# SpringCloud

SpringCloud整体核心架构只有一点:Rest服务，也就是说在整个SpringColud配置过程之中，所有的配置处理都是围绕这REST完成的，在整个Rest处理之中，一定要有两端：服务的提供者（Provider）,服务的消费者（Consumer）。

SpringColud微服务=Rest服务

## 2.2创建 dept微服务

所谓的微服务的核心本质就是JSON的传输，那么既然现在要求使用Mybastis进行数据库操作，所以应该在项目里面配置Druid数据库连接池，而后对外进行项目的发布

1.【microcloud-provider-depr-8001】

## 2.3SpringSecurity 安全认证

### 2.3.1Rest提供者安全访问

1.修改8001项目中POM.XML 配置文件，追加SpringSecurity相关依赖包引入：

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency> |

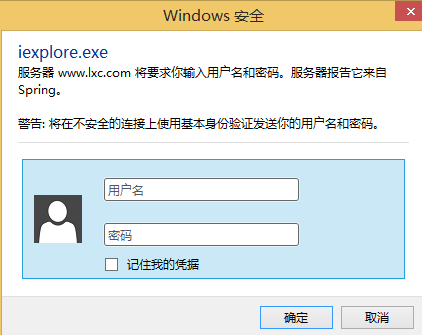
如果你现在配置了安全框架，则在启动时会出现有如下的提示信息：

|  |
| --- |
| Using default security password:  34cfc2a4-a04c-41c7-a588-7abb79f183c2 |

2.修改【8001】项目中的application.yml配置文件，进行安全的用户名配置

|  |
| --- |
| server:  port: 8001  mybatis:  config-location: classpath:mybatis/mybatis-config.xml #mybatis的配置文件路径  type-aliases-package: com.liuxc.microcloud.api #配置vo下对象的别名  mapper-locations:  - classpath:mybatis/mapper/\*\*/\*.xml #扫面mapper文件下的路径  security:  basic:  enabled: **true** #启用Spring Security 安全框架认证  user:  name: liuxc #认证用户名  password: liuxc #认证密码  role: #认证角色授权  - USER  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: com.mysql.jdbc.Driver #配置驱动类  url: jdbc:mysql://localhost:3306/master?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址  username: liuxc #配置数据库连接用户名  password: liuxc #配置数据库连接用户密码  dbcp2:  min-idle: 5 #配置连接池最小连接数  initial-size: 5 #配置连接池初始化连接数  max-total: 12 #配置连接池最大连接数  max-wait-millis: 2000 #配置最大超时断开时间 |

启动项目访问接口将提示输入用户名和密码



当输入密码后正常访问。

还有可以一种简化版的登录访问方式：

|  |
| --- |
| http://liuxc:liuxc@www.lxc.com:8001/dept/findAll |

### 2.3.2配置消费端安全访问

1.【80】修改RestConfig配置类，在这个配置类上追加新的配置

|  |
| --- |
| @Configuration  **public** **class** RestTemplatesConfig {  @Bean  **public** HttpHeaders getHeader() {// 进行 http头信息配置  HttpHeaders header = **new** HttpHeaders();// 定义一个认证头信息  String auth = "liuxc:liuxc"; // 认证原始信息  **byte**[] encode = Base64.*encodeBase64*(auth.getBytes(Charset.*forName*("US-ASCII")));// 进行加密处理  String authHeader = "Basic " + **new** String(encode);  header.set("Authorization", authHeader); // 加入头信息  **return** header;  }  @Bean  **public** RestTemplate getRestTemplate() {  **return** **new** RestTemplate();  }  } |

2.修改Controller调用rest接口方法

|  |
| --- |
| @RestController  @RequestMapping("/consumer")  **public** **class** DeptController {  @Autowired  **private** RestTemplate template;  @Autowired  **private** HttpHeaders headers;//加入请求头  @RequestMapping(value = "/dept/add", method = RequestMethod.***GET***)  **public** Object add(Dept dept) {  Boolean flag = template.exchange("http://www.lxc.com:8001/dept/add", HttpMethod.***GET***, **new** HttpEntity<Object>(dept,headers), Boolean.**class**). getBody();  //Boolean flag = template.postForObject("http://www.lxc.com:8001/dept/add", dept, boolean.class);  **return** flag;  }  @RequestMapping(value = "/dept/findById", method = RequestMethod.***GET***)  **public** Object findById(Long id) {  Dept dept =template.exchange("http://www.lxc.com:8001/dept/findById/" + id, HttpMethod.***GET***, **new** HttpEntity<Object>(headers), Dept.**class**).getBody();  //Dept dept = template.getForObject("http://www.lxc.com:8001/dept/findById/" + id, Dept.class);  **return** dept;  }  @SuppressWarnings("unchecked")  @RequestMapping(value = "/dept/findAll", method = RequestMethod.***GET***)  **public** Object findAll() {  List<Dept> list = template.exchange("http://www.lxc.com:8001/dept/findAll", HttpMethod.***GET***, **new** HttpEntity<Object>(headers), List.**class**).getBody();  //List<Dept> list = template.getForObject("http://liuxc:liuxc@www.lxc.com:8001/dept/findAll", List.class);  **return** list;  }  } |

### 2.3.3无状态Rest配置

微服务有可能同时并发访问几十万人，那么如果所有的session状态都被维护着?

1. 在有一些的SpringCloud的版本之中，默认是会保存有Session状态的，而后如果用户有需要则可以根据“SessionCreationPolicy”枚举类进行不同的Session状态配置，但是从整体的操作来说，session最好设置成无状态。

\*以下为保持Session的状态（服务器内存有可能被占满）：

|  |
| --- |
| security:  sessions: ALWAYS #一直保存着 |

\*以下为无状态的session设置（服务器不保存Session状态，每一次连接都是一个新的用户）

|  |
| --- |
| security:  sessions: stateless #无状态 |

### 2.3.4定义安全配置模块

在进行Rest服务开始时候，为了保证安全所有的程序里面都需要进行SpringSecuity认证处理都是在application.yml配置文件完成的，这样的配置明显是非常不符合逻辑的，因为此时开发的微服务有很多这些微服务都要求使用统一的用户名和密码的时候就非常不方便了。所以现在最简单的做法是进行统一的设置。

1. 创建一个【microcloud-secuity】模块;修改POM.XML

|  |
| --- |
| 1. <dependencies> 2. <!--spring-boot 依赖 --> 3. <dependency> 4. <groupId>org.springframework.boot</groupId> 5. <artifactId>spring-boot-starter-web</artifactId> 6. </dependency> 7. <!--rest 安全框架 --> 8. <dependency> 9. <groupId>org.springframework.boot</groupId> 10. <artifactId>spring-boot-starter-security</artifactId> 11. </dependency> 12. </dependencies> |

1. 创建Security配置类

|  |
| --- |
| 1. **package** com.liuxc.microcloud.config; 2. **import** org.springframework.beans.factory.annotation.Autowired; 3. **import** org.springframework.context.annotation.Configuration; 4. **import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder; 5. **import** org.springframework.security.config.annotation.web.builders.HttpSecurity; 6. **import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity; 7. **import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter; 8. **import** org.springframework.security.config.http.SessionCreationPolicy; 9. @Configuration 10. @EnableWebSecurity 11. **public** **class** WebSecuityConfig **extends** WebSecurityConfigurerAdapter { 12. @Autowired 13. **public** **void** configGlobal(AuthenticationManagerBuilder builder) **throws** Exception { 14. // 配置security 认证用户 15. builder.inMemoryAuthentication().withUser("liuxc").password("liuxc").roles("USER").and().withUser("admin") 16. .password("root").roles("USER", "ADMIN"); 17. } 18. @Override 19. **protected** **void** configure(HttpSecurity http) **throws** Exception { 20. // 所有的请求都需要通过认证后才可以正常进行 21. http.httpBasic().and().authorizeRequests().anyRequest().fullyAuthenticated(); 22. // 所有rest的服务都需要设置session无状态模式提高效率 23. http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***); 24. } 25. } |

3.删除【8001】项目中的application.yml中的security配置信息、在【8001】模块中pom.xml中引入【microcloud-security】模块

|  |
| --- |
| <?xml version=*"1.0"*?>  <project  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*  xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>com.liuxc</groupId>  <artifactId>microcloud</artifactId>  <version>0.0.1</version>  </parent>  <artifactId>microcloud-provider-dept-8001</artifactId>  <name>microcloud-provider-dept-8001</name>  <url>http://maven.apache.org</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>  <dependencies>  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-api</artifactId>  </dependency>  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-security</artifactId>  </dependency>  <!--spring-boot 依赖 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <!--rest 安全框架 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  </project> |

## 2.4Eureka服务注册与发现框架

对于服务发现框架可以简单的理解为服务的注册及使用操作步鄹。

在SpringCloud之中使用了大量的Netflix的开源项目，而其中EureKa就属于Netflix提供的发现服务组件。

* 使用EureKa发现服务的特性：

1. 利用统一的服务标记来进行微服务调用
2. 可以对同一项服务设计多个微服务，可以实现负载均衡
3. EureKa里面可以直接告诉用户那些服务不可用

EureKa的主要作用实际上和ZooKeeper是非常类似的，但是在SpringCloud虽然支持有ZooKeeper,不过从官方的宣传角度来说并不支持这样处理，推荐使用EureKa,因为速度更快，同时该服务组件是以程序的形式的方式出现的。

### 2.4.1Eureka基础环境配置

1.创建【microcloud-EureKa-7001】

2.修改【microcloud-EureKa-7001】项目中的pom.xml引入EureKa服务端的jar包

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency> |

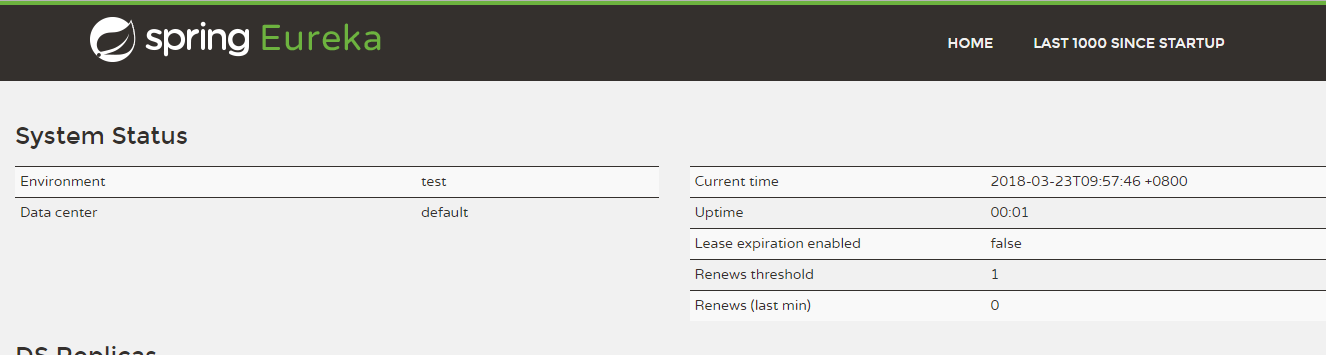
3.修改application.yml配置文件。配置EureKa服务

|  |
| --- |
| server:  port: 7001  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7001.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false**  fetch-registry: **false** |

4.【microcloud-EureKa-7001】修改EureKa程序启动类，追加EureKa服务声明。

|  |
| --- |
| package com.liuxc.microcloud;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.builder.SpringApplicationBuilder;  import org.springframework.boot.web.support.SpringBootServletInitializer;  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  @SpringBootApplication  @EnableEurekaServer  public class EureKa\_7001\_StartSpringCloundMain extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  return builder.sources(EureKa\_7001\_StartSpringCloundMain.class);  }  public static void main(String[] args) throws Exception {  SpringApplication.run(EureKa\_7001\_StartSpringCloundMain.class, args);  }  } |

当启动类访问7001看见此页面表示EureKa集成成功



### 2.4.2注册EureKa服务

1.修改【8001】模块POM.xml加入eureKa客服端依赖和健康监控模块

|  |
| --- |
| <?xml version=*"1.0"*?>  <project  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*  xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>com.liuxc</groupId>  <artifactId>microcloud</artifactId>  <version>0.0.1</version>  </parent>  <artifactId>microcloud-provider-dept-8001</artifactId>  <name>microcloud-provider-dept-8001</name>  <url>http://maven.apache.org</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>  <dependencies>  <!--api 模块 -->  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-api</artifactId>  </dependency>  <!--权限认真模块 -->  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-security</artifactId>  </dependency>  <!--spring-boot 依赖 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <!--rest 安全框架 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  <!--EureKa客服端 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!--congfig -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <!-- 健康监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!--mybatis -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  </dependency>  <!-- druid 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  </dependency>  <!-- mysql 驱动 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <!-- 测试模块 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <!-- 日志 -->  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  </project> |

2.修改启动【8001】application.yml文件配置EureKa客服端配置

|  |
| --- |
| eureka:  client:  service-url:  defaultZone: http://eureka-7001.com:7001/eureka |

3.修改启动类加上EureKa客服端注解

|  |
| --- |
| package com.liuxc.microcloud;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.builder.SpringApplicationBuilder;  import org.springframework.boot.web.support.SpringBootServletInitializer;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  @SpringBootApplication  @EnableEurekaClient  public class Dept\_8001\_StartSpringCloundMain extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  return builder.sources(Dept\_8001\_StartSpringCloundMain.class);  }  public static void main(String[] args) throws Exception {  SpringApplication.run(Dept\_8001\_StartSpringCloundMain.class, args);  }  } |

4.配置应用名称

|  |
| --- |
| spring:  application:  name: microcloud-provider-dept |

### 2.4.3EureKa服务信息

1.配置application.yml 服务信息显示

|  |
| --- |
| eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka-7001.com:7001/eureka  instance:  instance-id: www.lxc.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址 |

