# 微服务（Microservice）

## 什么是微服务（Microservice）

微服务英文名称Microservice，Microservice架构模式就是将整个Web应用组织为一系列小的Web服务。这些小的Web服务可以独立地编译及部署，并通过各自暴露的API接口相互通讯。它们彼此相互协作，作为一个整体为用户提供功能，却可以独立地进行扩。

## 微服务架构需要的功能或使用场景

1：我们把整个系统根据业务拆分成几个子系统。

2：每个子系统可以部署多个应用，多个应用之间使用负载均衡。

3：需要一个服务注册中心，所有的服务都在注册中心注册，负载均衡也是通过在注册中心注册的服务来使用一定策略来实现。

4：所有的客户端都通过同一个网关地址访问后台的服务，通过路由配置，网关来判断一个URL请求由哪个服务处理。请求转发到服务上的时候也使用负载均衡。

5：服务之间有时候也需要相互访问。例如有一个用户模块，其他服务在处理一些业务的时候，要获取用户服务的用户数据。

6：需要一个断路器，及时处理服务调用时的超时和错误，防止由于其中一个服务的问题而导致整体系统的瘫痪。

7：还需要一个监控功能，监控每个服务调用花费的时间等。

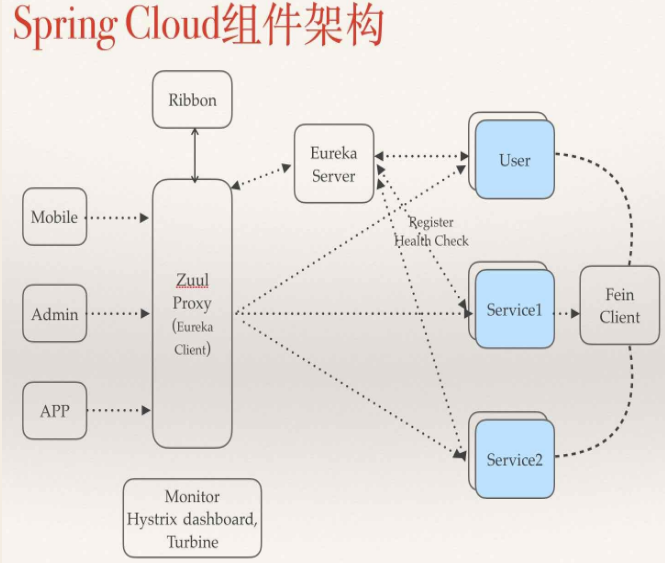
目前主流的微服务框架：Dubbo、 SpringCloud、thrift、Hessian等，目前国内的中小企业用的大多数都是Dubbo，SpringCloud估计很少，也许有些开发同学都没听说过。

# SpringCloud

## 简介

springCloud是基于SpringBoot的一整套实现微服务的框架。他提供了微服务开发所需的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等组件。

## 组件架构



spring cloud子项目包括：

Spring Cloud Config：配置管理开发工具包，可以让你把配置放到远程服务器，目前支持本地存储、Git以及Subversion。

Spring Cloud Bus：事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

Spring Cloud Netflix：针对多种Netflix组件提供的开发工具包，其中包括Eureka、Hystrix、Zuul、Archaius等。

Netflix Eureka：云端负载均衡，一个基于 REST 的服务，用于定位服务，以实现云端的负载均衡和中间层服务器的故障转移。

Netflix Hystrix：容错管理工具，旨在通过控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

Netflix Zuul：边缘服务工具，是提供动态路由，监控，弹性，安全等的边缘服务。

Netflix Archaius：配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

Spring Cloud for Cloud Foundry：通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。

Spring Cloud Sleuth：日志收集工具包，封装了Dapper,Zipkin和HTrace操作。

Spring Cloud Data Flow：大数据操作工具，通过命令行方式操作数据流。

Spring Cloud Security：安全工具包，为你的应用程序添加安全控制，主要是指OAuth2。

Spring Cloud Consul：封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

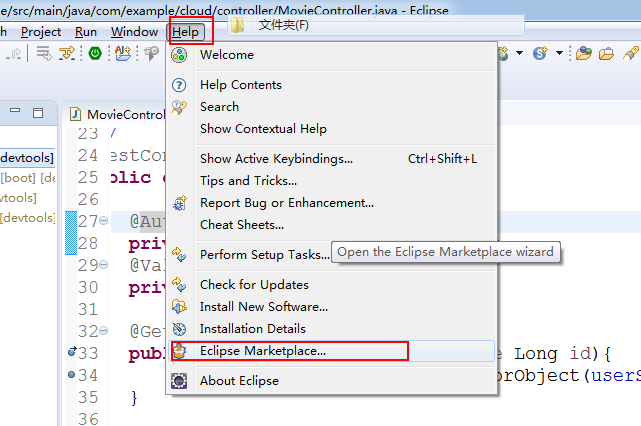
Spring Cloud Zookeeper：操作Zookeeper的工具包，用于使用zookeeper方式的服务注册和发现。

Spring Cloud Stream：数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

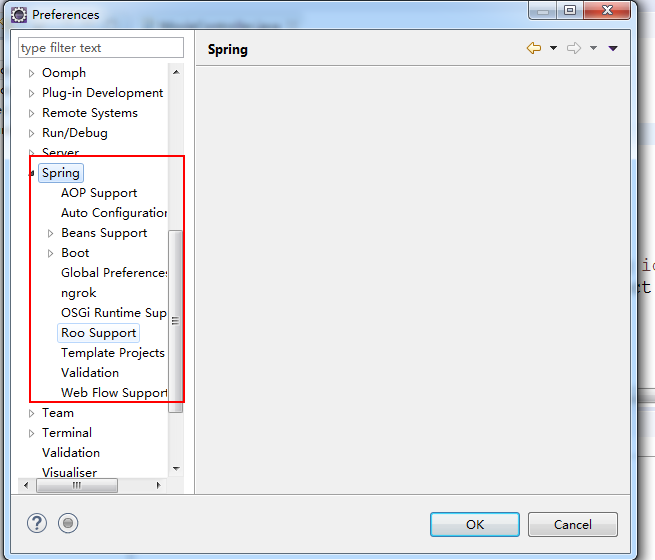
Spring Cloud CLI：基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。

