# Spring Cloud微服务架构

目录

[Spring Cloud微服务架构 1](#_Toc511408260)

[一、 微服务架构概述 2](#_Toc511408261)

[二、 Spring Cloud项目搭建 3](#_Toc511408262)

[1. 服务注册（Eureka） 4](#_Toc511408263)

[2. 服务消费者（Feign和Ribbon） 8](#_Toc511408264)

[3. 断路器（Hystrix） 11](#_Toc511408265)

[4. 路由网关（Zuul） 13](#_Toc511408266)

[5. 分布式配置中心（Spring Cloud Config） 16](#_Toc511408267)

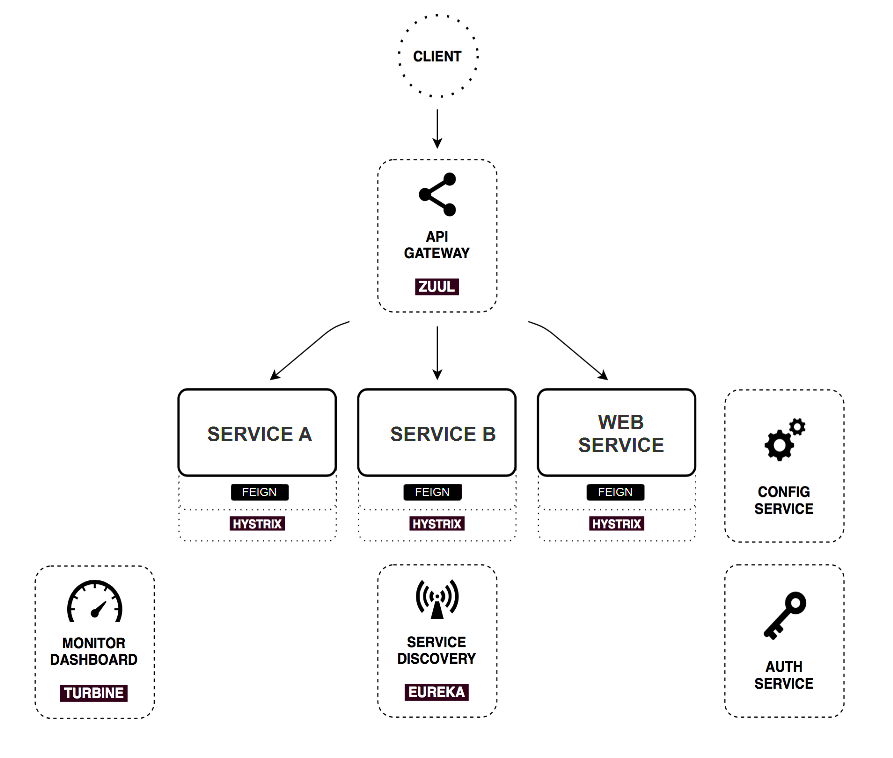
[6. 断路器监控（Hystrix Dashboard） 18](#_Toc511408268)

[7. 断路器聚合监控（Hystrix Turbine） 21](#_Toc511408269)

[8. 服务链路追踪（Spring Cloud Sleuth） 23](#_Toc511408270)

[9. 高可用服务注册中心 25](#_Toc511408271)

# 微服务架构概述



1. **API网关：**

API网关是外部请求访问系统的唯一入口，对外部请求进行过滤，通过路由将请求分发到恰当的后端服务或直接调用多个后端服务进行处理。此外，它可用于身份权限验证。

1. **Config模块：**

在分布式系统中，服务数量巨多，为了方便每个服务配置文件的统一管理，需要config模块来完成这项工作。Config模块可以将配置文件保存在本地或者git上进行管理，config模块作为server提供服务，其他模块作为client从config server中获取自己的配置文件。若要实现配置中心的高可用，需要将配置中心集群注册到eureka中。

1. **服务注册与发现模块：**

Spring Cloud中的核心部分就是服务的注册与发现，是通过Eureka来实现这个功能。Eureka分为Eureka Server和Eureka Client。

Eureka Server作为服务的注册中心，负责维护注册服务的列表，以REST API的形式为服务实例提供注册、管理和查询的操作。Eureka Server高可用集群就是多个实例两两之间相互注册，获取所有服务注册的信息，一旦一台server出现问题，Eureka Client会将请求自动切换到新的Eureka Server节点，当出现问题的server重新恢复时，又会被注册进来。

Eureka Client分为服务的提供方和服务的消费方，注册到Eureka Server中，注册的信息主要是服务名、机器ip、端口号和域名等。

1. **Hystrix和Hystrix Dashboard：**

在微服务架构中，系统被拆分成一个个服务单元，各单元应用间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会出现因等待出现故障的依赖方响应而形成任务积压，线程资源无法释放，最终导致自身服务的瘫痪，进一步甚至出现故障的蔓延最终导致整个系统的瘫痪。如果这样的架构存在如此严重的隐患，那么相较传统架构就更加的不稳定。为了解决这样的问题，因此产生了断路器等一系列的服务保护机制。

# Spring Cloud项目搭建

新建一个Maven Project，将目录下的其他内容删除，只保留一个pom.xml文件。



配置pom文件，将**<packaging>jar</packaging>**修改为**<packaging>pom</packaging>**，pom表示它是一个被继承的模块，添加本地maven库和子模块公用的依赖。

<repositories>

<repository>

<id>bizfocus</id>

<url>http://10.0.3.165:8081/repository/bizfocus/</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

</repositories>

<parent>

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

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

<version>1.5.3.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<dependencies>

