springCloud

工具：Idea，JDK1.8,Maven3.5,mysql

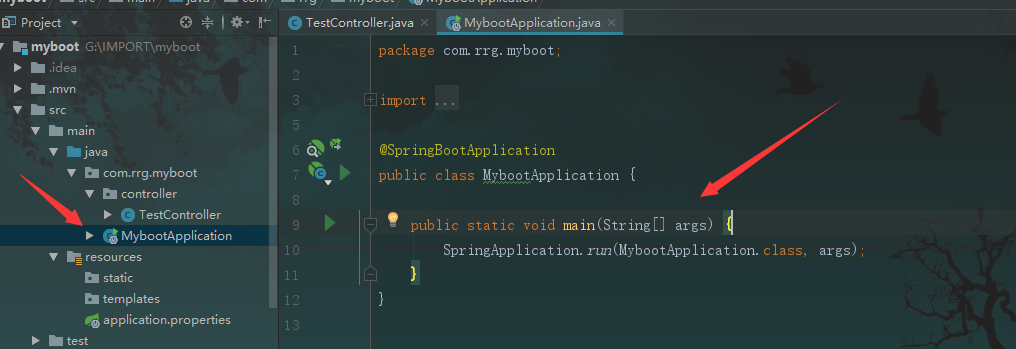
1. 创建第一个springboot工程

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。

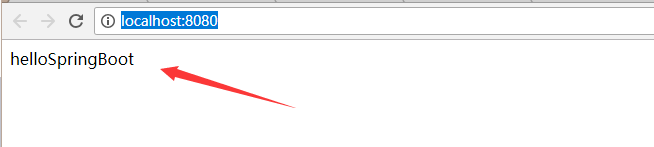
2）.在controller写下如下代码：

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

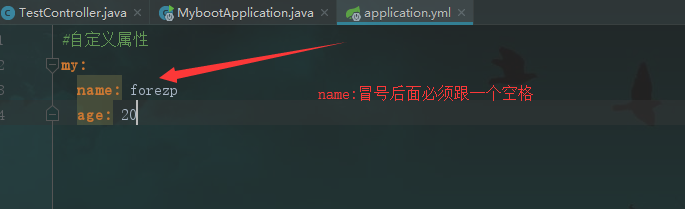
1. .打开MyBootApplication，并运行main方法，即是运行了项目。



1. .打开浏览器，输入：localhost:8080，回车

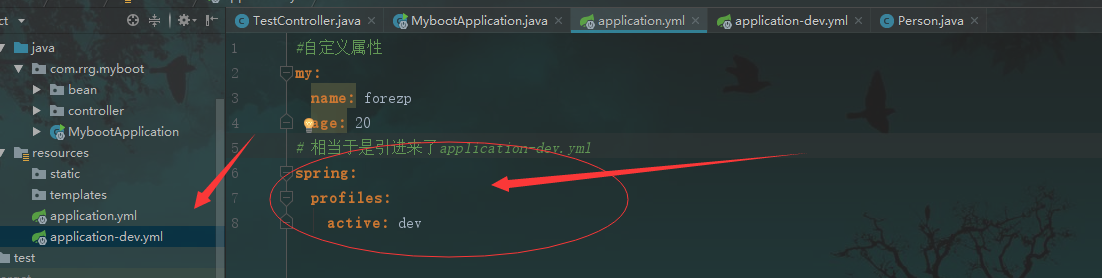


1. .ok，至此springboot已成功建立。
2. springboot的配置文件：properties.yml
3. [.自定义属性，我们可用@Value(“${my.name}”)获取](mailto:.自定义属性，我们可用@Value(\“${my.name}\”)获取)