2.如果现在要想查看所有的微服务详细信息，则需要追加监控配置

|  |
| --- |
| <!-- 健康监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency> |

3.在父pom中加入一个插件读取pom文件数据

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <configuration>  <delimiters>  <delimiter>$</delimiter>  </delimiters>  </configuration>  </plugin> |

4.将设置pom文件设置为可访问的状态

|  |
| --- |
| <resource>  <!--配置文件的路径 -->  <directory>src/main/resources</directory>  <!--引入配置文件 -->  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.yml</include>  <include>\*\*/\*.xml</include>  <include>\*\*/\*.tld</include>  <include>\*\*/\*.keystore</include>  </includes>  <filtering>true</filtering>  </resource> |

5.配置info信息

|  |
| --- |
| info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$ |

### 2.4.4EureKa发现服务

在实际的项目运行过程之中需要通过EureKa作为所有微服务的监控处理

* 新服务追加的时候应该立刻进行注册；
* 当某一个服务下线之后应该可以进行清理；

1.【microcloud-eureka-7001】设置服务的清理间隔,修改application.yml

|  |
| --- |
| eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7001.com #定义EureKa实例所在的主机名称  server:  eviction-interval-timer-in-ms: 1000 #设置清理的间隔时间，默认是60\*1000； 60秒 |

一旦配置了清理的间隔为1秒的时间，则会在每秒的时候进行一次服务的清理过程，会出现如下的错误提示信息：

|  |
| --- |
| Running the evict task with compensationTime 0ms |

一般情况下，该配置不建议进行修改，默认就是60秒。

2.在EureKa里面有一个问题，这个问题就是它默认支持有保护模式的概念；所谓的保护模式指定是即便现在某一个微服务不可用了，eureKa不会清理，依然会进行该微服务的信息保存

|  |
| --- |
|  |

如果现在要想去改变这种保护模式的启用，则可以修改配置文件：

|  |
| --- |
| eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7001.com #定义EureKa实例所在的主机名称  server:  enable-self-preservation: **false** #表示关闭包含模式  eviction-interval-timer-in-ms: 60000 #设置清理的间隔时间，默认是60\*1000； 60秒 |

理论上只有关闭了保护模式之后才可以进行无效微服务的清理操作，但是很多时候EureKa里面也会自带有清除过程。

在一般正常的情况下不要去做这两个操作项的变更。按照默认的配置即可

3.微服务客户端之所以可以与EureKa之间保持联系，依靠的是心跳机制。也就是说你客服端可以自己来进行心跳的配置处理。修改8001application.yml配置文件：

|  |
| --- |
| eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka-7001.com:7001/eureka  instance:  lease-renewal-interval-in-seconds: 2 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 5 #心跳间隔周期，默认是90秒  instance-id: www.lxc.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址 |

如果不是必需的情况，一般都不要去修改。

如果配置了此选项“lease-expiration-duration-in-seconds”距离上一次发生心跳之后等待下一次发送心跳的间隔时间，如果超过了此间隔时间，则认为改微服务已经宕机了。

4.【8001】现在对于注册到EureKa上的微服务端也可以通过发现服务来进行一些服务器信息的获取。

增加一个控制调用方法：

|  |
| --- |
| @Autowired  **private** DiscoveryClient client;  @RequestMapping(value = "desCover", method = RequestMethod.***GET***)  **public** Object desCover() {    **return** client;  } |

5.修改启动类增加发现服务注解

|  |
| --- |
| package com.liuxc.microcloud;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.builder.SpringApplicationBuilder;  import org.springframework.boot.web.support.SpringBootServletInitializer;  import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  @SpringBootApplication  @EnableEurekaClient  @EnableDiscoveryClient  public class Dept\_8001\_StartSpringCloundMain extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  return builder.sources(Dept\_8001\_StartSpringCloundMain.class);  }  public static void main(String[] args) throws Exception {  SpringApplication.run(Dept\_8001\_StartSpringCloundMain.class, args);  }  } |

6.通过地址访问：http://www.lxc.com:8001/dept/desCover

|  |
| --- |
| {"services":["microcloud-provider-dept"],"localServiceInstance":{"host":"192.168.16.249","port":8001,"metadata":{},"uri":"http://192.168.16.249:8001","serviceId":"microcloud-provider-dept","secure":false}} |

EureKa里面就是根据这些信息来进行应用列表显示的。

### 2.4.5EureKa安全配置

1.修改【7001】的pom.xml文件，增加SpringSecurity开发包的支持

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency> |

2.配置安全访问的用户和密码

|  |
| --- |
| server:  port: 7001  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7001.com #定义EureKa实例所在的主机名称  server:  enable-self-preservation: **true** #false表示关闭包含模式  eviction-interval-timer-in-ms: 60000 #设置清理的间隔时间，默认是60\*1000； 60秒  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/ #加入了认证用户和密码的请求连接  security:  basic:  enabled: **true** #启动安全认证  user:  name: eureka  password: eureka  role:  - USER |

3.通过连接访问：<http://eureka:eureka@eureka-7001.com:7001/eureka/>

### 2.4.6EureKa –HA机制

现在已经成功的实现了一个EureKa服务器，但是现在属于单节点的服务运行过程，如果说现在单节点的EureKa出现了错误，导致无法使用，那么对于所有的微服务的架构就将出现整体的瘫痪，就需要进行EureKa集群搭建，同时利用集群可以实现HA的处理机制。

1.首先增加两个host：

|  |
| --- |
| 127.0.0.1 eureKa-7001.com  127.0.0.1 eureKa-7002.com  127.0.0.1 eureKa-7003.com |

2.分别修改7001,7002,7003的application.yml文件；采用双向注册的方式

【7001】application.yml:

|  |
| --- |
| server:  port: 7001  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7001.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  security:  basic:  enabled: **true** #启动安全认证  user:  name: eureka  password: eureka |

【7002】application.yml配置：

|  |
| --- |
| server:  port: 7002  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7002.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  security:  basic:  enabled: **true** #启动安全认证  user:  name: eureka  password: eureka |

【7003】application.yml配置：

|  |
| --- |
| server:  port: 7003  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7003.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/ #加入了认证用户和密码的请求连接  security:  basic:  enabled: **true** #启动安全认证  user:  name: eureka  password: eureka |

3.分别访问

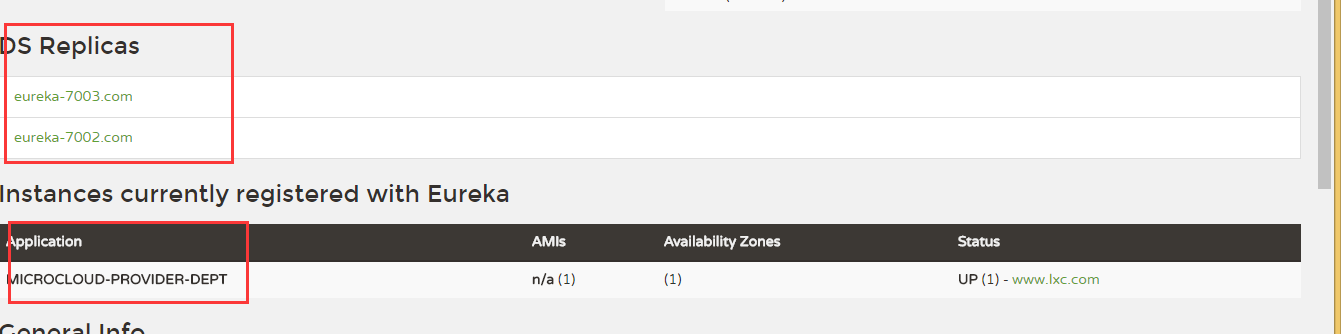
【7001EureKa主机后台】: http://eureka:eureka@eureka-7001.com:7001

【7003EureKa主机后台】: http://eureka:eureka@eureka-7002.com:7002

【7003EureKa主机后台】: <http://eureka:eureka@eureka-7003.com:7003>

4.修改【8001】application.yml文件将注册地址分别注册到这三台地址：

|  |
| --- |
| eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka,/http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 2 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 5 #心跳间隔周期，默认是90秒  instance-id: www.lxc.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址 |



表示EureKa集群搭建成功。

### 2.4.7EureKa 项目打包发布

1.修改在pom.xml 加入项目打包插件

|  |
| --- |
| <build>  <finalName>microcloud-eureka</finalName>  <plugins>  <plugin><!--该插值的主要功能是项目的打包处理 -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <!--设置程序执行的主类 -->  <mainClass>com.liuxc.microcloud.EureKa\_7003\_StartSpringCloundMain</mainClass>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> |

2.运行打包命令：clean package;

3.运行jar microcloud-eureka.jar

## 2.5 Ribbon负载均衡

Ribbon是一个服务调用的组件，基于EureKa注册中心的调用。并且是一个客服端实现负载均衡处理的组件。

### 2.5.1Ribbon 基础配置

1.修改【消费端】的pom.xml追加EureKa和Ribbon 依赖

|  |
| --- |
| <!--EureKa客服端 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!--congfig -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <!-- ribbon -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency> |

2.修改【消费者80】中的RestTemplastConfig配置加入Ribbon负载均衡配置

|  |
| --- |
| package com.liuxc.microcloud.config;  import java.nio.charset.Charset;  import org.apache.tomcat.util.codec.binary.Base64;  import org.springframework.cloud.client.loadbalancer.LoadBalanced;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.http.HttpHeaders;  import org.springframework.web.client.RestTemplate;  @Configuration  public class RestTemplatesConfig {  @Bean  public HttpHeaders getHeader() {// 进行 http头信息配置  HttpHeaders header = new HttpHeaders();// 定义一个认证头信息  String auth = "liuxc:liuxc"; // 认证原始信息  byte[] encode = Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASCII")));// 进行加密处理  String authHeader = "Basic " + new String(encode);  header.set("Authorization", authHeader); // 加入头信息  return header;  }  @Bean  @LoadBalanced//加入ribbon 负载均衡  public RestTemplate getRestTemplate() {  return new RestTemplate();  }  } |

3.修改【80】的application.yml 配置加入EureKa配置

|  |
| --- |
| server:  port: 80  eureka:  client: #客服端进行EureKa服务配置  register-with-eureka: **false**  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka,/http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接 |

4.在【80消费者端】启动主类中加入【@EnableEurekaClient】注解

5.使用EureKa注册中心的ServiceId进行调用Rest

|  |
| --- |
| @RestController  @RequestMapping("/consumer")  **public** **class** DeptController {  @Autowired  **private** RestTemplate template;  @Autowired  **private** HttpHeaders headers;//加入请求头  @RequestMapping(value = "/dept/add", method = RequestMethod.***GET***)  **public** Object add(Dept dept) {  Boolean flag = template.exchange("http://MICROCLOUD-PROVIDER-DEPT/dept/add", HttpMethod.***GET***, **new** HttpEntity<Object>(dept,headers), Boolean.**class**). getBody();  //Boolean flag = template.postForObject("http://www.lxc.com:8001/dept/add", dept, boolean.class);  **return** flag;  }  @RequestMapping(value = "/dept/findById", method = RequestMethod.***GET***)  **public** Object findById(Long id) {  Dept dept =template.exchange("http://MICROCLOUD-PROVIDER-DEPT/dept/findById/" + id, HttpMethod.***GET***, **new** HttpEntity<Object>(headers), Dept.**class**).getBody();  //Dept dept = template.getForObject("http://www.lxc.com:8001/dept/findById/" + id, Dept.class);  **return** dept;  }  @SuppressWarnings("unchecked")  @RequestMapping(value = "/dept/findAll", method = RequestMethod.***GET***)  **public** Object findAll() {  List<Dept> list = template.exchange("http://MICROCLOUD-PROVIDER-DEPT/dept/findAll", HttpMethod.***GET***, **new** HttpEntity<Object>(headers), List.**class**).getBody();  //List<Dept> list = template.getForObject("http://liuxc:liuxc@www.lxc.com:8001/dept/findAll", List.class);  **return** list;  }  } |

访问地址：<http://www.gavin.com/consumer/dept/findById?id=4>

这个时候有了Ribbon与EureKa之间的整合之后，用户不在具体的关注Rest服务的地址和端口。所有的信息获取都通过EuerKa获取。

### 2.5.2Ribbon 负载均衡

1.首先创建三个数据库，数据一致

2.建立对应的3个rest服务的域名：

|  |
| --- |
| 127.0.0.1 provider-8001.com  127.0.0.1 provider-8002.com  127.0.0.1 provider-8003.com |

3.修改【provider】每个的数据库配置和端口号

【provider-8001】

|  |
| --- |
| server:  port: 8001  url:jdbc:mysql://localhost:3306/master?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址 |

【provider-8002】

|  |
| --- |
| server:  port: 8002  url:jdbc:mysql://localhost:3306/slave?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址 |

【provider-8003】

|  |
| --- |
| server:  port: 8003  url:jdbc:mysql://localhost:3306/liuxc?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址 |

4.分别测试Rest接口：

<http://liuxc:liuxc@provider-8001.com:8001/dept/findAll>;

<http://liuxc:liuxc@provider-8002.com:8002/dept/findAll>;

<http://liuxc:liuxc@provider-8003.com:8003/dept/findAll>;

5.启动【消费端】调用服务接口。查看Ribbon负载均衡效果、默认采用的是轮询的负载均衡。

### 2.5.3自定义Ribbon负载均衡

1.在【消费者端】追加一个LoadBalanced配置类；

**注意:**这个类应该放在SpringCloud启动后找不到的位置，所以应该做一个新的包

|  |
| --- |
| **package** com.liuxc.commons;  **import** org.springframework.context.annotation.Bean;  **import** com.netflix.loadbalancer.IRule;  **public** **class** MyLoadBalancedConfig {  @Bean  **public** IRule ribbonRule(){    **return** **new** com.netflix.loadbalancer.RandomRule();//随机访问  }  } |