<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>Dalston.SR1</version>

<type>pom</type>

<scope>import</scope>

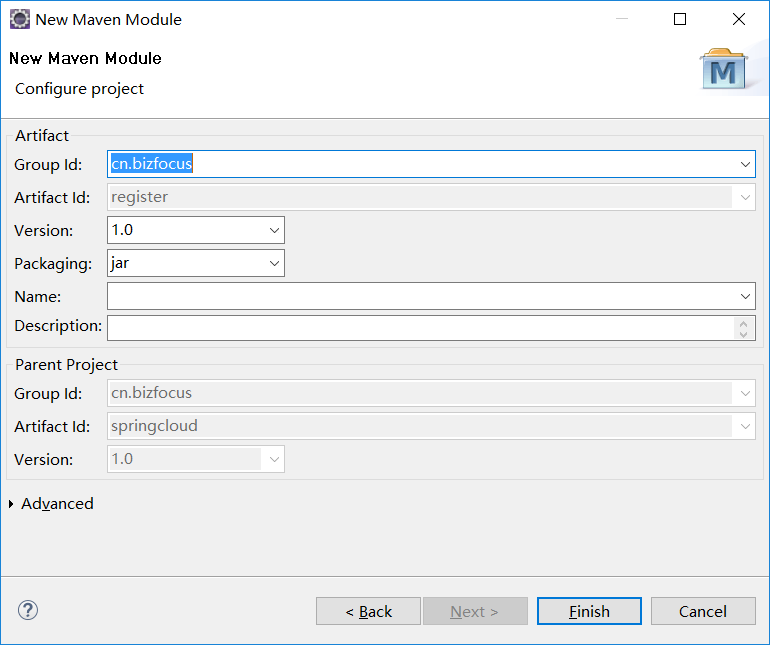
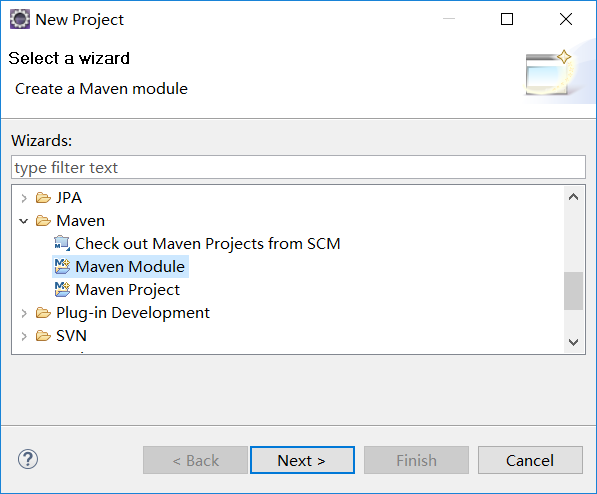
</dependency>

</dependencies>

</dependencyManagement>

## 服务注册（Eureka）

在springcloud项目下新建Maven Module



配置pom.xml文件，引入依赖

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

配置插件

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<jvmArguments>-Dfile.encoding=UTF-8</jvmArguments>

<fork>true</fork>

<!-- 如果没有该项配置，可能devtools不会起作用 -->

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<configuration>

<archive>

<index>true</index>

<manifest>

<addClasspath>true</addClasspath>

<mainClass>cn.bizfocus.feign.FeignStarter</mainClass>

<addDefaultImplementationEntries>true</addDefaultImplementationEntries>

<addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>

</manifest>

<manifestEntries>

<url>${project.url}</url>

<build-time>${maven.build.timestamp}</build-time>

</manifestEntries>

</archive>

</configuration>

</plugin>

</plugins>

</build>

新建启动类，通过@EnableEurekaServer开启。

@SpringBootApplication

@EnableEurekaServer

**public** **class** RegisterStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(RegisterStarter.**class**, args);

}

}

在resource目录下新建application.yml配置文件，添加配置

server:

port: 8761

eureka:

client:

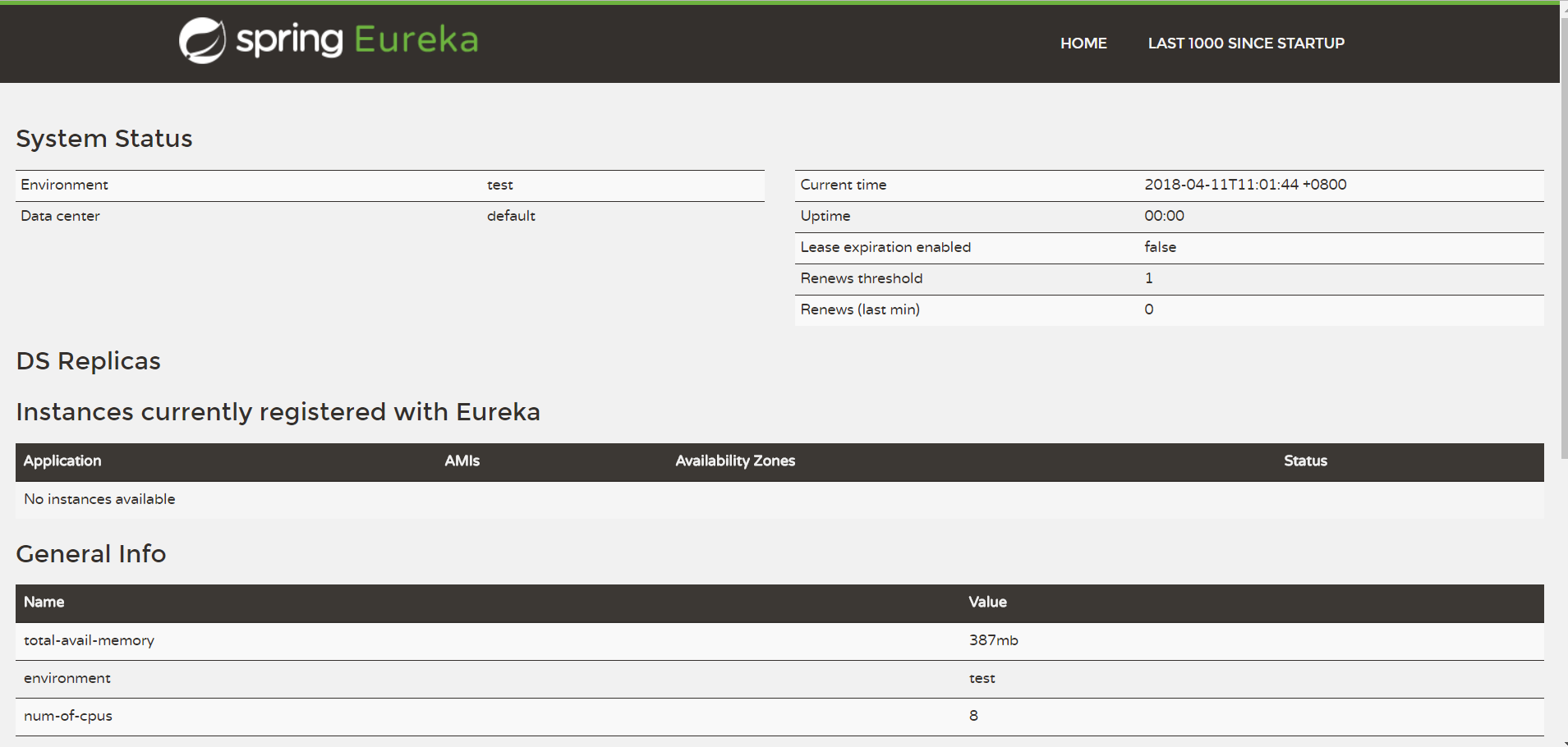
register-with-eureka: false

fetch-registry: false

serviceUrl:

defaultZone: [http://localhost:${server.port}/eureka/](http://localhost:$%7bserver.port%7d/eureka/)

