# 1.springcloud几个面试题

**1.**什么是微服务？

**2.**微服务之间的如何独立通讯的？

**3.**spring Cloud和Dubbo有哪些区别？

**答：**Sring Cloud是基于STful API调用。Dubbo是基于远程的RPC调用

**4.**springboot和springcloud，请你谈谈你对他们的理解？

**5.**什么是服务熔断？什么是服务降级？

**6.**微服务的优缺点是什么？说下你在项目中遇到的坑？

**7.**你所知道的微服务技术栈有哪些？请列举一二

**8.**euraka和zookeeper都也可以提供微服务注册和发现的功能，请说说两个的区别？

**用进废退，物竞天择！！！**

## 1.1.微服务是什么？

**答：**微服务的核心是将传统的一站式的应用，根据业务分成多个服务，彻底去耦合，每一个微服务提供单个业务功能的服务，一个服务只做一件事情。从技术角度看，就是一种小而独立的处理过程，类似进程的概念，能否自行独立启动或则销毁，拥有自己独立的数据库。

**练武不练功，到老一场空**

## 1.2.微服务与微服务架构

**微服务：**强调的是服务的大小，关注的是具体的某一个点，具体解决某一个问题/提供落地对应的一个服务应用，狭义的看，可以看作是idea里面的一个个微服务工程或则module。

（强调的是个体，每一个个体完成一个具体的任务或则功能）

**微服务架构：**是一种模式，提倡将单一的应用划分为一组小的服务，服务之间的相互协调，相互配合，每一个服务都运行在独立的进程中，服务之间采用轻量级的通信机制（HTTP协议的RESTful API）

## 1.3.微服务的优缺点

**优点：**

**1.**每一个服务**足够内聚**，足够小，代码容易理解，能聚焦一个指定的业务功能或者业务需求。

**2.**开发简单，开发效率提高，一个服务只干一件事情

**3.**微服务是**松耦合的**，是有功能意义的服务，无论是开发阶段或则部署阶段都是独立的

**4.**微服务可以用不同的语言开发

**5.**容易和第三方集成，微服务能容易且灵活的方式集成自动部署

**6.**微服务只是业务逻辑的代码，不会和html，css或则其他界面组件混合

**7.**每一个微服务都有自己的存储能力，**可以有自己的数据库**，也可以有统一的数据库。

**缺点：**

**1.**开发人员要处理分布式系统的复杂性

**2.**多服务运维的难度，随着服务的增加，运维的压力也在增大

**3.**系统部署依赖，服务间通信的成本

**4.**数据的一致性，系统集成测试，性能监控

## 1.4.微服务的技术栈

**1.**服务开发：Springboot，Spring，SpringMvc

**2.**服务配置与管理：Netflix公司的Archaius，阿里的Diamond等

**3.**服务注册与发现：Eureka，Consul，Zookeeper

**4.**服务调用：Rest，RPC，gRPC

**5.**服务熔断器：Hystrix，Envoy

**6.**负载均衡：Ribbon，Ngnix

**7.**服务接口调用（客户端调用微服务的简化工具）：Feign等

**8.**消息队列：Kafka，RabbitMQ，ActiveMQ等

**9.**服务配置中心管理：SpringCloudConfig，Chef等，

**10.**服务路由（API网管）：zuul

**11.**服务监控：Zabbix，Nagios，Metrics，Spectator等

**12.**全链路追踪：Zipkin,Brave,Dapper等

**13.**数据流操作开发包：SpringCloud Stream（封装与Redis，Rabbilt，Kafika等发送接受消息）

**14.** 服务部署：Docker，OpenStack，Kubernetes等

**15.**事件消息总线：Spring Cloud Bus

## 1.5.为什么使用Springcloud作为微服务架构

支持完整的微服务框架。相对于其他框架，相对比较全面，热度很高。

## 1.6. Springcloud和SpringBoot的关系

**答：**

**1.**Springboot专注于快速方便的开发单个个体微服务

**2.**Springcloud关注全局的微服务协调治理框架，它将SpringBoot开发的一个个单体服务整合并管理起来。为各个微服务之间提供：**配置管理，服务发现，断路器，路由，微代理，事件总线，全局锁，决策竞选，分布式会话等等集成服务**。

**3.**SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于**依赖关系**。

**4.**SpringBoot专注于快速开发单个微服务个体，SpringCloud关注于全局的微服务治理框架

## 1.7. Springcloud和Dubbo的区别：

**1.**活跃度：SpringCloud比Dubbo活跃度高

**2.**最大的区别是：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的Rest方法。后者牺牲了服务调用的性能，但是避免了RPC带来的问题，而且REST相比RPC更为灵活。不存在代码级别的依赖。

**3.**品牌机和组装机的区别：Springcloud功能比Dubbo更加强大，跟其他Spring的框架整合也更加完美。

## 1.8.SprinCloud能干什么

1.分布式/版本控制配置

2.服务注册与发现

3.路由

4.服务到服务的调用

5.负载均衡配置

6.断路器

7.分布式消息管理

## 1.9. SprinCloud怎么玩儿

**1.**服务注册与发现（Eureka）

**2.**服务消费者（rest+Ribbon）

**3.**服务消费者（Feign）

**4.**断路器（Hystrix）

**5.**断路器监控（Hystrix DashBoard）

**6.**路由网关（Zuul）

**7.**分布式配置中心（Spring Cloud Config）

**8.**消息总线（Spring Cloud Bus）

**9.**服务链路追踪（Spring Cloud Sleuth）

**总结：到这里了解了SpringCloud微服务的理论知识，接下来对具体的落地实现，做记录**

# 2.搭建父子工程结构

## 1.创建父工程

**1.**首先创建一个名为springcloudparent的父工程模块，需要将pom.xml中打包方式改为<packaging>pom</packaging>。

**2.**创建步骤：File-----New------Project------Maven------Next----然后填写好对应的Groupid和Artifactid------最后点击Finish

**3.**创建完成后，在pom.xml中添加如下配置：

<modelVersion>4.0.0</modelVersion>  
  
<groupId>com.ecology</groupId>  
<artifactId>springcloud-parent</artifactId>  
<packaging>pom</packaging>  
<version>1.0-SNAPSHOT</version>  
  
<properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
 <junit.version>4.12</junit.version>  
 <log4j.version>1.2.17</log4j.version>  
 <lombok.version>1.16.18</lombok.version>  
</properties>  
  
<!--这里管理 引入了公用的模块-->  
<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.16.18</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Dalston.SR1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-dependencies</artifactId>  
 <version>1.5.9.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>  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-core</artifactId>  
 <version>1.2.3</version>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>${junit.version}</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>${log4j.version}</version>  
 </dependency>  
 </dependencies>  
</dependencyManagement>  
<!--这个插件很重要，**后面会讲解到为什么要引入** -->  
<build>  
 <finalName>microservicecloud</finalName>  
 <!--表示打包时，将resources目录下的配置文件一并打入。-->  
 <resources>  
 <resource>  
 <directory>src/main/resources</directory>  
 <filtering>true</filtering>  
 </resource>  
 </resources>  
 <plugins>  
 <!--首先开启资源配置的插件，由此插件替换占位符-->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-resources-plugin</artifactId>  
 <configuration>  
 <delimiters>  
 <delimiter>$</delimiter>  
 </delimiters>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

4.父工程创建完毕，由于父工程中不需要写代码，可以直接将src文件夹删除掉。

## 2.创建子工程

### 2.1创建服务提供者

#### 1.创建工程：microservicecloud-provider-dept-8001

在springcloudparent上右键------New------Module------Maven------Next------填写好Artifactid（名为：microservicecloud-provider-dept-8001）------Next------Finish

#### 2.配置pom.xml:

<!--依赖的父模块-->  
<parent>  
 <artifactId>springcloud-parent</artifactId>  
 <groupId>com.ecology</groupId>  
 <version>1.0-SNAPSHOT</version>  
</parent>  
<modelVersion>4.0.0</modelVersion>  
<artifactId>microservicecloud-provider-dept-8001</artifactId>  
<!-- **当前Module需要用到的jar包，按自己需求添加，如果父类已经包含了，可以不用写版本号** -->  
<dependencies>  
 <!-- 引入自己定义的api通用包，可以使用User部门Entity -->  
 <dependency>  
 <groupId>com.ecology</groupId>  
 <artifactId>microservicecloud-api</artifactId>  
 <version>${project.version}</version>  
 </dependency>  
 <!--引入junit-->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 </dependency>  
 <!--数据库支持-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.48</version>  
 </dependency>  
 <dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-core</artifactId>  
 </dependency>  
 <!--springboot支持-->  
 <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>  
 </dependency>  
 <!-- 修改后立即生效，热部署 -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>springloaded</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
 </dependency>  
</dependencies>

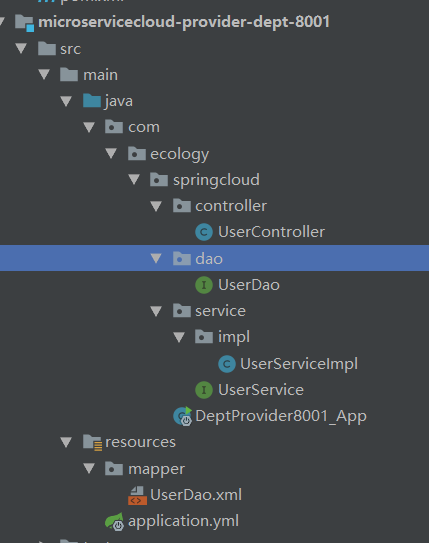
#### 3.配置application.yml:

*#服务端口***server**:  
 **port**: 8001  
*#mybatis配置***mybatis**:  
 **mapper-locations**: classpath:mapper/\*.xml *# mapper映射文件***spring**:  
 *#spring应用名称* **application**:  
 **name**: microservicecloud-dept  
 *#数据库配置* **datasource**:  
 **url**: jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8&useSSL=false  
 **username**: root  
 **password**: root  
 **type**: com.alibaba.druid.pool.DruidDataSource  
 **driver-class-name**: com.mysql.jdbc.Driver