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

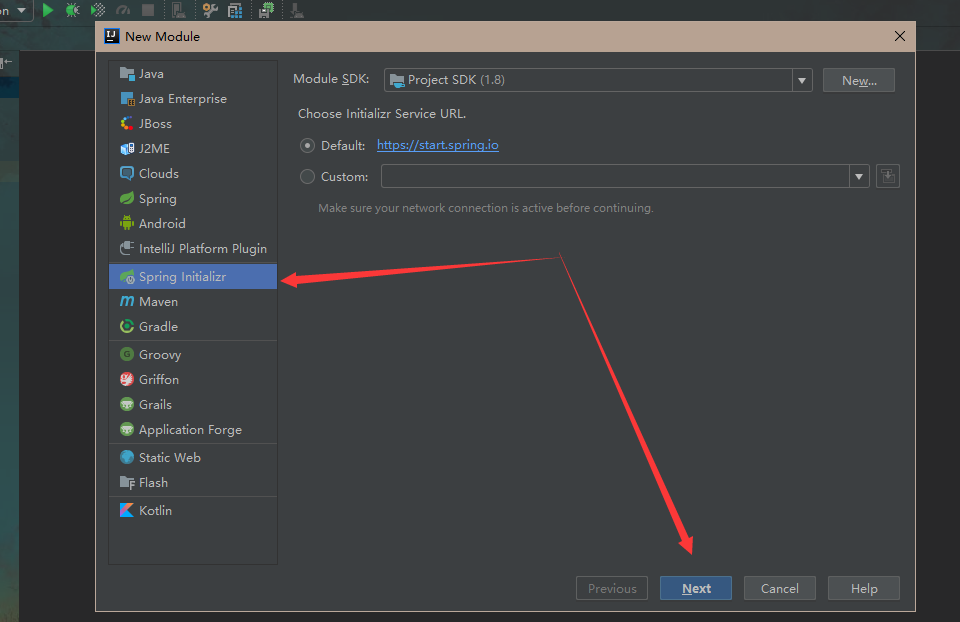
1. .开启事务，在service层的方法上加上注解即可：@Transactional
2. spring cloud为开发人员提供了一套快速构建分布式系统的一些工具，包括：配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会等等。
3. .创建服务注册中心

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

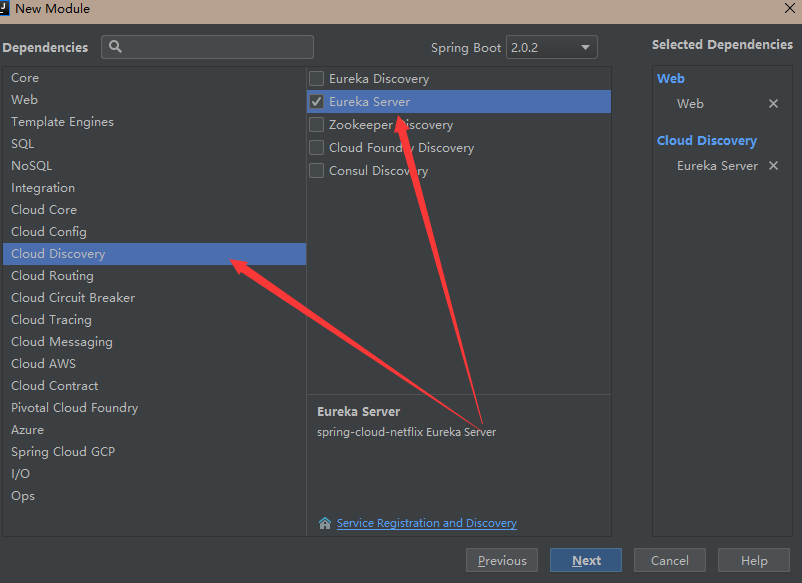
在springboot的基础上，创建两个model工程，一个是服务注册中心，即Eureka server，另一个是Eureka Client。

1. .开始创建Eureka Server

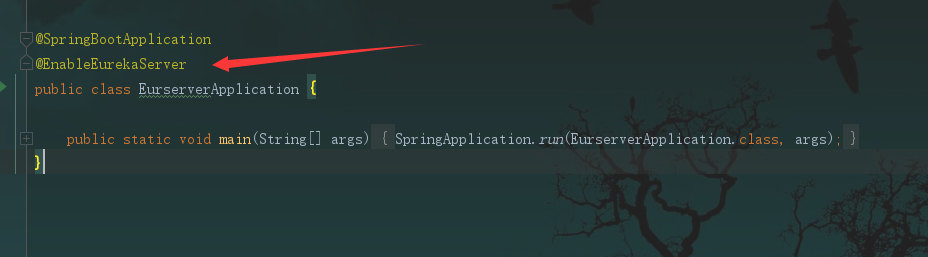
右键工程，new ==> model，然后选择Spring Initializr，next。



