SpringCloud开发详解

# 分布式开发与SpringCloud简介

## 分布式开发简介

当有第一台电脑之后就开始有人去想到如何将不同的电脑进行网络连接，而有了网络连接之后对于项目的开发就开始有了所谓的分布式设计，从最初的设计来讲，重要的数据肯定是要保存多分的。所以对于分布式的要求就逐步形成了。

实际上经过了这么多年的发展，很多的人开始认识到一个问题：单一的主机所能够处理的数据量是有限的，所以为了得到更高性能的设计方案一定要采用多台电脑，可是这多台主机之间该如何进行分配呢？

于是对于分布式的开发设计，最初就有了以下的几点思考？

* 分布式的开发如何可以让代码更加安全？
* 分布式开发的时候如何进行有效的通讯？
* 在进行分布式处理的时候到底如何进行程序的功能划分？
* 现在对于很多有经验的开发工程师实际上都已经清楚的知道了现在项目开发之中要进行的分布式的设计：
* WEB集群：考虑到多用户并发访问的处理速度；
* 业务中心：在进行一些庞大的项目设计过程之中，应该有更加完善的业务处理，这样所有的客户端（服务器）直接调用这些业务中心的操作就可以完成具体的功能；
* 数据库集群：解决了数据的存储问题，以及数据的分片管理。

对于分布式的项目开发按照历史的发展经历有如下的一些技术：

* CORBA：公共对象请求代理架构，它是一种开发的标准，而且也是许多语言都支持的开发标准；
* RMI（远程方法调用）：该技术是SUN提出的，该技术出现的最大特征是希望可以与CORBA进行市场竞争，于是这个时候出现了一个问题：很多的公司不认可这项技术；
* 后来SUN的设计师发现，RMI很好用，但是不如CORBA广泛，于是开始思考能否基于CORBA做一些更好的设计呢？这样就产生了一个新的协议：RMI-IIOP协议，而这个协议用在了EJB技术上；

EJB留给世界上的只是它优秀的理论，和它槽糕的实现，并且这个理论被一些开源框架无线的扩充与实现着。

.NET的出现，而后又因为J#的问题，微软和SUN就彻底决裂了，后来这个行业就乱了，因为有两套企业架构，公司就面临选择，于是尴尬的局面出现了，应该选择谁？后来软件行业继续推广，同时出现了一款足以改变世界的优秀语言：XML，很多的公司就觉得应该用XML作为数据交换的基础，这个时候著名的软件机构：WebService（WEB服务）。

如果你的项目之中要想使用WebService技术进行开发，则有如下问题：

* 速度太慢了，处理的速度不行；
* 如果要想采用远程接口的方式调用，则要利用开发工具生成一堆的工具类代码；

后来2005年的时候WebService在整个世界上继续发酵，形成了又一大核心神器：SOA（面向服务架构）。它提出了一个企业服务总线（ESB）的概念。

SOA提出的服务总线又称为了新一代分布式的讨论需求，人们发现如果将所有的服务统一管理起来，就称为了服务总线，那么所有的开发者直接调用里面的服务就完成一些功能。后来又不断出现了许多的RPC开发技术，其中在国内具有代表性的dubbo开发技术。不过这个时候成长最快的是Rest，因为JSON的广泛引用，就如同最初XML技术造就了WebService一样，JSON造就了Rest服务，人们认为我们的操作应该更加简单一些，Rest还是一个未成型的标准还在发展着，而随着Rest技术的广泛认可，Spring终于得到了一个新的机会：利用Rest进行RPC技术实现，这样的操作速度很快，而且占用的带宽要少。在SpringCloud之中就真正的将整个的Rest作为了RPC实现技术，并且这一技术已经开始出现有行业的统一之势。而且SpringCloud也依照于SpringBoot开发技术，可以实现项目的打包发布以及单独运行，这一点都符合于当前云时代的开发要求。

## SpringCloud开发架构

SpringCloud整体核心架构只有一点：Rest服务，也就是说在整个SpringCloud配置过程之中，所有配置处理都是围绕着Rest完成的，在整个Rest处理之中，一定要有两个端：服务的提供者（Provider）、服务的消费者（Consumer）。

既然SpringCloud的核心是Restful结构，那么如果要想更好的去使用Rest这些微服务还需要考虑如下几个问题？

1. 所有的微服务的地址一定会非常的多，所以为了统一管理这些地址信息，也为了可以即使的告诉用户那些服务不可用，所以应该准备一个分布式的注册中心，并且该注册中心应该支持有HA机制，为了告诉并且方便的进行所有服务的注册操作，在SpringCloud里面提供有一个Eureka的注册中心。
2. 对于整个的WEB端的架构（SpringBoot实现）可以轻松方便的进行WEB程序的编写，而后利用Nginx或Apache实现负载均衡处理，但是你WEB端出现了负载均衡，那么业务端呢？应该也提供有多个业务端进行负载均衡。那么这个时候就需要将所有需要参与到负载均衡的业务端在Eureka之中进行注册。  
    在进行客户端使用Rest架构调用的时候，往往都需要有一个调用地址，即使现在使用了Eureka作为注册中心，那么它也需要有一个明确的调用地址，可是所有的操作如果都利用调用地址的方式来处理，程序的开发者最方便应用的工具是接口，所以现在就希望可以将所有 的Rest服务的内容以接口的方式出现调用，所以它又提供了一个feign技术，利用此技术可以伪造接口。
3. 在进行整体的微架构设计的时候由于牵扯到的问题还是属于RPC，所以就必须考虑熔断处理机制，实际上所谓的熔断就好比生活之中使用的保险丝一样，有了保险丝在一些设备出现了故障之后依然可以保护家庭电器可以正常使用，如果说现在有若干个微服务，并且这些微服务之间允许互相调用，例如：A微服务调用了B微服务、B的微服务又调用了C的微服务。  
    如果要在实际的项目设计过程之中没有处理好熔断机制，那么就会产生雪崩效应，所以为了防止这样的问题出现，SpringCloud里面提供有一个Hystrix熔断处理机制，以保证某一个微服务即使出现了问题之后依然可以正常使用。
4. 在进行微服务访问的时候还有一点是非常可怕的。  
    通过Zuul的代理用户只需要知道指定的路由的路径就可以访问指定的微服务的信息，这样更好的体现了Java中的“key=value”的设计思想，而且所有的微服务通过Zuul进行代理之后也可以更加合理的进行名称的隐藏。
5. 在SpringBoot学习的时候一直强调过一个问题：在SpringBoot里面强调的是要给“零配置”的概念，本质在于不需要编写任何的配置文件，但是事实上这一点并没有完全的实现，因为在整体的设计里面，依然会提供有application.yml(application.properties)配置文件，那么如果在微服务的创建之中，一定会有成百上千个微服务的信息出现，于是这些配置文件的管理就成为了问题，例如：你突然有一天你的主机要进行机房变更，所有的服务的IP地址都可能发生改变，这样对于程序的维护是不方便的，为了解决这样的问题，在SpringCloud设计的时候提供有一个SpringCloudConfig的程序组件，利用这个组件就可以直接基于GIT或SVN来进行配置文件的管理。

在整体设计上SpringCloud更好的实现了RPC的架构设计，而且使用了Rest作为通讯的基础，这一点是它的成功之处，同时由于大量的使用了netfix公司的产品技术，所以这些技术也有可靠的保证。

# 搭建Rest服务

## 搭建项目环境

对于现在的项目创建一个：microcloud的整体父pom项目，那么随后为了方便管理，将创建其三个子模块：

* microcloud-api模块，作为公共的信息导入配置模块；
* microcloud-provider-dept-8001：作为服务提供者，该提供者负责使用mybatis与数据库交互；
* microcloud-consumer-80：作为微服务调用的客户端使用。

1. 创建一个新的maven项目：microcloud；
2. [microcloud]修改pom.xml文件，主要追加springcloud与springboot两个开发包的依赖关系；

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.kuhnwei</groupId>  <artifactId>microcloud</artifactId>  <version>1.0-SNAPSHOT</version>  <packaging>pom</packaging>  <properties>  <jdk.version>1.8</jdk.version>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.SR1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.5.4.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <finalName>microcloud</finalName>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>${jdk.version}</source>  <target>${jdk.version}</target>  <encoding>${project.build.sourceEncoding}</encoding>  </configuration>  </plugin>  </plugins>  </build> </project> |

1. microcloud-api，建立一个api的公共模块，该模块的主要功能就是提供有一个公共处理类，本次预计建立一个Dept数据表，里面的字段：deptno(Long)、dname(String)、loc（保存的数据库的名字）；

|  |
| --- |
| package com.kuhnwei.domain; import java.io.Serializable; public class Dept implements Serializable {  private Long deptno;  private String dname;  private String loc; } |

1. microcloud-provider-dept-8001，创建一个Rest提供者的项目模块，在这个模块里面主要定义要使用的数据库脚本：

|  |
| --- |
| **DROP** DATABASE IF **EXISTS** "mohist8001"; **CREATE** DATABASE mohist8001 **CHARACTER SET** UTF8; USE mohist8001; **CREATE TABLE** dept (  deptno BIGINT AUTO\_INCREMENT,  dname **VARCHAR**(50),  loc **VARCHAR**(50),  **CONSTRAINT** pk\_deptno **PRIMARY KEY**(deptno) ); **INSERT INTO** dept(dname, loc) **VALUES** ('开发部', database()); **INSERT INTO** dept(dname, loc) **VALUES** ('财务部', database()); **INSERT INTO** dept(dname, loc) **VALUES** ('市场部', database()); **INSERT INTO** dept(dname, loc) **VALUES** ('后勤部', database()); **INSERT INTO** dept(dname, loc) **VALUES** ('公关部', database()); |

由于在整个微服务里面需要进行负载均衡的操作，所以本次在使用的时候加入了数据库的名称信息。

## 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mybatis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-dept-8001，修改pom.xml配置文件，追加相关的以来程序支持包：

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.kuhnwei</groupId>  <artifactId>microcloud-api</artifactId>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jetty</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency> </dependencies> |

1. microcloud-provider-dept-8001，创建一个DeptDAO数据操作接口，这个接口里面将提供有三个数据操作方法：

|  |
| --- |
| package com.kuhnwei.microcloud.dao; import com.kuhnwei.domain.Dept; import org.apache.ibatis.annotations.Mapper; import java.util.List; @Mapper public interface DeptDAO {  boolean save(Dept dept);  Dept get(Long id);  List<Dept> list(); } |

1. microcloud-provider-dept-8001，修改application.yml配置文件，追加mybatis和服务的相关配置信息：

|  |
| --- |
| **server:  port:** 8001 **mybatis:  config-location:** classpath:mybatis/mybatis.cfg.xml  **type-aliases-package:** com.kuhnwei.domain  **mapper-locations:** - classpath:mybatis/mapper/\*\*/\*.xml **spring:  datasource:  type:** com.alibaba.druid.pool.DruidDataSource  **driver-class-name:** org.gjt.mm.mysql.Driver  **url:** jdbc:mysql://localhost:3306/mohist8001  **username:** root  **password:   dbcp2:  min-idle:** 5  **initial-size:** 5  **max-total:** 5  **max-wait-millis:** 200 |

1. microcloud-provider-dept-8001，定义src/main/resources/mybatis/mybatis.cfg.xml配置文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd" > <configuration>  <settings>  <setting name="cacheEnabled" value="true"/>  </settings> </configuration> |

1. microcloud-provider-dept-8001，定义src/main/resource/mybatis/mapper/com.kuhnwei.DeptDAO.xml配置文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd" > <mapper namespace="com.kuhnwei.microcloud.dao.DeptDAO">  <select id="get" resultType="Dept" parameterType="long">  SELECT deptno, dname, loc FROM dept WHERE deptno=#{deptno};  </select>  <select id="list" resultType="Dept">  SELECT deptno, dname, loc FROM dept;  </select>  <insert id="save" parameterType="Dept">  INSERT INTO dept(dname, loc) VALUES (#{dname}, database());  </insert> </mapper> |

1. microcloud-provider-dept-8001，建立DeptService接口，做业务实现；

|  |
| --- |
| package com.kuhnwei.microcloud.service; import com.kuhnwei.domain.Dept; import java.util.List;  public interface DeptService {  Dept get(long id);  boolean save(Dept dept);  List<Dept> list(); } |
| package com.kuhnwei.microcloud.service.impl;  import com.kuhnwei.domain.Dept; import com.kuhnwei.microcloud.dao.DeptDAO; import com.kuhnwei.microcloud.service.DeptService; import org.springframework.stereotype.Service; import javax.annotation.Resource; import java.util.List;@Service public class DeptServiceImpl implements DeptService {  @Resource  private DeptDAO deptDAO;   @Override  public Dept get(long id) {  return this.deptDAO.get(id);  }  @Override  public boolean save(Dept dept) {  return this.deptDAO.save(dept);  }  @Override  public List<Dept> list() {  return this.deptDAO.list();  } } |

1. microcloud-provider-dept-8001，定义程序的运行主类

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication;@SpringBootApplication public class Dept\_8001\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Dept\_8001\_StartSpringCloudApplication.class, args);  } } |

1. microcloud-provider-dept-8001，进行业务接口测试编写：

|  |
| --- |
| package com.kuhnwei.test; import com.kuhnwei.domain.Dept; import com.kuhnwei.microcloud.Dept\_8001\_StartSpringCloudApplication; import com.kuhnwei.microcloud.service.DeptService; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringJUnit4ClassRunner; import org.springframework.test.context.web.WebAppConfiguration; import javax.annotation.Resource; @SpringBootTest(classes = Dept\_8001\_StartSpringCloudApplication.class) @RunWith(SpringJUnit4ClassRunner.class) @WebAppConfiguration public class DeptServiceTest {  @Resource  private DeptService deptService;  @Test  public void testGet() {  System.*out*.println(this.deptService.get(1));  }  @Test  public void testSave() {  Dept dept = new Dept();  dept.setDname("测试部-" + System.*currentTimeMillis*());  System.*out*.println(this.deptService.save(dept));  }  @Test  public void testList() {  System.*out*.println(this.deptService.list());  } } |