## 准备工作

**将eclipse集成spring tool suite方便开发，或者直接用spring tool suite的开发工具也可以。**



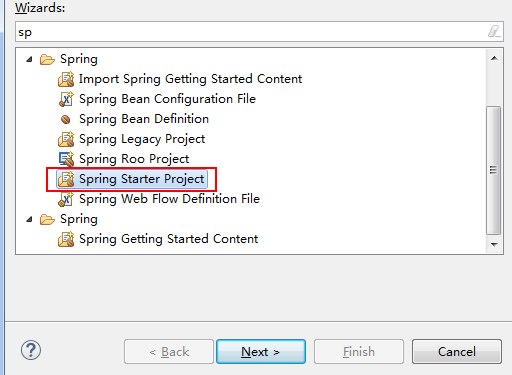
搜索 install即可，查看是否安装成功，出现下图的spring即安装成功

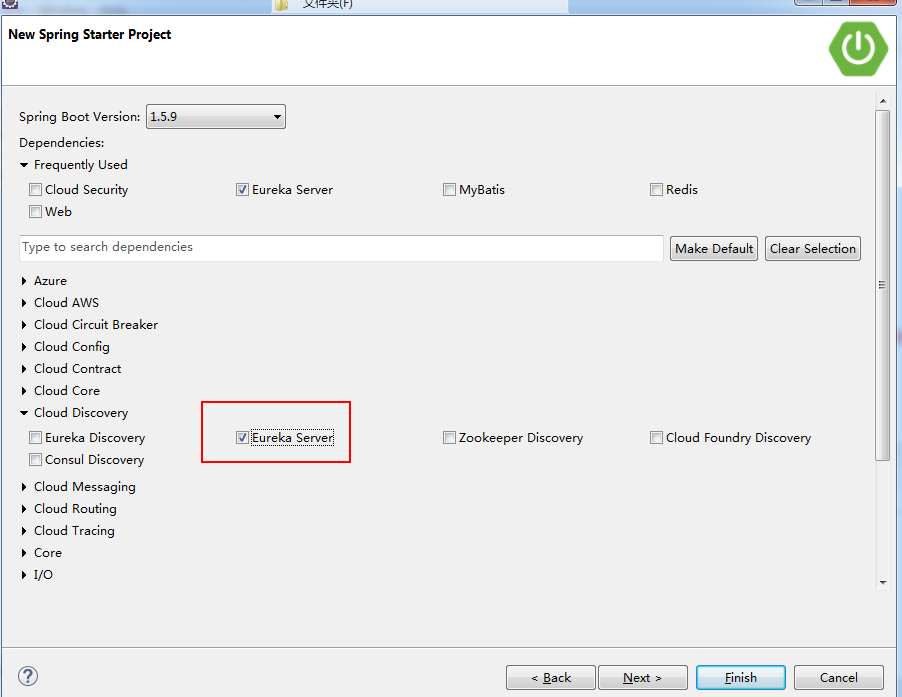


## Eureka

### 1.创建服务注册中心

我们需要用的组件是Spring Cloud Netflix的Eureka ,eureka是一个服务注册和发现模块。



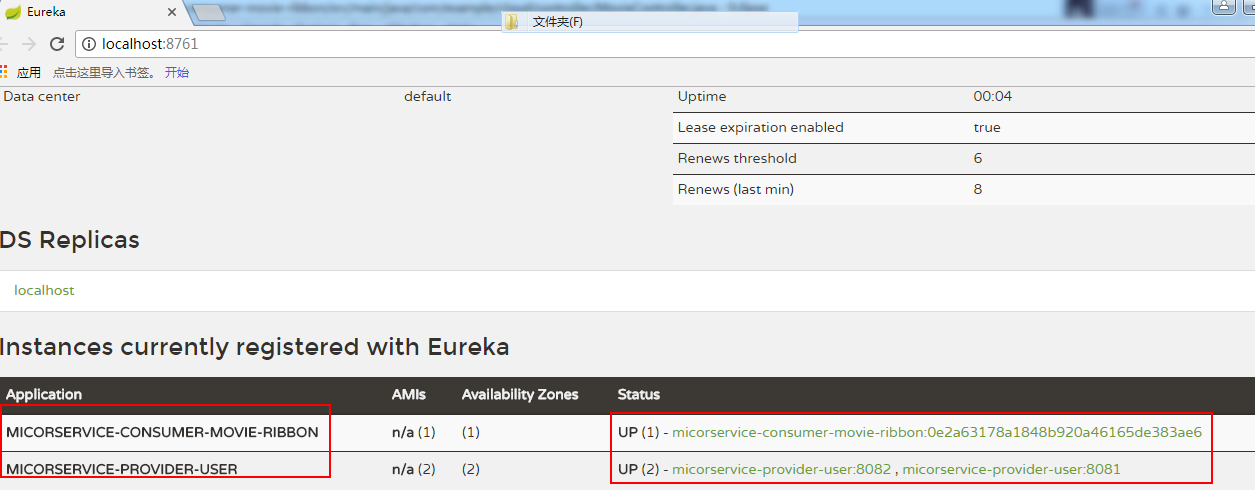


其余默认即可。具体实现和注释参照 microservice-discovery-eureka 代码，本文中所有代码都在 springcloud-Code 中，其中有使用到maven的一些项目管理的知识点，具体自行百度。

### 2.创建服务提供者和消费者

创建方式和上面一样，具体依赖选择可以多选，具体实现请参照micorservice-provider-user（提供者） 和 micorservice-consumer-movie（消费者）

访问:http://localhost:8761/ up表示服务可用



### 3.高可用的eureka

3.1 介绍

 提高系统的可靠性，我们使用Eureka集群，搭建的过程很简单，在Eureka工程的配置文件application.yml中进行配置。

3.2 实施

具体参考代码： microservice-discovery-eurekamoreserver

(1) 使Eureka之间相互注册。

(2) 用户微服务，注册到任意一个Eureka Server上都会进行同步。

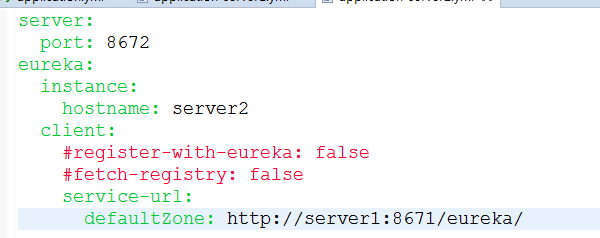
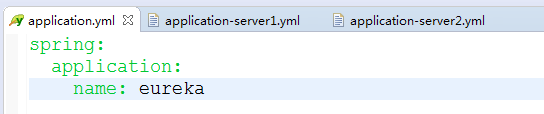
在网上大多实现方式：