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



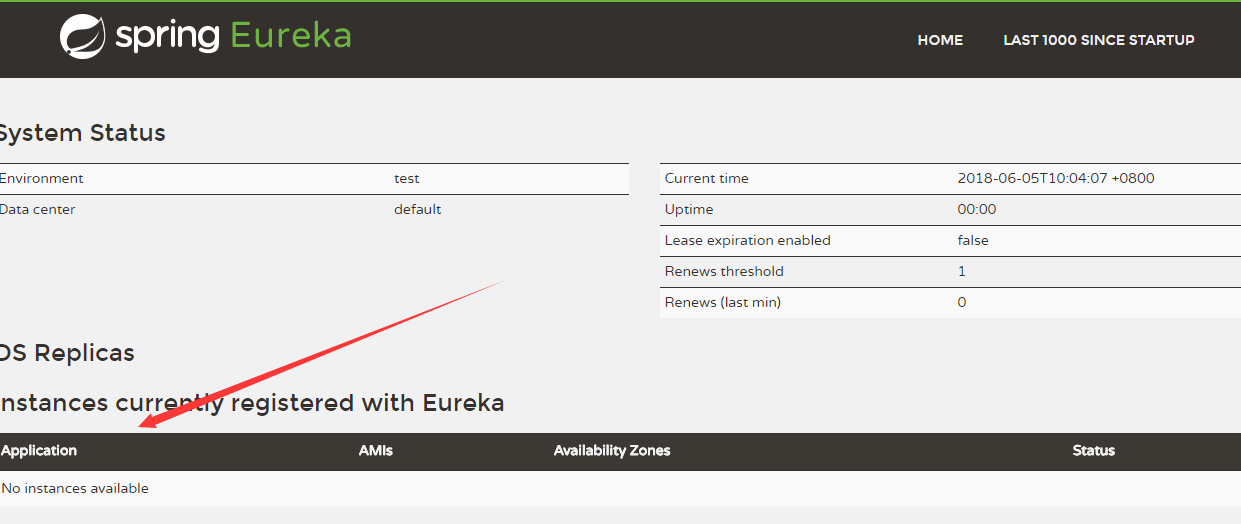
1. .启动一个服务注册中心，主需要一个注解@EnableServer，这个注解放在springboot的启动工程里。



