springCloud

工具：Idea，JDK1.8,Maven3.5,mysql

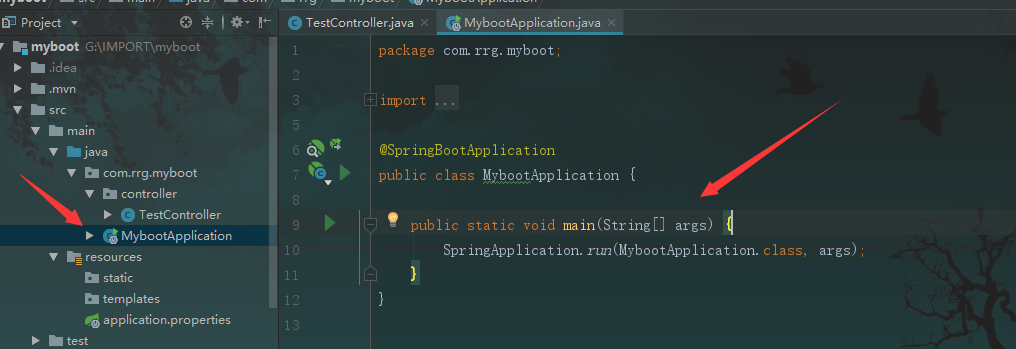
1. 创建第一个springboot工程

1）.打开Idea-> new Project ->Spring Initializr ->填写group、artifact ->钩上web(开启web功能）->点下一步就行了。

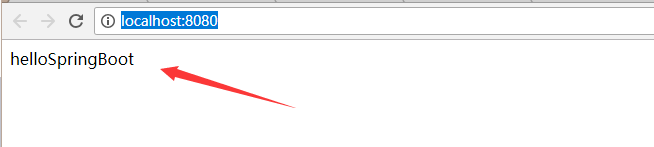
2）.在controller写下如下代码：

@RestController  
public class TestController {  
 **@RequestMapping("/")  
 public String helloSpringBoot(){**  
 return "helloSpringBoot";  
 }  
}

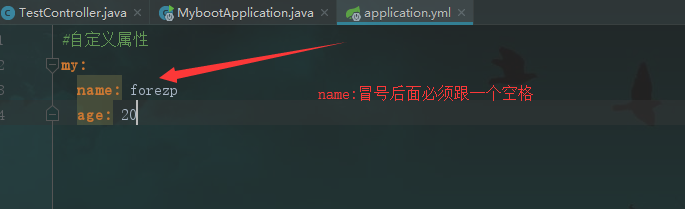
1. .打开MyBootApplication，并运行main方法，即是运行了项目。



1. .打开浏览器，输入：localhost:8080，回车

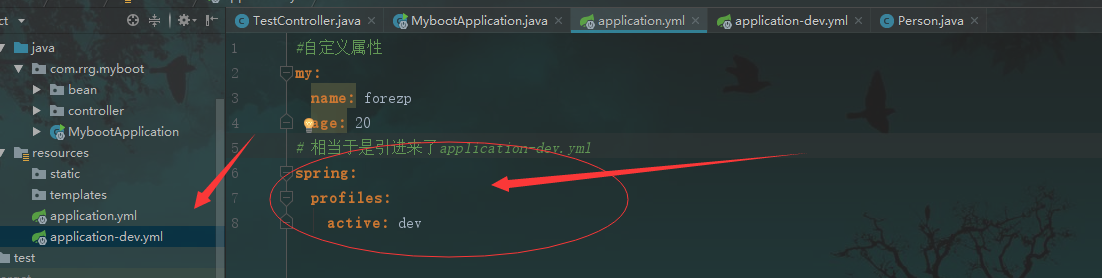


1. .ok，至此springboot已成功建立。
2. springboot的配置文件：properties.yml
3. [.自定义属性，我们可用@Value(“${my.name}”)获取](mailto:.自定义属性，我们可用@Value(\“${my.name}\”)获取)



1. .配置多个propertes-file.yml

在properties.yml里写上：



1. Springboot使用mybatis访问mysql
2. .在pom中加入依赖包：

<!--整合mybatis-->  
<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.2</version>  
</dependency>

引入msql连接类和连接池

<dependency>

<groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.29</version>

</dependency>

<!--开启web-->

<dependency>

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

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

</dependency>

1. .配置连接数据库

#jdbcTemplates连接mysql数据库  
spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/shoocl?useUnicode=true&characterEncoding=utf-8  
spring.datasource.username=root  
spring.datasource.password=root

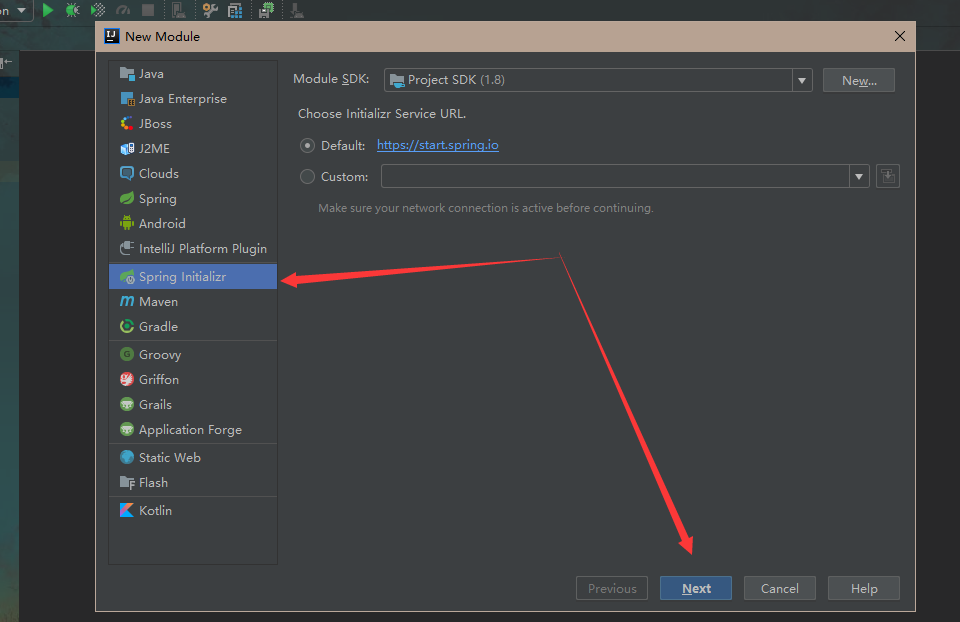
1. .开启事务，在service层的方法上加上注解即可：@Transactional
2. spring cloud为开发人员提供了一套快速构建分布式系统的一些工具，包括：配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会等等。
3. .创建服务注册中心

需要用到spring cloud netfixd的Eureka，Eureka是一个服务注册和发现模块。

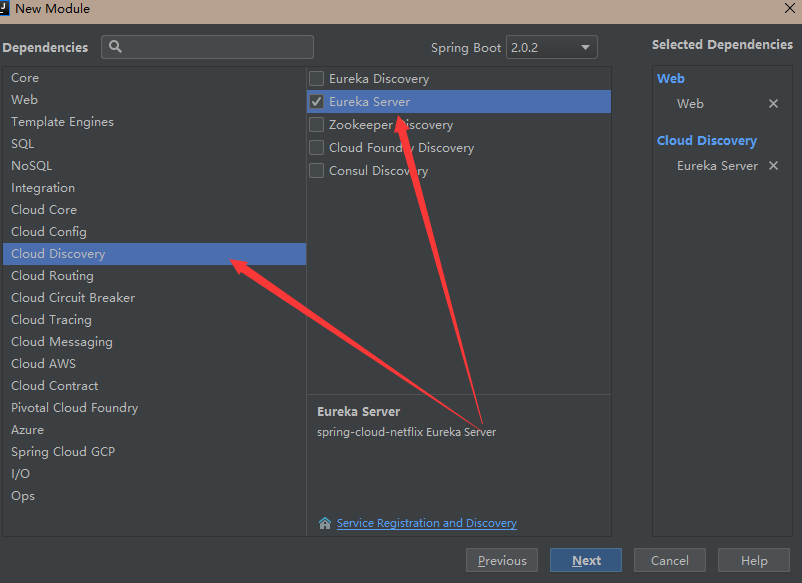
在springboot的基础上，创建两个model工程，一个是服务注册中心，即Eureka server，另一个是Eureka Client。

1. .开始创建Eureka Server

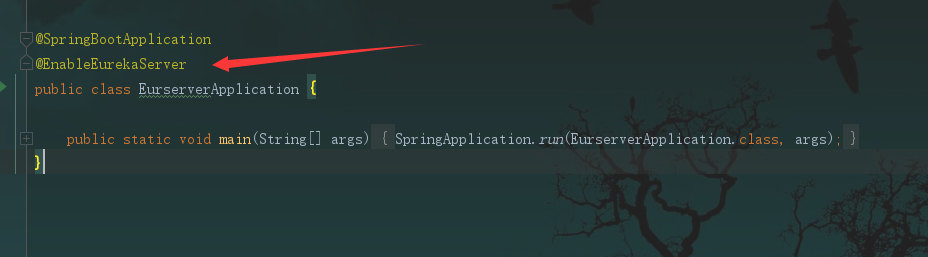
右键工程，new ==> model，然后选择Spring Initializr，next。



然后填写model名字，next，选择Cloud Discovery，Eureka Server，next ==>finish



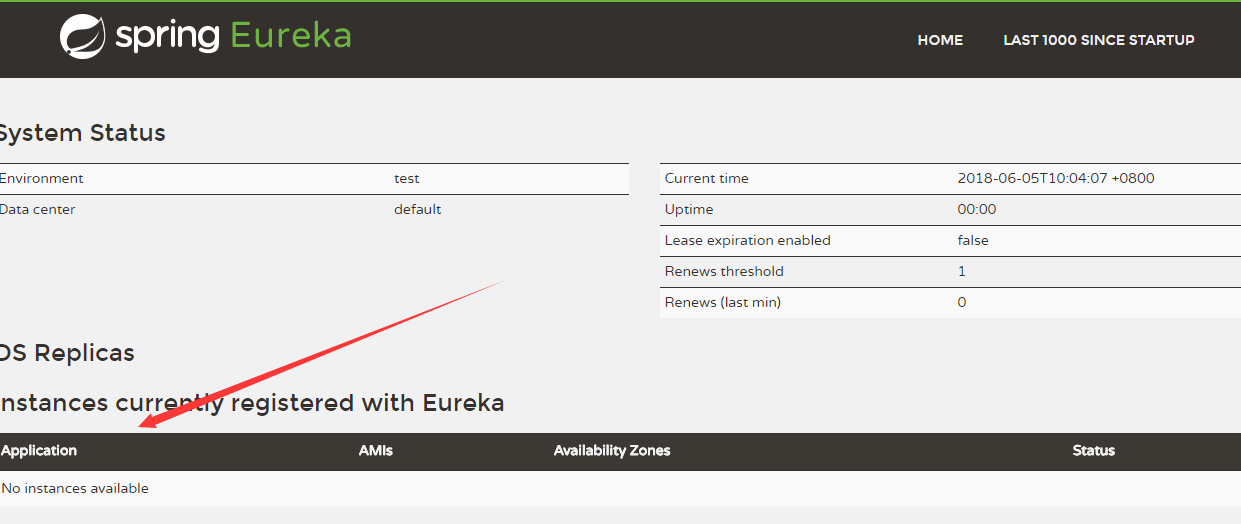
1. .启动一个服务注册中心，主需要一个注解@EnableServer，这个注解放在springboot的启动工程里。



