# springcloud开发框架

## 分布式开发与SpringCloud简介

### 分布式开发简介

对于分布式开发设计，最初就有了以下几点思考：

1. 分布式开发如何可以让代码更加安全
2. 分布式开发的时候如何进行有效的通信
3. 在进行分布式处理的时候到底如何进行程序的功能划分

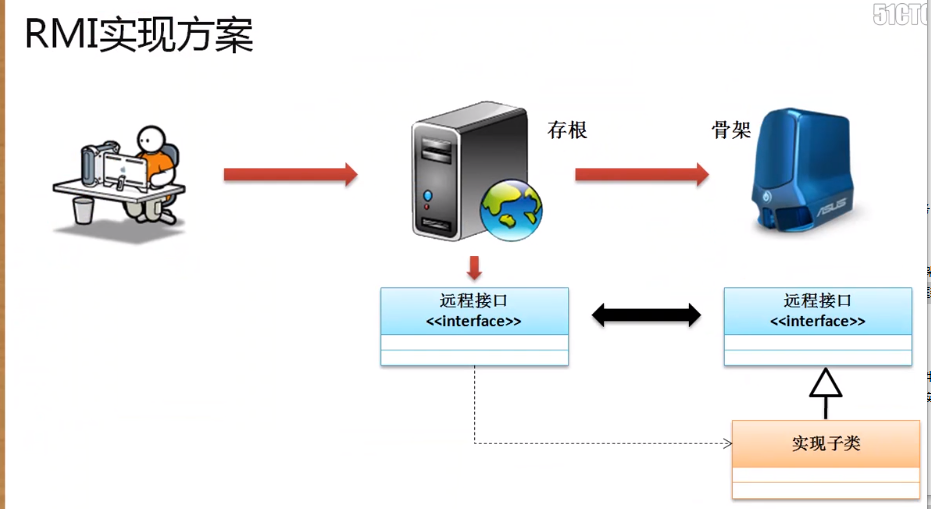
项目开发中分布式设计：

1. WEB集群：考虑到多用户并发访问的处理速度；
2. 业务中心：在进行一些庞大的项目设计过程中，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；
3. 数据库集群：解决了数据的存储问题，以及数据的分片管理。

对于分布式项目开发按照历史的发展经历过如下的一些技术：

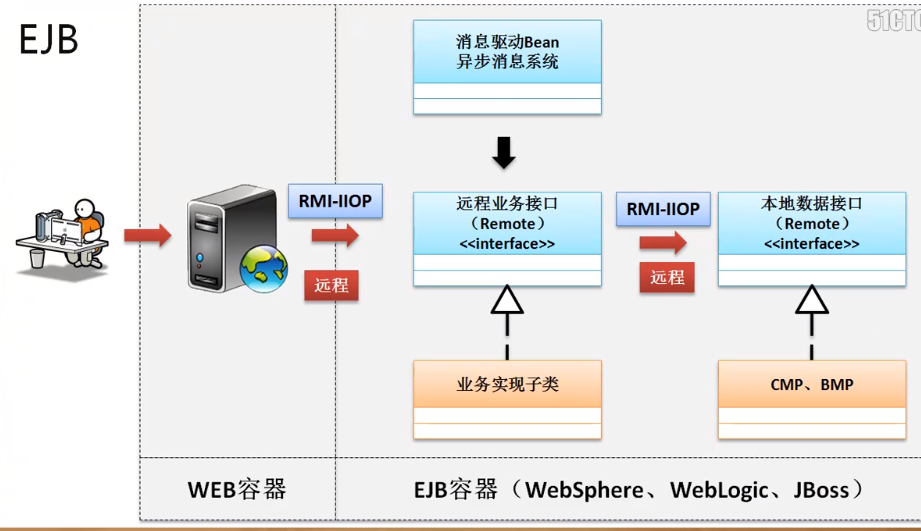
1. CORBA：公共对象请求代理架构，他是一种开发的标准，而且也是许多语音都支持的开发标准。
2. RMI(远程方法调用)：该技术是SUN提出来的，其是希望与CORBA进行市场竞争，很多公司不认可此技术；

|- 在java里面提出了远程接口的概念，不过RMI的实现不太好。



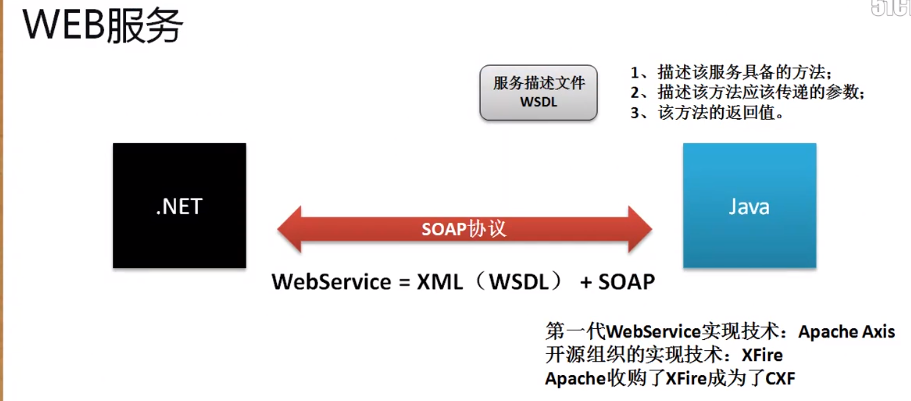
开发人员除了编写接口外，还要写存根和骨架。这比较麻烦。

1. 后来SUN事务设计师，RMI很好用（他们自己觉得），但是没有CORBA广泛，于是开始思考能否基于CORBA做一些更好的设计？于是产生了RMI-IIOP协议，其用在了EJB上。



EJB留给世界上的只是它的优秀理论，和糟糕的实现，这个理论被一些开源框架无限制的扩充与实现着。

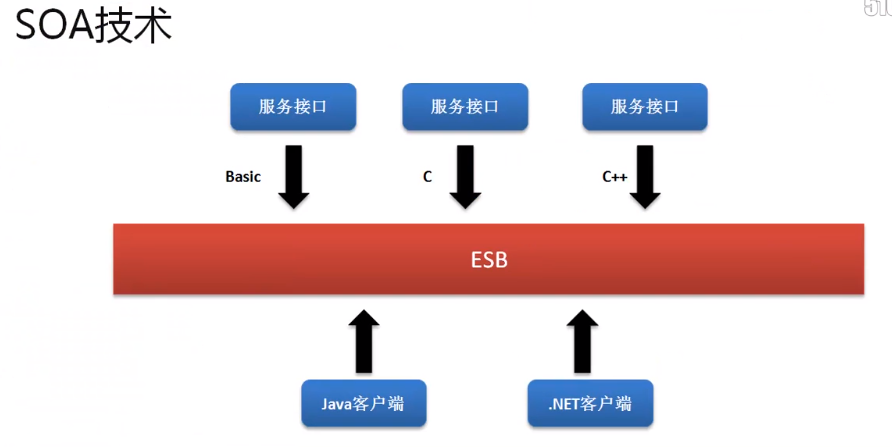
1. .NET出现，而后因为J#的问题，微软和SUN就彻底对立，后来整个行业就乱了，有两套企业系统架构，公司就面临选择，于是尴尬的局面出现了。选择谁？后来随着软件行业的继续推广，出现了XML标记语言，很多公司就觉得应该用XML作为数据交换的基础，整个社会著名的软件架构：WebService（web服务）登场了。



如果项目中要想使用WebService技术进行开发，择优如下问题：

* 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；

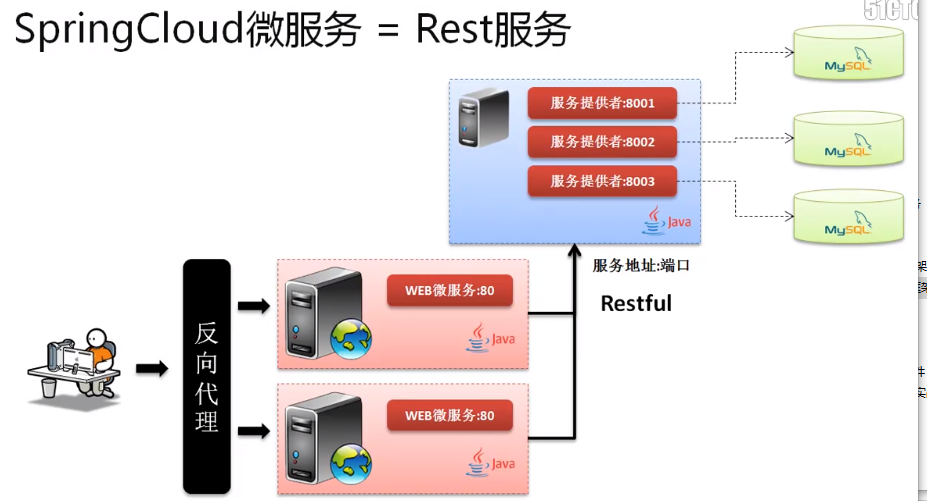
后来2005年的时候WebService在整个世界上继续发酵，形成了又一大核心神器：SOA(面向服务架构)，它提出了企业服务总线（ESB）的概念。



(5)SOA提出的服务总线又成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，就成为了服务总线，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，其中具有代表性的dubbo(阿里提供)开发技术。不过这个时候成长最快的就是Rest,因为JSON的广泛应用，如同最初的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，随着Rest的广泛认可，Spring终于得到了一个新机会：利用Rest进行RPC技术实现，这样的操作速度很快，而且占用额网络带宽要少。在SpringCloud中真正的将整个Rest作为了RPC实现技术,并且这一技术已经开始出现有行业统一之势。SpringBoot也依照与SpringCloud的开发技术，可以实现项目的打包分布以及单独运行，这一点符合当前云时代的开发要求。

### SpringCloud开发

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，所有的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：服务的提供者（provider）、服务的消费者（consumer），所以SpringCloud的基础结构就如下所示：

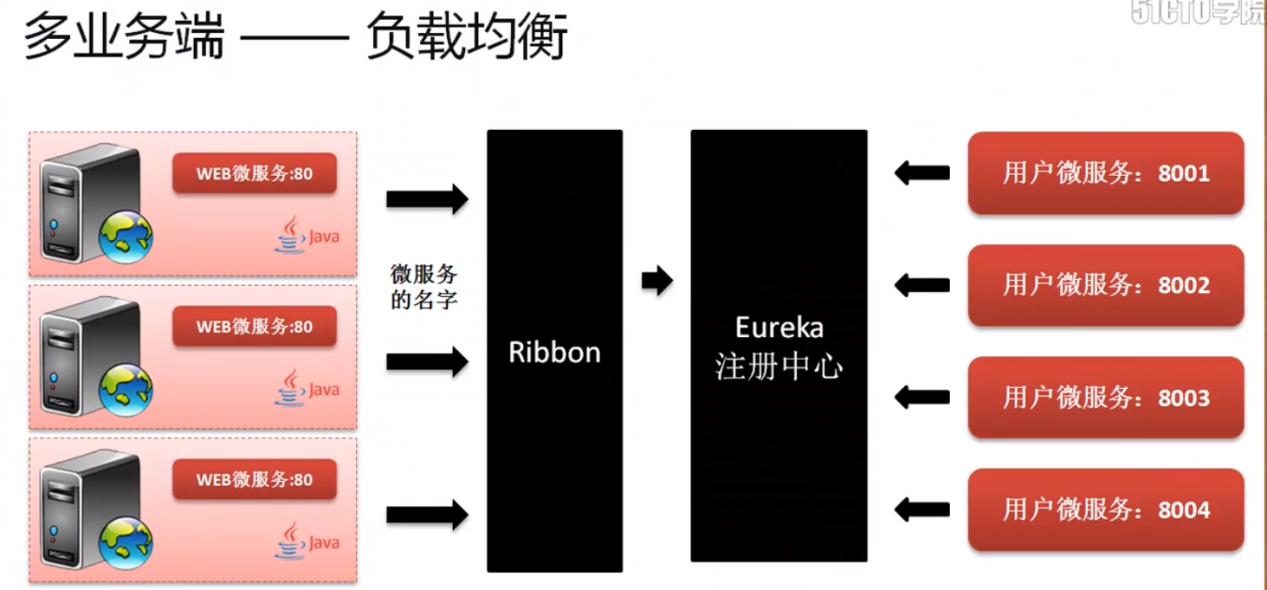




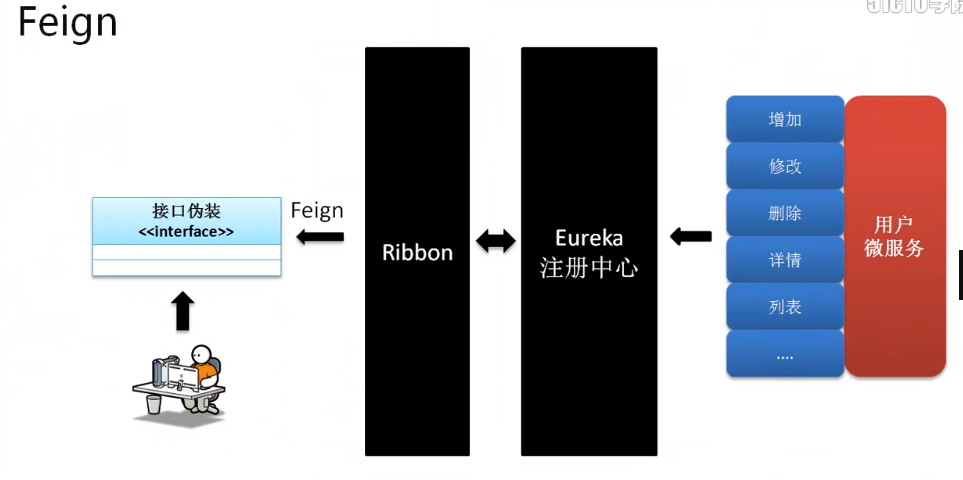
既然SpringCloud的核心是Restful结构，那么如果想要更好的去使用Rest微服务，还需要考虑如下几个问题：

