabbitmq  #spring.rabbitmq.host=localhost  #spring.rabbitmq.port=5672  #spring.rabbitmq.username=guest  #spring.rabbitmq.password=guest |

**application.yml**

|  |
| --- |
| server.port: 7002  ---  spring.profiles: client-2  server.port: 7003 |

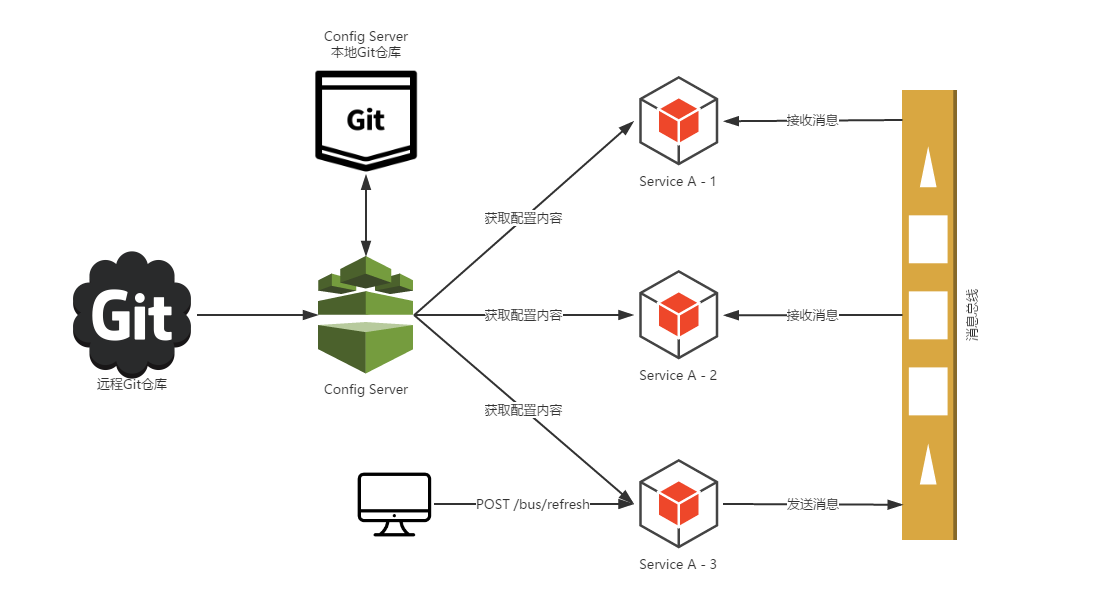
**java代码**

|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  @RestController  @RefreshScope  **public** **class** ConfigClientKafkaApplication {  @Value("${from}")  **private** String from;    @RequestMapping("/get")  **public** String get(){  **return** from;  }    **public** **static** **void** main(String[] args) {  SpringApplication app = **new** SpringApplication(ConfigClientKafkaApplication.**class**);  app.setAdditionalProfiles("client-2");  app.run(args);  }  } |

* 先启动eurekaserver服务，再启动configserver，最后启动configclient，先不指定profiles启动一个client实例,再指定profiles启动一个实例，访问任意一个client的/get返回初始化的信息。
* 接下来修改git仓库中的“from”属性配置，并发送POST请求到其中的一个服务/bus/refresh。
* 最后，我们再分别访问启动的两个config-clien的/get请求，此时这两个请求都会返回最新的git仓库中的from属性。

## 原理分析

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

[](http://blog.didispace.com/assets/5-6.png)

整个方案的[**架构**](http://lib.csdn.net/base/architecture)如上图所示，其中包含了Git仓库、Config Server、以及[**微服务**](http://lib.csdn.net/base/microservice)“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。

## 指定刷新范围

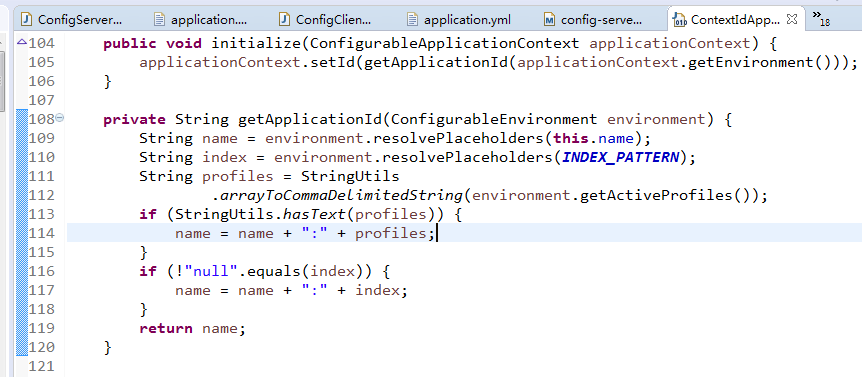
上面的例子中，我们通过向服务实例请求Spring Cloud Bus的/bus/refresh接口，从而触发总线上其他服务实例的/refresh。但是有些特殊场景下（比如：灰度发布），我们希望可以刷新微服务中某个具体实例的配置。

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。

**\*\*?\*\*-> destination的值是ApllicationId需要保证唯一，值是spring.application.name、**

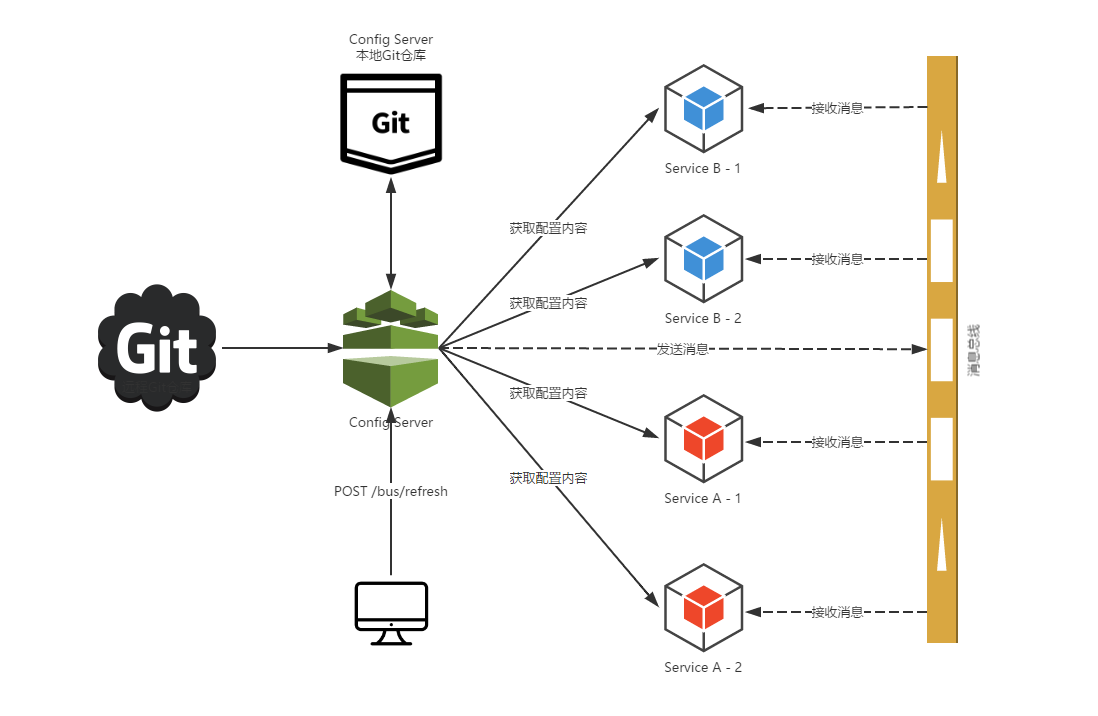
**active profiles 、server.port 和spring.application.index的组合，中间用“：”号连接。**