#### 4.集成mybatis

**1.**为了快速搭建springboot集成mybatis，从数据库到controller功能的实现。我们使用了**easycode**（该插件的使用，另行百度）的插件，会自动生成相关代码结构如下图：（需要注意的是，在application.xml

配置好的前提下，还要在UserDao接口中，手动加上注解@Mapper）：



到目前为止，在浏览器访问对应的路径，就能查询到数据库对应的数据。

### 2.2.创建服务消费者：

#### 1.创建工程：microservicecloud-consumer-dept-8000

创建步骤同：服务提供者

#### 2.创建ConfigBean配置类

在cfgbeans文件夹中创建ConfigBean.java，为了在上下文中创建好RestTemplate对象

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

#### 3.创建controller层调用

在controller文件夹下，创建DeptController\_Consumer.java，其中引入上面创建好的RestTemplate对象，然后进行调用：

@RestController  
public class DeptController\_Consumer{  
 private static final String *REST\_URL\_PREFIX* = "http://localhost:8001";  
 */\*\*  
 \* 使用 使用restTemplate访问restful接口非常的简单粗暴无脑。 (url, requestMap,  
 \* ResponseBean.class)这三个参数分别代表 REST请求地址、请求参数、HTTP响应转换被转换成的对象类型。  
 \*/* @Autowired  
 private RestTemplate restTemplate;  
 @RequestMapping(value = "/consumer/user/get/{id}")  
 public User get(@PathVariable("id") Long id){  
 // return restTemplate.getForObject(REST\_URL\_PREFIX + "/user/selectOne?id=" + id, User.class);  
 return restTemplate.getForObject(*REST\_URL\_PREFIX* + "/user/selectOne/" + id, User.class);  
 }  
}

**注意：**

访问路径是：**REST\_URL\_PREFIX + "/user/selectOne/" + id**。则在服务提供者的controller配置为：

@RequestMapping("/selectOne/{id}")  
public User selectOne(@PathVariable Integer id) {}

访问路径是：**REST\_URL\_PREFIX + "/user/selectOne?id=" + id**。则在服务提供者的controller配置为：

@GetMapping("selectOne")  
public User selectOne(Integer id) {}

### 2.3.创建公用的api模块

#### 1.创建工程：microservicecloud-api

#### 2.创建entity类，并使用maven clean install打包

需要注意的是：服务提供者的entity部分，移动到这个模块下，进行公用。然后在其他模块通过坐标引入即可。服务者和消费者的pom.xml已经配置好。

到这里搭建的所有步骤就已经搭建好了

# 3.Eureka服务注册和发现

## 1.是什么

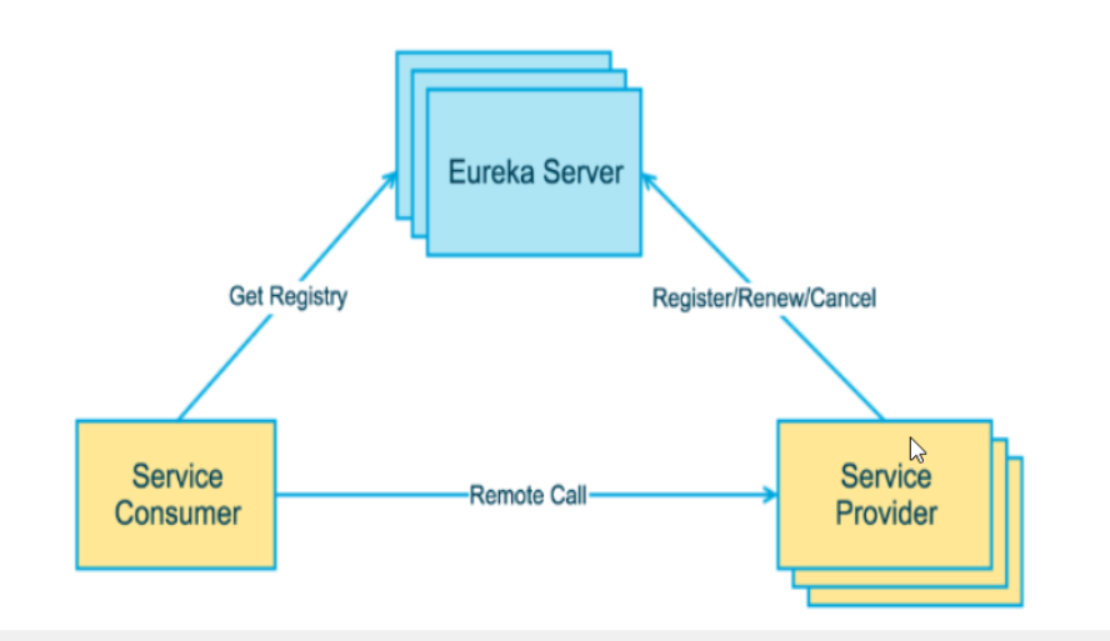
**1.**Netflix在设计Eureka时候，遵守的就是**AP原则**

**2.** Eureka是Netflix的一个子模块，Eureka是一个**基于REST的服务**，用于定位服务，以实现云端中间层服务发现和故障转移，服务注册和发现对于微服务架构来说非常重要的，有了服务发现与注册，**只需要使用服务的标识符，就可以访问到服务**，而不需要修改服务的配置文件，**功能类似于dubbo的注册中心，比如Zookeeper**

## 2.原理

**1.**Springcloud 封装了NetFlix公司开发的Eureka模块来实现服务注册和发现

**2.**Eureka采用了C-S的设计架构，**Eureka Server作为服务注册功能的服务器，它是服务注册中心**，而系统中的其它微服务，使用**Eureka的客户端**连接到Eureka Server并维持心跳连接，这样系统的维护人员就可以通过Eureka Server来监控系统中各个微服务是否正常运行。Springcloud的一些其他模块（比如Zuul）就可以通过Eureka Server来发现系统中的其他微服务，并执行相关的逻辑。



**3.**Eureka包含两个组件：**Eureka Server**和**Eureka Client**

**4.**Eureka Server提供注册服务，各个节点启动后，会在Eureka Server中进行注册，这样EurekaServer中的服务注册表中将会储存所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。

**5.**Eureka Client是一个java客户端，用于简化Eureka Server的交互，**客户端同时也具备一个内置的，使用轮询的负载算法的负载均衡器**，在应用启动后，将会向Eureka Server发送心跳（默认周期为30秒），如果Eureka Server在多个心跳周期内没有接收到一个节点的心跳，EurekaServer将会从服务注册中心把这个服务节点移除。

## 3.创建Eureka Server服务

### 1. 创建microservicecloud-eureka-7001

创建配置如上

### 2.pom.xml

只是截取了关键的配置

<dependencies>  
 <!--eureka-server服务端 -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka-server</artifactId>  
 </dependency>  
 <!-- 热部署插件 -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>springloaded</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
 </dependency>  
</dependencies>

### 3.application.yml

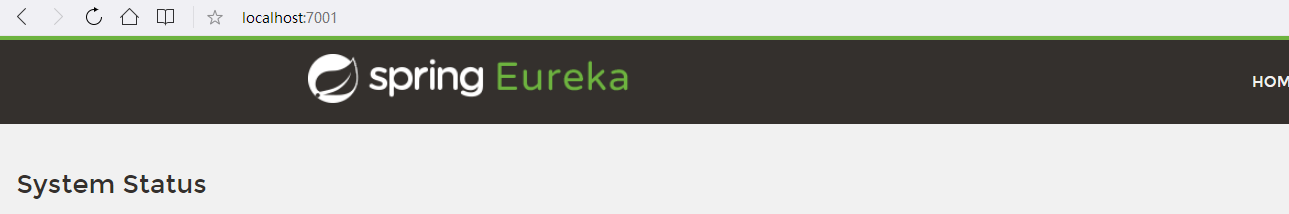
**server**:  
 **port**: 7001  
**eureka**:  
 **instance**:  
 **hostname**: localhost *#eureka服务端的实例名称* **client**:  
 **register-with-eureka**: **false** *#false表示不向注册中心注册自己。* **fetch-registry**: **false** *#false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务* **service-url**:  
 *#设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址。* **defaultZone**: http://${**eureka.instance.hostname**}:${**server.port**}/eureka/

### 4.创建主启动类:

创建EurekaServer7001\_App.java

@SpringBootApplication  
@EnableEurekaServer // EurekaServer服务器端启动类,接受其它微服务注册进来  
public class EurekaServer7001\_App{  
 public static void main(String[] args){  
 SpringApplication.*run*(EurekaServer7001\_App.class, args);  
 }  
}

经过上面的配置，就已经配置好了Eureka Server服务端。在浏览器中访问localhost:7001/即可出现对应的页面。



## 4.创建Eureka的客户端

在这里我们修改前面创建的microservicecloud-provider-dept-8001服务，使其注册进Eureka服务端

### 1.改造pom.xml文件

新增部分：

<!-- 将微服务provider侧注册进eureka -->  
<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>

### 2.改造application.yml

新增部分：

**eureka**:  
 **client**: *#客户端注册进eureka服务列表内* **service-url**:  
 **defaultZone**: http://localhost:7001/eureka

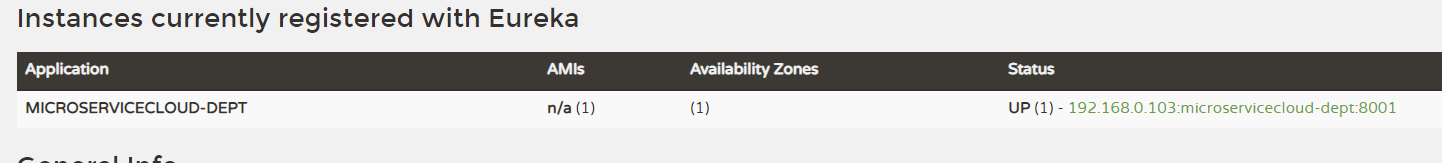
### 3.改造主启动类：

新增部分：

@SpringBootApplication  
@EnableEurekaClient  
public class DeptProvider8001\_App{  
 public static void main(String[] args){  
 SpringApplication.*run*(DeptProvider8001\_App.class, args);  
 }  
}