1. .配置文件

*#修改端口号***server:  
 port:** 8088  
*#通过eureka.client.registerWithEureka：false和fetchRegistry：false来表明自己是一个eureka server.***eureka:  
 instance:  
 hostname:** localhost  
 **client:  
 register-with-eureka: false  
 fetch-registry: false  
 service-url:  
 defaultZone:** http://${**eureka.instance.hostname**}:${**server.port**}/eureka/

1. .配置完成后，打开浏览器，输入localhost:8088



因为还没有注册服务，所以No application available 没有服务被发现

1. .创建一个服务提供者（Eureka Client），创建过程同创建Eureka Server类似。
2. .配置文件和启动

@SpringBootApplication  
@EnableEurekaClient  
@RestController  
public class EuclientApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(EuclientApplication.class, args);  
 }  
  
 @Value("${server.port}")  
 String port;  
 @RequestMapping("/hi")  
 public String hello(@RequestParam String name){  
 return "hi " + name + ",I am a port:" + port;  
 }  
}

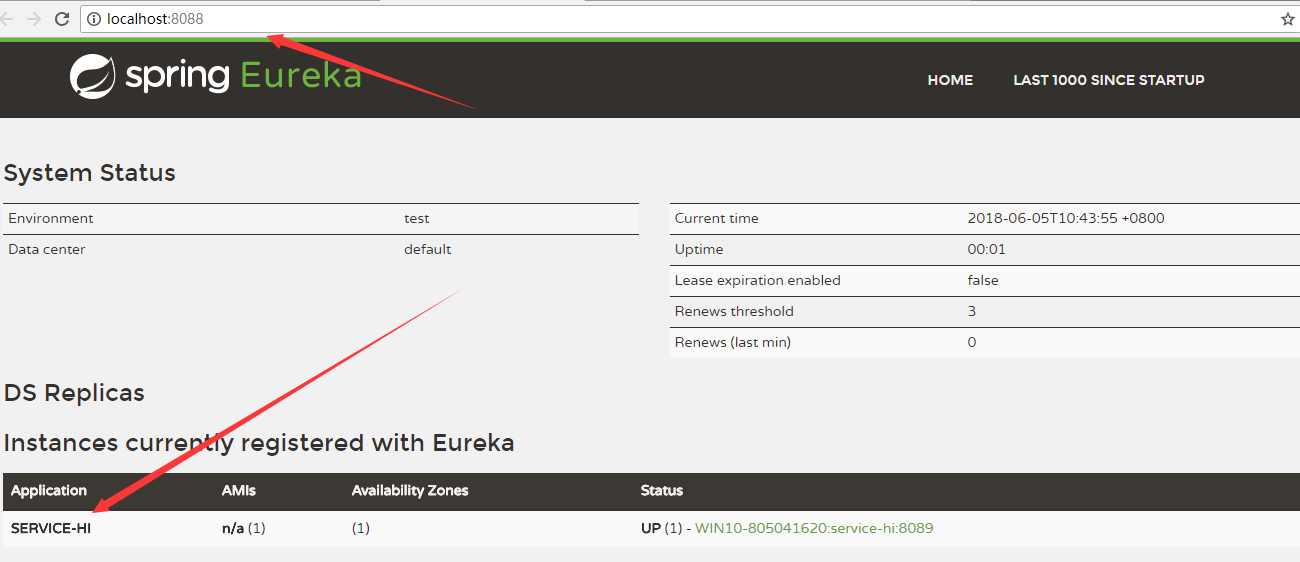
@EnableEurekaClient是Eureka Client的启动注解

@Value("${server.port}")是获取配置文件信息

配置文件：

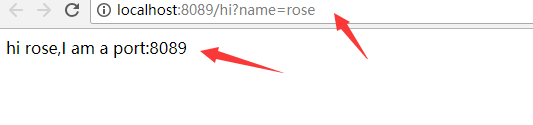
*#注明自己的服务注册中心的地址***eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8088/eureka/  
*#端口号***server:  
 port:** 8089  
*#需要指明spring.application.name,这个很重要，这在以后的服务与服务之间相互调用一般都是根据这个name***spring:  
 application:  
 name:** service-hi

1. .启动，要先启动Eureka Server，再启动Eureka Client
2. .访问local host:8088



这时候我们发现，已经有了一个应用。就是我们刚刚启动的Eureka Client。

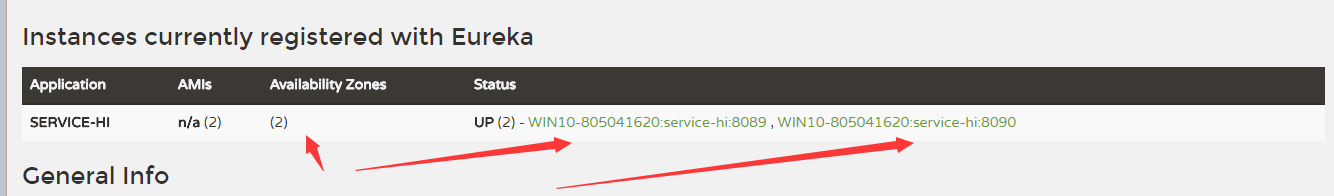
我们访问：localhost:8089/hi?name=rose



至此，我们已经成功的搭建了Eureka Server服务注册中心和服务发现。

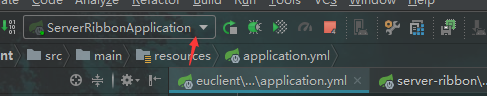
1. Spring cloud有两种服务调用方式，一种是ribbon+restTemplate，另一种是feign。
2. .ribbon是一个负载均衡客户端，可以很好的控制http和tcp的一些行为。Feign默认集成了ribbon。
3. .前期准备工作：启动euserver，再启动client，然后把euclient,再把euclient配置文件里的port改为8090，重启:

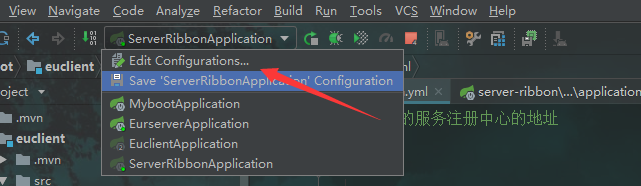
这时候访问euserver:locahost:8088

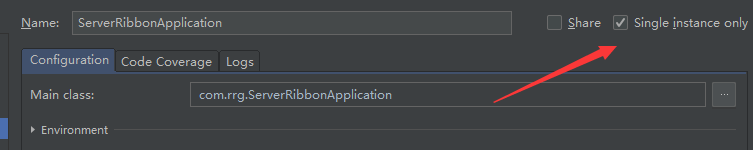


会发现，eureka-server注册了2个实例，这就相当于一个小的集群。

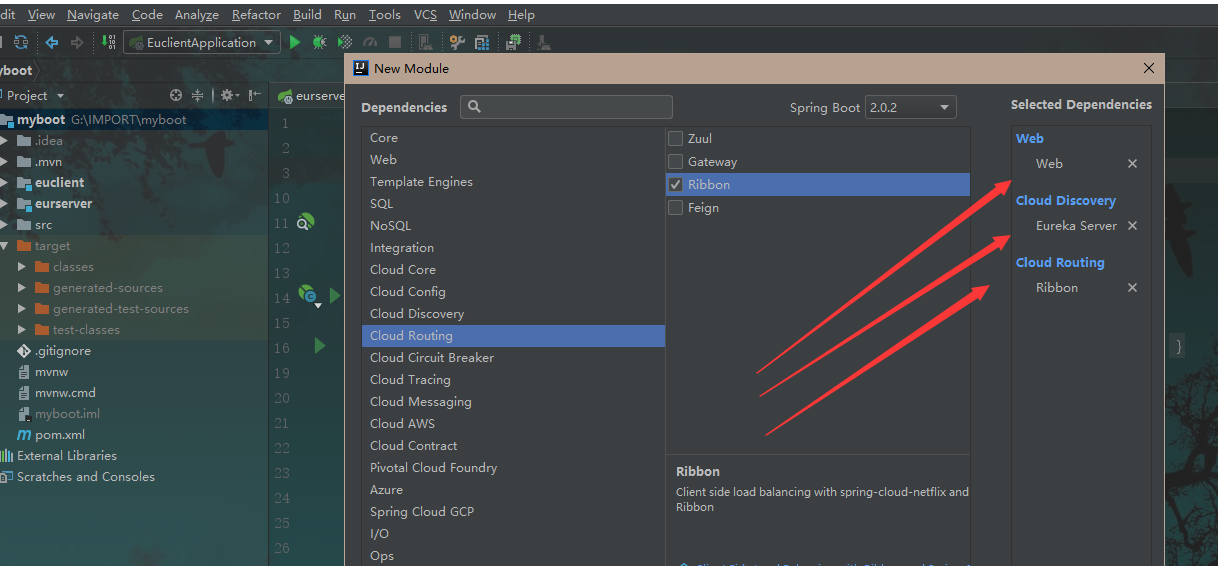
**这里，要设置一个工程启动多个实例：去掉勾。**







1. .建立一个消费服务者：建立一个新的boot工程，命名为server-ribbon



配置文件：

*#指定服务的注册中心地址***eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8088/eureka/  
**server:  
 port:** 8091  
**spring:  
 application:  
 name:** service-ribbon

启动类：

@SpringBootApplication  
// 通过@EnableDiscoveryClient向服务中心注册  
@EnableDiscoveryClient  
public class ServerRibbonApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ServerRibbonApplication.class, args);  
 }  
  
 //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。  
 @Bean  
 @LoadBalanced  
 RestTemplate restTemplate(){  
 return new RestTemplate();

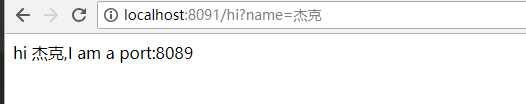
写一个测试类:

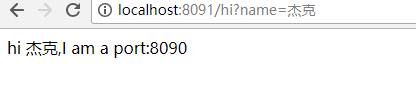
public class TestService {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 public String hiService(String name){  
 return restTemplate.getForObject("http://service-hi/hi?name=" + name,String.class);  
 }  
  
}

@RestController  
public class TestController {  
 @Autowired  
 private TestService testService;  
  
 @RequestMapping("/hi")  
 public String hi(@RequestParam String name){  
 return testService.hiService(name);  
 }  
}

1. .启动顺序：先启动服务注册中心，再启动service-hi，并且servier-hi有两个端口，分别是8089和8090，最后才启动serveice-ribbon，然后访问：

Localhost:8091/hi?name=杰克。刷新页面：





刷新页面，会轮流访问8089和8090。

这说明当我们通过调用restTemplate.getForObject(“http://service-hi/hi?name=“+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。

·一个服务注册中心，eureka server,端口为8088

·service-hi工程跑了两个实例，端口分别为8089，8090，分别向服务注册中心注册

·sercvice-ribbon端口为8091,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:8089和8090两个端口的hi接口。