## 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：

[](http://blog.didispace.com/assets/5-7.png)

主要做了这些改动：

1. 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。
2. /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。

# Eureka

## 搭建Eureka服务器

### 依赖管理

**parent**

|  |
| --- |
| <parent>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-parent</artifactId>  <version>Camden.RELEASE</version>  <relativePath />  </parent> |

**dependency**

**注意:spring-cloud-starter-eureka-server中已经包含了spring-cloud-starter**

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

### 开启EurekaServer

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

### application.yml

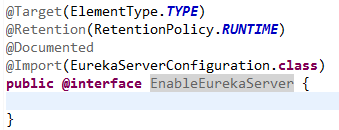
|  |
| --- |
| eureka:  instance:  hostname: localhost  client:  registerWithEureka: **true**  fetchRegistry: **true**  serviceUrl:  defaultZone: http://localhost:8761/eureka/ |

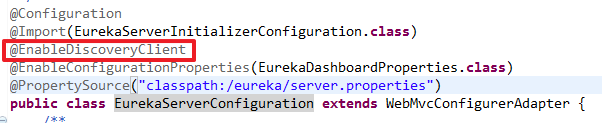
### 相关注解

@EnableEurekaServer

该注解用于开启Eureka服务功能

**@EnableEurekaServer**注解中包含**@EnableDiscoveryClient.**





## Eureka客户端

### 依赖管理

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

### 开启EurekaClient

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

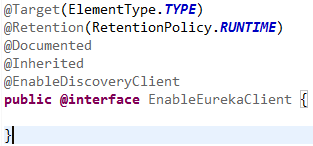
### application.yml

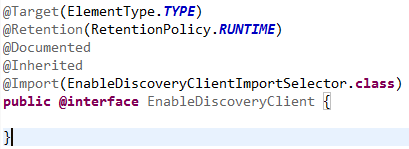
|  |
| --- |
| server:  port: 8080  eureka:  instance:  leaseRenewalIntervalInSeconds: 5  leaseExpirationDurationInSeconds: 6  client:  registerWithEureka: **true**  fetchRegistry: **true**  serviceUrl:  defaultZone: http://localhost:8761/eureka/  spring:  application:  name: cloud-eureka-client |

### 相关注解

@EnableEurekaClient注解用于明确指定Eureka作为注册中心服务端,并开启该项目为Eureka为客户端

@EnableDiscoveryClient注解用于开启服务注册客户端功能,但服务端可能是Eureka、Zookeeper、Console中的一个，根据classpath下的依赖来确定





该注解主要就是注册一个EnableDiscoveryClientImportSelector组件，该组件主要用于扫描spring.factories文件中配置的EnableDiscoveryClient实现类并注册为spring组件,spring cloud Eureka提供的EnableDiscoveryClient默认实现类为EurekaDiscoveryClientConfiguration,该类为一个配置bean

# Hystrix(熔断器)

Hystrix是一个为解决延迟和容错问题的库,旨在隔离有问题的远程系统服务,避免级联故障.

它是CircuitBreaker的一个实现

## 相关概念

fallback

当目标方法发生异常后Hystrix会调用指定的fallback方法,相当于做一个服务降级处理

熔断器

当使用@HystrixCommand注解的方法中发生异常,并且超过阙值(默认为5秒钟发生20 次异常),那么该方法的熔断器将会被开启,当熔断器开启后,如果有访问目标方法的请求, 那么目标方法不会被执行,直接执行指定的fallback(达不到阙值,会先尝试执行目标方法, 如果有异常才会执行指定的fallback方法)

HystrixCommand

被@HystrixCommand的方法就是一个HystrixCommand实例

## 依赖管理

|  |
| --- |
| <parent>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-parent</artifactId>  <version>Camden.SR3</version>  <relativePath />  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <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.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  </dependencies> |

## 开启Hystrix

通过添加@EnableHystrix注解开启Hystrix功能

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

## 相关注解

### @EnableHystrix

该注解用于开启Hystrix功能,使@HystrixCommand等注解起作用.

配置在被@Configuration标注的类上,并且一个项目中使用一次即可

### @HystrixCommand

在需要熔断器功能的方法上使用该注解可以为该方法开启熔断器功能

|  |
| --- |
| **public** **@interface** HystrixCommand {  /\*\*  \* The command group key is used for grouping together commands such as for  \* reporting, alerting, dashboards or team/library ownership.  \* 对HystrixCommand分组  \* 默认为被标注的方法所属类的类名  \*/  String groupKey() **default** "";  /\*\*  \* Hystrix command key.  \* 该属性用于为方法指定一个的标识  \* 默认为被标注的方法名  \*/  String commandKey() **default** "";  /\*\*  \* The thread-pool key is used to represent a  \* HystrixThreadPool for monitoring, metrics publishing, caching and other  \* such uses.  \* 该属性需要进一步调研  \*/  String threadPoolKey() **default** "";  /\*\*  \* Specifies a method to process fallback logic.  \* 指定发生异常时,要执行的方法名,该方法需要在同一个类中定义,并且回调方法需要  \* 和原方法有相同的参数和返回值  \*/  String fallbackMethod() **default** "";  /\*\*  \* Specifies command properties.  \* 为该方法单独配置Hystrix属性,在application.yml中可以提供default配置(同样  \* 可以在application.yml中通过commandKey为具体的方法配置个性化信息)  \*/  HystrixProperty[] commandProperties() **default** {};  /\*\*  \* Specifies thread pool properties.  \* 回调方法会使用线程池执行,该属性用于配置线程池相关信息  \*/  HystrixProperty[] threadPoolProperties() **default** {};  /\*\*  \* Defines exceptions which should be ignored and wrapped to throw in  \* HystrixBadRequestException.  \* 通过该属性可以排除一些不想触发回调的异常  \* 默认情况下HystrixBadRequestException不会触发回调  \*/  Class<? **extends** Throwable>[] ignoreExceptions() **default** {};  /\*\*  \* 该属性需要进一步调研  \* For more information see {@link ObservableExecutionMode}.  \*/  ObservableExecutionMode observableExecutionMode() **default** ObservableExecutionMode.***EAGER***;  } |