2.修改程序主类，追加Ribbon配置

|  |
| --- |
| import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.builder.SpringApplicationBuilder;  import org.springframework.boot.web.support.SpringBootServletInitializer;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  import org.springframework.cloud.netflix.ribbon.RibbonClient;  import com.liuxc.commons.MyLoadBalancedConfig;  @SpringBootApplication  @EnableEurekaClient  @RibbonClient(name="ribbonClient",configuration=MyLoadBalancedConfig.class)  public class Consumer\_Dept\_80\_StartSpringCloundMain extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  return builder.sources(Consumer\_Dept\_80\_StartSpringCloundMain.class);  }  public static void main(String[] args) throws Exception {  SpringApplication.run(Consumer\_Dept\_80\_StartSpringCloundMain.class, args);  }  } |

此时就完成了自定义的配置；

### 2.5.4Ribbon 脱离 EureKa

1.新建一个项目【microcloud-consumber-ribbon】

2.修改application.yml文件

|  |
| --- |
| server:  port: 80  ribbon:  euerKa:  enabled: **false** #关闭启用eureKa  microcloud-provider-dept:  ribbon:  listOfservers: http://provider-8001.com:8001,http://provider-8002.com:8002,http://provider-8003.com:8003 #ribbon 可访问的主机地址 |

3.修改RestTemplaseConfig配置类，不在需要@LoadBalanced注解

4.修改启动类型去掉：@RibbonClient注解

5.修改Controller.使用ribbon调用方式

|  |
| --- |
| package com.liuxc.microcloud.controller;  import java.net.URI;  import java.util.List;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.cloud.client.ServiceInstance;  import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;  import org.springframework.cloud.netflix.ribbon.RibbonClient;  import org.springframework.http.HttpEntity;  import org.springframework.http.HttpHeaders;  import org.springframework.http.HttpMethod;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;  import com.liuxc.commons.MyLoadBalancedConfig;  import com.liuxc.microcloud.api.Dept;  @RestController  @RequestMapping("/consumer")  @RibbonClient(name="microcloud-provider-dept",configuration=MyLoadBalancedConfig.class)  //这里name 应该和yml 文件中的配置名称一致  public class DeptController {  private final String REST\_DEPR\_TOPIC="microcloud-provider-dept";  @Autowired  private RestTemplate template;  @Autowired  private HttpHeaders headers;//加入请求头  @Autowired  private LoadBalancerClient ribbonCilent;  @SuppressWarnings("unchecked")  @RequestMapping(value = "/dept/findAll", method = RequestMethod.GET)  public Object findAll() {  ServiceInstance serviceInstance = ribbonCilent.choose(REST\_DEPR\_TOPIC);  System.out.println("主机："+serviceInstance.getHost()+",端口："+serviceInstance.getPort());  URI deptUri=URI.create(String.format("http://%s:%s/dept/findAll", serviceInstance.getHost(),serviceInstance.getPort()));  List<Dept> list = template.exchange(deptUri, HttpMethod.GET, new HttpEntity<Object>(headers), List.class).getBody();  return list;  }  } |

这种模式并不标准，只能说是Ribbon自己所具备的一项功能、

## 2.6 Feign 接口转换

此技术的核心是将rest服务数据转换为服务接口；

### 2.6.1 feign的基本使用、

1.新建一个模块【microcloud-consumber-feign】

2.为了在项目中使用到feign技术，修改【microcloud-consumber-feign】中的pom.xml加入我们feign的依赖支持：

|  |
| --- |
| <!-- ribbon 当项目中引入了 feign组件后 就可以注释掉ribbon 了 因为feign中包含了 ribbon-->  <!-- <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency> -->    <!-- feign 组件 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency> |

Feign 操作需要Ribbon支持，所以导入了以上的依赖包之后就表示项目之中已经存在有了Ribbon的项目支持库

3.建立一个【microcloud-service】模块，这个模块专门负责客户端接口的定义;修改pom.xml 中引入【microcloud-api】模块和feign的依赖；

|  |
| --- |
| <?xml version=*"1.0"*?>  <project  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*  xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>com.liuxc</groupId>  <artifactId>microcloud</artifactId>  <version>0.0.1</version>  </parent>  <artifactId>microcloud-service</artifactId>  <name>microcloud-service</name>  <url>http://maven.apache.org</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>  <dependencies>  <!--api -->  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-api</artifactId>  </dependency>  <!-- feign 组件 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency>  </dependencies>  </project> |

4.【microcloud-service】中增加Feign的配置类；【注意配置类的包的定义结构，放在启动类能扫描的到的包下】

|  |
| --- |
| **package** com.liuxc.microcloud.config;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** feign.auth.BasicAuthRequestInterceptor;  @Configuration  **public** **class** FeignClientConfig {  /\*\*  \* feign 处理权限认证处理  \* 这里替换原来的 header  \*/  @Bean  **public** BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  **return** **new** BasicAuthRequestInterceptor("liuxc", "liuxc");  }  } |

5.定义Rest服务转接口

|  |
| --- |
| package com.liuxc.microcloud.service;  import java.util.List;  import org.springframework.cloud.netflix.feign.FeignClient;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import com.liuxc.microcloud.api.Dept;  import com.liuxc.microcloud.config.FeignClientConfig;  /\*\*  \* value 为eureKa中的服务名称 configuration 为权限认证配置  \*/  @FeignClient(value = "MICROCLOUD-PROVIDER-DEPT", configuration = FeignClientConfig.class)  public interface IDeptClientService {  @RequestMapping(value = "/dept/add", method = RequestMethod.POST)  public boolean add(Dept dept);  @RequestMapping(value = "dept/findById/{id}", method = RequestMethod.GET)  public Dept findById(@PathVariable(value = "id") Long id);  @RequestMapping(value = "dept/findALL", method = RequestMethod.GET)  public List<Dept> findAll();  } |

6.修改【microcloud-consumber-feign】的controller使用接口调用；

|  |
| --- |
| package com.liuxc.microcloud.controller;  import java.util.List;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.liuxc.microcloud.api.Dept;  import com.liuxc.microcloud.service.IDeptClientService;  @RestController  @RequestMapping("/consumer")  public class DeptController {  @Autowired  private IDeptClientService deptService;  @RequestMapping(value = "/dept/add", method = RequestMethod.GET)  public Object add(Dept dept) {  Boolean flag = deptService.add(dept);  return flag;  }  @RequestMapping(value = "/dept/findById", method = RequestMethod.GET)  public Object findById(Long id) {  Dept dept = deptService.findById(id);  return dept;  }  @RequestMapping(value = "/dept/findAll", method = RequestMethod.GET)  public Object findAll() {  List<Dept> list = deptService.findAll();  return list;  }  } |

7.修改启动类增加Feign的启动配置；

|  |
| --- |
| package com.liuxc.microcloud;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.builder.SpringApplicationBuilder;  import org.springframework.boot.web.support.SpringBootServletInitializer;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  import org.springframework.cloud.netflix.feign.EnableFeignClients;  @SpringBootApplication  @EnableEurekaClient  @EnableFeignClients  public class Consumer\_Dept\_Feign\_StartSpringCloundMain extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  return builder.sources(Consumer\_Dept\_Feign\_StartSpringCloundMain.class);  }  public static void main(String[] args) throws Exception {  SpringApplication.run(Consumer\_Dept\_Feign\_StartSpringCloundMain.class, args);  }  } |

### 2.6.2feign相关配置

Feign之中最为核心的作用就是Rest服务的信息转换为接口，但是实际的使用之中也需要考虑到一些配置情况，例如数据压缩。

1.这个时候可以考虑修改application.yml配置文件对传输数据进行压缩；

|  |
| --- |
| feign:  compression:  request:  mime-types: #配置能够压缩的类型  - txt/xml  - application/xml  - application/json  min-request-size: 2048 #超过2048的字节的请求需要被压缩 |

2.开启feign的相关日志信息（默认不开启）；

|  |
| --- |
| logging:  level:  com.liuxc.microcloud.service: debug #数据接口的信息 |

修改【microcloud-service】feign配置，增加日志输出

|  |
| --- |
| **package** com.liuxc.microcloud.config;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** feign.Logger;  **import** feign.auth.BasicAuthRequestInterceptor;  @Configuration  **public** **class** FeignClientConfig {  @Bean  **public** Logger.Level getFeignLoggerLevel() {  **return** Logger.Level.***FULL***;  }  /\*\*  \* feign 处理权限认证处理 这里替换原来的 header  \*/  @Bean  **public** BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  **return** **new** BasicAuthRequestInterceptor("liuxc", "liuxc");  }  } |

观察整个feign调用的过程：

* 当使用Feign要通过接口的方法访问Rest服务的时候会根据设置的服务类型发出请求；这请求是发送给EureKa的。
* 随后由于配置了授权处理，所以继续发送授权信息（”Authorization”）;
* 在进行服务调用的时候Feign融合了Ribbon技术，所以也支持负载均衡的处理。

总结：Feign=RestTemplate+HttpHeader+Ribbon+EureKa综合体=业务接口的自动实例化。

## 2.7Hystrix熔断机制

为了保证不出现错误的提示信息页面，应该做一个失败的错误处理，而这种错误的处理就可以称为熔断机制，在实际之中更为复杂的SpringCloud调用里面熔断机制实际上更加有用处。

对于熔断技术的实现需要考虑一下几种情况：

* 出现错误之后可以fallback错误的处理信息；
* 如果要结合Feign一起使用的时候还需要在Feign进行熔断的配置；

### 2.7.1 Hystrix 服务端失败回调 fallback

1修改【microcloud-provider-dept-hystrix-8001】模块POM.xml追加Hystrix依赖；

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

2.修改【microcloud-provider-dept-hystrix-8001】DeptRest服务提供类；

|  |
| --- |
| **package** com.liuxc.microcloud.rest;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.liuxc.microcloud.api.Dept;  **import** com.liuxc.microcloud.service.IDeptService;  **import** com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;  @RestController  @RequestMapping("/dept")  **public** **class** DeptRest {  @Autowired  **private** IDeptService deptService;  @HystrixCommand(fallbackMethod = "fallbackFindById")//如果执行findById方法出现了错误则执行fallbackFindById方法  @RequestMapping(value = "/findById/{id}", method = RequestMethod.***GET***)  **public** Dept findById(@PathVariable Long id) {  Dept dept = deptService.findById(id);  **if** (dept == **null**) {  **throw** **new** RuntimeException();  }  **return** deptService.findById(id);  }  **public** Dept fallbackFindById(@PathVariable Long id) {// 此时方法的参数和findById 一致  Dept dept = **new** Dept();  dept.setId(id);  dept.setName("ERROR: ID not found dept");  dept.setLoc("null dataBase");  **return** dept;  }  } |

一旦findById()方法上抛出了错误信息，那么就认为该服务有问题，会默认使用@HystrixCommand(fallbackMethod = "fallbackFindById")注解上配置的method调用类中的指定方法，返回相应的数据。

3.修改启动主类，启动熔断处理；

|  |
| --- |
| **package** com.liuxc.microcloud;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.boot.builder.SpringApplicationBuilder;  **import** org.springframework.boot.web.support.SpringBootServletInitializer;  **import** org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;  **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;  **import** org.springframework.cloud.netflix.eureka.EnableEurekaClient;  @SpringBootApplication  @EnableEurekaClient // 启用eureka客服端  @EnableDiscoveryClient // 启用eureka复现服务  @EnableCircuitBreaker // 熔断服务  **public** **class** Dept\_8001\_StartSpringCloundMain **extends** SpringBootServletInitializer {  @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** builder.sources(Dept\_8001\_StartSpringCloundMain.**class**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  SpringApplication.*run*(Dept\_8001\_StartSpringCloundMain.**class**, args);  }  } |

现在的处理情况是：服务器出现了错误（但并不表示提供方停掉了服务），那么此时会调用指定方法的fallback处理。

### 2.7.2服务降级

所有的RPC技术里面服务降级是一个最为重要的话题，所谓的降级指的是当服务的提供方不可使用的时候，程序不会出现异常，而会出现本地的操作调用。

服务的降级处理是在客户端实现的，与你的服务器端没有关系。

1.修改【microcloud-service】模块增加Feign hystrix fallback 类

|  |
| --- |
| **package** com.liuxc.microcloud.fallback;  **import** java.util.List;  **import** org.springframework.stereotype.Component;  **import** com.liuxc.microcloud.api.Dept;  **import** com.liuxc.microcloud.service.IDeptClientService;  **import** feign.hystrix.FallbackFactory;  @Component  **public** **class** IDeptClientServiceFallBackFactory **implements** FallbackFactory<IDeptClientService> {  @Override  **public** IDeptClientService create(Throwable arg0) {  **return** **new** IDeptClientService() {  @Override  **public** **boolean** add(Dept dept) {  // **TODO** Auto-generated method stub  **return** **false**;  }  @Override  **public** Dept findById(Long id) {  Dept dept = **new** Dept();  dept.setId(id);  dept.setName("ERROR: ID not found dept");  dept.setLoc("客户端提供");  **return** dept;  }  @Override  **public** List<Dept> findAll() {  // **TODO** Auto-generated method stub  **return** **null**;  }  };  }  } |