1. microcloud-provider-dept-8001，建立DeptRest服务类；

|  |
| --- |
| package com.kuhnwei.microcloud.rest; import com.kuhnwei.domain.Dept; import com.kuhnwei.microcloud.service.DeptService; import org.springframework.web.bind.annotation.\*; import javax.annotation.Resource; @RestController public class DeptRest {  @Resource  private DeptService deptService;  @RequestMapping(value = "/dept/get/{id}", method = RequestMethod.*GET*)  public Object get(@PathVariable("id") long id) {  return this.deptService.get(id);  }  @RequestMapping(value = "/dept/save", method = RequestMethod.*POST*)  public Object save(@RequestBody Dept dept) {  return this.deptService.save(dept);  }   @RequestMapping(value = "/dept/list", method = RequestMethod.*GET*)  public Object list() {  return this.deptService.list();  } } |

1. 修改hosts配置文件，追加一个映射路径。

|  |
| --- |
| 127.0.0.1 dept-8001.com |

1. 观察Rest服务能否正常提供：
   1. 调用get操作：dept-8001.com:8001/dept/get/1;
   2. 调用list操作：dept-8001.com:8001/dept/list;

## 客户端调用微服务

1. 创建一个maven的新模块：microcloud-consumer-80;
2. microcloud-consumer-80，修改application.yml配置文件：

|  |
| --- |
| **server:  port:** 80 |

1. microcloud-consumer-80，创建一个Rest的配置程序类，主要进行RestTemplate类对象创建；

|  |
| --- |
| package com.kuhnwei.microcloud.config; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; import org.springframework.web.client.RestTemplate;@Configuration public class RestConfig {  @Bean  public RestTemplate getRestTemplate() {  return new RestTemplate();  } } |

1. microcloud-consumer-80，创建一个控制器，为了简化处理，本次不再进行页面定义了，所有服务结果都使用Rest返回；

|  |
| --- |
| package com.kuhnwei.microcloud.controller; import com.kuhnwei.domain.Dept; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController; import org.springframework.web.client.RestTemplate; import javax.annotation.Resource; import java.util.List;@RestController(value = "consumer/dept") public class ConsumerDeptController {  public static final String *DEPT\_GET\_URL* = "http://dept-8001.com:8001/dept/get/";  public static final String *DEPT\_LIST\_URL* = "http://dept-8001.com:8001/dept/list";  public static final String *DEPT\_SAVE\_URL* = "http://dept-8001.com:8001/dept/save";   @Resource  private RestTemplate restTemplate;   @RequestMapping(value = "get")  public Object getDept(long id) {  Dept dept = this.restTemplate.getForObject(*DEPT\_GET\_URL* + id, Dept.class);  return dept;  }   @RequestMapping(value = "list")  public Object listDept() {  List<Dept> allDepts = this.restTemplate.getForObject(*DEPT\_LIST\_URL*, List.class);  return allDepts;  }  @RequestMapping(value = "save")  public Object saveDept(Dept dept) {  Boolean flag = this.restTemplate.postForObject(*DEPT\_SAVE\_URL*, dept, Boolean.class);  return flag;  } } |

1. microcloud-consumer-80，编写启动程序类

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication;@SpringBootApplication public class Consumer\_80\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Consumer\_80\_StartSpringCloudApplication.class, args);  } } |

1. 修改hosts配置文件，追加访问控制路径：

|  |
| --- |
| 127.0.0.1 client.com |

1. 进行代码测试调用：
   1. 调用数据返回：<http://client.com/consumer/dept/get?id=1>;
   2. 测试列表数据：<http://client.com/consumer/dept/list>;
   3. 测试数据增加：<http://client.com/consumer/dept/save?dname=WEB>测试

# SpringSecurity安全认证

所有的Rest服务最终都是暴露在公网上的，也就是说如果你的rest服务属于一些你自己公司的私人业务，这样的结果会直接导致你信息的泄露，所以对于Rest访问，安全性是首要的因素。

## 配置安全验证

如果要想进行安全的验证处理，那么首先一定要现在服务的提供方上进行处理。

1. microcloud-provider-dept-8001，修改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> |

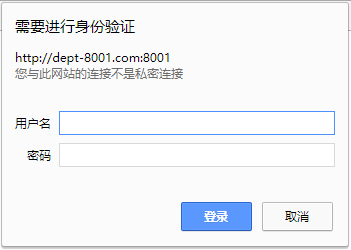
如果你现在配置了安全框架，则在启动时会出现有如下的一个提示信息：

|  |
| --- |
| Using default security password: b95e2886-3618-4a5b-883f-084855b67f88 |

1. microcloud-provider-dept-8001，修改application.yml配置文件，进行安全的用户名配置；

|  |
| --- |
| **security:  basic:  enabled:** true *# 启动SpringSecurity的安全配置项* **user:  name:** mohist *# 认证用户名* **password:** 650901 *# 认证密码* **role:** *# 授权角色* - USER |

随后在项目之中访问Rest服务接口：<http://dept-8001.com:8001/dept/list>，此时在访问的时候会直接询问用户要求用户输入用户名以及面。



这个时候又一种更简化的方法进行内容的输入：http://mohist:650901@dept-8001.com:8001/dept/list;

## 服务消费端处理

在实际的开发之中，对于Rest服务提供者是不可能被用户直接进行访问的，于是肯定需要有一个Rest客户端（WEB端、SpringBoot）进行调用，可是现在Rest提供者的服务上有了认证信息，那么该如何访问呢？

|  |
| --- |
| public static final String *DEPT\_GET\_URL* =  "http://mohist:650901@dept-8001.com:8001/dept/get/"; |

如果这个时候在Rest客户端上直接使用用户名和密码做加密处理，那么根本无法进行访问，此时会出现有401的错误代码，因为认证出现了错误。之所以无法访问，是以为所有的认证的处理操作，应该以头信息的模式来进行处理。而后使用Base64进行加密处理后才可以得到有一个正确的访问路径。

1. microcloud-consumer-80，修改RestConfig配置类，在这个配置类上追加有新的Bean配置项；

|  |
| --- |
| package com.kuhnwei.microcloud.config; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; import org.springframework.http.HttpHeaders; import org.springframework.web.client.RestTemplate; import java.nio.charset.Charset; import java.util.Base64;@Configuration public class RestConfig {   // 要进行一个HTTP头信息配置  @Bean  public HttpHeaders getHeaders() {  // 定义一个HTTP的头信息  HttpHeaders headers = new HttpHeaders();  String auth = "mohist:650901";  // 进行加密处理  byte[] encodedAuth = Base64.*getEncoder*().encode(auth.getBytes(Charset.*forName*("US-ASCII")));  // 在进行授权的头信息内容配置的时候加密的信息一定要与“Bassic”之间有一个空格  String authHeader = "Basic " + new String(encodedAuth);  headers.set("Authorization", authHeader);  return headers;  }   @Bean  public RestTemplate getRestTemplate() {  return new RestTemplate();  } } |

1. microcloud-consumer-80，修改ConsumerDeptController类，在进行Rest访问的时候设置好这个头部的信息

|  |
| --- |
| package com.kuhnwei.microcloud.controller; import com.kuhnwei.domain.Dept; 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.RestController; import org.springframework.web.client.RestTemplate; import javax.annotation.Resource; import java.util.List;@RestController public class ConsumerDeptController {  public static final String *DEPT\_GET\_URL* = "http://dept-8001.com:8001/dept/get/";  public static final String *DEPT\_LIST\_URL* = "http://dept-8001.com:8001/dept/list";  public static final String *DEPT\_SAVE\_URL* = "http://dept-8001.com:8001/dept/save";   @Resource  private RestTemplate restTemplate;  @Resource  private HttpHeaders httpHeaders;   @RequestMapping(value = "/consumer/dept/get")  public Object getDept(long id) {  Dept dept = this.restTemplate.exchange(*DEPT\_GET\_URL* + id, HttpMethod.*GET*,  new HttpEntity<>(this.httpHeaders), Dept.class).getBody();  return dept;  }   @SuppressWarnings("unchecked")  @RequestMapping(value = "/consumer/dept/list")  public Object listDept() {  List<Dept> allDepts = this.restTemplate.exchange(*DEPT\_LIST\_URL*, HttpMethod.*GET*,  new HttpEntity<>(this.httpHeaders), List.class).getBody();  return allDepts;  }   @RequestMapping(value = "/consumer/dept/save")  public Object saveDept(Dept dept) {  Boolean flag = this.restTemplate.exchange(*DEPT\_SAVE\_URL*, HttpMethod.*POST*,  new HttpEntity<>(dept, this.httpHeaders), Boolean.class).getBody();  return flag;  } } |

## 无状态Session配置

通过之前一系列的演示可以发现整个Rest项目之中的一个问题所在，所有的Rest都是基于HTTP协议的一种应用，而在这种应用上，所有的WEB容器一般都会提供有一个Session的机制，也就是说每一个用户访问之后如果该用户一直连接，则认为该用户应该一直被服务器保存状态，但是微服务又可能同时并发访问几十万人，那么如果有的Session状态都被维护着？

1. microcloud-provider-dept-8001，现在修改Rest程序类，追加一个取得session id的方法；

|  |
| --- |
| @RequestMapping(value = "dept/sessionId") public Object id(HttpServletRequest request) {  return request.getSession().getId(); } |

随后进行提供者的Rest连接访问：<http://mohist:650901@dept-8001.com:8001/dept/sessionId>;

1. 在有一些的SpringCloud的配置之中，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状态设置，但是从整体的操作来说，session最好设置为无状态。
   1. 以下为保持Session状态（服务器内存有可能被占满）；

|  |
| --- |
| **security:  sessions:** always |

* 1. 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 |

不管你以后的项目或者支持类中是否设置无状态的问题，你最好都进行一下设置，否则你的Rest服务将受到严重的内存困扰，最严重的问题就是内存溢出。

## 定义安全配置程序类

在进行Rest服务开发的时候，为了保证安全，所有的程序里面都需要进行Spring-Security安全认证处理，可是之前所进行的认证处理都是在application.yml配置文件完成的，这样的配置明显是非常不合乎逻辑的，因为如果此时你要开发的微服务很多，并且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microloud-security的Maven模块；
2. microcloud-security，修改pom.xml配置文件；

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency> </dependencies> |

1. microcloud-security，建立一个统一的安全配置类；

|  |
| --- |
| package com.kuhnwei.microcloud.config;  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.configuration.EnableWebSecurity; import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter; import org.springframework.security.config.http.SessionCreationPolicy;  import javax.annotation.Resource; @Configuration @EnableWebSecurity public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  @Resource  public void configGlobal(AuthenticationManagerBuilder auth) throws Exception {  auth.inMemoryAuthentication().withUser("mohist").password("650901").roles("USER")  .and().withUser("admin").password("650901").roles("USER", "ADMIN");  }   @Override  protected void configure(HttpSecurity httpSecurity) throws Exception {  // 表示所有的访问都必须进行认证处理后才可以正常进行  httpSecurity.httpBasic().and().authorizeRequests().anyRequest().fullyAuthenticated();  // 所有的Rest服务一定要设置为无状态，以提升操作性能  httpSecurity.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);  } } |

1. microcloud-provider-dept-8001，修改pom.xml配置文件，引入安全配置模块；

|  |
| --- |
| <dependency>  <groupId>com.kuhnwei</groupId>  <artifactId>microcloud-security</artifactId> </dependency> |

1. microcloud-provider-dept-8001，删除掉application.yml中与安全有关的配置项（以下内容全删）；

|  |
| --- |
| **security:  sessions:** stateless  **basic:  enabled:** true *# 启动SpringSecurity的安全配置项* **user:  name:** mohist *# 认证用户名* **password:** 650901 *# 认证密码* **role:** *# 授权角色* - USER |

由于现在所写的安全处理类在程序启动类的子包之中，应该可以自动扫描到。

1. 访问地址：<http://mohist:650901@dept-8001.com:8001/dept/sessionId>;

# Eureka服务发现框架

对于服务发现框架可以简单的理解为服务的注册以及使用操作步骤，例如：zookeeper组件，这个组件里面已经明确的描述了一个服务的注册以及发现操作流程，在整个Rest架构里面，会存在有大量的微服务的信息。

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，所有的微服务在使用之中全部向Eureka之中进行注册，而后客户端直接利用Eureka进行服务信息的获得。

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出现的，也就是说你只需要编写一个程序的项目类，而后就可以启动Eureka注册服务了。

## 定义Eureka服务端

1. 为了方便进行统一的微服务的管理，建议创建一个新的项目：microcloud-eureka-7001；
2. microcloud-eureka-7001的pom.xml配置文件，追加相应的依赖支持库

|  |
| --- |
| <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.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId> </dependency> |

1. microcloud-eureka-7001现在修改application.yml配置文件，在这个配置文件里面主要进行eureka服务的定义。

|  |
| --- |
| **server:  port:** 7001 **eureka:  instance:  hostname:** eureka-7001.com *# 定义Eureka实例所在的主机名称* |