1. 所有的微服务地址一定会非常多，所以为了统一管理这些地址信息，也为了可以及时的告诉用户哪些服务不可用，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HA机制。
2. 为高速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（虽然spring支持Zookeeper和Consul,但是性能都很差）。

对于整个的WEB端架构（SpringBoot实现）可以轻松过方便的进行WEB程序的编写，而后利用Nginx或Apache实现负载均衡，但是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中进行注册。

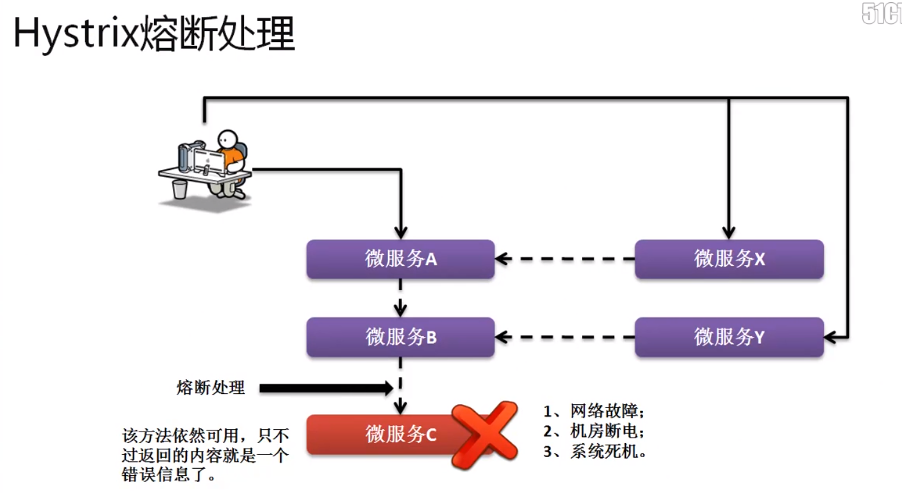


在客户端进行Rest架构调用的时候，往往都需要有一个调用地址，即使现在使用了Eureka作为注册中心，它也需要有一个明确的调用地址，可是所有的操作都利用调用地址的方式来处理，那就比较麻烦，而持续开发者最方便应用的工具是接口，所以现在就希望可以将所有的rest服务的内容以接口的形式调用，所以它又提供了一个feign技术，利用此技术可以伪造接口实现。

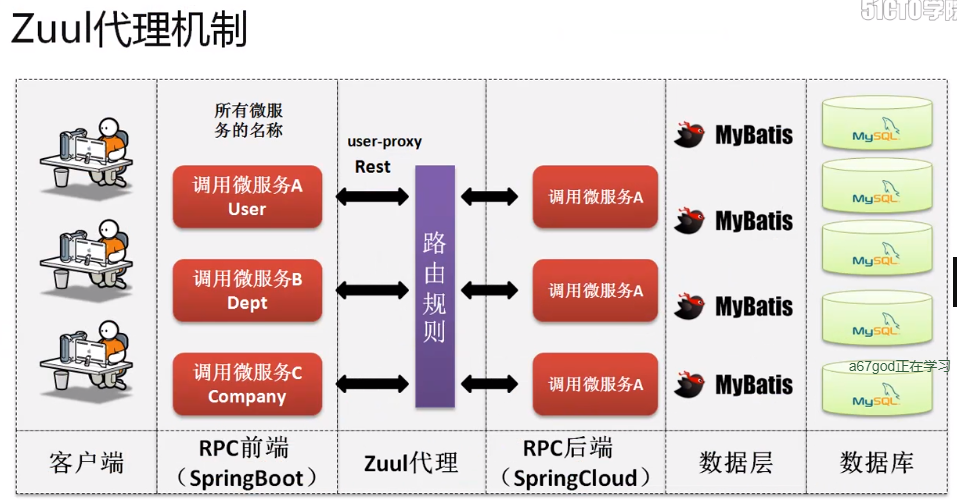


1. 在微架构设计的时候由于牵扯到的问题还是属于RPC，所以必须考虑熔断处理机制，实际上所谓的熔断就像生活中使用的保险丝一样，有了保险丝，在一些设备出现故障之后依然可以保护家庭电器可以正常使用，如果现在有若干个微服务，并且这些微服务之间允许互相调用，例如：A微服务调用了B的微服务、B的微服务又调用了C的微服务。

如果在实际的项目设计过程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可以正常使用。

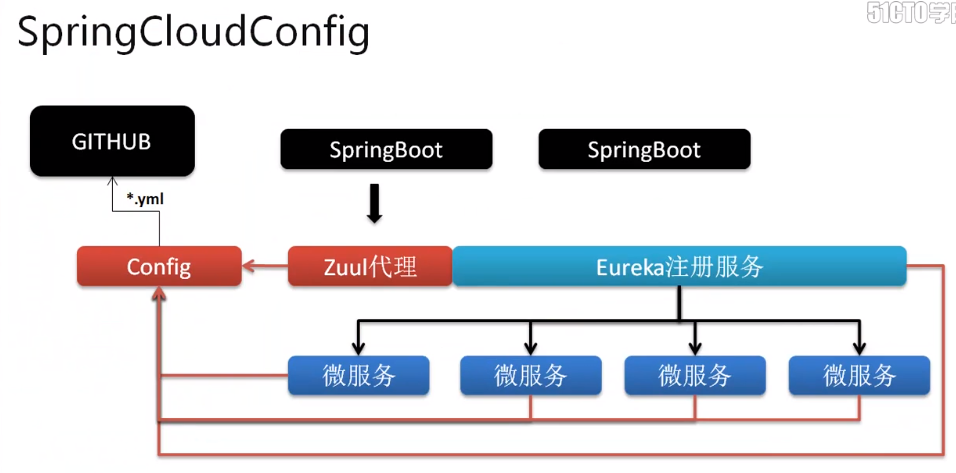


1. 在进行为服务访问的时候还有一点是非常可怕的。



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“Key=value”的设计思想，所有的微服务进行代理之后也可以更加合理的进行名称的隐藏。

1. 在SpringBoot学习的时候一直强调一个问题：“零配置”，本质上是希望不编写任何配置文件，但是事实上这一点并没有完全的实现，因为在整个设计里面，依然会提供有application.yml（application.properties）配置文件。在微服务的创建过程中，一定会有成百上千个微服务的信息出现，这些配置文件的管理就成为了问题，例如：某一天机房变更，那么所有的Ip地址都有可能发生变化，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netflix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

对于Rest基础架构实现处理时SpringCloud核心所在，其基本的操作形式在SpringBoot中已经有了基本的认识。

### 配置项目开发环境

对于现在的项目创建一个：microcloud的整体父pom项目，然后创建三个子模块：microcloud-api模块，作为公共的信息导入模块、microcloud-provider-dept-8001作为服务提供者，负责使用mybatis与数据库的交互、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；

注意：SpringBoot中针对于依赖包的版本并不是像传统那样采用数字的形式定义的，而是使用一序列的英国的地铁站或城市的名字来定义的。

*<?***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**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**properties**>  
 <**jdk.version**>1.8</**jdk.version**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**> *<!-- 进行SpringCloud依赖包的导入处理 -->* <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>Dalston.SR5</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 <**dependency**> *<!-- SpringCloud离不开SpringBoot，所以必须要配置此依赖包 -->* <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-dependencies</**artifactId**>  
 <**version**>1.5.14.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**>${jdk.version}</**source**>*<!-- 源代码使用的开发版本 -->* <**target**>${jdk.version}</**target**>*<!-- 需要生成的目标class文件的编译版本 -->* <**encoding**>${project.build.sourceEncoding}</**encoding**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
 </**build**>  
</**project**>

1. 【microcloud-api】主要功能是提供有公共的处理类，本次预计建立一个Dept数据表（实体）：

**public class** Dept **implements** Serializable {  
 **private** Long **deptno**;  
 **private** String **dname**;  
 **private** String **loc**;  
  
 **public** Long getDeptno() {  
 **return deptno**;  
 }  
  
 **public void** setDeptno(Long deptno) {  
 **this**.**deptno** = deptno;  
 }  
  
 **public** String getDname() {  
 **return dname**;  
 }  
  
 **public void** setDname(String dname) {  
 **this**.**dname** = dname;  
 }  
  
 **public** String getLoc() {  
 **return loc**;  
 }  
  
 **public void** setLoc(String loc) {  
 **this**.**loc** = loc;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Dept [deptno="** + **deptno** + **", dname="** + **dname** + **", loc="** + **loc** + **"]"**;  
 }  
}

1. 【microcloud-provider-dept-8001】创建一个Rest提供者的项目模块，在整个模块里面主要定义数据库的脚本：

DROP DATABASE IF EXISTS "yangmi8001" ;

CREATE DATABASE yangmi8001 CHARACTER SET UTF8 ;

USE yangmi8001 ;

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()) ;

### 创建部门微服务

所谓微服务的核心本质就是JSON的传输，那么既然现在要求用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，追加相关依赖：

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-provider-dept-8001</**artifactId**>  
  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>

<**version**>1.0-SNAPSHOT</**version**>  
 </**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**>  
</**project**>

同时修改父项目pom.xml：

<**dependencies**>  
 <**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>  
 <**version**>1.0</**version**>  
 </**dependency**>  
  
 <**dependency**> *<!-- 进行SpringCloud依赖包的导入处理 -->* <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>Dalston.SR1</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 <**dependency**> *<!-- SpringCloud离不开SpringBoot，所以必须要配置此依赖包 -->* <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-dependencies</**artifactId**>  
 <**version**>1.5.4.RELEASE</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.0.4</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>druid</**artifactId**>  
 <**version**>1.0.31</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>1.3.0</**version**>  
 </**dependency**>  
</**dependencies**>

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

@Mapper  
**public interface** IDeptDAO {  
 **public boolean** doCreate(Dept vo);  
  
 **public** Dept findById(Long id);  
  
 **public** List<Dept> findAll();  
}

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

**server:  
 port:** 8001  
**mybatis:  
 config-location:** classpath:mybatis/mybatis.config.xml *# mybatis配置文件所在路径* **type-aliases-package:** cn.ym.com.vo *# 定义所有操作类的别名所在包* **mapper-locations:** *# 所有的mapper映射文件* - classpath:mybatis/mapper/\*\*/\*.xml  
**spring:  
 datasource:  
 type:** com.alibaba.druid.pool.DruidDataSource *# 配置当前要使用的数据源的操作类型* **driver-class-name:** org.gjt.mm.mysql.Driver *# 配置MySQL的驱动程序类* **url:** jdbc:mysql://localhost:3306/yangmi8001 *# 数据库连接地址* **username:** root *# 数据库用户名* **password:** 12345 *# 数据库连接密码* **dbcp2:** *# 进行数据库连接池的配置* **min-idle:** 5 *# 数据库连接池的最小维持连接数* **initial-size:** 5 *# 初始化提供的连接数* **max-total:** 5 *# 最大的连接数* **max-wait-millis:** 200 *# 等待连接获取的最大超时时间*

1. 【microcloud-provider-dept-8001】定义src/main/resources/mybatis/mybatis.config.xml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**> *<!-- 进行Mybatis的相应的环境的属性定义 -->* <**settings**> *<!-- 在本项目之中开启二级缓存 -->* <**setting name="cacheEnabled" value="true"**/>  
 </**settings**>  
</**configuration**>

5、【microcloud-provider-dept-8001】创建

src/main/resources/mybatis/mapper/cn/ym/com/Dept.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="cn.ym.com.dao.IDeptDAO"**>  
 <**select id="findById" resultType="Dept" parameterType="long"**>  
 SELECT deptno,dname,loc FROM dept WHERE deptno=#{deptno} ;  
 </**select**>  
 <**select id="findAll" resultType="Dept"**>  
 SELECT deptno,dname,loc FROM dept ;  
 </**select**>  
 <**insert id="doCreate" parameterType="Dept"**>  
 INSERT INTO dept(dname,loc) VALUES (#{dname},database()) ;  
 </**insert**>  
</**mapper**>

6、【microcloud-provider-dept-8001】建立IDeptService接口，做业务实现：

**package** cn.ym.com.service;  
  
**import** cn.ym.com.vo.Dept;  
  
**import** java.util.List;  
  
**public interface** IDeptService {  
 **public** Dept get(**long** id);  
  
 **public boolean** add(Dept dept);  
  
 **public** List<Dept> list();  
}

**package** cn.ym.com.service.impl;  
  
**import** cn.ym.com.dao.IDeptDAO;  
**import** cn.ym.com.service.IDeptService;  
**import** cn.ym.com.vo.Dept;  
**import** org.springframework.stereotype.Service;  
**import** javax.annotation.Resource;  
**import** java.util.List;  
  
@Service  
**public class** DeptServiceImpl **implements** IDeptService {  
 @Resource  
 **private** IDeptDAO **deptDAO**;  
  
 @Override  
 **public** Dept get(**long** id) {  
 **return this**.**deptDAO**.findById(id);  
 }  
  
 @Override  
 **public boolean** add(Dept dept) {  
 **return this**.**deptDAO**.doCreate(dept);  
 }  
  