### 4.效果：

这样我们8001的服务就注册进了Eureka服务器中，这里改造的逻辑大致为：**添加对应的maven模块，配置好application.yml相关配置，在主启动类中添加对应的注解@EnableEurekaClient**。

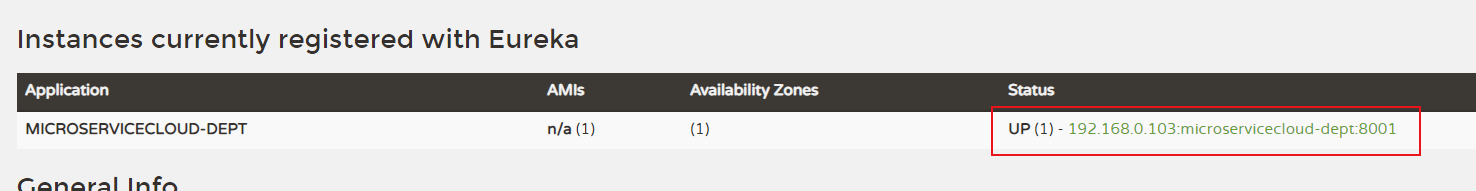


其中的**MICROSERVICECLOUD-DEPT**为application.yml中配置的spring应用名称

## 5.actuator与注册微服务信息完善

### 1.主机名称：服务名称修改

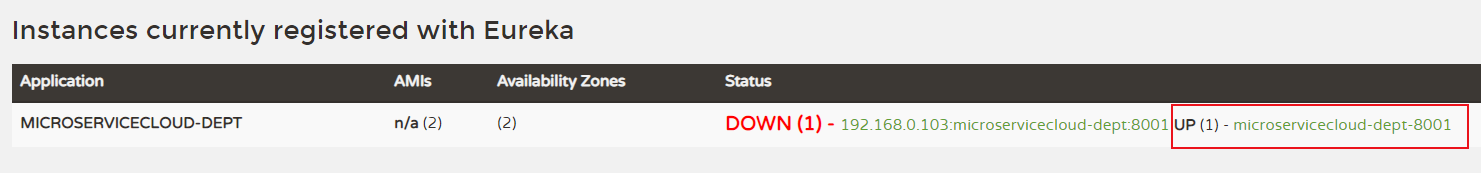
修改之前：



修改application.yml:

**eureka**:  
 **client**: *#客户端注册进eureka服务列表内* **service-url**:  
 **defaultZone**: http://localhost:7001/eureka  
  **instance:  
 instance-id: microservicecloud-dept-8001**

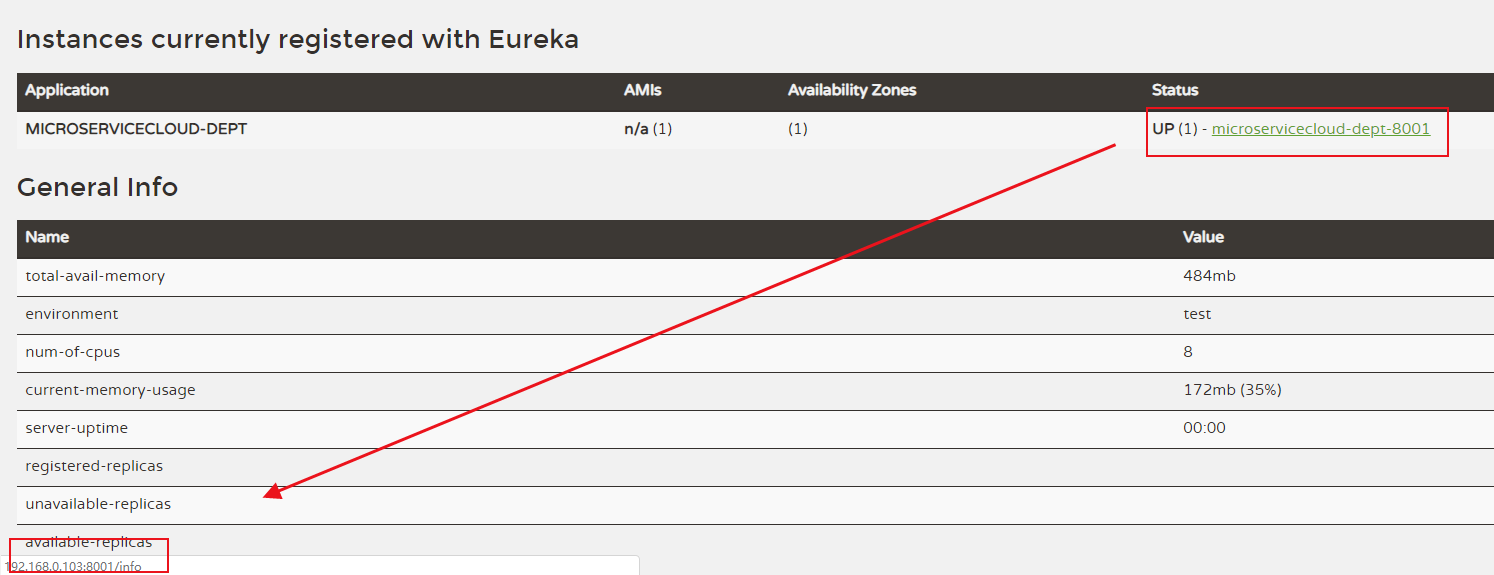
修改之后：



这里的down和up是**自我保护机制**的效果，这里先不做解释，如果重启下服务，正常显示

### 2. 主机IP信息提示

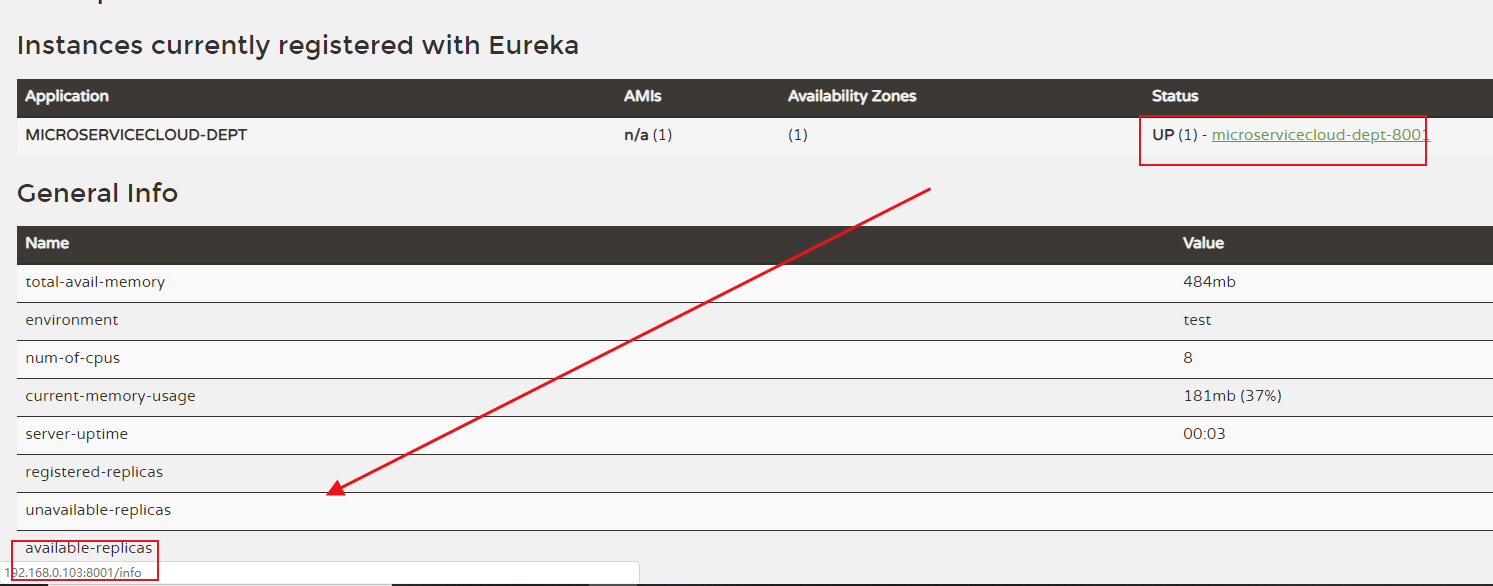
修改效果前：



修改application.yml:

**eureka**:  
 **client**: *#客户端注册进eureka服务列表内* **service-url**:  
 **defaultZone**: http://localhost:7001/eureka  
 **instance**:  
 **instance-id**: microservicecloud-dept-8001  
 **prefer-ip-address**: **true** *#访问路径可以显示IP地址*

修改后：



注意：这里演示的效果跟预计的不一样，本身修改效果前应该是localhost:8001/info,但是实际上显示除了ip地址，这里暂且先记录下这个application.yml中的配置

### 3. info内容构建

1.当前问题是点击上面的链接，会显示error错误页面

2.修改pom.xml:增加部分：

<!-- actuator监控信息完善 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

3.application.yml:

**info**:   
 **app.name**: atguigu-microservicecloud  
 **company.name**: www.atguigu.com  
 **build.artifactId**: $project.artifactId$  
 **build.version**: $project.version$

4.在父工程的pom.xml中增加：

<!--这个插件很重要，明天需要查询下资料-->  
<build>  
 <finalName>microservicecloud</finalName>  
 <!--表示打包时，将resources目录下的配置文件一并打入。-->  
 <resources>  
 <resource>  
 <directory>src/main/resources</directory>  
 <filtering>true</filtering>  
 </resource>  
 </resources>  
 <plugins>  
 <!--首先开启资源配置的插件，由此插件替换占位符-->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-resources-plugin</artifactId>  
 <configuration>  
 <delimiters>  
  **<delimiter>$</delimiter>**  
 </delimiters>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

**注意：Delimiter不要写成delimit，不然子工程中的application中$xxxx.xxxx$不会解析**

## 6.Eureka自我保护

**1.**微服务某一时刻不可用了，eureka不会立刻清理，依旧会对微服务的信息进行保存。有可能因为网络拥堵或则调用超时，eureka还是会保留

**2.**默认情况下，如果Eureka Server在一定时间内没有收到某一个微服务实例心跳，EurekaServer将会注销该实例（默认是90秒）。但是当网络分区故障发生时，微服务域EurekaServer之间无法正常通信，以上行为可能会变成非常危险的，因为微服务本身是健康的，此时不应该注销这个微服务，Eureka通过“自我保护”，来解决这个问题，