1.一个 application.yml 配置文件 通过 ‘---’ 分割 配置多个server，然后通过 active 来指定启动哪个server，亲测无效。

2.分成三个application.yml application-server1.yml application-server2.yml 然后通过 application.yml 中的active 来指定启动，亲测无效。

正确的姿势：

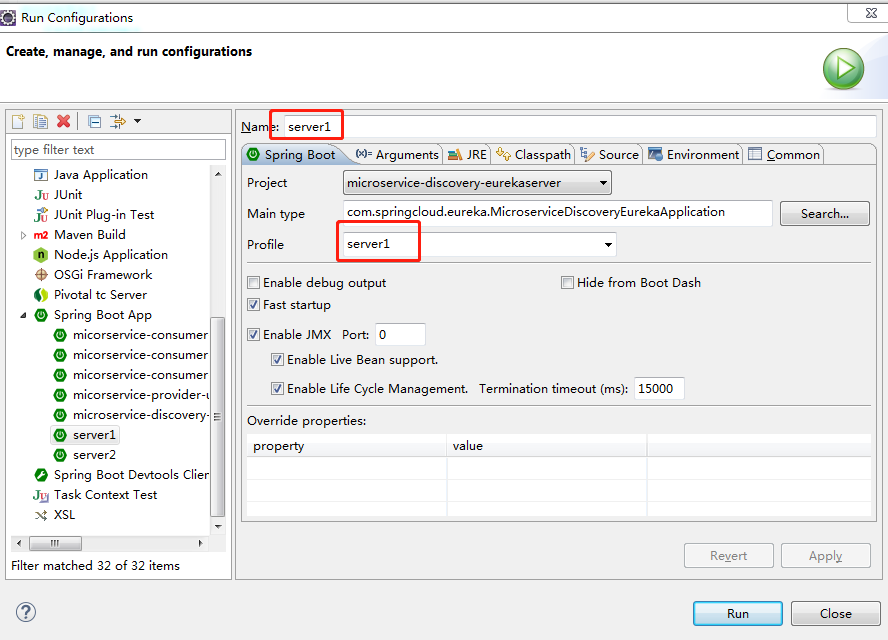
1.三个配置文件： application.yml application-server1.yml application-server2.yml