1. 服务消费者（Feign）

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。

简而言之：

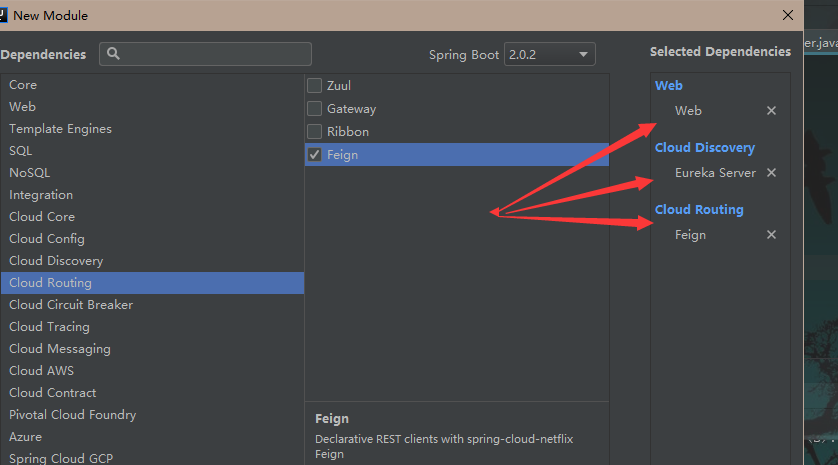
·Feign 采用的是基于接口的注解

·Feign 整合了ribbon

1. .准备工作

先启动服务注册中心，再启动service-hi，并且servier-hi有两个端口，分别是8089和8090

1. .新建一个spring-boot工程，取名为serice-feign



配置文件：

**eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8088/eureka/  
**server:  
 port:** 8092  
**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：

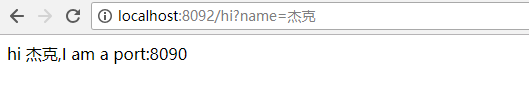
// 通过@ FeignClient（“服务名”），来指定调用哪个服务。  
@FeignClient("service-hi")  
public interface TestService {  
  
 @RequestMapping("/hi")  
 String sayHi(@RequestParam String name);  
}

写一个controlle

@RestController  
public class TestController {  
 @Autowired  
 private TestService testService;  
  
 @RequestMapping("/hi")  
 public String hi(@RequestParam String name){  
 return testService.hiService(name);  
 }  
}

1. .启动项目来访问：localhost:8092





刷新页面，会反复的访问的是8089和8090

1. .Feign的工作原理：

Feign是一个伪客户端，即是它不做任何的请求处理。Feign通过注解生成request，从而实现简化HTTP API开发的目的，即开发人员可以使用注解的方式定制request api模板，在发送http request请求之前，feign通过处理注解的方式替换掉request模板中的参数，这种实现方式显得更为直接、可理解。通过包扫描注入FeignClient的bean：

首先在启动配置上检查是否有@EnableFeignClients注解，如果有该注解，则开启包扫描，扫描被@FeignClient注解的接口。

程序启动后通过包扫描，当类有@FeignClient注解，将注解的信息取出，连同类名一起取出，赋给BeanDefinitionBuilder，然后根据BeanDefinitionBuilder得到beanDefinition，最后beanDefinition式注入到ioc容器中。

注入bean之后，通过jdk的代理，当请求FeignClient的方法时会被拦截。

总到来说，Feign的源码实现的过程如下：

·首先通过@EnableFeignCleints注解开启FeignCleint

·根据Feign的规则实现接口，并加@FeignCleint注解

·程序启动后，会进行包扫描，扫描所有的@ FeignCleint的注解的类，并将这些信息注入到ioc容器中。

·当接口的方法被调用，通过jdk的代理，来生成具体的RequesTemplate

RequesTemplate在生成Request

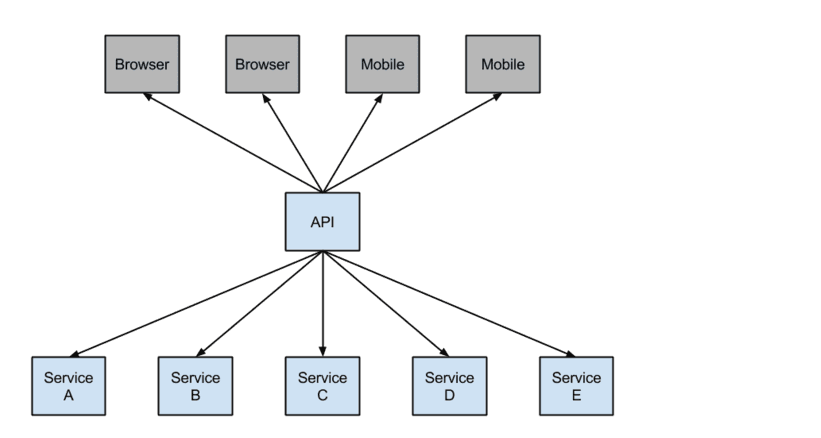
·Request交给Client去处理，其中Client可以是HttpUrlConnection、HttpClient也可以是Okhttp。

·最后Client被封装到LoadBalanceClient类，这个类结合类Ribbon做到了负载均衡。

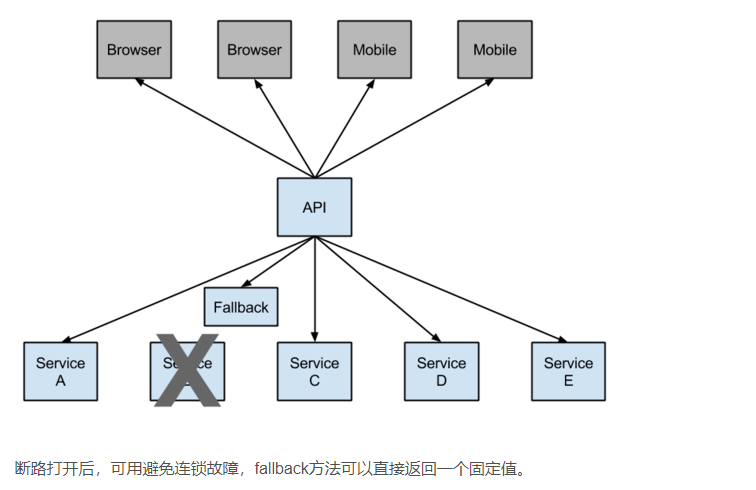
7. 在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以相互调用（RPC），在Spring Cloud可以用RestTemplate+Ribbon和Feign来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。

为了解决这个问题，业界提出了断路器模型。

1. .断路器：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。



1. .在ribbon使用断路器：打开pom，加入依赖：

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

在程序的启动类加@EnableHystrix注解开启Hystrix：

@SpringBootApplication  
// 通过@EnableDiscoveryClient向服务中心注册  
@EnableDiscoveryClient  
// 开启hystris  
@EnableHystrix  
public class ServerRibbonApplication {

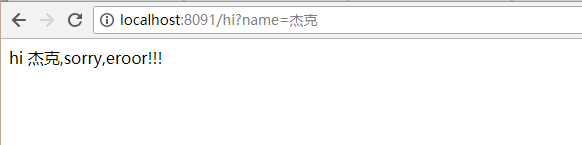
改造service方法：在方法上面加入@HystrixCommand(fallbackMethod = "hiError")注解，hiError是下面方法的名字，遇到错误就返回一个固定值。

@Service  
public class TestService {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 // hiError是对应的方法名字  
 @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,eroor!!!";  
 }  
}

正常访问：



但是当关掉service-hi后：



这就等于是service-hi不可用的时候，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。

1. .在Feign中使用断路器：Feign是自带断路器的。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：

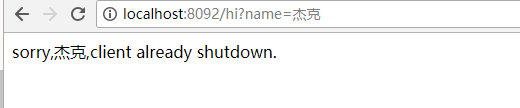
*#打开断路器***feign:  
 hystrix:  
 enabled:** true

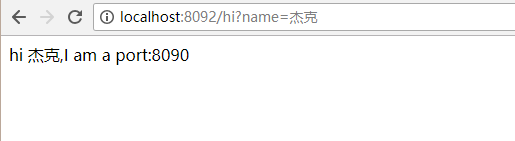
然后在service接口里加上代码：fallback = SchedualServiceHiHystric.class，SchedualServiceHiHystric是实现TestService的类。

// 通过@ FeignClient（“服务名”），来指定调用哪个服务。  
@FeignClient(value = "service-hi",fallback = SchedualServiceHiHystric.class)  
public interface TestService {  
  
 @RequestMapping("/hi")  
 String sayHi(@RequestParam(value = "name") String name);  
}

@Component  
public class SchedualServiceHiHystric implements TestService {  
  
 @Override  
 public String sayHi(String name) {  
 return "sorry," + name + ",client already shutdown.";  
 }  
}

当关掉service-hi或者service-hi出现异常时，会直接断路，访问到的是实现的方法SchedualServiceHiHystric ，反之service-hi正常启动且无异常的话，访问的就是正常的接口TestService 。





1. .Hystrix Dashboard (断路器：Hystrix 仪表盘)

基于ribbon改造，首先得引入依赖包：

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

然后在启动类加上注解：

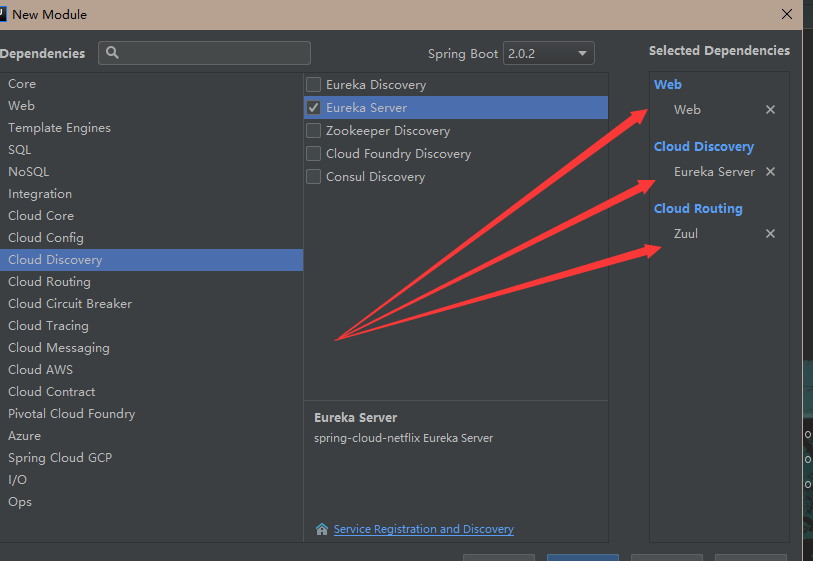
@EnableHystrixDashboard  
@EnableCircuitBreaker

并加入一下代码：

@Bean  
public ServletRegistrationBean getServlet(){  
 HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();  
 ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);  
 registrationBean.setLoadOnStartup(1);  
 registrationBean.addUrlMappings("/hystrix.stream");  
 registrationBean.setName("HystrixMetricsStreamServlet");  
 registrationBean.setName("HystrixMetricsStreamServlet");  
 return registrationBean;  
  
}

在Feign改造：与ribbon类似，但是启动类不要加@EnableCircuitBreaker注解

1. Zuul的主要功能是路由转发和过滤器。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。
2. .创建一个新的model，命名为service-zuul



配置文件：

**eureka:  
 client:  
 service-url:  
 defaultZone:** http://loaclhost:8088/eureka/  
**server:  
 port:** 8092  
**spring:  
 application:  
 name:** service-zuul  
**zuul:  
 routes:  
 api-a:  
 path:** /api-a/\*\*  
 **serviceId:** service-ribbon  
 **api-b:  
 path:** /api-b/\*\*  
 **serviceId:** service-feign

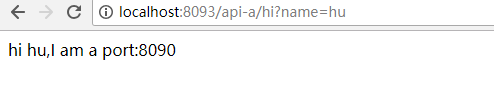
首先指定服务注册中心的地址为http://localhost:8088/eureka/，服务的端口为8092，服务名为service-zuul；以/api-a/ 开头的请求都转发给service-ribbon服务；以/api-b/开头的请求都转发给service-feign服务。