**3.**一旦进去自我保护模式，EurekaServer就会保护服务注册表中的信息，不再删除注册服务表中的数据（也不会注销任何微服务），当网络故障恢复后，该EurekaServer节点会自动退出自我保护模式。

**4.**它的设计哲学是：**宁可保护错误的服务注册信息，也不盲目注销任何可能健康的服务实例**。

## 7.服务发现

通过注册进Eureka里面的微服务，可以通过服务发现来获得该服务的信息，大致的意思是：对于注册进Eureka的服务信息，获取到该服务的信息，大致演示下：

### 1.修改microservicecloud-provider-dept-8001

#### 1.修改controller类中，

新增方法：

import org.springframework.cloud.client.discovery.DiscoveryClient;

@Autowired  
private DiscoveryClient client;

@RequestMapping(value = "/dept/discovery", method = RequestMethod.*GET*)  
public Object discovery(){  
 List<String> list = client.getServices();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*" + list);  
 List<ServiceInstance> srvList = client.getInstances("MICROSERVICECLOUD-DEPT");  
 for (ServiceInstance element : srvList) {  
 System.*out*.println(element.getServiceId() + "\t" + element.getHost() + "\t" + element.getPort() + "\t"+ element.getUri());  
 }  
 return this.client;  
}

#### 2.修改主启动类：

@SpringBootApplication  
@EnableEurekaClient/ /开启Eureka客户端  
@EnableDiscoveryClient //服务发现  
public class DeptProvider8001\_App{  
 public static void main(String[] args){  
 SpringApplication.*run*(DeptProvider8001\_App.class, args);  
 }  
}

#### 3.在浏览器中访问

<http://192.168.0.103:8001/user/dept/discovery>即可获取注册进EurekaServer中的信息

#### 4.服务调用者访问该服务发现接口

**1.**新增microservicecloud-consumer-dept-8000服务中的controller方法

// 测试@EnableDiscoveryClient,消费端可以调用服务发现  
@RequestMapping(value = "/consumer/dept/discovery")  
public Object discovery(){  
 return restTemplate.getForObject(*REST\_URL\_PREFIX* + "/user/dept/discovery",Object.class);  
}

然后就能在服务调用端microservicecloud-consumer-dept-8000，访问到服务提供端的服务发现接口，并获取到在Eureka的注册信息（**注意：服务发现只能是服务提供方来获取，服务调用当并不能直接获取**）

## 8. Eureka集群

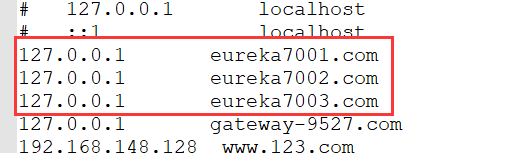
### 1.创建3个集群服务器：

按照microservicecloud-eureka-**7001**服务的创建过程，如法炮制的创建microservicecloud-eureka-**7002**，microservicecloud-eureka-**7003**。并修改对应的**端口**和**主启动类**即可。

### 2.域名的配置

1.找到c:\Windows\System32\driver\etc路径下的hosts文件

2.修改映射配置添加进hosts文件



### 3.修改application.yml

**server**:  
 **port**: 7003  
**eureka**:  
 **instance**:  
 **hostname**: **eureka7003.com** *#eureka服务端的实例名称* **client**:  
 **register-with-eureka**: **false** *#false表示不向注册中心注册自己。* **fetch-registry**: **false** *#false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务* **service-url**:  
 *#设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址。  
 #defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/* **defaultZone**: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/

其中7001和7002的配置同上，修改下对应的端口即可

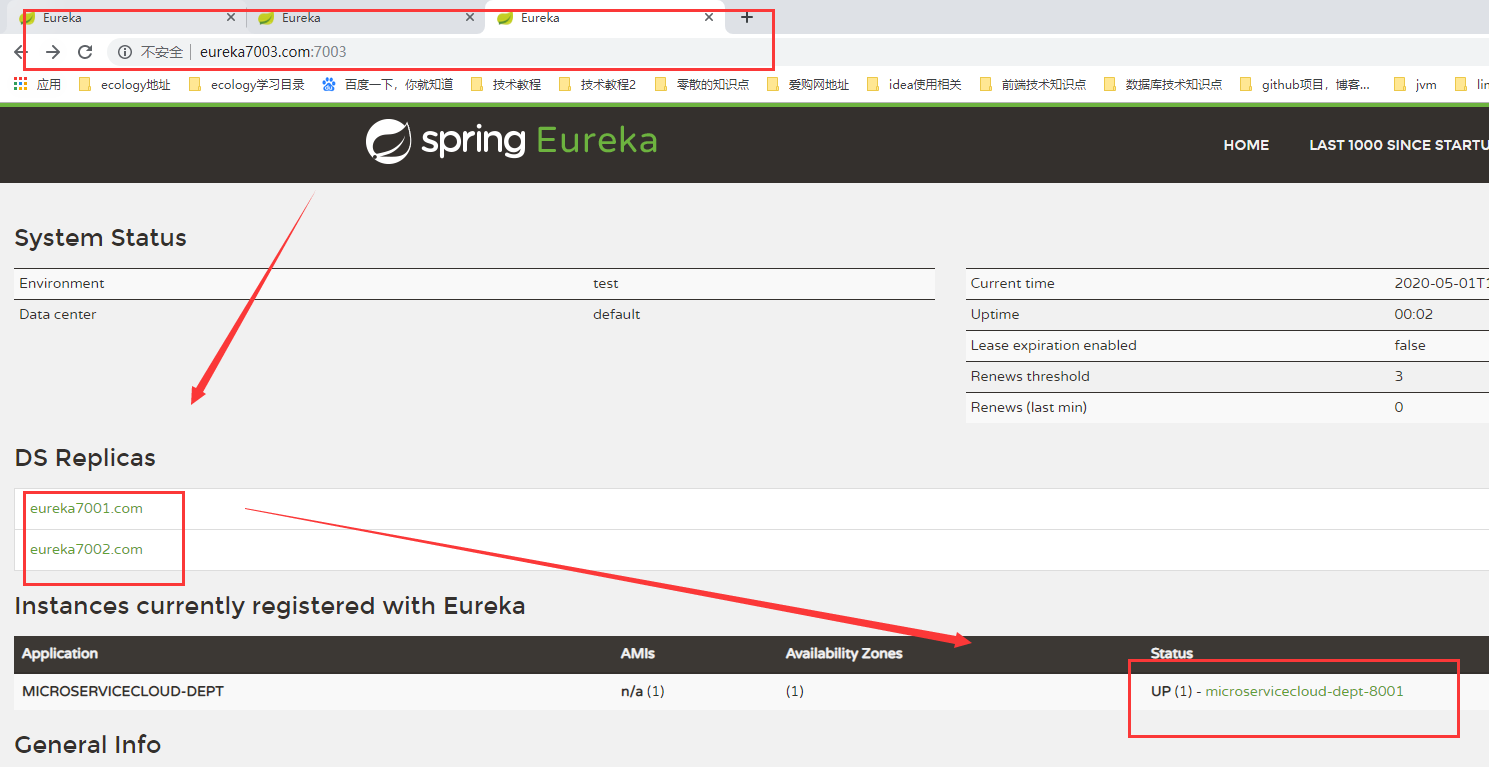
### 4.修改服务提供者8001的application.yml配置

由于做了Eureka集群，服务提供者需要注册进多个Eureka集群中。修改如下：

**eureka**:  
 **client**: *#客户端注册进eureka服务列表内* **service-url**:  
 *#defaultZone: http://localhost:7001/eureka* **defaultZone**: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

### 5.验证

到目前为止，服务提供者 注册进了3个EurekaServer



## 9.Eureka比Zookeeper好在哪里

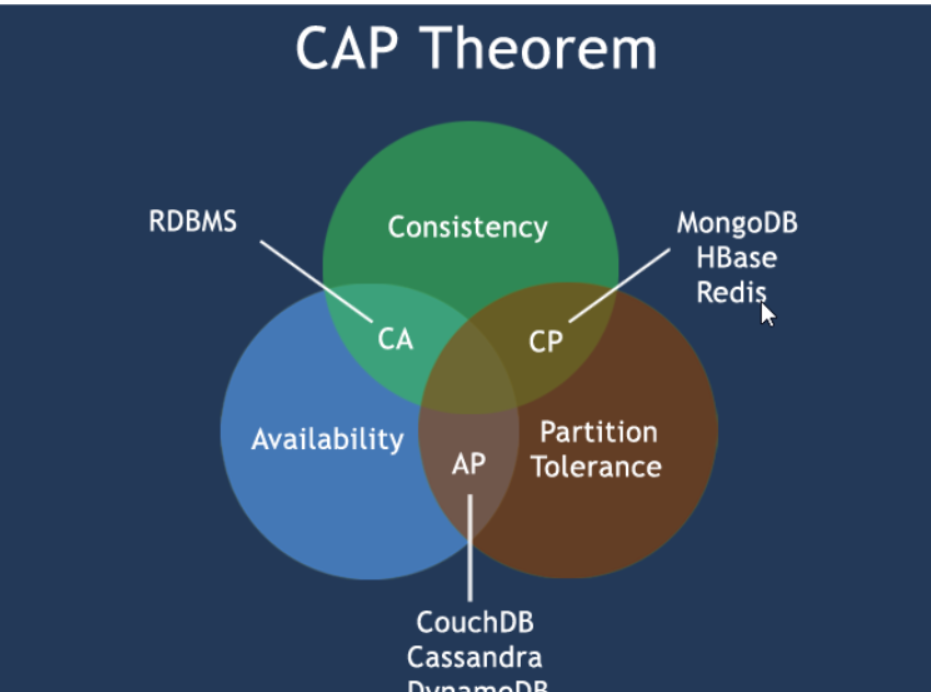
### 1.先了解下CAP概念（nosql）：

**C：**Consistency（强一致性）

**A:**  Availability（可用性）

**P:**  Partition tolerance（分区容错性）

任何一个分布式系统没有办法3个都足，只能3选2



CAP理论的核心：一个分布式系统不能同时很好的满足**一致性，可用性，分区容错性**。因此根据CAP原理将Nosql数据库分解如下：

**CA:**单点集群，满足一致性，可用性的系统，通常咋扩展性上不太强大

**CP:**满足一致性，分区容错的系统，通常性能不是特别的高

**AP:**满足可用性，分区容错的系统，通常对一致性要求低一点

### 2.进一步解释

著名的CAP理论指出：一个分布式系统不可能同时满足C（一致性）,A（可用性），P（分区容错性），由于**分区容错性是分布式系统中必须要保证**的，因此我们只能在A和C之间进行权衡。