2.修改IDeptClinetService类型，增加fallback配置；

|  |
| --- |
| **package** com.liuxc.microcloud.service;  **import** java.util.List;  **import** org.springframework.cloud.netflix.feign.FeignClient;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** com.liuxc.microcloud.api.Dept;  **import** com.liuxc.microcloud.config.FeignClientConfig;  **import** com.liuxc.microcloud.fallback.IDeptClientServiceFallBackFactory;  /\*\*  \* value 为eureKa中的服务名称 configuration 为权限认证配置  \*/  @FeignClient(value = "MICROCLOUD-PROVIDER-DEPT", configuration = FeignClientConfig.**class**, fallbackFactory = IDeptClientServiceFallBackFactory.**class**)  **public** **interface** IDeptClientService {  @RequestMapping(value = "/dept/add", method = RequestMethod.***POST***)  **public** **boolean** add(Dept dept);  @RequestMapping(value = "/dept/findById/{id}", method = RequestMethod.***GET***)  **public** Dept findById(@PathVariable(value = "id") Long id);  @RequestMapping(value = "/dept/findAll", method = RequestMethod.***GET***)  **public** List<Dept> findAll();  } |

3.修改【microcloud-consumber-hystrix】模块application.yml文件开启hystrix支持；

|  |
| --- |
| feign:  hystrix:  enabled: **true** |
| 此时即使服务端无法继续提供服务了，由于存在有服务降级机制，也会保证服务不可用时，也会获得一些服务的提供信息。 |

### 2.7.3HystrixDashBoard监控

1.首先为了方便监控，将建立一个新的监控模：microcloud-consumer-hystrix-dashboard

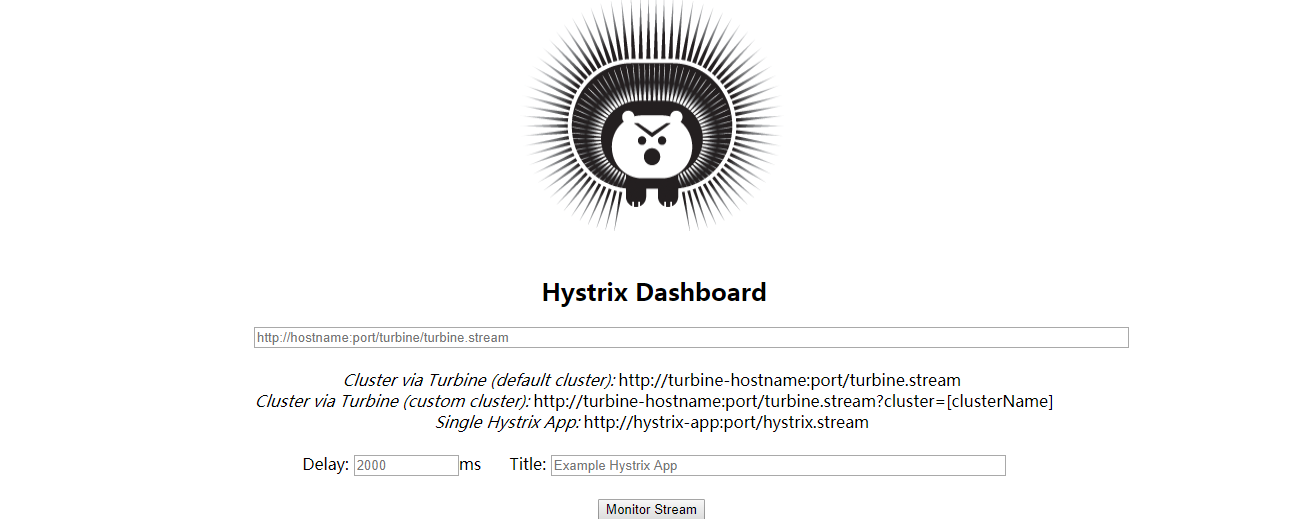
2.修改【microcloud-consumer-hystrix-dashboard】的pom.xml文件追加监控依赖支持；

|  |
| --- |
| <dependencies>  <!--spring-boot 依赖 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <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>  </dependencies> |

3.修改我们的application.yml文件将端口修改为9091

4.修改我们启动类，增加监控面板开启

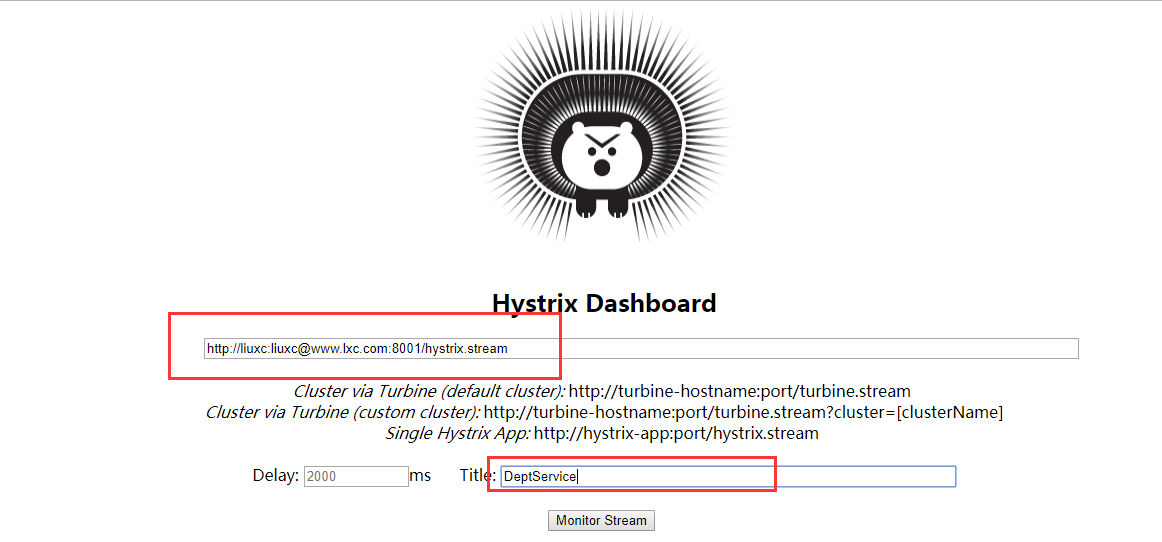
|  |
| --- |
| **package** com.liuxc.microcloud;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.boot.builder.SpringApplicationBuilder;  **import** org.springframework.boot.web.support.SpringBootServletInitializer;  **import** org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;  @SpringBootApplication  @EnableHystrixDashboard  **public** **class** HystrixDashBoardApplication\_9001 **extends** SpringBootServletInitializer {  @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** builder.sources(HystrixDashBoardApplication\_9001.**class**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  SpringApplication.*run*(HystrixDashBoardApplication\_9001.**class**, args);  }  } |



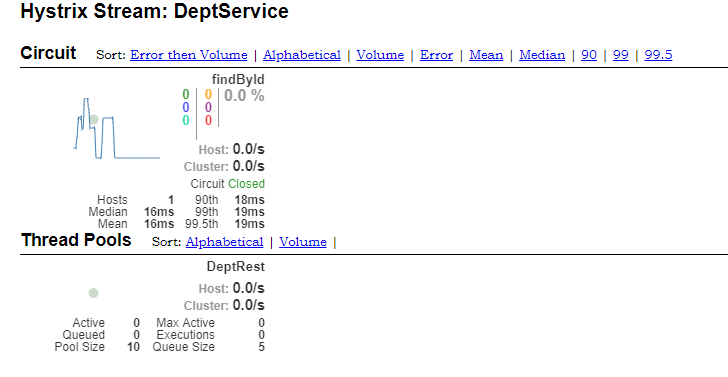
看见访问：<http://www.gavin.com:9001/hystrix> 看见此页面表示监控成功；

5.访问服务端监控地址： http://www.lxc.com:8001/hystrix.stream

6.填写监控的信息



启动监控，调用整个微服务。



### 2.7.4Turbine聚合监控

HystrixDashboard主要的功能是可以针对于某一项微服务进行监控，但是如果说现在有许多微服务，那么这种情况下就可以利用turbine技术实现。

准备出一个新的微服务：Company，这个微服务不打算使用SpringSecurity安全处理，以及数据库的操作。只是简单的数据信息。

1.新建一个【microcloud-provider-company-8101】微服务模块，并修改pom.xml文件

|  |
| --- |
| <dependencies>  <!--api 模块 -->  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-api</artifactId>  </dependency>  <!--spring-boot 依赖 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <!--EureKa客服端 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!--congfig -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <!-- hystrix 熔断 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>  <!-- 健康监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- 测试模块 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies> |

2.【microcloud-api】模块中增加Company,VO类

|  |
| --- |
| **package** com.liuxc.microcloud.api;  **import** java.io.Serializable;  @SuppressWarnings("serial")  **public** **class** Company **implements** Serializable {  **private** String title;  **private** String note; |

3.修改【microcloud-provider-company-8101】application.yml文件

|  |
| --- |
| server:  port: 8101  eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: company-8101.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址  spring:  application:  name: microcloud-provider-dept  info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$ |

4.创建company微服务；

|  |
| --- |
| **package** com.liuxc.microcloud.rest;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.liuxc.microcloud.api.Company;  **import** com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;  @RestController  @RequestMapping("/company")  **public** **class** DeptRest {  @HystrixCommand(fallbackMethod = "fallbackGetTitle")  @RequestMapping(value = "/getTitle/{title}", method = RequestMethod.***GET***)  **public** Company findById(@PathVariable("title") String title) {  Company company = **new** Company();  company.setTitle(title);  company.setNote("www.liuxc.com");  **return** company;  }  **public** Company fallbackGetTitle(@PathVariable("title") String title) {// 此时方法的参数和findById  Company company = **new** Company();  company.setTitle(title);  company.setNote("www.liuxc.com");  **return** company;  }  } |

启动【microcloud-provider-company-8101】访问地址：

<http://company-8101.com:8101/company/getTitle/hello>

监控地址: <http://company-8101.com:8101/hystrix.stream>

5.如果要想实现trubine的配置，则需要建立一个turbine项目，这个项目可以直接通过之前的【microcloud-consumber-hystrix-dashboard】进行复制为【microcloud-consumber-turbine】

6. 【microcloud-consumber-turbine】修改pom.xml文件追加turbine依赖程序包；

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

7.修改【microcloud-consumber-turbine】的application.yml

|  |
| --- |
| server:  port: 9101  turbine:  app-config: MICROCLOUD-PROVIDER-COMPANY,MICROCLOUD-PROVIDER-DEPT#监控eureka中的应用名称  cluster-name-expression: new String("default") #监控表达式    eureka:  client: #客服端进行EureKa服务配置  register-with-eureka: **false**  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接 |

此时给出的两个微服务里面：无安全认证(MICROCLOUD-PROVIDER-COMPANY)，有安全认证（MICROCLOUD-PROVIDER-DEPT）

8.建立【microcloud-consumber-turbine】的启动程序类

|  |
| --- |
| package com.liuxc.microcloud;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;  import org.springframework.cloud.netflix.turbine.EnableTurbine;  @SpringBootApplication  @EnableHystrixDashboard  @EnableTurbine  public class TurbineApplication\_9001 {  public static void main(String[] args) throws Exception {  SpringApplication.run(TurbineApplication\_9001.class, args);  }  } |

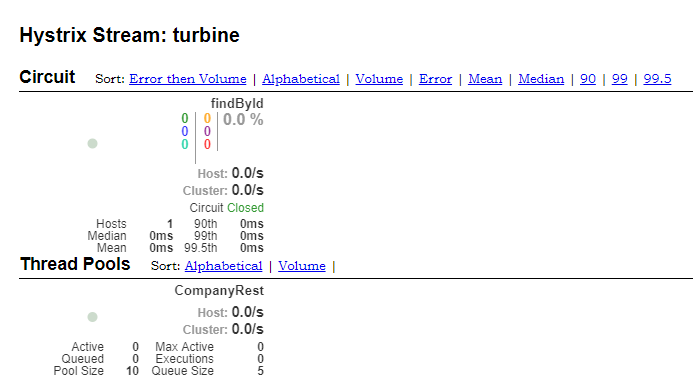
9.运行【microcloud-consumber-hystrix-dashboard】hystrix-dashboard监控

运行【microcloud-consumber-turbine】turbine 监控

Turbine访问路径：<http://localhost:9101/turbine.stream>

HystrixDashBorad访问路径：<http://www.lxc.com:9001/hystrix.stream>

在监控的位置上填写之前设置好的turbine监控地址；



10.【microcloud-consumber-turbine】如果现在需要turbine进行加密的微服务的访问操作，只能采用一种折中的访问方式

就是去修改整个项目中的安全策略。修改【microcloud-security】中web配置，追加我们不需要认证的过滤；

|  |
| --- |
| **package** com.liuxc.microcloud.config;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  **import** org.springframework.security.config.annotation.web.builders.HttpSecurity;  **import** org.springframework.security.config.annotation.web.builders.WebSecurity;  **import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  **import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  **import** org.springframework.security.config.http.SessionCreationPolicy;  @Configuration  @EnableWebSecurity  **public** **class** WebSecuityConfig **extends** WebSecurityConfigurerAdapter {  @Override  **public** **void** configure(WebSecurity web) **throws** Exception {  //表示/hystrix.stream和/turbine.stream访问路径不需要安全认证；  web.ignoring().antMatchers("/hystrix.stream","/turbine.stream");  }  @Autowired  **public** **void** configGlobal(AuthenticationManagerBuilder builder) **throws** Exception {  // 配置security 认证用户  builder.inMemoryAuthentication().withUser("liuxc").password("liuxc").roles("USER").and().withUser("admin")  .password("root").roles("USER", "ADMIN");  }  @Override  **protected** **void** configure(HttpSecurity http) **throws** Exception {  // 所有的请求都需要通过认证后才可以正常进行  http.httpBasic().and().authorizeRequests().anyRequest().fullyAuthenticated();  // 所有rest的服务都需要设置session无状态模式提高效率  http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);  }  } |