 @Override  
 **public** List<Dept> list() {  
 **return this**.**deptDAO**.findAll();  
 }  
  
}

7、【microcloud-provider-dept-8001】定义程序启动类：

@SpringBootApplication  
**public class** Dept\_8001\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Dept\_8001\_StartSpringCloudApplication.**class**, args);  
 }  
}

8、【microcloud-provider-dept-8001】编写业务接口测试类：

1. 【microcloud-provider-dept-8001】建立DeptRest服务类（控制器）：

@RestController  
**public class** DeptController {  
 @Resource  
 **private** IDeptService **deptService**;  
  
 @RequestMapping(value = **"/dept/get/{id}"**, method = RequestMethod.***GET***)  
 **public** Object get(@PathVariable(**"id"**) **long** id) {  
 **return this**.**deptService**.get(id);  
 }  
  
 @RequestMapping(value = **"/dept/add"**, method = RequestMethod.***GET***)  
 **public** Object add(@RequestBody Dept dept) {  
 **return this**.**deptService**.add(dept);  
 }  
  
 @RequestMapping(value = **"/dept/list"**, method = RequestMethod.***GET***)  
 **public** Object list() {  
 **return this**.**deptService**.list();  
 }  
}

1. 修改本机host文件，追加一个映射路径：

127.0.0.1 dept-8001.com

1. 【microcloud-provider-dept-8001】观察Rest服务能否正常提供：
2. 调用get操作

http://dept-8001.com:8001/dept/get/1

1. 调用list操作

http://dept-8001.com:8001/dept/list

1. 调用add操作

http://dept-8001.com:8001/dept/add?deptno=100&dname=%E7%BE%8E%E5%A5%B3%E9%83%A8&loc=%E5%A4%A9%E7%95%8C

### 创建服务消费端

1. 创建一个Maven的新项目：mircrocloud-consumer-80

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-consumer-80</**artifactId**>  
  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**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**>  
</**project**>

1. 【mircrocloud-consumer-80】修改application.yml配置文件：

**server:  
 port:** 80

3、【mircrocloud-consumer-80】创建一个Rest的配置程序类。主要进行RestTemplate类对象创建：

@Configuration  
**public class** RestConfig {  
  
 @Bean  
 **public** RestTemplate getRestTemplate() {  
 **return new** RestTemplate();  
 }  
}

4、【mircrocloud-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，所有服务的结果都使用Rest返回：

@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\_ADD\_URL*** = **"http://dept-8001.com:8001/dept/add"**;  
  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @RequestMapping(value = **"/consumer/dept/get"**)  
 **public** Object getDept(**long** id) {  
 **return this**.**restTemplate**.getForObject(***DEPT\_GET\_URL*** + id, Dept.**class**);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/list"**)  
 **public** Object listDept() {  
 **return this**.**restTemplate**.getForObject(***DEPT\_LIST\_URL***, List.**class**);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/add"**)  
 **public** Object addDept(Dept dept) {  
 **return this**.**restTemplate**.postForObject(***DEPT\_ADD\_URL***, dept, Boolean.**class**);  
 }  
}

5、【mircrocloud-consumer-80】编写程序启动类

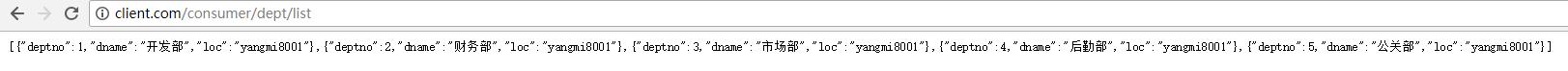
**package** cn.ym.com;  
  
**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.autoconfigure.SpringBootApplication;  
@SpringBootApplication  
**public class** Consumer\_80\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Consumer\_80\_StartSpringCloudApplication.*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

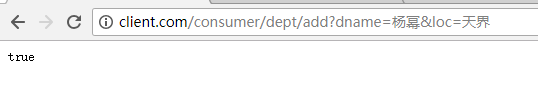
6、修改hosts配置文件，追加访问控制路径：

127.0.0.1 client.com

7、【mircrocloud-consumer-80】测试调用







## SpringSecutity安全认证

### Rest提供者安全访问

要进行安全的一种处理，首先一定要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：

<dependency>

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

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

<version>2.0.3.RELEASE</version>

</dependency>

如果现在配置了安全框架，则在启动的时候会有如下的提示信息：

Using default security password: cccdd4bb-e5e1-49fb-9582-ded7f8a85b22

1. 【microcloud-provider-dept-8001】修改application.yml配置文件，进行安全相关的用户名配置：

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

随后访问接口：http://dept-8001.com:8001/dept/list，此时需要认证信息：

也可以有一种简化的访问：<http://yangmi:ym@dept-8001.com:8001/dept/list>

### 配置消费端访问

所有的认证处理操作，应该以以头信息的模式进行处理。要通过base64加密处理后才可以得到一个正确的访问路径。

1. 【microcloud-consumer-80】修改Rest配置类，追加新的配置项：

*/\*\*  
 \* 配置httpheader头信息  
 \*  
 \** ***@return*** *\*/*@Bean  
**public** HttpHeaders getHeaders() {  
 *//定义httpheader头信息* HttpHeaders httpHeaders = **new** HttpHeaders();  
 String auth = **"yangmi:ym"**;  
 *//加密的处理* **byte**[] encodeAuth = 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(encodeAuth);  
 httpHeaders.set(**"Authorization"**, authHeader);  
 **return** httpHeaders;  
}

1. 【microcloud-consumer-80】修改ConsumerDeptController配置类，在进行访问的时候设置好头信息：

@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\_ADD\_URL*** = **"http://dept-8001.com:8001/dept/add"**;  
  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @Resource  
 **private** HttpHeaders **httpHeaders**;  
  
 @RequestMapping(value = **"/consumer/dept/get"**)  
 **public** Object getDept(**long** id) {  
 **return this**.**restTemplate**.exchange(***DEPT\_GET\_URL*** + id, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), Dept.**class**);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/list"**)  
 **public** Object listDept() {  
 **return this**.**restTemplate**.exchange(***DEPT\_LIST\_URL***, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), List.**class**);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/add"**)  
 **public** Object addDept(Dept dept) {  
 **return this**.**restTemplate**.exchange(***DEPT\_ADD\_URL***, HttpMethod.***POST***, **new** HttpEntity<Object>(dept, **this**.**httpHeaders**), Boolean.**class**);  
 }  
}

### 无状态Rest配置（Session）

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

1. 【microcloud-provider-dept-8001】修改Rest程序类，追加一个取得session id的方法：

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

随后访问这个rest：<http://dept-8001.com:8001/dept/session>

每次访问，sessio都不同。

1. 在一些的SpringCloud的配置之中，默认是会保存有Session状态的，而后如果需要修改则可以根据“SessionCreationPolicy”枚举类，修改application.yml进行不同的session配置，但是从整体的操作来说，session最好设置为无状态（服务器端不保存session状态）。

**package** org.springframework.security.config.http;  
  
**public enum** SessionCreationPolicy {  
 ALWAYS,  
 NEVER,  
 IF\_REQUIRED,  
 STATELESS;  
  
 **private** SessionCreationPolicy() {  
 }  
}

1. 以下为保持Session状态（服务器内存有可能被占满）：

security:

sessions: always

(2)以下为无状态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

不管以后的项目中或者支持类中是否有设置无状态的问题，最好都进行设置为无状态，负责服务器性能将受到影响（内存溢出）。

### 4、安全配置模块

1、创建一个microcloud-security的Maven模块：

2、【microcloud-security】修改pox.xml配置文件

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-security</**artifactId**>  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-security</**artifactId**>  
 </**dependency**>  
 </**dependencies**>  
  
</**project**>

1. 【microcloud-security】建立一个统一的安全配置类：

@Configuration  
@EnableWebSecurity  
**public class** WebSecurityConfig **extends** WebSecurityConfigurerAdapter {  
  
 @Resource  
 **public void** configGlobal(AuthenticationManagerBuilder authenticationManagerBuilder) **throws** Exception {  
 authenticationManagerBuilder.inMemoryAuthentication().withUser(**"yangmi"**).password(**"ym"**).roles(**"USER"**).and().withUser(**"admin"**).password(**"admin"**).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：

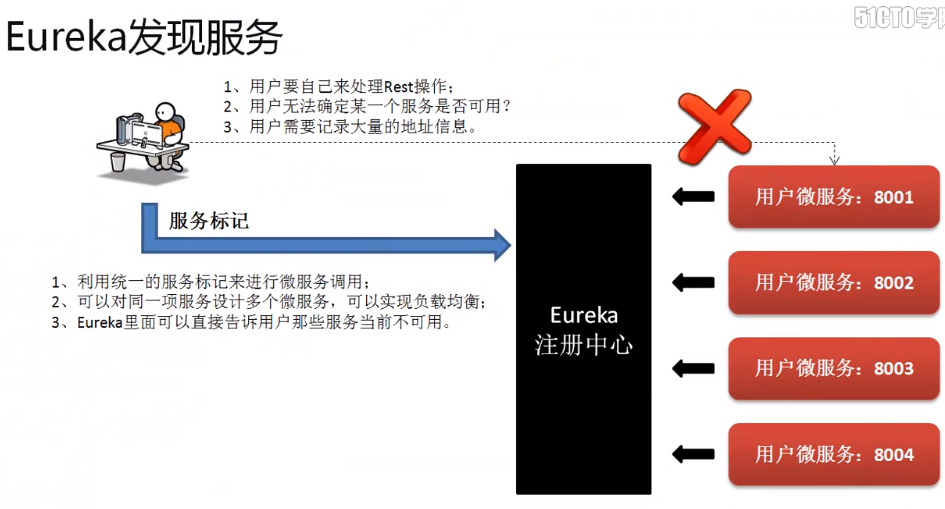
<**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-security</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>

5、【microcloud-provider-dept-8001】注释application.yml安全（session）配置项：

## Eureka服务注册与发现框架

对于服务发现框架可以简单的理解为服务的注册以及使用操作步奏，比如：在Zookeeper的使用流程里面，已经明确的描述了一个服务的注册以及发现操作，在整个Rest架构里面，会存在有大量额微服务的信息。

在SpringCloud之中使用了大量的Netflix的开源项目，而其中Eureka就属于Netflix提供的发现微服务组件，所有的微服务使用之中全部向Eureka之中进行注册，而后客户端直接了利用Eureka进行服务信息的获得。



Eureka的作用实际上和Zookeeper是非常类似的，但是Springcloud虽然支持有Zookeeper，不过从官方推荐使用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。

### Eureka服务配置

1. 创建项目模块microcloud-eureka-7001
2. 【microcloud-eureka-7001】修改pom.xml文件，追加eureka支持：

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-eureka-7001</**artifactId**>  
  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-eureka-server</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-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**>  
</**project**>

1. 【microcloud-eureka-7001】修改application.yml配置文件，主要进行eureka定义。

**server:  
 port:** 7001  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7001.com

1. 【microcloud-eureka-7001】修改本机hosts文件，追加项目映射地址：

127.0.0.1 eureka-7001.com

1. 【microcloud-eureka-7001】修改Eureka程序启动类，追加有Eureka服务说明：

@SpringBootApplication  
@EnableEurekaServer  
**public class** Eureka\_7001\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Eureka\_7001\_StartSpringCloudApplication.**class**);  
 }  
}

1. 【microcloud-eureka-7001】启动服务，访问：<http://eureka-7001.com:7001/>

### Eureka服务注册

将所有的微服务信息注册到eureka服务中，这样就可以被客户端执行并且调用了。

1. 【microcloud-provider-dept-8001】修改pom文件，追加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，定义进行注册的eureka服务地址，这个地址就是eureka客户端配置。

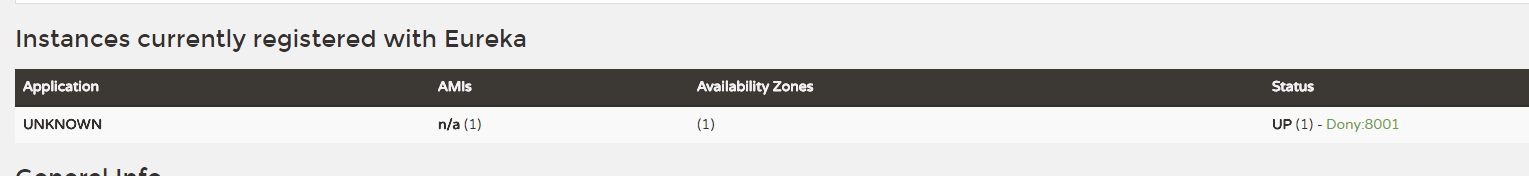
**eureka:** *# 客户端进行Eureka注册的配置* **client:  
 service-url:  
 defaultZone:** http://eureka-7001.com:7001/eureka

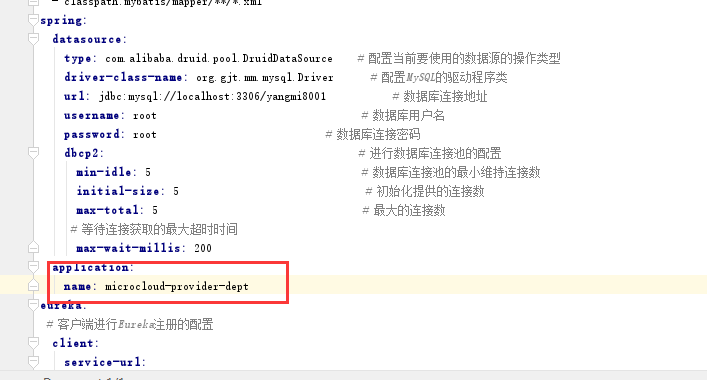
1. 【microcloud-provider-dept-8001】修改项目主类，追加有eureka客户端的启用注解：