因此：**Zookeeper保证的是CP，Eureka保证的AP**

### 3. Zookeeper保证的是CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接Down掉不可用，也就是说，服务注册成功对可用性的要求高于一致性，但是zookeeper会出现这样一种情况，当master节点因为网络故障与其他节点失去联系的时，剩余的节点会重新进行leader选举，问题在于，选择leader的事件太长，30-120s。且**选举期间zk集群都不可用，这就导致选举期间服务注册瘫痪**。

### 4. Eureka保证的AP

Eureka在设计的时候，**优先保证可用性**，Eureka各个节点都是平等的，几个节点挂掉，剩余节点依然能提供服务注册。而Eureka客户端向某个Eureka注册或发生连接失败，则会自动切换其余节点，只要有一台节点还在，就能保证正常可用。只不过查询到的信息可能不是最新的（不保证强一致性）除此之外，**Eureka还有一种自我保护机制**，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障。会出现以下几种情况：

**1.**Eureka不在向注册列表中移除因为长时间没有收到心跳而应该过期的服务

**2.** Eureka仍然能够接受服务的注册和查询请求，但是不会被同步到其他节点上（既保证当前节点依然可用）

**3.**当网络稳定时，当前实例新的注册信息会被同步到其他节点中

**因此：Eureka可以很好地应对因为网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪**

# 4. Ribbon负载均衡

## 1.是什么

**1.**Spring Cloud Ribbon是基于Netflix Ribbon实现的一套**客户端+负载均衡**的工具

**2.**简单的说，Ribbon是Netflix发布的开源项目，**主要功能是提供客户端的负载均衡算法**，将Netflix的中间层服务连接在一起，Ribbon客户端组件提供一系列的配置项，如连接超时，重试等等。简单的说，就是在配置文件中列出Load Balancer（简称LB）后面所有的机器，Ribbon会自动的帮助你基于某种规则（轮询，随机）去连接这些机器，我们也很容易使用Ribbon**实现自定义的负载均衡算法**。

**3.**集中式的负载均衡（LB）

在服务的消费方和提供方之间使用独立的LB设施（可以是硬件，如F5，也可以是软件，如Nginx），由该设施负责把访问请求通过某种策略转发至服务的提供方

**4.**进程内负载均衡

将LB逻辑集中到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择一个合适的服务器。**Ribbon就属于进程内的LB**。他只是一个类库，集成于消费方通过它来获取到服务提供方的地址。

## 2.Ribbon配置

大致思路：**maven坐标+@EnableXxxxx**

### 1. 修改microservicecloud-consumer-dept-8000

修改pom.xml：

<!-- Ribbon相关 -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-ribbon</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>

这里一起引入了Ribbon的组件和Eureka组件，两个是需要一起使用的。

### 2.修改application.yml:

**eureka**:  
 **client**:  
 **register-with-eureka**: **false** *#false表示不向注册中心注册自己。* **service-url**:  
 **defaultZone**: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

### 3.对ConfigBean.java修改

在注册类中添加@LoadBalanced，获取Rest时加入Ribbon的配置

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

### 4.主启动类的修改

在主启动类上添加注解@EnableEurekaClient：

@SpringBootApplication  
**@EnableEurekaClient**  
public class DeptConsumer80\_App{  
 public static void main(String[] args){  
 SpringApplication.*run*(DeptConsumer80\_App.class, args);  
 }  
}

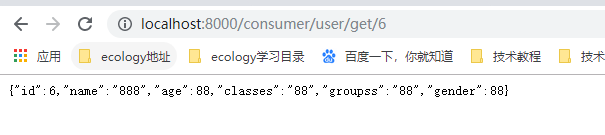
### 5.修改DeptController\_Consume客户端的访问类：

这里修改的目的是通过在eureka注册的服务名进行调用。还不是通过localhost地址

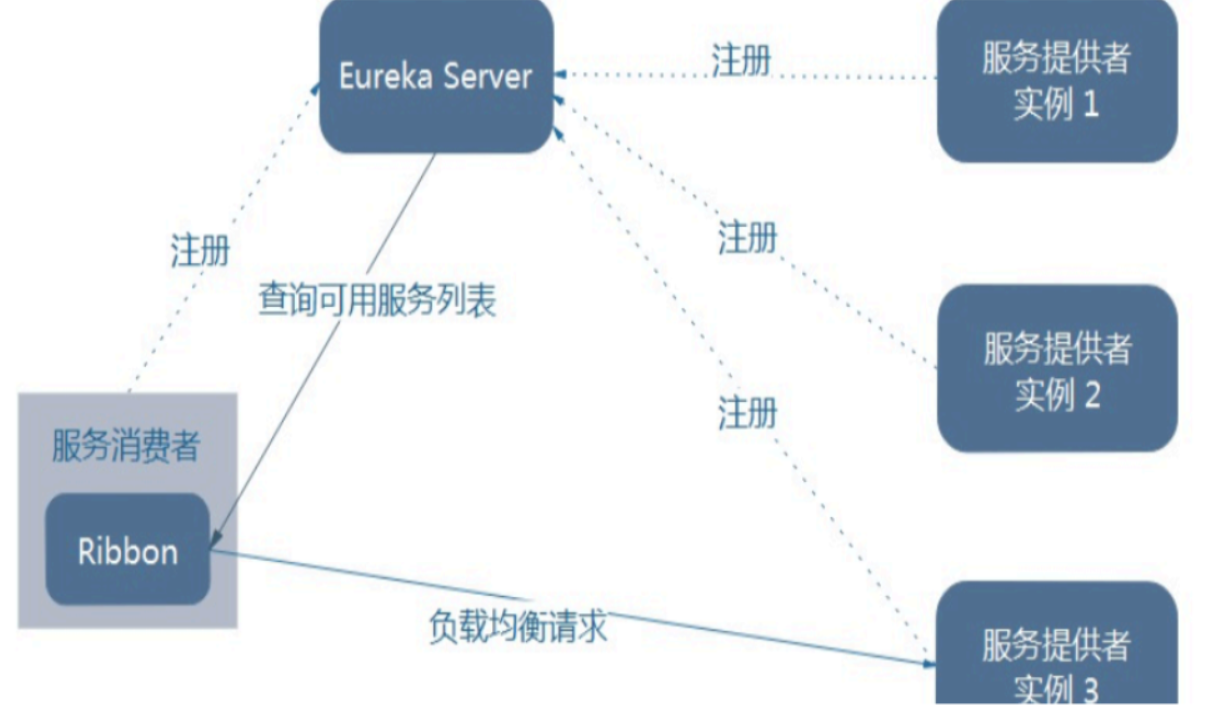
@RestController  
public class DeptController\_Consumer{  
 //private static final String REST\_URL\_PREFIX = "http://localhost:8001";  
 **private static final String *REST\_URL\_PREFIX* = "http://MICROSERVICECLOUD-DEPT";**

}

这里**消费方的Ribbon的配置就完成了**，此时消费端是通过负载均衡进行请求的（这里还演示不出效果，后面继续讲解），现在启动几个服务，测试下



## 3.Ribbon的负载均衡



这里实现了多Eureka集群和服务提供者集群，来演示服务消费者通过负载均衡实现请求

### 1. 创建服务提供者集群服务

根据microservicecloud-provider-dept-8001创建8002和8003端口的提供方，修改**端口**和**主启动类**和**配置文件**等

注意：这里三个服务使用不用的数据库，数据库分别建立一个测试库。

### 2. ConfigBean配置负载均衡算法

microservicecloud-consumer-dept-8000中ConfigBean.java中添加：

@Bean  
public IRule myRule(){  
 //return new RoundRobinRule();  
 return new RandomRule();//达到的目的，用我们重新选择的随机算法替代默认的轮询。  
 //return new RetryRule();  
}

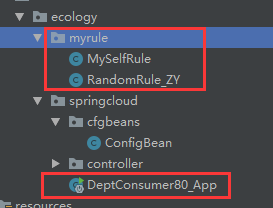
注意：这里不配置的话，默认使用的是轮询算法。配置需要的只需要在上下文声明对应的bean

### 3.启动microservicecloud-consumer-dept-8000测试

经过上面的配置后，使用的随机算法，会随机去调用3个服务提供者

## 4.自定义负载均衡

### [1.在@ComponentScan](mailto:1.在@ComponentScan)扫描范围外新建一个目录：myrules



### 2. MySelfRule.java

@Configuration  
public class MySelfRule{  
 @Bean  
 public **IRule** myRule(){  
 //return new RandomRule();// Ribbon默认是轮询，我自定义为随机  
 //return new RoundRobinRule();// Ribbon默认是轮询，我自定义为随机  
 return new RandomRule\_ZY();// 我自定义为每台机器5次  
 }  
}