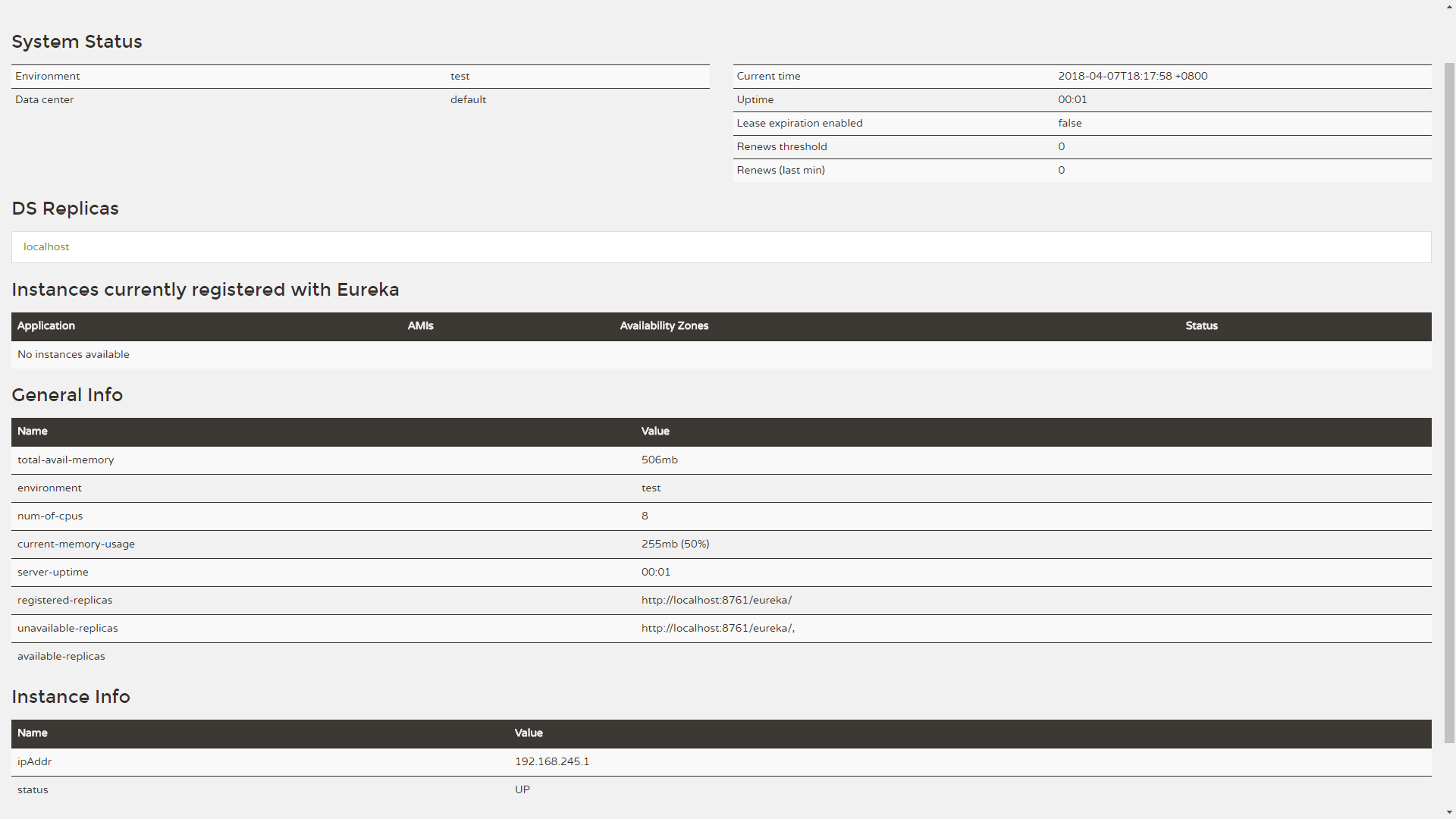
1. microcloud-eureka-7001，修改hosts配置文件，追加eureka的映射地址。

|  |
| --- |
| 127.0.0.1 eureka-7001.com |

1. microcloud-eureka-7001，修改Eureka程序启动类，追加有Eureka服务声明：

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;@SpringBootApplication @EnableEurekaServer public class Eureka\_7001\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Eureka\_7001\_StartSpringCloudApplication.class, args);  } } |

1. 运行程序后通过浏览器执行路径：<http://eureka-7001.com:7001/>



## 向Eureka中注册部门微服务

现在Eureka已经可以正常启动了，那么随后就需要在项目之中将所有的微服务信息注册到Eureka服务之中，那么这样就可以被客户端执行并且调用了。

1. microcloud-provider-dept-8001，修改pom.xml配置文件，追加有eureka的相关依赖支持包；

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

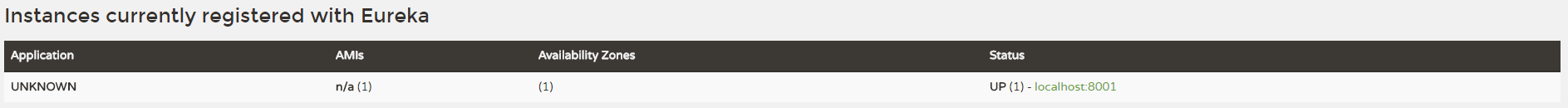
1. microcloud-provider-dept-8001，修改application.yml配置文件，在这个配置文件之中主要是定义要进行注册的Eureka服务的地址，而这个地址就是Eureka的客户端配置

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

1. microcloud-provider-dept-8001，修改项目的运行主类，在这个主类上追加有Eureka客户端的启用注解；

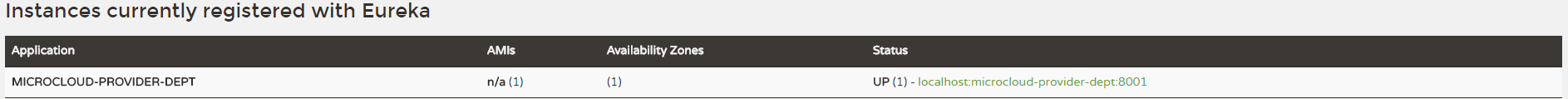
|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.EnableEurekaClient;@SpringBootApplication @EnableEurekaClient public class Dept\_8001\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Dept\_8001\_StartSpringCloudApplication.class, args);  } } |

此时由于存在有“@EnableEurekaClient”注解信息，所以当服务启动之后该服务会自动注册到Eureka服务器之中；



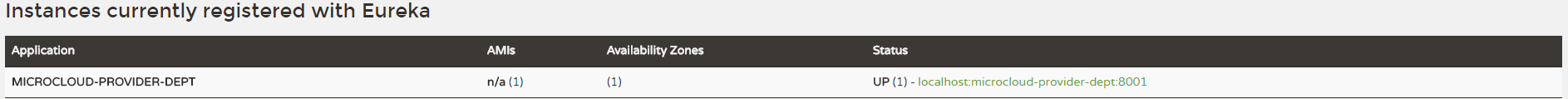
1. microcloud-provider-dept-8001，修改application.yml配置文件，为此微服务设置一个名字（这个名字将作为日后负载均衡）

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



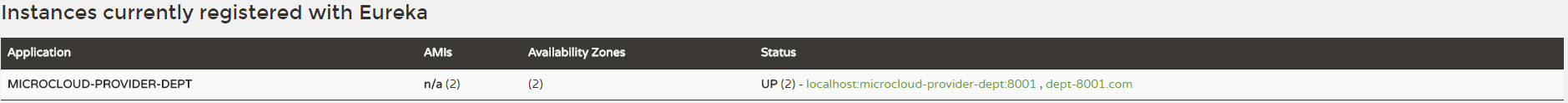
## Eureka服务信息

现在虽然成功的实现了微服务的Eureka注册，但是所表现出来的微服务的信息并不完整，因为给定地址信息是你的主机名称，而我们现在是一个自定义的路径地址。



1. microcloud-provider-dept-8001，修改application.yml配置文件，追加主机名称的显示；

|  |
| --- |
| **eureka:  client:  service-url:  defaultZone:** http://eureka-7001.com:7001/eureka  **instance:  instance-id:** dept-8001.com |



1. microcloud-provider-dept-8001，在服务信息查看的时候应该以IP地址作为连接项

|  |
| --- |
| **eureka:  client:  service-url:  defaultZone:** http://eureka-7001.com:7001/eureka  **instance:  instance-id:** dept-8001.com  **prefer-ip-address:** true *# 访问的路径变为IP地址* |

1. microcloud-provider-dept-8001，如果现在要想查看所有的微服务详细信息，则需要修改pom.xml文件，追加监控配置

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

1. microcloud，修改pom.xml文件，追加一个信息匹配的插件

|  |
| --- |
| <resources>  <resource>  <directory>src/main/resources</directory>  <filtering>true</filtering>  </resource> </resources> |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <configuration>  <delimiters>  <delimiter>$</delimiter>  </delimiters>  </configuration> </plugin> |

1. microcloud-provider-dept-8001，修改application.yml配置文件，追加info的相关信息；

|  |
| --- |
| **info:  app.name:** mohist-microcloud  **company.name:** www.kuhnwei.com  **build.artifactId:** $project.artifactiId$  **build.version:** $project.version$ |

由于项目之中微服务可能会有成百上千个，所以微服务的信息一定要认真填写，否则你的项目维护会非常痛苦。

## Eureka发现管理

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程序，但是对于监控程序那么就必然要面临以下问题：

* 新服务追加的时候应该立刻可以进行注册；
* 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配置文件

|  |
| --- |
| **server:  port:** 7001 **spring:  application:   name:** microcloud-eureka-7001 **eureka:  instance:  hostname:** eureka-7001.com  **server:  eviction-interval-timer-in-ms:** 1000  **client:  service-url:   defaultZone:** http://eureka-7001.com:7001/eureka  **register-with-eureka:** false  **fetch-registry:** false |

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，会初夏你如下错误提示：

一般情况下，该配置不建议进行修改，默认就是60秒，也就是说你的微服务如果60秒没有心跳了，那么久认为可以清理掉。

1. microcloud-eureka-7001，在Eureka里面又一个问题，这个问题就是它默认支持有保护模式的概念，所谓的保护模式指的是即便现在某一个微服务不可用了，eureka不会清零，依然会进行该微服务信息的保存。

如果现在要想去改变这种保护模式的启用，则可以修改application.yml配置文件

|  |
| --- |
| **eureka:**  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false |

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，但是很多的时候Eureka里面也会自带有清除过程。

1. microcloud-provider-dept-8001，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，也就是说你客户端可以自己来进行心跳的配置处理，修改application.yml配置文件

|  |
| --- |
| **eureka:  client:  service-url:  defaultZone:** http://eureka-7001.com:7001/eureka  **instance:  instance-id:** dept-8001.com  **prefer-ip-address:** true  **lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90 |

“**lease-renewal-interval-in-seconds**”此选项，表示距离上一次发送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，则认为该服务已经宕机了。

1. microcloud-provider-dept-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信息的获取，修改DeptRest程序类，追加一个控制调用方法

|  |
| --- |
| @Resource private DiscoveryClient discoveryClient; @RequestMapping("/dept/discover") public Object discover() {  return this.discoveryClient; } |

1. microcloud-provider-dept-8001，在主程序之中启用Eureka发现服务项；

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.client.discovery.EnableDiscoveryClient; import org.springframework.cloud.netflix.eureka.EnableEurekaClient;@SpringBootApplication @EnableEurekaClient @EnableDiscoveryClient public class Dept\_8001\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Dept\_8001\_StartSpringCloudApplication.class, args);  } } |

1. 输入访问地址：<http://dept-8001.com:8001/dept/discover>

|  |
| --- |
| {"services":["microcloud-provider-dept"],"localServiceInstance":{"host":"192.168.245.1","port":8001,"uri":"http://192.168.245.1:8001","serviceId":"microcloud-provider-dept","metadata":{},"secure":false}} |

Eureka里面就是根据这些信息进行应用列表显示的。

## Eureka安全配置

现在已经成功的实现了一个Eureka的服务启动以及微服务的注册配置操作，但是现在的程序有一个问题，你自己公司的Eureka服务应该可以注册的服务只能够是满足于认证要求的微服务，所以这样一来在之前所进行的Eureka里面配置缺少关键性的一步：安全认证，所以应该为Eureka配置安全认证处理。

1. microcloud-eureka-7001，修改pom.xml文件，引入SpringSecurity的依赖

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

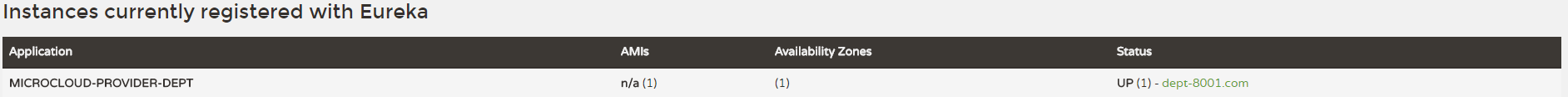
1. microcloud-eureka-7001，一旦我们的项目之中导入了security开发包，则每一次启动微服务的时候都会自动生成一个密码，而这个密码由于会改变，所以一般都不适用，所以要修改application.yml配置文件，追加密码的配置项：

|  |
| --- |
| **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 |
| **eureka:  instance:  hostname:** eureka-7001.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka  **register-with-eureka:** false  **fetch-registry:** false |

此时访问eureka的服务地址为：<http://edmin:650901@eureka-7001.com:7001/>

1. microcloud-provider-dept-8001，修改application.yml配置文件，进行授权的注册链接；

|  |
| --- |
| **eureka:  client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka  **instance:  instance-id:** dept-8001.com  **prefer-ip-address:** true  **lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90 |



## Eureka-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脚骨就将出现整体的瘫痪，就需要进行Eureka集群搭建，同时利用集群可以有效的实现HA的处理机制，如果要进行集群的搭建一定要选择两台或以上的电脑完成，而基本的流程如下。

1. 修改hosts配置文件进行多个主机名称的定义

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

1. microcloud-eureka-7001，为了方便进行Eureka操作，建议将“microcloud-eureka-7001”项目进行负载，复制为“microcloud-eureka-7002”、“microcloud-eureka-7003”。
2. microcloud-eureka-7001，修改application.yml配置文件，这个配置文件主要注意端口号以及eureka服务注册位置；

|  |
| --- |
| **server:  port:** 7001 **spring:  application:  name:** microcloud-eureka-7001 **eureka:  instance:  hostname:** eureka-7001.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **register-with-eureka:** false  **fetch-registry:** false **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 |