现在所有的安全策略会自动抛开以上两个访问路径，这种是基于Bean配置。如果要是你现在是基于的是application.yml的配置则追加如下内容

|  |
| --- |
| security:  ignored:  - /hystrix.stream  - /turbine.stream |

## 2.8 Zuul路由访问

现在的所有微服务都是通过EureKa找到的，但是在很多的开发之中为了规范微服务的使用,提供有一个路由的处理组件：Zuul,也就是说Zuul就作为中间的一个代理层出现。

### 2.8.1Zuul的基本使用

1. 新建一个模块【microcloud-zuul-gateway】

2. 修改【microcloud-zuul-gateway】pom.xml文件追加Zuul的依赖包

注意：Zuul服务最终还是会注册到EureKa之中，那么千万要将其配置好。

|  |
| --- |
| <!--spring-boot 依赖 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <!--EureKa客服端 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!--congfig -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>    <!-- zuul 路由 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId>  </dependency>    <!-- 健康监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- 测试模块 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies> |

3.修改application.yml的配置

|  |
| --- |
| server:  port: 9501  eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: gateway-9501.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址  spring:  application:  name: microcloud-zuul-gateway  info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$ |

4.编写启动类程序

|  |
| --- |
| **package** com.liuxc.microcloud;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.boot.builder.SpringApplicationBuilder;  **import** org.springframework.boot.web.support.SpringBootServletInitializer;  **import** org.springframework.cloud.netflix.zuul.EnableZuulProxy;  @SpringBootApplication  @EnableZuulProxy  **public** **class** Zuul\_9501\_StartSpringCloundMain **extends** SpringBootServletInitializer {  @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** builder.sources(Zuul\_9501\_StartSpringCloundMain.**class**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  SpringApplication.*run*(Zuul\_9501\_StartSpringCloundMain.**class**, args);  }  } |

5.访问公司微服务信息：

\*原始访问路径：<http://company-8101.com:8101/company/getTitle/hello>

\*zuul代理后路径：

<http://gateway-9501.com:9501/microcloud-provider-company/company/getTitle/hello>

### 2.8.2Zuul的路由功能

整体的Zuul运行之后你会发现，zuul所实现的就是一个代理功能，那么现在就会出现一个问题。

|  |
| --- |
| <http://gateway-9501.com:9501/microcloud-provider-company/company/getTitle/hello> |

此时必须知道应用程序的名称，但是如果不知道这个名称肯定无法访问，可是如果让用户知道了这个名称，那么使用zuul就没有任何的实际意义了，直接调用即可。而zuul的主要功能是代理，那么代理的功能就是不让用户看见真实的操作。所以在实际的使用之中就需要为zuul设置一些路由规则。

1.修改【microcloud-zuul-gateway-9501】为指定的应用设置路径，修改application.yml

|  |
| --- |
| zuul:  routes:  microcloud-provider-company: company-proxy/\*\* |

那么此时就可以通过company-proxy 来访问microcloud-provider-company 了。

地址：<http://gateway-9501.com:9501/company-proxy/company/getTitle/hello>

但是现在还会存在有一个实际的问题，虽然现在开启了路由访问支持，但是依然支持通过应用名称进行访问。

2.【microcloud-zuul-gateway-9501】修改application.yml配置文件忽悠掉应用名称访问。

|  |
| --- |
| zuul:  ignored-services: #忽略掉使用应用名称的访问  microcloud-provider-company  routes:  microcloud-provider-company: company-proxy/\*\* #代理应用名称访问路径 |

以上配置任由比较大的缺陷，当我们应有名称有上百个的时候需要一个一个配置，所以在这里可以使用我们的统配符的方式进行配置：

|  |
| --- |
| zuul:  ignored-services: #忽略掉使用应用名称的访问  "\*"  routes:  microcloud-provider-company: company-proxy/\*\* #代理应用名称访问路径 |

现在表示所有的EureKa中的服务名称的访问都要忽略掉。

3. 【microcloud-zuul-gateway-9501】如果说现在不想通过EureKa进行访问，则也可以直接连接到company微服务地址

|  |
| --- |
| zuul:  ignored-services: #忽略掉使用应用名称的访问  "\*"  routes:  company.path: /company-proxy/\*\*  company.url: http://company-8101.com:8101 |

4.设置公共前缀，一旦存在有前缀定义之后所有微服务的访问就必须有追加前缀名称

|  |
| --- |
| zuul:  prefix: /liuxc #设置公共前缀  ignored-services: #忽略掉使用应用名称的访问  "\*"  routes:  microcloud-provider-company: /company-proxy/\*\* |

访问地址：

<http://gateway-9501.com:9501/liuxc/company-proxy/company/getTitle/hello>

以上的地址：

“/liuxc”：整个zuul的前缀；

“/company-proxy/c”：是在zuul中定义的映射路径；

“/company/getTitle/hello”：是微服务提供者提供的操作路径。

### 2.8.3zuul过滤

对于zuul的功能本质上就是属于一个代理操作，但是在实际的使用之中，所有的微服务一定都要有自己的认真信息，那么在这样的状态下，如果你当前所代理的微服务具有认真信息，那么就必须其访问前追加认证的头部操作，这样的功能就需要zuul的过滤操作。

1. 【microcloud-zuul-gateway-9501】修改application.yml配置，这个配置之中代理我们需要认证的dept微服务。

|  |
| --- |
| zuul:  prefix: /liuxc #设置公共前缀  ignored-services: #忽略掉使用应用名称的访问  "\*"  routes:  microcloud-provider-company: /company-proxy/\*\*  microcloud-provider-dept: /dept-proxy/\*\* |

此时访问路径：

<http://liuxc:liuxc@gateway-9501.com:9501/liuxc/dept-proxy/dept/findById/4>用户名密码只是传递给zuul，而zuul并不能够将认证的信息传递给部门微服务之中。

2.创建认证过滤器

|  |
| --- |
| **package** com.liuxc.microcloud;  **import** java.nio.charset.Charset;  **import** org.apache.tomcat.util.codec.binary.Base64;  **import** com.netflix.zuul.ZuulFilter;  **import** com.netflix.zuul.context.RequestContext;  **public** **class** AuthorizationRequestFilter **extends** ZuulFilter {// 认证授权过滤器  @Override  **public** Object run() {  RequestContext currentContext = RequestContext.*getCurrentContext*();  String auth = "liuxc:liuxc"; // 认证原始信息  **byte**[] encode = Base64.*encodeBase64*(auth.getBytes(Charset.*forName*("US-ASCII")));// 进行加密处理  String authHeader = "Basic " + **new** String(encode);  currentContext.addZuulRequestHeader("Authorization", authHeader);  **return** **null**;  }  @Override  **public** **boolean** shouldFilter() {// 是否需要执行该过滤器  **return** **true**;  }  @Override  **public** **int** filterOrder() {  **return** 0;//设置优先级  }  @Override  **public** String filterType() {  //在设置zuul过滤时，可以设置其过滤的位置  //1.pre:在请求发出前，执行过滤。如果要进行认证访问，肯定在请求前设置头信息  //2.route：在进行路由请求的时候被调用  //3.post：在路由之后发送请求时被调用  //4.error：出现错误之后进行调用  **return** "pre";  }  } |

3.建立一个配置程序类作为认证请求的配置Bean

|  |
| --- |
| **package** com.liuxc.microcloud.config;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** com.liuxc.microcloud.filter.AuthorizationRequestFilter;  @Configuration  **public** **class** ZuulConfig {  @Bean  **public** AuthorizationRequestFilter getAuthorizationRequestFilter() {    **return** **new** AuthorizationRequestFilter();  }  } |

4.【microcloud-zuul-gateway-9501】考虑到zuul也需要进行安全访问，所有应改修改项目中的pom.xml配置文件，追加springSecurity,并且修改application.yml配置安全认证信息

|  |
| --- |
| spring:  application:  name: microcloud-zuul-gateway  server:  port: 9501  zuul:  prefix: /liuxc #设置公共前缀  ignored-services: #忽略掉使用应用名称的访问  "\*"  routes:  microcloud-provider-company: /company-proxy/\*\*  microcloud-provider-dept: /dept-proxy/\*\*  security:  basic:  enabled: **true** #开启权限认证  user:  name: zuul  password: zuul  eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: gateway-9501.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址  info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$ |

访问地址：

<http://zuul:zuul@gateway-9501.com:9501/liuxc/dept-proxy/dept/findById/4>

5.【microcloud-service】现在所有的服务要通过zuul的代理来进行操作，对于代理的配置如果要想通过feign进行访问，那么在编写feign的时候就必须设置代理的服务名称

|  |
| --- |
| **package** com.liuxc.microcloud.service;  **import** java.util.List;  **import** org.springframework.cloud.netflix.feign.FeignClient;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** com.liuxc.microcloud.api.Dept;  **import** com.liuxc.microcloud.config.FeignClientConfig;  **import** com.liuxc.microcloud.fallback.IDeptClientServiceFallBackFactory;  /\*\*  \* value 为eureKa中的服务名称 configuration 为权限认证配置  \*/  @FeignClient(value = "MICROCLOUD-ZUUL-GATEWAY", configuration = FeignClientConfig.**class**, fallbackFactory = IDeptClientServiceFallBackFactory.**class**)  **public** **interface** IDeptClientService\_Zuul {  @RequestMapping(value = "/liuxc/dept-proxy/dept/add", method = RequestMethod.***POST***)  **public** **boolean** add(Dept dept);  @RequestMapping(value = "/liuxc/dept-proxy/dept/findById/{id}", method = RequestMethod.***GET***)  **public** Dept findById(@PathVariable(value = "id") Long id);  @RequestMapping(value = "/liuxc/dept-proxy/dept/findAll", method = RequestMethod.***GET***)  **public** List<Dept> findAll();  } |

6.修改【microcloud-service】配置zuul代理的认证信息

|  |
| --- |
| **package** com.liuxc.microcloud.config;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** feign.Logger;  **import** feign.auth.BasicAuthRequestInterceptor;  @Configuration  **public** **class** FeignClientConfig {  @Bean  **public** Logger.Level getFeignLoggerLevel() {  **return** Logger.Level.***FULL***;  }  /\*\*  \* feign 处理权限认证处理 这里替换原来的 header  \*/  @Bean  **public** BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  //return new BasicAuthRequestInterceptor("liuxc", "liuxc");  **return** **new** BasicAuthRequestInterceptor("zuul", "zuul");  }  } |

7.启动【microcloud-consumber-feign】访问地址：

<http://localhost/consumer/dept/findById?id=4>

成功表示集成了zuul代理，并且权限认证加上了

8、【microcloud-zuul-gateway-9501】默认情况下只要配置了认证授权处理,就可以进行一个正常的启动，如果现在有些过滤器不想使用了，则也可以通过修改application,yml配置文件让其禁用

|  |
| --- |
| zuul:  AuthorizationRequestFilter:  pre:  disable: **true** #表示禁用 AuthorizationRequestFilter 过滤器 |

### 2.8.4 zuul服务降级

那么如果被代理的服务突然断掉了，那么这个时候zuul上面一定会显示出错误信息。

|  |
| --- |
|  |

如果现在程序无法使用，则zuul的代理执行时就会出现timeout的提示信息。由于现在的客服端已经提供有了feign中的服务降级的配置支持。所以客户端没有任何的问题。问题只出现在代理端。但是对一个完善的zuul代理应实现服务降级处理操作，所以如果有需要也可以在zuul中进行服务降级配置。

1.建立一个FallBack的程序处理类

|  |
| --- |
| **package** com.liuxc.microcloud.fallback;  **import** java.io.ByteArrayInputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** org.springframework.cloud.netflix.zuul.filters.route.ZuulFallbackProvider;  **import** org.springframework.http.HttpHeaders;  **import** org.springframework.http.HttpStatus;  **import** org.springframework.http.client.ClientHttpResponse;  **import** org.springframework.stereotype.Component;  @Component  **public** **class** DeptProviderFallBack **implements** ZuulFallbackProvider {  @Override  **public** String getRoute() {// 设置路由的地址  **return** "microcloud-provider-dept";  }  @Override  **public** ClientHttpResponse fallbackResponse() {// 设置回退信息  **return** **new** ClientHttpResponse() {  @Override  **public** InputStream getBody() **throws** IOException {  **return** **new** ByteArrayInputStream("\"message\":\"路由失败！\"".getBytes());  }  @Override  **public** HttpHeaders getHeaders() {  HttpHeaders headers = **new** HttpHeaders();  headers.add("Content-Type", "text/html; charset=UTF-8");  **return** headers;  }  @Override  **public** HttpStatus getStatusCode() **throws** IOException {  **return** HttpStatus.***OK***;  }  @Override  **public** **int** getRawStatusCode() **throws** IOException {  **return** HttpStatus.***OK***.value();  }  @Override  **public** String getStatusText() **throws** IOException {  **return** HttpStatus.***OK***.getReasonPhrase();  }  @Override  **public** **void** close() {  }  };  }  } |

## 2.9上传微服务

### 2.9.1搭建上传微服务

1.创建一个新的项目，项目【microcloud-provider-upload-8201】

2.修改application.yml的配置文件，对上传文件进行基础配置

|  |
| --- |
| server:  port: 8201    spring:  application:  name: microcloud-provider-upload  http:  multipart:  enabled: **true** #启动http上传  max-file-size: 100MB #设置单个文件最大长度  max-request-size: 100MB #设置最大请求上传文件为100MB  file-size-threshold: 1MB #当文件上传达到1MB时进行磁盘写入  location: / #上传临时文件路径    eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: provider-8201.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址  info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$ |