2.修改hosts 位置在 C:\Windows\System32\drivers\etc 下

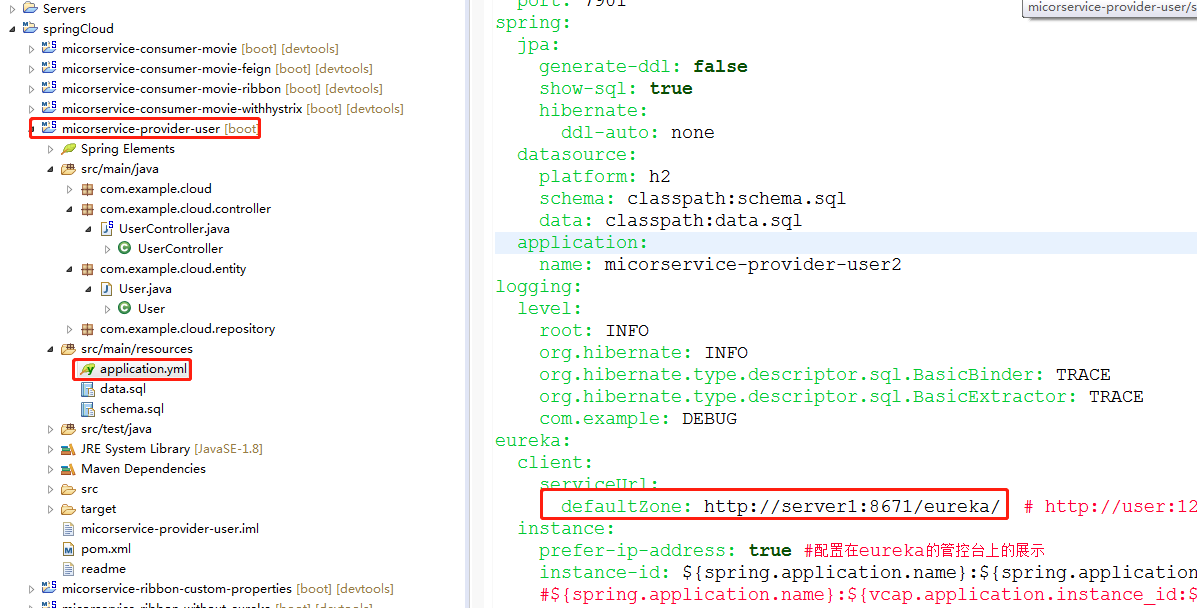


3.配置两个启动项



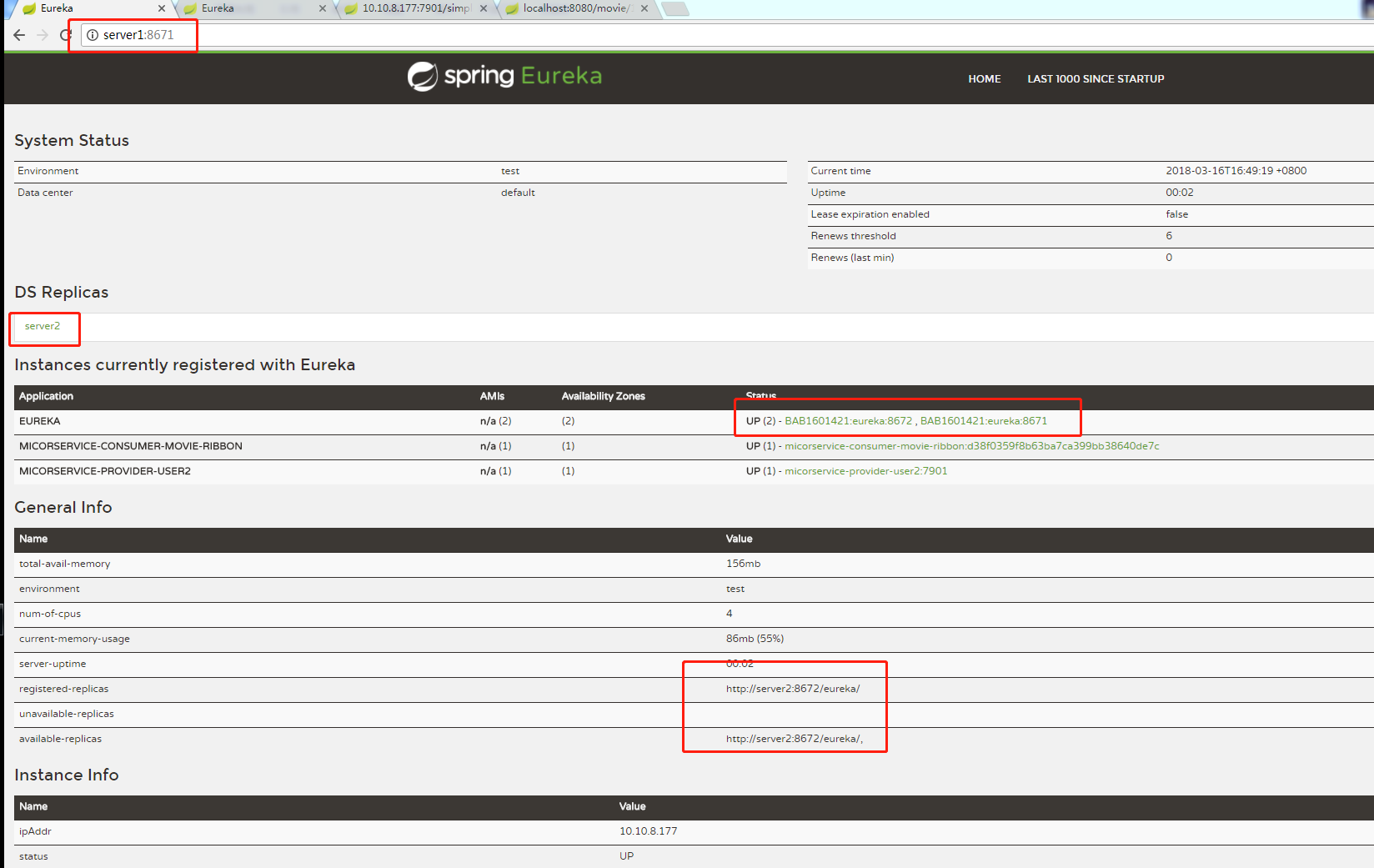
4.启动client

defaultZone 只配置一个即可，网上大多都全部配置了，这样不符合高可用的要求。



5.结果

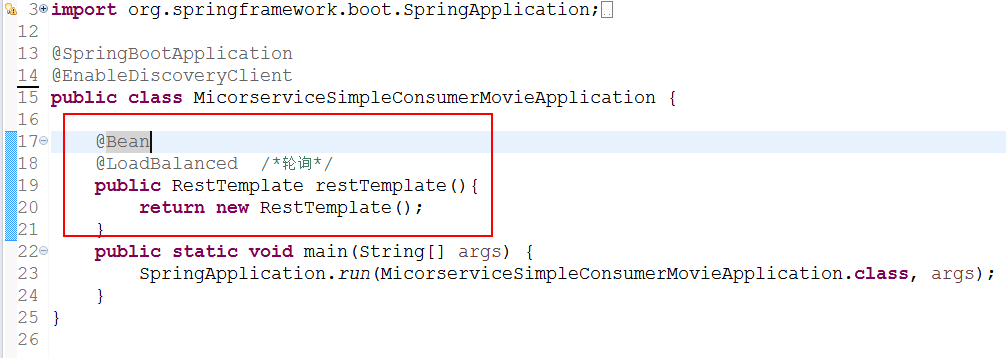
分别访问 <http://server1:8671/> 和 [http://server2:8672/](http://server1:8671/) 可以看到服务已经全部注册。可以停了server1 进行测试，发现服务是可用的。