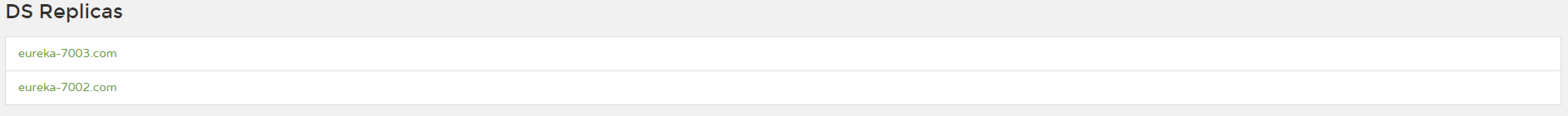
1. microcloud-eureka-7002，修改application.yml配置文件；

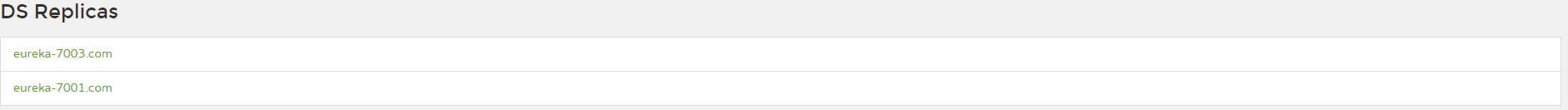
|  |
| --- |
| **server:  port:** 7002 **spring:  application:  name:** microcloud-eureka-7002 **eureka:  instance:  hostname:** eureka-7002.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **register-with-eureka:** false  **fetch-registry:** false **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 |

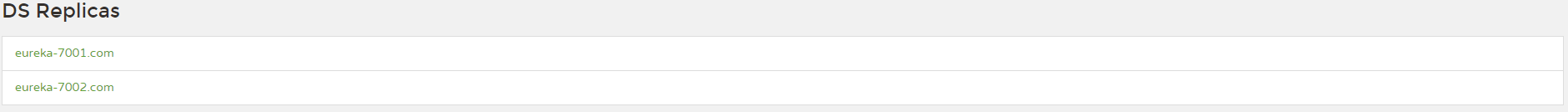
1. microcloud-eureka-7003，修改application.yml配置文件

|  |
| --- |
| **server:  port:** 7003 **spring:  application:  name:** microcloud-eureka-7003 **eureka:  instance:  hostname:** eureka-7003.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **register-with-eureka:** false  **fetch-registry:** false **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 |

1. 启动所有的eureka服务，而后进入到每一个服务的后台去观察运行的副本效果；
   1. 登录7001控制台：<http://edmin:650901@eureka-7001.com:7001/>;
   2. 登录7002控制台：http://edmin:650901@eureka-7002.com:7002/;
   3. 登录7003控制台：<http://edmin:650901@eureka-7003.com:7003/>;







1. microcloud-provider-dept-8001，修改application.yml配置文件，进行多台主机注册：

|  |
| --- |
| **eureka:  client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **instance:  instance-id:** dept-8001.com  **prefer-ip-address:** true  **lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90 |

运行之后该微服务会出现在所哟逇Eureka主机之中。这样即使某一台Eureka出现了问题，那么依然可以保证服务的可用。

## Eureka服务发布

现在已经成功的实现了Eureka编写，但是在实际的运行之中，需要将Eureka发布到具体的服务器上进行执行，而这就需要对项目进行打包处理，同样在进行打包处理的时候也必须考虑到项目的各种环境：开发（dev）、测试（beta）、生产（product），那么下面也将基于这样的方式进行eureka项目打包操作。

本次的打包处理将直接基于yml配置文件完成，对于properties配置与SpringBoot讲解微服务发布的处理过程一样。

1. microcloud-eureka-server，修改application.yml配置文件

|  |
| --- |
| **spring:  profiles:  active:** - dev-7001 --- **server:  port:** 7001 **spring:  profiles:** dev-7001  **application:  name:** microcloud-eureka-7001 **eureka:  instance:  hostname:** eureka-7001.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **register-with-eureka:** false  **fetch-registry:** false **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 --- **server:  port:** 7002 **spring:  profiles:** dev-7002  **application:  name:** microcloud-eureka-7002 **eureka:  instance:  hostname:** eureka-7002.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **register-with-eureka:** false  **fetch-registry:** false **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 --- **server:  port:** 7003 **spring:  profiles:** dev-7003  **application:  name:** microcloud-eureka-7003 **eureka:  instance:  hostname:** eureka-7003.com  **server:  eviction-interval-timer-in-ms:** 60000  **enable-self-preservation:** false  **client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **register-with-eureka:** false  **fetch-registry:** false **security:  basic:  enabled:** true  **user:  name:** edmin  **password:** 650901 |

1. microcloud-eureka-server，添加一个打包的处理插件，修改pom.xml文件；

|  |
| --- |
| <build>  <finalName>eureka-server</finalName>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <mainClass>com.kuhnwei.microcloud.EurekaServerStartSpringCloudApplication</mainClass>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins> </build> |

1. 运行maven：mvn clean install package;
2. 采用默认的方式执行eureka-server.jar，那么此时将运行在7001端口上：java –jar eureka-server.jar
3. 运行其它的两个profile配置：
   1. 运行“dev-7002”：java -jar eureka-server.jar --spring.profiles.active=dev-7002;
   2. 运行“dev-7003”：java -jar eureka-server.jar --spring.profiles.active=dev-7003。

# Ribbon负载均衡

现在所有的服务已经通过了Eureka进行了注册，那么使用Eureka注册的目的是希望所有的服务都统一归属到Eureka之中进行处理，但是现在的问题，所有的微服务汇集到了Eureka之中，而客户端的调用应该通过Eureka完成。而这种调用就可以利用Ribbon技术来实现。

Ribbon是一个服务调用的组件，并且是一个客户端实现负载均衡处理的组件。服务器端实现负载均衡可以使用Nginx、HAProxy、LVS等。

## Ribbon基本使用

1. microcloud-consumer-80，修改pom.xml配置文件，追加Ribbon相关依赖支持包；

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

1. microcloud-consumer-80，修改RestConfig配置类，在获取RestTemplate对象的时候加入Ribbon的配置。

|  |
| --- |
| package com.kuhnwei.microcloud.config; 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; import java.nio.charset.Charset; import java.util.Base64;@Configuration public class RestConfig {   // 要进行一个HTTP头信息配置  @Bean  public HttpHeaders getHeaders() {  // 定义一个HTTP的头信息  HttpHeaders headers = new HttpHeaders();  String auth = "mohist:650901";  // 进行加密处理  byte[] encodedAuth = Base64.*getEncoder*().encode(auth.getBytes(Charset.*forName*("US-ASCII")));  // 在进行授权的头信息内容配置的时候加密的信息一定要与“Bassic”之间有一个空格  String authHeader = "Basic " + new String(encodedAuth);  headers.set("Authorization", authHeader);  return headers;  }   @Bean  @LoadBalanced  public RestTemplate getRestTemplate() {  return new RestTemplate();  } } |

1. microcloud-consumer-80，修改application.yml配置文件，追加Eureka的服务注册地址配置。

|  |
| --- |
| **server:  port:** 80 **eureka:  client:  register-with-eureka:** false  **service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka |

1. microcloud-consumer-80，修改项目的启动类，追加Eureka客户端的配置

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.EnableEurekaClient;@SpringBootApplication @EnableEurekaClient public class Consumer\_80\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Consumer\_80\_StartSpringCloudApplication.class, args);  } } |

1. microcloud-consumer-80，修改控制器调用类
   1. 现在在eureka之中注册的所有服务的名称都是大写字母：**MICROCLOUD-PROVIDER-DEPT**

|  |
| --- |
| package com.kuhnwei.microcloud.controller; import com.kuhnwei.domain.Dept; 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.RestController; import org.springframework.web.client.RestTemplate; import javax.annotation.Resource; import java.util.List;@RestController public class ConsumerDeptController {  public static final String *DEPT\_GET\_URL* = "http://MICROCLOUD-PROVIDER-DEPT/dept/get/";  public static final String *DEPT\_LIST\_URL* = "http://MICROCLOUD-PROVIDER-DEPT/dept/list";  public static final String *DEPT\_SAVE\_URL* = "http://MICROCLOUD-PROVIDER-DEPT/dept/save";   @Resource  private RestTemplate restTemplate;  @Resource  private HttpHeaders httpHeaders;   @RequestMapping(value = "/consumer/dept/get")  public Object getDept(long id) {  Dept dept = this.restTemplate.exchange(*DEPT\_GET\_URL* + id, HttpMethod.*GET*,  new HttpEntity<>(this.httpHeaders), Dept.class).getBody();  return dept;  }   @SuppressWarnings("unchecked")  @RequestMapping(value = "/consumer/dept/list")  public Object listDept() {  List<Dept> allDepts = this.restTemplate.exchange(*DEPT\_LIST\_URL*, HttpMethod.*GET*,  new HttpEntity<>(this.httpHeaders), List.class).getBody();  return allDepts;  }   @RequestMapping(value = "/consumer/dept/save")  public Object saveDept(Dept dept) {  Boolean flag = this.restTemplate.exchange(*DEPT\_SAVE\_URL*, HttpMethod.*POST*,  new HttpEntity<>(dept, this.httpHeaders), Boolean.class).getBody();  return flag;  } } |

访问地址：<http://client.com/consumer/dept/list>。这个时候又Ribbon与Eureka整合之后用户不再去关注具体的Rest服务的地址与端口号了，所有的信息获取都通过Eureka完成。

## Ribbon负载均衡

通过上面的代码可以发现在Ribbon里面有一个负载均衡的注解：@LoadBalanced，那么就意味着现在就可以实现负载均衡的处理了。

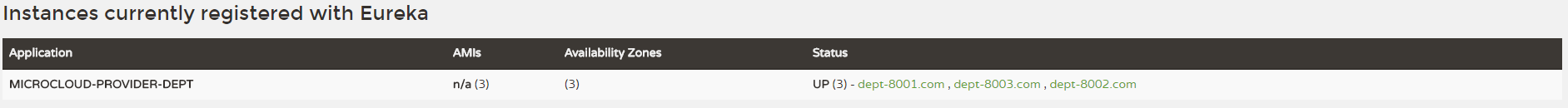
1. microcloud-provider-dept-8001，将此项目复制为两份“microcloud-provider-dept-8002”、“ microcloud-provider-dept-8003”；
2. microcloud-provider-dept-\*，执行各自的数据库脚本，随后修改各自的数据库链接配置；
3. microcloud-provider-dept-\*，修改各自服务的application.yml配置文件；

千万记住，现在所有的服务名字一定要保持一致，如果不一致则会认为是两个服务，无法进行负载均衡。

1. 修改项目中的hosts配置文件，在这个配置文件之中追加新的域；

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

1. microcloud-provider-dept-\*，在保证eureka已经正确启动之后启动所有的部门微服务信息；



|  |
| --- |
| <http://mohist:650901@dept-8001.com:8001/dept/list>  <http://mohist:650901@dept-8002.com:8002/dept/list>  <http://mohist:650901@dept-8003.com:8002/dept/list> |

1. microcloud-consumer、启动消费端，消费端在RestTemplate配置的时候使用了负载均衡的注解。
   1. 访问地址：<http://client.com/consumer/dept/list>;

现在发现每一次获取数据都是通过不同的微服务获得的，所以现在同一个消费端就可以通过Ribbon实现了负载均衡配置处理。

## 自定义Ribbon配置

在之前使用了一个“@LoadBalance”注解，该注解描述的是一个负载均衡，但是对于负载均衡也是可以由用户修改负载均衡策略的。那么如果说现在要想去修改这样的策略，也是可以的，可以使用自定义的LoadBalance的配置类。

1. microcloud-consumer-80，追加一个LoadBalance的配置类，这个类应该放在SpringCloud启动后找不到的位置上，所以应该做一个新的包；

|  |
| --- |
| package com.kuhnwei.common; import com.netflix.loadbalancer.IRule; import com.netflix.loadbalancer.RandomRule; import org.springframework.context.annotation.Beanpublic class MyLoadBalanceConfig {  @Bean  public IRule ribbonRule() { // 其中IRule就是所有规则的标准  // 随机的访问策略  return new RandomRule();  } } |

1. microcloud-consumer-80，修改程序主类，追加Ribbon的配置操作；

|  |
| --- |
| package com.kuhnwei.microcloud;  import com.kuhnwei.common.MyLoadBalanceConfig; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.EnableEurekaClient; import org.springframework.cloud.netflix.ribbon.RibbonClient; @SpringBootApplication @EnableEurekaClient @RibbonClient(name = "ribbonClient", configuration = MyLoadBalanceConfig.class) public class Consumer\_80\_StartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(Consumer\_80\_StartSpringCloudApplication.class, args);  } } |

那么此时就实现了负载均衡的配置处理操作。

1. microcloud-consumer-80，修改控制器程序类，在这个程序类之中可以通过负载均衡的客户端获取服务器的相关信息；

|  |
| --- |
| @Resource private LoadBalancerClient loadBalancerClient; @RequestMapping(value = "/consumer/dept/get") public Object getDept(long id) {  ServiceInstance serviceInstance = this.loadBalancerClient.choose("MICROCLOUD-PROVIDER-DEPT");  System.*out*.println("【\*\*\* ServiceInstance \*\*\*】"  + "host = " + serviceInstance.getHost()  + "、port = " + serviceInstance.getPort()  + "、serviceId = " + serviceInstance.getServiceId());  Dept dept = this.restTemplate.exchange(*DEPT\_GET\_URL* + id, HttpMethod.*GET*,  new HttpEntity<>(this.httpHeaders), Dept.class).getBody();  return dept; } |