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;  
 }  
}

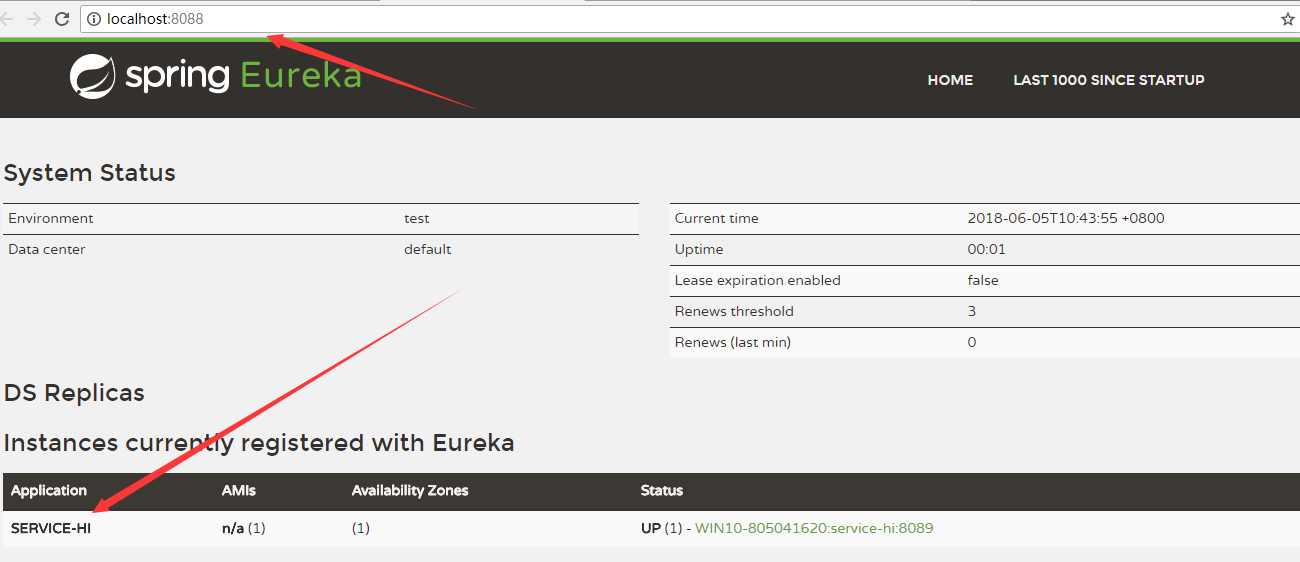
@EnableEurekaClient是Eureka Client的启动注解

@Value("${server.port}")是获取配置文件信息

配置文件：

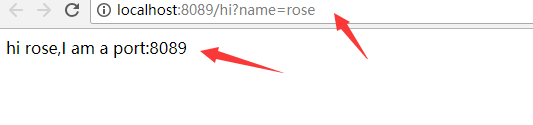
*#注明自己的服务注册中心的地址***eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8088/eureka/  
*#端口号***server:  
 port:** 8089  
*#需要指明spring.application.name,这个很重要，这在以后的服务与服务之间相互调用一般都是根据这个name***spring:  
 application:  
 name:** service-hi

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。

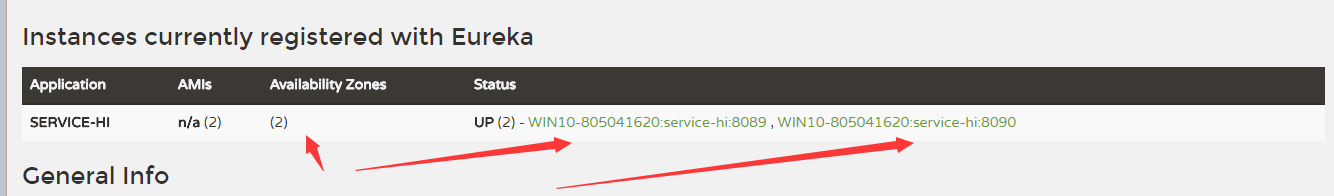
我们访问：localhost:8089/hi?name=rose



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。

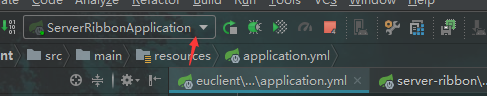
1. Spring cloud有两种服务调用方式，一种是ribbon+restTemplate，另一种是feign。
2. .ribbon是一个负载均衡客户端，可以很好的控制htt和tcp的一些行为。Feign默认集成了ribbon。
3. .前期准备工作：启动euserver，再启动client，然后把euclient,再把euclient配置文件里的port改为8090，重启:

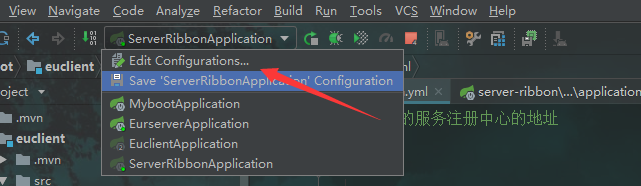
这时候访问euserver:locahost:8088

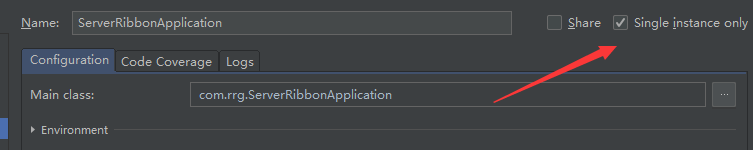


会发现，eureka-server注册了2个实例，这就相当于一个小的集群。

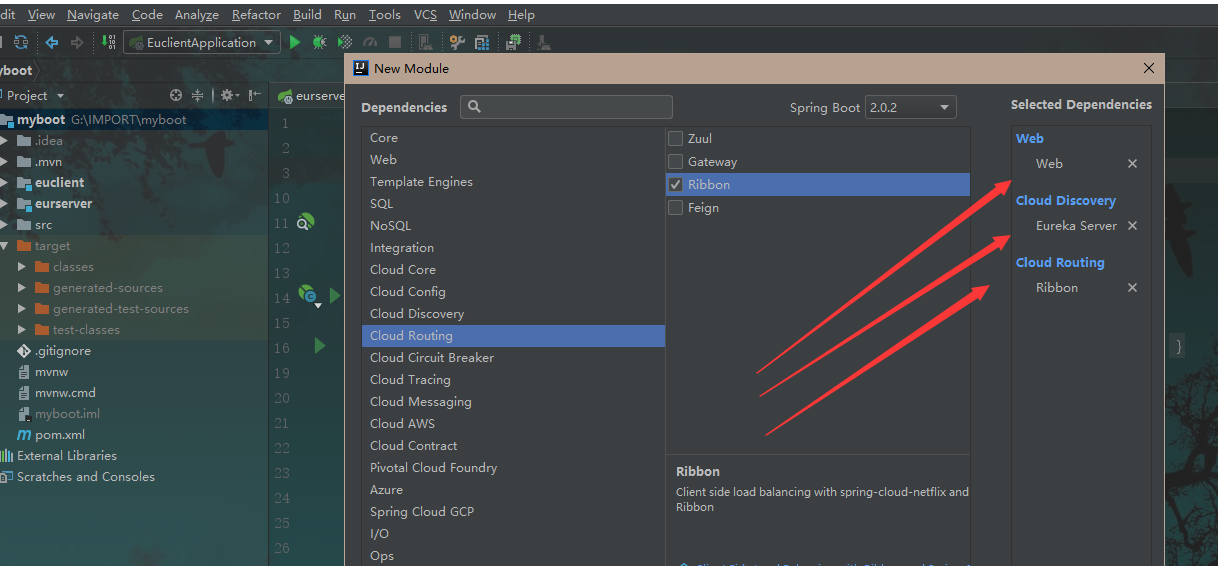
**这里，要设置一个工程启动多个实例：去掉勾。**







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();

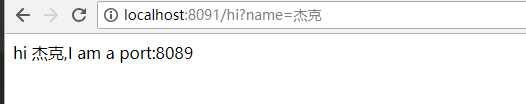
写一个测试类:

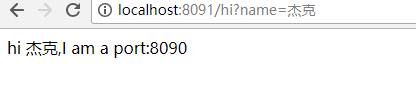
public class TestService {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 public String hiService(String name){  
 return restTemplate.getForObject("http://service-hi/hi?name=" + name,String.class);  
 }  
  
}

@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：

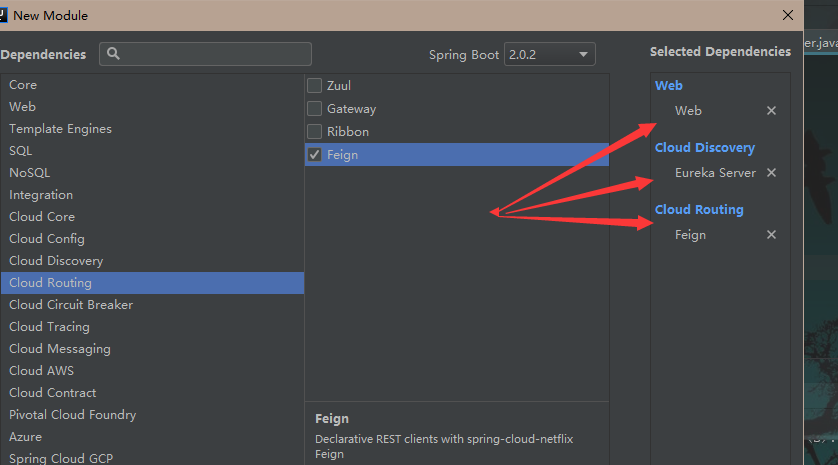
Feign 采用的是基于接口的注解

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：

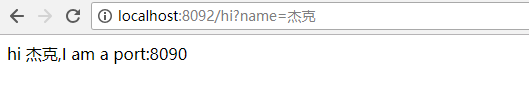
// 通过@ FeignClient（“服务名”），来指定调用哪个服务。  
@FeignClient("service-hi")  
public interface TestService {  
  