@SpringBootApplication  
@EnableEurekaClient  
**public class** Dept\_8001\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Dept\_8001\_StartSpringCloudApplication.**class**, args);  
 }  
}

此时由于存在有这样的“EnableEurekaClient”注解信息，所以当启动服务之后该（项目）服务将自动注册到eureka服务器中。

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





5、启动eureka和微服务，访问：http://eureka-7001.com:7001/

### Eureka微服务信息

1. 【microcloud-provider-dept-8001】修改application.yml文件，追加主机名称的显示：

**eureka:  
 client:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://eureka-7001.com:7001/eureka  
 **instance:  
 instance-id:** dept-8001.com *# 在信息列表时显示主机名称*

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:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://eureka-7001.com:7001/eureka  
 **instance:** *# 在信息列表时显示主机名称* **instance-id:** dept-8001.com  
 *# 访问的路径变为IP地址* **prefer-ip-address:** true

1. 【microcloud-provider-dept-8001】如果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：

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

同时追加一下内容

<**resources**>  
 <**resource**>  
 <**directory**>src/main/resources</**directory**>  
 <**filtering**>true</**filtering**>  
 </**resource**>  
</**resources**>

1. 【microcloud-provider-dept-8001】修改application.yml配置文件，追加info相关信息：

**info:  
 app.name:** yangmi-microcloud  
 **company.name:** www.ym.cn  
 **build.artifactId:** $project.artifactId$  
 **build.version:** $project.version$

1. 【microcloud-eureka-7001】追加一下配置，禁止eureka启动自己注册自己：

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

### Eureka发现服务

在实际的项目开发中需要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的问题：

1. 新服务追加的时候应该立刻可以进行注册；
2. 当某一服务下线之后应该可以进行清理；
3. 【microcloud-eureka-7001】设置服务的清理间隔，修改application.yml文件：

**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7001.com  
 **server:** *# 设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）* **eviction-interval-timer-in-ms:** 60000

一般情况下该配置不建议进行修改，使用默认的60秒

1. 【microcloud-eureka-7001】在Eureka里面有个问题，它默认支持有保护模式的概念，所谓的保护模式是指即便现在某一个微服务不可用了，eureka不会清理，依然会进行该微服务信息的保存。如果要修改保护模式，则可以修改application.yml配置文件：

**server:  
 port:** 7001  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7001.com  
 **server:** *# 设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）  
 # eviction-interval-timer-in-ms: 60000  
 # 设置为false表示关闭保护模式  
 #enable-self-preservation: true* **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** 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，但是很多时候Eureka里面也会自带有清除过程，一般情况下不建议修改。

1. 【microcloud-provider-dept-8001】微服务客户端之所以与Eureka之间保持联系，依靠的是心跳机制，也就是说客户端可以自己来进行心跳的配置，修改application.yml配置文件：

由于所有的服务都注册到了Eureka之中，这样如果配置了”lease-expiration-duration-in-seconds: 90”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，则认为该微服务已经宕机了。

**eureka:  
 client:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://eureka-7001.com:7001/eureka  
 **instance:** *# 在信息列表时显示主机名称* **instance-id:** dept-8001.com  
 *# 访问的路径变为IP地址* **prefer-ip-address:** true  
 *#设置心跳的时间间隔（默认是30秒）  
 #lease-renewal-interval-in-seconds: 30  
 #如果现在超过了90秒的间隔（默认是90秒）,则认为挂了  
 #lease-expiration-duration-in-seconds: 90*

1. 【microcloud-provider-dept-8001】对于注册到Eureka上的微服务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：

*// 进行Eureka的发现服务*@Resource  
**private** DiscoveryClient **discoveryClient**;  
  
@RequestMapping(**"/dept/discover"**)  
**public** Object discover() {  
 *// 直接返回发现服务信息* **return this**.**deptService**;  
}

1. 【microcloud-provider-dept-8001】在主程序之中启用Eureka发现服务：

@SpringBootApplication  
@EnableEurekaClient  
**@EnableDiscoveryClient**  
**public class** Dept\_8001\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Dept\_8001\_StartSpringCloudApplication.**class**, args);  
 }  
}

6、访问：http://dept-8001.com:8001/dept/discover

### Eureka安全配置

1. 【microcloud-eureka-7001】修改pom.xml，引入SpringSecurity的依赖包：

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

2、【microcloud-eureka-7001】一旦项目中导入了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，而这个密码显然是动态改变的，所以要修改application.yml配置文件，追加密码配置项：

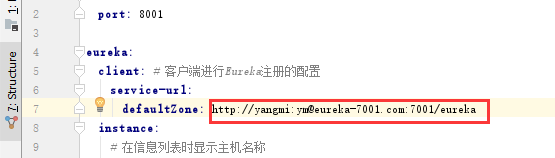
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym

此时再次访问：http://eureka-7001.com:7001，就需要密码。

如果不想每次手动输入则可以使用：

http://yangmi:ym@eureka-7001.com:7001方式访问，当然微服务客户端也要以这种的形式访问。

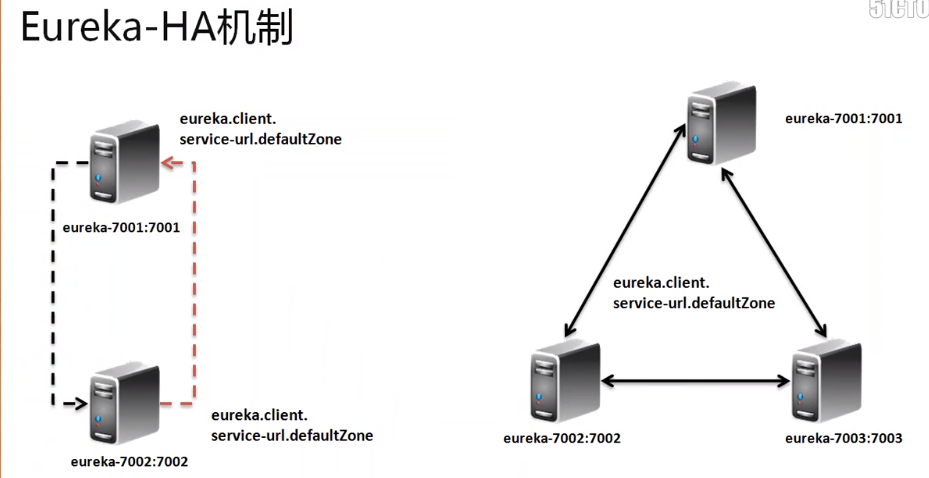
3、【microcloud-provider-dept-8001】修改application.yml配置文件，进行授权的注册链接：



4、【microcloud-provider-dept-8001】启动项目，看是否能注册到eureka中。

### Eureka-HA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，基本流程如下：



1、修改本机hosts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】为方便进行Eureka操作，将重新复制出两份，分别为“microcloud-eureka-7002”、“microcloud-eureka-7003”。
2. 【microcloud-eureka-7001】修改application.yml文件，注意端口号以及Eureka服务注册位置；

7001：

**server:  
 port:** 7001  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7001.com  
 *#server:  
 # 设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）  
 # eviction-interval-timer-in-ms: 60000  
 # 设置为false表示关闭保护模式  
 #enable-self-preservation: true* **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym

1. 【microcloud-eureka-7002】修改application.yml文件：

**server:  
 port:** 7002  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7002.com  
 **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka   
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym

1. 【microcloud-eureka-7003】修改application.yml文件：

**server:  
 port:** 7003  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7003.com  
 *#server:  
 # 设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）  
 # eviction-interval-timer-in-ms: 60000  
 # 设置为false表示关闭保护模式  
 #enable-self-preservation: true* **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka  
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym

1. 启动所有的eureka服务，而后观察每一个服务的后台运行的副本效果：

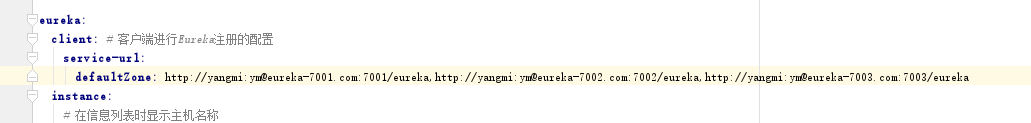
依次登录：

[http://eureka-7003.com:7003/](http://eureka-7003.com:7003/eureka)

[http://eureka-7002.com:7002/](http://eureka-7002.com:7002/eureka)

[http://eureka-7001.com:7001/](http://eureka-7001.com:7001/eureka)

1. 【microcloud-provider-dept-8001】修改application.yml进行索泰主机的注册与重启：



### Eureka打包部署

打包处理需要考虑到项目的各种情况：开发（dev）、测试（beta）、生产（product）。本次的打包处理直接基于yml配置文件完成，对于properties配置与springboot微服务发布处理过程一样，将microcloud-eureka-7001模块复制为“microcloud-eureka-server”。

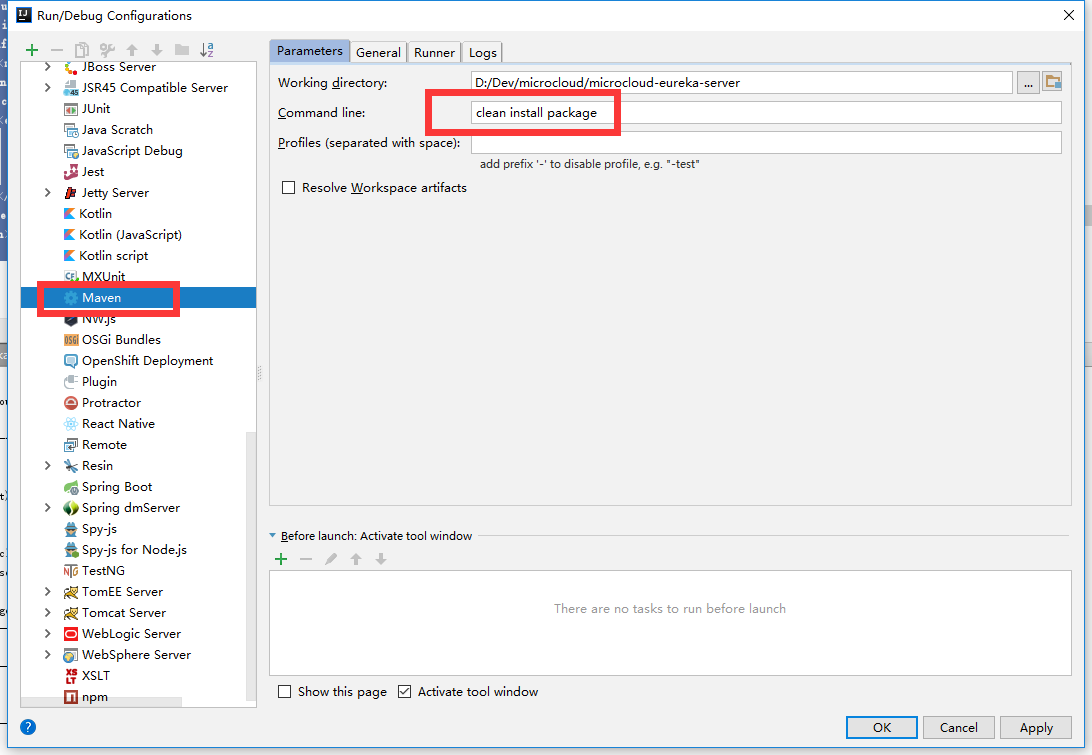
1. 【microcloud-eureka-server】修改application.yml配置文件：

**spring:  
 profiles:  
 active:** - dev-7001  
---  
**spring:  
 profiles:** dev-7001  
 **application:  
 name:** microcloud-eureka-7001  
**server:  
 port:** 7001  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7001.com  
 *#server:  
 # 设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）  
 # eviction-interval-timer-in-ms: 60000  
 # 设置为false表示关闭保护模式  
 #enable-self-preservation: true* **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym  
---  
**spring:  
 profiles:** dev-7002  
 **application:  
 name:** microcloud-eureka-7002  
**server:  
 port:** 7002  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7002.com  
 **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym  
---  
**spring:  
 profiles:** dev-7003  
 **application:  
 name:** microcloud-eureka-7003  
**server:  
 port:** 7003  
**eureka:***# eureak实例定义* **instance:** *#定义Eureka实例所在的主机名称,也可以直接使用IP* **hostname:** eureka-7003.com  
 *#server:  
 # 设置清理的间隔时间，而后这个时间使用的是毫秒单位（默认是60秒）  
 # eviction-interval-timer-in-ms: 60000  
 # 设置为false表示关闭保护模式  
 #enable-self-preservation: true* **client:** *# 当前的微服务不注册到eureka之中(自己就是发现服务)* **register-with-eureka:** false  
 *# 不通过eureka获取注册信息* **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka  
**security:  
 basic:** *# 启用安全认证处理* **enabled:** true  
 **user:** *#用户名* **name:** yangmi  
 *#密码* **password:** ym

1. 【microcloud-eureka-server】添加一个打包的插件：

<**build**>  
 <**finalName**>eureka-server</**finalName**>  
 <**plugins**>  
 <**plugin**> *<!-- 该插件的主要功能是进行项目的打包发布处理 -->* <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 <**configuration**> *<!-- 设置程序执行的主类 -->* <**mainClass**>cn.ym.com.StartSpringCloudApplication</**mainClass**>  
 </**configuration**>  
 <**executions**>  
 <**execution**>  
 <**goals**>  
 <**goal**>repackage</**goal**>  
 </**goals**>  
 </**execution**>  
 </**executions**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