那么此时后台通过指定的负载均衡的配置程序，就可以取得即将操作的主机的信息。

## 禁用Eureka实现Ribbon调用

在之前使用的技术都是通过Eureka访问了Eureka之中的指定名称的服务而获得了所需要的数据，但是在Ribbon设计的时候考虑到了一个脱离Eureka使用的环境。如果要真进行了Eureka的脱离，代码将变为如下的形式。

1. microcloud-consumer-ribbon，修改application.yml配置文件

|  |
| --- |
| **ribbon:  eureka:  enabled:** false **microcloud-provider-dept:  ribbon:  listOfServers:** http://dept-8001.com:8001, http://dept-8002.com:8002, http://dept-8003.com:8003 |

1. microcloud-consumer-ribbon，修改RestConfig配置程序类，不再需要使用“@LoadBalanced”注解；

|  |
| --- |
|  |

这种模式并不标准，只能够说是Ribbon自己所具备的一项功能而已，实际之中如果不是非常必要，不建议使用。

# Feign接口服务

现在为止所进行的所有的Rest服务调用实际上都会出现一个非常尴尬的局面，例如：以如下代码为例：

|  |
| --- |
| Dept dept = this.restTemplate.exchange(*DEPT\_GET\_URL* + id, HttpMethod.*GET*,  new HttpEntity<>(this.httpHeaders), Dept.class).getBody(); |

所有的数据的调用和转换都必须由用户自己来完成，而我们本身不擅长这些，我们习惯的编程模式是：通过接口来实现业务的操作，而不是通过具体的Rest数据。

## Feign基本使用

为了方便起见现在将“microcloud-consumer-80”模块复制为“microcloud-consumer-feign”模块。

1. microcloud-consumer-feign，为了可以使用到feign支持，需要修改pom.xml配置文件，引入相关依赖包；

|  |
| --- |
| <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相关支持库。

1. microcloud-consumer-feign，此时如果要通过Feign进行远程Rest调用，那么必须要考虑服务的认证问题。
   1. 此时可以剔除原始的RestConfig进行的配置处理；

|  |
| --- |
| package com.kuhnwei.common; import feign.auth.BasicAuthRequestInterceptor; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;@Configuration public class FeignClientConfig {  @Bean  public BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  return new BasicAuthRequestInterceptor("mohist", "650901");  } } |

1. microcloud-consumer-feign，建立一个DeptClientService接口

|  |
| --- |
| package com.kuhnwei.service; import com.kuhnwei.common.FeignClientConfig; import com.kuhnwei.domain.Dept; 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 java.util.List;@FeignClient(value = "MICROCLOUD-PROVIDER-DEPT", configuration = FeignClientConfig.class) public interface DeptClientService {  @RequestMapping(method = RequestMethod.*GET*, value = "/dept/get/{id}")  public Dept get(@PathVariable("id") long id);  @RequestMapping(method = RequestMethod.*GET*, value = "/dept/list")  public List<Dept> list();  @RequestMapping(method = RequestMethod.*GET*, value = "/dept/save")  public boolean save(Dept dept); } |

1. microcloud-consumer-feign，修改ConsumerFeignStartSpringCloudApplication，追加service包的扫描处理。

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.EnableEurekaClient; import org.springframework.cloud.netflix.feign.EnableFeignClients;@SpringBootApplication @EnableEurekaClient @EnableFeignClients(basePackages = {"com.kuhnwei.service"}) public class ConsumerFeignStartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(ConsumerFeignStartSpringCloudApplication.class, args);  } } |

1. microcloud-consumer-feign，修改ConsumerDeptController控制器类

|  |
| --- |
| package com.kuhnwei.microcloud.controller; import com.kuhnwei.domain.Dept; import com.kuhnwei.service.DeptClientService; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController; import javax.annotation.Resource;@RestController public class ConsumerDeptController {  @Resource  private DeptClientService deptService;  @RequestMapping(value = "/consumer/dept/get")  public Object getDept(long id) {  return this.deptService.get(id);  }  @SuppressWarnings("unchecked")  @RequestMapping(value = "/consumer/dept/list")  public Object listDept() {  return this.deptService.list();  }  @RequestMapping(value = "/consumer/dept/save")  public Object saveDept(Dept dept) {  return this.deptService.save(dept);  } } |

这次使用的接口是通过Rest服务的定义内容自动帮助用户生成的。

1. 启动测试：<http://client.com/consumer/dept/list>
   1. 可以发现Feign在处理的时候自带有负载均衡的配置项。

## Feign相关配置

1. microcloud-consumer-feign，Feign之中最为核心的作用就是将Rest服务的信息转换为接口，但是在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，Rest的核心本质在于：JSON数据传输，XML文本，于是就必须思考一种情况，某用户发送的数据很大呢？所以这个时候可以考虑修改application.yml配置文件传输数据进行压缩；

|  |
| --- |
| **feign:  compression:  request:  mime-types:** *# 可以被压缩的类型* - text/xml  - application/xml  - application/json  **min-request-size:** 2048 *# 超过2048个字节的数据进行压缩* |

1. 如果有需要则可以在项目之中开启feign的相关日志信息（默认不开启）
   1. microcloud-consumer-feign，修改application.yml配置文件，追加日志追踪；

|  |
| --- |
| **logging:  level:   com.kuhnwei.service:** debug |

* 1. microcloud-consumer-feign，修改FeignClientConfig，开启日志的输出；

|  |
| --- |
| package com.kuhnwei.common; import feign.Logger; import feign.auth.BasicAuthRequestInterceptor; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;@Configuration public class FeignClientConfig {  @Bean  public Logger.Level getFeignLoggerLevel() {  return Logger.Level.*FULL*;  }  @Bean  public BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  return new BasicAuthRequestInterceptor("mohist", "650901");  } } |

当再次运行的时候可以发现有如下的日志信息输出：

|  |
| --- |
| <http://client.com/consumer/dept/list> |
| 2018-04-09 22:59:30.161 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] ---> GET http://MICROCLOUD-PROVIDER-DEPT/dept/list HTTP/1.1  2018-04-09 22:59:30.161 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] Authorization: Basic bW9oaXN0OjY1MDkwMQ==  2018-04-09 22:59:30.161 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] ---> END HTTP (0-byte body)  2018-04-09 22:59:30.165 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] <--- HTTP/1.1 200 OK (4ms)  2018-04-09 22:59:30.165 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] cache-control: no-cache, no-store, max-age=0, must-revalidate  2018-04-09 22:59:30.165 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] content-type: application/json;charset=utf-8  2018-04-09 22:59:30.165 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] date: Mon, 09 Apr 2018 14:59:30 GMT  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] expires: 0  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] pragma: no-cache  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] transfer-encoding: chunked  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] x-application-context: microcloud-provider-dept:8003  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] x-content-type-options: nosniff  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] x-frame-options: DENY  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] x-xss-protection: 1; mode=block  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list]  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] [{"deptno":1,"dname":"开发部","loc":"mohist8003"},{"deptno":2,"dname":"财务部","loc":"mohist8003"},{"deptno":3,"dname":"市场部","loc":"mohist8003"},{"deptno":4,"dname":"后勤部","loc":"mohist8003"},{"deptno":5,"dname":"公关部","loc":"mohist8003"},{"deptno":7,"dname":"测试部-1523088284655","loc":"mohist8003"},{"deptno":8,"dname":"测试部-1523088311502","loc":"mohist8003"},{"deptno":9,"dname":"WEB测试","loc":"mohist8003"},{"deptno":10,"dname":"WEB测试2","loc":"mohist8003"}]  2018-04-09 22:59:30.166 DEBUG 2384 --- [tp1551070117-26] com.kuhnwei.service.DeptClientService : [DeptClientService#list] <--- END HTTP (499-byte body) |

现在可以观察到如下的流程：

1. 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，这个请求是发送给Eureka（地址：“http://MICROCLOUD-PROVIDER-DEPT/dept/list”）；
2. 随后由于配置了授权处理，所以继续发送授权信息（“Authorizaiton”）；
3. 在进行服务调用的时候Feign融合了Ribbon技术，所以也支持有负载均衡的处理。

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

# Hystrix熔断机制

所谓的熔断机制和日常生活中见到电路保险丝是非常相似的，当出现了问题之后，保险丝会自动烧断以保护我们的电器，那么如果换到了程序之中呢？

当现在服务的提供方出现了问题之后整个的程序将出现错误的信息显示，而这个时候如果不想出现这样的错误信息，而希望替换为一个错误时的内容。

为了保证不先出这样的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以下几种情况：

* 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（客户端）进行熔断的配置。

## Hystrix基本配置

在SpringCloud给出的官方文档之中对于Hystrix信息的描述还是非常到位的。

1. microcloud-provider-dept-hystrix-8001，修改pom.xml配置文件，追加Hystrix配置类；

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

1. microcloud-provider-dept-hystrix-8001，修改DeptRest程序

|  |
| --- |
| package com.kuhnwei.microcloud.rest; import com.kuhnwei.domain.Dept; import com.kuhnwei.microcloud.service.DeptService; import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand; import org.springframework.web.bind.annotation.\*; import javax.annotation.Resource; import javax.servlet.http.HttpServletRequest;@RestController public class DeptRest {  @Resource  private DeptService deptService;  // 如果当前调用的get()方法出现了错误，则执行fallback  @HystrixCommand(fallbackMethod = "getFallback")  @RequestMapping(value = "/dept/get/{id}", method = RequestMethod.*GET*)  public Object get(@PathVariable("id") long id) {  return this.deptService.get(id);  }   // 此时方法的参数与get()一致  public Object getFallback(@PathVariable("id") long id) {  Dept dept = new Dept();  dept.setDeptno(999999L);  // 错误的提示  dept.setDname("[ERROR] Microcloud-Dept-Hystrix");  dept.setLoc("DEPT-PROVIDER");  return dept;  } } |

一旦get()方法上抛出了错误的信息，那么就认为该服务有问题，会默认使用“@HystrixCommand”注解之中配置好的fallbackMethod调用类中的指定方法，返回响应的数据。

1. microcloud-provider-dept-hystrix-8001，在主类之中启动熔断处理；

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker; import org.springframework.cloud.client.discovery.EnableDiscoveryClient; import org.springframework.cloud.netflix.eureka.EnableEurekaClient;@SpringBootApplication @EnableEurekaClient @EnableDiscoveryClient @EnableCircuitBreaker public class DeptHystrixStartSpringCloudApplication\_8001 {  public static void main(String[] args) {  SpringApplication.*run*(DeptHystrixStartSpringCloudApplication\_8001.class, args);  } } |

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

## 服务降级（服务回退）

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，二回出现本地的操作调用。

例如：在每年春运期间12306都是最繁忙的时候，那么在这个情况会发现有一些神奇的情况：当到了指定的时间大家开始抢票的时候，如果你不抢，而后查询一些冷门的车次，票有可能查询不出来。因为这个时候会将所有的系统资源给抢票调度了，而其它的服务由于其暂时不受到过多的关注，这个时候可以考虑将服务降级（服务暂停）。

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. microcloud-service，扩充一个DeptService的失败调用（服务降级）处理；

|  |
| --- |
| package com.kuhnwei.service.fallback; import com.kuhnwei.domain.Dept; import com.kuhnwei.service.DeptClientService; import feign.hystrix.FallbackFactory; import org.springframework.stereotype.Component; import java.util.List;@Component public class DeptClientServiceFallbackFactory implements FallbackFactory<DeptClientService> {  @Override  public DeptClientService create(Throwable throwable) {  return new DeptClientService() {  @Override  public Dept get(long id) {  Dept dept = new Dept();  dept.setDeptno(88888888L);  dept.setDname("[ERROR]Feign-Hystrix");  dept.setLoc("Consumer客户端提供");  return dept;  }  @Override  public List<Dept> list() {  return null;  }  @Override  public boolean save(Dept dept) {  return false;  }  };  } } |

1. microcloud-service，修改DeptClientService接口，追加本地的Fallback配置。

|  |
| --- |
| package com.kuhnwei.service; import com.kuhnwei.common.config.FeignClientConfig; import com.kuhnwei.domain.Dept; import com.kuhnwei.service.fallback.DeptClientServiceFallbackFactory; 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 java.util.List;@FeignClient(value = "MICROCLOUD-PROVIDER-DEPT",   configuration = FeignClientConfig.class,   fallbackFactory = DeptClientServiceFallbackFactory.class) public interface DeptClientService {  @RequestMapping(method = RequestMethod.*GET*, value = "/dept/get/{id}")  public Dept get(@PathVariable("id") long id);  @RequestMapping(method = RequestMethod.*GET*, value = "/dept/list")  public List<Dept> list();  @RequestMapping(method = RequestMethod.*GET*, value = "/dept/save")  public boolean save(Dept dept); } |

此时当服务不可用的时候就会执行“DeptClientServiceFallbackFactory”类中返回的DeptClientService接口的匿名对象信息。

1. microcloud-consumer-hystrix，修改application.yml配置文件，追加feign配置启用。

|  |
| --- |
| **feign:****hystrix:  enabled:** true |

1. microcloud-consumer-hystrix，修改程序启动主类

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.EnableEurekaClient; import org.springframework.cloud.netflix.feign.EnableFeignClients; import org.springframework.context.annotation.ComponentScan;@SpringBootApplication @EnableEurekaClient @ComponentScan({"com.kuhnwei.service", "com.kuhnwei.microcloud"}) @EnableFeignClients(basePackages = {"com.kuhnwei.service"}) public class ConsumerHystrixStartSpringCloudApplication {  public static void main(String[] args) {  SpringApplication.*run*(ConsumerHystrixStartSpringCloudApplication.class, args);  } } |

当追加了“@ComponentScan”注解之后才可以进行包的扫描配置。

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可以得到一些固定的提示信息。

1. microcloud-consumer-hystrix，启动程序
   1. 访问<http://client.com/consumer/dept/get?id=1>