 @RequestMapping("/hi")  
 String sayHi(@RequestParam String name);  
}

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

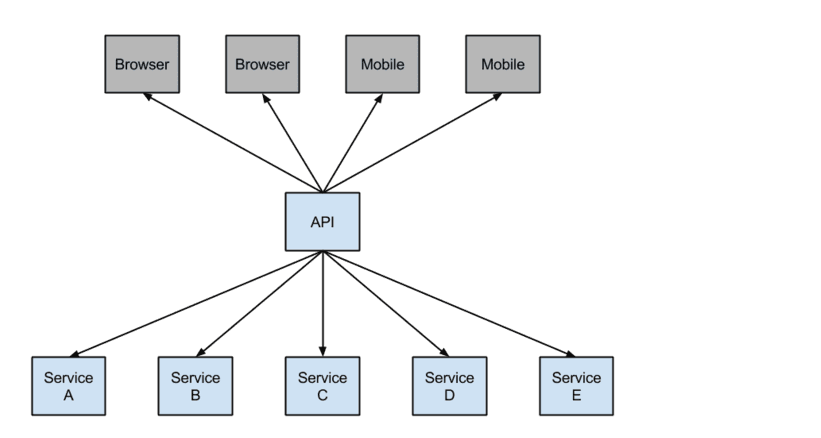
·Request交给Client去处理，其中Client可以是HttpUrlConnection、HttpClient也可以是Okhttp。

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

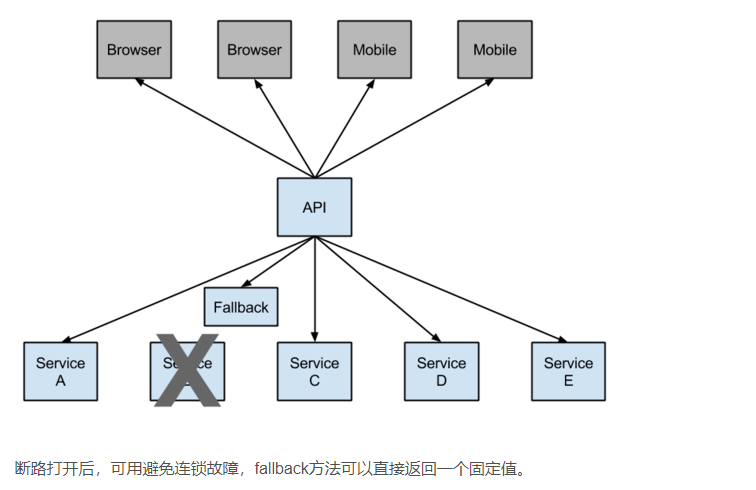
7. 在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以相互调用（RPC），在Spring Cloud可以用RestTemplate+Ribbon和Feign来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。

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 {

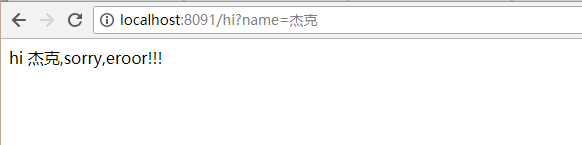
改造service方法：在方法上面加入@HystrixCommand(fallbackMethod = "hiError")注解，hiError是下面方法的名字，遇到错误就返回一个固定值。

@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：

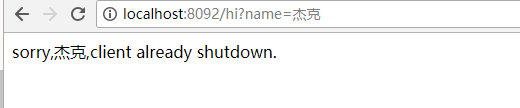
*#打开断路器***feign:  
 hystrix:  
 enabled:** true

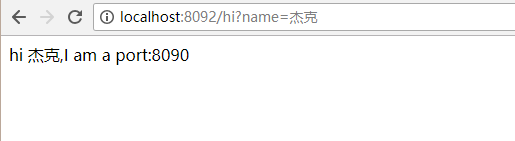
然后在service接口里加上代码：fallback = SchedualServiceHiHystric.class，SchedualServiceHiHystric是实现TestService的类。

