**SpringCloud微服务**

# 一、微服务与微服务架构：

1、微服务：

强调的是服务的大小，它关注的是某一个点，是**具体解决某一个问题/提供落地对应服务的一个服务应用**,狭意的看,可以看作Eclipse里面的一个个微服务工程/或者Module。

2、微服务架构：

微服务架构是⼀种**架构模式**，它提倡将单⼀应⽤程序划分成⼀组⼩的服务，服务之间互相协调、互相配合，为⽤户提供最终价值。**每个服务运⾏在其独⽴的进程中**，服务与服务间采⽤轻量级的通信机制互相协作（**通常是基于HTTP协议的RESTful API**）。每个服务都围绕着具体业务进⾏构建，并且能够被独⽴的部署到⽣产环境、类⽣产环境等。另外，**应当尽量避免统⼀的、集中式的服务管理机制**，对具体的⼀个服务⽽⾔，应根据业务上下⽂，选择合适的语⾔、⼯具对其进⾏构建。

3、微服务的技术栈：

**微服务条目** **落地技术**

服务开发 Springboot、Spring、SpringMVC

服务配置与管理 Netflix公司的Archaius、阿里的Diamond等

服务注册与发现 Eureka、Consul、Zookeeper等

服务调用 Rest、RPC、gRPC

服务熔断器 Hystrix、Envoy等

负载均衡 Ribbon、Nginx等

服务接口调用(客户端调用服务的简化工具) Feign等

消息队列 Kafka、RabbitMQ、ActiveMQ等

服务配置中心管理 SpringCloudConfig、Chef等

服务路由（API网关） Zuul等

服务监控 Zabbix、Nagios、Metrics、Spectator等

全链路追踪 Zipkin，Brave、Dapper等

服务部署 Docker、OpenStack、Kubernetes等

数据流操作开发包 SpringCloud Stream（封装与Redis,Rabbit、Kafka等发送接收消息）

事件消息总线 Spring Cloud Bus

4、SpringCloud：

基于SpringBoot提供了一套微服务解决方案，包括服务注册与发现，配置中心，全链路监控，服务网关，负载均衡，熔断器等组件，除了基于NetFlix的开源组件做高度抽象封装之外，还有一些选型中立的开源组件。

SpringCloud利用SpringBoot的开发便利性巧妙地简化了分布式系统基础设施的开发，SpringCloud为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、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,它们都可以用SpringBoot的开发风格做到一键启动和部署。

SpringBoot并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过SpringBoot风格进行再封装屏蔽掉了复杂的配置和实现原理，**最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包**。

5、SpringBoot与SpringCloud的关系：

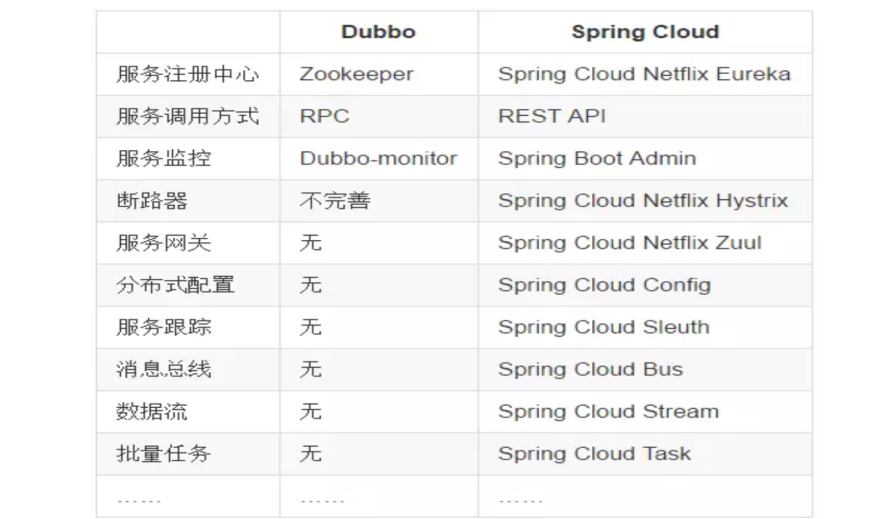
**SpringBoot**专注于快速方便的开发单个个体微服务。

**SpringCloud**是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微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。

SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于依赖的关系。

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

6、SpringCloud与Dubbo:



**最大区别**：**SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。**

严格来说，这两种方式各有优劣。虽然从一定程度上来说，**后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题**。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

**品牌机与组装机的区别：**

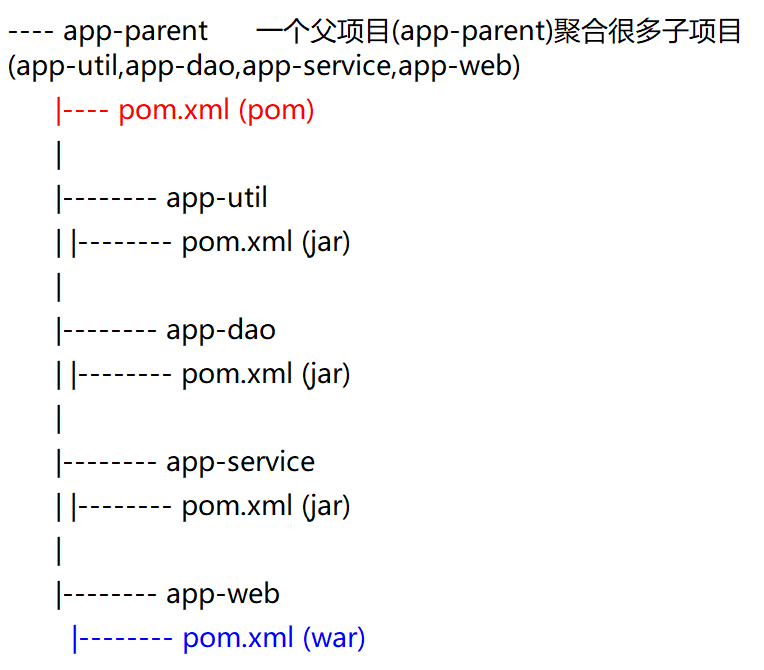
很明显，**Spring Cloud的功能比DUBBO更加强大**，涵盖面更广，而且作为Spring的拳头项目，它也能够与Spring Framework、Spring Boot、Spring Data、Spring Batch等其他Spring项目完美融合，这些对于微服务而言是至关重要的。使用Dubbo构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题；而Spring Cloud就像品牌机，在Spring Source的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础有足够的了解。

**社区支持与更新力度：**

最为重要的是，DUBBO停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX），这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。

# 二、REST微服务构建工程模块：

1、一个简单的Maven模块结构（多模块开发）：



2、RestTemplate是什么：

RestTemplate提供了多种便捷访问远程Http服务的方法， 是一种简单便捷的访问restful服务模板类，**是Spring提供的用于访问Rest服务的客户端模板工具集**。

# 三、Eureka服务的注册于发现：

1、Eureka是什么：

Eureka是Netflix的一个子模块，也是核心模块之一。**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**。

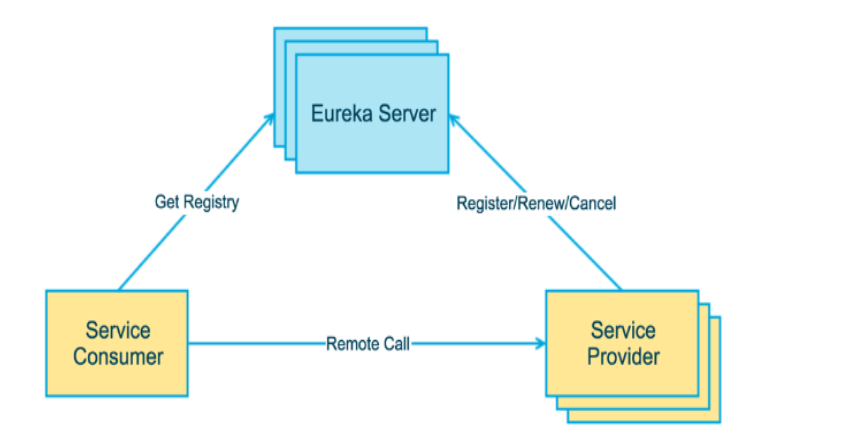
服务注册与发现对于微服务架构来说是非常重要的，有了服务发现与注册，只需要**使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件**了。功能类似于dubbo的注册中心，比如Zookeeper。

2、原理讲解：

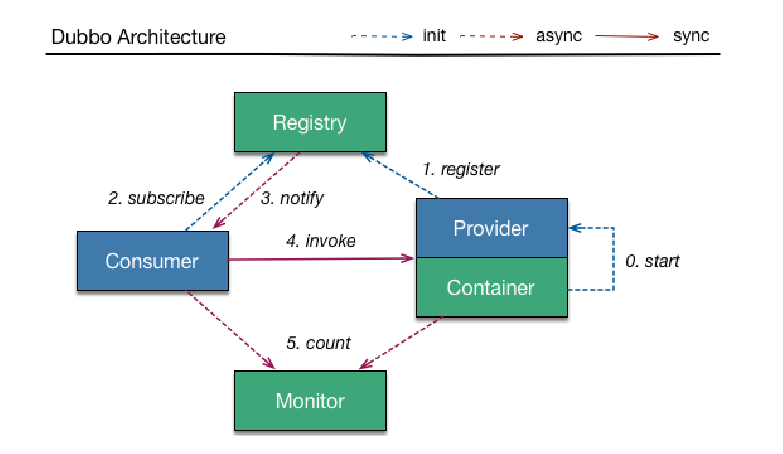
Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现(**请对比Zookeeper**)。

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**。

而系统中的其他微服务，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。



**Dubbo的架构：**



**Eureka包含两个组件（Eureka Server和Eureka Client）：**

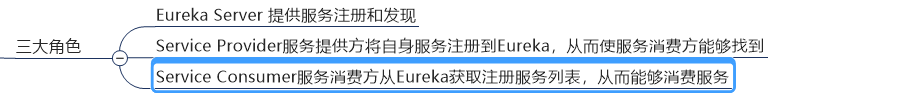
**Eureka Server提供服务注册服务：**

**各个节点启动后，会在EurekaServer中进行注册**，这样EurekaServer中的服务注册表中将会存储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。

**EurekaClientJava客户端：**

EurekaClient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(round-robin)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某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除（默认90秒）

**三大角色：**



3、构建步骤：

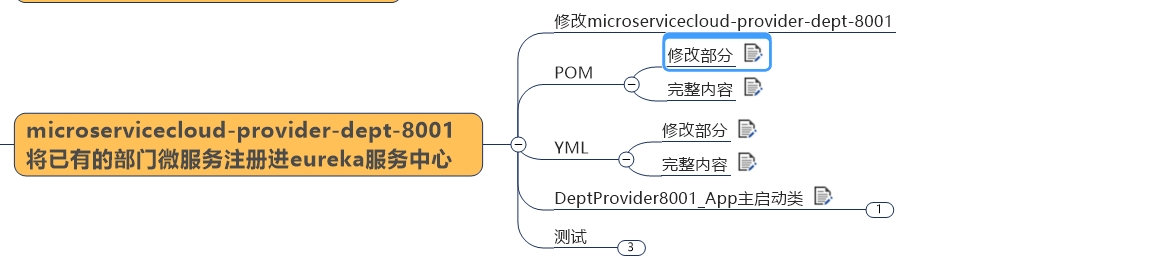
**①、新建一个Eureka服务端（供微服务注册）：**



**yml文件：**

|  |
| --- |
| **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:**  **defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #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。** |

**②、将生产者注入到Eureka服务中：**



**pom文件:**

|  |
| --- |
| **<!-- 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>** |

**yml文件（客户端注册进eureka服务列表内）：**

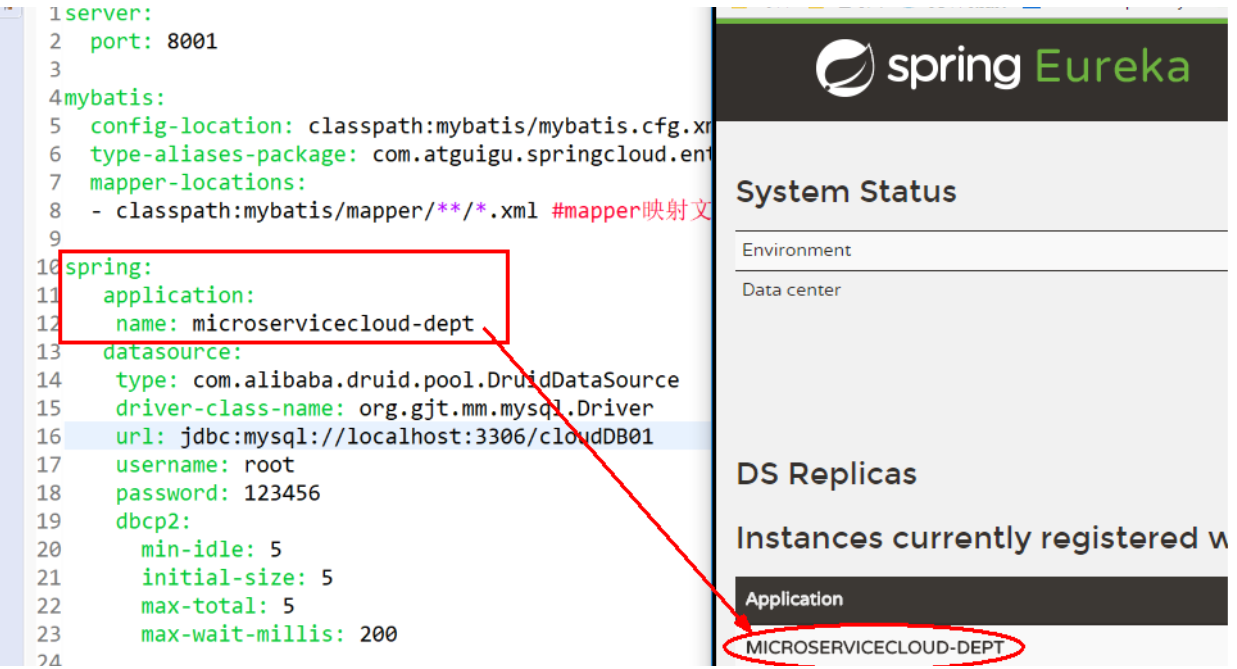
|  |
| --- |
| **eureka:**  **client: #客户端注册进eureka服务列表内**  **service-url:**  **defaultZone: http://localhost:7001/eureka** |