在启动类加上注解：

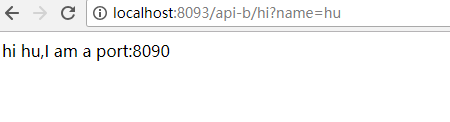
@EnableEurekaClient  
@EnableZuulProxy

然后依次启动5个model：

访问：http://localhost:8093/api-a/hi?name=hu，显示如下：



访问：http://localhost:8093/api-b/hi?name=hu，显示如下：



访问到，则说zuul已起到了路由的作用！

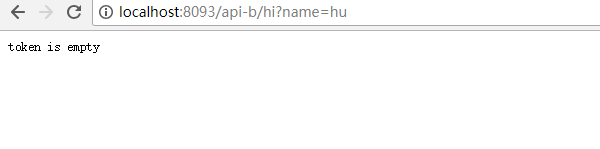
1. .服务过滤，zuul不仅起到路由，还有过滤，可做一些安全验证。

改造代码：

@Component  
public class MyFilter extends ZuulFilter {  
 private static Logger *log* = LoggerFactory.*getLogger*(MyFilter.class);  
 @Override  
 public String filterType() {  
 return "pre";  
 }  
  
 @Override  
 public int filterOrder() {  
 return 0;  
 }  
  
 @Override  
 public boolean shouldFilter() {  
 return true;  
 }  
  
 @Override  
 public Object run() throws ZuulException {  
 // 获取上下文  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 HttpServletRequest request = ctx.getRequest();  
 *log*.info(String.*format*("%s >>> %s",request.getMethod(),request.getRequestURI().toString()));  
 Object accessToken = request.getParameter("token");  
 if(accessToken == null){  
 *log*.info("token is empty");  
 ctx.setSendZuulResponse(false);  
 ctx.setResponseStatusCode(401);  
 try {  
 ctx.getResponse().getWriter().write("token is empty");  
 }catch (Exception e){}  
 }  
 *log*.info("ok");  
 return null;  
 }  
}

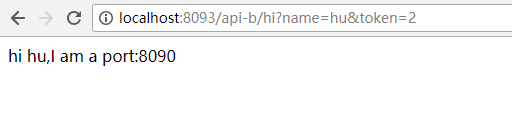
然后再浏览器访问：

http://localhost:8093/api-b/hi?name=hu，直接被拦截



<http://localhost:8093/api-b/hi?name=hu&token=2>

有数据。



1. Spring cloud sonfig

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。

1. .构建config service

<?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.rrg</groupId>  
 <artifactId>service-confog</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <name>service-confog</name>  
 <description>Demo project for Spring Boot</description>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>1.5.2.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.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-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
  
 <dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Camden.SR6</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>  
  
 <repositories>  
 <repository>  
 <id>spring-milestones</id>  
 <name>Spring Milestones</name>  
 <url>https://repo.spring.io/milestone</url>  
 <snapshots>  
 <enabled>false</enabled>  
 </snapshots>  
 </repository>  
 </repositories>  
  
  
</project>

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

1. .注意的是，目前用的版本boot版本是2.02，spring-cloud的版本是： <spring-cloud.version>Finchley.RC2</spring-cloud.version>有个bug，需要把RC2改为RC1。

配置文件：

spring.application.name=service-config  
server.port=8888  
  
spring.cloud.config.server.git.uri=https://github.com/huangshengz/huangsz  
spring.cloud.config.server.git.search-paths=respo  
spring.cloud.config.label=master  
spring.cloud.config.server.git.username=  
spring.cloud.config.server.git.password=

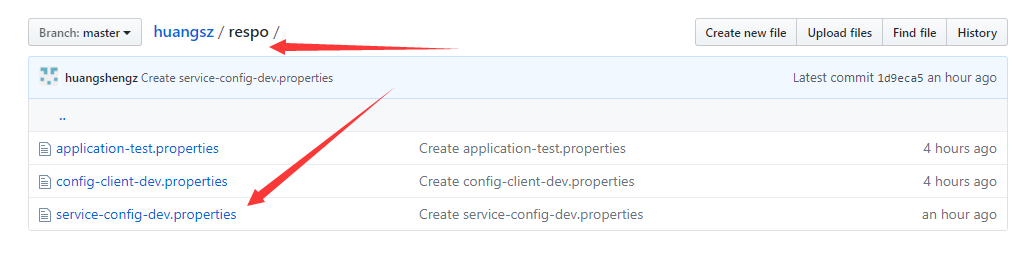
·spring.cloud.config.server.git.uri是github地址

·spring.cloud.config.server.git.search-paths是路径，文件夹路径

·spring.cloud.config.label是分支

·spring.cloud.config.server.git.username账号，如果git是public的话不用填写

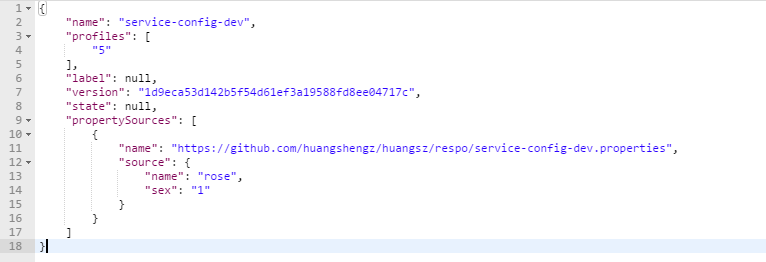
·spring.cloud.config.server.git.password密码，如果git是public的话不用填写

1. .

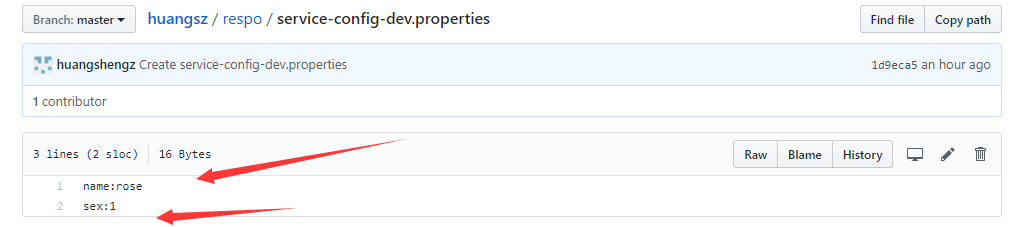
·respo就是search-paths路径，而git路径是直至仓库名的

·http://localhost:8888/service-config-dev/dev访问时，端口后访问的是文件名，不加后缀，service-config-dev/dev

下面是返回信息：



以下是git文件里的内容：



至此，说明我们已经成功的读取到git远程里的配置文件信息了。

5）.然后构建一个config client，再建一个model，命名为config-client:

<?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.rrg</groupId>  
 <artifactId>config-client</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <name>config-client</name>  
 <description>Demo project for Spring Boot</description>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>1.5.2.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.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.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.RC1</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>  
  
 <repositories>  
 <repository>  
 <id>spring-milestones</id>  
 <name>Spring Milestones</name>  
 <url>https://repo.spring.io/milestone</url>  
 <snapshots>  
 <enabled>false</enabled>  
 </snapshots>  
 </repository>  
 </repositories>  
  
  
</project>

配置文件：bootstrap.propertes

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

·spring.application.name=configTest-test #名字configTest-test是github上要读取的文件名的前缀，即xxx1。Github上文件命名为：xxx1-xxx2.properties。

·spring.cloud.config.label=master #对应的是cloud config配置文件的label分支。

·pring.cloud.config.profile=dev #dev就是xxx2

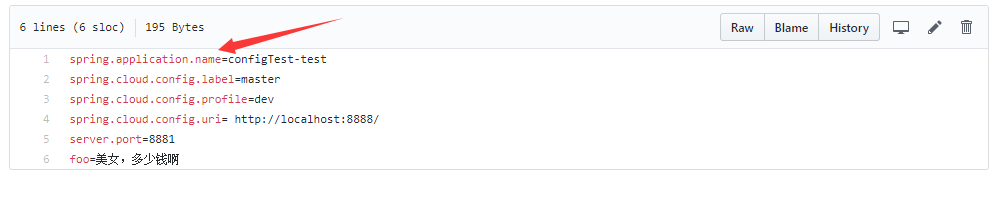
·spring.cloud.config.uri= <http://localhost:8888/> #指明服务中心网址。

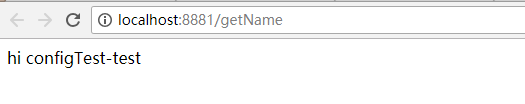
改造一下请求：

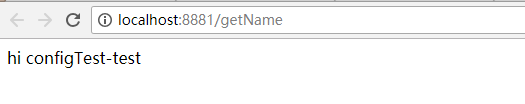
@SpringBootApplication  
@RestController  
public class ConfigClientApplication {  
   
 public static void main(String[] args) {  
 SpringApplication.*run*(ConfigClientApplication.class, args);  
 }  
  
 @Value("${spring.application.name}")  
 private String name;  
 @RequestMapping("/getName")  
 public String getName(){  
// name = env.getProperty("foo");  
 return "hi " + name;  
 }  
}

请求：<http://localhost:8881/getName>

spring.application.name是config读取到的配置文件，如下图：



请求的结果应该是 hi configTest-test：



总结：config service从git仓库读取配置文件，然后client读取config service读取出来的配置文件的属性。

1. 高可用的分布式配置中心：spring cloud config
2. .在server config的启动类添加注解：@EnableEurekaClient
3. .server config和clinet都加入eureka的依赖包。

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

1. .server config的配置文件要加入配置指向注册中心：

eureka.client.service-url.defaultZone=http://localhost:8088/eureka/

Server client的配置文件则需要加入配置：

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

#指明服务注册中心

eureka.client.service-url.defaultZone=http://localhost:8088/eureka/

#从配置中心读取文件  
spring.cloud.config.discovery.enabled=**true**

**#配置中心的servieId，即服务名**spring.cloud.config.discovery.service-id=service-config

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。

1. 消息总线（spring cloud bus）

在使用spring cloud bus之前，先了解且需要安装RabbitMQ和Kafka消息中间件。并且Spring Cloud Bus仅支持他们两个。

1. .RabbitMQ实现消息总线

RabbitMQ是实现了高级消息列队协议（AMQP）的开源消息代理软件，也称为面向消息的中间件。

高级消息列队协议：Advanced Message Queuing Protocol.