// 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：

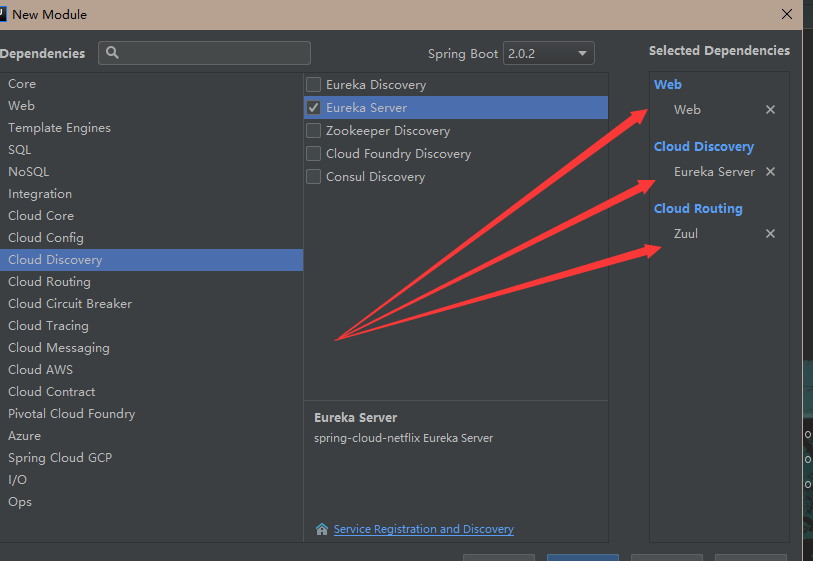
@EnableHystrixDashboard  
@EnableCircuitBreaker

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

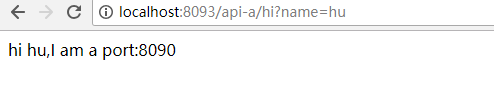
首先指定服务注册中心的地址为http://localhost:8088/eureka/，服务的端口为8092，服务名为service-zuul；以/api-a/ 开头的请求都转发给service-ribbon服务；以/api-b/开头的请求都转发给service-feign服务。

在启动类加上注解：

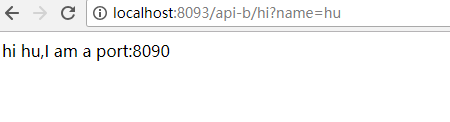
@EnableEurekaClient  
@EnableZuulProxy

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！

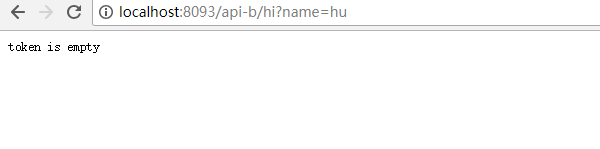
1. .服务过滤，zuul不仅起到路由，还有过滤，可做一些安全验证。

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;  
 }  
}

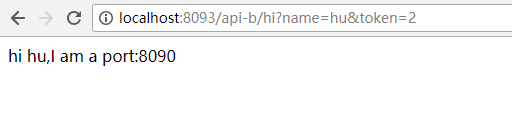
然后再浏览器访问：

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



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

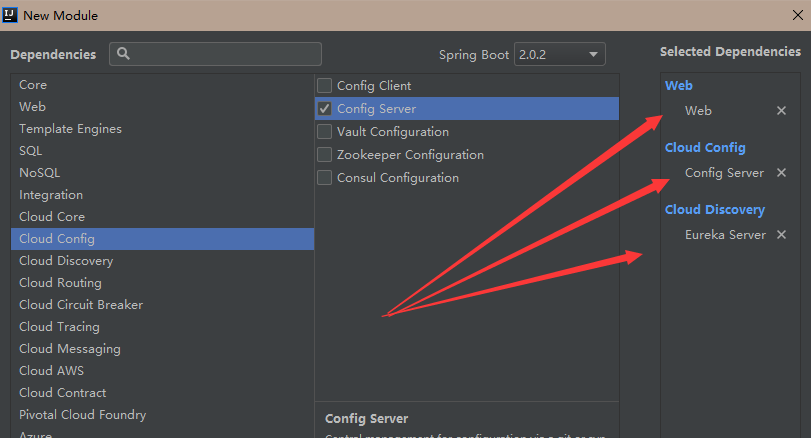
有数据。



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



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=

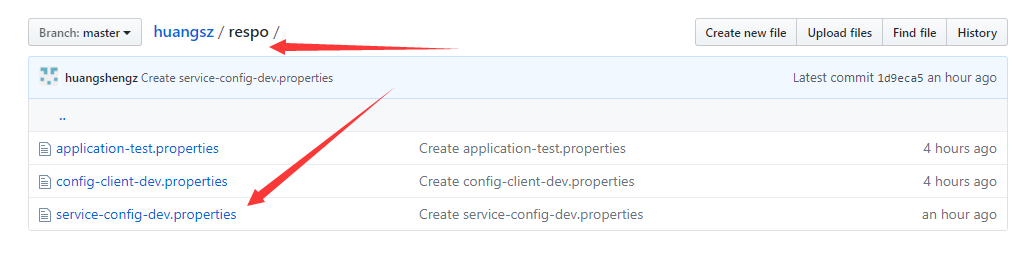
·spring.cloud.config.server.git.uri是github地址