启动工程，访问<http://127.0.0.1:8761>，进入eureka注册页面



新建两个服务模块service-ribbon和service-feign，添加依赖

<dependency>

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

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

</dependency>

启动类通过注解@EnableEurekaClient开启

@SpringBootApplication

@EnableEurekaClient

**public** **class** RibbonStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(RibbonStarter.**class**, args);

}

}

添加配置文件application.yml

spring:

application:

name: service-ribbon

server:

port: 8081

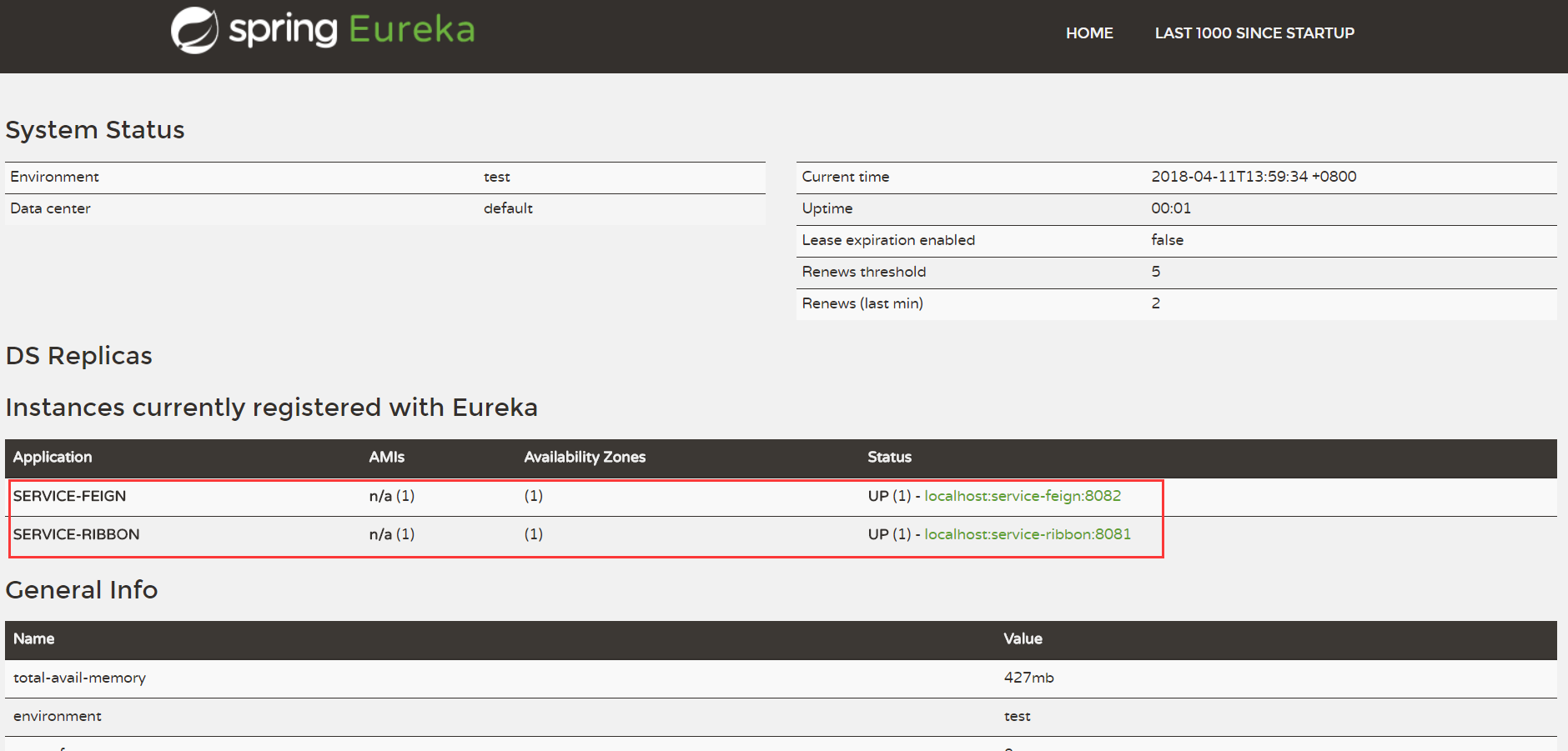
eureka:

client:

serviceUrl:

defaultZone: http://127.0.0.1:8761/eureka/

启动服务，service-ribbon和service-feign就注册到eureka上了



## 服务消费者（Feign和Ribbon）

服务间调用有两种方式Feign和Ribbon。

1. **Ribbon实现服务间调用：**

在pom.xml中引入依赖

<dependency>

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

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

</dependency>

创建RestTemplate实例，并通过@LoadBalanced注解开启均衡负载能力。

@SpringBootApplication

@EnableEurekaClient

**public** **class** RibbonStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(RibbonStarter.**class**, args);

}

@Bean

@LoadBalanced

RestTemplate restTemplate(){

**return** **new** RestTemplate();

}

}

写一个测试类RibbonService，通过之前注入ioc容器的restTemplate来消费service-feign服务的“/feign”接口，在这里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

@Service

**public** **class** RibbonService {

@Autowired

**private** RestTemplate restTemplate;

**public** String test(String name){

**return** restTemplate.getForObject("http://SERVICE-FEIGN/feign?name="+name,String.**class**);

}

}

在controller中调用RibbonService的test方法

@RestController

**public** **class** RibbonController {

@Value("${server.port}")

String port;

@RequestMapping("/ribbon")

**public** String home(@RequestParam String name) {

**return** "hi "+name+",i am from port:" +port;

}

@Autowired

**private** RibbonService ribbonService;

@RequestMapping("/test")

**public** String test(@RequestParam String name){

**return** ribbonService.test(name);

}

}

这样就实现了在service-ribbon服务中调用service-feign服务的接口，分别以端口8082和8092启动service-feign服务，访问<http://127.0.0.1:8081/test?name=ribbon>,会出现

hi ribbon,i am from port:8082

hi ribbon,i am from port:8092

说明调用成功，并且实现了负载均衡。

1. **Feign实现服务间调用。**

在pom.xml中引入依赖

<dependency>

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

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

</dependency>

在启动类中加入@EnableFeignClients开启Feign的功能

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

**public** **class** FeignStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(FeignStarter.**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-ribbon服务的“/ribbon”接口，代码如下