**启动类（添加启动讲服务注册进Eureka）：**



**③、启动服务类和生产者：**

**浏览器输入：localhost:7001,结果如下，表示注入成功：**



4、Eureka自我保护模式：



默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生时，微服务与EurekaServer之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护模式”来解决这个问题——**当EurekaServer节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式**。一旦进入该模式，EurekaServer就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该Eureka Server节点会自动退出自我保护模式。

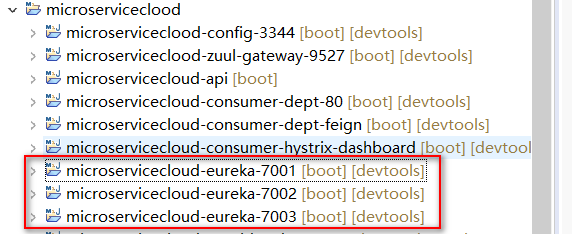
在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。当它收到的心跳数重新恢复到阈值以上时，该Eureka Server节点就会自动退出自我保护模式。它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话讲解：好死不如赖活着。

综上，**自我保护模式是一种应对网络异常的安全保护措施**。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

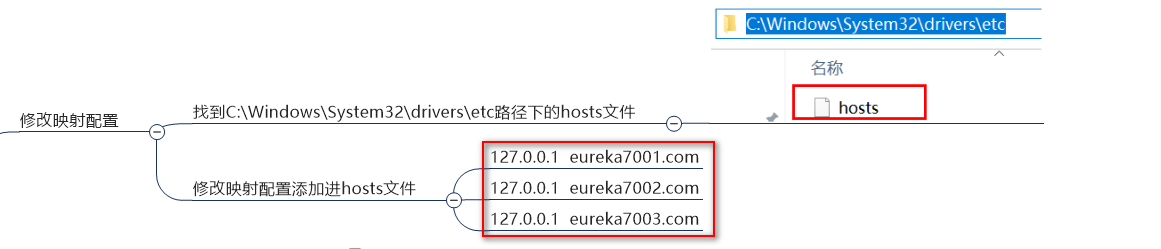
在Spring Cloud中，可以使用**eureka.server.enable-self-preservation = false 禁用自我保护模式**。

5、集群配置：

**①、再复制两个Eureka服务端：**

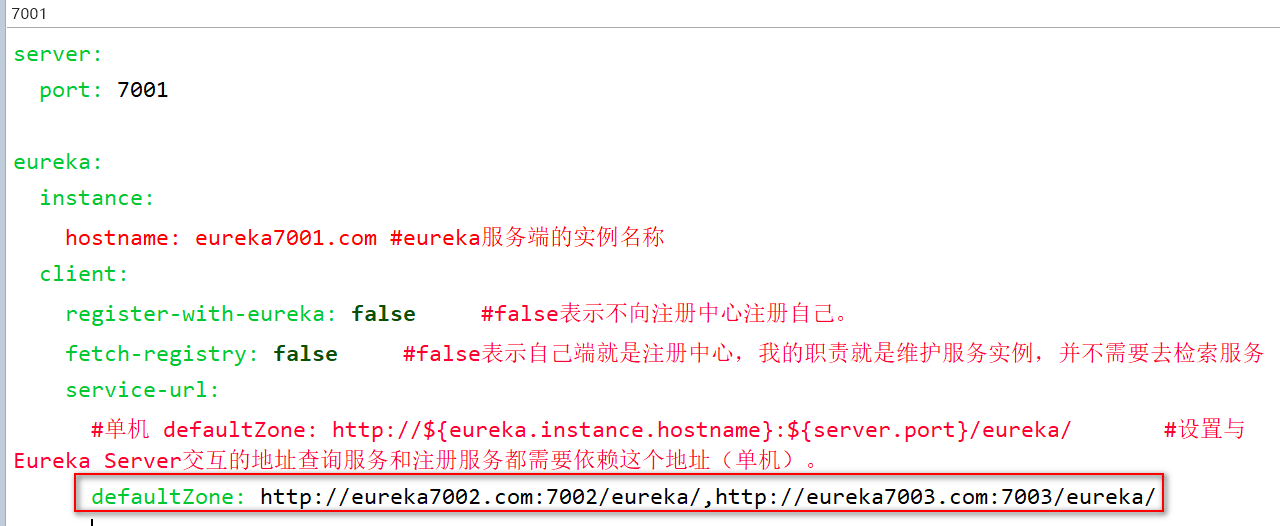


**②、修改host文件：**



**③、配置三台Eureka服务的yml文件：**

**Eureka7001：**



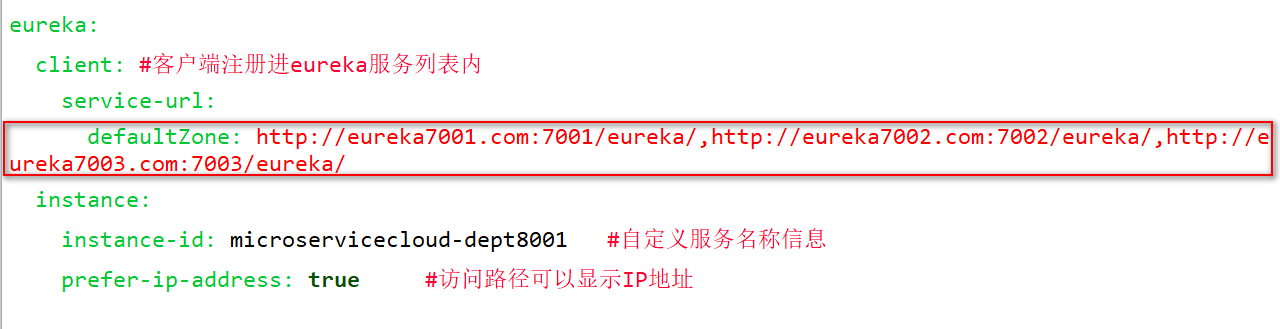
**Eureka7002：**



**Eureka7003：**



**④、将生产者注入三个Eureka服务中（生产者yml文件）：**



# 四、Ribbon负载均衡（集中在消费方）:

1、Ribbon是什么：

**Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端负载均衡的工具。**

简单的说，Ribbon是Netflix发布的开源项目，主要功能是提供**客户端的软件负载均衡算法**，将Netflix的中间层服务连接在一起。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。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（如简单轮询，随机连接等）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。

2、负载均衡（LB）：

LB，即负载均衡(Load Balance)，在微服务或分布式集群中经常用的一种应用。

负载均衡简单的说就是**将用户的请求平摊的分配到多个服务上，从而达到系统的HA**。

常见的负载均衡有软件Nginx，LVS，硬件 F5等。

相应的在中间件，例如：dubbo和SpringCloud中均给我们提供了负载均衡，**SpringCloud的负载均衡算法可以自定义**。

1. **、集中式LB：**

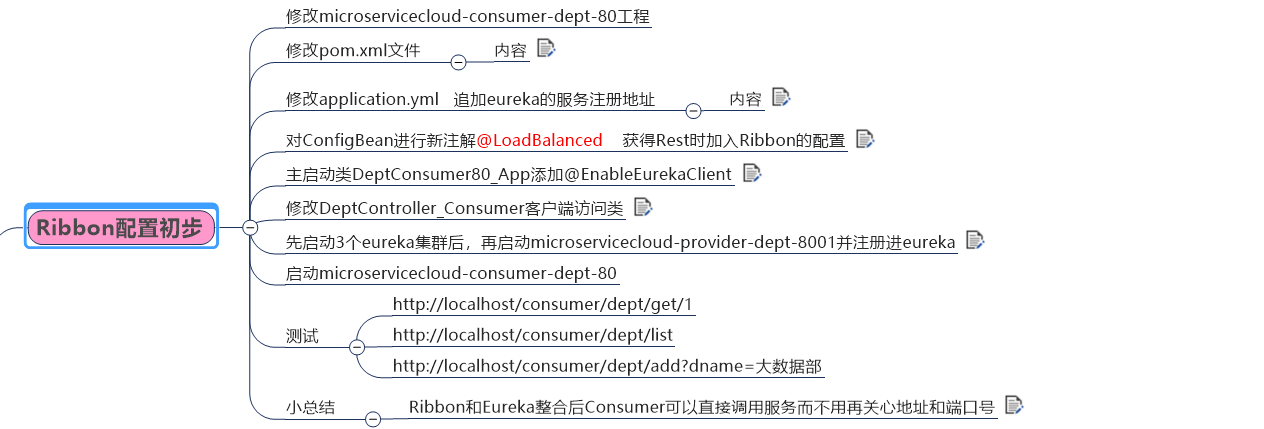
**即在服务的消费方和提供方之间使用独立的LB设施**(可以是硬件，如F5, 也可以是软件，如nginx), 由该设施负责把访问请求通过某种策略转发至服务的提供方；

1. **进程内LB：**

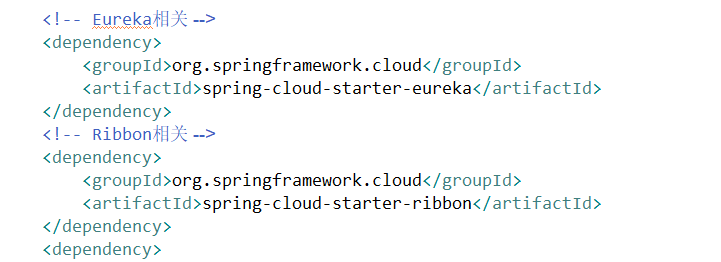
**将LB逻辑集成到消费方**（如点餐，选人少的队伍排队），消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

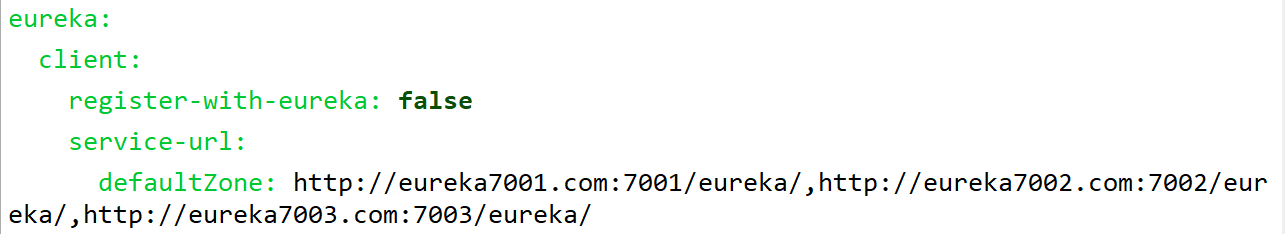
3、Ribbon配置初步：



1. **、消费端添加Ribbon依赖：**



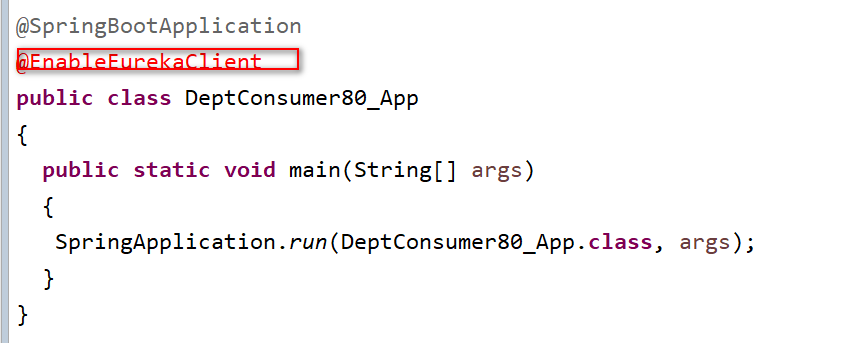
1. **、yml文件中配置Eureka相关信息：**



1. **、开启Ribbon负载均衡：**



1. **、启动类添加Eureka客户端：**



1. **、修改消费端controller调用的地址：**



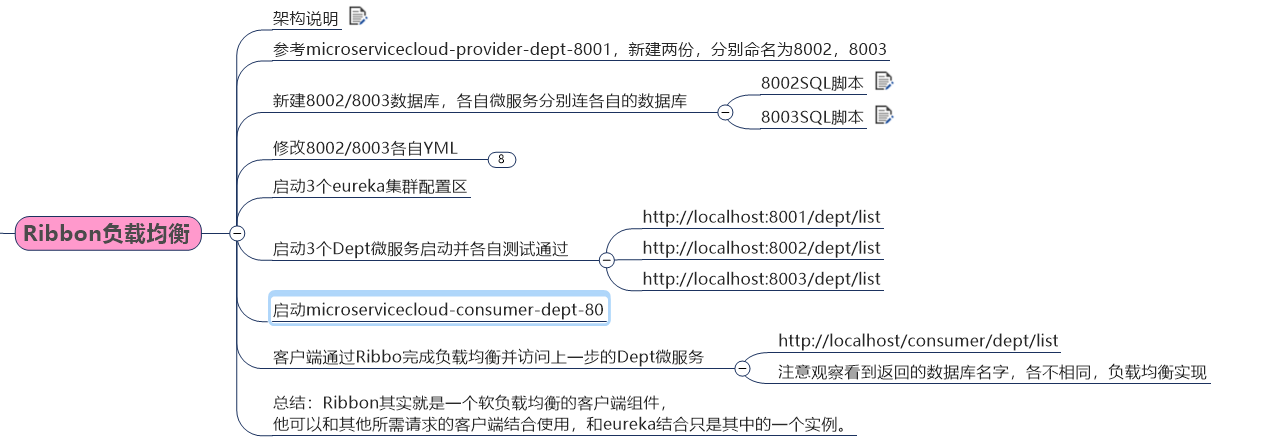
**调用地址为注册进Eureka的微服务名：**



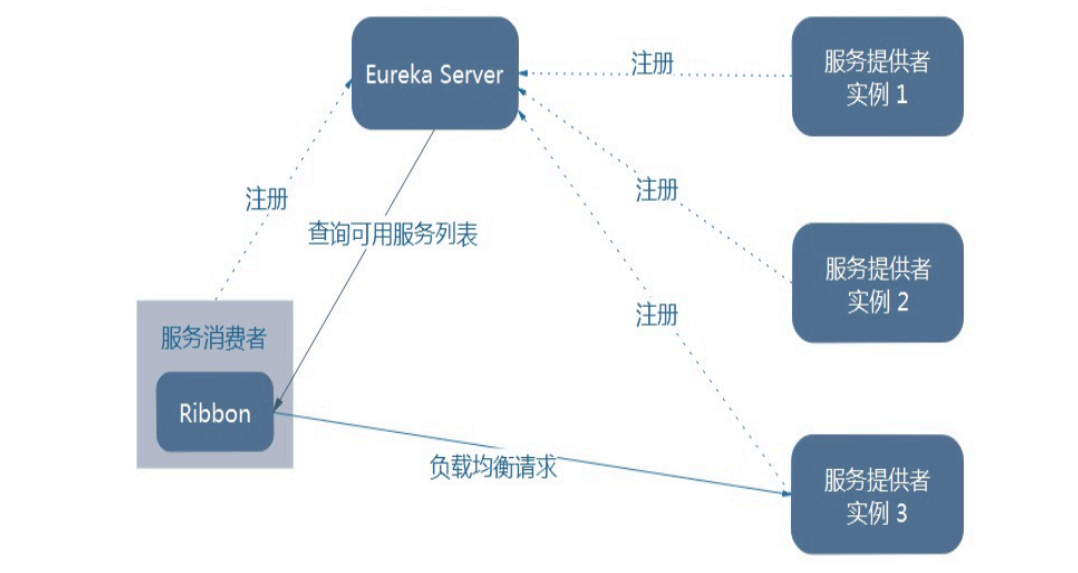
1. **、测试：**

分别启动三个Eureka服务，在启动生产者，最后启动消费者。访问消费者的controller。可以得到生产者的信息。

4、Ribbon负载均衡：



**①、架构：**



**Ribbon在工作时分成两步:**

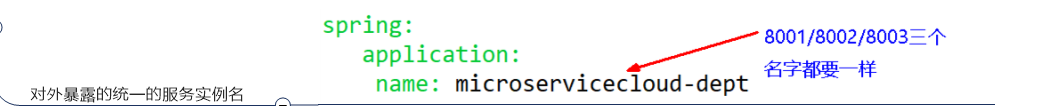
**第一步先选择 EurekaServer ,它优先选择在同一个区域内负载较少的server.**