### 3.自定义的RandomRule\_ZY.java

继承了AbstractLoadBalancerRule

public class RandomRule\_ZY extends **AbstractLoadBalancerRule**{  
 // total = 0 // 当total==5以后，我们指针才能往下走，  
 // index = 0 // 当前对外提供服务的服务器地址，  
 // total需要重新置为零，但是已经达到过一个5次，我们的index = 1  
 // 分析：我们5次，但是微服务只有8001 8002 8003 三台，OK？  
 private int total = 0; // 总共被调用的次数，目前要求每台被调用5次  
 private int currentIndex = 0; // 当前提供服务的机器号  
 public Server choose(ILoadBalancer lb, Object key){  
 if (lb == null) {  
 return null;  
 }  
 Server server = null;  
 while (server == null) {  
 if (Thread.*interrupted*()) {  
 return null;  
 }  
 List<Server> upList = lb.getReachableServers();  
 List<Server> allList = lb.getAllServers();  
 int serverCount = allList.size();  
 if (serverCount == 0) {  
 /\*  
 \* No servers. End regardless of pass, because subsequent passes only get more  
 \* restrictive.  
 \*/  
 return null;  
 }  
// int index = rand.nextInt(serverCount);// java.util.Random().nextInt(3);  
// server = upList.get(index);   
// private int total = 0; // 总共被调用的次数，目前要求每台被调用5次  
// private int currentIndex = 0; // 当前提供服务的机器号  
 if(total < 5){  
 server = upList.get(currentIndex);  
 total++;  
 }else {  
 total = 0;  
 currentIndex++;  
 if(currentIndex >= upList.size())  
 {  
 currentIndex = 0;  
 }  
 }   
 if (server == null) {  
 /\*  
 \* The only time this should happen is if the server list were somehow trimmed.  
 \* This is a transient condition. Retry after yielding.  
 \*/  
 Thread.*yield*();  
 continue;  
 }  
 if (server.isAlive()) {  
 return (server);  
 }  
 // Shouldn't actually happen.. but must be transient or a bug.  
 server = null;  
 Thread.*yield*();  
 }  
 return server;  
 }  
 @Override  
 public Server choose(Object key) {  
 return choose(getLoadBalancer(), key);  
 }  
 @Override  
 public void initWithNiwsConfig(IClientConfig clientConfig){  
 // *TODO Auto-generated method stub* }  
}

### 4.修改启动类：

@SpringBootApplication  
@EnableEurekaClient  
@RibbonClient(name="MICROSERVICECLOUD-DEPT",configuration= MySelfRule.class)  
public class DeptConsumer80\_App{  
 public static void main(String[] args){  
 SpringApplication.*run*(DeptConsumer80\_App.class, args);  
 }  
}

这里需要添加上**RibbonClient**注解。到这里再次启动时。就是按照自定义的算法进行轮询

# 5.Feign负载均衡

## 1.是什么

Feign是一个声明式的Web服务客户端，使得编写web服务客户端变得非常容易。**是需要创建一个接口，然后再上面添加注解即可**。

## 2.能干什么

**1.**以前在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模板化的调用方法，但是实际开发中，由于对微服务的调用可能不止一处，**往往一个接口会被多处调用，所以通常都会对每一个微服务自行封装一些客户端类来包装这些依赖服务的调用**

**2.**所以，Feign在此基础上做了进一步的封装，由他来帮助我们定义和现实依赖服务接口的定义，**在Feign的实现下，我们只需要创建一个接口，并且使用注解的方式来配置它**（以前是Dao接口上标注Mapper注解，现在是一个微服务接口上面标注一个Feign注解即可），即可完成对微服务提供方的接口绑定，简化了使用Spring Cloud Ribbon时，自动封装服务调用端的开发量

## 3.使用步骤

### 1.创建工程

创建工程名为：microservicecloud-consumer-dept-feign-8888。创建步骤跟80的项目一样

修改好端口，主启动类

### 2.修改pom.xml

在80服务的基础上添加：

<!--Feign相关-->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-feign</artifactId>  
</dependency>

### 3.修改microservicecloud-api

在microservicecloud-api服务中，添加一个公用的接口类DeptClientService.java：

@FeignClient(value = "MICROSERVICECLOUD-DEPT")  
public interface DeptClientService {  
 @RequestMapping(value = "/user/selectOne/{id}", method = RequestMethod.*GET*)  
 public User selectOne(@PathVariable(value = "id") Integer id) ;  
}

这里定义了一个接口，并且标注上FeignClient注解，后期会通过Feign根据微服务名找到对应的接口进行调用。完成后，**mvn clean install**

### 4.8888服务端controller修改

这里需要改造controller通过定义的接口进行调用

@RestController  
public class DeptController\_Consumer{  
 @Autowired  
 private DeptClientService service;  
 @RequestMapping(value = "/consumer1/dept/get/{id}")  
 public User selectOne(@PathVariable(value = "id") Integer id){  
 return this.service.selectOne(id);  
 }  
}

导致的思路就是在8888服务中，通过接口直接调用microservicecloud-api中定义的接口，由于Feign帮我做了通过接口调用的工作。

### 5.主启动类的修改

@SpringBootApplication  
@EnableEurekaClient  
**@EnableFeignClients(basePackages= {"com.ecology.springcloud"})**  
public class DeptConsumer8888\_App{  
 public static void main(String[] args) {  
 SpringApplication.*run*(DeptConsumer8888\_App.class, args);  
 }  
}

### 6.验证通过

**由于Feign集成了Ribbon。所以也支持负载均衡**，使用方法同Ribbon一样，已经在项目中进行了验证

# 6.Hystrix

## 1.分布式系统面临的问题

**1.**复杂的分布式系统结构中的应用程序有数十个依赖关系，每一个依赖关系有些时候不可避免的失败。

**2.**服务雪崩：多个微服务之间的调用的时候，假设微服务A调用微服务B和微服务C。微服务B和微服务C又调用其他的微服务。这就是所谓的“**扇出**”，如果扇出的链路上某个微服务的调用响应时间过长或则不可用，对微服务A的调用就会占用越来越的系统资源。进而引起系统的崩溃。

**3.**对于高流量的应用来说，单一的后端依赖可能会导致所有的服务器上的所有的资源在几秒钟内饱和，比失败更糟糕的是：这些应用程序可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障，这些都表示需要对故障的延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或则系统

**4.**只要是分布式系统，就会产生服务雪崩的问题

## 2.是什么

**1.**Hystrix是一个用于**处理分布式系统的延迟和容错的开源库**，在分布式系统里，许多的依赖不可避免的会调用失败，比如超时，异常等，Hystrix能够保证在一个依赖出问题的情况下，不会导致这个服务的失败，避免级联故障，以提高分布式系统的弹性

**2.**断路器 本身是一种开关装置，当某个服务单元发生故障后，通过断路器的故障监控，**向调用方返回一个符合预期的，可处理的备选响应（FallBack）**，而不是长时间等待或则跑出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间，没有必要的占用。从而避免了故障在分布式系统中的蔓延。乃至雪崩

## 3.服务熔断

**1.**熔断机制是对应雪崩效应的一种微服务链路保护机制

**2.**当扇出链路的某个微服务不可用或则响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回自定义的错误的响应信息，但检测到该节点微服务调用响应正常恢复调用链路，在Springcloud框架里，熔断机制通过Hystrix实现，Hystrix会监控微服务之间的调用，当失败的调用到一定的阈值的时候，缺省是5秒内20次调用失败就会启动熔断机制，熔断机制的注解是：**@HystrixCommand**

## 4创建服务熔断的服务

创建名为microservicecloud-provider-dept-hystrix-8011的服务参照8001服务，套路跟以前的一样

### 1.添加pom.xml:

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

### 2.修改controller类：

@RestController  
@RequestMapping("user")  
public class UserController {  
 @Autowired  
 private UserService service;  
 @RequestMapping(value = "/dept/get/{id}", method = RequestMethod.*GET*)  
 //一旦调用服务方法失败并抛出了错误信息后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中的指定方法  
 **@HystrixCommand(fallbackMethod = "processHystrix\_Get")** public User get(@PathVariable("id") Integer id){  
 User dept = this.service.queryById(id);  
 if (null == dept) {  
 throw new RuntimeException("该ID：" + id + "没有没有对应的信息");  
 }  
 return dept;  
 }  
 public User **processHystrix\_Get**(@PathVariable("id") Integer id){  
 User user = new User();  
 user.setAge(99).setGender(1).setName("服务熔断了").setId(id);  
 return user;  
 }  
}

### 3.主启动类加注解

@SpringBootApplication  
@EnableEurekaClient  
@EnableDiscoveryClient //服务发现  
**@EnableCircuitBreaker//对hystrixR熔断机制的支持**  
public class DeptProvider8011\_App{  
 public static void main(String[] args){  
 SpringApplication.*run*(DeptProvider8011\_App.class, args);  
 }  
}

### 4.测试通过

启动服务的时候，当访问对象为空的时候，抛出异常，被Hystrix捕获。返回定义的信息

## 5.服务降级

### 1.是什么

**1.**整体资源不够了，忍痛将某些服务先关掉，待度过难关，再开启回来

**2.**服务降级处理是在客户端实现完成的，与服务端没有关系

### 2.改造microservicecloud-api工程

根据已经有的DeptClientService接口，新建一个实现了FallbackFactory接口的类DeptClientServiceFallbackFactory：

@Component // 不要忘记添加，不要忘记添加  
public class DeptClientServiceFallbackFactory implements FallbackFactory<DeptClientService>{  
 @Override  
 public DeptClientService create(Throwable throwable){  
 return new DeptClientService() {  
 @Override  
 public User selectOne(Integer id) {  
 return new User().setName("服务降低或则服务异常，导致该服务暂时不可用").setId(id);  
 }  
 };  
 }  
}

DeptClientService.java:

@FeignClient(value = "MICROSERVICECLOUD-DEPT",**fallbackFactory = DeptClientServiceFallbackFactory.class**)  
public interface DeptClientService {  
 @RequestMapping(value = "/user/selectOne/{id}", method = RequestMethod.*GET*)  
 public User selectOne(@PathVariable(value = "id") Integer id) ;  
  
}

### 3.创建服务降级的服务

1.创建名为; microservicecloud-consumer-dept-feign-hystrix-8111.项目参照microservicecloud-consumer-dept-feign-8888来进行改动端口。套路同上。

2.修改application.yml:

**server**:  
 **port**: 8111  
*#开启feign的服务熔断配置***feign:  
 hystrix:  
 enabled: true**  
**eureka**:  
 **client**:  
 **register-with-eureka**: **false  
 service-url**:   
 **defaultZone**: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