### @HystrixCollapser

根据说明应该是将多个目标方法的fallback合并到一个方法中进行处理,但该注解的用法还需要进行进一步的调研,还不明确具体的用途,官方给的说明是:

|  |
| --- |
| Suppose you have some command which calls should be collapsed in one backend call .For this goal you can use @HystrixCollapser annotation |

参考资料:

https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#hystrix-collapser

### @HystrixProperty

配合@HystrixCommand使用,用于为HystrixCommand传入配置属性,详情见@HystrixCommand注解的commandProperties和threadPoolProperties属性

## 相关属性说明

### Hystrix属性

**spring.cloud.circuit.breaker.enabled**

开启circuit breaker功能,默认为true

### HystrixCommand相关属性

**fallback.enabled**

是否开启执行fallback的功能

**circuitBreaker.enabled**

是否开启熔断器功能

execution.isolation.thread.timeoutInMilliseconds

设置线程超时时间,超过该值,hystrix就认为该方法执行失败就会调用fallback方法向客户端返回结果,默认为1000 ms

注: HystrixCommand相关属性主要用于配置Hystrix在拦截方法时使用到的一些信息,如熔断 器开启后重试目标方法的时间间隔circuitBreaker.sleepWindowInMilliseconds,单位为毫 秒.

HystrixCommand相关属性可以在@HystrixCommand注解中通过commandProperties属 性进行指定,也可以通过application.yml文件进行指定,如果在application.yml进行配置, 那么需要加上hystrix.command.default或者hystrix.command.{commandKey}作为前缀,默 认情况下commandKey就是目标方法名,如hystrix.command.default.fallback.enabled=true

使用hystrix.command.default作为前缀,那么就是为所有的方法进行配置,如果使用 hystrix.command.{commandKey}作为前缀那么就是为具体的方法进行配置

**HystrixCommand可用属性配置地址https://github.com/Netflix/Hystrix/wiki/Configuration**

## 异常捕捉

### 问题与解决方案

正常情况下,如果我们为HystrixCommand配置了fallback方法,那么发生异常时会直接执行fallback指定的方法,在fallback指定的方法中我们获取不到异常信息,如果希望拿到异常信息,可以使用HystrixPlugin捕获异常

### HystrixPlugin

Hystrix支持以下几种Plugin

* HystrixEventNotifier
* HystrixConcurrencyStrategy
* HystrixMetricsPublisher
* HystrixPropertiesStrategy
* HystrixCommandExecutionHook

### HystrixCommandExecutionHook

**HystrixCommandExecutionHook**是HystrixPlugin中的一种

在HystrixCommandExecutionHook的**onExecutionError**方法中可以获取到exception信息

### 注册HystrixPlugin

HystrixPlugins.getInstance().registerCommandExecutionHook(new **HystrixExceptionHook**());

其中**HystrixExceptionHook**是HystrixCommandExecutionHook的实现类,可以在项目启动时调用该方法

## 源码分析

Hystrix的原理就是使用AOP技术拦截所有添加了@HystrixProperty注解的方法

1. @EnableHystrix

通过该注解引入@EnableCircuitBreaker注解

1. @EnableCircuitBreaker

通过该注解注册**EnableCircuitBreakerImportSelector.class**

|  |
| --- |
| @Target(ElementType.***TYPE***)  @Retention(RetentionPolicy.***RUNTIME***)  @Documented  @Inherited  @Import(EnableCircuitBreakerImportSelector.**class**)  **public** **@interface** EnableCircuitBreaker {  } |

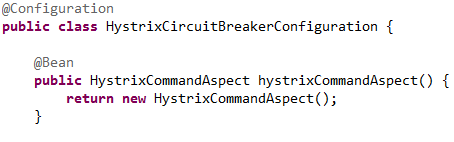
1. **EnableCircuitBreakerImportSelector**负责在spring.factories文件中查找EnableCircuitBreaker的实现类,并注册为spring组件

spring-cloud-netflix-core-version.jar包中spring.factories文件下的配置

|  |
| --- |
| org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker=\  org.springframework.cloud.netflix.hystrix.HystrixCircuitBreakerConfiguration |

1. **HystrixCircuitBreakerConfiguration**

该类是一个添加了**@Configuration**注解的类,在该类中声明了一个HystrixCommandAspect组件



1. **HystrixCommandAspect**

该类中定义了切入点和通知的具体内容



# Hystrix Dashboard

Hystrix的功能不止于熔断功能,他还可以对外界暴露方法的一些基本信息.比如生产环境下我们可能比较关心某些方法被调用的频率以及方法是否发生了故障,配合Hystrix Dashboard,这些信息可以在浏览器上展示出来.

通过HystrixDashboard可以对每一个HystrixCommand进行监控,包括访问频率,出错次数,线程池使用情况,是否开启熔断器等.HystrixDashboard就是为Hystrix功能服务的.

## 依赖管理

在Hystrix相关依赖的基础上需要添加如下依赖

下面的是dashboard的依赖是对actuator的依赖.