3.建立上传的Rest服务配置:

|  |
| --- |
| **package** com.liuxc.microcloud.rest;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestParam;  **import** org.springframework.web.bind.annotation.RestController;  **import** org.springframework.web.multipart.MultipartFile;  **import** com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;  @RestController  @RequestMapping("/upload")  **public** **class** UploadRest {  @RequestMapping("/index")  @HystrixCommand(fallbackMethod = "uploadFallback") // 熔断机制  **public** String upload(@RequestParam MultipartFile photo) {  **if** (photo != **null**) {  System.***out***.println("\*\*\*\*uploadRest\*\*\*\*" + photo.getOriginalFilename() + ",size:" + photo.getSize());  }  **return** "xx" + System.*currentTimeMillis*() + ".jpg";  }  /\*\*  \* 处理服务器上传失败的回调  \*/  **public** String uploadFallback(@RequestParam MultipartFile photo) {  **return** "nophoto.jpg";  }  } |

4.测试上传微服务，为了方便测试，使用curl的命令完成、当然，这个命令是需要单独配置的;

\*使用curl上传语法：curl-F”参数名称=@文件本地路径”上传地址；

curl -F "photo=@5.jpg" http://liuxc:liuxc@upload-8201.com:8201/upload/index

这个时候如果你的csrf跨站请求配置被关闭了，则会出现如下错误提示信息：

{"timestamp":1526795670524,"status":403,"error":"Forbidden","message":"Could not verify the provided CSRF token because your session was not found.","path":"/upload/index"}请求跨站了。

跨站的访问如果在安全前提下是一定要被禁止的，但是如果牵扯到了微服务的架构问题，那么这个时候不能够进行禁止了。

5.【microcloud-security】修改安全策略，将csfr禁用；

|  |
| --- |
| @Override  **protected** **void** configure(HttpSecurity http) **throws** Exception {  // 所有的请求都需要通过认证后才可以正常进行  http.httpBasic().and().authorizeRequests().anyRequest().fullyAuthenticated().and().csrf().disable();  // 所有rest的服务都需要设置session无状态模式提高效率  http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);  } |

此时表示取消了crsf的验证操作，那么就证明当前的微服务允许跨站请求。那么这个时候文件就可以完成正常处理了。

### 2.9.2使用zuul上传代理

现在已经实现了一个上传的微服务操作，但是现在的微服务是客服端直接调用了指定的微服务的信息来操作的，从实际的开发来讲所有的微服务都应该被zuul代理上传。

1.【microcloud-zuul-gateway-9501】修改application.yml配置文件，追加上传微服务的代理映射地址；

|  |
| --- |
| zuul:  prefix: /liuxc #设置公共前缀  ignored-services: #忽略掉使用应用名称的访问  "\*"  routes:  microcloud-provider-company: /company-proxy/\*\*  microcloud-provider-dept: /dept-proxy/\*\*  microcloud-provider-upload: /upload-proxy/\*\* |

2.下面将利用zuul作为上传微服务的代理操作，可在命令行上使用curl进行配置。

|  |
| --- |
| curl -F  "photo=@5.jpg" http://zuul:zuul@gateway-9501.com:9501/liuxc/upload-proxy/upload/index |

3【microcloud-provider-upload-8201】现在对于当前的上传微服务已经设置好了一个允许最大上传文件为：100MB,现在传一个大于10M的文件，访问路径：

|  |
| --- |
| curl -F "photo=@F:\java视频\pss.zip" http://zuul:zuul@gateway-9501.com:9501/liuxc/upload-proxy/upload/index |

但是上传失败会出现如下的问题：

|  |
| --- |
| org.apache.tomcat.util.http.fileupload.FileUploadBase$FileSizeLimitExceededException: The field photo exceeds its maximum permitted size of 1048576 bytes. |

4.现在通过SpringCloud的官方文档可以发现，在文档之中已经明确的要求了，在上传大文件的时候，那么就必须明确的将我们zuul当中的上传控制交由处理的微服务来进行，所以应该在访问的路径前追加有一个“/zuul/\*\*”映射路径。

|  |
| --- |
| curl -F "photo=@F:\java视频\pss.zip" http://zuul:zuul@gateway-9501.com:9501/zuul/liuxc/upload-proxy/upload/index |

此时就表示当前的上传的代理操作，zuul不在进行限制，而是直接交由目的地的微服务进行。

5.【microcloud-zuul-gateway-9501】修改代理微服务的上传超时设置

|  |
| --- |
| hystrix:  command:  default:  execution:  isolation:  thread:  timeoutInMilliseconds: 60000  ribbon:  ConnectTimeout: 3000  ReadTimeout: 60000 |

由于上传的文件较大，所以需要进行超时时间的配置，才能正常实现上传。

### 2.9.2客服端调用上传微服务

现在为止所有的上传微服务虽然可以调用了，但是却使用的是curl命令完成的，很明显，这样的操作根本不可能在开发之中使用，那么如果要想使用，肯定要通过表单来完成上传处理，也就是说这个时候应该会有一个客服端，调用zuul,然后在有zuul去代理上传微服务。

但是现在非常遗憾的事情是，对于上传的微服务的操作，无法利用Fegin进行接口转换。也无法直接使用RestTemplate做代理操作。也就是说此时如果想要调用，那么唯一的方式就是使用HttpClient完成。

1. 新建上传微服务【microcloud-consumber-upload】

2.修改pom.xml

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!--thymeleaf 模板依赖库 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency>  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  </dependency>  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpmime</artifactId>  </dependency> |

3.修改application.yml文件

|  |
| --- |
| server:  port: 80  spring:  messages:  basename: properties/message |

4.实现微服务上传调用：

|  |
| --- |
| **package** com.lxc.intro.microboot.thymeleaf.web.controller;  **import** java.io.IOException;  **import** java.nio.charset.Charset;  **import** org.apache.http.HttpEntity;  **import** org.apache.http.auth.AuthScope;  **import** org.apache.http.auth.Credentials;  **import** org.apache.http.auth.UsernamePasswordCredentials;  **import** org.apache.http.client.CredentialsProvider;  **import** org.apache.http.client.methods.CloseableHttpResponse;  **import** org.apache.http.client.methods.HttpPost;  **import** org.apache.http.client.protocol.HttpClientContext;  **import** org.apache.http.entity.ContentType;  **import** org.apache.http.entity.mime.MultipartEntityBuilder;  **import** org.apache.http.impl.client.BasicCredentialsProvider;  **import** org.apache.http.impl.client.CloseableHttpClient;  **import** org.apache.http.impl.client.HttpClients;  **import** org.apache.http.util.EntityUtils;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.ResponseBody;  **import** org.springframework.web.multipart.MultipartFile;  **import** com.lxc.intro.microboot.thymeleaf.AbstractBaseController;  @Controller  @RequestMapping("/upload")  **public** **class** UploadController **extends** AbstractBaseController {  **public** **static** **final** String ***UPLOAD\_URL*** = "http://gateway-9501.com:9501/zuul/liuxc/upload-proxy/upload/index";  @RequestMapping(value = "/index", method = RequestMethod.***GET***) // 接受参数，跳转页面  **public** String index() {  **return** "upload/upload\_page";  }  @ResponseBody  @RequestMapping(value = "/uploading", method = RequestMethod.***POST***) // 接受参数，跳转页面  **public** String upload(String name, MultipartFile photo) **throws** IOException {  **if** (photo != **null**) {// 处理上传  CloseableHttpClient httpClient = HttpClients.*createDefault*();// 创建http客服端  CredentialsProvider provider = **new** BasicCredentialsProvider();// 创建认证信息  Credentials credentials = **new** UsernamePasswordCredentials("zuul", "zuul");  provider.setCredentials(AuthScope.***ANY***, credentials);// 认证的范围  HttpClientContext clientContext = HttpClientContext.*create*();  clientContext.setCredentialsProvider(provider);  HttpPost post = **new** HttpPost(***UPLOAD\_URL***);  HttpEntity entity = MultipartEntityBuilder.*create*()  .addBinaryBody("photo", photo.getBytes(), ContentType.*create*("image/jpeg"), "temp.jpg").build();  post.setEntity(entity);  CloseableHttpResponse response = httpClient.execute(post, clientContext);  HttpEntity responseEntity = response.getEntity();    **return** EntityUtils.*toString*(responseEntity, Charset.*forName*("UTF-8"));  }  **return** "upload/index";  }  } |

## 3.0 SpringCloudConfig基础配置

### 3.0.1配置SpringCloudConfig服务端

1.【microcloudconfig】建立一个公共的配置文件：application.yml,随后将这个文件推送到GITHUB厂库中。

Application.yml配置内容如下：

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  spring:  profiles: dev  application:  name: microconfig-test-dev  ---  spring:  profiles: default  application:  name: microconfig-test-default |

2.创建【microcloud-config-7101】项目，修改pom.xml文件，在配置文件中追加有springcloudconfig的依赖服务端。

|  |
| --- |
| <dependencies>  <!--spring-boot 依赖 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <!--EureKa客服端 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!--congfig -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <!-- hystrix 熔断 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>  <!-- 健康监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- springColudConfig 服务端 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency>  <!-- 微服务认证信息 -->  <dependency>  <groupId>com.liuxc</groupId>  <artifactId>microcloud-security</artifactId>  </dependency>  </dependencies> |

3.修改程序的启动类，增加启动注解

|  |
| --- |
| @SpringBootApplication  @EnableConfigServer  **public** **class** Config\_7101\_StartSpringCloundMain **extends** SpringBootServletInitializer {  @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** builder.sources(Config\_7101\_StartSpringCloundMain.**class**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  SpringApplication.*run*(Config\_7101\_StartSpringCloundMain.**class**, args);  }  } |

4.修改application.yml配置文件

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig.git |

5.启动项目，随后就可以通过如下的地址进行访问了：

|  |  |
| --- | --- |
| /{application}-{profile}.yml | <http://liuxc:liuxc@config-7101.com:7101/application-dev.yml> |
| /{label}/{application}-{profile}.yml | <http://liuxc:liuxc@config-7101.com:7101/master/application-dev.yml> |
| /{application}/{profile}[/{label}] | http://liuxc:liuxc@config-7101.com:7101/application/dev/master |

### 3.0.2 SpringCloudConfig 客户端抓取信息

1.准备好配置文件提交到GITHUB的远程库中

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  server:  port: 8101    spring:  profiles: dev  application:  name: microconfig-test-dev  eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: company-8101.com #在信息列表中显示主机名称  prefer-ip-address: true #访问的路径变成ip地址  ---  server:  port: 8102    spring:  profiles: bate  application:  name: microconfig-test-bate    eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eureka:eureka@eureka-7001.com:7001/eureka/,http://eureka:eureka@eureka-7002.com:7002/eureka/,http://eureka:eureka@eureka-7003.com:7003/eureka/ #加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: company-8101.com #在信息列表中显示主机名称  prefer-ip-address: true #访问的路径变成ip地址 |

2.新建一个项目【microcloud-config-client-8201】修改其POM.xml文件追加springCloud-config-client的依赖

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

3.【microcloud-config-client-8201】现在往往都会习惯性的认为应该在application.yml配置文件之中进行SpringCloudConfig服务端的配置，但是在springCloud官方文档里面会发现此时真正应该配置SpringCloudConfig的信息文件应该为“bootstrap.yml”

|  |
| --- |
| 提示：  关于application.yml与bootstrap.yml配置文件说明：  \*”application.yml”对应的是用户级的资源配置项；  \*“bootstrap.yml”对应的是系统级的资源配置，优先级会更高一些。 |

4.所以根据官方文档提示，在资源文件目录下创建“bootstrap.yml”文件

|  |
| --- |
| spring:  cloud:  config:  name: microcloud-config-client #定义要读取的资源文件的名称  profile: dev #定义profile的名称  label: master #定义仓库分支名称  uri: http://config-7101.com:7101/ #访问springConfig server地址  username: liuxc #认证用户名  password: liuxc #认证密码 |

5. 【microcloud-config-client-8201】建立新的服务程序类

|  |
| --- |
| **package** com.liuxc.microcloud.rest;  **import** org.springframework.beans.factory.annotation.Value;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  @RequestMapping("/configClient")  **public** **class** ConfigClientRest {  @Value("${spring.application.name}")  **private** String applicationName; // 应用名称  @Value("${eureka.client.service-url.defaultZone}")  **private** String eurekaServers; // eurka服务注册路径  @RequestMapping("/getConfig")  **public** String getConfig() {  **return** "【服务名称：" + applicationName + "、服务注册地址：" + eurekaServers + "】";  }  } |

6.启动项目访问地址：

<http://company-8101.com:8101/configClient/getConfig>

注意：如果本地的配置与我们远程的配置相同，那么会以bootstrap.yml的远程配置为主。所有bootstrap.yml的优先级一定是要高于applicatin.yml的。

### 3.0.3客户端使用SpringCloudConfig进行配置

现在已经实现了SpringConfig的客户端以及服务端的配置，但是对于现在的客户源端程序而不是只是简单的获取一些信息就够了。应该将之前的许多的配置文件的内容都交给服务端进行保存。

本次将利用SpringCloudConfig实现一个EureKa的注册中心，和一个Dept微服务的信息加载操作；

1.新增两个项目

【microcloud-config-EureKa-client-7004】，

【microcloud-config-dept-client-8001】

编写【microcloud-config-EureKa-client-7004】配置文件提交到GITHUB厂库中，microcloud-config-EureKa-client配置文件如下