1. 运行maven clean install package(也可以在命令行直接运行mvn clean install package),到项目下找到jar;



1. 执行jar此时运行在7001端口；

java -jar eureka-server.jar

1. 运行其他的两个profile配置：
2. 运行“dev-7002” profile

java -jar eureka-server.jar --spring.profiles.active=dev-7002

或者java -jar -Dspring.profiles.active=dev-7002 eureka-server.jar

1. 运行“dev-7003” profile

java -jar eureka-server.jar --spring.profiles.active=dev-7003

或者java -jar -Dspring.profiles.active=dev-7003 eureka-server.jar

6、【microcloud-provider-dept-8001】启动微服务，当然也可以用打包的方式启动

## Ribbion负载均衡

使用eureka的服务注册的目的是希望所有的服务统一归属到进行处理，但是问题是，所有微服务汇集到了Eureka中，客户端调用也应该通过eureka完成，而这种调用就可以使用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。

### Ribbon基础配置

1. 【microcloud-consumer-80】修改pom.xml文件，追加Eureka和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对象的时候假如Ribbon的配置。

@Configuration  
**public class** RestConfig {  
  
 */\*\*  
 \* 配置httpheader头信息  
 \*  
 \** ***@return*** *\*/* @Bean  
 **public** HttpHeaders getHeaders() {  
 *//定义httpheader头信息* HttpHeaders httpHeaders = **new** HttpHeaders();  
 String auth = **"yangmi:ym"**;  
 *//加密的处理* **byte**[] encodeAuth = 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(encodeAuth);  
 httpHeaders.set(**"Authorization"**, authHeader);  
 **return** httpHeaders;  
 }  
  
 @Bean  
 @LoadBalanced  
 **public** RestTemplate getRestTemplate() {  
 **return new** RestTemplate();  
 }  
}

1. 【microcloud-consumer-80】在application.xml中追加Eureka的服务地址配置：

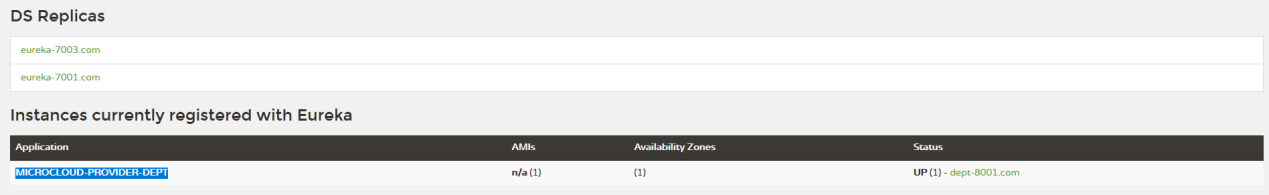
**server:  
 port:** 80  
**eureka:  
 client:  
 service-url:   
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 *#客户端不作为微服务注册到Eureka中* **register-with-eureka:** false

1. 【microcloud-consumer-80】修改启动类，追加Eureka客户端配置注解：

@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：

在Eureka里面注册的所有服务名称都是大写字母：**MICROCLOUD-PROVIDER-DEPT**

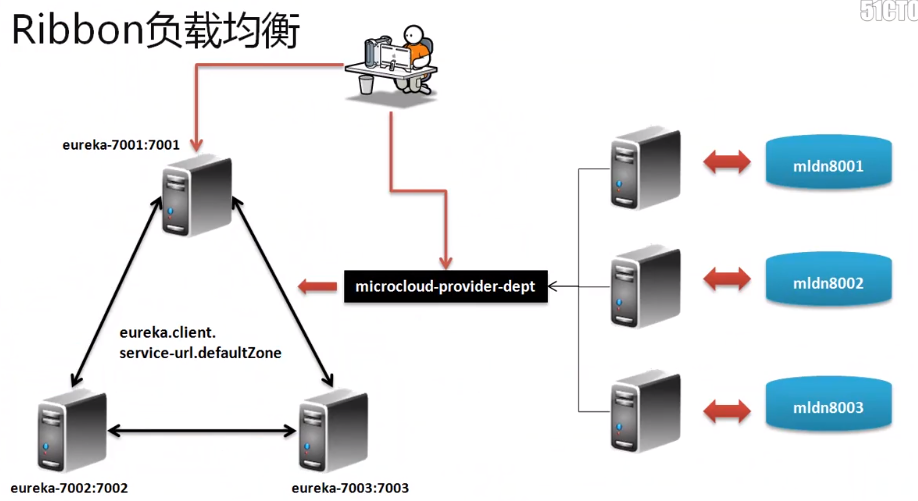


@RestController  
**public class** ConsumerDeptController {  
 **public static final** String ***DEPT\_GET\_URL*** = **"http://MICROCLOUD-PROVIDER-DEPT/dept/get/"**;  
 **public static final** String ***DEPT\_LIST\_URL*** = **"http://MICROCLOUD-PROVIDER-DEPT/dept/list/"**;  
 **public static final** String ***DEPT\_ADD\_URL*** = **"http://MICROCLOUD-PROVIDER-DEPT/dept/add"**;  
  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @Resource  
 **private** HttpHeaders **httpHeaders**;  
  
 @RequestMapping(value = **"/consumer/dept/get"**)  
 **public** Object getDept(**long** id) {  
 **return this**.**restTemplate**.exchange(***DEPT\_GET\_URL*** + id, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), Dept.**class**);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/list"**)  
 **public** Object listDept() {  
 **return this**.**restTemplate**.exchange(***DEPT\_LIST\_URL***, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), List.**class**);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/add"**)  
 **public** Object addDept(Dept dept) {  
 **return this**.**restTemplate**.exchange(***DEPT\_ADD\_URL***, HttpMethod.***POST***, **new** HttpEntity<Object>(dept, **this**.**httpHeaders**), Boolean.**class**);  
 }  
}

1. 【microcloud-consumer-80】启动项目，访问：http://client.com/consumer/dept/get?id=1

有了Eureka和Ribbon整合后，用户不再关注Rest的服务地址与端口，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，这就意味着现在就可以进行负载均衡的处理了。

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，受修改各自的数据库连接配置：

**8001：**

DROP DATABASE IF EXISTS yangmi8001 ;

CREATE DATABASE yangmi8001 CHARACTER SET UTF8 ;

USE yangmi8001 ;

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()) ;

**8002：**

DROP DATABASE IF EXISTS yangmi8002 ;

CREATE DATABASE yangmi8002 CHARACTER SET UTF8 ;

USE yangmi8002 ;

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()) ;

**8003：**

DROP DATABASE IF EXISTS yangmi8003 ;

CREATE DATABASE yangmi8003 CHARACTER SET UTF8 ;

USE yangmi8003 ;

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()) ;

1. 【microcloud-provider-dept-\*】修改各自服务的application.yml配置文件（服务端口和数据库端口）；



千万注意，所有服务的名字一定要保持一致，发走会认为是不同的服务，无法进行负载均衡。

1. 修改本机的hosts配置文件，追加新的域名：

127.0.0.1 dept-8001.com

127.0.0.1 dept-8002.com

127.0.0.1 dept-8003.com

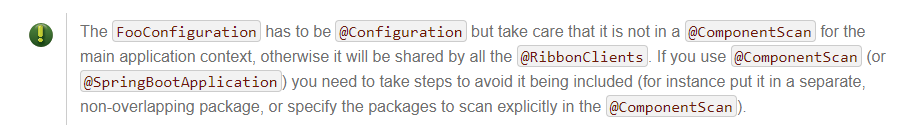
1. 【microcloud-provider-dept-\*】在保证Eureka已经准确启动之后启动所有的部门微服务信息。
2. 【microcloud-consumer-80】启动消费端，消费端在RestTemplate配置的时候使用了负载均衡的注解，观察负载均衡（每次返回对于服务器的数据）。

以上为了便于观察使用了不同数据库，实际中数据库一定是相同的。

### 自定义Ribbon配置

此前使用的“@LoadBalanced”注解，讲解的是一个负载均衡，但是对于负载均衡也是可以修改策略的，可以使用自定义的LoadBalance配置类。

1. 【microcloud-consumer-80】追加一个LoadBalance配置类，这个类应该复杂爱SpringCloud启动后找不到的位置上（即不能放在启动类所在包及子包下）：



**package** cn.ym.commons.config;  
  
**import** com.netflix.loadbalancer.IRule;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \* User: Dony  
 \* Date: 2018/7/3  
 \* Time: 11:46  
 \* Description:  
 \*/***public class** MyLoadBalanceConfig {  
 **public** IRule ribbonRule() {  
 *// 其中IRule就是所有规则的标准* **return new** com.netflix.loadbalancer.RandomRule();  
 }  
}

在com.netflix.loadbalancer下就是Ribbon的所有负责均衡策略。

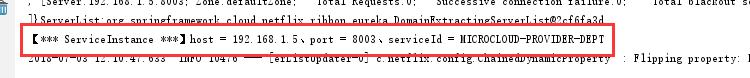
1. 【microcloud-consumer-80】修改程序主类，追加Ribbon的配置：

@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);  
 }  
}

1. 【microcloud-consumer-80】修改控制器程序类，在整个程序类中可以通过负载均衡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());  
 **return this**.**restTemplate**.exchange(***DEPT\_GET\_URL*** + id, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), Dept.**class**);  
}

1. 访问：http://client.com/consumer/dept/get?id=1，观察输出信息；



### 脱离Eureka使用Ribbon

在之前使用的技术都是通过Ribbon访问了Eureka中指定名称的服务，但是在Ribbon设计的时候考虑到了一个脱离Eureka使用的环境。复制模块microcloud\microcloud-consumer-80为“microcloud-consumer-ribbon”

1. 【microcloud-consumer-ribbin】修改application.yml配置文件：

**server:  
 port:** 80  
  
**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-ribbin】修改RestConfig配置程序类，不再需要“@LoadBalanced”注解：

@Bean  
*// @LoadBalanced* **public** RestTemplate getRestTemplate() {  
 **return new** RestTemplate();  
 }

1. 【microcloud-consumer-ribbin】将启动类中Ribbon配置,转移到ConsumerDeptController，同时获取微服务服务地址，同时去掉有关eureka的配置：

@RestController  
@RibbonClient(name = **"microcloud-provider-dept"**, configuration = MyLoadBalanceConfig.**class**)  
**public class** ConsumerDeptController {  
 **public static final** String ***DEPT\_REST\_TOPIC*** = **"microcloud-provider-dept"**;  
  
 @Resource  
 **private** RestTemplate **restTemplate**;  
  
 @Resource  
 **private** HttpHeaders **httpHeaders**;  
  
 @Resource  
 **private** LoadBalancerClient **loadBalancerClient**;  
  
 @RequestMapping(value = **"/consumer/dept/get"**)  
 **public** Object getDept(**long** id) {  
 ServiceInstance serviceInstance = **this**.**loadBalancerClient**.choose(***DEPT\_REST\_TOPIC***);  
 System.***out***.println(  
 **"【\*\*\* ServiceInstance \*\*\*】host = "** + serviceInstance.getHost()  
 + **"、port = "** + serviceInstance.getPort()  
 + **"、serviceId = "** + serviceInstance.getServiceId() + **"、uri="** + serviceInstance.getUri());  
 URI deptUri = URI.*create*(String.*format*(**"http://%s:%s/dept/get/"** + id, serviceInstance.getHost(), serviceInstance.getPort()));  
 **return this**.**restTemplate**.exchange(deptUri, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), Dept.**class**);  
 }  
}

1. 【microcloud-consumer-ribbin】启动服务访问：http://client.com/consumer/dept/get?id=1

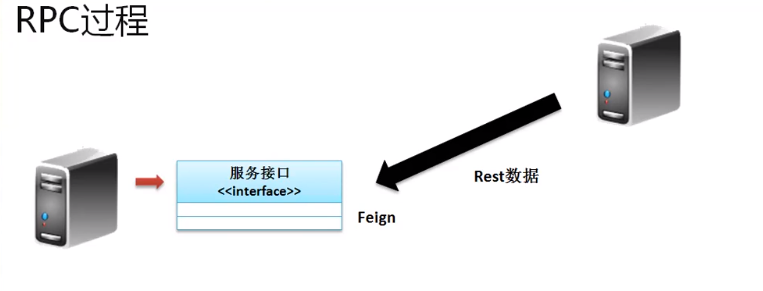
这种模式并不标准，只能够说是Ribbon自己所具备的一项功能而已，实际中如果不是非常必要不建议使用。

## Feign接口转换

现在为止所进行的所有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：

@RequestMapping(value = **"/consumer/dept/list"**)  
**public** Object listDept() {  
 **return this**.**restTemplate**.exchange(***DEPT\_LIST\_URL***, HttpMethod.***GET***, **new** HttpEntity<Object>(**this**.**httpHeaders**), List.**class**);  
}

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，习惯的编程模式是通过接口实现业务的操作，而不是通过具体的Rest数据（json）。



### Feign基础配置

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（feign开发包中自带了ribbon），并且引用microclou-consumer-api：

<**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-feign</**artifactId**>  
</**dependency**>

1. 【microcloud-consumer-service】建立一个新的模块，这个模块专门负责客户端接口的定义；

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-service</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-feign</**artifactId**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