当不启动任何服务的时候，客户端访问不到服务时，执行了“服务降级”机制的处理。

|  |
| --- |
| {"deptno":88888888,"dname":"[ERROR]Feign-Hystrix","loc":"Consumer客户端提供"} |

## HystrixDashboard

在Hystrix里面提供有一种监控的功能，那么这个功能就是“Hystrix Dashboard”，可以利用它来进行整体微服务的监控操作。

1. 首先为了方便监控，将建立一个新的监控项目：microcloud-consumer-hystrix-dashboard；
2. microcloud-consumer-hystrix-dashboard，修改项目中的pom.xml配置文件

|  |
| --- |
| <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> |

1. microcloud-provider-\*，所有的服务提供者之中都一定要提供有监控服务依赖库；

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

1. microcloud-consumer-hystrix-dashboard，修改application.yml配置文件，主要进行端口的配置；

|  |
| --- |
| **server:  port:** 9001 |

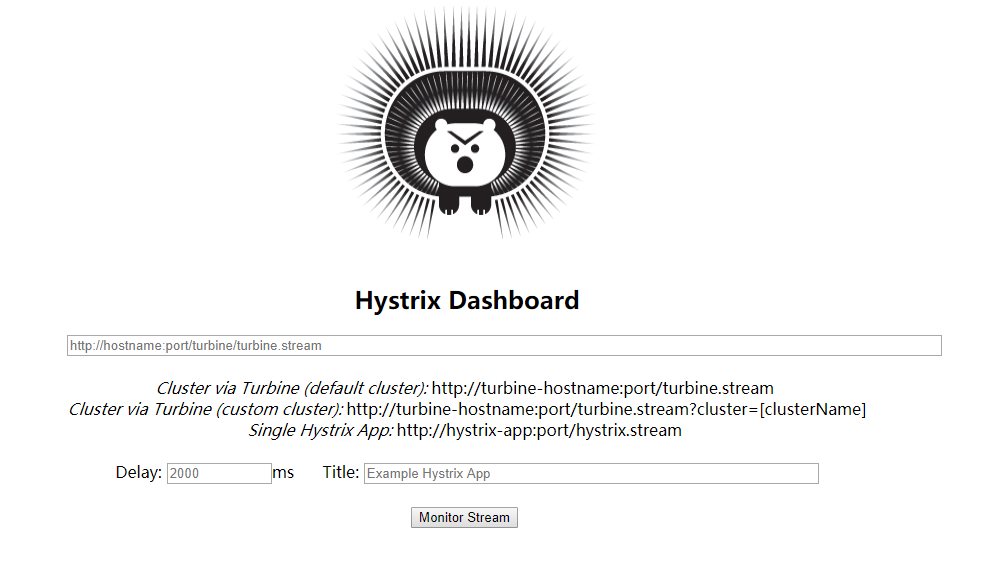
1. microcloud-consumer-hystrix-dashboard，创建一个监控的主体类；

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

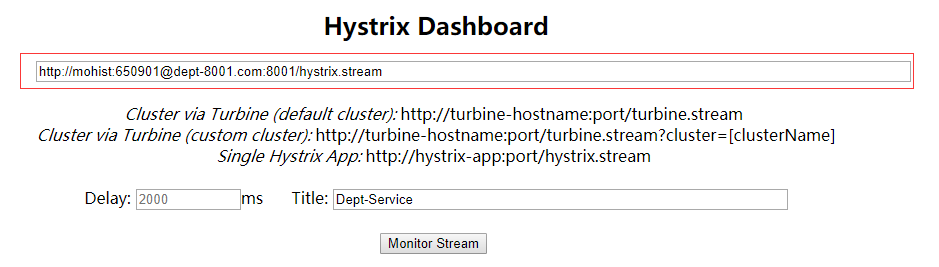
1. 修改hosts主机文件，增加主机列表；

|  |
| --- |
| 127.0.0.1 dashboard.com |

服务运行地址：<http://dashboard.com:9001/hystrix>;



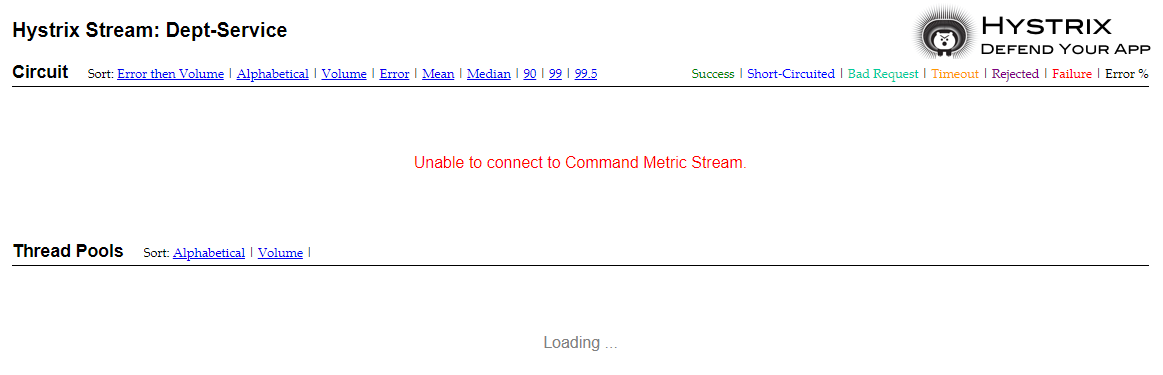
1. 的到microcloud-provider-dept的监控信息：<http://mohist:650901@dept-8001.com:8001/hystrix.stream>;
   1. 如果此时要想获取监控信息必须去运行微服务；
2. 将之前的监控的路径填写到之前启动好的dashboard程序页面之中；





1. 如果微服务的方法都没有配置@HystrixCommand，则使用监控时会出现如下错误情况；

|  |
| --- |
| 2018-04-11 20:43:42.665 INFO 12048 --- [tp1933250332-63] ashboardConfiguration$ProxyStreamServlet :  Proxy opening connection to: <http://mohist:650901@dept-8001.com:8001/hystrix.stream>  2018-04-11 20:43:42.665 INFO 12048 --- [tp1933250332-51] ashboardConfiguration$ProxyStreamServlet :  Proxy opening connection to: <http://mohist:650901@dept-8001.com:8001/hystrix.stream>  2018-04-11 20:43:42.672 WARN 12048 --- [tp1933250332-63] ashboardConfiguration$ProxyStreamServlet : Failed opening connection to <http://mohist:650901@dept-8001.com:8001/hystrix.stream> : 404 : HTTP/1.1 404 Not Found  2018-04-11 20:43:42.674 WARN 12048 --- [tp1933250332-51] ashboardConfiguration$ProxyStreamServlet : Failed opening connection to <http://mohist:650901@dept-8001.com:8001/hystrix.stream> : 404 : HTTP/1.1 404 Not Found |



所以应该对微服务项目做如下配置：

|  |
| --- |
| @HystrixCommand(fallbackMethod = "getFallback") @RequestMapping(value = "/dept/get/{id}", method = RequestMethod.*GET*) public Object get(@PathVariable("id") long id) {  return this.deptService.get(id); } // 此时方法的参数与get()一致 public Object getFallback(@PathVariable("id") long id) {  Dept dept = new Dept();  dept.setDeptno(999999L);  // 错误的提示  dept.setDname("[ERROR] Microcloud-Dept-Hystrix");  dept.setLoc("DEPT-PROVIDER");  return dept; } |

以及在application.yml做如下配置：

|  |
| --- |
| **hystrix:  config:  stream:  maxConcurrentConnections:** 50 |

而且只有配置了@HystrixCommand的方法，才能被监控监听。

## Turbine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的微服务需要进行整体的监控，那么这种情况下就可以利用turbine技术来实现。

1. 下面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MyBatis数据库链接，只是做一个简单的数据信息。通过一个已有的microcloud-provider-hystrix-8001复制一个新的项目：microcloud-provider-company-8101；
2. microcloud-provider-company-8101，修改项目中的pom.xml配置文件，将与安全有关的依赖包删除掉；

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</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-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>com.kuhnwei</groupId>  <artifactId>microcloud-api</artifactId>  </dependency>  <dependency>  <groupId>com.kuhnwei</groupId>  <artifactId>microcloud-security</artifactId>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jetty</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency> </dependencies> |

随后将与数据库连接池、mybatis的相关的程序类或接口全部删除掉；

1. microcloud-api，追加一个新的Java类：Company。

|  |
| --- |
| package com.kuhnwei.domain; import java.io.Serializable;@SuppressWarnings("serial") public class Company implements Serializable {  private String title;  private String note; } |

1. microcloud-provider-company-8101，建立一个新的微服务的程序类：CompanyRest

|  |
| --- |
| package com.kuhnwei.microcloud.rest; import com.kuhnwei.domain.Company; import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand; 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;@RestController public class CompanyRest {    // 如果需要进行性能监控，必须要有此注解  @HystrixCommand  @RequestMapping(value = "/company/get/{title}", method = RequestMethod.*GET*)  public Object get(@PathVariable("title") String title) {  Company company = new Company();  company.setTitle(title);  company.setNote("www.kuhnwei.com");  return company;  } } |

1. microcloud-provider-company-8101，修改程序的启动主类；

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker; import org.springframework.cloud.client.discovery.EnableDiscoveryClient; import org.springframework.cloud.netflix.eureka.EnableEurekaClient;@SpringBootApplication @EnableEurekaClient @EnableCircuitBreaker @EnableDiscoveryClient public class CompanyStartSpringCloudApplication\_8101 {  public static void main(String[] args) {  SpringApplication.*run*(CompanyStartSpringCloudApplication\_8101.class, args);  } } |

1. microcloud-provider-company-8101，修改application.yml配置文件

|  |
| --- |
| **info:  app.name:** mohist-microcloud  **company.name:** www.kuhnwei.com  **build.artifactId:** $project.artifactiId$  **build.version:** $project.version$ **server:  port:** 8101 **spring:  application:  name:** microcloud-provider-company **eureka:  client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **instance:  instance-id:** company-8101.com  **prefer-ip-address:** true  **lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90 **hystrix:  config:  stream:  maxConcurrentConnections:** 50 |

1. microcloud-provider-company-8101，追加微服务，税后取得监控信息；
   1. 在host配置文件之中追加有一个映射路径；

|  |
| --- |
| 127.0.0.1 company-8101.com |

* 1. 访问地址：<http://company-8101.com:8101/company/get/hello>;
  2. hystrix监控地址：<http://company-8101.com:8101/hystrix.stream>;

1. 如果要想实现trubine的配置，则需要建立一个turbine项目模块，这个项目可以直接通过之前microcloud-comsumer-hystrix-dashboard模块进行复制为“microcloud-consumer-turbine”模块；
2. microcloud-consumer-turbine，修改pom.xml配置文件，追加turbine的依赖程序包；

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

1. microcloud-consumer-turbine，修改application.yml配置文件；

|  |
| --- |
| **server:  port:** 9101 **eureka:  client:  service-url:   defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **instance:** *# 设置心跳的时间间隔（默认是30秒）* **lease-renewal-interval-in-seconds:** 2   *#如果现在超过了5秒间隔（默认90秒）* **lease-expiration-duration-in-seconds:** 5   *# 在信息列表时显示主机名称* **instance-id:** company-8101.com   *# 访问的路径变为IP地址* **prefer-ip-address:** true  **turbine:** *# 定义所有要监控的微服务信息* **app-config:** MICROCLOUD-PROVIDER-COMPANY, MICROCLOUD-PROVIDER-DEPT   *# 设置监控的表达式，通过此表达式表示要获取监控信息名称* **cluster-name-expression:** new String("default") |

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）;

1. microcloud-consumer-turbine，建立一个turbine的使用主类信息

|  |
| --- |
| package com.kuhnwei.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\_9101 {  public static void main(String[] args) {  SpringApplication.*run*(TurbineApplication\_9101.class, args);  } } |

1. microcloud-consumer-hystrix-dashboard，运行hystrix-dashboard监控程序；
2. microcloud-consumer-turbine，运行trubine聚合监控程序；
   1. 但是在正常启动trubine的时候出现了一下的错误提示信息；

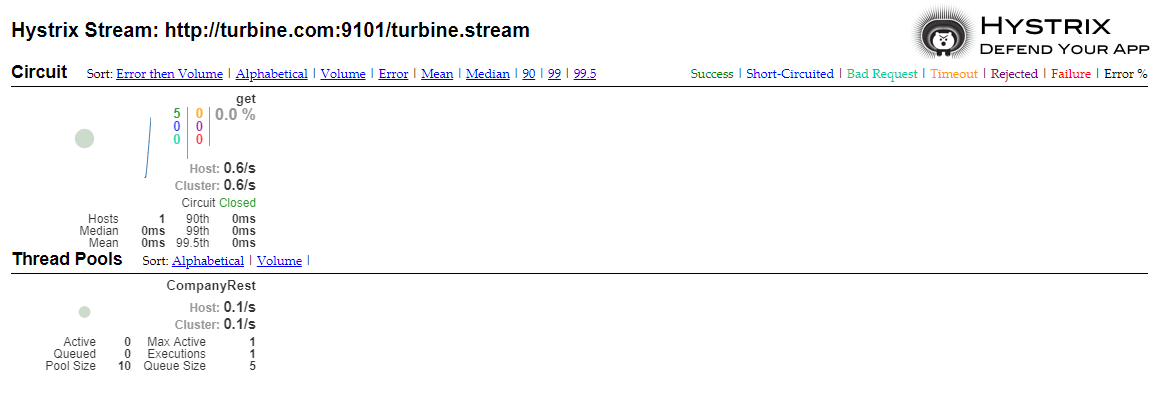
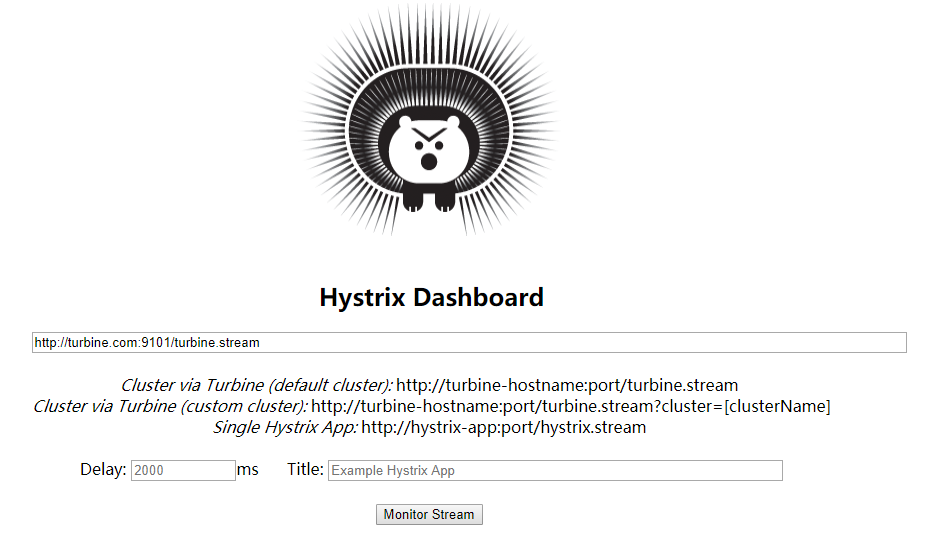
|  |
| --- |
| com.netflix.turbine.monitor.instance.InstanceMonitor$MisconfiguredHostException: [{"timestamp":1523454841013,"status":401,"error":"Unauthorized","message":"Full authentication is required to access this resource","path":"/hystrix.stream"}] |