@FeignClient(name="Service-ribbon")

**public** **interface** FeignServiceClient {

@RequestMapping("/ribbon")

**public** String home(String name);

}

在controller中调用该接口中的方法。

@RestController

**public** **class** FeignController {

@Value("${server.port}")

String port;

@Autowired

**private** FeignServiceClient feignServiceClient;

@RequestMapping("/feign")

**public** String home(@RequestParam String name) {

**return** "hi "+name+",i am from port:" +port;

}

@RequestMapping("/test")

**public** String test(@RequestParam String name){

**return** feignServiceClient.home(name);

}

}

这样就通过Feign实现了在service-feign服务中调用service-ribbon服务的接口，分别以端口8081和8091启动service-ribbon服务，访问<http://127.0.0.1:8082/test?name=feign>,会出现

hi feign,i am from port:8081

hi feign,i am from port:8091

Feign默认集成了Ribbon，并和Eureka结合，默认实现了负载均衡的效果。

## 断路器（Hystrix）

1. **Ribbon中引入Hystrix：**

在pom.xml中添加依赖

<dependency>

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

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

</dependency>

在启动类中增加@EnableHystrix开启Hystrix

@SpringBootApplication

@EnableEurekaClient

@EnableHystrix

**public** **class** RibbonStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(RibbonStarter.**class**, args);

}

@Bean

@LoadBalanced

RestTemplate restTemplate(){

**return** **new** RestTemplate();

}

}

改造原来的服务消费方式，新增RibbonService类，在使用ribbon消费服务的函数上增加@HystrixCommand注解来指定回调方法

@Service

**public** **class** RibbonService {

@Autowired

**private** RestTemplate restTemplate;

@HystrixCommand(fallbackMethod = "testError")

**public** String test(String name){

**return** restTemplate.getForObject("http://SERVICE-FEIGN/feign?name="+name,String.**class**);

}

**public** String testError(String name){

**return** "error!!!";

}

}

关闭service-feign服务，访问测试接口，页面显示

error!!!

说明断路器生效。

Hystrix默认超时1000ms就会执行回退逻辑，可以在application.yml中配置

hystrix:

command:

default:

execution:

isolation:

thread:

timeoutInMilliseconds: 5000

设置超时时间为5000ms，也可以针对每一个方法进行配置，具体配置如下

@HystrixCommand(fallbackMethod = "testError", commandProperties = {@HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="5000")})

**public** String test(String name){

**return** restTemplate.getForObject("http://SERVICE-FEIGN/feign?name="+name,String.**class**);

}

在@HystrixCommand中加入@HystrixProperty，name为具体要配置的参数，value为配置的值，这样就可以实现对每个方法执行不同的设置，其他参数的配置在<https://github.com/Netflix/Hystrix/wiki/Configuration#execution.timeout.enabled> 中都有介绍。

1. **Feign中引入Hystrix。**

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**ribbon**接口的注解中加上fallback的指定类就行了：

@FeignClient(name="Service-ribbon", fallback= FeignServiceHystrix.**class**)

**public** **interface** FeignServiceClient {

@RequestMapping("/ribbon")

**public** String home(@RequestParam("name") String name);

}

FeignServiceHystrix需要实现FeignServiceClient接口，并注入到Ioc容器中，代码如下：

@Component

**public** **class** FeignServiceHystrix **implements** FeignServiceClient{

@Override

**public** String home(String name) {

**return** "error!!!";

}

}

Feign配置Hystrix参数不能像Ribbon一样配置在@HystrixCommand中，只能配置在配置文件里，针对单个方法进行配置时，需要指定具体的HystrixCommandKey，具体配置如下：

hystrix:

command:

"FeignServiceClient#home(String)":

execution:

isolation:

thread:

timeoutInMilliseconds: 2000

default:

execution:

isolation:

thread:

timeoutInMilliseconds: 5000

其中" FeignServiceClient#home(String)"就是HystrixCommandKey，这样就可以实现home这个方法配置的超时时间是2000ms，其他方法超时时间是5000ms。

## 路由网关（Zuul）

1. **路由转发**

新建网关模块项目gateway，在pom.xml中引入依赖

<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>

在启动类中增加注解@EnableZuulProxy开启Zuul

@SpringBootApplication

@EnableEurekaClient

@EnableZuulProxy

**public** **class** GatewayStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(GatewayStarter.**class**, args);

}

}

在resource目录增加application.yml配置文件

server:

port: 2222

spring:

application:

name: gateway

eureka:

client:

serviceUrl:

defaultZone: http://127.0.0.1:8761/eureka/

zuul:

ignoredServices: '\*'

host:

#路由转发至url，超时时间取这两个配置参数

socket-timeout-millis: 10000

connect-timeout-millis: 10000

routes:

service-ribbon:

path: /ribbon/\*\*

serviceId: service-ribbon

# url: http://127.0.0.1:8081/

service-feign:

path: /feign/\*\*

serviceId: service-feign

# url: http://127.0.0.1:8082/

max:

host:

connections: 5000

#路由转发至serviceId,超时时间用的以下配置

ribbon:

ReadTimeout: 10000

# SocketTimeout: 10000

ConnectTimeout: 3000

该配置中，**ignoredServices: '\*'**为忽略默认的转发规则，这样可以是路由转发只执行我们自定义的转发规则；将符合**/ribbon/\*\***格式的请求都转发到**service-ribbon**服务中；路由转发也可以配置转发指定的url。

1. **服务过滤**

在完成了服务路由之后，我们对外开放服务还需要一些安全措施来保护客户端只能访问它应该访问到的资源。所以我们需要利用Zuul的过滤器来实现我们对外服务的安全控制。

在服务网关中定义过滤器只需要继承**ZuulFilter**抽象类实现其定义的四个抽象函数就可对请求进行拦截与过滤。

比如下面的例子，定义了一个Zuul过滤器，实现了在请求被路由之前检查请求中是否有accessToken参数，若有就进行路由，若没有就拒绝访问，返回**401 Unauthorized**错误。

**public** **class** AccessFilter **extends** ZuulFilter{

**private** **final** Logger logger = Logger.*getLogger*(getClass());

@Override

**public** Object run() {

RequestContext ctx = RequestContext.*getCurrentContext*();

HttpServletRequest request = ctx.getRequest();

logger.info(String.*format*("%s request to %s", request.getMethod(), request.getRequestURL().toString()));

Object accessToken = request.getParameter("accessToken");