**第二步再根据用户指定的策略，在从server取到的服务注册列表中选择一个地址。**

**其中Ribbon提供了多种策略：比如轮询、随机和根据响应时间加权。**

1. **、新建生产者2和生产者3：**

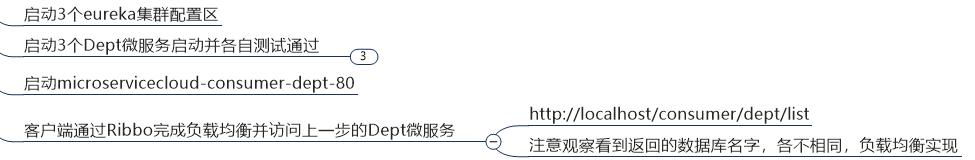
**参考生产者1，配置各自的数据库，yml文件，pom文件等。其中yml文件中对外暴露的微服务名称必须相同。**



1. **、更改负载均衡算法：**



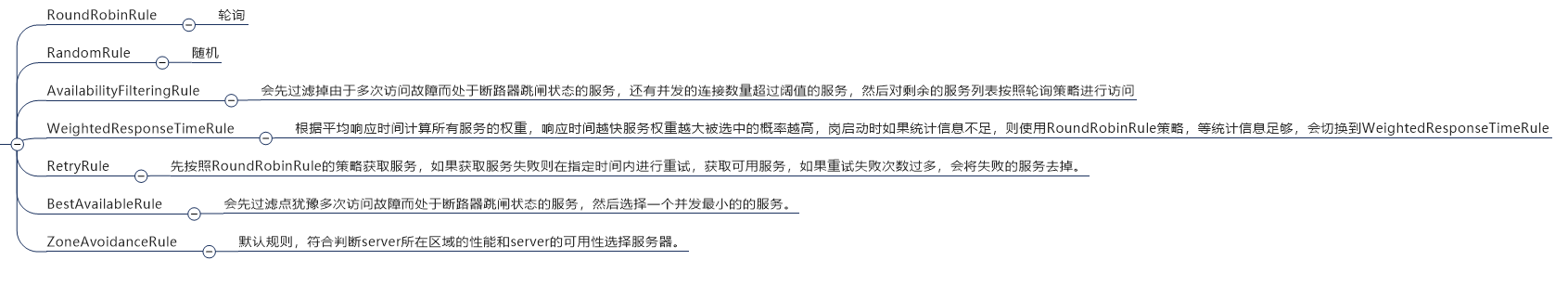
1. **、测试：**



5、Ribbon的核心算法-IRULE：



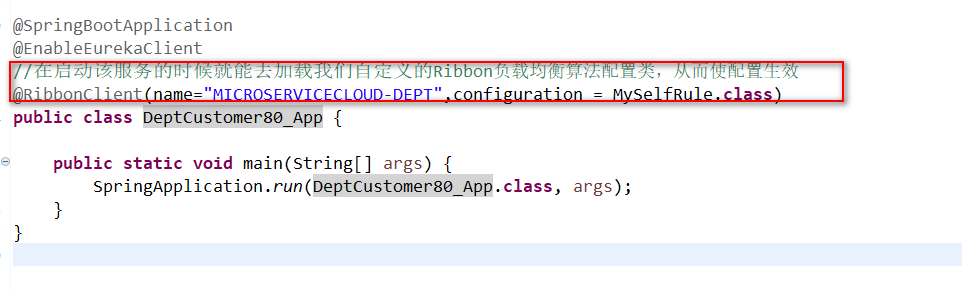
**IRULE算法的种类：**



6、自定义Ribbon算法：



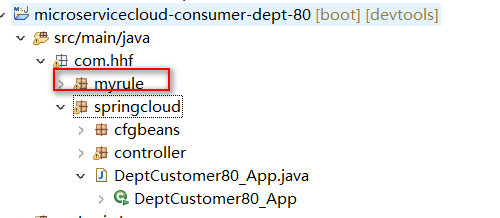
1. **、在住启动类上添加自定义算法相关配置：**



1. **配置注意事项：**



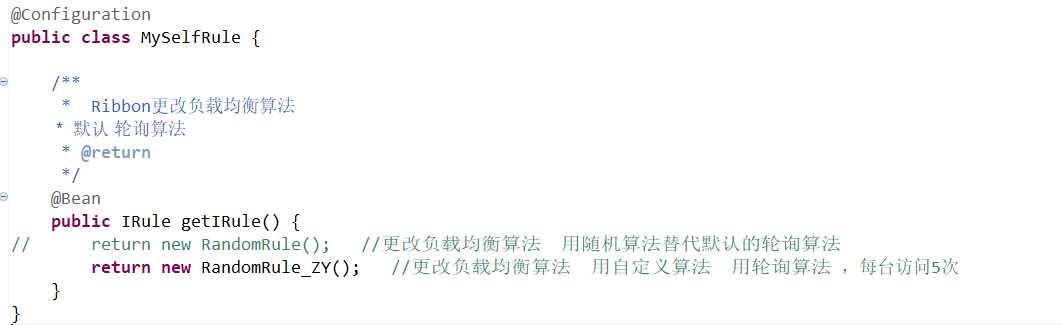
**我的项目结构如下：**



1. **自定义算法配置类：**



**相关配置：**



|  |
| --- |
| **import** java.util.List;  **import** com.netflix.client.config.IClientConfig;  **import** com.netflix.loadbalancer.AbstractLoadBalancerRule;  **import** com.netflix.loadbalancer.ILoadBalancer;  **import** com.netflix.loadbalancer.Server;  /\*\*  \* 自定义负载均衡算法 轮询算法，每台分别使用5次  \* **@author** hhf1311843248  \*  \*/  **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  }  } |

# 六、Feign负载均衡：

1、Feign是什么：

**官网解释：**

http://projects.spring.io/spring-cloud/spring-cloud.html#spring-cloud-feign

**Feign是一个声明式WebService客户端**。使用Feign能让编写Web Service客户端更加简单, **它的使用方法是定义一个接口，然后在上面添加注解，同时也支持JAX-RS标准的注解**。Feign也支持可拔插式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和HttpMessageConverters。**Feign可以与Eureka和Ribbon组合使用以支持负载均衡**。



2、Feign能干什么：

**Feign旨在使编写Java Http客户端变得更容易**。

前面在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模版化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每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。所以，**Feign在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义**。**在Feign的实现下，我们只需创建一个接口并使用注解的方式来配置它(以前是Dao接口上面标注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，自动封装服务调用客户端的开发量**。

3、Feign集成了Ribbon：

利用Ribbon维护了MicroServiceCloud-Dept的服务列表信息，并且通过轮询实现了客户端的负载均衡。而**与Ribbon不同的是，通过feign只需要定义服务绑定接口且以声明式的方法，优雅而简单的实现了服务调用。**

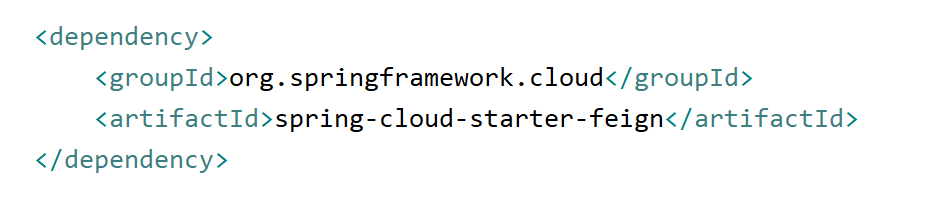
4、Feign的使用步骤：



1. **、参考80项目新建一个feigin消费端项目：**

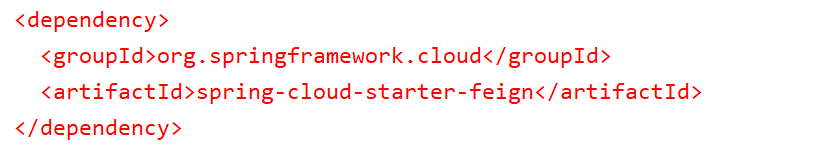


1. **、pom文件添加feign的依赖：**



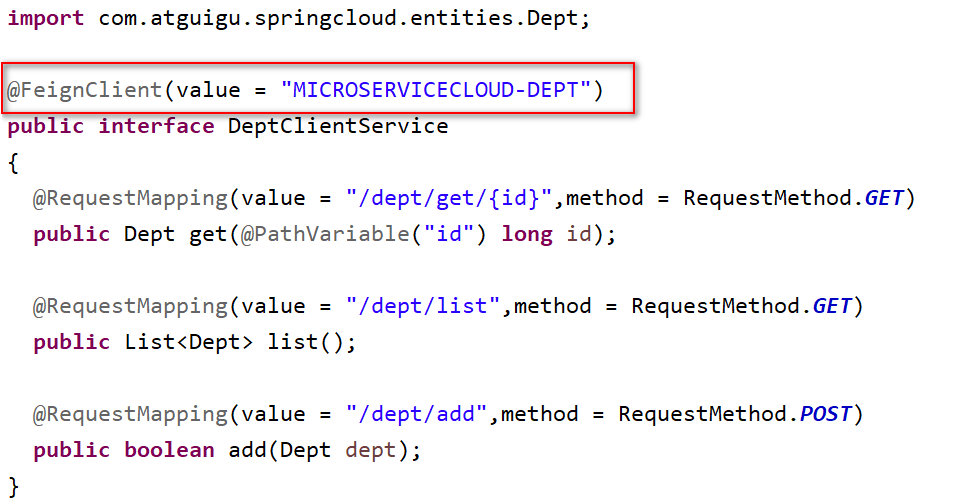
1. **、修改公共的api服务：**

**pom.xml添加依赖：**



**新建DeptClientService接口：**

**此接口作为服务端提供客户端调用接口，之前一直使用的是resttemplate调用，现在直接使用该接口即可访问。**



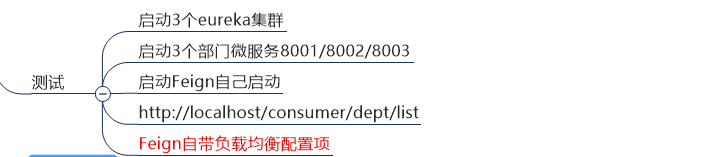
1. **、feign项目添加controller（调用api新建的feign接口）：**



1. **、启动类添加启动feign注解：**



1. **、测试访问：**



5、小结：

**Feign通过接口的方法调用Rest服务（之前是Ribbon+RestTemplate），**

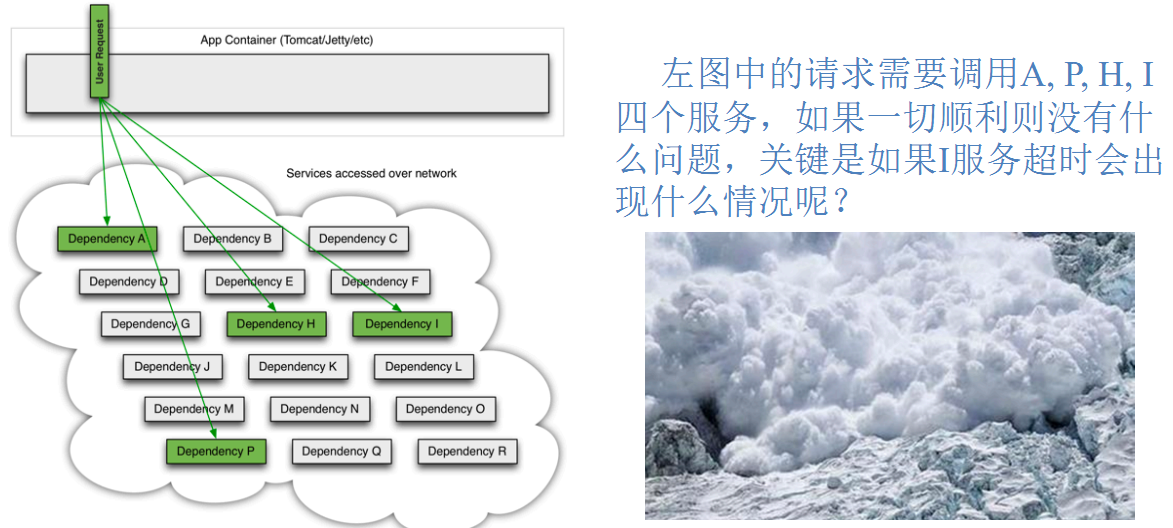
**该请求发送给Eureka服务器（http://MICROSERVICECLOUD-DEPT/dept/list）,**

**通过Feign直接找到服务接口，由于在进行服务调用的时候融合了Ribbon技术，所以也支持负载均衡作用。**

# 七、Histrix熔断器：

1、 分布式系统面临的问题：

**复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免地失败。**



**服务雪崩：**

**多个微服务之间调用的时候，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又调用其它的微服务，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者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”.**