3.启动服务进行测试：

测试功能正常

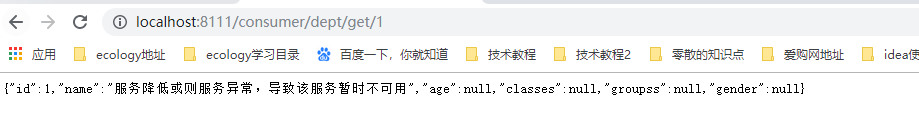
## 6.总结：

**服务熔断：**一般是某一个服务或则异常引起的，类似于现实世界中的保险丝，当某个异常条件被触发，直接熔断整个服务，而不是一直等到此服务超时

**服务降级：**所谓降级，一般是从整体符合来考虑，就是当某一个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallBack回调，返回一个缺省值，这样做，虽然服务水平下降，但好歹可用，比直接挂掉强

**服务降级**这种处理方式是在api的接口类中 ，织入对应的降级返回的信息。降低了**服务熔断**方式代码的耦合，这种方式类似于Spring Aop 面向切面的编程。避免了方法的膨胀。

注意服务降级和服务熔断的区别：前者是：服务不可用的备案方案，后者是服务异常的熔断返回



# 7.服务监控hystrixDashboard

## 1.创建服务

创建名称为：microservicecloud-consumer-hystrix-dashboard-9001的服务。可以拷贝microservicecloud-consumer-dept-feign-hystrix-8111的基础上修改端口。

## 2.修改pom.xml：

添加部分：

<!-- hystrix和 hystrix-dashboard相关 -->  
<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>

## 3.修改application.yml:

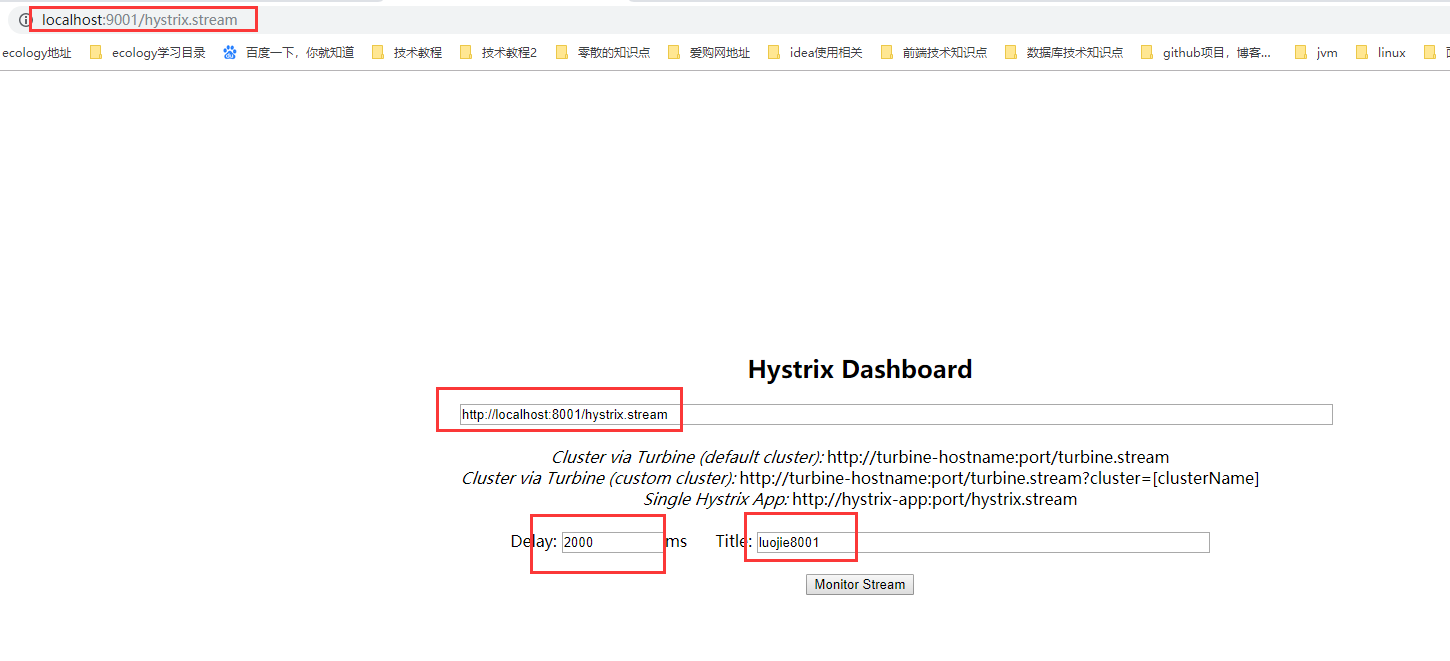
**server**:  
 **port**: 9001

## 4.修改主配置类：

添加注解：

@SpringBootApplication  
@EnableHystrixDashboard  
public class DeptConsumer\_DashBoard\_App  
{  
 public static void main(String[] args)  
 {  
 SpringApplication.*run*(DeptConsumer\_DashBoard\_App.class, args);  
 }  
}

到这里就继承好了，启动访问即可：localhost:9001/hystrix



## 5.服务提供者也需要做对应的修改

以microservicecloud-provider-dept-hystrix-8011为例：

pom.xml中需要保证的组件有：

<!-- hystrix -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-hystrix</artifactId>  
</dependency>  
<!-- actuator监控信息完善 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

主配置类需要保证@EnableCircuitBreaker，对hystrix熔断机制的支持

最后访问效果：



# 8.zuul

## 1.是什么

Zuul包含了请求的**路由**和**过滤**两个最主要的功能：

其中路由功能负责外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础，而过滤功能则负责对请求的处理过程进行干预，是实现请求校验，服务聚合等功能的基础，zuul和Eureka进行整合，将zuul自身注册为Eureka服务治理下的应用。同时从Eureka中获得其他微服务的消息，也即以后的访问微服务都是通过zuul跳转后获得。

注意：zuul服务最终还是会注册进Eureka

**提供=代理+路由+过滤**

## 2.搭建服务

### 1.新建路由模块

创建名称为：microservicecloud-zuul-gateway-9527模块

### 2.pom.xml:

主要保证有：

<!-- zuul路由网关 -->  
<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>  
<!-- actuator监控 -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<!-- hystrix容错 -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-hystrix</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>

3.修改application.yml:

**server**:   
 **port**: 9527  
**spring**:   
 **application**:  
 **name**: microservicecloud-zuul-gateway  
**eureka**:   
 **client**:   
 **service-url**:   
 **defaultZone**: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka,http://eureka7003.com:7003/eureka   
 **instance**:  
 instance-id: gateway-9527.com  
 **prefer-ip-address**: **true  
zuul:**  
 *#ignored-services: microservicecloud-dept* **prefix**: /ecology  
 **ignored-services**: "\*"  
 **routes**:  
 **mydept.serviceId**: microservicecloud-dept  
 **mydept.path**: /mydept/\*\*  
 **mydept1.serviceId**: microservicecloud-dept-hystrix  
 **mydept1.path**: /mydept1/\*\*  
**info**:  
 **app.name**: atguigu-microcloud  
 **company.name**: www.atguigu.com  
 **build.artifactId**: $project.artifactId$  
 **build.version**: $project.version$

说明下：这里zuul需要注册进Eureka中，所以配置跟其他整合进Eureka中配置一致。

### 3.主启动类：

@SpringBootApplication  
**@EnableZuulProxy**  
public class Zuul\_9527\_StartSpringCloudApp{  
 public static void main(String[] args){  
 SpringApplication.*run*(Zuul\_9527\_StartSpringCloudApp.class, args);  
 }  
}

### 4.测试访问：

到目前为止：访问某一个微服务提供者，可以通过两种方式：

1.自己的地址访问：[http://localhost:8001/user//selectOne/1](http://localhost:8001/user/selectOne/1)

2.网关地址：**http://gateway.com:9527**/ **microservicecloud-dept/user/selectOne/1**

**microservicecloud-dept是对应的spring 的name**

**spring**:  
 *#spring应用名称* **application**:  
 **name**: microservicecloud-dept

### 5.路由配置映射规则：

**zuul**:  
 *#ignored-services: microservicecloud-dept* **prefix**: /ecology  
 **ignored-services**: "\*"  
 **routes**:  
 **mydept.serviceId**: microservicecloud-dept  
 **mydept.path**: /mydept/\*\*  
 **mydept1.serviceId**: microservicecloud-dept-hystrix  
 **mydept1.path**: /mydept1/\*\*

说明下：

**Prefix：**代表需要加前缀

**ignored-service：**忽略掉通估微服务名字访问

**mydept.serviceId/ mydept.path：**微服务名：microservicecloud-dept通过/mydept/\*\*来进行映射

### 6.测试成功

到这里上面的配置成功

# 9.Spring Cloud Config分布式配置中心

## 1.是什么

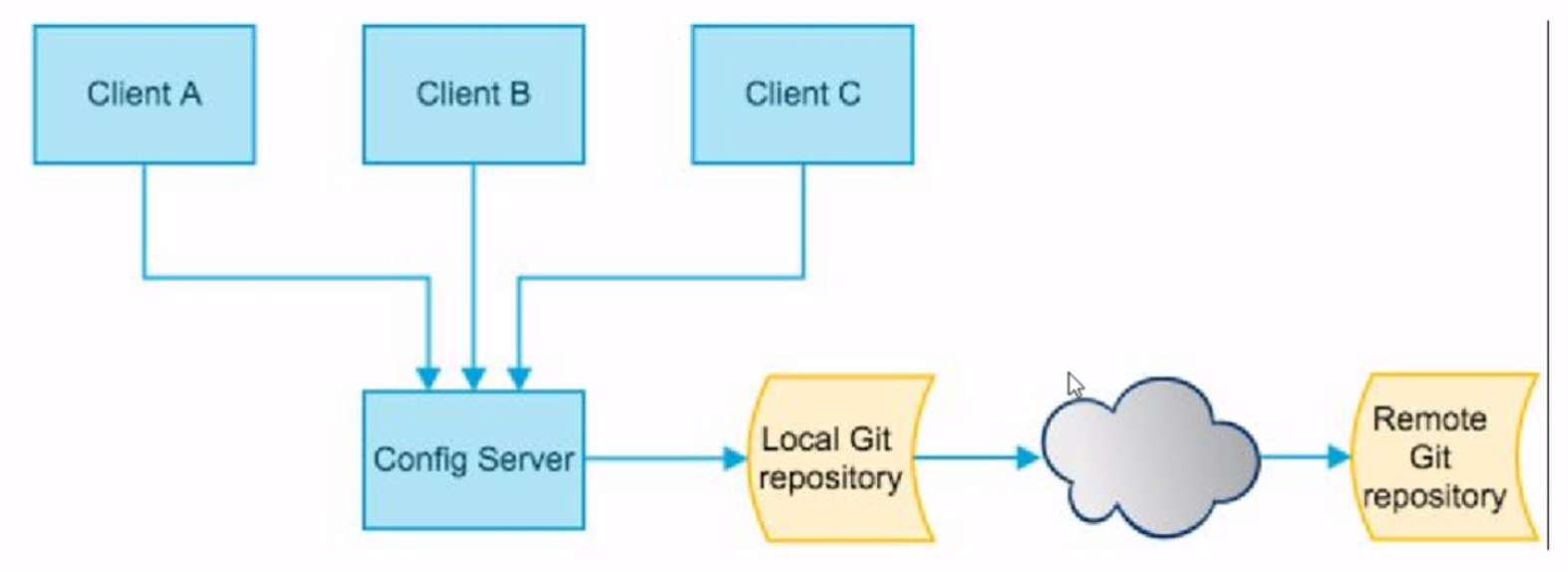
**1.**分布式系统面临的问题：配置问题。每一个微服务都有一个配置文件，管理比较麻烦