1. 【microcloud-consumer-service】修改pom.xml文件，引用“microcloud--api”模块

<**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>

1. 【microcloud-consumer-feign】此时如果要通过Feign进行远程Rest调用，则必须要考虑服务的认证问题，在microcloud-consumer-service中处理认证。

@Configuration  
**public class** FeignClientConfig {  
 @Bean  
 **public** BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  
 *//feign 远程rest认证* **return new** BasicAuthRequestInterceptor(**"yangmi"**, **"ym"**);  
 }  
}

1. 【microcloud-consumer-service】创建一个IDeptClientService接口：

@FeignClient(value = **"MICROCLOUD-PROVIDER-DEPT"**, configuration = FeignClientConfig.**class**)  
**public interface** IDeptClientService {  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/dept/get/{id}"**)  
 Dept get(@PathVariable(**"id"**) **long** id);  
  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/dept/list"**)  
 List<Dept> list();  
  
 @RequestMapping(method = RequestMethod.***POST***, value = **"/dept/addo"**)  
 **boolean** add(Dept dept);  
}

1. 【microcloud-consumer-feign】修改pom.xml配置文件，引入microclou-service开发包：

<**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-service</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>

1. 【microcloud-consumer-feign】修改ConsumerDeptController控制器程序类：

@RestController  
**public class** ConsumerDeptController {  
  
 @Resource  
 **private** IDeptClientService **deptClientService**;  
  
 @RequestMapping(value = **"/consumer/dept/get"**)  
 **public** Object getDept(**long** id) {  
 **return this**.deptService.get(id);  
 }  
  
 @RequestMapping(value = **"/consumer/dept/list"**)  
 **public** Object listDept() {  
 **return this**.deptService.list();  
 }  
  
 @RequestMapping(value = **"/consumer/dept/add"**)  
 **public** Object addDept(Dept dept) **throws** Exception {  
 **return this**.deptService.add(dept);  
 }  
}

1. 【microcloud-consumer-feign】修改程序启动主类，追加操作处理。

@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients(basePackages = **"cn.ym.com.service"**)  
**public class** Consumer\_Feign\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Consumer\_Feign\_StartSpringCloudApplication.**class**,  
 args);  
 }  
}

1. 【microcloud-consumer-feign】启动服务，测试访问：

<http://client.com/consumer/dept/get?id=1>

<http://client.com/consumer/dept/list>

可以发现Feign在处理的时候自带负载均衡的配置项；

### Feign相关配置

1. 【microcloud-consumer-feign】Feign中最为核心的作用就是将Rest服务的信息转换为接口，但是实际使用中要考虑到一些情况，例如：数据压缩，Rest的核心本质在于：JSON数据传输（XML、文本），于是就必须考虑一种情况，万一用户发送的数据很大呢？这个时候可以考虑修改application.yml配置文件对数据进行压缩；

**feign:  
 compression:  
 request:** *# 可以被压缩的类型* **mime-types:** - text/xml  
 - application/xml  
 - application/json  
 *# 超过2048的字节进行压缩* **min-request-size:** 2048

1. 如果需要则可以开启feign的相关日志信息（默认不开启），修改application.yml配置文件：
2. 【microcloud-consumer-feign】追加日志跟踪：

**logging:  
 level:** cn.ym.com.service:DEBUG

1. 【microcloud-service】修改FeignClientConfig,开启日志输出：

@Configuration  
**public class** FeignClientConfig {  
 @Bean  
 **public** Logger.Level getFeignLoggerLevel(){  
 **return** feign.Logger.Level.***FULL***;   
 }  
  
 @Bean  
 **public** BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  
 *//feign 远程rest认证* **return new** BasicAuthRequestInterceptor(**"yangmi"**, **"ym"**);  
 }  
}

1. 【microcloud-consumer-feign】启动服务，访问：

观察日志可以得到如下流程：

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：

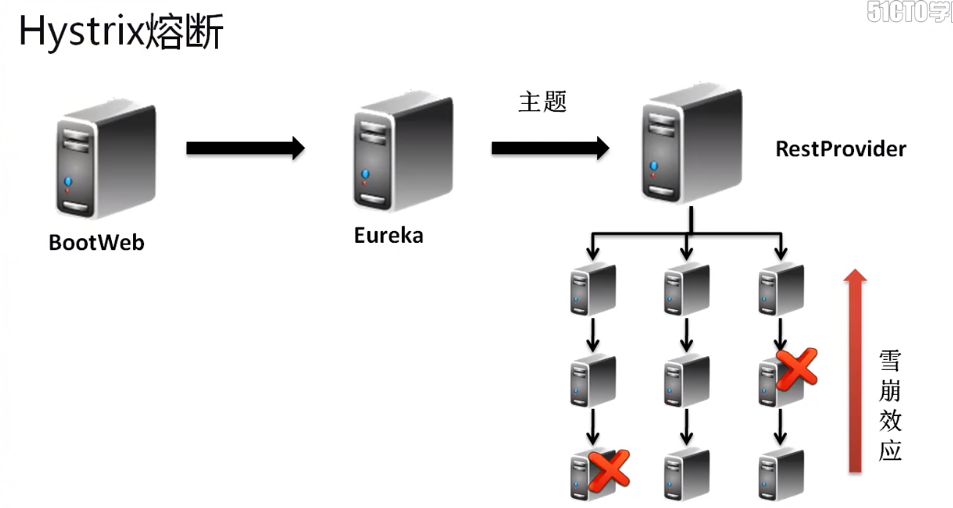
1. 随后由于配置授权处理，所以继续发送授权信息（“Authorization”）
2. 在进行服务调用的时候Feign整合了ribbon技术，所以也支持有负载均衡处理。

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

## Hystrix熔断机制

### Hystrix简介

当服务宕机时，为了保证不出现这样的错误提示信息页面，应该做一个失败的错误处理，而这种处理就称为熔断机制，当然在在实际中更为复杂的springcloud调用里面这个熔断机制实际上更有用处。



对于熔断技术的实现需要考虑以下几种情况：

1. 出现错误以后可以fallback错误的处理信息；
2. 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

### 服务端失败回调（FallBack）

1、【microcloud-provider-dept-hystrix-8001】修改pom.xml，追加Hystrix依赖：

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

2、【microcloud-provider-dept-hystrix-8001】修改控制器类：

@RestController  
**public class** DeptController {  
 @Resource  
 **private** IDeptService **deptService**;  
  
 @RequestMapping(value = **"/dept/get/{id}"**, method = RequestMethod.***GET***)  
 @HystrixCommand(fallbackMethod = **"getFallback"**) *// 如果当前调用的get()方法出现了错误，则执行fallback* **public** Object get(@PathVariable(**"id"**) **long** id) {  
 Dept vo = **this**.**deptService**.get(id);*// 接收数据库的查询结果* **if** (vo == **null**) {*// 数据不存在，假设让它抛出个错误* **throw new** RuntimeException(**"部门信息不存在！"**);  
 }  
 **return** vo;  
 }  
  
 */\*\*  
 \* get方法的fallback  
 \*  
 \** ***@param id*** *\** ***@return*** *\*/* **public** Object getFallback(@PathVariable(**"id"**) **long** id) {*// 此时方法的参数 与get()一致* Dept vo = **new** Dept();  
 vo.setDeptno(999999L);  
 vo.setDname(**"【ERROR】系统繁忙，请稍后再试！"**); *// 错误的提示* vo.setLoc(**"DEPT-Provider"**);  
 **return** vo;  
 }  
}

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”注解中配置好的fallbackMethod调用类中指定方法，返回数据。

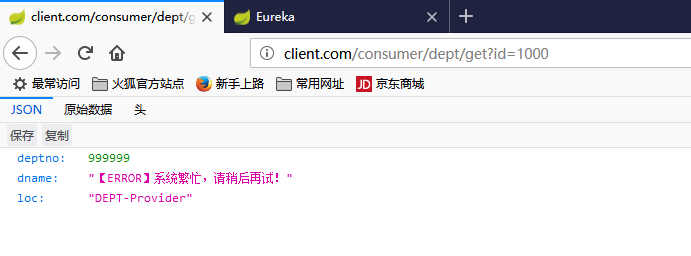
1. 【microcloud-provider-dept-hystrix-8001】在主类中启用熔断处理：

@SpringBootApplication  
@EnableEurekaClient  
@EnableCircuitBreaker  
**public class** Dept\_hysrtix\_8001\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Dept\_hysrtix\_8001\_StartSpringCloudApplication.**class**, args);  
 }  
}

1. 【microcloud-provider-dept-hystrix-8001】启动服务，测试：

<http://client.com/consumer/dept/get?id=1>

<http://client.com/consumer/dept/get?id=1000>



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），此时会调用指定方法的fallback处理。

### 3、服务降级

所有的RPC子系统里面服务的降级是一个最为重要的话题，所谓的降级指的是当服务的提供方不可使用的时候，程序不会出现异常而是出现本地的操作调用，降级的本质是服务暂停，资源调度给其他一些重要的业务。

服务的降级处理时在客户端实现的，与你的服务器端没有关系。

1. 【microcloud-service】扩充一个IDeptService的失败调用（严格来讲是服务降级）处理：

@Component  
**public class** IDeptClientServiceFallbackFactory **implements** FallbackFactory<IDeptClientService> {  
  
 @Override  
 **public** IDeptClientService create(Throwable throwable) {  
 *//匿名内部* **return new** IDeptClientService() {  
 @Override  
 **public** Dept get(**long** id) {  
 Dept vo = **new** Dept();  
 vo.setDeptno(888888L);  
 vo.setDname(**"【ERROR】Feign-Hystrix"**); *// 错误的提示* vo.setLoc(**"Consumer客户端提供"**);  
 **return** vo;  
 }  
  
 @Override  
 **public** List<Dept> list() {  
 **return null**;  
 }  
  
 @Override  
 **public boolean** add(Dept dept) {  
 **return false**;  
 }  
 };  
 }  
}

1. 【microcloud-service】修改IDeptClientService接口，追加本地的FallBack配置。

@FeignClient(value = **"MICROCLOUD-PROVIDER-DEPT"**, configuration = FeignClientConfig.**class**,fallbackFactory = IDeptClientServiceFallbackFactory.**class**)  
**public interface** IDeptClientService {  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/dept/get/{id}"**)  
 Dept get(@PathVariable(**"id"**) **long** id);  
  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/dept/list"**)  
 List<Dept> list();  
  
 @RequestMapping(method = RequestMethod.***POST***, value = **"/dept/addo"**)  
 **boolean** add(Dept dept);  
}

此时当服务不可用的时候会执行“IDeptClientServiceFallbackFactory”类中返回的IDeptClientService接口的匿名对象信息。

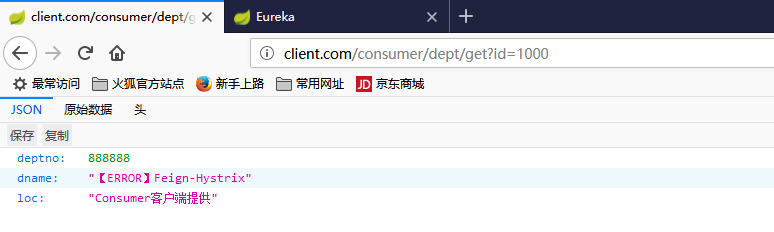
1. 复制microcloud-consumer-feign模块为“microcloud-consumer-hystrix”

**server:  
 port:** 80  
**eureka:  
 client:  
 service-url:   
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 *#客户端不作为微服务注册到Eureka中* **register-with-eureka:** false  
**feign:  
 hystrix:   
 enabled:** true   
 **compression:  
 request:** *# 可以被压缩的类型* **mime-types:** - text/xml  
 - application/xml  
 - application/json  
 *# 超过2048的字节进行压缩* **min-request-size:** 2048  
**logging:  
 level:  
 cn.ym.com.service:** DEBUG

4、【microcloud-consumer-hystrix】修改application.xml配置文件，追加feign的配置启用。

**server:  
 port:** 80  
**eureka:  
 client:  
 service-url:   
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 *#客户端不作为微服务注册到Eureka中* **register-with-eureka:** false  
**feign:  
 hystrix:   
 enabled: true**   
 **compression:  
 request:** *# 可以被压缩的类型* **mime-types:** - text/xml  
 - application/xml  
 - application/json  
 *# 超过2048的字节进行压缩* **min-request-size:** 2048  
**logging:  
 level:  
 cn.ym.com.service:** DEBUG

5、【microcloud-consumer-hystrix】修改程序启动主类，启动服务测试：



当service与启动类不再统一包是需要追加上

@ComponentScan("cn.mldn.service,cn.mldn.microcloud")

@EnableFeignClients(basePackages={"cn.mldn.service"})

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信息。

### HystrixDashBoard监控服务

在Hystrix里面提供有一种监控功能，就是“HsytrixDashboard”，可以利用它来进行整体微服务的监控操作。

1. 新建模块【microcloud-consumer-hystrix-dashboard】;
2. 【microcloud-consumer-hystrix-dashboard】修改pom.xml配置文件：

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-consumer-hystrix-dashboard</**artifactId**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-hystrix</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-hystrix-dashboard</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**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**>  
</**project**>