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更糟糕的是，这些应用程序还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**对故障和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系统。**

2、 Hystrix熔断器是什么：

**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，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性**。

“断路器”本身是一种开关装置，**当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛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、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩**。

3、 能干什么：



1. 服务熔断：

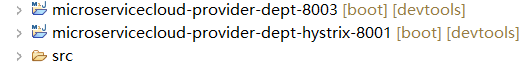
**①、是什么：**

**熔断机制是应对雪崩效应的一种微服务链路保护机制。**

**当扇出链路的某个微服务不可用或者响应时间太长时，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"错误"的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，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。**

**②、使用服务熔断的步骤：**

**a、参照dept项目新建一个新的生产者项目:**



**b、pom文件添加hystrix依赖：**



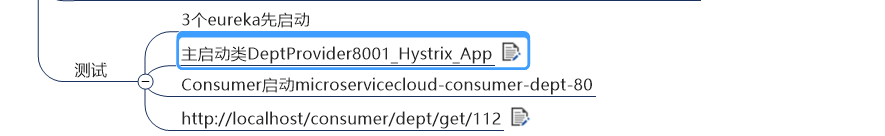
**c、修改controller:**

**一旦调用服务方法失败并抛出错误信息后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中指定的方法。**

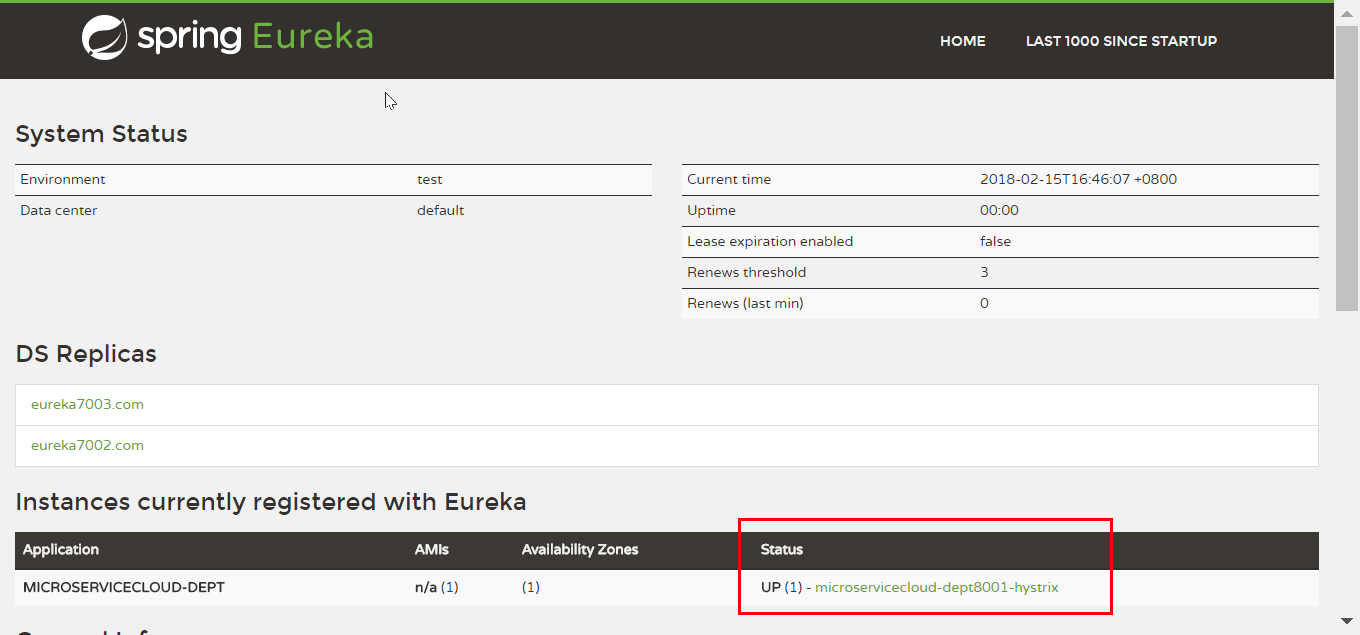
**d、启动类添加对Hystrix的支持：**



**e、测试：**



**Eureka成功注册了Hystrix-dept的服务：**



**服务端出现异常后调用fallbackMethod调用类中的方法。**

1. 服务降级：



**①、是什么：**

整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来。

**服务降级是在客户端完成，与服务端无关。**

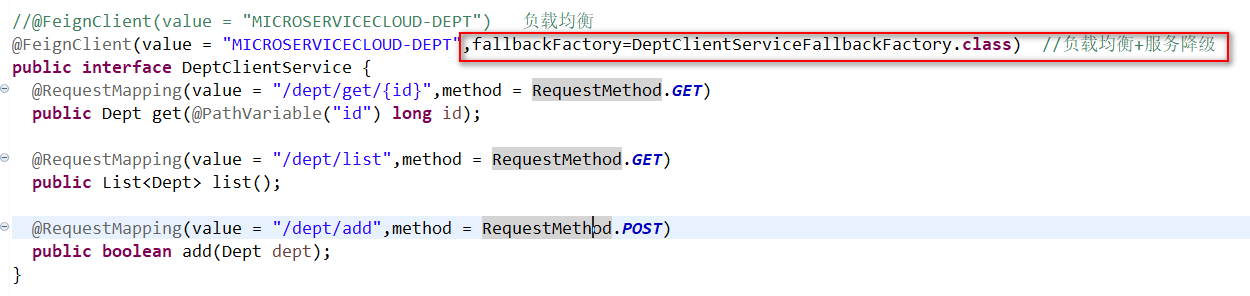
**②、使用服务降级的步骤：**

**a、在api公共服务中添加DeptClientServiceFallbackFactory类：**

|  |
| --- |
| package com.atguigu.springcloud.service;  import java.util.List;  import org.springframework.stereotype.Component;  import com.atguigu.springcloud.entities.Dept;  import feign.hystrix.FallbackFactory;  **@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 Dept get(long id)  {  return new Dept().setDeptno(id)  .setDname("该ID："+id+"没有没有对应的信息,Consumer客户端提供的降级信息,此刻服务Provider已经关闭")  .setDb\_source("no this database in MySQL");  }  @Override  public List<Dept> list()  {  return null;  }  @Override  public boolean add(Dept dept)  {  return false;  }  };  }  } |

**主要是对一些服务调用失败后统一做处理。**

**b、修改DeptClientService（添加fallbackFactory属性）:**



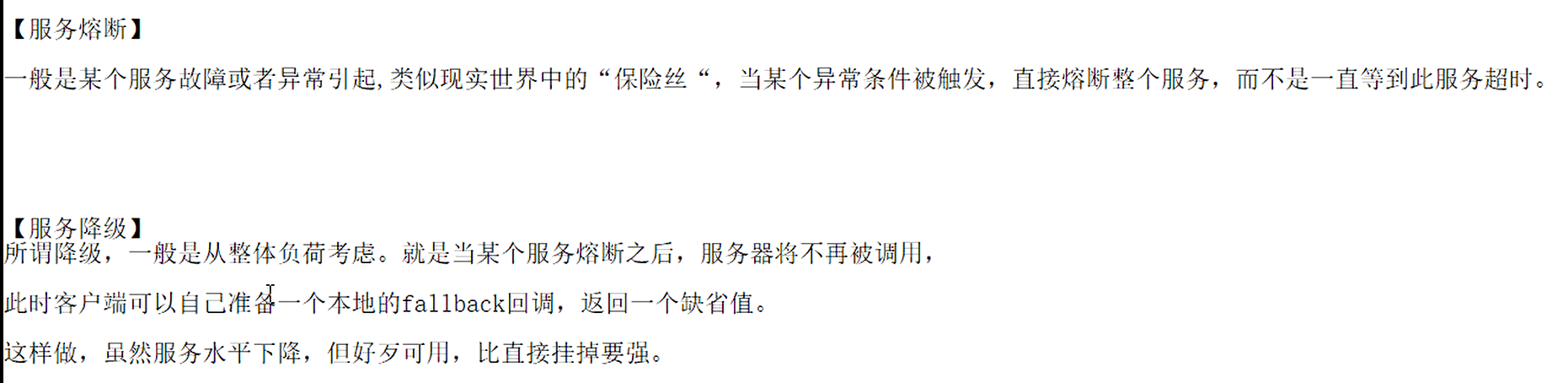
**c、消费端yml文件修改：**



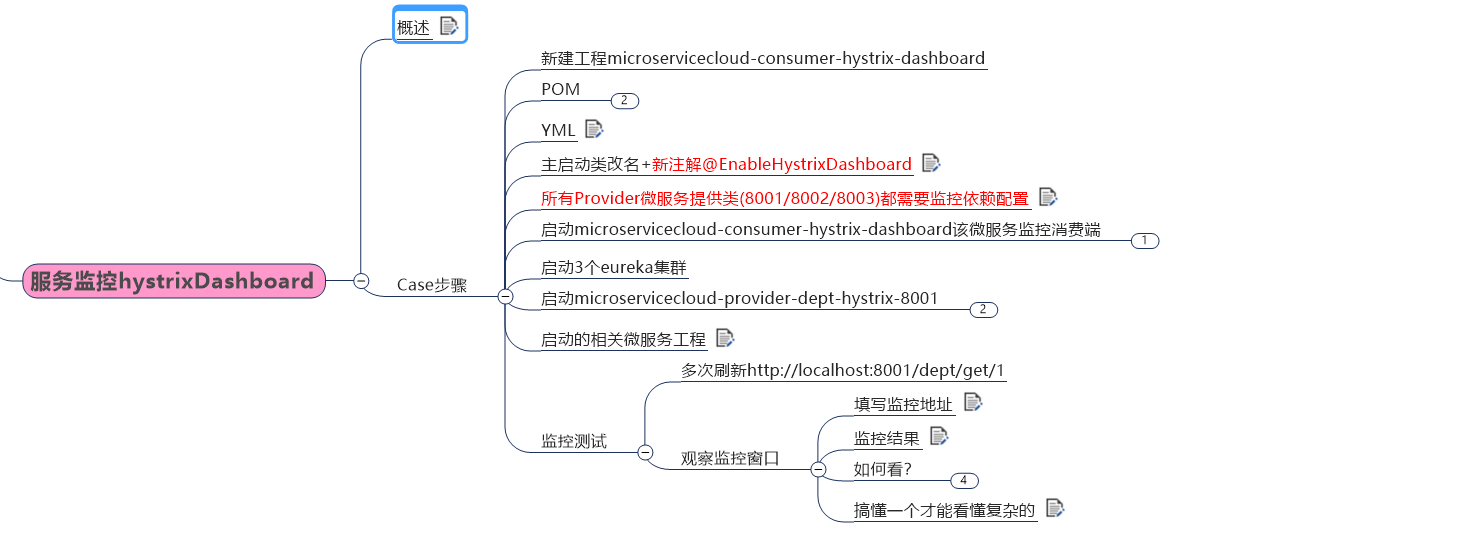
**d、测试：**



1. 服务熔断和服务降级含义：



1. 服务监控hystrixDashboard:

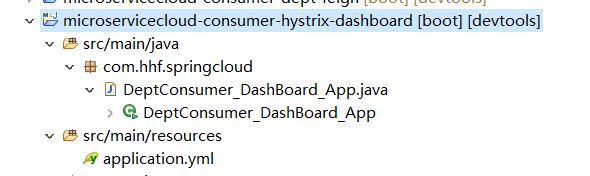


**①、概述：**

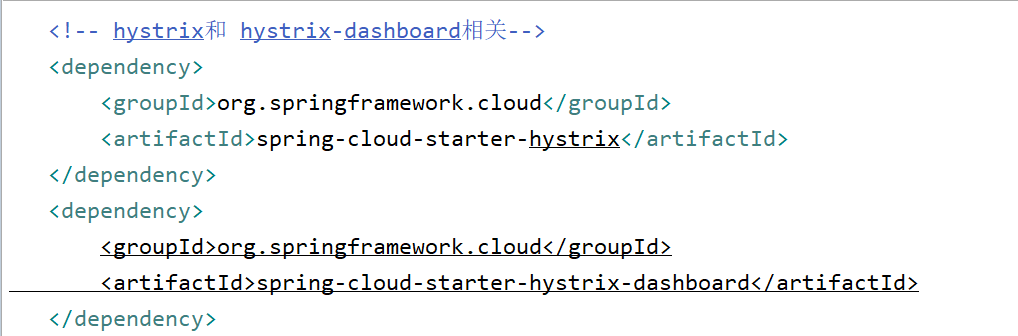
除了隔离依赖服务的调用以外，Hystrix还提供了**准实时的调用监控（Hystrix Dashboard）**，Hystrix会持续地记录所有通过Hystrix发起的请求的执行信息，**并以统计报表和图形的形式展示给用户**，包括每秒执行多少请求多少成功，多少失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。Spring Cloud也提供了Hystrix Dashboard的整合，对监控内容转化成可视化界面。

**②、实现步骤：**

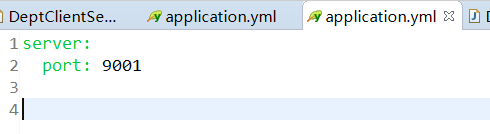
**a、新建项目microservicecloud-consumer-hystrix-dashboard：**