说明:actuator是spring boot的一个功能,actuator负责向外界暴露应用的信息.

hystrix利用actuator向外界暴露hystrix的相关信息,dashboard主动请求hystrix并将信息以一定的格式向用户展示

|  |
| --- |
| <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> |

## 开启HystrixDashboard

通过@EnableHystrixDashborad注解可以开启HystrixDashboard功能

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

## application.yml

需要暴露hystrix.stream作为endpoints,endpoints是spring boot的一个基础功能,用于暴露应用信息,比如info、health、beans等信息

|  |
| --- |
| spring:  boot:  admin:  routes:  endpoints: hystrix.stream |

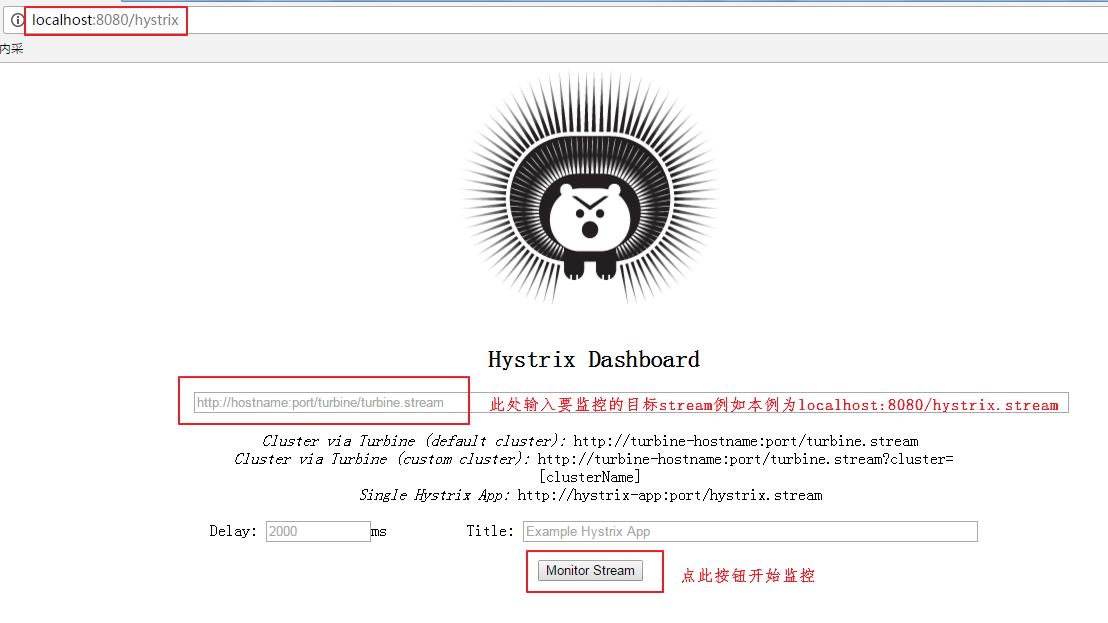
## 相关注解

**@EnableHystrixDashborad**

用于开启HystrixDashboard功能

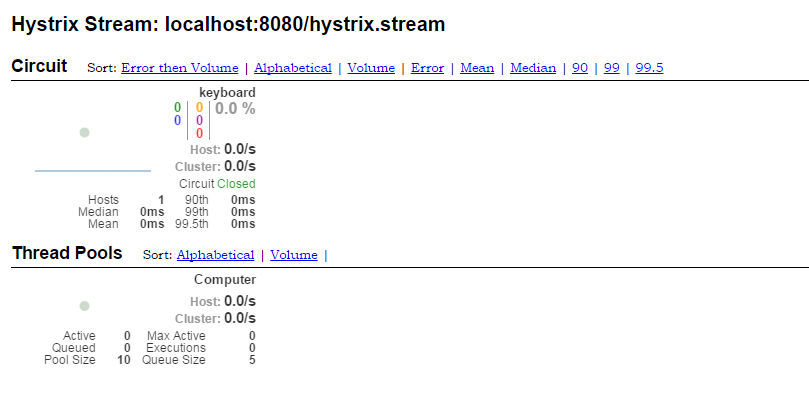
## 前台展示

访问http://serviceIp:port /hystrix如果有应用上下文需要带上应用上下文，看到如下界面后在中间部位的输入框中输入要监控的stream地址如：localhost：8080/hystrix.stream,然后点击MonitorStream按钮就可以看到具体的监控画面了



**监控画面**

注:下图中各属性详细解释待补充

****

# turbine

turbine的主要作用就是将多个项目中Hystrix Dashboard的信息整合到一起进行展示

使用时turbine需要明确三点:

* 要监控哪些项目,通过appConfig属性指定
* 将所有项目分成几个cluster,通过clusterConfig属性指定
* 将项目分到不同的cluster的规则,通过clusterNameExpression属性指定

具体配置说明见application.yml

## 依赖管理

**parent**

|  |
| --- |
| <parent>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-parent</artifactId>  <version>Camden.RELEASE</version>  <relativePath />  </parent> |

**dependency**

|  |
| --- |
| <dependencies>  <!--dashboard依赖-->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>  </dependency>  <!--turbine依赖-->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-turbine</artifactId>  </dependency>  <!--eureka依赖-->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!--actuator依赖-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  </dependencies> |

## 开启Trubine

使用@EnableTurbine注解开启turbine功能

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

## application.yml

|  |
| --- |
| server.port: 8082  spring:  application:  name: hystrix-turbine  #eureka相关配置  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  #turbine相关配置  turbine:  #配置需要聚合的应用,值为目标应用在Eureka中注册的serviceId值  appConfig: hystrix,hystrix-dashboard  aggregator:  #该配置相当于对所有应用进行分组监控,这里的clusterConfig值相当于创建几个分组  #只有先在这里配置才能在后续根据这里配的值进行监控  #例如:  # http://localhost:8080/turbine.stream?cluster=HYSTRIX  # http://localhost:8080/turbine.stream?cluster=HYSTRIX-DASHBOARD  #注: 经测试这里的值必须大写才能正常工作  clusterConfig: HYSTRIX,HYSTRIX-DASHBOARD  #clusterNameExpression相当于配置选取组员的规则  #clusterNameExpression的值从InstanceInfo类中选,默认为appName  #metadata['cluster']的意思就是InstanceInfo中字段metadata(一个map)中key为cluster  clusterNameExpression: metadata['cluster']  #结合appConfig、clusterConfig、clusterNameExpression三个配置的含义就是  #将appConfig包含的应用中metadata.cluster值为HYSTRIX的应用分为一组,值为HYSTRIX-DASHBOARD的应用分为一组  ###############################    #combineHostPort属性还没有理解明白,下面是官方给的说明  # by default Spring Cloud allows Turbine to use the host and port to allow multiple processes per host, per cluster.  #If you want the native Netflix behaviour built into Turbine that does not allow multiple processes per host, per cluster  #(the key to the instance id is the hostname), then set the property turbine.combineHostPort=false.  combineHostPort: **true** |

# ribbon

ribbon的作用就是实现客户端负载均衡,可以配合RestTemplate和Feign使用

## 依赖管理

|  |
| --- |
| <parent>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-parent</artifactId>  <version>Camden.SR3</version>  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <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>  </dependencies> |

## 使用ribbon

目前发现有两种方式使用ribbon做客户端负载均衡:

* 配合RestTemplate
* 配合Feign

### 配合RestTemplate使用

定义RestTemplate

|  |
| --- |
| @Bean  @LoadBalanced  **public** RestTemplate restTemplate(){  **return** **new** RestTemplate();  } |

使用RestTemplate

|  |
| --- |
| @Autowired  **private** RestTemplate client;  @RequestMapping("/helloRibbon")  **public** String hello(){  ResponseEntity<String> forEntity =  client.getForEntity("http://hystrix/helloSpringBoot",String.**class**);  String body = forEntity.getBody();  **return** body;  } |

使用@Bean注解注册一个RestTemplate实例,通过@LoadBalanced注解,标注该RestTemplate实例后端采用负载均衡

使用RestTemplate时,访问路径中的hystrix不是真实的服务地址,而是服务提供方在注册中心中注册的serviceId值,ribbon通过serviceId在注册中心找到该服务下的实例列表,然后ribbon通过一定的策略选出具体请求哪个服务实例.