**if**(accessToken == **null**) {

logger.warn("access token is empty");

ctx.setSendZuulResponse(**false**);

ctx.setResponseStatusCode(401);

**return** **null**;

}

logger.info("access token ok");

**return** **null**;

}

@Override

**public** **boolean** shouldFilter() {

**return** **true**;

}

@Override

**public** **int** filterOrder() {

**return** 0;

}

@Override

**public** String filterType() {

**return** "pre";

}

}

自定义过滤器的实现，需要继承**ZuulFilter**，需要重写实现下面四个方法：

* **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**：可以在请求被路由之前调用
  + **routing**：在路由请求时候被调用
  + **post**：在routing和error过滤器之后被调用
  + **error**：处理请求时发生错误时被调用
* **filterOrder**：通过int值来定义过滤器的执行顺序
* **shouldFilter**：返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关。在上例中，我们直接返回true，所以该过滤器总是生效。
* **run**：过滤器的具体逻辑。需要注意，这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回的错误码，当然我们也可以进一步优化我们的返回，比如，通过ctx.setResponseBody(body)对返回body内容进行编辑等。

在实现了自定义过滤器之后，还需要实例化该过滤器才能生效，我们只需要在应用主类中增加如下内容：

@SpringBootApplication

@EnableEurekaClient

@EnableZuulProxy

**public** **class** GatewayStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(GatewayStarter.**class**, args);

}

@Bean

**public** AccessFilter accessFilter() {

**return** **new** AccessFilter();

}

}

## 分布式配置中心（Spring Cloud Config）

新建config模块作为Config Server，在pom.xml文件中引入依赖

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

在启动类中通过注解@EnableConfigServer开启ConfigServer

@SpringBootApplication

@EnableConfigServer

**public** **class** ConfigStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ConfigStarter.**class**, args);

}

}

新建application.yml配置文件，增加配置服务信息。Config提供从git和本地两种方式读取配置文件：

1. **从本地读取配置文件**

spring:

cloud:

config:

server:

native:

search-locations: classpath:/dev

profiles:

active: native

application:

name: config-server

server:

port: 7001

profiles.active=native使用本地进行存储配置；search-locations: classpath:/dev默认从dev目录下读取配置文件

1. **从git读取配置**

# git管理配置

spring:

cloud:

config:

server:

git:

uri: https://github.com/SpringCloud.git

searchPaths: config-repo

username: username

password: password

**uri**：配置git仓库位置

**searchPaths**：配置仓库路径下的相对搜索位置，可以配置多个

**username**：访问git仓库的用户名

**password**：访问git仓库的用户密码

URL与配置文件的映射关系如下：

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

/{application}-{profile}.yml

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

/{application}-{profile}.properties

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

上面的url会映射{application}-{profile}.properties对应的配置文件，{label}对应git上不同的分支，默认为master。

1. **在register服务中获取config中存的配置文件：**

在register项目的pom.xml中引入依赖

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

在resource目录增加bootstrap.yml配置文件，配置如下：

spring:

cloud:

config:

uri: http://localhost:7001

fail-fast: true

application:

name: register

* **spring.application.name**：对应前配置文件中的{application}部分
* **spring.cloud.config.profile**：对应前配置文件中的{profile}部分
* **spring.cloud.config.label**：对应前配置文件的git分支
* **spring.cloud.config.uri**：配置中心的地址

**这里需要格外注意：上面这些属性必须配置在bootstrap.yml中，config部分内容才能被正确加载。因为config的相关配置会先于application. yml，而bootstrap. yml的加载也是先于application. yml。**