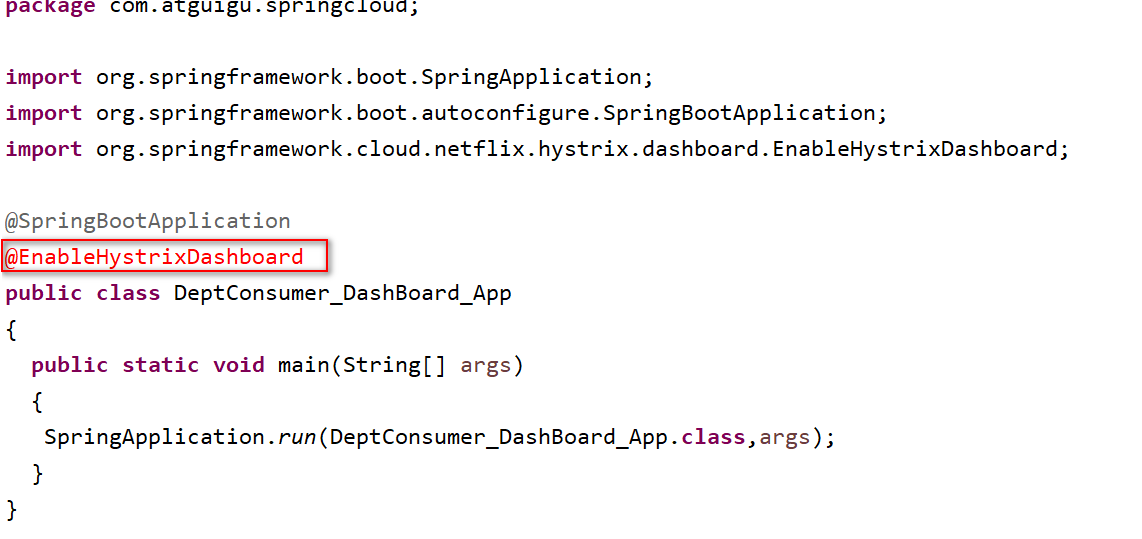
**b、pom文件添加依赖：**



**c、配置yml文件：**

****

**d、启动类添加开启仪表盘注解：**

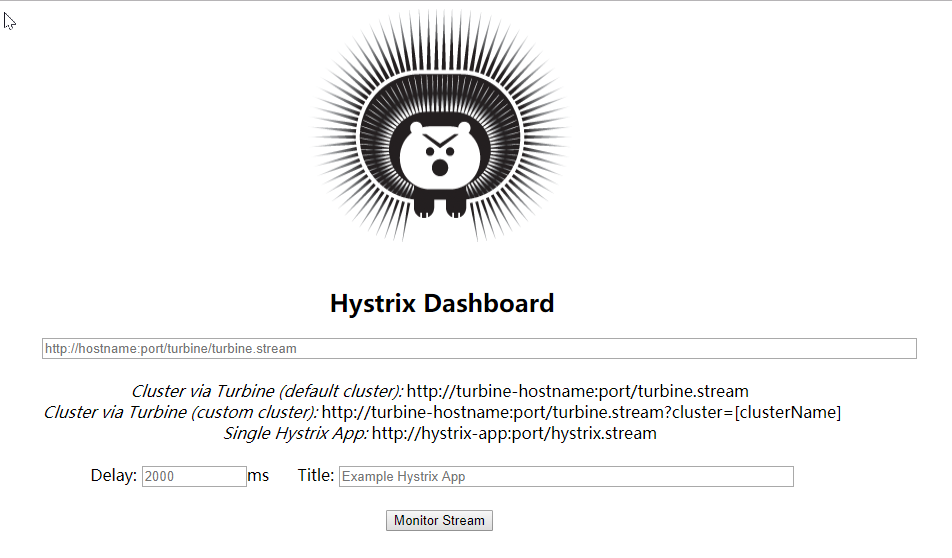
****

**e、所有生产者添加监控依赖（8001/8002/8003）：**



**f、启动microservicecloud-consumer-hystrix-dashboard监控消费者：**

**访问**<http://localhost:9001/hystrix>**，显示页面如下，即成功：**



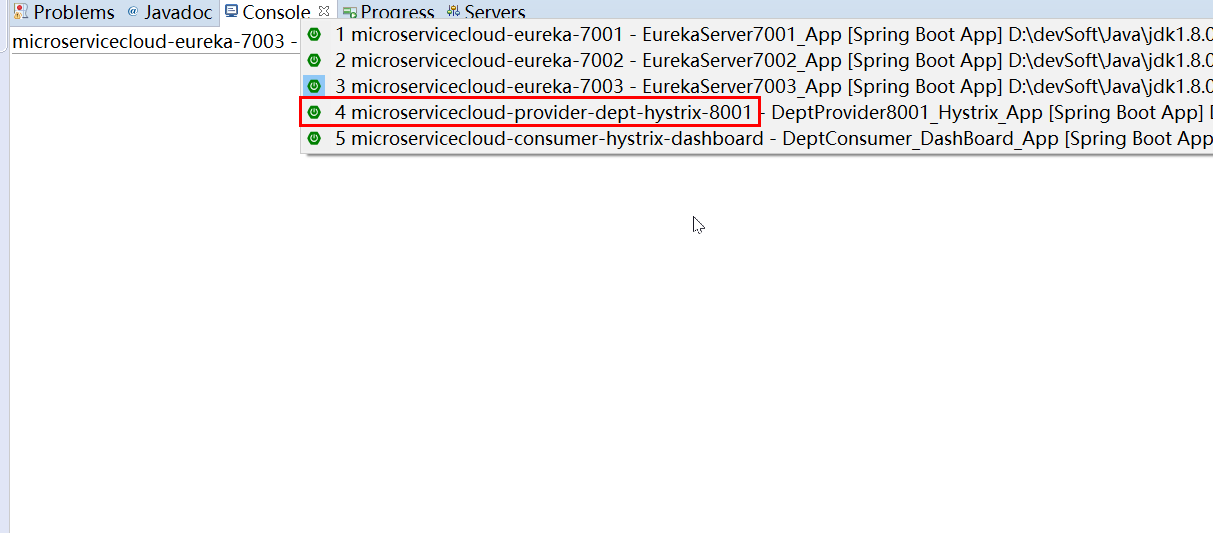
**g、启动三个eureka集群和microservicecloud-provider-dept-hystrix-8001生产者:**

访问<http://localhost:8001/dept/get/1>

访问<http://localhost:8001/hystrix.stream>，可得到如下**：**



**h、启动其他微服务:**

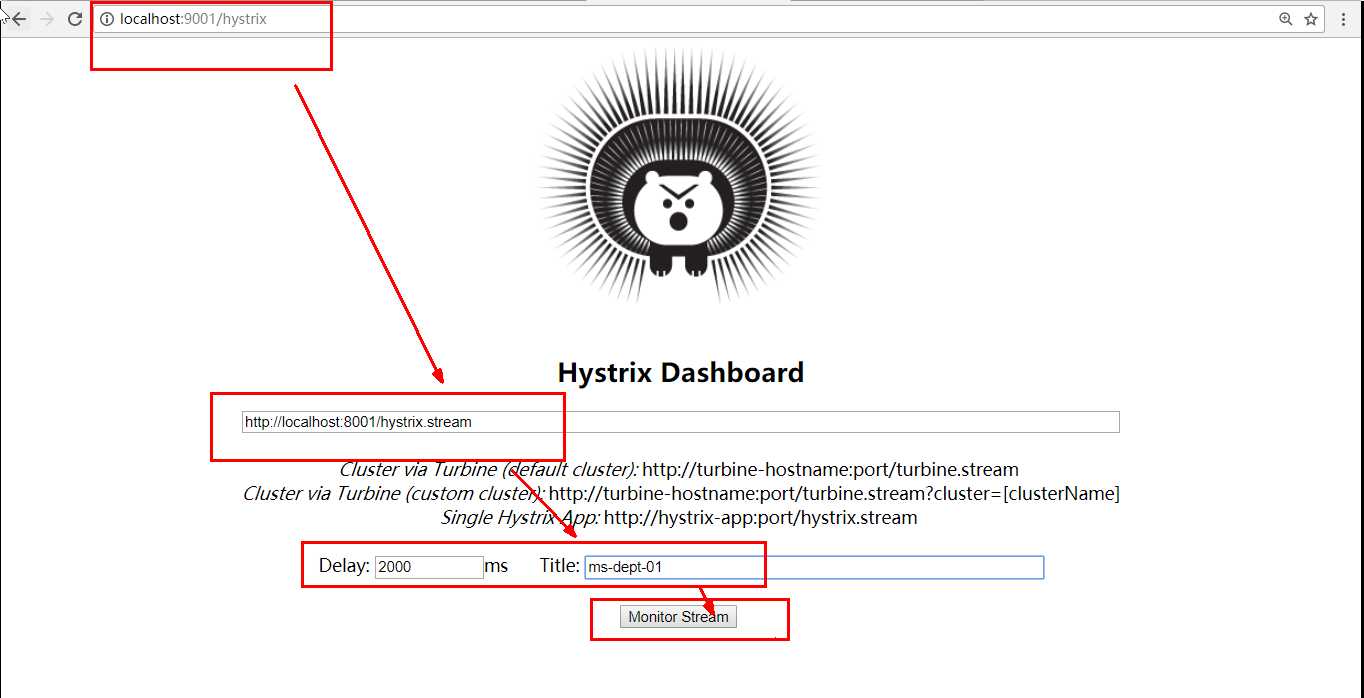


**I、 监控测试：**

多次刷新**：**<http://localhost:8001/dept/get/1>

观察监控窗口：

**填写监控地址**：



**监控结果：**



# 八、zuul路由网关：

1、是什么：

Zuul包含了对请求的路由和过滤两个最主要的功能：

其中**路由功能负责将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**过滤器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。**

2、能干什么：

**提供=代理+路由+过滤三大功能**

3、zuul路由的基本配置：

**①、新建model: microserviceclood-zuul-gateway-9527**

**②、pom.xml配置：**

|  |
| --- |
| **<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>**  **<parent>**  **<groupId>com.atguigu.springcloud</groupId>**  **<artifactId>microservicecloud</artifactId>**  **<version>0.0.1-SNAPSHOT</version>**  **</parent>**  **<artifactId>microservicecloud-zuul-gateway-9527</artifactId>**  **<dependencies>**  **<!-- 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>**  **<!-- 日常标配 -->**  **<dependency>**  **<groupId>com.atguigu.springcloud</groupId>**  **<artifactId>microservicecloud-api</artifactId>**  **<version>${project.version}</version>**  **</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>**  **<!-- 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>**  **</project>** |

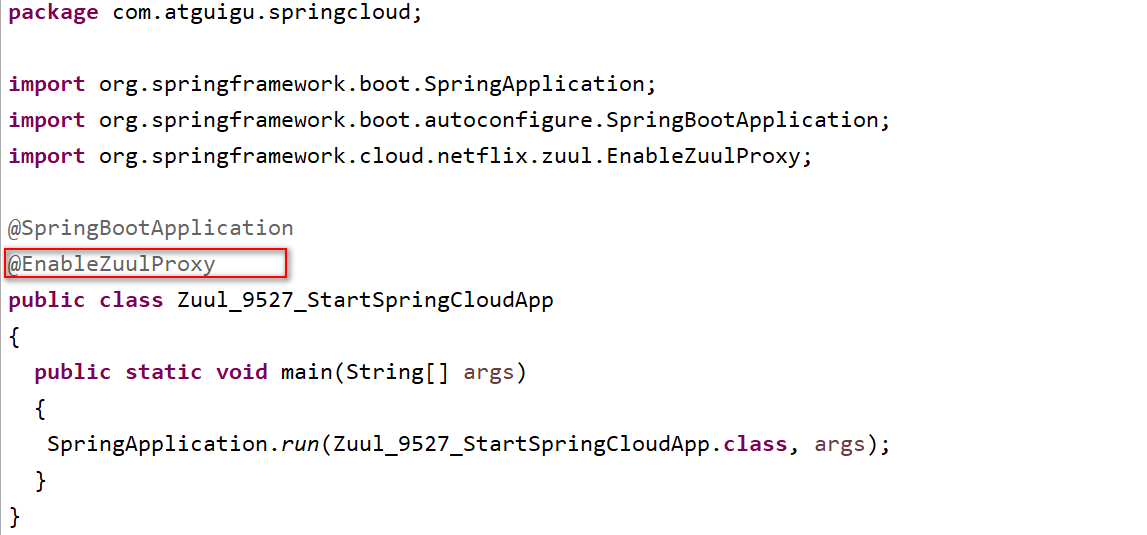
**③、yml文件的配置：**

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

**④、hosts修改：**



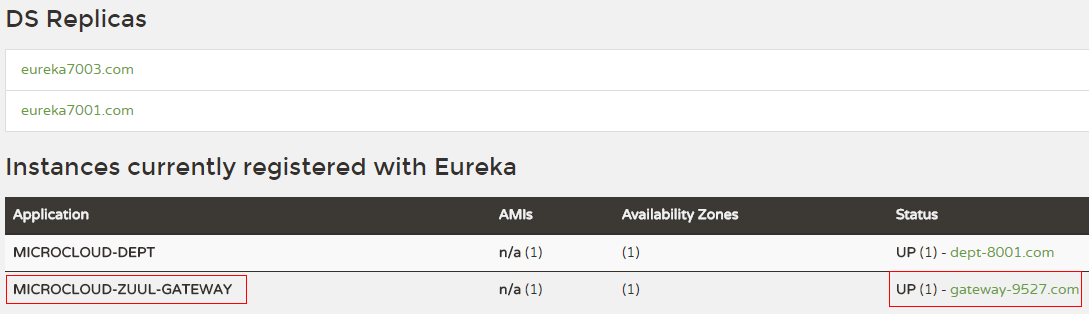
**⑤、主启动类添加注解：**



**⑥、启动服务：**



**启动microserviceclood-zuul-gateway-9527路由服务，查看Eureka注入情况如下：**



**⑦、测试：**



4、zuul路由访问映射规则：

**①、yml文件修改：**



**此时遇到的问题（原路径也能访问）：**



**②、解决原路径访问的问题（忽略原路径访问）：**



**单个用具体的服务名，多个用\*。**

**③、设置统一公共前缀：**



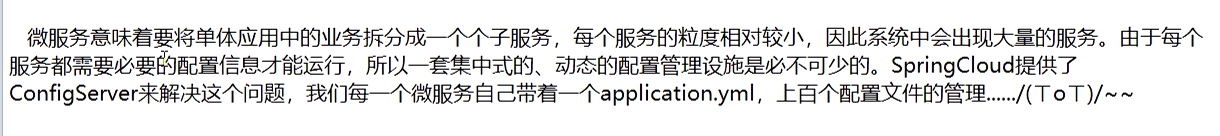
**④、最终的yml文件：**

|  |
| --- |
| **server:**  **port: 9527**  **spring:**  **application:**  **name: microservicecloud-zuul-gateway**  **zuul:**  **prefix: /atguigu**  **ignored-services: "\*"**  **routes:**  **mydept.serviceId: microservicecloud-dept**  **mydept.path: /mydept/\*\***  **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**  **info:**  **app.name: atguigu-microcloud**  **company.name: www.atguigu.com**  **build.artifactId: $project.artifactId$**  **build.version: $project.version$** |