### 配合Feign使用

如果使用Feign,那么默认使用ribbon做客户端负载均衡,详情见Feign中的介绍

### Ribbon中的retry

在生产中一个服务可能对应多个服务实例,当ribbon请求的服务实例出现故障后,ribbon会使用spriing retry功能进行重试,该功能默认是关闭的,需要使用spring.cloud.loadbalancer.retry.enalbed=true开启该功能

ribbon中的retry从两方面进行,一是针对当前故障实例的重试,二是重新选择一个实例进行重试.

经测试配合Feign使用Ribbon时如果使用默认配置retry功能会有问题,因为Feign默认开启Hystrix功能,Hystrix的默认超时时间是1s,当使用feign调用某一个服务实例失败后进行重试时,重试花费的时间一般都会超过1s,所以在重试功能执行完成之前hystrix就将本次服务调用视为超时,直接调用fallback方法生成返回结果.所以在使用时可以将超时时间调大,或关闭feign的hystrix功能.

## application.yml

ribbon相关配置的格式就是<clientName>.ribbon.propertyName=propertyValue

clientName是目标服务的serviceId值,如果去掉clientName直接使用ribbon.propertyName进行配置就是设置默认配置.

其中propertyName的可选值可以在CommonClientConfigKey类中查看

propertyValue默认值可以在DefaultClientConfigImpl类中查找

|  |
| --- |
| server:  port: 8085  spring:  application:  name: ribbon  cloud:  loadbalancer:  retry:  #开启负载均衡重试功能  enabled: **true**  hystrix:  ribbon:  #当前实例最大重试次数  MaxAutoRetries: 1  #最大重试实例数  MaxAutoRetriesNextServer: 1  #eureka配置  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  ########################################  #ribbon重试相关配置说明  #MaxAutoRetries  #MaxAutoRetriesNextServer  #当访问到故障请求的时候,ribbon会再尝试访问MaxAutoRetries次当前实例,  #如果不行,就换一个实例进行访问,如果还是不行,  #再换一次实例访问,更换次数由MaxAutoRetriesNextServer配置决定,  #如果依然不行,返回失败信息 |

# sleuth

## sleuth简介

Spring Cloud Sleuth 是 Spring Cloud 框架中实现分布式追踪的组件。通过和 OpenZipkin 集成可以实现将分布式系统中各系统的追踪信息进行汇总,并生成系统依赖关系和每一次访问的调用信息(如整个调用链上每一个环节的消耗时间,可以方便的查看到每一次系统调用间通信的延迟.)

## 基本概念

* **span**

一个Span就是一个最基本的任务单位。每个 Span 一定是唯一的，并通过一个64位的 Span ID 和 64位的 Trace ID 进行区分。除此以外，每个 Span 还会包含描述，带时间戳的事件，键值标签等信息。

经测试在系统接收到一个请求时会开启一个span(如spanA),如果请求头中包含spanId,说明该请求来自另一个sleuth项目,那么将请求头中的spanId作为该span(spanA)的parent span,当调用RestTemplate或Feign客户端时会开启另一个span(如spanB),并且将spanA作为spanB的parent span当远程调用返回响应后关闭该spanB,当本次请求全部完成后给客户端做出响应前关闭spanA

* **Trace**

Trace 就是一群 Span 的集合,该集合的结构由span中的parent span属性串成一个链,这条链就是一个trace,可以理解为用户的一次请求

* **Annotation**

每一个span生命周期内可能都会发生一些我们关心的事,可以通过Annotation来标记这些事件的发生,annotation功能其实就是在一个时间点打一个tag,记录下时间戳和tag名.在日志或zipkin界面中看到某一个tag时我们就知道在哪个时间点发生了什么事

其中一些重要的 Annotations 可以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 Annotation 表示了一个 Span 的开始。

**sr - Server Received** - 服务端收到了一个请求，并开始处理开请求。对 cs 的时间戳和 sr 的时间戳做减法，可以计算出网络延时。

**ss - Server Sent** - 向客户端发送 Response （标示着一个请求处理的完成）。对 sr 时间戳 和 ss 时间戳做减法，可以计算出服务器处理一个请求所需要的时间。

**cr - Client Received** - 客户端成功收到了来自服务端的 Response (标示着一个 Span 的结束)。对 cs 时间戳 和 cr 时间戳做减法，可以计算出客户端从服务端获取一个 Response 所需的总时间。

## 在项目中使用sleuth

### 单独使用sleuth

一般sleuth和zipkin结合使用,但可能我们只希望使用sleuth,而不使用zipkin.单独使用sleuth对我们项目的影响就是在我们的日志中添加traceID,spanID等标记信息,没有其他影响.

添加依赖后系统会自动开启sleuth功能,不需要显式开启

单独使用sleuth,需要在项目中添加如下依赖.

|  |
| --- |
| <dependencyManagement>    <dependencies>      <dependency>        <groupId>org.springframework.cloud</groupId>         <artifactId>spring-cloud-dependencies</artifactId>         <version>Brixton.RELEASE</version>         <type>pom</type>        <scope>import</scope>      </dependency>    </dependencies>  </dependencyManagement>   <dependency>     <groupId>org.springframework.cloud</groupId>     <artifactId>spring-cloud-starter-sleuth</artifactId>   </dependency> |

### sleuth通过http和zipkin结合

sleuth通过http和zipkin结合的意思就是sleuth使用http通过header将采集到的信息发送到zipkin中,只要将相关依赖添加到classpath下,发送信息到zipkin的功能自动开启

下图为截取到的sleuth发送给zipkin的header信息,包括span-name,spanid,parentspanid和traceid等



header信息

**配置:**

* sleuth默认会将采集到的信息发送到http://localhost:9411中,如果zipkin server地址不是该默认地址可以通过spring.zipkin.baseUrl指定,该属性配置可以在ZipkinProperties类中找到
* sleuth有采样频率的概念,因为如果sleuth把每一条采集到的信息都发送到zipkin中可能会影响系统本身的性能,可以通过spring.sleuth.sampler.percentage属性配置采样频率,默认为0.1f,就是10个请求会采集一次.如果一次请求涉及到了多个sleuth服务,并且每个sleuth配置的采样频率不同,那么请求方的采样频率影响服务方的采样频率,在header中有一个x-b3-sampled,该header记录了本次请求调用方是否进行了采样,如果调用方采样,那么服务方在本次请求中也要进行采样,这样才能在zipkin UI中看到一个完整的链路.