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  spring:  profiles: dev  application:  name: microcloud-config-EureKa-client  server:  port: 7004  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7004.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eurekaDev:eurekaDev@eureka-7004.com:7004/eureka/  security:  basic:  enabled: **true** #启动安全认证  user:  name: eurekaDev  password: eurekaDev  ---  spring:  profiles: bate  application:  name: microcloud-config-EureKa-client  server:  port: 7004  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7004.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eurekaBate:eurekaBate@eureka-7004.com:7004/eureka/  security:  basic:  enabled: **true** #启动安全认证  user:  name: eurekaBate  password: eurekaBate |

microcloud-config-dept-client.yml配置文件

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  server:  port: 8001  eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eurekaDev:eurekaDev@eureka-7004.com:7004/eureka/#加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: config-client-8001.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址    mybatis:  config-location: classpath:mybatis/mybatis-config.xml #mybatis的配置文件路径  type-aliases-package: com.liuxc.microcloud.api #配置vo下对象的别名  mapper-locations:  - classpath:mybatis/mapper/\*\*/\*.xml #扫面mapper文件下的路径    spring:  profiles: dev  application:  name: microcloud-config-dept-client  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: com.mysql.jdbc.Driver #配置驱动类  url: jdbc:mysql://localhost:3306/master?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址  username: liuxc #配置数据库连接用户名  password: liuxc #配置数据库连接用户密码  dbcp2:  min-idle: 5 #配置连接池最小连接数  initial-size: 5 #配置连接池初始化连接数  max-total: 12 #配置连接池最大连接数  max-wait-millis: 2000 #配置最大超时断开时间    info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$    ---  server:  port: 8001  eureka:  client: #客服端进行EureKa服务配置  service-url:  defaultZone: http://eurekaBate:eurekaBate@eureka-7004.com:7004/eureka/#加入了认证用户和密码的请求连接  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: config-client-8001.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址    mybatis:  config-location: classpath:mybatis/mybatis-config.xml #mybatis的配置文件路径  type-aliases-package: com.liuxc.microcloud.api #配置vo下对象的别名  mapper-locations:  - classpath:mybatis/mapper/\*\*/\*.xml #扫面mapper文件下的路径    spring:  profiles: bate  application:  name: microcloud-config-dept-client  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: com.mysql.jdbc.Driver #配置驱动类  url: jdbc:mysql://localhost:3306/master?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址  username: liuxc #配置数据库连接用户名  password: liuxc #配置数据库连接用户密码  dbcp2:  min-idle: 5 #配置连接池最小连接数  initial-size: 5 #配置连接池初始化连接数  max-total: 12 #配置连接池最大连接数  max-wait-millis: 2000 #配置最大超时断开时间    info:  app.name: microcloud-provider  app.version: $project.version$  app.artifactId: $project.artifactId$ |

2. 【microcloud-config-EureKa-client-7004】，

【microcloud-config-dept-client-8001】项目中保证都有springCloudConfig的依赖包、

3. 【microcloud-config-EureKa-client-7004】新增“bootstrap.yml”配置文件

|  |
| --- |
| spring:  cloud:  config:  name: microcloud-config-EureKa-client #定义要读取的资源文件的名称  profile: dev #定义profile的名称  label: master #定义仓库分支名称  uri: http://config-7101.com:7101/ #访问springConfig server地址  username: liuxc #认证用户名  password: liuxc #认证密码 |

养成好的习惯在application.yml中配置好服务名称

|  |
| --- |
| spring:  application:  name: microcloud-config-EureKa-client |

4. 【microcloud-config-dept-client-8001】新增bootstrap.yml配置文件

|  |
| --- |
| spring:  cloud:  config:  name: microcloud-config-dept-client #定义要读取的资源文件的名称  profile: dev #定义profile的名称  label: master #定义仓库分支名称  uri: http://config-7101.com:7101/ #访问springConfig server地址  username: liuxc #认证用户名  password: liuxc #认证密码 |

Application.yml配置文件：

|  |
| --- |
| spring:  application:  name: microcloud-config-dept-client |

5.随后启动EureKa项目，访问地址：

<http://eureka-7004.com:7004/>

6.启动【dept-client】项目访问rest服务

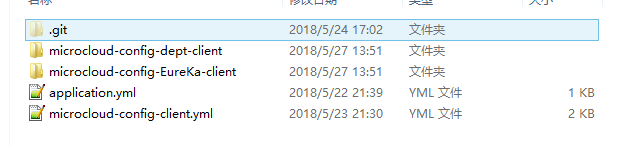
<http://provider-8001.com:8001/dept/findAll>

7.查看EureKa注册的服务信息，发现有【**MICROCLOUD-CONFIG-DEPT-CLIENT**】注册到Eureka之中说明整个springCloudConfig客户端抓取成功。

### 3.0.4 单仓库目录匹配

现在已经实现了一个GIT厂库的配置文件的统一管理操作，但是在这个时候，如果将所有的微服务配置信息都保存在一个厂库里面文件会显得比较多，建议使用一些文件目录来进行保存。

1.【GITHUB】进行仓库的目录配置



2.提交到远程GITHUB厂库之中。

3.修改配置项【microcloud-config-7101】

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig.git  search-paths:  - microcloud-config-EureKa-client  - microcloud-config-dept-client |

启动项目访问测试：

<http://config-7101.com:7101/microcloud-config-EureKa-client-dev.yml>

4. 【microcloud-config-7101】如果说此时微服务有几百个，你这样目录配置也会有几百个，所以为了解决这样的问题，可以使用统配符“\*”进行处理。

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig.git  search-paths:  - microcloud-\* #表示所有以microcloud文件夹开头的目录都可以作为查找目录 |

### 3.0.5 应用仓库自动选择

在之前已经成功的实现了单GIT仓库的配置文件管理处理，但是在SpringCloudConfig之中也提供有多仓库的配置支持。也就是说用户只需要设置要访问的配置应用的名称（spring.application.name），那么就可以根据指定的名称进行配置文件的加载处理了。

1.【GITHUB】为了进行统一管理，在GITHUB上分别创建有两个新的厂库：

microcloud-config-dept-client，

microcloud-config-EureKa-client

在从GITHUB远程服务器上克隆到本地厂库：

git clone [git@github.com:liuxc1/microcloud-config-EureKa-client.git](mailto:git@github.com:liuxc1/microcloud-config-EureKa-client.git)

git clone [git@github.com:liuxc1/microcloud-config-dept-client.git](mailto:git@github.com:liuxc1/microcloud-config-dept-client.git)

随后在每个仓库的克隆所在目录之中各自创建有application.yml配置文件：

之后提交到远程厂库之中。

2.修改【microcloud-config-7101】application.yml配置文件

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/{application} #根据应用名称寻找厂库 |

随后启动SpringCloud服务端程序，随后进行配置文件的访问：

\*访问“microcloud-config-dept-client”

|  |
| --- |
| http://config-7101.com:7101/microcloud-config-dept-client-dev.yml |

\*访问“microcloud-config-EureKa-client”

|  |
| --- |
| http://config-7101.com:7101/microcloud-config-EureKa-client-dev.yml |

现在就相当于一个SpringCloudConfig可以根据操作的应用名称访问不同的GITHUB仓库的内容。

### 3.0.6 仓库匹配模式

仓库匹配模式指的是在进行仓库配置资源获得的时候可以通过一些限制让用户无法获得一些资源。

例如：现在虽然进行了多厂库的配置，但是在多仓库配置分过程之中可以发现用户可以加载所有的profile信息。设置一些规则，让用户只能够加载指定内容。

1.【GITHUB】GITHUB上依然采用与之前的多仓库匹配同样的处理操作；

2.修改【microcloud-config-7101】application.yml配置文件、

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig #当所有的配置找寻不到时选择这个公共的厂库  repos:  microcloud-config-dept-client:  cloneOnStart: **true** #服务器在启动之前克隆  uri: git@github.com:liuxc1/microcloud-config-dept-client.git #指定应用仓库地址  pattern: microcloud-config-dept-client\*/dev\* #只能够访问的路径  microcloud-config-EureKa-client:  uri: git@github.com:liuxc1/microcloud-config-EureKa-client.git #指定应用仓库地址  pattern: microcloud-config-EureKa-client\*/bate\* #只能够访问的路径 |

## 3.1 SpringCloudConfig高级配置

### 3.1.1密钥加密处理

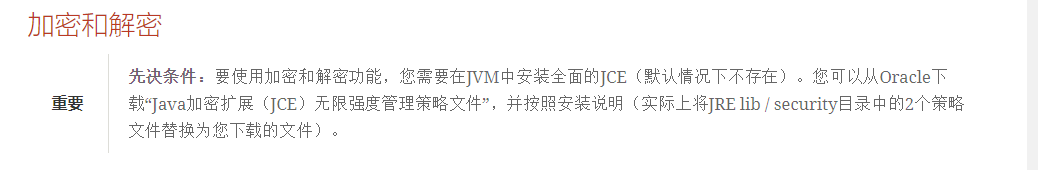
在SpringCloudConfig之中考虑到所有的配置文件都暴露在远程仓库之中，安全性问题所以呢我们提供有我们安全访问机制，可以对一些数据进行加密以及在读取的时候解密的控制。

所谓的密钥的处理指的就是设置一个公共的操作访问密码。而后通过curl命令对要进行访问的数据做一个加密处理即可。

1.【microcloud-config-7101】修改application.yml配置文件，进行密钥的设置：

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig #当所有的配置找寻不到时选择这个公共的厂库  search-paths:  - microcloud-\*  encrypt:  key: liuxc #设置了一个加密访问的密钥 |

2.配置新的JCE使其支持加密解密的支持。



1. 通过curl密令获取加密后的密文

加密：

|  |
| --- |
| curl -X POST http://liuxc:liuxc@config-7101.com:7101/encrypt -d liuxc  密文：  33a7ad06963a9ac5e224231d22ac1c81939f22b05c4a9c45c8e63d9f80fab0e6 |

解密：

|  |
| --- |
| curl -X POST http://liuxc:liuxc@config-7101.com:7101/decrypt -d  33a7ad06963a9ac5e224231d22ac1c81939f22b05c4a9c45c8e63d9f80fab0e6  明文：  liuxc |

3.【GITHUB】将加密后的密码配置到“microcloud-config-dept-client.yml”并且提交到远程库中

|  |
| --- |
| spring:  profiles: dev  application:  name: microcloud-config-dept-client  datasource:  type: com.alibaba.druid.pool.DruidDataSource #配置当前要操作的数据源的类型  driver-class-name: com.mysql.jdbc.Driver #配置驱动类  url: jdbc:mysql://localhost:3306/master?useUnicode=true&amp;characterEncoding=UTF-8&useSSL=false #配置数据库连接地址  username: liuxc #配置数据库连接用户名  password: '{cipher}33a7ad06963a9ac5e224231d22ac1c81939f22b05c4a9c45c8e63d9f80fab0e6' |

4.现在只是保存在GITHUB上的内容进行了加密处理，但是由于有SpringCloudConfig配置中心的存在（配置好了我们的密钥）用户按照传统的模式正常访问即可：

http://config-7101.com:7101/microcloud-config-dept-client-dev.yml

### 3.1.2 keystore加密处理

在加密处理的时候还有一种加密的方式会更加的方便，及时直接利用JKS操作的模式来完成。这种方式会更加的安全，因为只需要一个jsk配置文件即可实现加密与解密。

1【本地系统】需要生成一个进行加密处理的key信息（主要是得到\*.jks文件）

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

将生成好的server.jks文件拷贝到【microcloud-config-7101】resources目录下

2.修改【microcloud-config-7101】application.yml配置文件，增加jks文件配置

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig #当所有的配置找寻不到时选择这个公共的厂库  search-paths:  - microcloud-\*  encrypt:  keyStore:  location: classpath:/server.jks #jks文件路径  password: liuxiancheng #keystore 密码  alias: mytestkey #别名  secret: changeit #keypass |

3.修改pom.xml文件

|  |
| --- |
| <build>  <finalName>microcloud-config</finalName>  <resources>  <resource>  <!--配置文件的路径 -->  <directory>src/main/resources</directory>  <filtering>false</filtering>  </resource>  </resources>  </build> |

4.测试加密解密：

|  |  |
| --- | --- |
| 加密 | curl -X POST http://liuxc:liuxc@config-7101.com:7101/encrypt -d liuxc |
| 解密 | curl -X POST http://liuxc:liuxc@config-7101.com:7101/decrypt -d |

5.将生成的密文密匙加密处理方式一样更新到github上在浏览器访问测试

### 3.1.3 SpringSpringCloudConfig高可用

1.新创建一个eureka节点【microcloud-EureKa-config-7005】修改application.yml配置文件

|  |
| --- |
| server:  port: 7005  eureka:  instance: #EureKa 实例的定义  hostname: eureKa-7005.com #定义EureKa实例所在的主机名称  client: #在默认设置下，Eureka服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为。  register-with-eureka: **false** #false 表示当前的服务器不注册到eureKa  fetch-registry: **false** #不从eureKa服务中获取信息  service-url:  defaultZone: http://eureka:eureka@eureka-7005.com:7005/eureka/ #加入了认证用户和密码的请求连接  security:  basic:  enabled: **true** #启动安全认证  user:  name: eureka  password: eureka |

2.【microcloud-config-\*】修改pom.xml配置文件，追加eureka的pom文件

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

3. 【microcloud-config-\*】修改application.yml配置文件，主要是进行eureka配置处理

|  |
| --- |
| server:  port: 7101  spring:  application:  name: microcloud-config  cloud: #进行springCloud相关配置  config: #进行springCloudConfig相关配置  server: #设置服务的连接地址  git: #设置Git厂库的连接地址  uri: git@github.com:liuxc1/microcloudconfig #当所有的配置找寻不到时选择这个公共的厂库  search-paths:  - microcloud-\*  #eureka客服端配置  eureka:  client:  service-url:  defaultZone: http://eureka:eureka@eureka-7005.com:7005/eureka  instance:  lease-renewal-interval-in-seconds: 30 #设置心跳间隔时间，默认是30秒  lease-expiration-duration-in-seconds: 90 #心跳间隔周期，默认是90秒  instance-id: config-7101.com #在信息列表中显示主机名称  prefer-ip-address: **true** #访问的路径变成ip地址  encrypt:  keyStore:  location: classpath:/server.jks #jks文件路径  password: liuxiancheng #keystore 密码  alias: mytestkey #别名  secret: changeit #keypass |

4.【microcloud-config-\*】修改启动类，追加上EureKa启动注解。

5.【microcloud-config-client-8201】修改bootstrap.yml配置文件，此时的配置文件需要通过eureka获取config服务信息

|  |
| --- |
| spring:  cloud:  config:  name: microcloud-config-client #定义要读取的资源文件的名称  profile: bate #定义profile的名称  label: master #定义仓库分支名称  #uri: http://config-7101.com:7101/ #访问springConfig server地址  username: liuxc #认证用户名  password: liuxc #认证密码  discovery:  enabled: **true** #启动在配置中心加载配置文件  service-id: MICROCLOUD-CONFIG #eureka配置中心的注册名称  eureka:  client: #目的是为了让客户端通过eureka找到SpringCloudConfig微服务  register-with-eureka: **false**  service-url:  defaultZone: http://eureka:eureka@eureka-7005.com:7005/eureka/ |

6.【microcloud-config-client-8201】在启动类上追加Eureka客户端启动注解。

如果此时某一个SpringCloudConfig的微服务被停止了，那么则在启动的时候会进行其他配置微服务的信息加载。那么现在就实现了SpringCloudConfig基于EureKa的高可用处理。

### 3.1.4 springCloudBus服务总线

在整个SpringCloudCofigs设计之中，主要考虑的是所有微服务配置文件过多的统一管理问题，在整个的设置之上还存在一点点小小的缺陷。就是配置信息的及时更新；

在实现springCloudConfigBus的过程之中必须要有一个明确的消息服务组件。而这种服务组件一般会有两类处理模式：RabbitMQ，kafka.