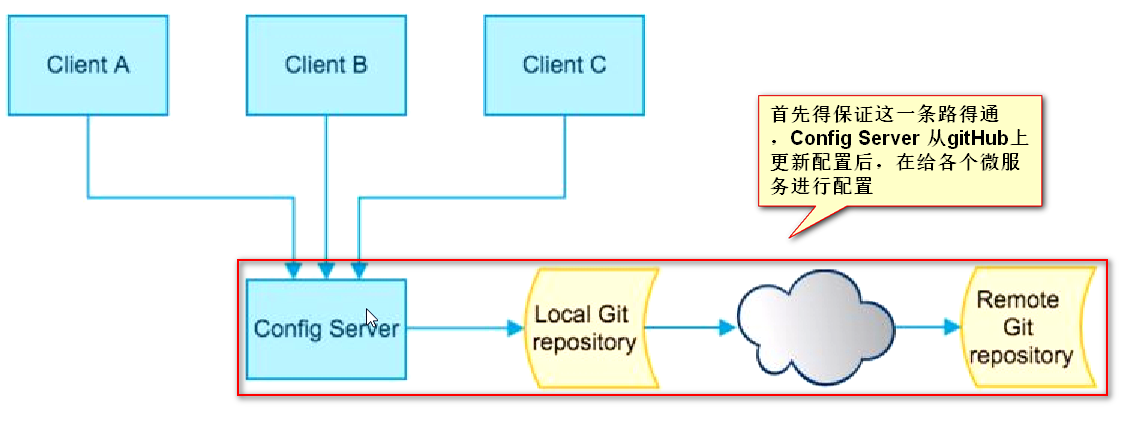
# 九、SpringCloud Config分布式配置中心：

1、概述：

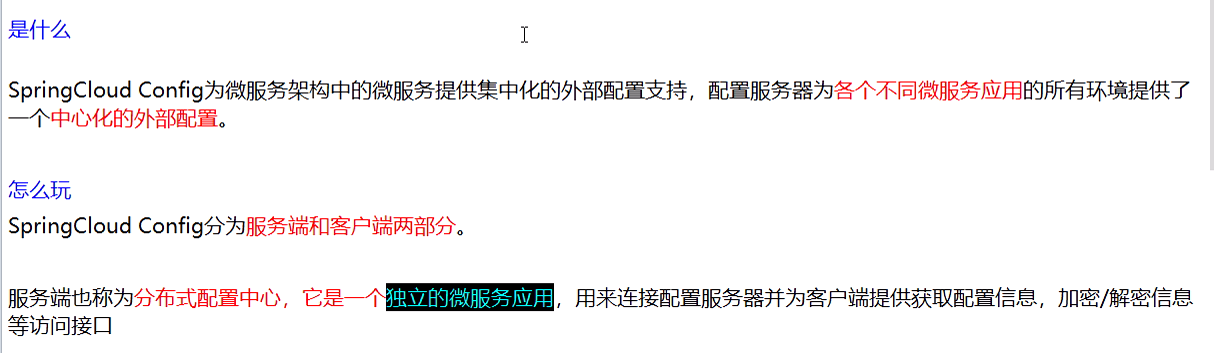
1. **分布式系统面临的问题：**

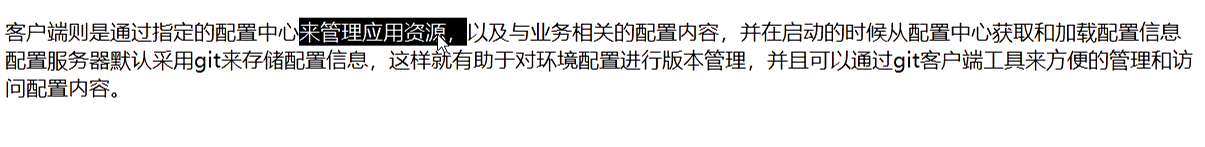


1. **是什么：**

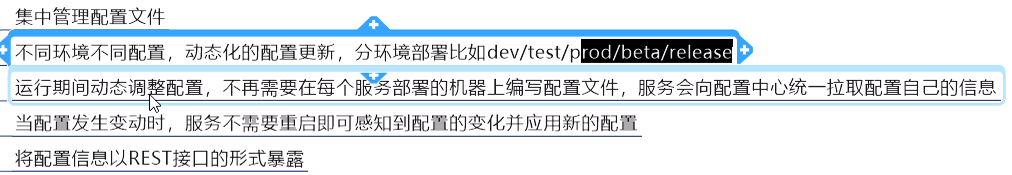


各个 客户端 通过 Spring-Cloud-Config-Server 从 配置仓库 拉取自己服务的配置。

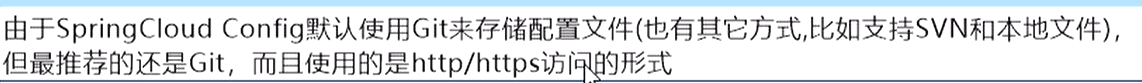




1. **能干嘛：**

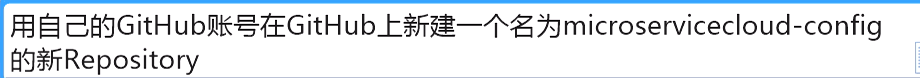


1. **与GitHub整合配置：**



2、SpringCloud Config服务端配置：

**（1） 建仓库：**



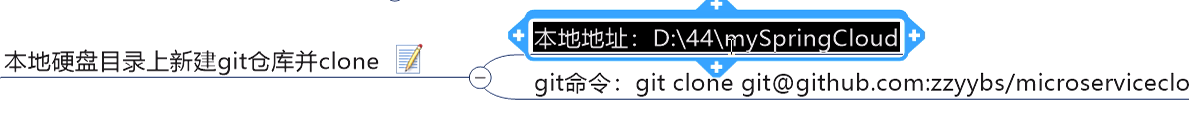


**（2） 获取git仓库地址：**





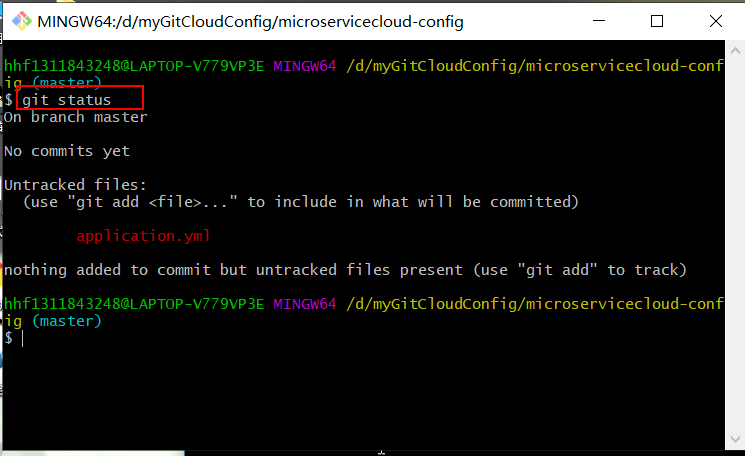
**（3） 在本地硬盘新建Git仓库并克隆：‘**



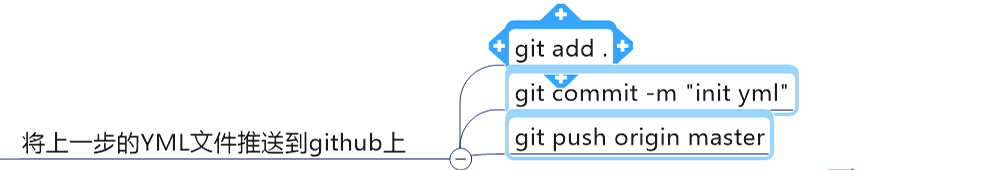
**（4） 编写application.yml文件，上传到github**

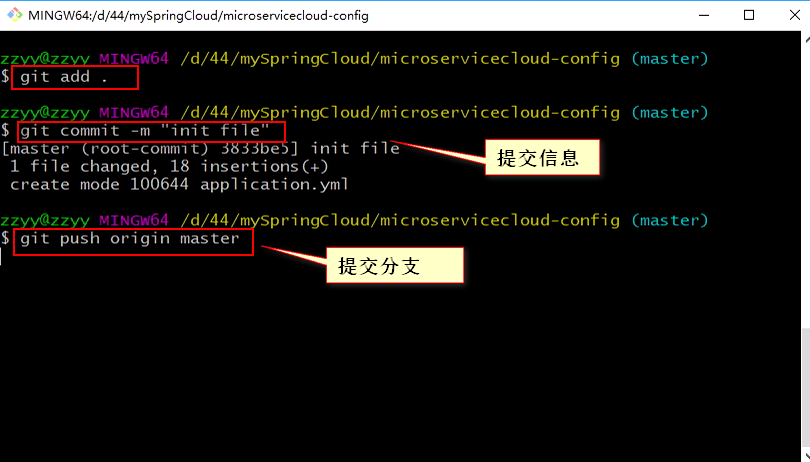


**查看详情：**

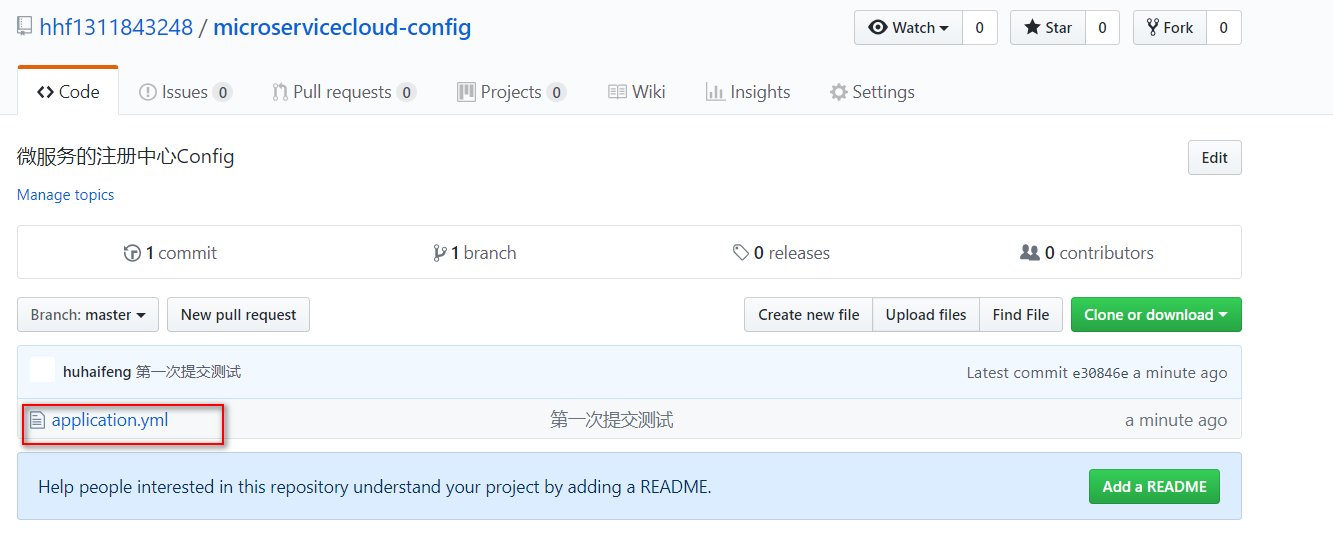


**提交github:**





**结果：**



**（5） 建microserviceclood-config-3344项目**

**配置pom.xml文件：**

|  |
| --- |
| <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>  <parent>  <groupId>com.hhf.springcloud</groupId>  <artifactId>microserviceclood</artifactId>  <version>0.0.1-SNAPSHOT</version>  </parent>  <artifactId>microserviceclood-config-3344</artifactId>    <dependencies>  <!-- springCloud Config -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency>  <!-- 避免Config的Git插件报错：org/eclipse/jgit/api/TransportConfigCallback -->  <dependency>  <groupId>org.eclipse.jgit</groupId>  <artifactId>org.eclipse.jgit</artifactId>  <version>4.10.0.201712302008-r</version>  </dependency>  <!-- 图形化监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- 熔断 -->  <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-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>  <!-- 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>    </project> |

**配置yml文件：**

|  |
| --- |
| server:  port: 3344    spring:  application:  name: microservicecloud-config  cloud:  config:  server:  git:  uri: git@github.com/hhf1311843248/microservicecloud-config.git #GitHub上面的git仓库名字 |

**修改启动类：**

|  |
| --- |
| **package com.hhf.springcloud;**  **import org.springframework.boot.SpringApplication;**  **import org.springframework.boot.autoconfigure.SpringBootApplication;**  **import org.springframework.cloud.config.server.EnableConfigServer;**  **@SpringBootApplication**  **@EnableConfigServer**  **public class Config\_3344\_StartSpringCloudApp**  **{**  **public static void main(String[] args)**  **{**  **SpringApplication.run(Config\_3344\_StartSpringCloudApp.class, args);**  **}**  **}** |

**（6） 修改host文件：**



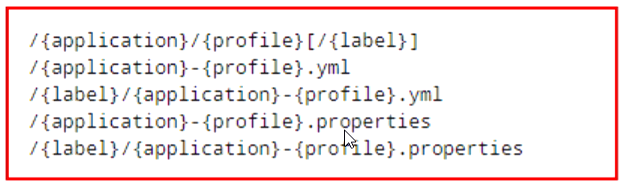
**（7） 浏览器访问：**

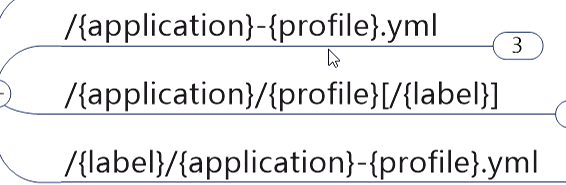
[**http://config-3344.com/application-dev.yml**](http://config-3344.com/application-dev.yml)