1. 【microcloud-provider-\*】所有的服务提供者之中都一定要提供有监控服务支持包：

<**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，主要进行端口配置：

**server:  
 port:** 9001

1. 【microcloud-consumer-hystrix-dashboard】创建一个监控主类：

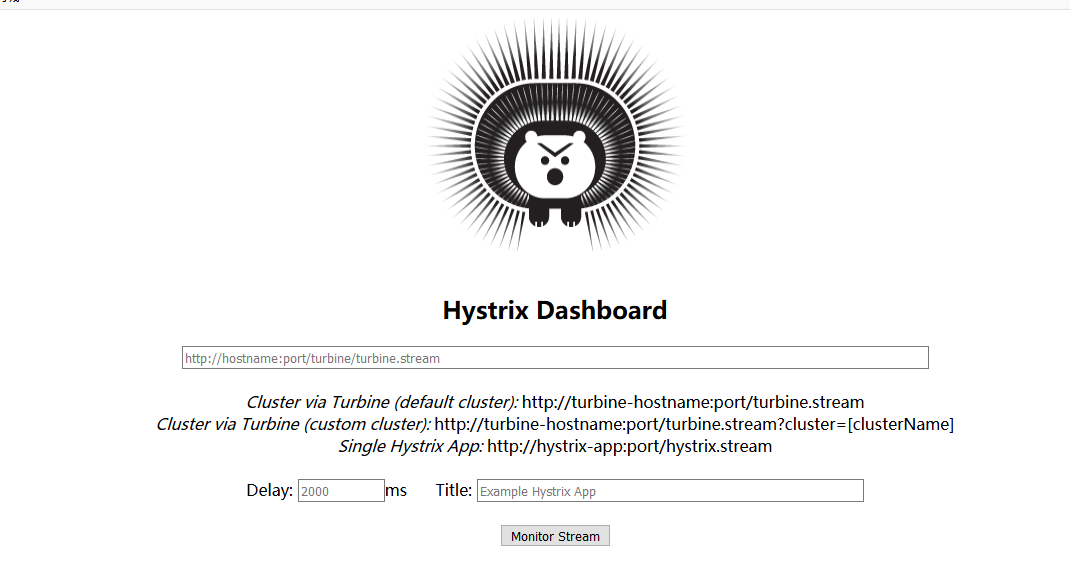
@SpringBootApplication  
@EnableHystrixDashboard  
**public class** HystrixDashboardApplication\_9001 {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(HystrixDashboardApplication\_9001.**class**);  
 }  
}

1. 修改本机hosts文件，增加主机列表：

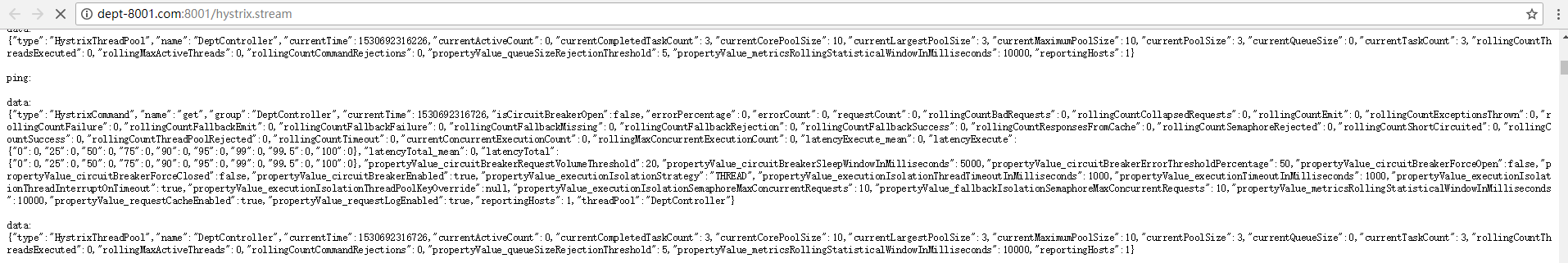
127.0.0.1 hystrix-dashbord-9001.com

1. 启动eureka服务、(hystrix)微服务、(hystrix)消费端微服务、HystrixDashBoard服务，访问：

<http://hystrix-dashbord-9001.com:9001/hystrix.stream>



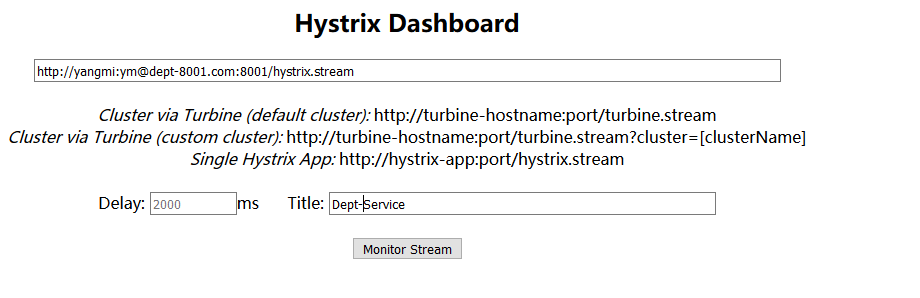
访问：<http://dept-8001.com:8001/hystrix.stream>

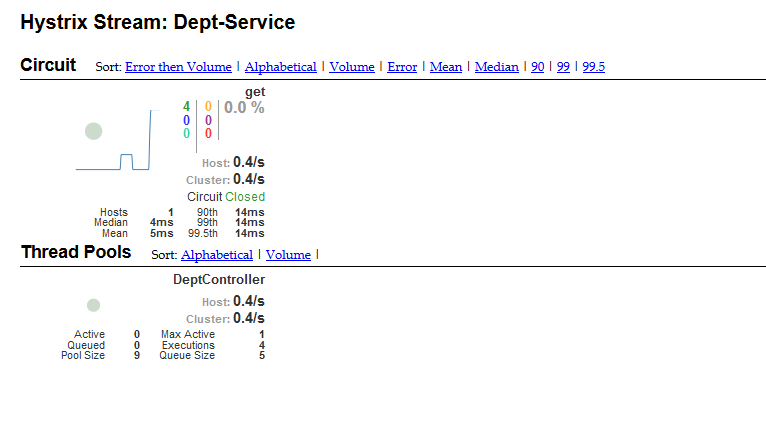


这显然看不明白。

1. 将之前的监控填写到之前启动好的dashboard程序页面中，点击监控，然后访问：

http://client.com/consumer/dept/get?id=1，观察监控页面：





### 5、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，但是如果说有许多的微服务需要监控那么这种情况下就可以利用Turbine技术来实现。

1. 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的数据库连接，只是做一个数据信息。 复制模块microcloud-provider-hystrix-8001为microcloud-provider-company-8101。
2. 【microcloud-provider-company-8101】修改pom.xml，添加依赖包；

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-provider-company-8101</**artifactId**>  
  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**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**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**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**>  
</**project**>

1. 【microcloud-api】追加一个新的VO类：company

**public class** Company **implements** Serializable {  
 **private** String **title**;  
 **private** String **note**;  
  
 **public** String getTitle() {  
 **return title**;  
 }  
  
 **public void** setTitle(String title) {  
 **this**.**title** = title;  
 }  
  
 **public** String getNote() {  
 **return note**;  
 }  
  
 **public void** setNote(String note) {  
 **this**.**note** = note;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Company [title="** + **title** + **", note="** + **note** + **"]"**;  
 }  
}

1. 【microcloud-provider-company-8101】新建一个程序类MemberController：

@RestController  
**public class** MemberController {  
 @RequestMapping(value = **"/company/get/{title}"**, method = RequestMethod.***GET***)  
 @HystrixCommand *// 如果需要进行性能监控，那么必须要有此注解* **public** Object get(@PathVariable(**"title"**) String title){  
 Company vo = **new** Company() ;  
 vo.setTitle(title);  
 vo.setNote(**"www.mldn.cn"**);  
 **return** vo ;  
 }  
}

1. 【microcloud-provider-company-8101】修改程序启动主类：

@SpringBootApplication  
@EnableEurekaClient  
@EnableCircuitBreaker  
**public class** Dept\_Company\_8101\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Dept\_Company\_8101\_StartSpringCloudApplication.**class**, args);  
 }  
}

1. 【microcloud-provider-company-8101】修改application.xml配置文件：

**server:  
 port:** 8101  
  
**eureka:  
 client:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 **instance:** *# 在信息列表时显示主机名称* **instance-id:** dept-8001.com  
 *# 访问的路径变为IP地址* **prefer-ip-address:** true  
 *#设置心跳的时间间隔（默认是30秒）* **lease-renewal-interval-in-seconds:** 30  
 *#如果现在超过了90秒的间隔（默认是90秒）,则认为挂了* **lease-expiration-duration-in-seconds:** 90  
**info:  
 app.name:** yangmi-microcloud  
 **company.name:** www.ym.cn  
 **build.artifactId:** $project.artifactId$  
 **build.version:** $project.version$  
  
 **application:  
 name:**microcloud-provider-company

同时删除mybatis有关程序及配置。

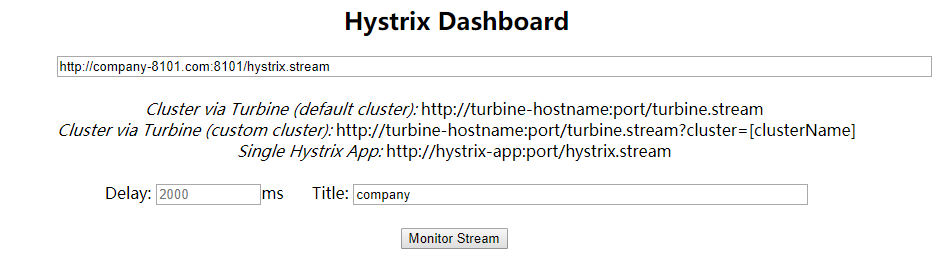
1. 在hosts文件中追加主机映射

127.0.0.1 company-8101.com

1. 【microcloud-provider-company-8101】启动微服务

访问：<http://company-8101.com:8101/company/get/hello>

hystrix监听：



1. 如果想要实现trubine配置，则需要建立一个turbin项目模块，这个项目可以直接复制模块microcloud-consumer-hystrix-dashboard为microcloud-consumer-trubine模块；
2. 【microcloud-consumer-trubine】修改pom.xml，追加turbine依赖包：

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

1. 【microcloud-consumer-trubine】修改application.yml配置文件：

**server:***# turbine的监听端口为9101* **port:** 9101  
**eureka:  
 client:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 **instance:** *# 在信息列表时显示主机名称* **instance-id:** dept-8001.com  
 *# 访问的路径变为IP地址* **prefer-ip-address:** true  
 *#设置心跳的时间间隔（默认是30秒）* **lease-renewal-interval-in-seconds:** 30  
 *#如果现在超过了90秒的间隔（默认是90秒）,则认为挂了* **lease-expiration-duration-in-seconds:** 90  
**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：无安全认证（MICROCLOUD-PROVIDER-COMPANY）、有安全认证（MICROCLOUD-PROVIDER-DEPT）

1. 【microcloud-consumer-trubine】建立一个trubine主类信息：

@SpringBootApplication  
@EnableHystrixDashboard  
@EnableTurbine  
**public class** HystrixTurbineApplication\_9001 {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(HystrixTurbineApplication\_9001.**class**);  
 }  
}

1. 【microcloud-consumer-hystrix-dashboard】运行hystrix-dashboard监控程序；
2. 【microcloud-consumer-trubine】运行trubine集合监控程序；

修改hosts文件，追加一个映射路径：

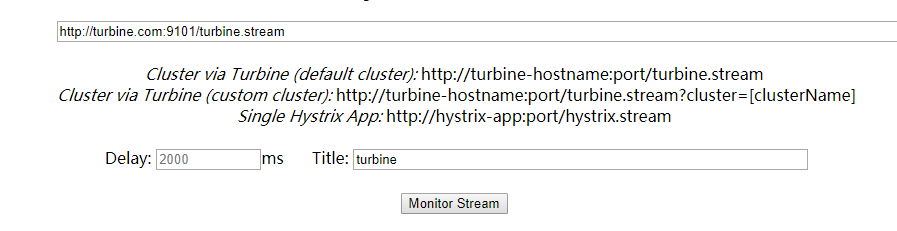
127.0.0.1 turbine.com

访问：

http://turbine.com:9101/turbine.stream

1. 运行HystrixDashboard监控程序：

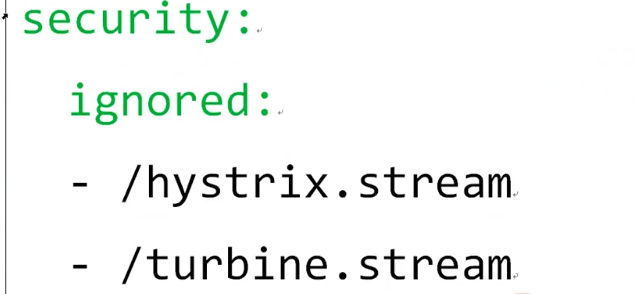
访问：<http://hystrix-dashbord-9001.com:9001/hystrix.stream>



1. 【microcloud-security】如果现在需要trubine进行加密的微服务的访问操作，只需要采取一种折中（不提倡）的方案，就是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两个访问路径，这是基于Bean配置，如果采用application.yml文件的配置，则需要追加一下内容：



## Zuul路由访问

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（路由与过滤）。

### Zuul基础访问

本次将使用Zuul访问无安全的微服务信息；

1. 为突出Zuul的功能，建立一个新的主机映射：



以后微服务的访问不再进行处理了，而是通过Zuul进行跳转后获得。

1. 新建一个模块【microcloud-zuul-gateway-9501】;
2. 【microcloud-zuul-gateway-9501】修改pom.xml文件，追加Zuul相关依赖包：

* 注意：Zuul服务最终还是会注册到Eureka之中

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-zuul-gateway-9501</**artifactId**>  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**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-zuul</**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**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**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**>  
</**project**>