·Broker：可以理解为消息队列服务器的实体，它是一个中间件应用，负责接收消息生产者的消息，然后将消息发送至消息接收者或其他的Broker。

·Exchange：消息交换机，是消息第一个到达的地方，消息通过它制定的路由规则，分发到不同的消息队列中去。

·Queue：消息队列，消息通过发送和路由之后最终到达的地方，到达queue的消息即进入逻辑上等待消费的状态。每个消息都会被发动到一个或多个队列。

·Binding：绑定，把Exchange和Queue按照路由规则绑定起来，也就是Exchange和Queue之间的虚拟链接。

·Routing Key：路由关键字，Exchange会根据这个关键字进行消费投递。

·Virtual host：虚拟主机

·Connection：连接

·Channel：消息通道，用于连接生产者和消费者的逻辑结构，在客户端的每个连接里，可建立多个Channel，每个Channel代表一个会话服务，通过Channel可以隔离同一连接中的不同交互任务。

·Producer：消费生产者：制造消息并发送消息的程序

·Consumer：消息消费者：接收消息并处理消息的程序

消息投递到队列的过程：

·客户端连接到消息队列服务器，打开一个Channel

·客户端声明一个Exchange，并设置相关属性

·客户端声明一个Queue，并设置相关属性

·客户端使用Routing Key，在Exchange和Queue之间建立好绑定关系

·客户端投递消息到Exchange

·Exchange收到消息后，根据消的key和已设置的Binding，进行消息路由，将消息头送到一个或多个Queue里。

·RabbitMQ支持消息持久化，也就是将数据写在磁盘上，为了数据的安全，大多数都会选择持久化，消息队列持久化包括三个部分：

① Exchange持久化，声明时指定durable => 1

② Queue持久化，声明时指定durable => 1

③ 消息持久化，在投递时指定delivery\_mode => 2 （1是非持久化的）

·如果Exchange和Queue的都是持久化的，那么他们之间的Binding也是持久化的，如果两者之一一个是持久化的，一个不是，就不允许建立绑定。

安装与使用，版本+地址，依次安装：

Erlang/OTP 20.3

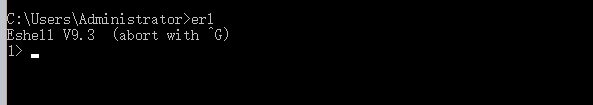
http://www.erlang.org/downloads/19.1

版本+地址：

RabbitMQ server 3.7.5 http://www.rabbitmq.com/download.html#server-installation-guides

配置ERLANG的环境变量：在path加上D:\Program Files\erl9.3\bin

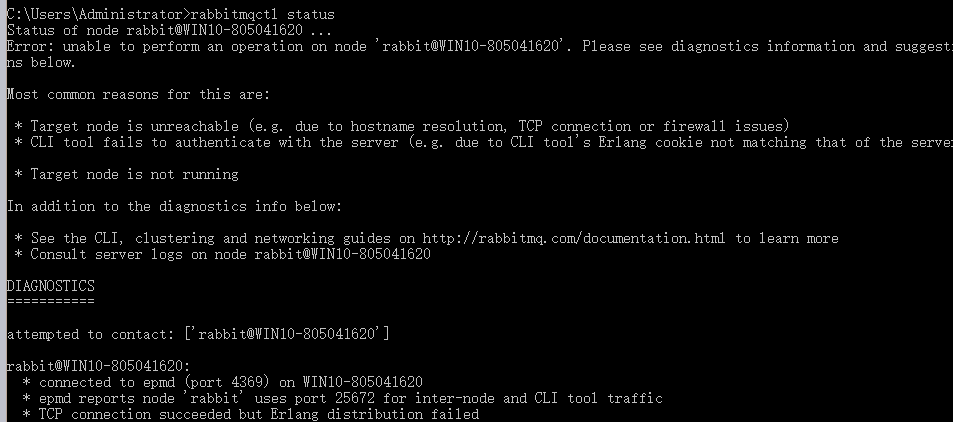
在dom里输入erl看看是否安装成功



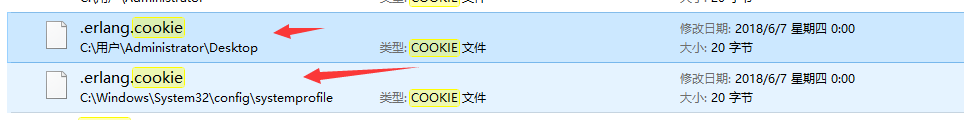
同理配置RabbitMQ的环境变量：

在path后面加上D:\Server\RabbitMQ\rabbitmq\_server-3.7.5\sbin

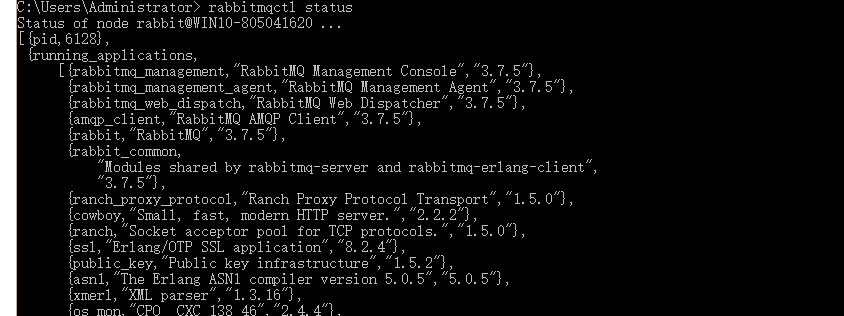
在dom里输入rabbitmqctl status看看是否安装成功



出现以上情况说明出问题了，原因是erlang和rabbitmq的.erlang.cookie不一致，在c盘搜索.erlang.cookie，把其中一处的覆盖另外一处即可。



重新输入：rabbitmqctl status



成功。

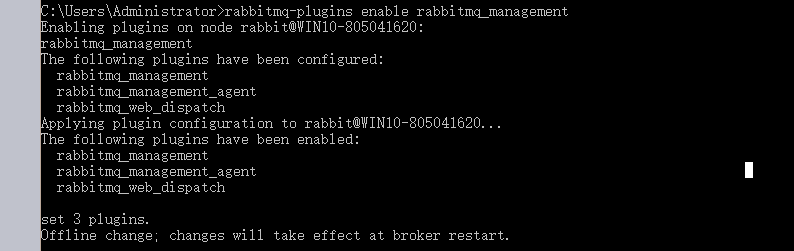
Rabbit管理：通过访问web进行管理，也可以访问配置文件。

·以下是通过访问web进行管理。

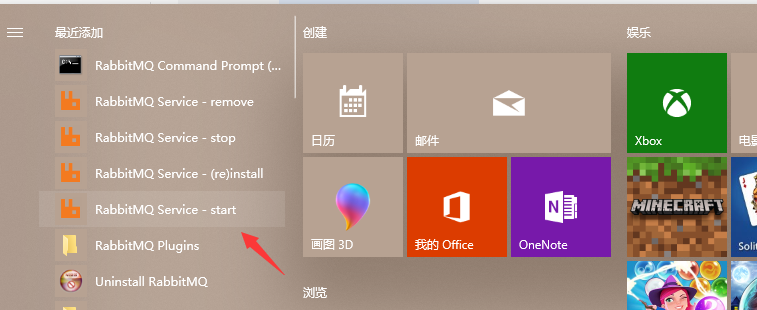
激活 RabbitMQ's Management Plugin，使用RabbitMQ 管理插件，可以更好的可视化方式查看Rabbit MQ 服务器实例的状态。

在dom里输入：

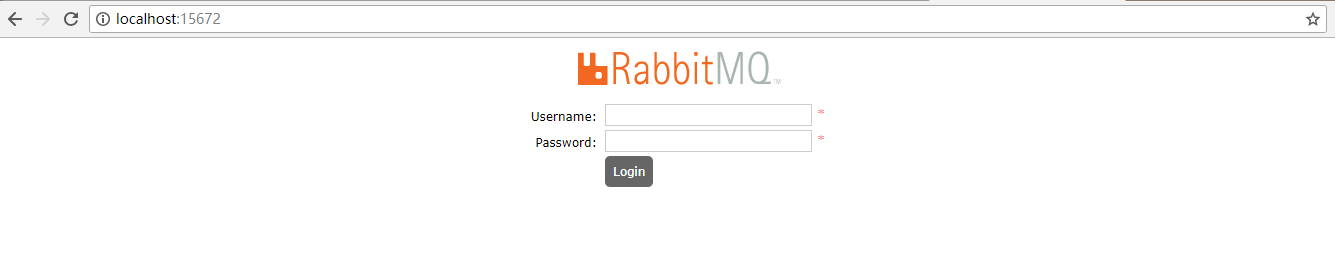
D:\Server\RabbitMQ\rabbitmq\_server-3.7.5\sbin\rabbitmq-plugins.bat enable rabbitmq\_management



启动RabbitMQ Service:

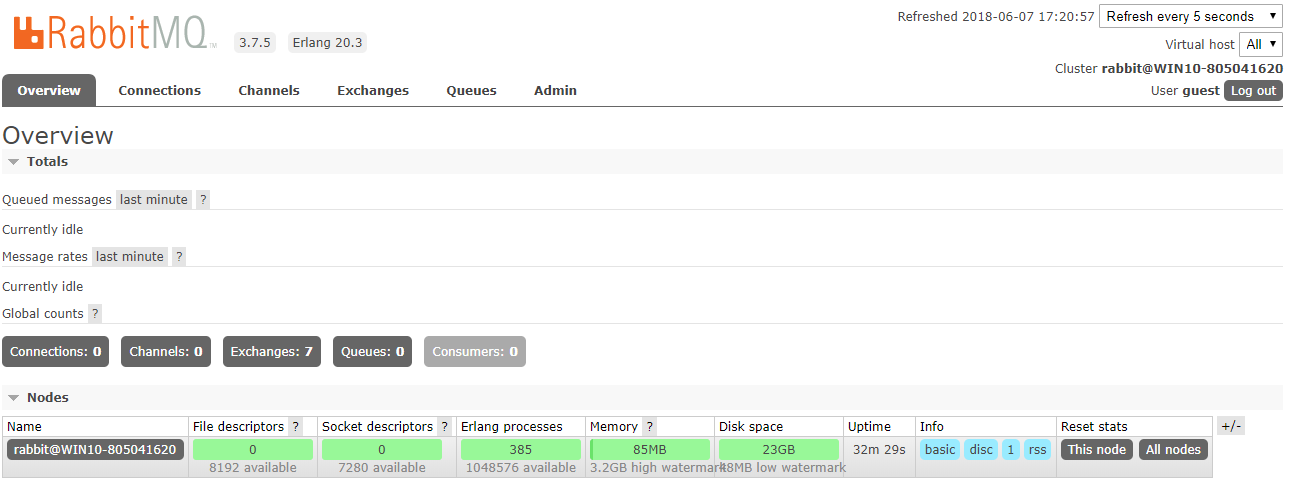


在浏览器输入localhost:15672访问

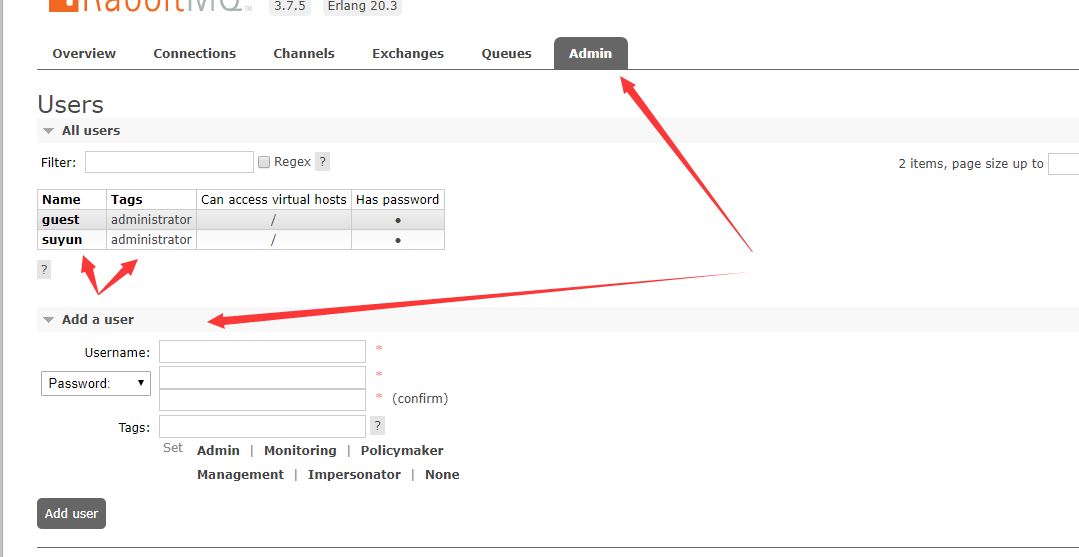


用户名和密码默认都是：guest

登陆后页面：



点击Admin创建一个用户：



改造server-client的代码，pom加上amqp依赖：

<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>

配置文件新增配置：

#保证调用 /bus/refresh的时候不需要验证

management.security.enabled=false

spring.rabbitmq.host=localhost

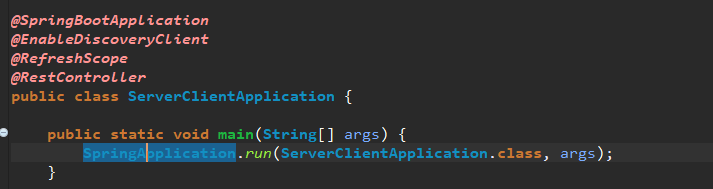
spring.rabbitmq.port=5672

spring.rabbitmq.username=suyun

spring.rabbitmq.password=123456

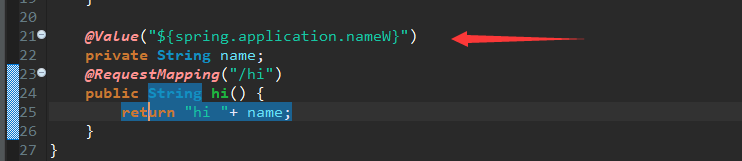
用户名和密码是在rabbitmq新增的时候设置的。

在启动类里心中注解：@EnableDiscoveryClient@RefreshScope，如下：



依次启动eureka-server，config，client。

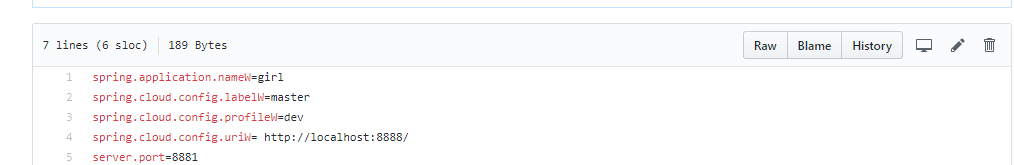
Client：http://localhost:8881/hi获取config从git读取的文件属性：



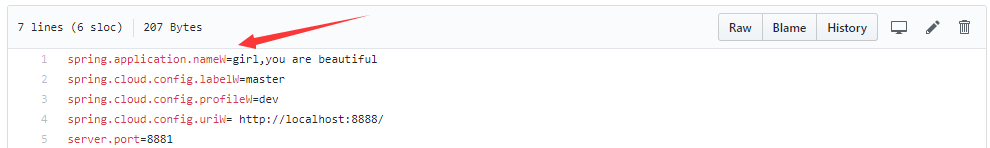
Config：http://localhost:8888/config-client-dev/master从git读取出的配置文件信息：



Github里配置文件信息：



然后直接修改github里的属性值：

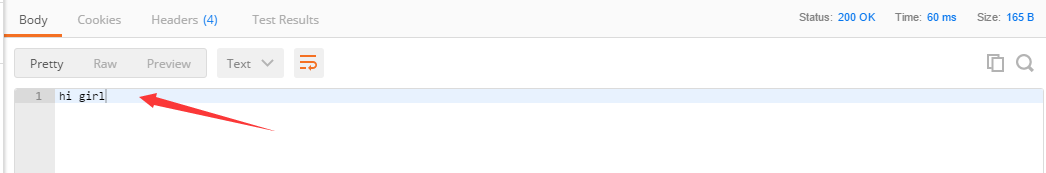


如果换作是传统的项目，则需要重启项目，然后再用client再次获取其属性值，就会获得修改后的值。但在springcloud，我们仅仅只需要用post的方式去请求：

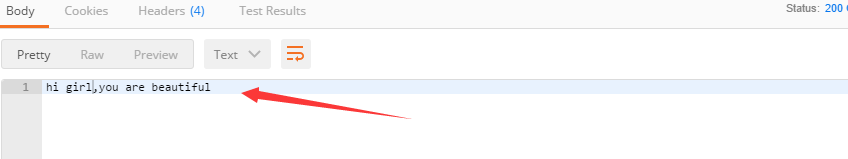
http://localhost:8881/bus/refresh，然后用client再次获取其属性值，就会获得修改后的值。

如下：

Client：http://localhost:8881/hi获取config从git读取的文件属性，刷新前：



用http://localhost:8881/bus/refresh刷新一下后再去取值：



1. 服务链路追踪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，只需要在pom文件中引入相应的依赖即可。

1. .Spring Cloud Sleuth集成了zipkin服务追踪组建
2. .微服务架构是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。

1. .构建三个工程：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。

·server-zipkin

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.rrg</groupId>

<artifactId>server-zipkin</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>server-zipkin</name>

<description>Demo project for Spring Boot</description>

<parent>

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

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

<version>1.5.2.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-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>Camden.SR6</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

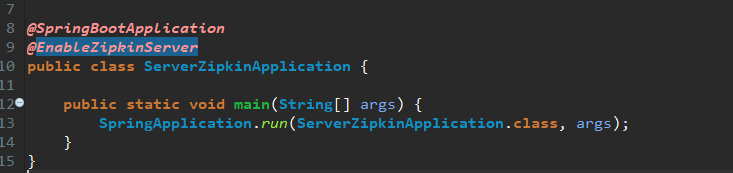
</snapshots>

</repository>

</repositories>

</project>

然后在启动类, 加上注解@EnableZipkinServer，开启ZipkinServer的功能：



在配置文件加上端口号：server.port:9411

·service-hi

<?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.rrg</groupId>

<artifactId>server-hi</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>server-hi</name>

<description>Demo project for Spring Boot</description>

<parent>

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

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

<version>1.5.2.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.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</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>Camden.SR6</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

配置文件：

server.port=8988

#指定zipkin server的地址

spring.zipkin.base-url=http://localhost:9411

spring.application.name=service-hi

写点逻辑：

***@SpringBootApplication***

***@RestController***

**public** **class** **ServerHiApplication** {

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

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

}

**private** **static** **final** Logger ***log*** = **LoggerFactory**.*getLogger*(**ServerHiApplication**.**class**);

***@Autowired***

**private** **RestTemplate** restTemplate;

***@Bean***

**public** **RestTemplate** **getRestTemplate**(){

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

}

***@RequestMapping***("/hi")

**public** **String** **hi**() {

***log***.info("calling trace service-hi ");

**return** restTemplate.getForObject("http://localhost:8989/miya", **String**.**class**);

}

***@RequestMapping***("/getService")

**public** **String** **getService**(){

**System**.***out***.println("infoinfoinfo");

***log***.info("calling trace service-hi ");

**return** "我是service-hi，首先我先调用自己的hi方法，hi就指向了service-miya，miya转过头指向了我的info！";

}

***@Bean***

**public** **AlwaysSampler** **defaultSampler**(){

**return** **new** AlwaysSampler();

}

}

·server-miya

pom文件和server-hi一样

配置文件：

server.port=8989

spring.zipkin.base-url=http://localhost:9411

spring.application.name=service-miya

启动类写点逻辑：

***@SpringBootApplication***

***@RestController***

**public** **class** **ServerMiyaApplication** {

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

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

}

**private** **static** **final** Logger ***log*** = **LoggerFactory**.*getLogger*(**ServerMiyaApplication**.**class**);

***@RequestMapping***("/hi")

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

***log***.info( "hi is being called");

**return** "hi i'm miya!";

}

***@RequestMapping***("/miya")

**public** **String** **info**(){

***log***.info( "info is being called");

**return** restTemplate.getForObject("http://localhost:8988/getService",**String**.**class**);

}

***@Autowired***

**private** **RestTemplate** restTemplate;

***@Bean***

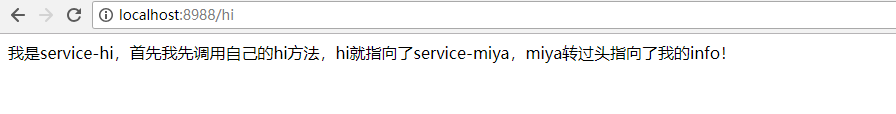
**public** **RestTemplate** **getRestTemplate**(){

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

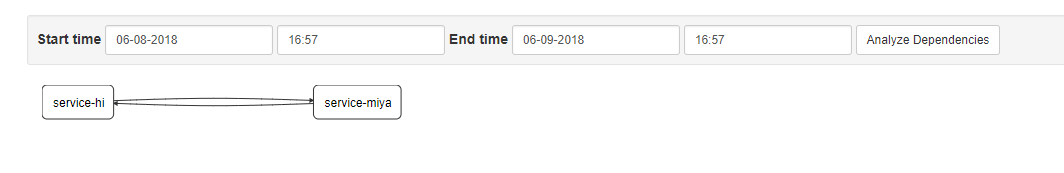
}

}

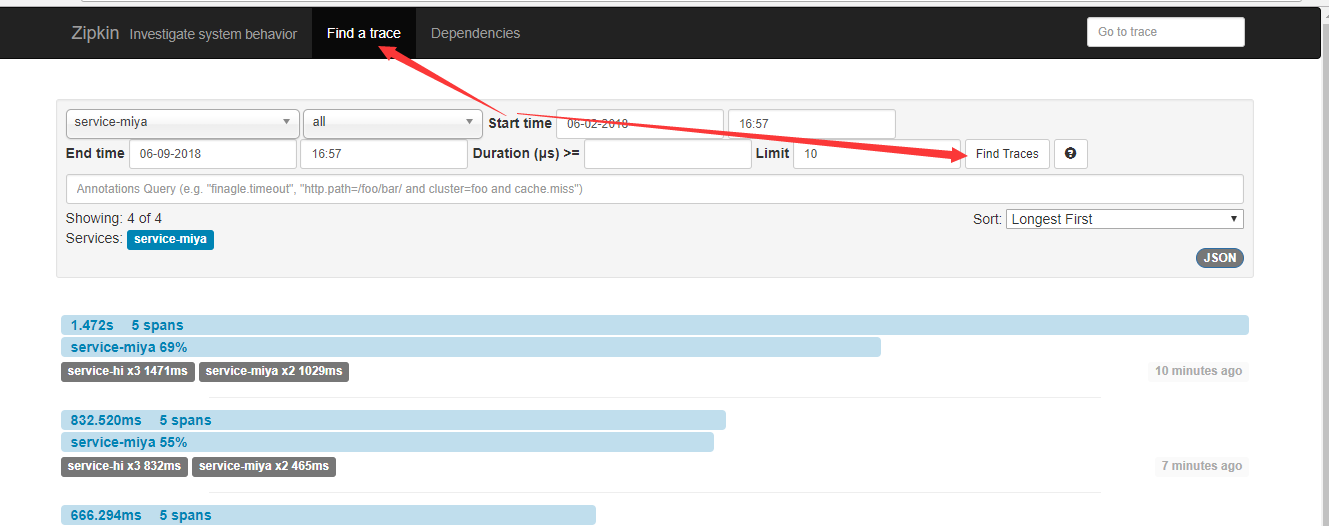
然后执行：localhost:8988/hi会得到以下结果：



Zipkin可以看到：



两个服务之间的对调。



可以看到调用方法的耗时。

1. 高可用的配置中心
2. .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。所以要将将Eureka Server集群化。
3. .创建两个注册中心，分别叫做：server-eurekahi，server-highteureka