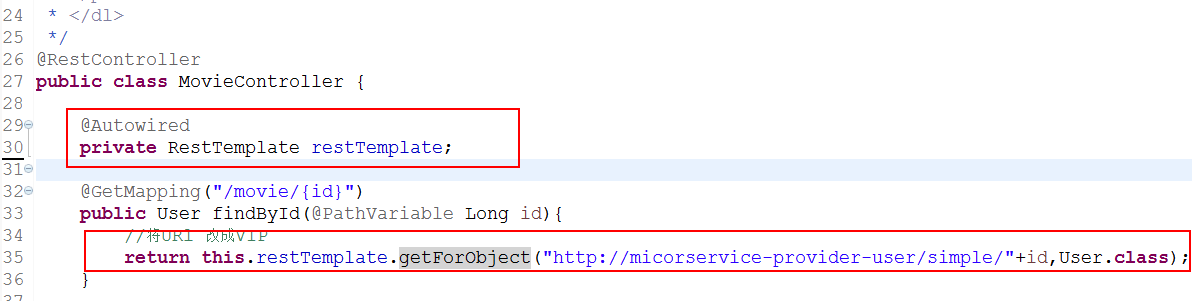


## Ribbon

### 1.ribbon的简单使用

具体参考代码 ：micorservice-consumer-movie-ribbon

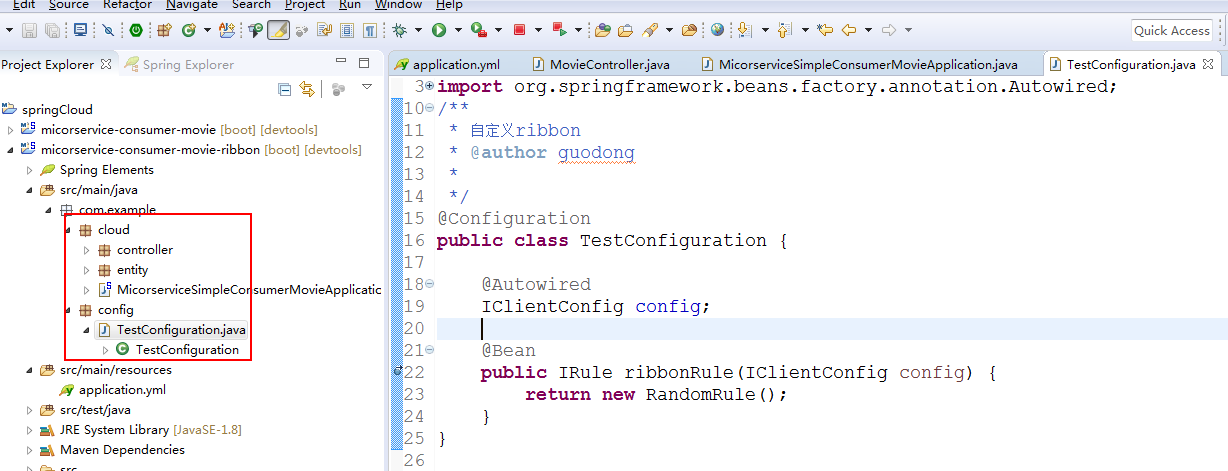




### 2.自定义ribbon

2.1 代码方式的实现

具体参考代码 ：micorservice-consumer-movie-ribbon





注意包结构，TestConfiguration类不能放在启动扫描到的包中,也可以通过注解配置来去除这种障碍,在此不做展示，一般使用下面配置文件的方式实现。

2.2 配置文件的实现

具体参考代码 ：micorservice-ribbon-custom-properties

**配置文件的方式相对于代码的方式优先级高**



只需在配置文件中添加服务和路由规则即可。

## Feign

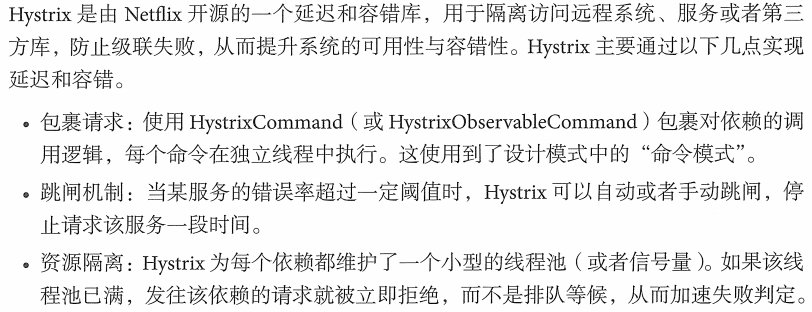
### Feign的简单使用

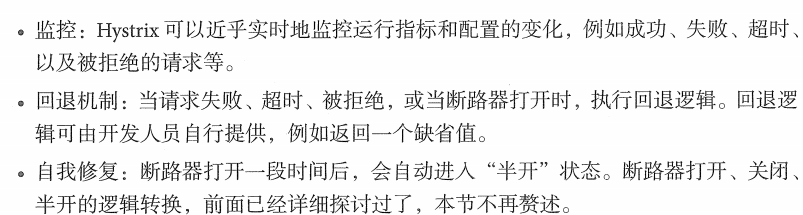
具体参考代码 ： micorservice-consumer-movie-feign

## Hystrix

### 1.介绍

在微服务框架中，一个服务消费者可能是其他服务消费者的提供者，而当低层次的服务提供者出现问题时，会导致系统资源被耗尽。出现雪崩效应。Hystrix就是解决这个问题的。





### 2.ribbon 对 hystrix 的支持

具体参考代码 ：micorservice-consumer-movie-ribbon-hystrix

##### 1.添加依赖

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

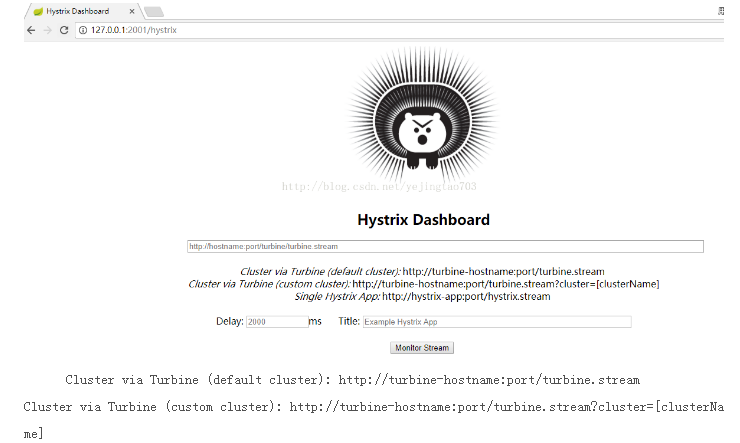
##### 2.启动类上添加 @EnableCircuitBreaker或@EnableHystrix 注解表示开启对hystrix的支持。

##### 3.代码实现

|  |
| --- |
| @RestController  public class MovieController {  @Autowired  private RestTemplate restTemplate;    @HystrixCommand(fallbackMethod = "findByIdFallBack")  @GetMapping("/movie/{id}")  public User findById(@PathVariable Long id){  //将URl 改成VIP  return this.restTemplate.getForObject("http://micorservice-provider-user/simple/"+id,User.class);  }    /\*\*  \* 熔断回调  \* @param id  \* @return User  \* guodong  \*/  public User findByIdFallBack(Long id){  User user = new User();  user.setId(1L);  user.setName("默认用户");  return user;  } |