1.新建一个配置文件提交到github,配置文件如下：

|  |
| --- |
| **spring**:  **profiles**:  **active**:  - dev --- **server**:  **port**: 8101 **spring**:  **profiles**: dev  **application**:  **name**: microconfig-config-bus-client **eureka**:  **client**: *#客服端进行EureKa服务配置* **service-url**:  **defaultZone**: http://eureka:eureka@eureka-7004.com:7004/eureka/  **instance**:  **lease-renewal-interval-in-seconds**: 30 *#设置心跳间隔时间，默认是30秒* **lease-expiration-duration-in-seconds**: 90 *#心跳间隔周期，默认是90秒* **instance-id**: client-8101.com *#在信息列表中显示主机名称* **prefer-ip-address**: true *#访问的路径变成ip地址* **info**:  **app.name**: microconfig-config-dev-1  **company.name**: www.liuxc.com  **app.version**: v.0.0.1  --- **server**:  **port**: 8102 **spring**:  **profiles**: bate  **application**:  **name**: microconfig-config-bus-client **eureka**:  **client**: *#客服端进行EureKa服务配置* **service-url**:  **defaultZone**: http://eureka:eureka@eureka-7004.com:7004/eureka/  **instance**:  **lease-renewal-interval-in-seconds**: 30 *#设置心跳间隔时间，默认是30秒* **lease-expiration-duration-in-seconds**: 90 *#心跳间隔周期，默认是90秒* **instance-id**: client-8102.com *#在信息列表中显示主机名称* **prefer-ip-address**: true *#访问的路径变成ip地址* **info**:  **app.name**: microconfig-config-bate-1  **company.name**: www.liuxc.com  **app.version**: v.0.0.1 |

2.新建【microcloud-config-bus-server-7201】项目

3.修改【microcloud-config-bus-server-7201】pom.xml 追加jar包

|  |
| --- |
| <**dependencies**>  <!--spring-boot web -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <!--config 服务端-->  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-config-server</**artifactId**>  </**dependency**>  <!--EureKa客服端 -->  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  </**dependency**>  <!--RabbitMQ-->  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-bus-amqp</**artifactId**>  </**dependency**>  <!-- 健康监控 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**>  <!--控制权限模块-->  <**dependency**>  <**groupId**>com.liuxc.www</**groupId**>  <**artifactId**>microcloud-security</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**>  <!-- 热部署模块 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**optional**>true</**optional**> <!-- 这个需要为 true 热部署才有效 -->  </**dependency**>  <!-- 测试模块 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**> </**dependencies**> |

4.修改application.yml配置项，增加我们RabbitMQ的配置项

|  |
| --- |
| **server**:  **port**: 7201  **spring**:  **application**:  **name**: microcloud-config-server  **cloud**:  **config**:  **server**:  **git**:  **uri**: git@github.com:liuxc1/microcloudconfig.git *#设置GIT厂库的连接地址* **search-paths**: *#采用单厂库多目录的方式获取配置文件信息* - microcloud-\*  **rabbitmq**: *#集成RabbitMQ作为消息服务总线处理* **host**: 192.168.20.85  **port**: 5672  **virtual-host**: /my-virtual-hosts  **username**: java  **password**: java   **eureka**:  **client**:  **service-url**:  **defaultZone**: http://eureka:eureka@eureka-7004.com:7004/eureka/  **instance**:  *#在信息列表中显示主机名称* **instance-id**: www.lxc.com  *#访问的路径变成ip地址* **prefer-ip-address**: **true** |

5.启动项目，如果启动日志中打印出如下信息：就具备的bus的功能。

|  |
| --- |
| 2019-01-15 13:19:54.457 INFO 8460 --- [ restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "{[/bus/refresh],methods=[POST]}" onto public void org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String) |

6.新建一个【microcloud-config-bus-client-8201】项目修改pom.xml文件，追加config-server项目的相同依赖包

|  |
| --- |
| <**dependencies**>  <!--spring-boot web -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <!--config 客户端-->  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-config</**artifactId**>  </**dependency**>  <!--EureKa客服端 -->  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  </**dependency**>  <!--RabbitMQ-->  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-bus-amqp</**artifactId**>  </**dependency**>  <!-- 健康监控 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**>  <!--控制权限模块-->  <**dependency**>  <**groupId**>com.liuxc.www</**groupId**>  <**artifactId**>microcloud-security</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**>  <!-- 热部署模块 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**optional**>true</**optional**> <!-- 这个需要为 true 热部署才有效 -->  </**dependency**>  <!-- 测试模块 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**> </**dependencies**> |

8.修改bootstrap.yml配置文件，追加RabbitMQ项目配置项

|  |
| --- |
| **spring**:  **cloud**:  **config**:  **name**: microcloud-config-bus-client *#需要从config server 端 抓取配置的文件名称* **profile**: bate *#定义使用配置文件里面的那个配置* **label**: master *#定义git厂库分支* **username**: liuxc *#定义访问config server 的 认证* **password**: liuxc  **discovery**: *#通过eureka 服务id 找到config server* **enabled**: **true  service-id**: MICROCLOUD-CONFIG-BUS-SERVER  **rabbitmq**: *#集成RabbitMQ作为消息服务总线处理* **host**: 192.168.20.85  **port**: 5672  **virtual-host**: /my-virtual-hosts  **username**: java  **password**: java |

9.新建一个configInfo配置类，该类的主要功能是可以进行配置的刷新获取：

|  |
| --- |
| @Component @RefreshScope //如果没有该注解，则通过springCloudConfigBus 无法实现配置刷新服务 **public class** InfoConfig {  //将所有需要动态的配置写在一个类之中  @Value("${info.app.name}")  **private** String appName;   @Value("${info.company.name}")  **private** String companyName;   @Value("${info.app.version}")  **private** String appVersion;   **public** String getAppName() {  **return** appName;  }   **public** String getCompanyName() {  **return** companyName;  }   **public** String getAppVersion() {  **return** appVersion;  } } |

10.新建一个InfoConfigRest服务类

|  |
| --- |
| @RestController @RequestMapping("configClient") **public class** ConfigClientRest {   @Autowired  **private** InfoConfig config;   @RequestMapping("/getConfig")  **public** String getConfig() {   **return** "【服务名称：" + **this**.config.getAppName() + "、公司名称：" + **this**.config.getCompanyName() + "、版本信息：" + **this**.config.getAppVersion() + "】";  }  } |

11.启动项目，访问连接：<http://localhost:8102/configClient/getConfig>

12.【GITHUB】修改microcloud-config-elient.yml配置文件，修改beta部分内容：

|  |
| --- |
| **info**:  **app.name**: microconfig-config-bate-2  **company.name**: www.liuxc.com  **app.version**: v.0.0.2 |

此时GITHUB 上的配置文件已经发送了改变，但是客户端并不知道GITHUB上的这次操作，所以其显示的内容，还是原始内容，如果要想加载新的内容，则就必须进行服务总线的刷新。

13.如果要想进行总线服务刷新，用户必须具备有“ACTUATOR”角色，所以需要修改用户的配置程序类

|  |
| --- |
| */\*\*  \* 配置security 认证用户  \*/* @Autowired **public void** configGlobal(AuthenticationManagerBuilder builder) **throws** Exception {  builder.inMemoryAuthentication().withUser("liuxc").password("liuxc").roles("USER","ACTUATOR")  .and().withUser("admin").password("admin").roles("USER", "ADMIN"); } |

14.本地系统通过curl命名发出一个post请求访问指定bus微服务。

|  |
| --- |
| curl -X POST http://liuxc:liuxc@localhost:7201/bus/refresh |

## 3.2springCloudStream

## 3.3springCloudSleuth

微服务可以将整个系统拆分为无数个子系统，于是这样一来就有可能出现几种可怕的场景比如说代码的调试

Sleuth技术就可以实现微服务的调用跟踪，也就是说它可以自动的形成一个调用的连接线，通过连接线使得开发者可以轻松的找到所有微服务的关系。同时也可以获取微服务的所耗费的时间。

### 3.3.1springCloudSleuth基本使用

SpringCloudSleuth使用的核心组件在于Twitter推出的zipkin监控组件，所以本次的配置的模块一定要包含zipkin相关配置。

1.新建【microcloud-sleuth】项目，修改pom.xml配置文件

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-sleuth</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-zipkin</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>io.zipkin.java</**groupId**>  <**artifactId**>zipkin-server</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>io.zipkin.java</**groupId**>  <**artifactId**>zipkin-autoconfigure-ui</**artifactId**>  </**dependency**>   <!--spring-boot web -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <!-- 健康监控 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**>  <!-- 热部署模块 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**optional**>true</**optional**> <!-- 这个需要为 true 热部署才有效 -->  </**dependency**>  <!-- 测试模块 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**> </**dependencies**> |

2.修改【microcloud-sleuth】application.yml配置文件

|  |
| --- |
| **server**:  **port**: 8501  **spring**:  **application**:  **name**: microcloud-sleuth |

3.修改程序启动类

|  |
| --- |
| @SpringBootApplication @EnableZipkinServer **public class** Sleuth\_8501\_StartSpringCloudApplication {  **public static void** main(String[] args) {   SpringApplication.*run*(Sleuth\_8501\_StartSpringCloudApplication.**class**, args);  } } |

4.修改

【microcloud-consumer-product-feign，microcloud-provider-product-8001，microcloud-zuul-gateway-9501】pom.xml文件追加zipkin配置。

|  |
| --- |
| <!--调用服务追终--> <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-zipkin</**artifactId**> </**dependency**> |

5.修改

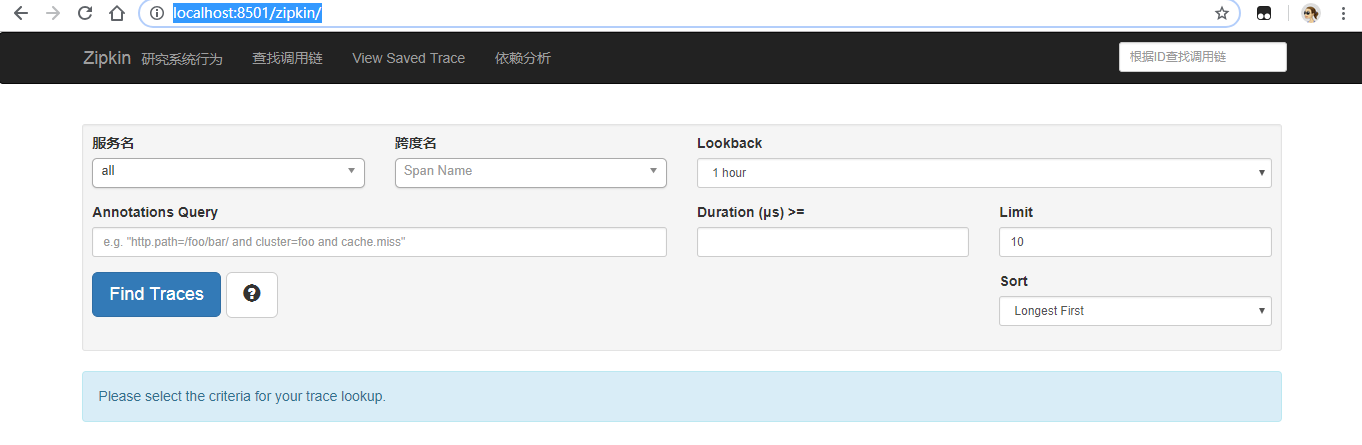
【microcloud-consumer-product-feign，microcloud-provider-product-8001，microcloud-zuul-gateway-9501】application.yml文件

|  |
| --- |
| **spring**:  **application**:  **name**: microcloud-consumer-product-feign  **zipkin**:  **base-url**: http://localhost:8501 *#所有数据提交到此服务中* **sleuth**:  **sampler**:  **percentage**: 1.0 *#定义抽样比例 默认为0.1* |

一定要有我们每一个微服务的名称，这样会比较好观测程序的执行轨迹。

6.依次启动所有的服务：sleuth、zuul、provider、consumer

访问地址：http://localhost:8501/zipkin/



### 3.3.2springCloudSleuth信息采集