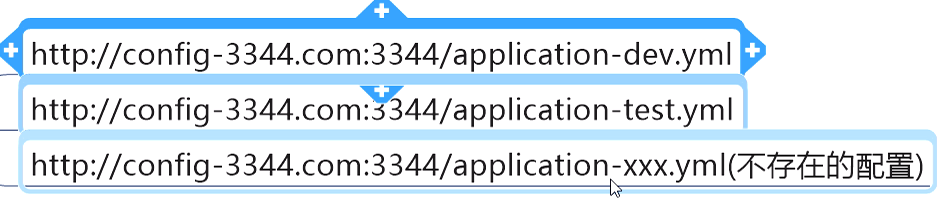
**结果（读取到了相关的配置）：**

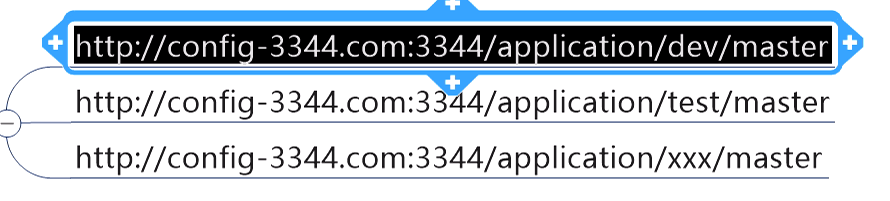


**（8） 读取配置的方式:**

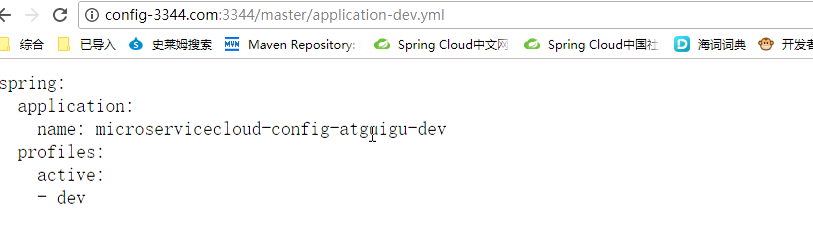






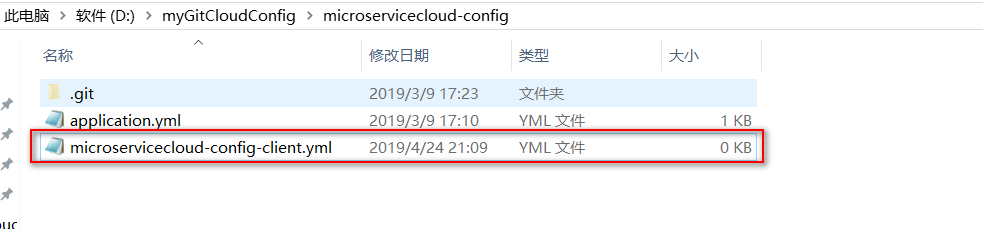






2、SpringCloud Config 客户端的配置与测试：

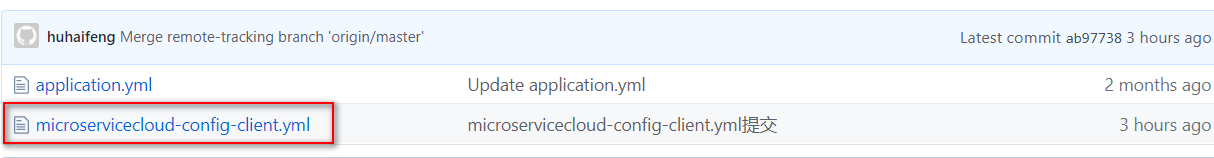
**①、在git本地SpringCloud仓库中新建一个yml文件：**



**yml文件内容为：**

|  |
| --- |
| **spring:**  **profiles:**  **active:**  **- dev**  **---**  **server:**  **port: 8201**  **spring:**  **profiles: dev**  **application:**  **name: microserviceclood-config-client**  **eureka:**  **client: #客户端注册进eureka服务列表内**  **service-url:**  **defaultZone: http://eureka-dev.com:7001/eureka/ #Eureka单机版**  **---**  **server:**  **port: 8202**  **spring:**  **profiles: sit**  **application:**  **name: microserviceclood-config-client**  **eureka:**  **client: #客户端注册进eureka服务列表内**  **service-url:**  **defaultZone: http://eureka-sit.com:7001/eureka/ #Eureka单机版** |

**②、将yml文件推送到gitHub上面去：**



**③、新建一个Module：microserviceclood-config-client-3355:**

**pom.xml文件为：**

|  |
| --- |
| *<?***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**>microserviceclood</**artifactId**>  <**groupId**>com.hhf.springcloud</**groupId**>  <**version**>0.0.1-SNAPSHOT</**version**>  </**parent**>  <**modelVersion**>4.0.0</**modelVersion**>   <**artifactId**>microserviceclood-config-client-3355</**artifactId**>  <**dependencies**>  *<!--Springcloud 客户端 -->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-config</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**>  *<!-- 熔断 -->* <**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-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**>  *<!-- 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**>  </**project**> |

**新建bootstrap.yml：**

|  |
| --- |
| **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** |

**修改hosts文件映射：**



**④、bootstrap.yml是什么：**



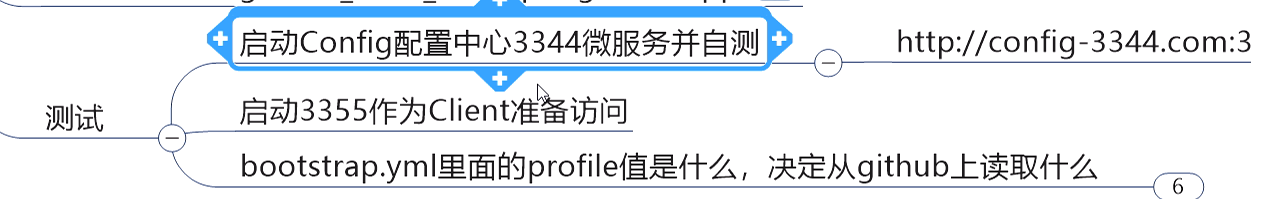
**⑤、新建启动类：**

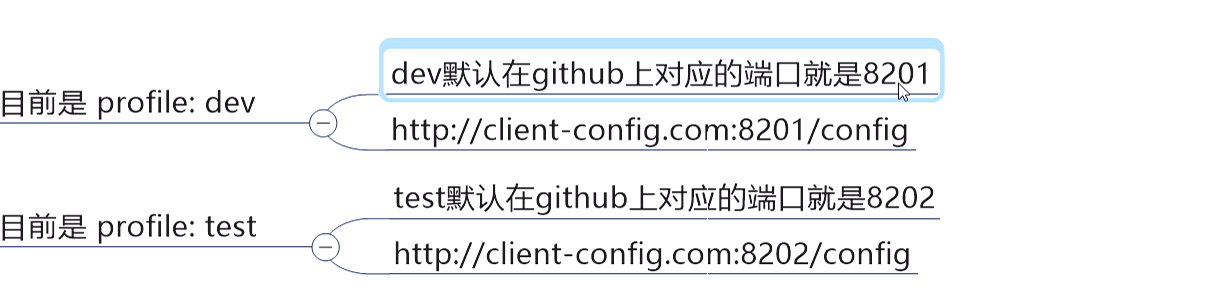
|  |
| --- |
| **package** com.hhf.springcloud;  **import** org.springframework.boot.SpringApplication; **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication **public class** ConfigClient\_3355\_StartSpringCloudApp {  **public static void** main(String[] args)  {  SpringApplication.*run*(ConfigClient\_3355\_StartSpringCloudApp.**class**, args);  } } |

**⑥、新建测试类：**

|  |
| --- |
| **package** com.hhf.springcloud.rest; **import** org.springframework.beans.factory.annotation.Value; **import** org.springframework.web.bind.annotation.RequestMapping; **import** org.springframework.web.bind.annotation.RestController;  @RestController **public class** ConfigClientRest {  @Value(**"${spring.application.name}"**)  **private** String **applicationName**;   @Value(**"${eureka.client.service-url.defaultZone}"**)  **private** String **eurekaServers**;   @Value(**"${server.port}"**)  **private** String **port**;   @RequestMapping(**"/config"**)  **public** String getConfig()  {  String str = **"applicationName: "** + **applicationName** + **"\t eurekaServers:"** + **eurekaServers** + **"\t port: "** + **port**;  System.***out***.println(**"\*\*\*\*\*\*str: "** + str);  **return "applicationName: "** + **applicationName** + **"\t eurekaServers:"** + **eurekaServers** + **"\t port: "** + **port**;  } } |

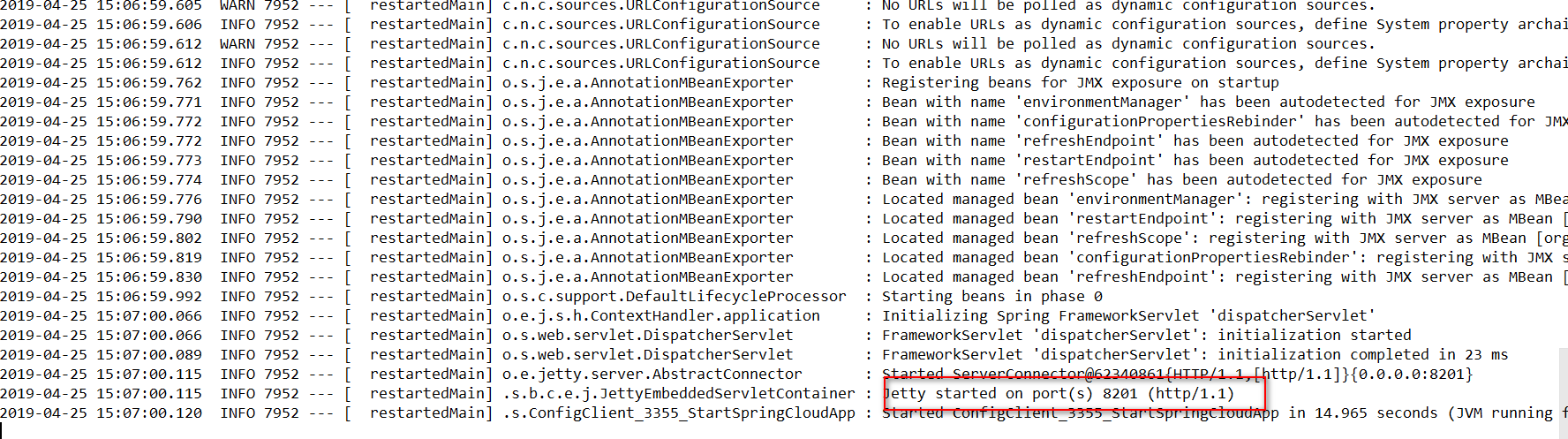
**⑦、测试：**



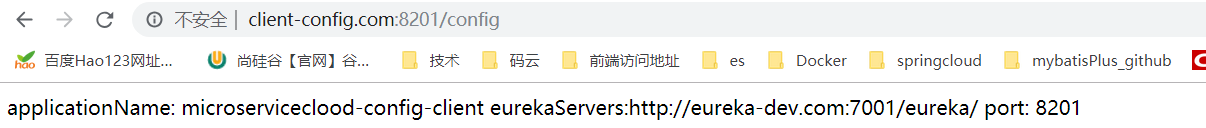


**我们项目中profile的值为dev，所以取的端口号为：8201**

**观察3355Module控制台信息，如下表示配置成功：**

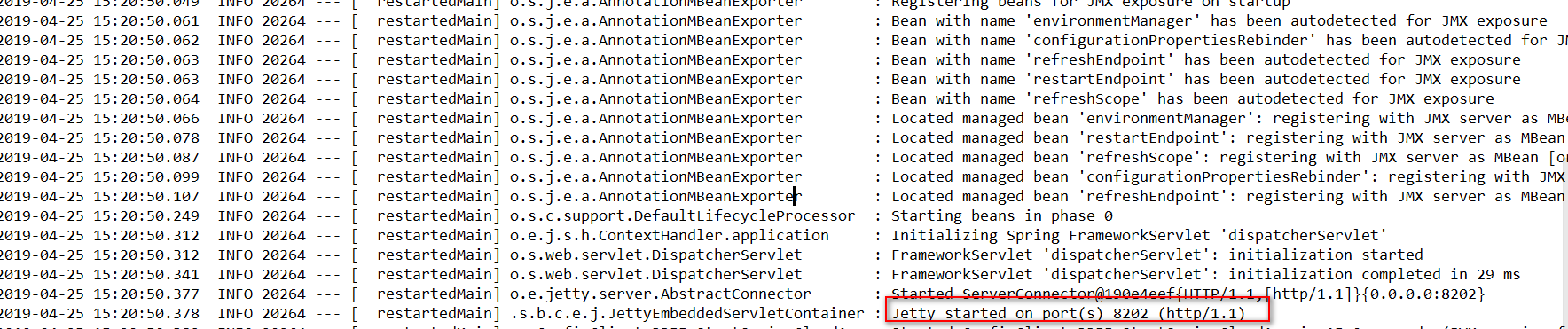


**访问页面8201获取配置信息：**

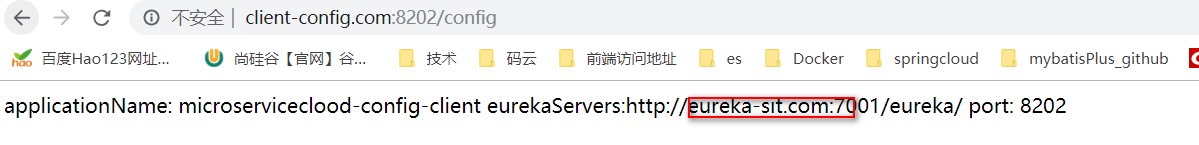


**切换profile的值为sit，所取的端口号应为：8202**

**观察3355Module控制台信息，如下：**



**访问页面8202获取配置信息：**

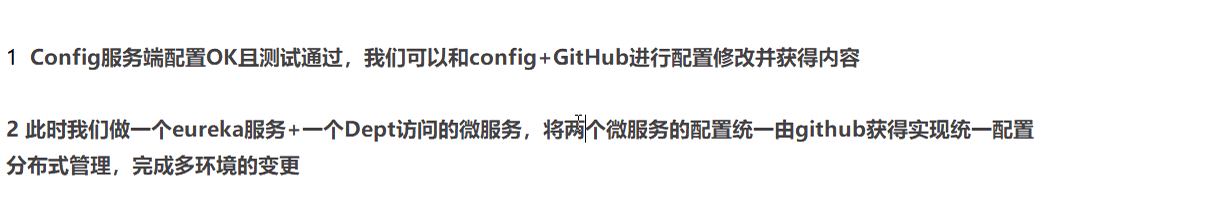


**说明动态配置成功。**

3、SpringCloud Config 配置实战：

**一个配置的provider-dept、eureka、都从3344中获取配置信息，并完成调用。**

**①、目前情况：**



**②、本地gitHub仓库新建yml文件：**

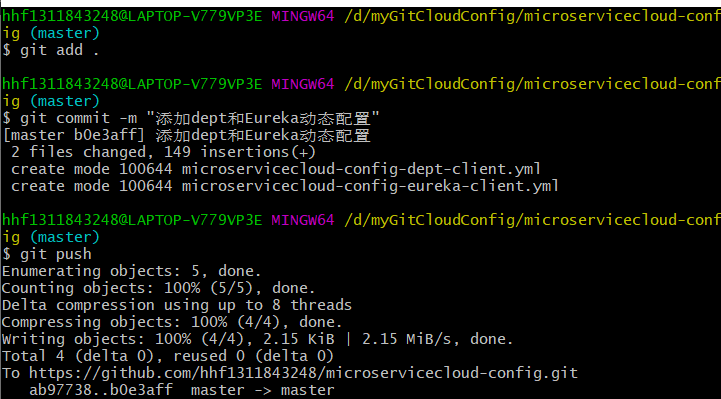
**microserviceclood-config-client-eureka-client.yml：**