说明：可以将服务启动可以正常访问后再断开服务测试。

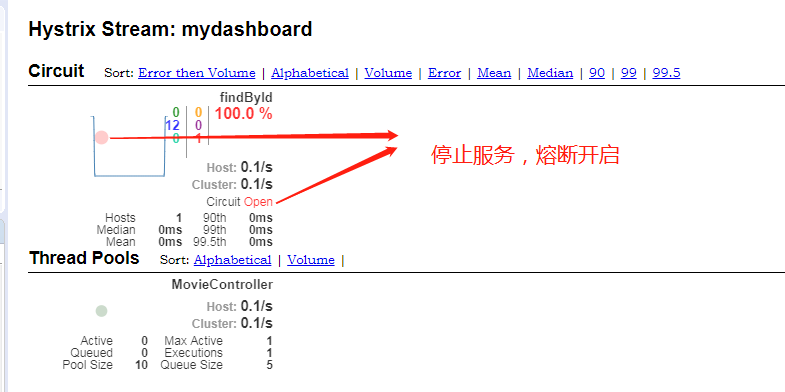
访问 <http://10.10.8.177:8080/hystrix>



然后监控 <http://10.10.8.177:8080/hystrix>.stream 心跳和名称随意，可以看到如下图。



将提供者停止进行测试。刚停止后发现熔断并未开启，因为此时还没有达到熔断的阈值（默认是 5秒20次），这时需要我们快速的请求，然后达到这个阈值。



### 3.feign 对 hystrix 的支持

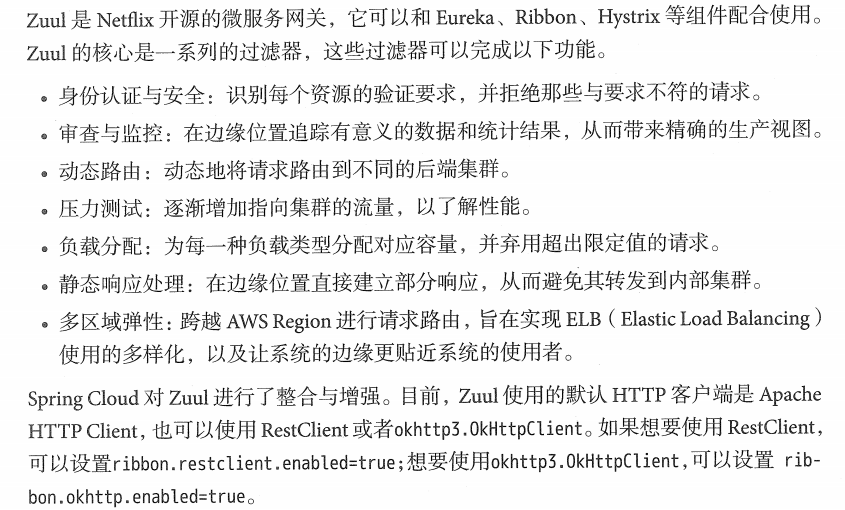
具体参考代码 ：micorservice-consumer-movie-feignwithhystrix

注意：低版本的需要在配置文件中开启feign对hystrix的支持！！！

feign.hystrix.enabled: true #开启hystrix对feign的支持.

## Zuul

### 1.简介



### 2.简单的服务网关实现

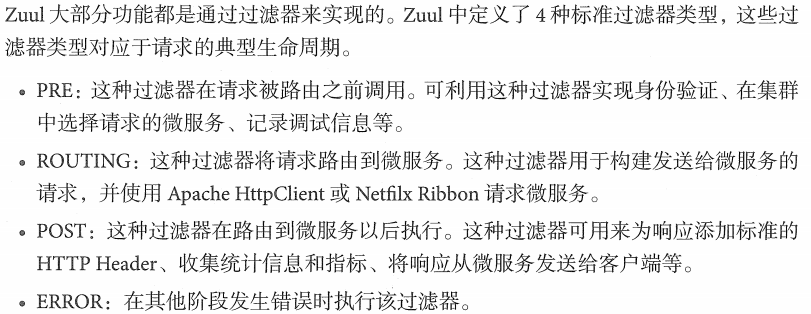
具体参考代码： microservice-gateway-zuul

|  |
| --- |
| 1.添加依赖  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId> </dependency>   1. 启动类增加注解   @EnableZuulProxy // 声明一个zuul代理   1. 修改配置文件      1. 启动项目   microservice-discovery-eureka  micorservice-provider-user  micorservice-consumer-movie-ribbon  microservice-gateway-zuul   1. 首先确保项目可以正常访问，然后可以通过zuul 代理访问，访问方式：[http://zuul\_host:zuul\_port/微服务注册在eureka上的serviceId/\*\*](http://zuul_host:zuul_port/微服务注册在eureka上的serviceId/**)   Eg: localhost:8040/micorservice-provider-user/simple/2,会发现zuul 可以代理eureka上面注册的所有服务 |

说明： 负载均衡（默认是轮询）和熔断机制测试在此不做说明，可自行测试。

### 3.zuul的过滤器

#### 3.1 zuul 过滤器简介



#### 3.2 简单实现

具体参考代码： microservice-gateway-zuul-filter

### 4.zuul 实现容错回退

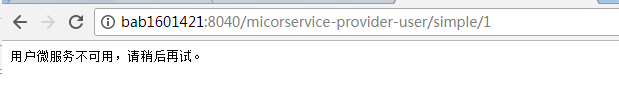
Zuul中默认整合了hystrix，可以直接访问测试。

#### 4.1 配置回退方法

|  |
| --- |
| @Component public class UserFallbackProvider implements ZuulFallbackProvider{  @Override  public String getRoute() {  // 表明为哪个服务提供回退  return "micorservice-provider-user";  }  @Override  public ClientHttpResponse fallbackResponse() {  return new ClientHttpResponse() {  @Override  public HttpStatus getStatusCode() throws IOException {  // fallback 时的状态码  return HttpStatus.OK;  }  @Override  public int getRawStatusCode() throws IOException {  // 状态码 200  return this.getStatusCode().value();  }   @Override  public String getStatusText() throws IOException {  // 状态文本，本例中返回 ok  return this.getStatusCode().getReasonPhrase();  }   @Override  public void close() {   }   @Override  public InputStream getBody() throws IOException {  return new ByteArrayInputStream("用户微服务不可用，请稍后再试。".getBytes());  }   @Override  public HttpHeaders getHeaders() {  // 设定 headers  HttpHeaders headers = new HttpHeaders();  MediaType mediaType = new MediaType("application", "json", Charset.forName("UTF-8"));  headers.setContentType(mediaType);  return headers;  }  };  } } |