* 1. 修改hosts配置文件，追加一个新的映射路径；

|  |
| --- |
| 127.0.0.1 turbine.com |

* 1. trubine访问路径：<http://turbine.com:9101/turbine.stream>

1. 运行HystrixDashboard监控程序：<http://dashboard.com:9001/hystrix.stream>;
   1. 在监控的位置上填写之前设置好的turbine的访问地址：



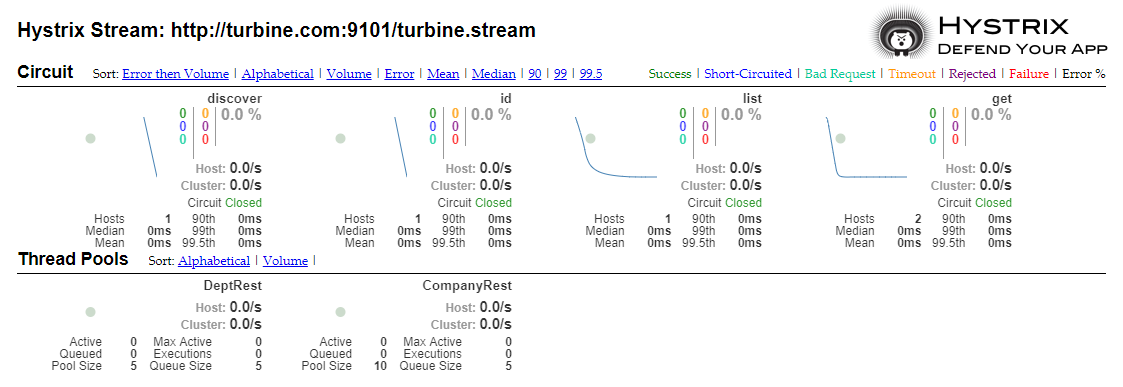
1. microcloud-security，如果现在需要turbine进行加密的微服务的方位操作，只能够采取一种折中的方案，就是要去修改整个项目中的安全策略，追加WEB安全策略配置；

|  |
| --- |
| @Override public void configure(WebSecurity web) throws Exception {  web.ignoring().antMatchers("/hystrix.stream", "/turbine.stream"); } |

现在所有的安全策略会自动抛开以上的两个访问路径，折中是基于Bean配置，如果要是你现在基于的是application.yml文件的配置，则就必须修改application.yml配置文件，追加如下内容：

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

这个时候如果启动那个之后没有出现任何的错误提示，那么就表现在已经可以绕过了Security的配置而直接进行服务的访问了。



# Zuul路由访问

在现在为止所有的微服务都是通过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控制组件：Zuul，也就是说Zuul就作为中间的一个代理层出现。

## Zuul的基本使用

本次使用Zuul将访问无安全认证的微服务信息，例如：公司信息就属于无安全认证的微服务；

1. 为了突出Zuul的功能，建立一个新的主机映射；

|  |
| --- |
| 127.0.0.1 gateway-9501.com |

以后所有的微服务的访问不再直接进行处理了，而是通过Zuul进行跳转后获得。

1. 建立一个新的模块：microcloud-zuul-gateway；
2. microcloud-zuul-gateway-9501，修改pom.xml文件，追加zuul相关依赖包；
   1. 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；

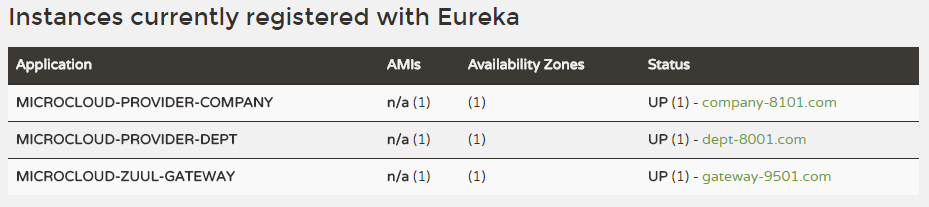
|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId> </dependency> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId> </dependency> |

1. microcloud-zuul-gateway-9501，修改application.yml配置文件；

|  |
| --- |
| **server:  port:** 9501 **eureka:  client:  service-url:   defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **instance:  lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90  **instance-id:** gateway-9501.com  **prefer-ip-address:** true **info:  app.name:** mohist-microcloud  **company.name:** www.kuhnwei.com  **build.artifactId:** $project.artifactId$  **build.version:** $project.version$ **spring:  application:  name:** microcloud-zuul-gateway |

1. microcloud-zuul-gateway-9501，创建程序启动的主类；

|  |
| --- |
| package com.kuhnwei.microcloud; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.zuul.EnableZuulProxy;@SpringBootApplication @EnableZuulProxy public class ZuulStartSpringCloudApplication\_9501 {  public static void main(String[] args) {  SpringApplication.*run*(ZuulStartSpringCloudApplication\_9501.class, args);  } } |



1. 访问公司微服务信息：
   1. 原始访问路径：<http://company-8101.com:8101/company/get/hello>;
   2. Zuul代理访问：<http://gateway-9501.com:9501/microcloud-provider-company/company/get/hello>;

同时现在可以观察到Zuul后面的日志输出信息：

|  |
| --- |
|  |

现在使用Zuul代理的只有company，因为dept上有安全认证。

## Zuul路由功能

整体的Zuul运行之后你会发现，zuul所实现就是一个代理功能，那么现在就会出现一个问题，例如：以之前访问的路径为路径：

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

此时必须要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这个名称，那么使用zuul就没有任何的实际意义，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. microcloud-zuul-gateway-9501，为指定的应用设置路径，修改application.yml配置文件；

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

那么此时就可以通过“/company-proxy”来访问“microcloud-provider-company”名称。

|  |
| --- |
| <http://gateway-9501.com:9501/company-proxy/company/get/hello> |

但是现在还会存在有一个实际问题，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：

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

1. microcloud-zuul-gateway-9501，修改application.yml配置文件忽略掉应用名称访问：
   1. 忽略掉“microcloud-provider-company”应用名称；

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

这个时候就可以进行代理的安全使用，但是如果你一个系统之中存在有几百个微服务，如果按照如上的方式进行佩恩在就会非常的麻烦，所以最简单的做法是可以采用一个通配符”\*”的模式来完成：

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

现在表示所有的Eureka中的服务名称信息访问都要忽略掉，所有的访问都需要配置一个映射路径模式来完成。

1. microcloud-zuul-gateway-9501，除了以上的模式进行服务定义之外，在zuul之中也可以采用如下的方式进行处理：

|  |
| --- |
| **zuul:  ignored-services:** "\*"  **routes:  mycompany.path:** /company-proxy/\*\*  **mycompany.serviceId:** microcloud-provider-company |

1. 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微服务的地址；

|  |
| --- |
| **zuul:  ignored-services:** "\*"  **routes:  mycompany.path:** /company-proxy/\*\*  **mycompany.serviceId:** http://company-8101.com:8101/company |

此时的地址上由于已经存在有了“company”前缀，所以访问地址为：

|  |
| --- |
| <http://gateway-9501.com:9501/company-proxy/get/hello> |

但是从实际的开发来讲不建议采用此类模式处理，因为所有的服务如果直接绑定了指定的服务提供者地址，那么将不方便进行负载均衡的配置处理，而且没有Eureka所有微服务的管理也非常不方便。

1. microcloud-zuul-gateway-9501，设置公共前缀：

|  |
| --- |
| **zuul:  prefix:** /mohist-proxy   **ignored-services:** "\*"  **routes:  microcloud-provider-company:** /company-proxy/\*\* |

一旦存在有前缀定义之后所有微服务的访问上就必须追加有前缀名称：

|  |
| --- |
| <http://gateway-9501.com:9501/mohist-proxy/company-proxy/company/get/hello> |

以前的地址：

* “/mohist-proxy”：整个zuul的前缀；
* “/company-proxy”：是在zuul中定义的映射路径；
* “/company/get/hello”：是微服务提供者提供的操作路径。

## zuul过滤访问

对于zuul的功能本质就属于一个代理操作，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证信息，那么在这样的状态下，如果你当前所代理的微服务具有认证信息，那么就必须在其访问前追加认证的头部操作，这样的功能就需要通过zuul的过滤操作完成。

1. microcloud-zuul-gateway，修改application.yml配置，这个配置之中追加dept微服务的代理；

|  |
| --- |
| **zuul:  prefix:** /mohist-proxy  **ignored-services:** "\*"  **routes:  microcloud-provider-company:** /company-proxy/\*\*  **microcloud-provider-dept:** /dept-proxy/\*\* |

此时的访问路径：<http://mohist:650901@gateway-9501.com:9501/mohist-proxy/dept-proxy/dept/get/1>;现在的密码只是设置给了zuul，而zuul并不能够将认证的信息传递到部门微服务之中。

1. microcloud-zuul-gateway，追加过滤处理操作；

|  |
| --- |
| package com.kuhnwei.microcloud.filter; import com.netflix.zuul.ZuulFilter; import com.netflix.zuul.context.RequestContext; import java.nio.charset.Charset; import java.util.Base64; */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/14 12:45  \*/* public class AuthorizedRequestFilter extends ZuulFilter {  // 表示具体的过滤执行操作  @Override  public Object run() {  // 获取请求的上下文  RequestContext currentContext = RequestContext.*getCurrentContext*();  // 认证的原始信息  String auth = "mohist:650901";  // 进行加密处理  byte[] encodedAuth = Base64.*getEncoder*().encode(auth.getBytes(Charset.*forName*("US-ASCII")));  // 在进行授权的头信息内容配置的时候加密的信息一定要与“Basic”之间有一个空格  String authHeader = "Basic " + new String(encodedAuth);  currentContext.addZuulRequestHeader("Authorization", authHeader);  return null;  }  @Override  public String filterType() {  /\*  \* 在进行Zuul过滤的时候可以设置其过滤执行的位置，那么此时有如下几种类型  \* 1、pre：在请求发出之前执行过滤，如果要进行访问，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信息的时候被调用  \* 4、error：出现错误之后进行调用  \* \*/  return "pre";  }  // 设置优先级，数字越大优先级越低  @Override  public int filterOrder() {  return 0;  }  // 该Filter是否要执行  @Override  public boolean shouldFilter() {  return true;  } } |

1. microcloud-zuul-gateway，建立一个配置程序类作为认证请求的配置Bean。

|  |
| --- |
| package com.kuhnwei.microcloud.config;  import com.kuhnwei.microcloud.filter.AuthorizedRequestFilter; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;  */\*\*  \** ***@author*** *Kuhn Wei, email@kuhnwei.com  \** ***@version*** *2018/4/14 13:04  \*/* @Configuration public class ZuulConfig {  @Bean  public AuthorizedRequestFilter getAuthorizedRequestFilter() {  return new AuthorizedRequestFilter();  } } |

那么这个时候就意味着你现在的程序可以直接利用zuul的代理访问所有加密的微服务。

1. microcloud-zuul-gateway，考虑到zuul也需要进行安全访问，所以应该修改项目中的pom.xml配置文件，追加Spring安全访问配置处理操作的依赖；

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

1. microcloud-zuul-gateway，建议application.yml配置文件，最佳有用户信息配置；

|  |
| --- |
| **security:  basic:  enabled:** true  **user:  name:** zdmin  **password:** 650901 |

那么此时表示zuul的代理上有了认证信息，则访问的地址上必须加上zuul的认证操作；

|  |
| --- |
| <http://zdmin:650901@gateway-9501.com:9501/mohist-proxy/dept-proxy/dept/get/1> |

1. 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.kuhnwei.service; import com.kuhnwei.common.config.FeignClientConfig; import com.kuhnwei.domain.Dept; import com.kuhnwei.service.fallback.DeptClientServiceFallbackFactory; 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 java.util.List;@FeignClient(value = "MICROCLOUD-PROVIDER-DEPT",  configuration = FeignClientConfig.class,  fallbackFactory = DeptClientServiceFallbackFactory.class) public interface DeptClientService {  @RequestMapping(method = RequestMethod.*GET*, value = "/mohist-proxy/dept-proxy/dept/get/{id}")  public Dept get(@PathVariable("id") long id);  @RequestMapping(method = RequestMethod.*GET*, value = "/mohist-proxy/dept-proxy/dept/list")  public List<Dept> list();  @RequestMapping(method = RequestMethod.*GET*, value = "/mohist-proxy/dept-proxy/dept/save")  public boolean save(Dept dept); } |