采样频率的取值范围为0-1,只保留一位小数,不支持两位小数如0.01是不支持的

**依赖:**

|  |
| --- |
| <dependencyManagement>    <dependencies>      <dependency>        <groupId>org.springframework.cloud</groupId>        <artifactId>spring-cloud-dependencies</artifactId>        <version>Brixton.RELEASE</version>        <type>pom</type>        <scope>import</scope>      </dependency>    </dependencies>  </dependencyManagement>   <dependency>     <groupId>org.springframework.cloud</groupId>     <artifactId>spring-cloud-starter-zipkin</artifactId>   </dependency> |

### sleuth通过stream和zipkin结合

sleuth通过stream和zipkin结合的意思就是sleuth使用spring cloud stream将采集到的信息发送到zipkin中, 只要将相关依赖添加到classpath下,发送信息到zipkin的功能自动开启

配置:

* sleuth默认会将采集到的信息发送到http://localhost:9411中,如果zipkin server地址不是该默认地址可以通过spring.zipkin.baseUrl指定,该属性配置可以在ZipkinProperties类中找到
* sleuth有采样频率的概念,因为如果sleuth把每一条采集到的信息都发送到zipkin中可能会影响系统本身的性能,可以通过spring.sleuth.sampler.percentage属性配置采样频率,默认为0.1f,就是10个请求会采集一次.如果一次请求涉及到了多个sleuth服务,并且每个sleuth配置的采样频率不同,那么请求方的采样频率影响服务方的采样频率,在header中有一个x-b3-sampled,该header记录了本次请求调用方是否进行了采样,如果调用方采样,那么服务方在本次请求中也要进行采样,这样才能在zipkin UI中看到一个完整的链路.
* 配置stream服务的地址,本项目中使用的rabbitmq,所以需要配置rabbitmq信息

spring.rabbitmq.host: \*.\*.\*.\*

spring.rabbitmq.port: 5672

spring.rabbitmq.username: username

spring.rabbitmq.password: password

|  |
| --- |
| <dependencyManagement> (1)    <dependencies>      <dependency>        <groupId>org.springframework.cloud</groupId>        <artifactId>spring-cloud-dependencies</artifactId>        <version>Brixton.RELEASE</version>        <type>pom</type>        <scope>import</scope>      </dependency>    </dependencies>  </dependencyManagement>       <dependency> (2)         <groupId>org.springframework.cloud</groupId>         <artifactId>spring-cloud-sleuth-stream</artifactId>     </dependency>     <dependency> (3)         <groupId>org.springframework.cloud</groupId>         <artifactId>spring-cloud-starter-sleuth</artifactId>     </dependency>     <!-- EXAMPLE FOR RABBIT BINDING -->     <dependency> (4)         <groupId>org.springframework.cloud</groupId>         <artifactId>spring-cloud-stream-binder-rabbit</artifactId>     </dependency> |

# Zipkin

zipkin是一个分布式追踪系统,它可以收集一些用于定位微服务架构下系统延迟等问题所需要的带有时间戳的数据.zipkin不但可以收集数据还提供zipkin UI负责展示收集到的数据,可以对数据进行组织和过滤.

**对trace过滤的维度:**

application name

starttime-endtime

duration(持续时间)

limit(展示结果条数)

annotations

traceId

**关于异常调用的测试:**

sleuth有采样频率的设置,也就是说在采样频率不是100%的情况下,如果出现异常访问,异常访问有概率不会被采集,比如,如果采样频率为10%,那么发生异常访问时90%的可能异常访问不会被采集到.

当产生异常的方法使用了Hystrix功能,那么当发生异常时,在zipkin中也不会展示为异常调用

## 搭建zipkin服务

下面提到的搭建zipkin服务都是基于spring boot进行的

### 最简单的Zipkin Server服务

这种搭建方式会通过http的形式接收采样信息

**依赖:**

|  |
| --- |
| <parent>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-parent</artifactId>  <version>Camden.SR3</version>  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>io.zipkin.java</groupId>  <artifactId>zipkin-server</artifactId>  </dependency>  <dependency>  <groupId>io.zipkin.java</groupId>  <artifactId>zipkin-autoconfigure-ui</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

**开启zipkinserver**

使用@EnableZipkinServer注解可以开启zipkinserver功能

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

**application.yml配置**

一般zipkin运行在9411端口上

|  |
| --- |
| server:  port: 9411  spring:  application:  name: zipkin |

### 基于消息中间件的 Zipkin Server

**依赖:**

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.0.RELEASE</version>  <relativePath/>  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>  </dependency>  <dependency>  <groupId>io.zipkin.java</groupId>  <artifactId>zipkin-autoconfigure-ui</artifactId>  <scope>runtime</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Brixton.SR5</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

**开启ZipkinServer**

和简单的ZipkinServer开启方式不同,如果用消息中间件接收信息,需要使用@EnableZipkinStreamServer

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

**application.yml配置**

需要配置所使用的消息中间件的信息,此处配置为rabbitmq

|  |
| --- |
| spring.application.name: rabbit-zipkin  server.port: 9411  spring.rabbitmq.host: 10.130.202.57  spring.rabbitmq.port: 5672  spring.rabbitmq.username: UsernameToBeReplaced  spring.rabbitmq.password: PasswordToBeReplaced |

### 为ZipkinServer配置数据库

如果不为ZipkinServer配置存储策略,那么ZipkinServer默认将追踪信息存储到内存中,当系统重启后信息就被清除了

可以为ZipkinServer配置数据库,将采集到的信息持久化

如果在[最简单的ZipkinServer服务]的基础上添加持久化的支持,以mysql为例,需要添加以下依赖表

|  |
| --- |
| <dependency>**(1)**  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>**(2)**  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>**(3)**  <groupId>io.zipkin.java</groupId>  <artifactId>zipkin-autoconfigure-storage-mysql</artifactId>  </dependency> |