#### 4.2 测试

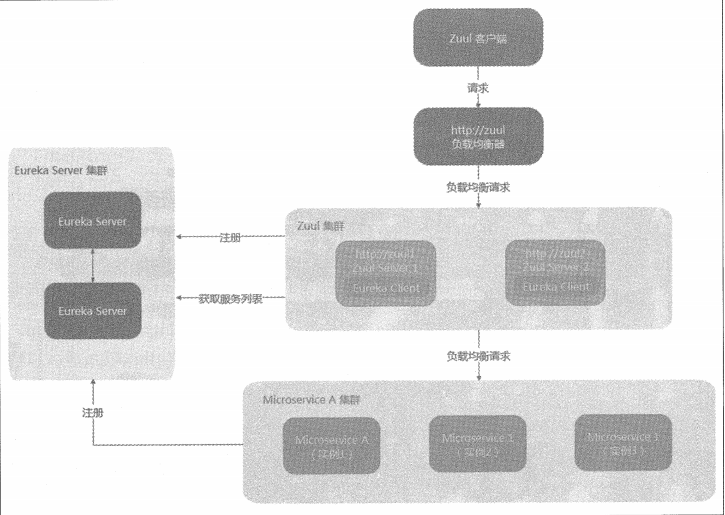
启动服务可正常访问后，停止生产者的服务，用zuul 代理访问



### 5.zuul 的高可用

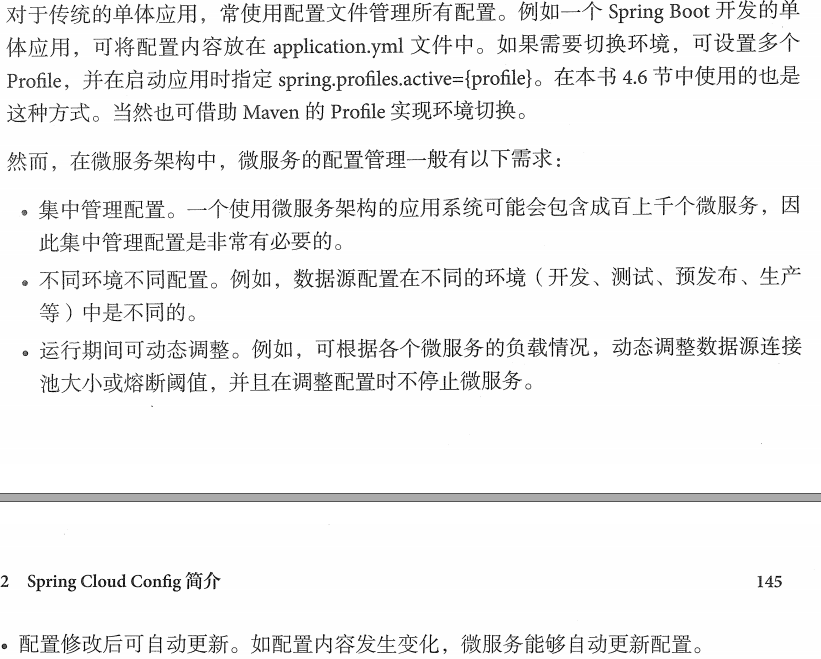
#### 1.注册在eureka上面 zuul_eureka

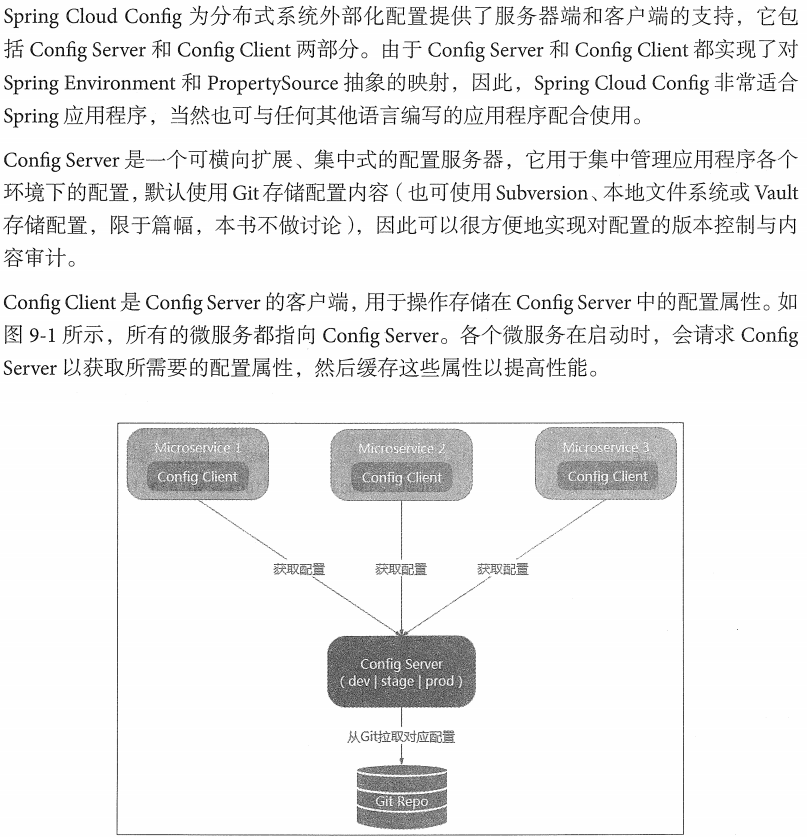
#### 2.利用haproxy、Nginx、f5 等做负载均衡器来实现



## Config

### 1.简介





### 2.简单实现

#### 2.1 服务端

具体实现参考： microservice-config-server

|  |
| --- |
| 1.在git上创建测试文件      2.添加依赖  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId> </dependency>  3.启动类添加注释 @EnableConfigServer  4.配置application.yml  server:  port: 8080 spring:  application:  name: microservice-config-server  cloud:  config:  server:  git:  uri: https://github.com/Monkey-D-GD/springcloudconfig #仓库地址  username: guodong260007@163.com #git 用户名  password: #密码  5.访问 http://localhost:8080/microservice-foo-dev.properties  http://localhost:8080/config-label-v2.0/microservice-foo-dev.properties |