1. microcloud-service，修改服务的配置类，测试访问的应该是zuul的地址；

|  |
| --- |
| package com.kuhnwei.common.config; import feign.Logger; import feign.auth.BasicAuthRequestInterceptor; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;@Configuration public class FeignClientConfig {  @Bean  public Logger.Level getFeignLoggerLevel() {  return Logger.Level.*FULL*;  }  @Bean  public BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  return new BasicAuthRequestInterceptor("zdmin", "650901");  } } |

1. microcloud-gateway，默认情况下只要配置了过滤器，就可以进行一个正常的启动，如果现在有些过滤器突然不想让它用了，则也可以通过修改application.yml配置文件让其禁用；

|  |
| --- |
| **zuul:**  **AuthorizedRequestFilter:  pre:   disable:** true |

此时表示“AuthorizedRequestFilter”过滤器将被禁止使用，如果以后有多个过滤器服务出现的话，可以通过配置文件实现过滤的启用与禁用控制。

## Zuul服务降级

Zuul是一个代理服务，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。例如：现在停掉“dept-8001.com:8001”端口上的服务，而后输入以下的代理地址：

|  |
| --- |
| <http://zdmin:650901@gateway-9501.com:9501/mohist-proxy/dept-proxy/dept/get/1> |



如果现在程序无法使用，则zuul的代理执行时就会出现有timeout的信息。但是千万要记住，由于现在的客户端已经提供有了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。但是对于一个完善的zuul的代理应该更好的实现服务降级的处理操作，所以如果有需要也可以在zuul中进行服务降级配置。

1. microcloud-zuul-gate，建立一个Fallback的回退处理类。

|  |
| --- |
| package com.kuhnwei.microcloud.fallback; 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; import java.io.ByteArrayInputStream; import java.io.IOException; import java.io.InputStream;@Component public class DeptProviderFallback implements ZuulFallbackProvider {  @Override  public String getRoute() {  return "microcloud-provider-dept";  }  @Override  public ClientHttpResponse fallbackResponse() {  return new ClientHttpResponse() {  @Override  public HttpStatus getStatusCode() throws IOException {  return HttpStatus.*BAD\_REQUEST*;  }  @Override  public int getRawStatusCode() throws IOException {  return HttpStatus.*BAD\_REQUEST*.value();  }  @Override  public String getStatusText() throws IOException {  return HttpStatus.*BAD\_REQUEST*.getReasonPhrase();  }  @Override  public void close() {   }  @Override  public InputStream getBody() throws IOException {  // 当出现服务调用错误之后返回的数据内容  return new ByteArrayInputStream("{\"deptno\":7777777,\"dname\":\"[ERROR]Zuul-Fallback\",\"loc\":\"Gateway客户端提供\"}".getBytes());  }  @Override  public HttpHeaders getHeaders() {  HttpHeaders headers = new HttpHeaders();  headers.set("Content-Type", "text/html, charset=UTF-8");  return headers;  }  };  } } |

1. microcloud-zuul-gateway，直接访问地址；

|  |
| --- |
| <http://zdmin:650901@gateway-9501.com:9501/mohist-proxy/dept-proxy/dept/get/1> |

由于此时你返回的都是错误的代码，则客户端接收到此代码之后会认为服务器端已经死掉了。

# 上传微服务

学习完了Zuul的代理机制，但是需要强调的是，在使用Zuul代理微服务的时候实际上也可以进行上传微服务的代理。

## 建立上传微服务

1. 建议通过之前的项目随意做一个简单的复制，复制一个新的项目：microcloud-provider-upload-8201，在这个项目之中不再需要任何的MyBatis操作等等，所以将所有无用的配置全部删除掉；
2. microcloud-provider-upload-8201，修改application.yml配置文件，对上传文件做一些基础配置。

|  |
| --- |
| **info:  app.name:** mohist-microcloud  **company.name:** www.kuhnwei.com  **build.artifactId:** $project.artifactiId$  **build.version:** $project.version$ **server:  port:** 8201 **spring:  application:  name:** microcloud-provider-upload  **http:  multipart:  enabled:** true  **max-file-size:** 100MB  **max-request-size:** 100MB  **file-size-threshold:** 1MB  **location:** / **eureka:  client:  service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **instance:  instance-id:** upload-8201.com  **prefer-ip-address:** true  **lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90  **hystrix:  config:  stream:  maxConcurrentConnections:** 50 |

1. microcloud-provider-upload-8201，建立上传的Rest服务配置；

|  |
| --- |
| package com.kuhnwei.microcloud.rest; import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.bind.annotation.RequestParam; import org.springframework.web.bind.annotation.RestController; import org.springframework.web.multipart.MultipartFile;@RestController public class UploadRest {  @HystrixCommand(fallbackMethod = "uploadFallback")  @RequestMapping(value = "/upload", method = RequestMethod.*POST*)  public String upload(@RequestParam("photo")MultipartFile photo) {  if (photo != null) {  System.*out*.println("[UploadRest]文件名称：" + photo.getOriginalFilename() + "、文件大小：" + photo.getSize());  }  return "mohist-file-" + System.*currentTimeMillis*() + ".jpg";  }  public String uploadFallback(@RequestParam("photo") MultipartFile photo) {  return "nophoto.jpg";  } } |

现在实现的上传处理操作与之前讲解SpringBoot的形式是几乎完全一样的。

1. 如果要进行上传测试，可以简单一些，使用curl的命令完成，当然，这个命令是需要单独配置的；
   1. 为了方便配置修改hosts配置文件，追加一个访问映射路径；

|  |
| --- |
| 127.0.0.1 upload-8201.com |

* 1. 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=@dog.jpg” <http://mohist:650901@upload-8201.com:8201/upload> |

这个时候如果你的csrf跨站请求配置被关闭了，则会出现如下的错误提示信息：

|  |
| --- |
|  |

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，那么这个时候就不能禁止了。

1. microcloud-security，修改安全策略，将csrf禁用；

|  |
| --- |
| @Override protected void configure(HttpSecurity httpSecurity) throws Exception {  // 表示所有的访问都必须进行认证处理后才可以正常进行  httpSecurity.httpBasic().and().authorizeRequests().anyRequest().fullyAuthenticated()  .and().csrf().disable();  // 所有的Rest服务一定要设置为无状态，以提升操作性能  httpSecurity.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*); } |

此时表示取消了csrf的验证操作，那么久证明当前的微服务允许跨站访问。那么这个时候文件就可以正常进行上传处理了。

## 通过Zuul代理上传微服务

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户端直接调用了指定的微服务的信息来操作，从实际的开发来讲，所有的微服务应该都被zuul进行代理上传。

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；

|  |
| --- |
| **zuul:  prefix:** /mohist-proxy  **ignored-services:** "\*"  **routes:  microcloud-provider-company:** /company-proxy/\*\*  **microcloud-provider-dept:** /dept-proxy/\*\*  **microcloud-provider-upload:** /upload-proxy/\*\*  **AuthorizedRequestFilter:  pre:  disable:** false |

修改完成之后来启动zuul的代理微服务；

1. 既然现在zuul已经正常启动了，那么下面将利用zuul作为上传微服务的代理操作，于是命令上可以使用curl命令进行上传配置，路径：curl –F “photo=@dog.jpg” <http://zdmin:650901@gateway-9501.com:9501/mohist-proxy/upload-proxy/upload>.
2. 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的上传文件大小：100M，于是现在上传一个16M的大小文件，访问路径：

|  |  |
| --- | --- |
| curl –F “photo=@test.jpg” <http://zdmin:650901@gateway-9501.com:9501/mohist-proxy/upload-proxy/upload> | |
| 错误信息： |  |

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

|  |
| --- |
| curl –F “photo=@test.jpg” <http://zdmin:650901@gateway-9501.com:9501/zuul/mohist-proxy/upload-proxy/upload> |

那么此时就表示当前的上传的代理操作，zuul不再进行限制，而是直接交由目的微服务进行。

1. microcloud-zuul-gateway-9501，修改application.yml配置文件，进行超时时间的配置：

|  |
| --- |
| **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，才可以正常实现上传。

## 客户端调用上传微服务

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实际的开发之中使用，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户端（WEB端、SpringBoot实现）调用zuul，而后再由zuul去代理上传微服务。

于是这个时候大家肯定首先想要的就是Feign，但是非常遗憾的告诉大家，对于上传的这种微服务的操作时无法利用Feign做接口转换，也无法直接使用RestTemplate做代理操作。也就是说此时如果要想调用，那么唯一的方案就是利用httpclient完成。

1. 建立一个新的客户端项目：microcloud-consumer-upload，而后这个项目主要是运行SpringBoot程序，同时要结合thymeleaf作为页面显示；
2. microcloud-consumer-upload，修改pom.xml配置文件，这个配置文件里面需要追加httpclient的两个依赖包；

|  |
| --- |
| <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId> </dependency> <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpmime</artifactId> </dependency> <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId> </dependency> |

1. microcloud-consumer-upload，修改application.yml配置文件，在这个配置文件里面追加如下的配置项：

|  |
| --- |
| **server:  port:** 80 **eureka:  client:  register-with-eureka:** false  **service-url:  defaultZone:** http://edmin:650901@eureka-7001.com:7001/eureka,http://edmin:650901@eureka-7002.com:7002/eureka,http://edmin:650901@eureka-7003.com:7003/eureka  **instance:  instance-id:** upload.com  **prefer-ip-address:** true  **lease-renewal-interval-in-seconds:** 30  **lease-expiration-duration-in-seconds:** 90 |

1. microcloud-consumer-upload，建立一个src/main/view的目录，在这个目录里面主要是将其提升为源文件目录，随后将所需要的页面文件设置到此目录之中，并且在里面建立static、templates两个子目录；
2. microcloud-consumer-upload，建立src/mian/view/templates/upload.html页面

|  |
| --- |
| <!DOCTYPE html> <html lang="en" xmlns:th="http://www.w3.org/1999/xhtml"> <head>  <meta charset="UTF-8"/>  <title>SpringCloud微服务</title> </head> <body> <form th:action="@{/consumer/upload}" method="post" enctype="multipart/form-data">  姓名：<input type="text" name="name" id="name" value="MOHIST" /><br/>  照片：<input type="file" name="photo" id="photo" /><br/>  <input type="submit" value="提交"/>  <input type="reset" value="重置"/> </form> </body> </html> |

1. microcloud-consumer-upload，建立一个控制层的程序代码。

|  |
| --- |
| package com.kuhnwei.microcloud.controller; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.multipart.MultipartFile;@Controller public class ConsumerUploadController {  @RequestMapping(value = "/consumer/upload.html", method = RequestMethod.*GET*)  public String uploadPre() {  return "upload";  }  @ResponseBody  @RequestMapping(value = "/consumer/upload", method = RequestMethod.*POST*)  public String upload(String name, MultipartFile photo) {  if (photo != null) {  return "[消费端] name = " + name  + "、photoName = " + photo.getOriginalFilename()  + "、ContentType = " + photo.getContentType();  }  return "nophoto.jpg";  } } |

1. 修改hosts配置文件，追加映射的域名：

|  |
| --- |
| 127.0.0.1 upload.com |

1. microcloud-consumer-upload，实现上传微服务的调用；
   1. 访问路径：<http://upload.com/consumer/upload.html>；
2. microcloud-consumer-upload，通过httpclient进行远程调用：
   1. 远程上传微服务的代理地址为：<http://zdmin:650901@gateway-9501.com:9501/zuul/mohist-proxy/upload-proxy/upload>

|  |
| --- |
| package com.kuhnwei.microcloud.controller; import org.apache.http.HttpEntity; import org.apache.http.HttpResponse; 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.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 java.nio.charset.Charset;@Controller public class ConsumerUploadController {   // 设置要进行远程上传微服务调用的代理地址  public static final String *UPLOAD\_URL* = "http://gateway-9501.com:9501/zuul/mohist-proxy/upload-proxy/upload";   @RequestMapping(value = "/consumer/upload.html", method = RequestMethod.*GET*)  public String uploadPre() {  return "upload";  }   @ResponseBody  @RequestMapping(value = "/consumer/upload", method = RequestMethod.*POST*)  public String upload(String name, MultipartFile photo) throws Exception {  if (photo != null) {  // 创建一个HttpClient对象  CloseableHttpClient httpClient = HttpClients.*createDefault*();  // 创建一个具有认证访问的信息  CredentialsProvider credsProvider = new BasicCredentialsProvider();  // 创建一条认证操作信息  Credentials credentials = new UsernamePasswordCredentials("zdmin", "650901");  // 闲杂所有的认证请求都使用一个认证信息  credsProvider.setCredentials(AuthScope.*ANY*, credentials);  // 创建http处理操作的上下文对象  HttpClientContext httpContext = HttpClientContext.*create*();  httpContext.setCredentialsProvider(credsProvider);  // 设置要进行访问的请求地址  HttpPost httpPost = new HttpPost(*UPLOAD\_URL*);  HttpEntity entity = MultipartEntityBuilder.*create*()  .addBinaryBody("photo", photo.getBytes(), ContentType.*create*(photo.getContentType()),"temp.jpg")  .build();  // 将请求的实体信息进行发送  httpPost.setEntity(entity);  // 执行请求的发送  HttpResponse response = httpClient.execute(httpPost, httpContext);  return EntityUtils.*toString*(response.getEntity(), Charset.*forName*("UTF-8")); // return "[消费端] name = " + name + "、photoName = " + photo.getOriginalFilename() // + "、ContentType = " + photo.getContentType();  }  return "nophoto.jpg";  } } |

现在是伪造了一个http协议的post请求，才将突破传递到了上传微服务之中。