在配置文件中添加数据源配置并指定zipkin storage策略为mysql,并开启初始化表功能

|  |
| --- |
| spring:  datasource:  schema: classpath:/mysql.sql  url: jdbc:mysql://localhost:3306/test  username: root  password: root  initialized: **true**  zipkin:  storage:  type: mysql |

注意:

配置好数据库之后不需要用户自己创建任何表,在zipkin-storage-mysql-version.jar中包含一个mysql.sql文件,如果数据库中没有zipkin需要的表,zipkin使用该文件对数据库进行初始化

注意:

如果在[基于简单消息中间件的ZipkinServer]基础上添加数据库配置,那么在添加依赖时,只需要添加依赖表中(1)和(2)即可,spring-cloud-sleuth-zipkin-stream中已经有了对(3)的依赖

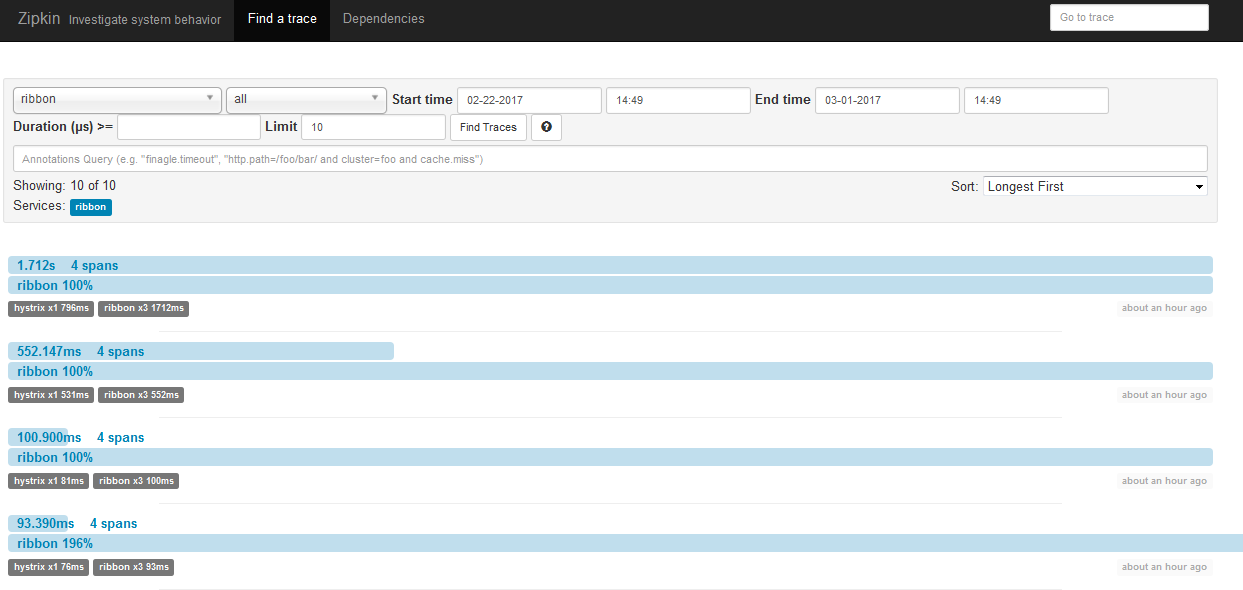
## Zipkin UI

Zipkin UI主要用来以图表的形式展示zipkin收集得到的信息,zipkin UI中可以看到请求调用链路.Zipkin以trace表示一次请求,又把每个trace拆分为若干个有依赖关系的span,在微服务架构中,一个用户的请求可能会被后台的若干个服务处理,这完整的一次用户请求就是一条链路trace,,每个调用处理请求的服务可以理解为一个span,这个服务也可能继续调用其他的服务,因此形成了一个span的树形结构,以体现服务间的调用关系.

除了展示span间的依赖关系,还可以看到每个span的耗时情况,可以一目了然的看到各个服务的性能状况.

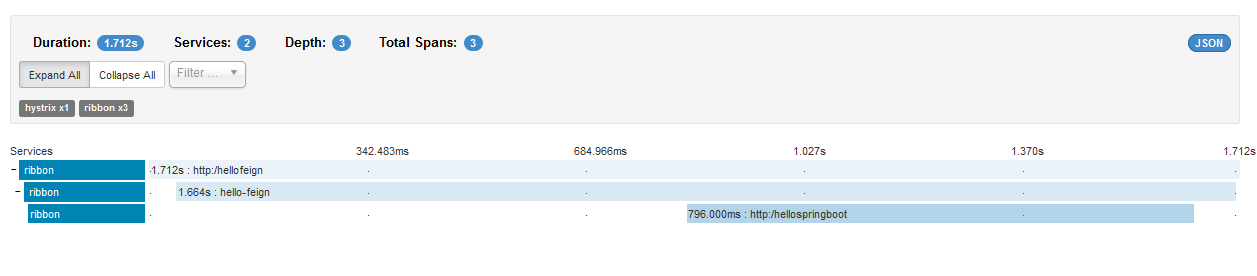
### 展示正常trace信息

进入zipkin UI首页,指定好过滤条件后点击find traces可以得到trace列表,如下图:



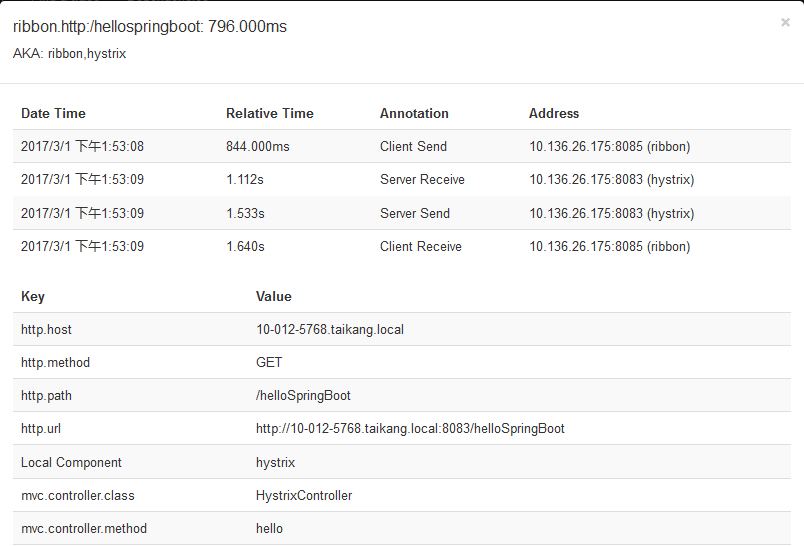
根据服务名/时间/annotation对trace进行过滤后的结果

点击某一个trace后会看到下面的信息



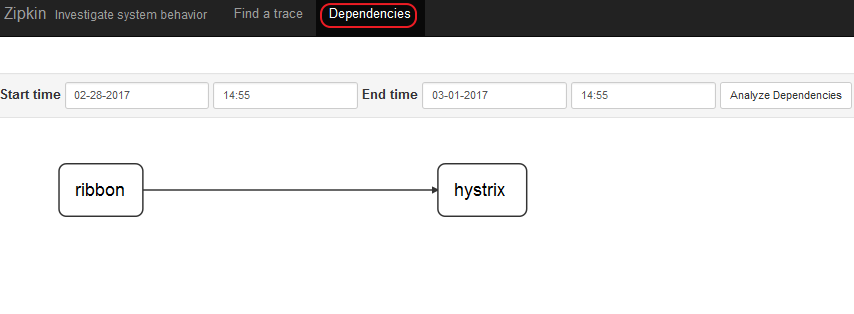
一个具体的trace信息

点击某一个span后会看到下面的信息



一个具体的span信息

点击页面顶部的dependencies按钮可以得到所有的服务依赖图,如下:

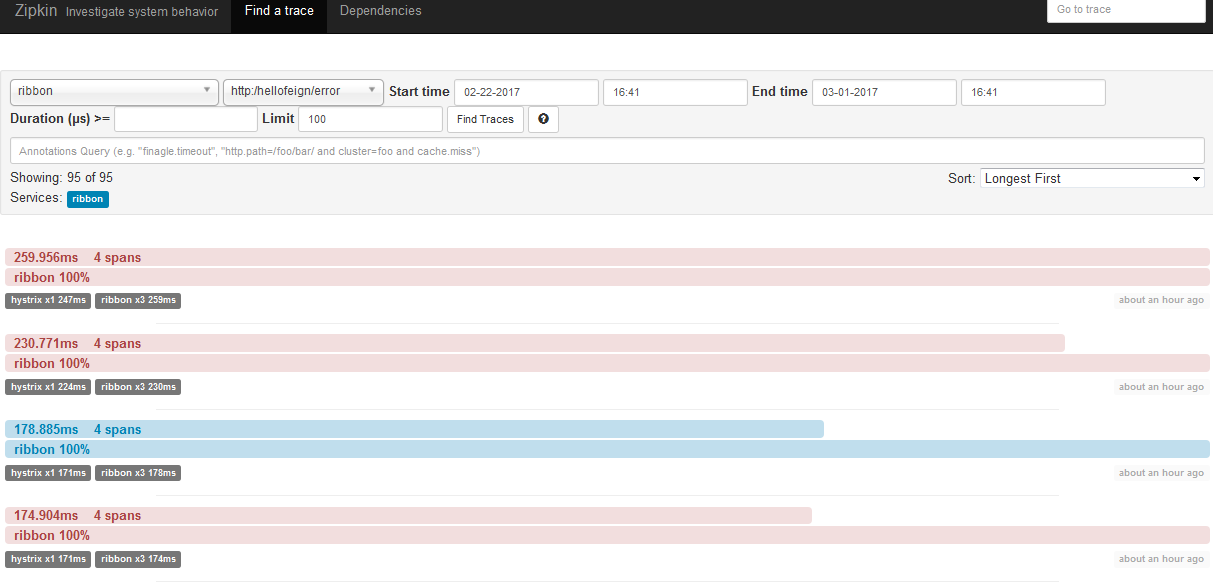


服务依赖关系图

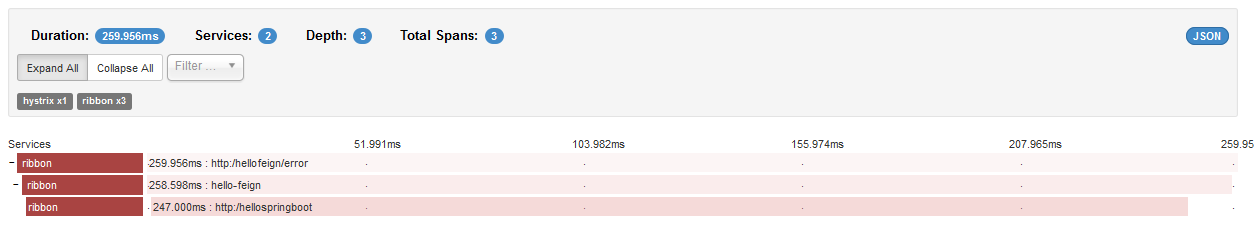
### 展示异常trace信息

在一些请求过程中可能会发生一些错误,在zipkin中对于异常访问会用红色进行标识.

下图中红色条目就是存在异常的访问记录:

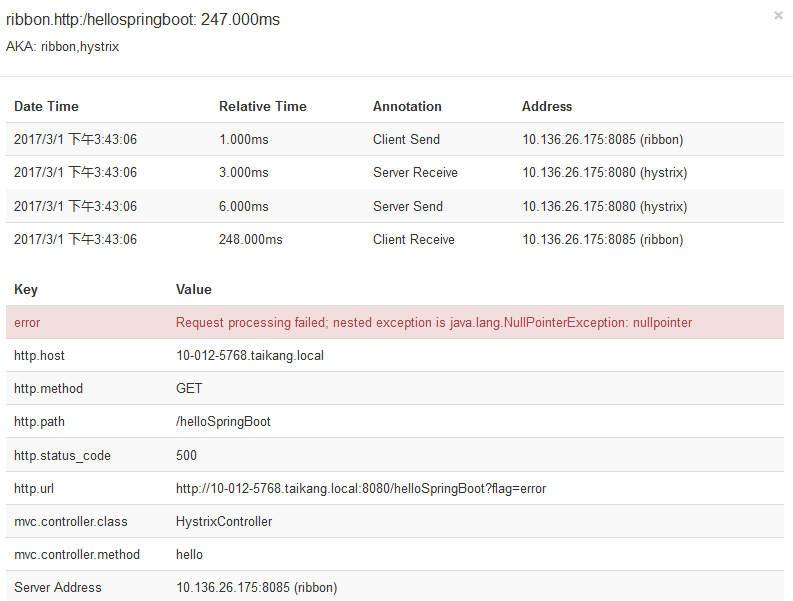


包含异常访问的trace记录



异常trace的详细信息

点击某一个span后会展示出错误详细信息



span详细信息

在上图中可以看到异常的服务器地址,类名,方法名和具体异常信息