## 断路器监控（Hystrix Dashboard）

Hystrix Dashboard实现了Hystrix指标数据的可视化面板，在service-ribbon项目的pom.xml文件中引入依赖

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

在启动类中通过@EnableHystrixDashboard开启Hystrix Dashboard功能

@SpringBootApplication

@EnableEurekaClient

@EnableHystrix

@EnableHystrixDashboard

**public** **class** RibbonStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(RibbonStarter.**class**, args);

}

@Bean

@LoadBalanced

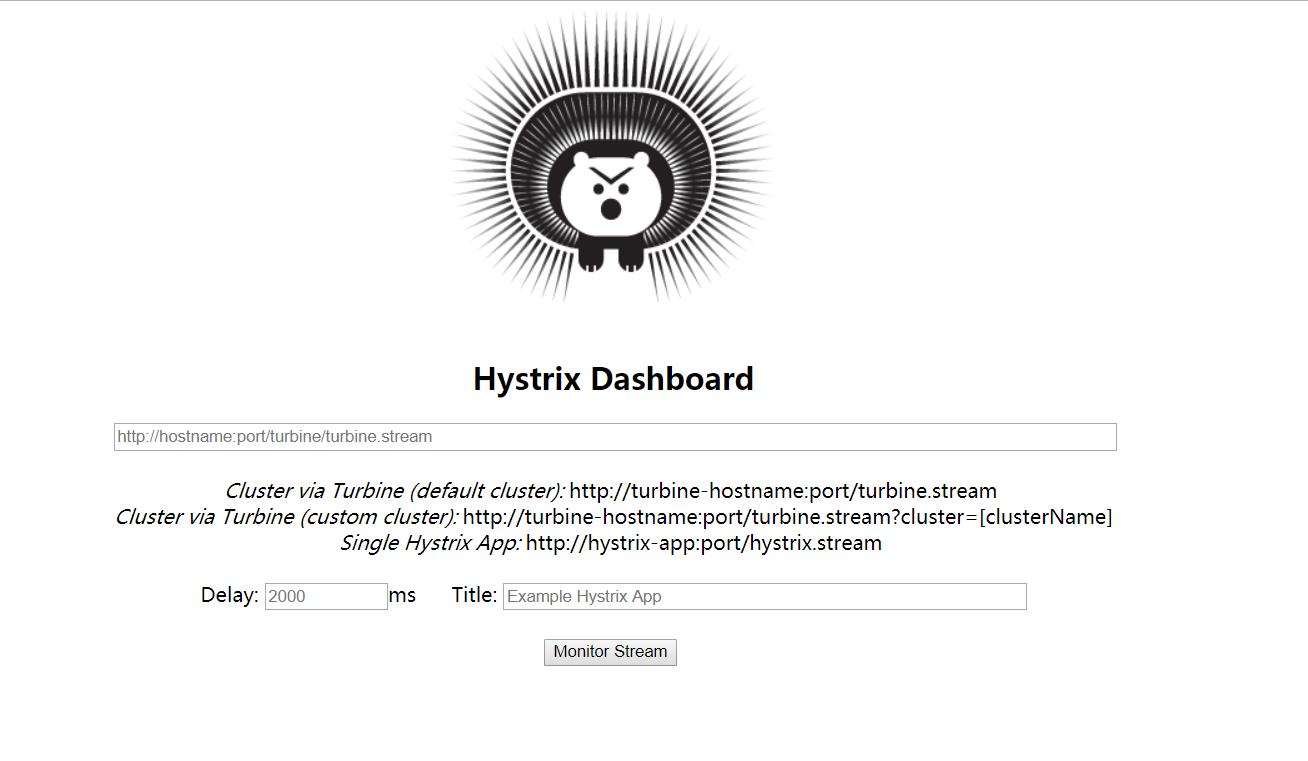
RestTemplate restTemplate(){

**return** **new** RestTemplate();

}

}

访问<http://localhost:8081/hystrix>，可以看到以下页面

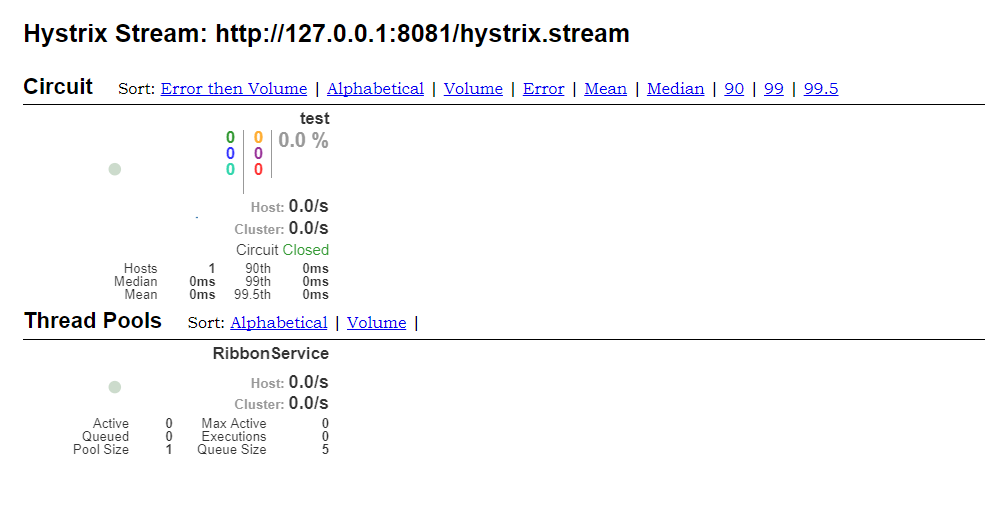


这是Hystrix Dashboard的监控首页，该页面中并没有具体的监控信息。从页面的文字内容中我们可以知道，Hystrix Dashboard共支持三种不同的监控方式，依次为：

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

前两者都对集群的监控，需要整合Turbine才能实现。

在首页中输入<http://localhost:8081/hystrix.stream>，点击Monitor Stream，可以看到如下页面，实现了对**test**接口的监控



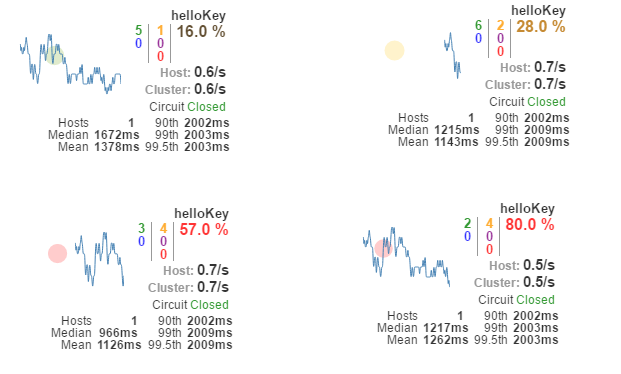
在对该页面介绍前，我们先看看在首页中我们还没有介绍的两外两个参数：

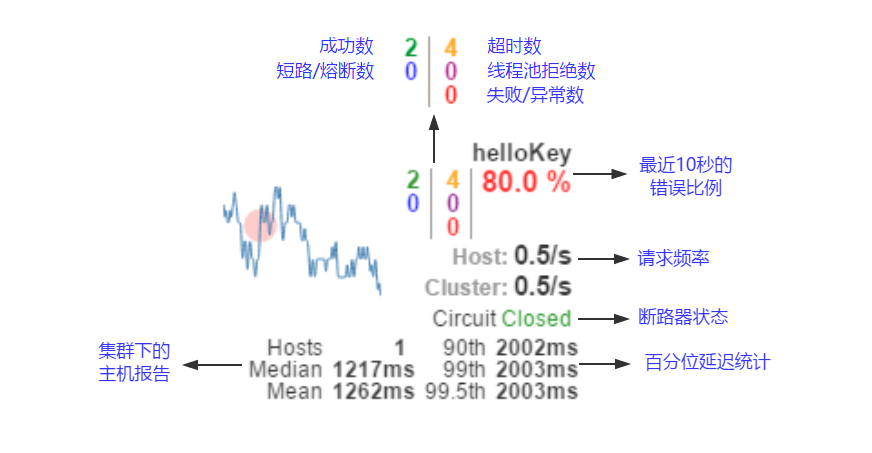
* **Delay**：该参数用来控制服务器上轮询监控信息的延迟时间，默认为2000毫秒，我们可以通过配置该属性来降低客户端的网络和CPU消耗。
* **Title**：该参数对应了上图头部标题Hystrix Stream之后的内容，默认会使用具体监控实例的URL，我们可以通过配置该信息来展示更合适的标题。

回到监控页面，我们来详细说说其中各元素的具体含义：

我们可以在监控信息的左上部分找到两个重要的图形信息：一个实心圆和一条曲线。

* 实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，如下图所示，它的健康度从绿色、黄色、橙色、红色递减。该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，我们就可以在大量的实例中快速的发现故障实例和高压力实例。
* 曲线：用来记录2分钟内流量的相对变化，我们可以通过它来观察到流量的上升和下降趋势。

[](http://blog.didispace.com/content/images/posts/spring-cloud-starter-dalston-5-1-3.png)

* 其他一些数量指标如下图所示：  
  [](http://blog.didispace.com/content/images/posts/spring-cloud-starter-dalston-5-1-4.png)

## 断路器聚合监控（Hystrix Turbine）

新建一个monitor模块，在pom.xml中引入依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-turbine</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
 </dependency>