配置文件分别为：

server-eurekahi的配置

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

spring.profiles.active=peer1

eureka.instance.hostname=peer1

eureka.client.service-url.defaultZone:http://peer2:8762/eureka/

Server-highteureka的配置

server.port=8762

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

spring.application.name=peer2

eureka.instance.hostname=peer2

eureka.client.service-url.defaultZone:http://peer1:8761/eureka/

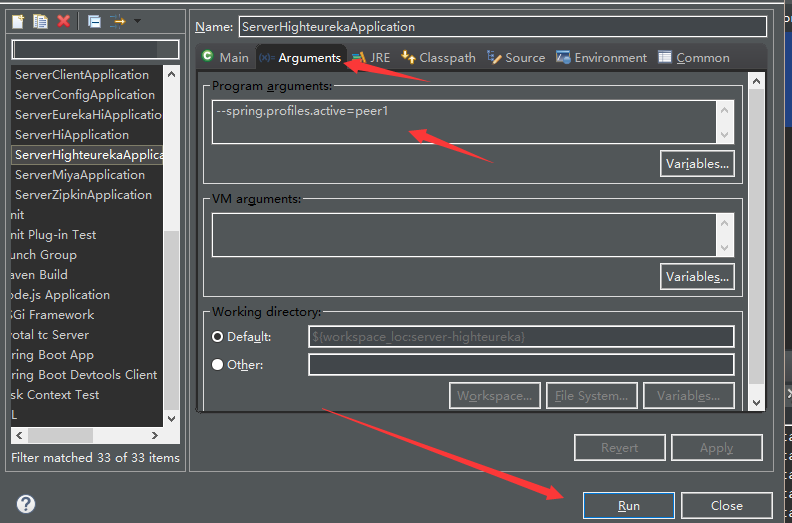
1. .分别启动：启动方式有两种，如果打包在linux下启动：

Java -jar 包1 --spring.profiles.active=peer1

Java -jar 包2 --spring.profiles.active=peer2

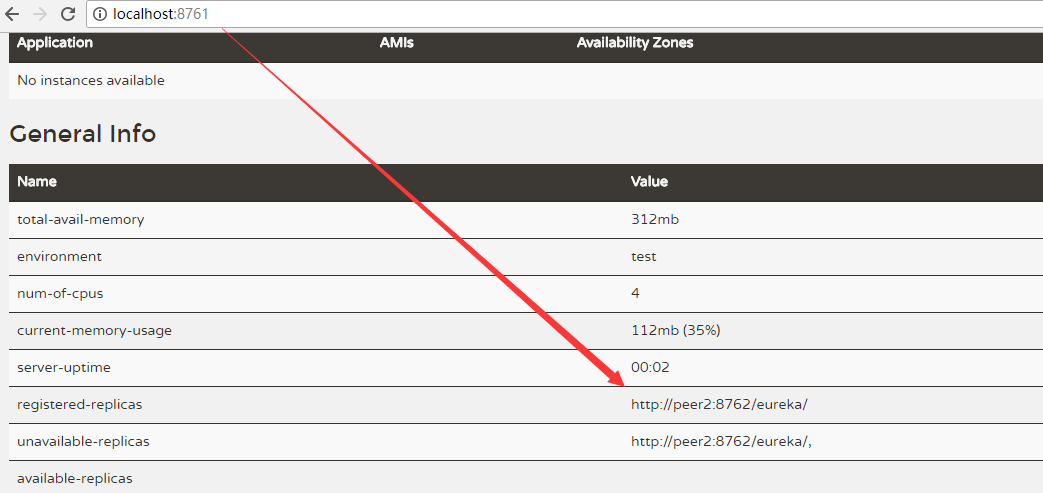
如果在tomcat直接启动，则需要添加参数：--spring.profiles.active=peer1。

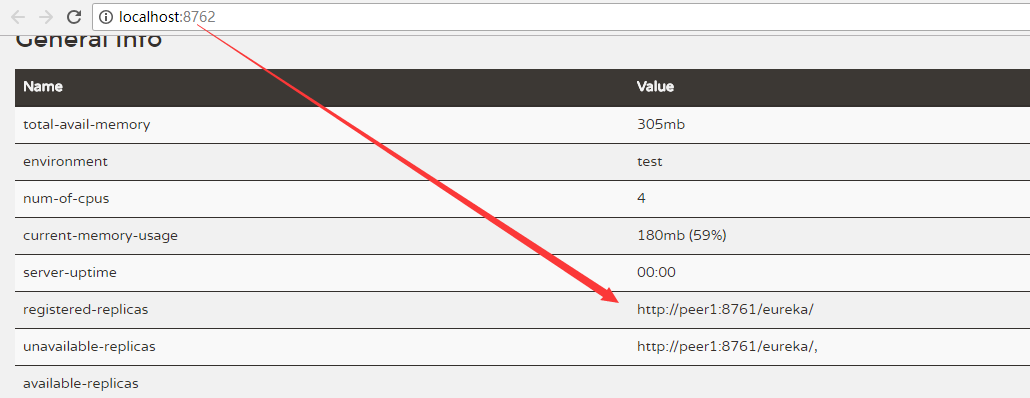
例如：点击启动类，run as-->run configurations



添加完参数后直接启动。

完成后分别打开：localhost:8761,localhost:8762





创建一个server-client1，

配置文件：向两个服务中心注册

server.port=8763

spring.application.name=service-client1

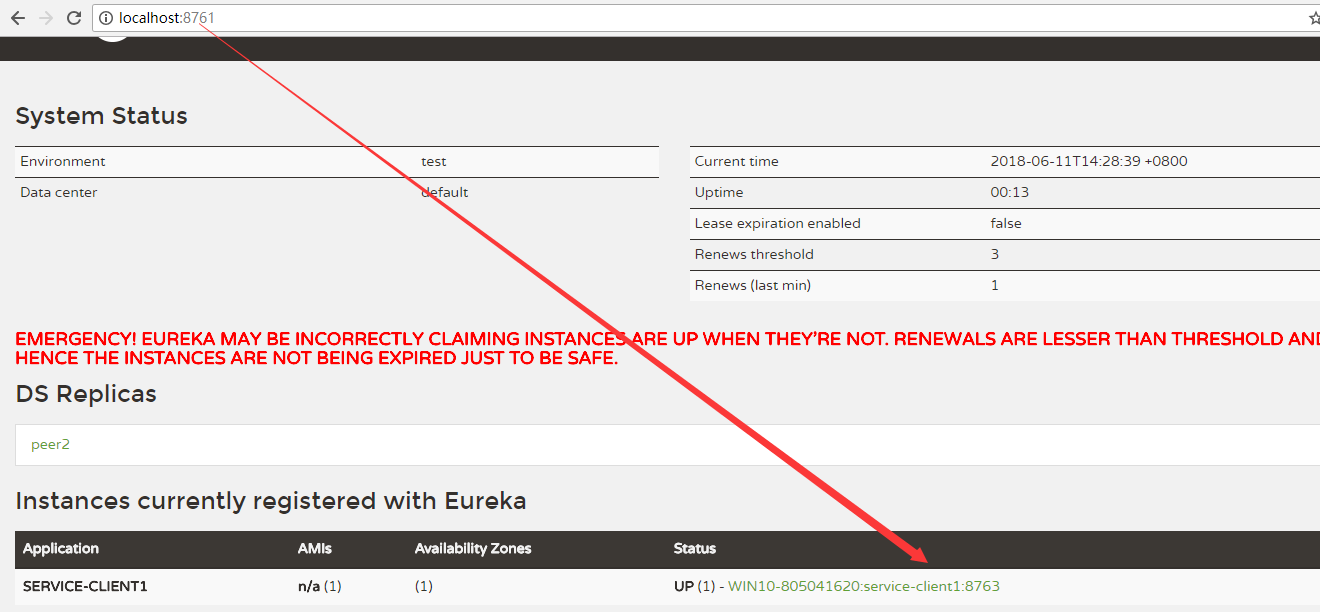
#向服务中心注册

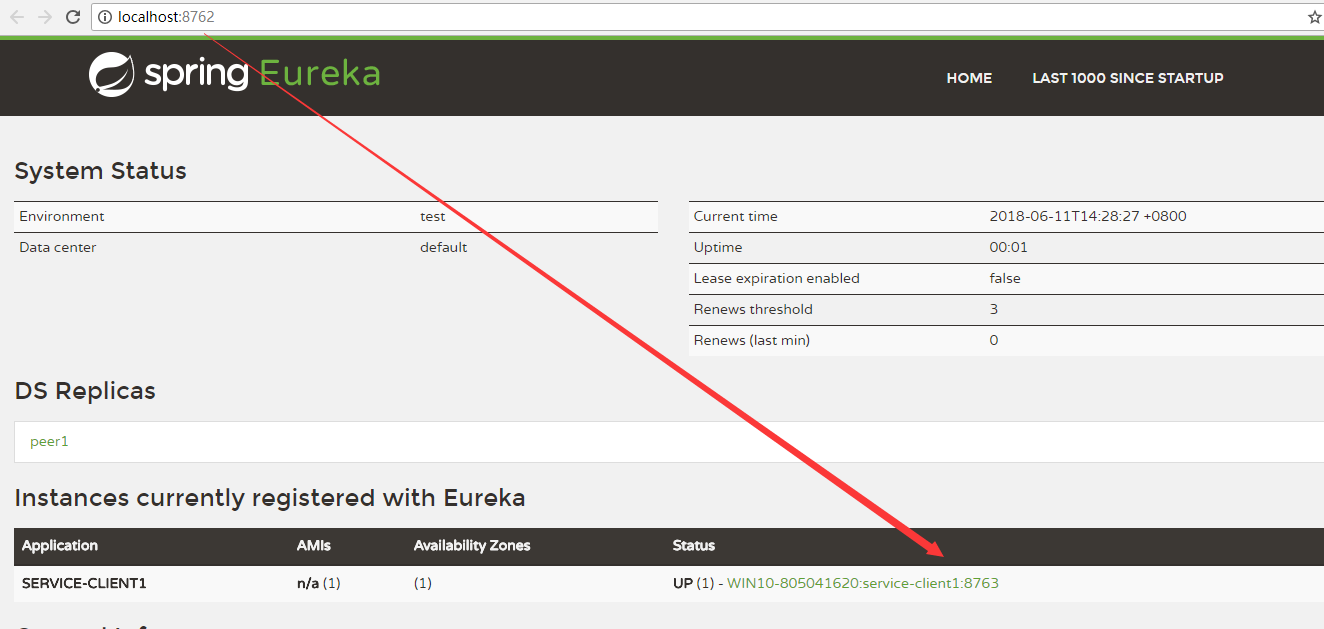
eureka.client.service-url.defaulZone=http://peer1:8761/eureka/,http://peer2:8762/eureka/