#### 2.2 客户端

具体参考代码： micorservice-config-client

|  |
| --- |
| 1.添加依赖  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId> </dependency>  2.修改配置文件  添加一个bootstrap.yml 文件（ps: bootstrap.yml 比application.yml优先级高 ）  spring:  application:  name: microservice-foo #与git里的文件名对应  cloud:  config:  uri: http://localhost:8080/ # 指明配置服务中心的网址  # dev 开发环境配置文件 | test 测试环境 | pro 正式环境  # 和git里的文件名对应  profile: production  label: config-label-v2.0 # 指定Git仓库的分支  3.新增controller  @RestController public class ConfigClientController {   // git配置文件里的key  @Value("${profole}")  private String profole;  @GetMapping("/profile")  public String hello() {  return this.profole;  } }   1. 访问controller即可 |

### 3.客户端手动刷新配置

具体实现参考: micorservice-config-client

当配置文件修改并提交至git时，项目访问返回值仍然为原来的值，此时可以通过手动刷新来实现。

|  |
| --- |
| 1.确保client端有actuator包的依赖  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId> </dependency>  2.在controller上添加手动刷新注解 @RefreshScope  3.在application.yml 中关闭验证  4.启动代码访问，然后修改git仓库的配置文件，然后执行;  curl -X POST http://localhost:8081/refresh  当返回profile时表示成功然后再次访问配置文件就可看到最新的配置。 |

### 4.集成bus 实现客户端的自动刷新

具体实现参考： micorservice-config-client-refresh-bus

|  |
| --- |
| 1.集成bus\_amqp jar包  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId> </dependency> 2.在配置文件中配置rabbitmq的信息  rabbitmq:  host: localhost  port: 5672  username: guest  password: guest 3.启动多个实例 4.修改后git仓库中的配置文件后执行  curl -X POST <http://localhost:8081/bus/refresh> 发现 8082的配置文件也更新了 5.也可以利用 destination 参数来实现局部刷新  eg: 仅刷新8081的配置文件 curl -X POST http://localhost:8081/bus/refresh?destination=microservice-foo:8081  如果命令是 curl -X POST http://localhost:8081/bus/refresh?destination=microservice-foo:8082  则两个都会刷新 6.执行命令的实质是更新了本地库中的配置文件，在第一次请求时会去Git仓库中进行配置文件的下载，然后便不在请求git仓库，只有显示的调用才会再次请求更新本地文件。 |

### 5.服务端实现自动刷新

由于使用客户端微服务来实现配置文件的刷新违背的微服务的单一原则，所以将配置文件交由config\_server来完成。

具体实现参考代码; microservice-config-server

|  |
| --- |
| 全部刷新： curl -X POST http://localhost:8080/bus/refresh 刷新 microservice-foo 下的所有配置 curl -X POST http://localhost:8080/bus/refresh?destination=microservice-foo:\*\* 仅刷新8081: curl -X POST http://localhost:8080/bus/refresh?destination=microservice-foo:8081 |

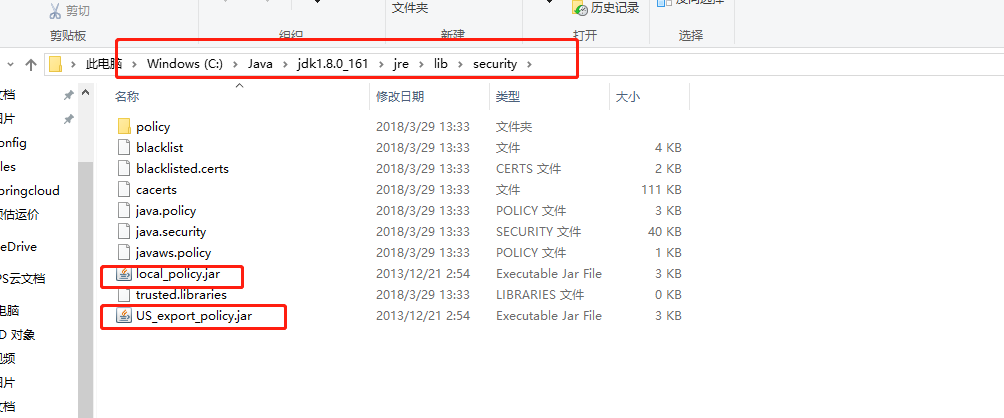
### 6.加密解密

在使用Spring Cloud Config的加密解密功能时，有一个必要的前提需要我们注意。为了启用该功能，我们需要在配置中心的运行环境中安装不限长度的JCE版本（Unlimited Strength Java Cryptography Extension）。虽然，JCE功能在JRE中自带，但是默认使用的是有长度限制的版本。我们可以从Oracle的官方网站中下载到它，它是一个压缩包，解压后可以看到下面三个文件：

README.txt

local\_policy.jar

US\_export\_policy.jar

我们需要将 local\_policy.jar和 US\_export\_policy.jar两个文件复制到 $JAVA\_HOME/jre/lib/security目录下，覆盖原来的默认内容。到这里，加密解密的准备工作就完成了。

在完成了JCE的安装后，可以尝试启动配置中心。在控制台中，将会输出了一些配置中心特有的端点，主要包括：

/encrypt/status：查看加密功能状态的端点

/key：查看密钥的端点

/encrypt：对请求的body内容进行加密的端点

/decrypt：对请求的body内容进行解密的端点

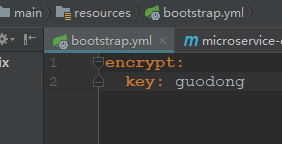
可以尝试通过GET请求访问 /encrypt/status端点，我们将得到如下内容：



该返回说明当前配置中心的加密功能还不能使用，因为没有为加密服务配置对应的密钥。

#### 6.1 对称加密

加密的密钥需要配置在bootstrap.yml中。



加解密如下

|  |
| --- |
| curllocalhost:8080/encrypt -d guodong 50cbda40f052df25b840cddd181bd1fa22a461b0fe8e6edf1ef5653056ff8d50  curl localhost:8080/decrypt -d 50cbda40f052df25b840cddd181bd1fa22a461b0fe8e6edf1ef5653056ff8d50 |