在启动类增加注解@EnableTurbine开启Turbine

@SpringBootApplication

@EnableEurekaClient

@EnableTurbine

**public** **class** MonitorStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(MonitorStarter.**class**, args);

}

}

在application.yml中加入以下配置：

server:

port: 7002

spring:

application:

name: monitor

eureka:

client:

service-url:

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

turbine:

app-config: service-ribbon,service-feign

cluster-name-expression: new String("default")

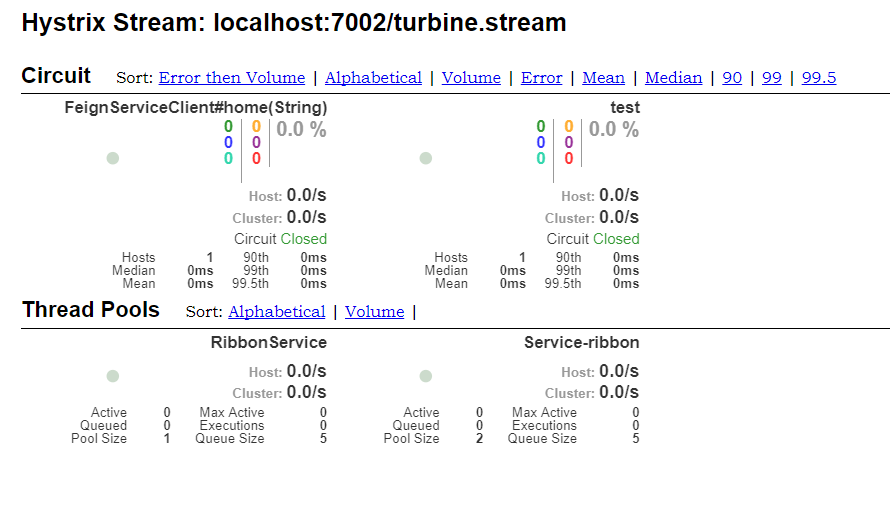
combine-host-port: true

参数说明

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

访问<http://localhotst:8081/hystrix>，在输入框中输入<http://localhost:7002/turbine.stream>，

点击Monitor Stream，可以看到以下页面，实现了对service-ribbon和service-feign服务的监控



## 服务链路追踪（Spring Cloud Sleuth）

新建zipkin服务模块，在pom.xml中引入依赖

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-server</artifactId>

</dependency>

<dependency>

<groupId>io.zipkin.java</groupId>

<artifactId>zipkin-autoconfigure-ui</artifactId>

</dependency>

在启动类中增加注解@EnableZipkinServer开启ZipkinServer功能

@SpringBootApplication

@EnableZipkinServer

**public** **class** ZipkinStarter {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ZipkinStarter.**class**, args);

}

}

增加配置文件application.yml，在配置文件中配置

server:

port: 7003

在service-ribbon和service-feign模块的pom.xml中引入依赖

<dependency>

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

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

</dependency>

在application.yml中配置ZipkinServer的地址

spring:

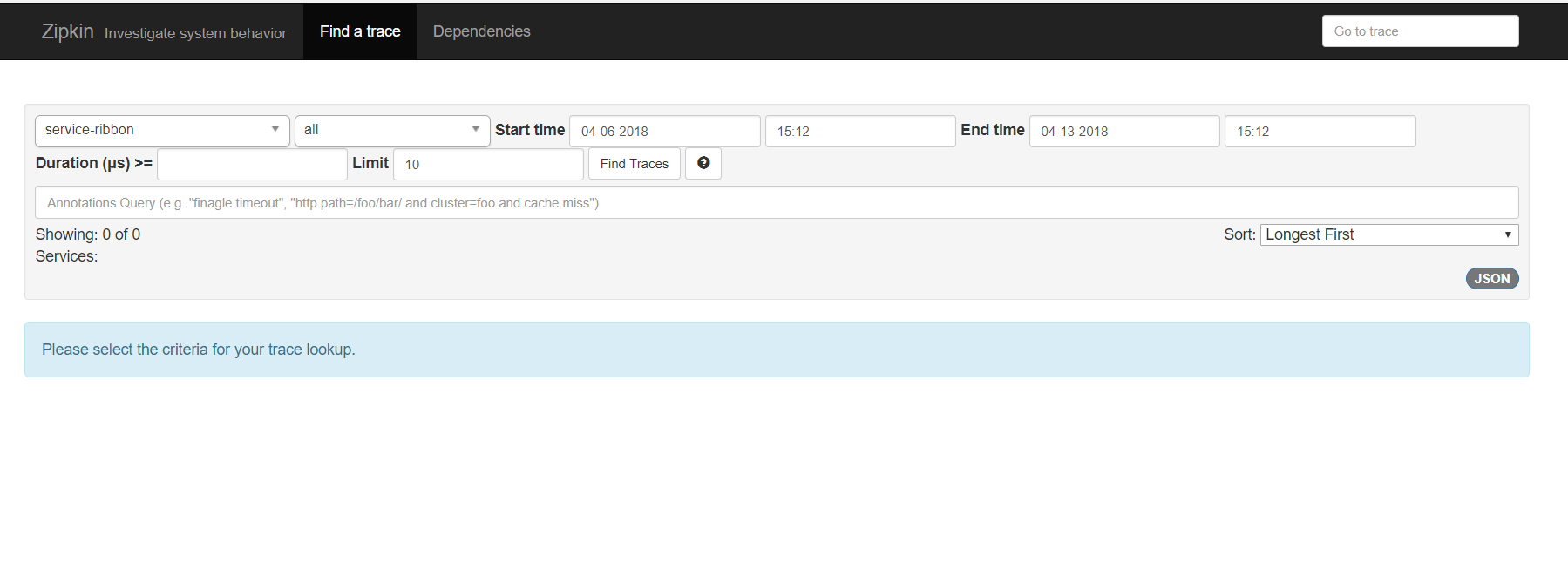
application:

name: service-ribbon

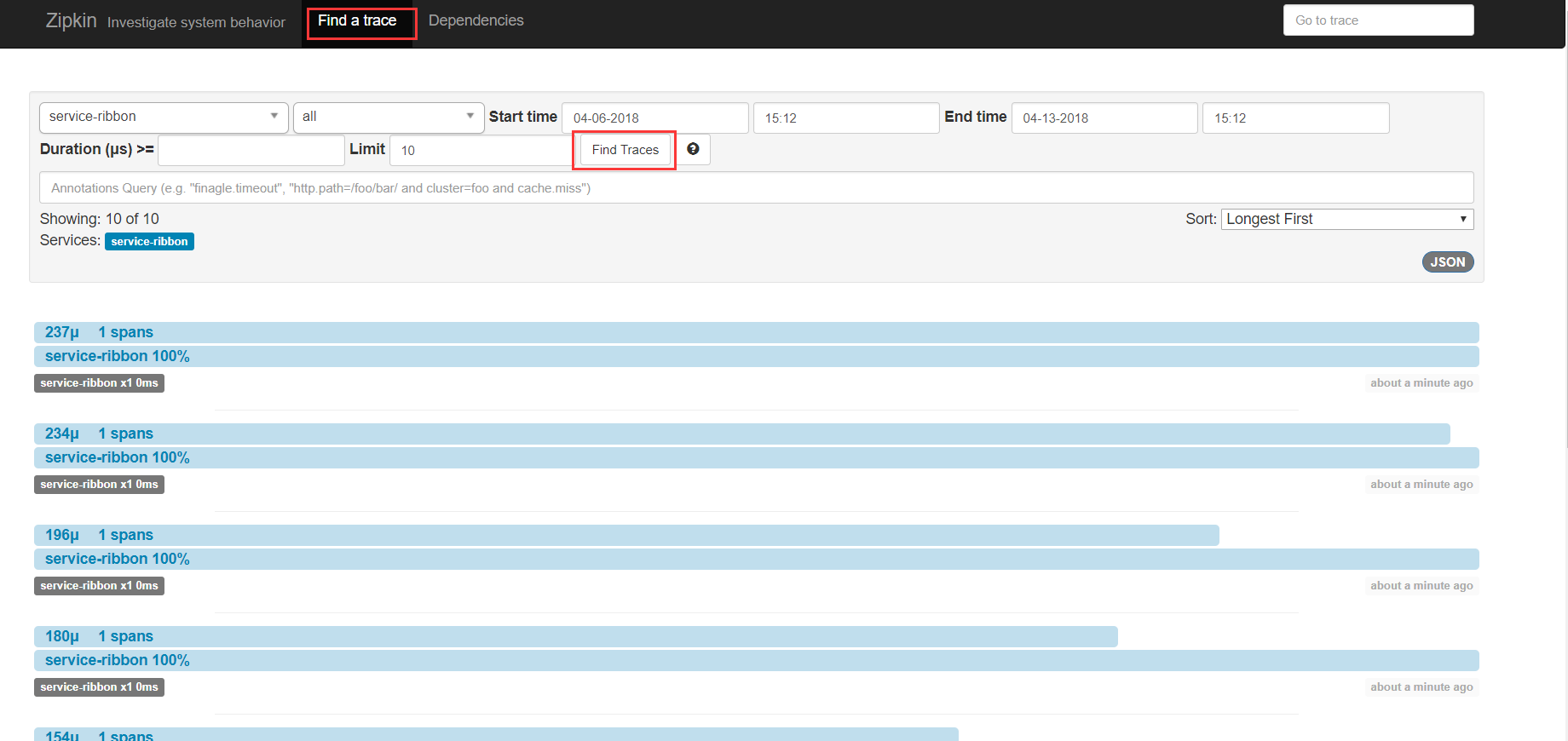
zipkin:

base-url: <http://127.0.0.1:7003>

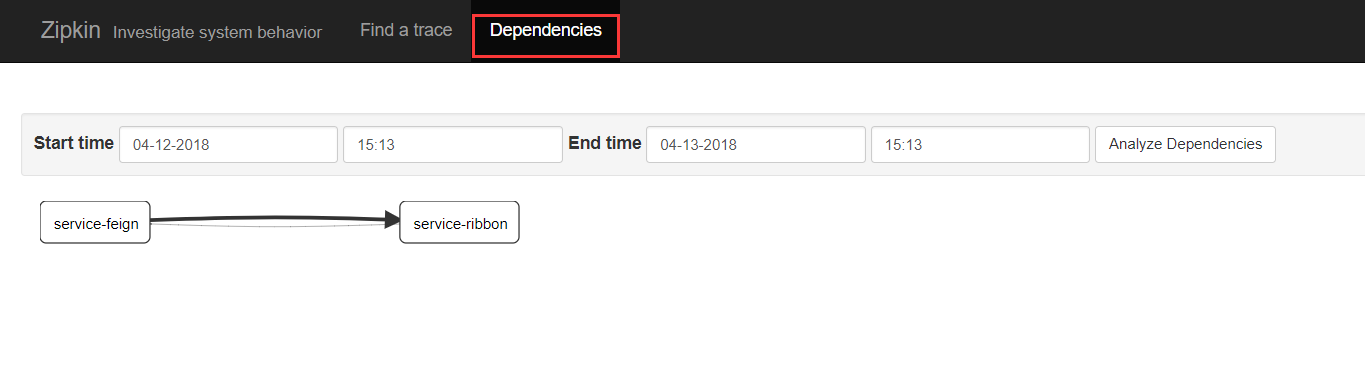
启动工程，访问<http://localhost:7003>，进入Zipkin页面



点击find traces,可以看到具体服务相互调用的数据



点击Dependencies,可以发现服务的依赖关系：



## 高可用服务注册中心

Eureka Server除了单点运行之外，还可以通过运行多个实例，并进行互相注册的方式来实现高可用的部署，所以我们只需要将Eureke Server配置其他可用的serviceUrl就能实现高可用部署。

改造register项目，在resource目录新建application-register1.yml和application-register2.yml文件，分别配置

**application-register1.yml**

server:

port: 8761

spring:

application:

name: register1

eureka:

client:

service-url:

defaultZone: http://register2:8762/eureka/

**application-register2.yml**

server:

port: 8762

spring:

application:

name: regoster2

eureka:

client:

service-url:

defaultZone: http://register1:8761/eureka/

修改/etc/hosts文件，windows电脑，在c:/windows/systems/drivers/etc/hosts 修改，添加以下两行：

127.0.0.1 peer1

127.0.0.1 peer2

分别以application-register1.yml和application-register2.yml启动项目

java -jar -Xmx200m register-1.0.jar --spring.profiles.active=register1

java -jar -Xmx200m register-1.0.jar --spring.profiles.active=register2

修改service-ribbon的application.yml文件

eureka:

client:

serviceUrl:

# defaultZone: http://127.0.0.1:8761/eureka/

defaultZone: http://register1:8761/eureka/, http://register2:8762/eureka/

启动service-ribbon服务，访问<http://127.0.0.1:8761>和<http://127.0.0.1:8762>,看到以下页面，实现了高可用注册中心。