**2.**Spring Cloud Config为微服务架构中的微服务提供集中化的外部配置支持。配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置

**3.** Spring Cloud Config分为：**服务端**和**客户端**

**4.**服务端：也称为**分布式配置中心**，它是一个**独立的微服务应用**，用来连接配置服务器并为客户端提供获取配置信息，加密/解密等访问接口

**5.**客户端：通过指定的配置中心来管理应用的资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息，**配置服务器默认采用git来存储配置信息**，这样就有助于对环境进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容



## 2. 能干什么

**1.**集中管理配置文件

**2**.不同环境不同配置，动态化的配置更新，分环境部署，比如：dev/test/prod/beta

**3.**运行期间动态调整配置，不再需要每一个服务部署的机器上编写配置文件，服务会向配置中心统一拉取自己的配置文件

**4.**当配置发生变化时，服务不需要重启即可感知到配置的变化，并应用新的配置

**5.**将配置信息以REST借口的形式暴露

**6.**与GitHub整合：由于Spring Cloud Config默认使用的是Git来存储配置文件（也有其它方式，比如SVN和本地文件），但是**推荐的还是Git**，而且使用的是http/https访问的形式

## 3.配置

### 1.GitHub上创建账号

用自己的GitHub账号在GitHub上新建一个microservicecloud-config的新的Respository

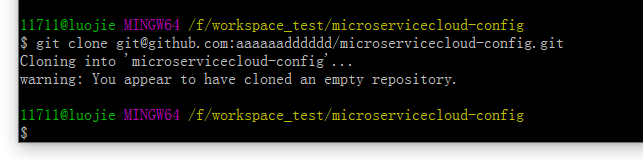
### 2.获取git地址：

[git@github.com:aaaaaadddddd/microservicecloud-config.git](mailto:git@github.com:aaaaaadddddd/microservicecloud-config.git)

### 3.本地建库clone

**本地地址：**F:\workspace\_test\microservicecloud-config

**Git命令：**git clone [git@github.com:aaaaaadddddd/microservicecloud-config.git](mailto:git@github.com:aaaaaadddddd/microservicecloud-config.git)



### 4.新建一个application.yml:

**spring**:  
 **profiles**:  
 **active**:  
 -dev  
---  
**spring**:  
 **profiles**: dev *#开发环境* **application**:  
 **name**: microservicecloud-config-dev  
---  
**spring**:  
 **profiles**: dev *#测试环境* **application**:  
 **name**: microservicecloud-config-test

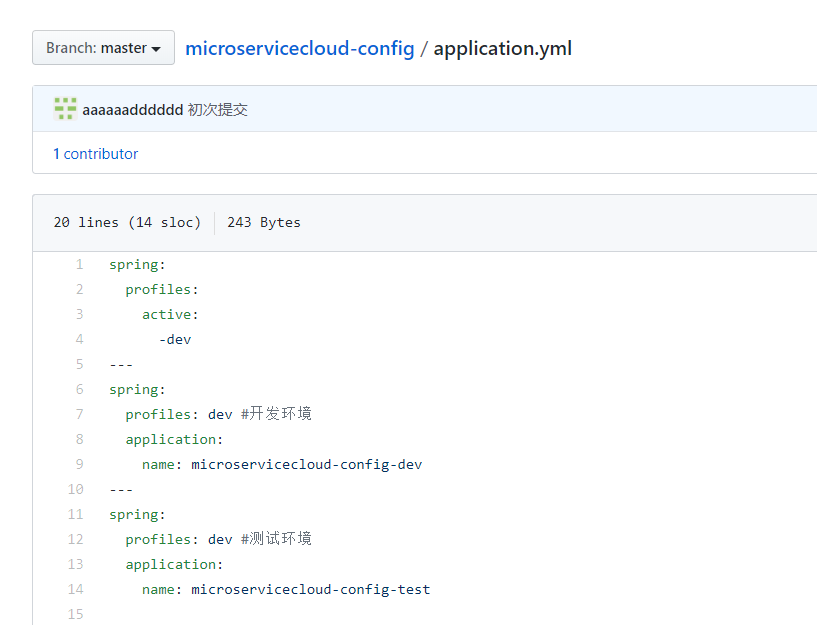
### 5.提交到gitHub上：

git status

git add .

git commit -m "初次提交"

git push origin master



## 4.创建Config服务：

### 1.创建服务：

创建一个名为：microservicecloud-config-3344。

### 2.配置pom.xml:

<dependencies>  
 <!-- springCloud Config -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-**server**</artifactId>  
 </dependency>  
 </dependencies>

这是主要的maven模块

### 3.配置application,yml:

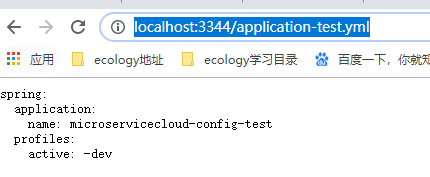
**server**:   
 **port**: 3344   
**spring**:  
 **application**:  
 **name**: microservicecloud-config  
 **cloud**:  
 **config**:  
 **server**:  
 **git**:  
 **uri**: https://github.com/aaaaaadddddd/microservicecloud-config.git *#GitHub上面的git仓库名字* **skipSslValidation**: true

### 4.配置主启动类：

@SpringBootApplication  
**@EnableConfigServer**  
public class Config\_3344\_StartSpringCloudApp{  
 public static void main(String[] args){  
 SpringApplication.*run*(Config\_3344\_StartSpringCloudApp.class, args);  
 }  
}

### 5.测试通过：

访问：<http://localhost:3344/application-test.yml>或则<http://localhost:3344/application-dev.yml>



## 5.客户端的配置服务

### 1.本地新建microservicecloud-config-client.yml

**spring**:  
 **profiles**:  
 **active**:  
 -dev  
---  
**server**:  
 **port**: 8201  
**spring**:  
 **profiles**: dev *#开发环境* **application**:  
 **name**: microservicecloud-config-dev  
**eureka**:  
 **client**: *#客户端注册进eureka服务列表内* **service-url**:  
 *#defaultZone: http://localhost:7001/eureka* **defaultZone**: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/  
 **instance**:  
 **instance-id**: microservicecloud-config-dev-8201  
 **prefer-ip-address**: true *#访问路径可以显示IP地址*---  
**server**:  
 **port**: 8202  
**spring**:  
 **profiles**: test *#测试环境* **application**:  
 **name**: microservicecloud-config-test  
**eureka**:  
 **client**: *#客户端注册进eureka服务列表内* **service-url**:  
 *#defaultZone: http://localhost:7001/eureka* **defaultZone**: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/  
 **instance**:  
 **instance-id**: microservicecloud-config-test-8202  
 **prefer-ip-address**: true *#访问路径可以显示IP地址*

### 2.推送到GitHub上

### 3.新建客户端服务

#### 1.创建一个名称为：

microservicecloud-config-client-3355。

#### 2.pom.xml:

<!-- SpringCloud Config客户端 -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>

#### 3.新建application.yml和bootstrap.xml

application.yml:

**spring**:  
 **application**:  
 **name**: microservicecloud-config-client

bootstrap.xml:（bootstrap它的优先权重很高）

**spring**:  
 **cloud**:  
 **config**:  
 **name**: microservicecloud-config-client *#需要从github上读取的资源名称，注意没有yml后缀名* **profile**: **dev** *#本次访问的配置项* **label**: master   
 **uri**: http://config-3344.com:3344 *#本微服务启动后先去找3344号服务，通过SpringCloudConfig获取GitHub的服务地址*

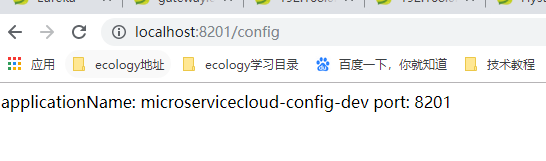
#### 4.添加测试的controller：

@RestController  
public class ConfigClientRest{  
 @Value("${spring.application.name}")  
 private String applicationName;  
 @Value("${server.port}")  
 private String port;  
 @RequestMapping("/config")  
 public String getConfig()  
 {  
 String str = "applicationName: " + applicationName + "\t port: " + port;  
 System.*out*.println("\*\*\*\*\*\*str: " + str);  
 return "applicationName: " + applicationName + "\t port: " + port;  
 }  
}

5.主启动类：

@SpringBootApplication  
public class ConfigClient\_3355\_StartSpringCloudApp{  
 public static void main(String[] args){  
 SpringApplication.*run*(ConfigClient\_3355\_StartSpringCloudApp.class, args);  
 }  
}

6.测试访问：



**到这里配置Spring Cloud的学习到此结束，后期还会有第二季的学习，再接再厉，做好笔记，温故而知新**