1. 【microcloud-zuul-gateway-9501】修改application.yml:

**server:  
 port:** 9501  
**eureka:  
 client:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 **instance:** *# 在信息列表时显示主机名称* **instance-id:** gateway-9501.com  
 *# 访问的路径变为IP地址* **prefer-ip-address:** true  
 *#设置心跳的时间间隔（默认是30秒）* **lease-renewal-interval-in-seconds:** 30  
 *#如果现在超过了90秒的间隔（默认是90秒）,则认为挂了* **lease-expiration-duration-in-seconds:** 90  
**info:  
 app.name:** yangmi-microcloud  
 **company.name:** www.ym.cn  
 **build.artifactId:** $project.artifactId$  
 **build.version:** $project.version$  
**spring:**  
 **application:  
 name:** microcloud-zuul-gateway

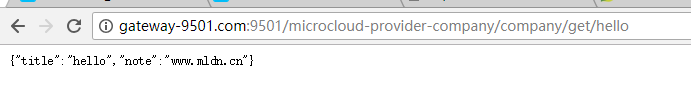
1. 【microcloud-zuul-gateway-9501】创建程序启动类：

@SpringBootApplication  
@EnableZuulProxy  
**public class** Zuul\_9501\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Zuul\_9501\_StartSpringCloudApplication.**class**);  
 }  
}

6、启动Zuul微服务，看zuul是否注册到eureka,访问：

(1)原始路径：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

### Zuul路由访问

整体的Zuul运行之后你会发现，Zuul所实现的就是一个代理功能，现在回出现一个问题，以之前访问的路径为例：

http://gateway-9501.com:9501/microcloud-provider-company/company/get/hello，此时必须知道应用的名称，但是如果不知道找个名称肯定无法访问，可以让调用者知道这个名称，那么使用zuul就没有任何意义了，直接调用即可，而zuul的主要功能是代理，代理的功能就是不让用户看见真实的操作，实际使用中我们就需要为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文件

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

此时就可以用“/company-proxy”来访问“microcloud-provider-company”名称；

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

但是现在存在一个问题，虽然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：

1. 【microcloud-zuul-gateway-9501】修改application.yml文件忽略掉应用名称访问：

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

如果有上百个微服务，以上的配置就比较麻烦了，可以采用通配符“\*”的模式来完成：

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

这样所有的微服务名称都会忽略掉。

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

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

其中mycompany为逻辑名称，该名称的主要作用是将path与serviceId绑定在一起。

4、【microcloud-zuul-gateway-9501】如果不想通过Eureka进行访问，则也可以直接连接到company微服务的地址：

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

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

但是从实际的开发来将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的地址，那么不方便负载均衡的配置处理，而且所有微服务的管理也非常不方便。

5、【microcloud-zuul-gateway-9501】设置公共前缀：

**zuul:  
 routes:  
 microcloud-provider-company:** /company-proxy/\*\*  
 **ignored-services: "\*"  
 prefix:** /yangmi-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/yangmi-proxy/company-proxy/company/get/hello

以上地址：

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

### Zuul过滤

如果被代理的微服务认证信息，那么就必须在其访问钱追加认证的头部操作，这样的规则就必须要zuul的过滤操作完成。

1. 【microcloud-zuul-gateway-9501】修改application.yml配置文件，这个配置中追加dept微服务的代理；

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

1. 【microcloud-zuul-gateway-9501】创建授权请求操作：

**package** cn.ym.com.filter;  
  
**import** com.netflix.zuul.ZuulFilter;  
**import** com.netflix.zuul.context.RequestContext;  
  
**import** java.nio.charset.Charset;  
**import** java.util.Base64;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \* User: Dony  
 \* Date: 2018/7/5  
 \* Time: 15:50  
 \* Description:  
 \*/***public class** AuthorizedRequestFilter **extends** ZuulFilter {*// 进行授权访问处理* @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"**;  
 }  
  
 @Override  
 **public int** filterOrder() {*// 设置优先级，数字越大优先级越低,0最大* **return** 0;  
 }  
  
 @Override  
 **public boolean** shouldFilter() { *// 该Filter是否要执行* **return true**;  
 }  
  
 @Override  
 **public** Object run() {*// 表示具体的过滤执行操作* RequestContext requestContext = RequestContext.*getCurrentContext*(); *// 获取当前请求的上下文* String auth = **"yangmi:mi"**; *// 认证的原始信息* **byte**[] encodeAuth = 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(encodeAuth);  
 requestContext.addZuulRequestHeader(**"Authorization"**, authHeader);  
 **return null**;  
 }  
}

1. 【microcloud-zuul-gateway-9501】建立一个配置程序类，配置Bean：

@Configuration  
**public class** ZuulConfig {  
 @Bean  
 **public** AuthorizedRequestFilter getAuthorizedRequestFilter() {  
 **return new** AuthorizedRequestFilter();  
 }  
}

此时就意味着可以直接利用zuul的代理访问所有加密的微服务。

1. 【microcloud-zuul-gateway-9501】考虑到zuul也要进行安全访问，所以修改项目中的pom.xml配置，追加springsecurity安全依赖包；

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

1. 【microcloud-zuul-gateway-9501】修改application.yml，配置用户信息；

**security:  
 basic:  
 enabled:** true  
 **user:  
 password:** ym  
 **name:** yangmi

此时表示zuul代理上有了认证信息，则访问我的地址上应该加上zuul的认证操作：

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；

@FeignClient(value = **"MICROCLOUD-ZUUL-GATEWAY"**, configuration = FeignClientConfig.**class**,fallbackFactory = IDeptClientServiceFallbackFactory.**class**)  
**public interface** IDeptClientService {  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/yangmi-proxy/dept-proxy/dept/get/{id}"**)  
 Dept get(@PathVariable(**"id"**) **long** id);  
  
 @RequestMapping(method = RequestMethod.***GET***, value = **"/yangmi-proxy/dept-proxy/dept/list"**)  
 List<Dept> list();  
  
 @RequestMapping(method = RequestMethod.***POST***, value = **"/yangmi-proxy/dept-proxy/dept/addo"**)  
 **boolean** add(Dept dept);  
}

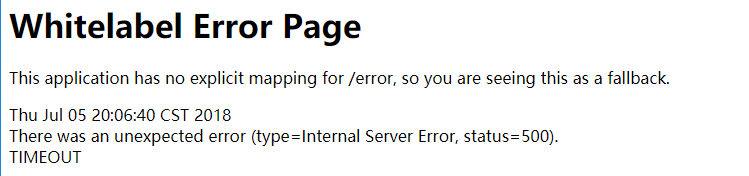
1. 【microcloud-service】修改服务的配置类，此时访问的是zuul的地址：

@Configuration  
**public class** FeignClientConfig {  
 @Bean  
 **public** Logger.Level getFeignLoggerLevel(){  
 **return** feign.Logger.Level.***FULL***;  
 }  
  
 @Bean  
 **public** BasicAuthRequestInterceptor getBasicAuthRequestInterceptor() {  
 *//feign 远程rest认证* **return new** BasicAuthRequestInterceptor(**"yangmi"**, **"ym"**);  
 }  
}

8、【microcloud-zuul-gateway-9501】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，如果有些过滤器不想使用，则可以修改application.yml配置文件让其禁用。

### Zuul失败回退

zuul是一个代理服务，如果被代理的服务突然断掉了，那么这个时候zuul上面一定回显示出错误信息。例如：现在断掉“dept-8001:8001”端口上的微服务，而后输入以下的代理地址：<http://gateway-9501.com:9501/yangmi-proxy/dept-proxy/dept/get/1>



由于客户端配置feign配置了降级处理，所以客户端是没有任何问题的，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-gateway-9501】建立一个FallBack的回退处理类。

@Component  
**public class** DeptProviderFallback **implements** ZuulFallbackProvider {  
 @Override  
 **public** String getRoute() {  
 **return "microcloud-provider-dept"**; *// 设置好处理的失败路由* }  
  
 @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\":777777,\"dname\":\"【ERROR】Zuul-Fallback\",\"loc\":\"Gateway客户端提供\"}"**.getBytes());  
 }  
  
 @Override  
 **public** HttpHeaders getHeaders() {  
 HttpHeaders httpHeaders = **new** HttpHeaders();  
 httpHeaders.set(**"Content-Type"**, **"text/html; charset=UTF-8"**);  
 **return** httpHeaders;  
 }  
 };  
 }  
}

1. 【microcloud-zuul-gateway-9501】重启zuul服务，访问地址：

http://gateway-9501.com:9501/yangmi-proxy/dept-proxy/dept/get/1

## 上传微服务

### 搭建上传微服务

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。

1. 新建项目模块【microcloud-provider-upload-8201】。

*<?***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"**>  
 <**parent**>  
 <**artifactId**>microcloud</**artifactId**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>microcloud-provider-upload-8201</**artifactId**>  
  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 </**properties**>  
 <**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**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-security</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>cn.ym.com</**groupId**>  
 <**artifactId**>microcloud-security</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**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**>  
</**project**>

1. 【microcloud-provider-upload-8201】修改application.yml配置文件，对上传文件配置：

**server:  
 port:** 8201  
**spring:  
 http:  
 multipart:** *# 启用http上传处理* **enabled:** true  
 *# 设置单个文件的最大长度* **max-file-size:** 100MB  
 *# 当上传文件达到1MB的时候进行磁盘写入* **file-size-threshold:** 2MB  
 *# 设置最大的请求文件的大小* **max-request-size:** 100MB  
 **location:** /  
 *# 上传的临时目录* **application:   
 name:** microcloud-provider-upload  
  
**eureka:  
 client:** *# 客户端进行Eureka注册的配置* **service-url:  
 defaultZone:** http://yangmi:ym@eureka-7001.com:7001/eureka,http://yangmi:ym@eureka-7002.com:7002/eureka,http://yangmi:ym@eureka-7003.com:7003/eureka  
 **instance:  
 lease-renewal-interval-in-seconds:** 2 *# 设置心跳的时间间隔（默认是30秒）* **lease-expiration-duration-in-seconds:** 5 *# 如果现在超过了5秒的间隔（默认是90秒）* **instance-id:** dept-8001.com *# 在信息列表时显示主机名称* **prefer-ip-address:** true *# 访问的路径变为IP地址***info:  
 app.name:** mldn-microcloud  
 **company.name:** www.mldn.cn  
 **build.artifactId:** $project.artifactId$  
 **build.version:** $project.verson$

1. 【microcloud-provider-upload-8201】建立上传的rest服务配置：

@RestController  
**public class** UploadRest {  
  
 @RequestMapping(value = **"/upload"**, method = RequestMethod.***POST***)  
 @HystrixCommand(fallbackMethod = **"uploadFallback"**)  
 **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 "yangmi-file-"** + System.*currentTimeMillis*() + **".jpg"**;  
 }  
  
 **public** String uploadFallback(@RequestParam(**"photo"**) MultipartFile photo) {  
 **return "nophoto.jpg"**;  
 }  
}

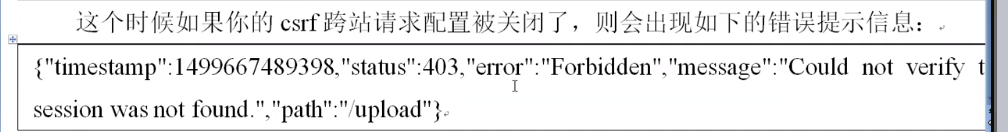
1. 【microcloud-provider-upload-8201】启动服务：

@SpringBootApplication  
@EnableEurekaClient  
@EnableCircuitBreaker  
@EnableDiscoveryClient  
**public class** Upload\_8201\_StartSpringCloudApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Upload\_8201\_StartSpringCloudApplication.**class**, args);  
 }  
}

1. 如果要进行上传测试，可以简单一些，使用curl的命令完成（前提是先安装），当然这个命令是需要单独配置的；
2. 修改本机hosts，追加一个访问映射地址：127.0.0.1 upload.com



curl -F [“photo=@D:\girls.jpg”](mailto:\“photo=@dog.jpg\”) http://yangmi:ym@upload.com:8201/upload

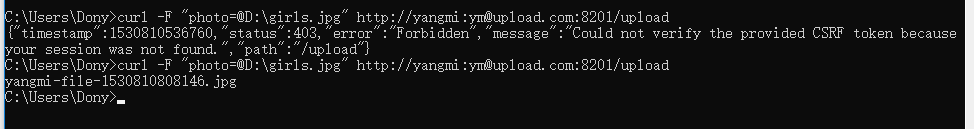


跨站的访问如果安全前提下一定要被禁止的，但是如果牵扯到了微服务的架构问题，这个时候就不蹦禁止了。

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

@Configuration  
@EnableWebSecurity  
**public class** WebSecurityConfig **extends** WebSecurityConfigurerAdapter {  
  
 @Override  
 **public void** configure(WebSecurity web) **throws** Exception {  
 web.ignoring().antMatchers(**"hystrix.stream"**, **"turbine.stream"**);  
 }  
  
 @Resource  
 **public void** configGlobal(AuthenticationManagerBuilder authenticationManagerBuilder) **throws** Exception {  
 authenticationManagerBuilder.inMemoryAuthentication().withUser(**"yangmi"**).password(**"ym"**).roles(**"USER"**).and().withUser(**"admin"**).password(**"admin"**).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().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***);  
 }  
}

6、【microcloud-provider-upload-8201】再次执行上传；





### 使用zuul代理上传

### 客户端调用上传微服务

## SpringCloudConfig基础配置

## SpringCloudConfig高级配置

## SpringCloudStream简介

## SpringCloudSleuth