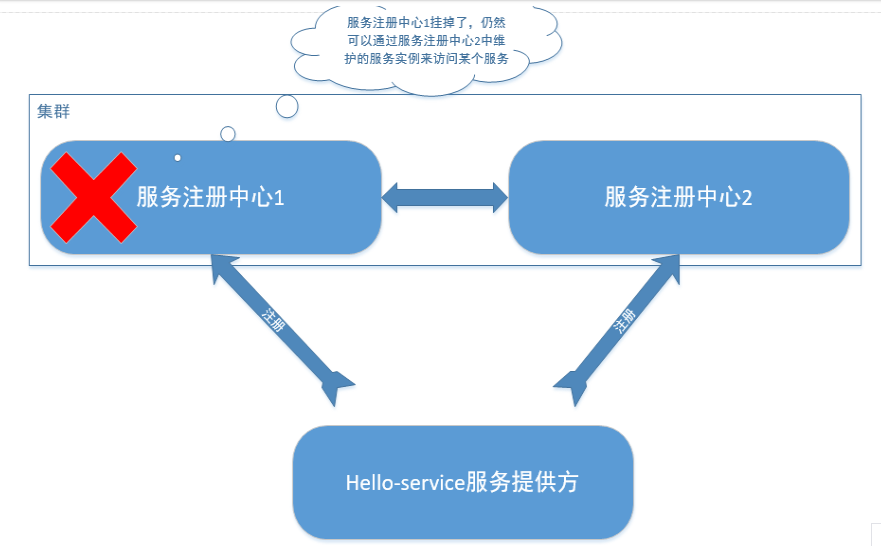
改造一下启动类。

分别启动三个服务，8761，8762，8763



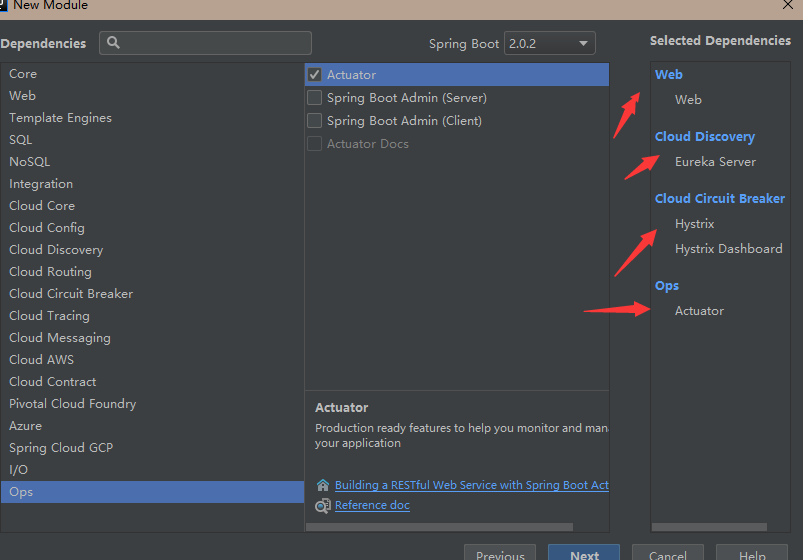


这时候发现，client1已经都向8761，8762注册了。



1. docker部署springcloud项目
2. 断路器监控Hystrix Dashboard

创建一个service-hi，依赖包如下：



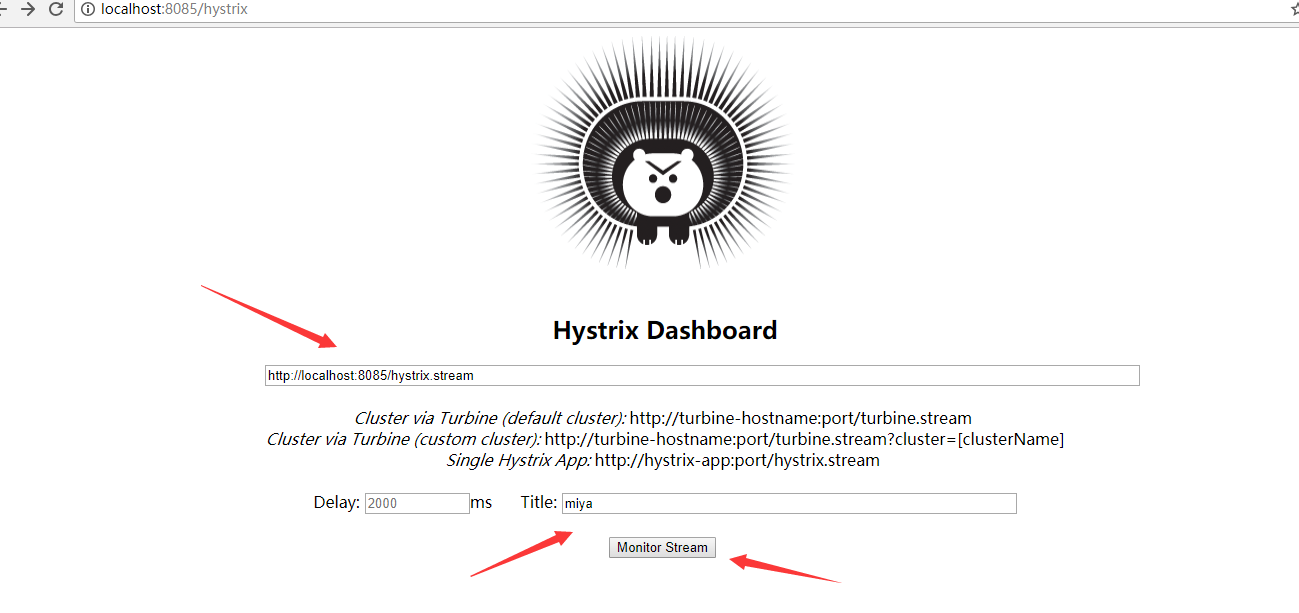
配置文件：

server.port=8085  
eureka.client.service-url.defaultZone=http://localhost:8088/eureka/  
spring.application.name=service-hi

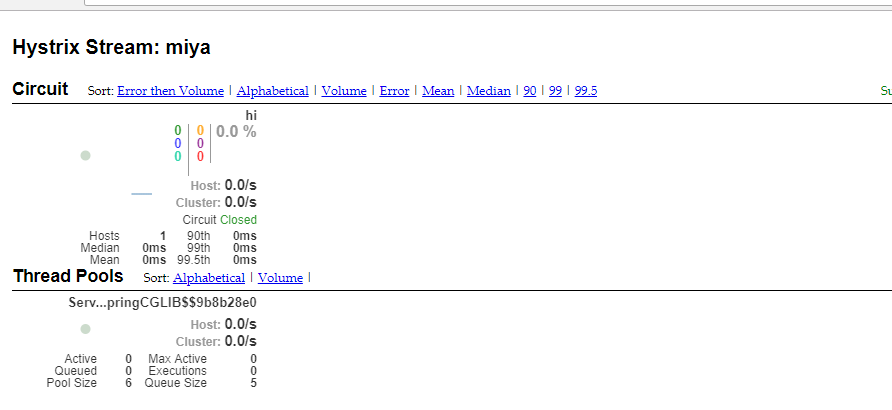
启动类新增几个注解：

@SpringBootApplication  
@EnableEurekaClient  
// 断路器  
@EnableHystrix  
// 断路器仪表盘  
@EnableHystrixDashboard  
@RestController  
public class ServiceHiApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ServiceHiApplication.class, args);  
 }  
  
 // 这个是2.02要添加的，不然仪表盘不显示  
 @Bean  
 public ServletRegistrationBean getServlet(){  
 HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();  
 ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);  
 registrationBean.setLoadOnStartup(1);  
 registrationBean.addUrlMappings("/hystrix.stream");  
 registrationBean.setName("HystrixMetricsStreamServlet");  
 registrationBean.setName("HystrixMetricsStreamServlet");  
 return registrationBean;  
 }  
  
 @Value("${server.port}")  
 private String port;  
 @RequestMapping("/hi")  
 @HystrixCommand(fallbackMethod = "hiErr")  
 public String hi(@RequestParam String name){  
 return "hi," + name + "I am a port:" + port;  
 }  
  
 // 当服务中心挂掉时，断路器要跑的方法  
 public String hiErr(@RequestParam String name){  
 return "sorry," + name + ",have a error";  
 }  
}

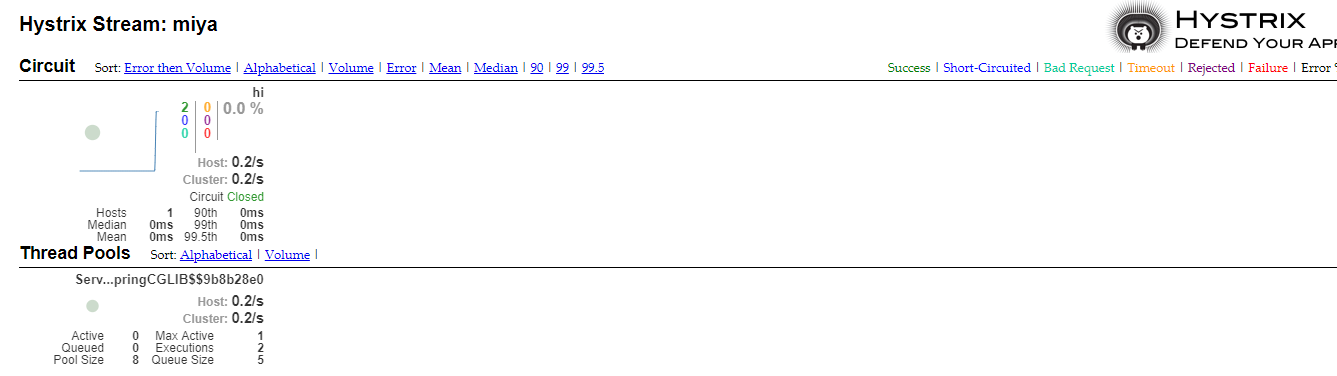
启动注册中心服务和service-hi,访问http://localhost:8085/hystrix



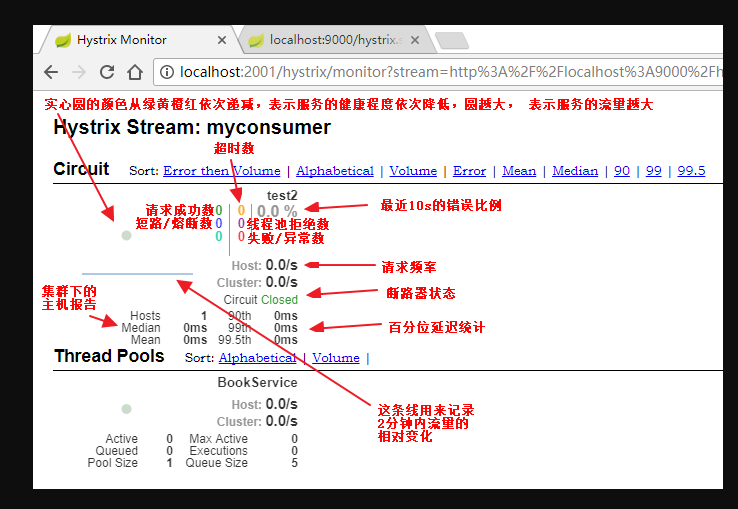
然后输入：<http://localhost:8085/hystrix.stream，miya，点击Monitor>



上面显示的就是仪表盘，我们们访问：<http://localhost:8085/hi?name=rose>

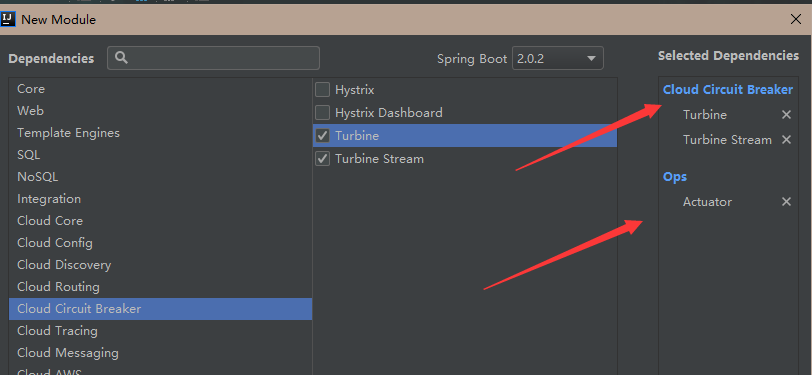


就会看到仪表盘的变化。



1. 断路器聚合监控Hystrix Turbine

看单个的Hystrix Dashboard的数据并没有什么多大的价值，要想看这个系统的Hystrix Dashboard数据就需要用到Hystrix Turbine。Hystrix Turbine将每个服务Hystrix Dashboard数据进行了整合。Hystrix Turbine的使用非常简单，只需要引入相应的依赖和加上注解和配置就可以了。



1. 服务注册consul