·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

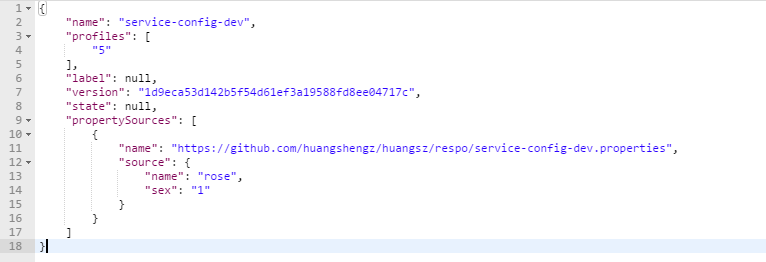
·spring.cloud.config.server.git.password密码，如果git是public的话不用填写

1. .

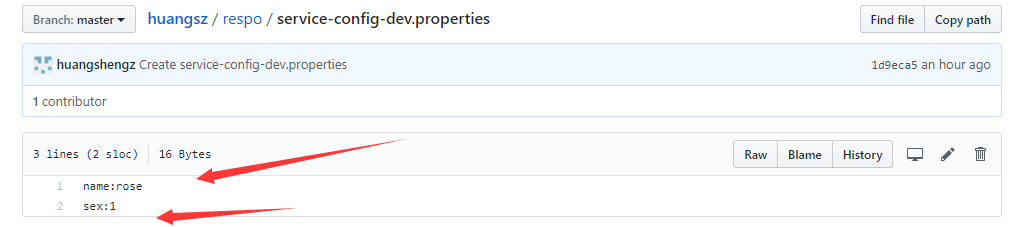
·respo就是search-paths路径，而git路径是直至仓库名的

·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：

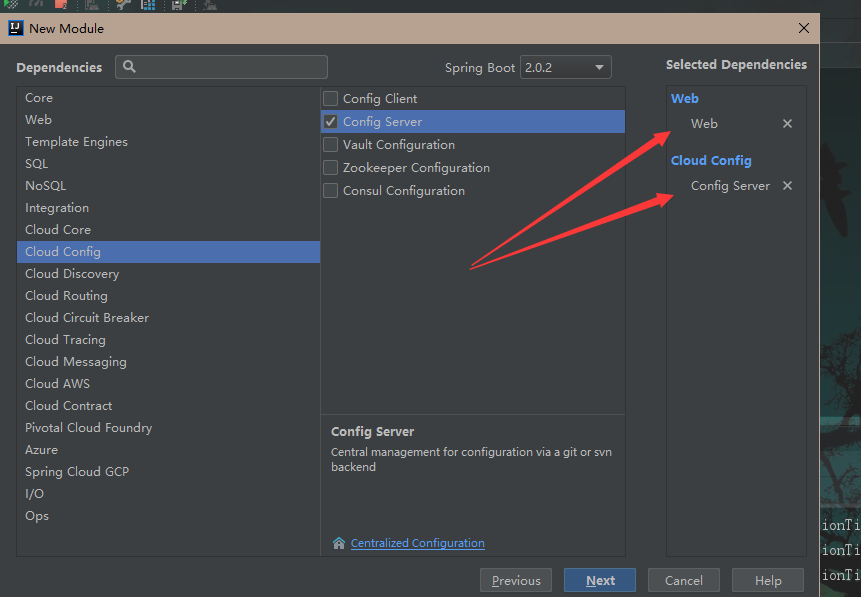


以下是git文件里的内容：



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:



Pom，把<spring-cloud.version>Finchley.RC2</spring-cloud.version>有个bug，需要把RC2改为RC1。

配置文件：bootstrap.propertes