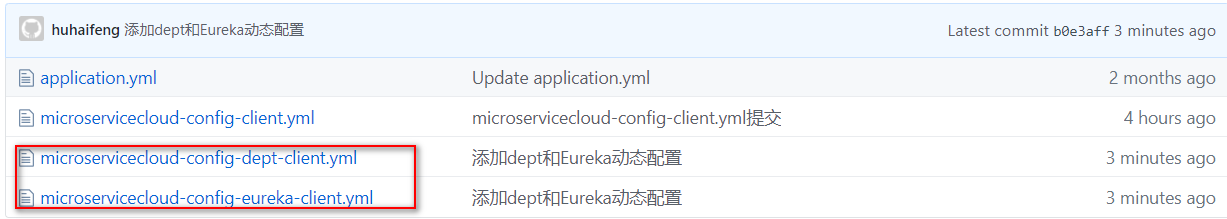
|  |
| --- |
| **spring:**  **profiles:**  **active:**  **- dev**  **---**  **server:**  **port: 7001**  **spring:**  **profiles: dev**  **application:**  **name: microserviceclood-config-client-eureka-client**  **eureka:**  **instance:**  **hostname: localhost #eureka服务端的实例名称（单机版的）**  **# hostname: eureka7001.com #eureka服务端的实例名称（找到C:\Windows\System32\drivers\etc路径下的hosts文件 在hosts中配置的地址映射） 集群**  **# server:**  **# enable-self-preservation: false #禁用自我保护模式 建议不禁用**  **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:**  **defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #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（单机）。**  **# defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/ 集群**  **---**  **server:**  **port: 7002**  **spring:**  **profiles: sit**  **application:**  **name: microserviceclood-config-client-eureka-client**  **eureka:**  **instance:**  **hostname: localhost #eureka服务端的实例名称（单机版的）**  **# hostname: eureka7001.com #eureka服务端的实例名称（找到C:\Windows\System32\drivers\etc路径下的hosts文件 在hosts中配置的地址映射） 集群**  **# server:**  **# enable-self-preservation: false #禁用自我保护模式 建议不禁用**  **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:**  **defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #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（单机）。**  **# defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/ 集群** |

**microservicecloud-config-dept-client.yml：**

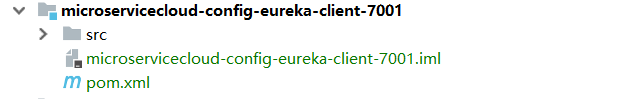
|  |
| --- |
| **spring:**  **profiles:**  **active:**  **- dev**  **---**  **server:**  **port: 8001**  **mybatis:**  **# config-location: classpath:mybatis/mybatis.cfg.xml # mybatis配置文件所在路径 （在启动类中已经配置）**  **type-aliases-package: com.hhf.springcloud.entity # 所有Entity别名类所在包**  **mapper-locations:**  **- classpath:mybatis/mapper/\*\*/\*.xml # mapper映射文件**  **configuration:**  **jdbcTypeForNull: 'null'**  **mapUnderscoreToCamelCase: true**  **log-impl: org.apache.ibatis.logging.stdout.StdOutImpl #控制台打印sql语句**  **spring:**  **profiles: dev**  **application:**  **name: microservicecloud-config-dept-client #很重要 对外暴露的微服务名字**  **datasource:**  **name: dataSource**  **type: com.alibaba.druid.pool.DruidDataSource # 当前数据源操作类型**  **# driver-class-name: org.gjt.mm.mysql.Driver # mysql驱动包**  **driver-class-name: com.mysql.jdbc.Driver # mysql驱动包**  **url: jdbc:mysql://127.0.0.1:3306/cloudDB01?serverTimezone=GMT%2B8 # 数据库名称**  **username: root**  **password: root**  **dbcp2:**  **min-idle: 5 # 数据库连接池的最小维持连接数**  **initial-size: 5 # 初始化连接数**  **max-total: 5 # 最大连接数**  **max-wait-millis: 200 # 等待连接获取的最大超时时间**  **eureka:**  **client: #客户端注册进eureka服务列表内**  **service-url:**  **defaultZone: http://eureka7001.com:7001/eureka #Eureka单机版**  **# defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/ # Eureka 集群设置**  **instance:**  **instance-id: microservicecloud-dept8001**  **prefer-ip-address: true #访问路径可以显示IP地址**  **#微服务的Info信息的详情**  **info:**  **app.name: hhf-microservicecloud**  **company.name: www.hhf.com**  **build.artifactId: $project.artifactId$**  **build.version: $project.version$**  **---**  **server:**  **port: 8002**  **mybatis:**  **# config-location: classpath:mybatis/mybatis.cfg.xml # mybatis配置文件所在路径 （在启动类中已经配置）**  **type-aliases-package: com.hhf.springcloud.entity # 所有Entity别名类所在包**  **mapper-locations:**  **- classpath:mybatis/mapper/\*\*/\*.xml # mapper映射文件**  **configuration:**  **jdbcTypeForNull: 'null'**  **mapUnderscoreToCamelCase: true**  **log-impl: org.apache.ibatis.logging.stdout.StdOutImpl #控制台打印sql语句**  **spring:**  **profiles: sit**  **application:**  **name: microservicecloud-config-dept-client #很重要 对外暴露的微服务名字**  **datasource:**  **name: dataSource**  **type: com.alibaba.druid.pool.DruidDataSource # 当前数据源操作类型**  **# driver-class-name: org.gjt.mm.mysql.Driver # mysql驱动包**  **driver-class-name: com.mysql.jdbc.Driver # mysql驱动包**  **url: jdbc:mysql://127.0.0.1:3306/cloudDB02?serverTimezone=GMT%2B8 # 数据库名称**  **username: root**  **password: root**  **dbcp2:**  **min-idle: 5 # 数据库连接池的最小维持连接数**  **initial-size: 5 # 初始化连接数**  **max-total: 5 # 最大连接数**  **max-wait-millis: 200 # 等待连接获取的最大超时时间**  **eureka:**  **client: #客户端注册进eureka服务列表内**  **service-url:**  **defaultZone: http://eureka7002.com:7002/eureka #Eureka单机版**  **# defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/ # Eureka 集群设置**  **instance:**  **instance-id: microservicecloud-dept8001**  **prefer-ip-address: true #访问路径可以显示IP地址**  **#微服务的Info信息的详情**  **info:**  **app.name: hhf-microservicecloud**  **company.name: www.hhf.com**  **build.artifactId: $project.artifactId$**  **build.version: $project.version$** |

**③、将新建yml文件上传到gitHub上面：**





**④、新建配置版的Eureka Module：**



**pom.xml文件（直接从7001上面复制过来,注意config添加客户端依赖）：**

|  |
| --- |
| *<?***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**>microserviceclood</**artifactId**>  <**groupId**>com.hhf.springcloud</**groupId**>  <**version**>0.0.1-SNAPSHOT</**version**>  </**parent**>  <**modelVersion**>4.0.0</**modelVersion**>   <**artifactId**>microservicecloud-config-eureka-client-7001</**artifactId**>  <**dependencies**>  *<!--Springcloud 客户端 -->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-config</**artifactId**> </**dependency**>  *<!--eureka-server服务端 -->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-eureka-server</**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**> </**project**> |

**启动类（直接从7001上面复制过来）：**

|  |
| --- |
| **package** com.hhf.springcloud;  **import** org.springframework.boot.SpringApplication; **import** org.springframework.boot.autoconfigure.SpringBootApplication; **import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer; @SpringBootApplication @EnableEurekaServer*//EurekaServer服务器端启动类,接受其它微服务注册进来* **public class** ConfigEurekaServer7001\_App {  **public static void** main(String[] args)  {  SpringApplication.*run*(ConfigEurekaServer7001\_App.**class**, args);  } } |

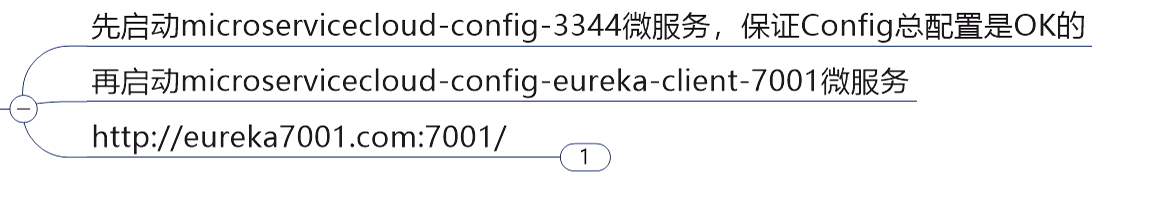
**Bootstrap.yml文件：**

|  |
| --- |
| **spring:   cloud:   config:   name:** microservicecloud-config-eureka-client *#需要从github上读取的资源名称，注意没有yml后缀名* **profile:** dev   **label:** master   **uri:** http://config-3344.com:3344 *#SpringCloudConfig获取的服务地址* |

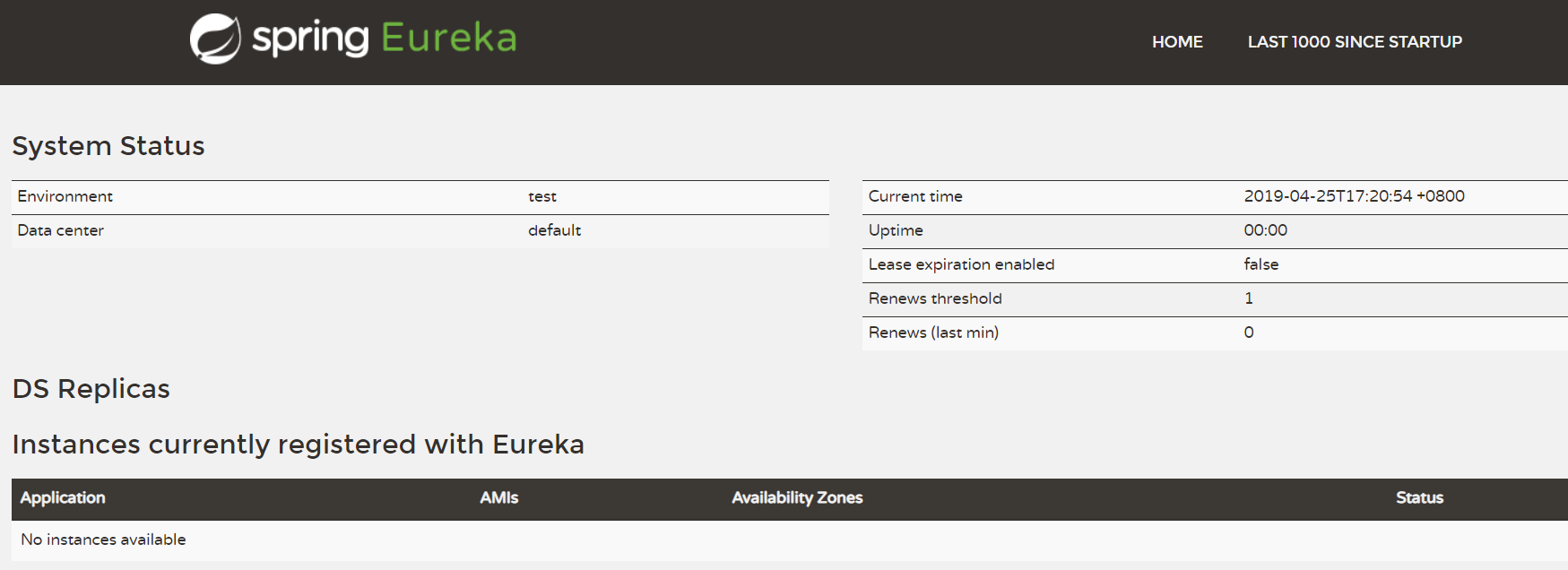
**application.yml文件：**

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

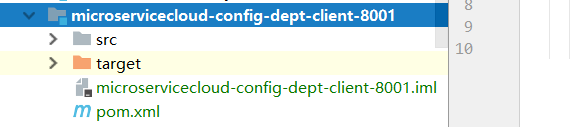
**测试：**



**访问localhost:7001，显示如下：**



**⑤、新建配置版的dept Module：**



**pom.xml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**>microserviceclood</**artifactId**>  <**groupId**>com.hhf.springcloud</**groupId**>  <**version**>0.0.1-SNAPSHOT</**version**>  </**parent**>  <**modelVersion**>4.0.0</**modelVersion**>  <**artifactId**>microservicecloud-config-dept-client-8001</**artifactId**>  <**dependencies**>  *<!-- 引入自己定义的api通用包，可以使用Dept部门Entity -->* <**dependency**>  <**groupId**>com.hhf.springcloud</**groupId**>  <**artifactId**>microservicecloud-api</**artifactId**>  <**version**>${project.version}</**version**>  </**dependency**>  *<!--Springcloud 客户端 -->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-config</**artifactId**>  </**dependency**>   *<!-- actuator监控信息完善 hystrix仪表盘监控 添加了此依赖 生产者才能被监控-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**>  *<!-- 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**>   <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  </**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**>  </**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**>  </**project**> |

**Bootstrap.yml文件：**

|  |
| --- |
| **spring:  cloud:  config:  name:** microservicecloud-config-dept-client *#需要从github上读取的资源名称，注意没有yml后缀名* **profile:** sit *#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* |

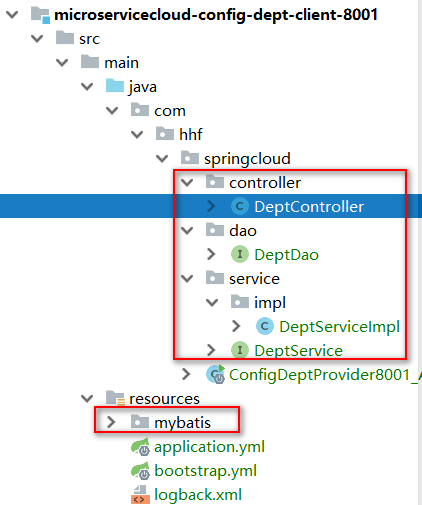
**application.yml文件：**

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

**启动类为:**

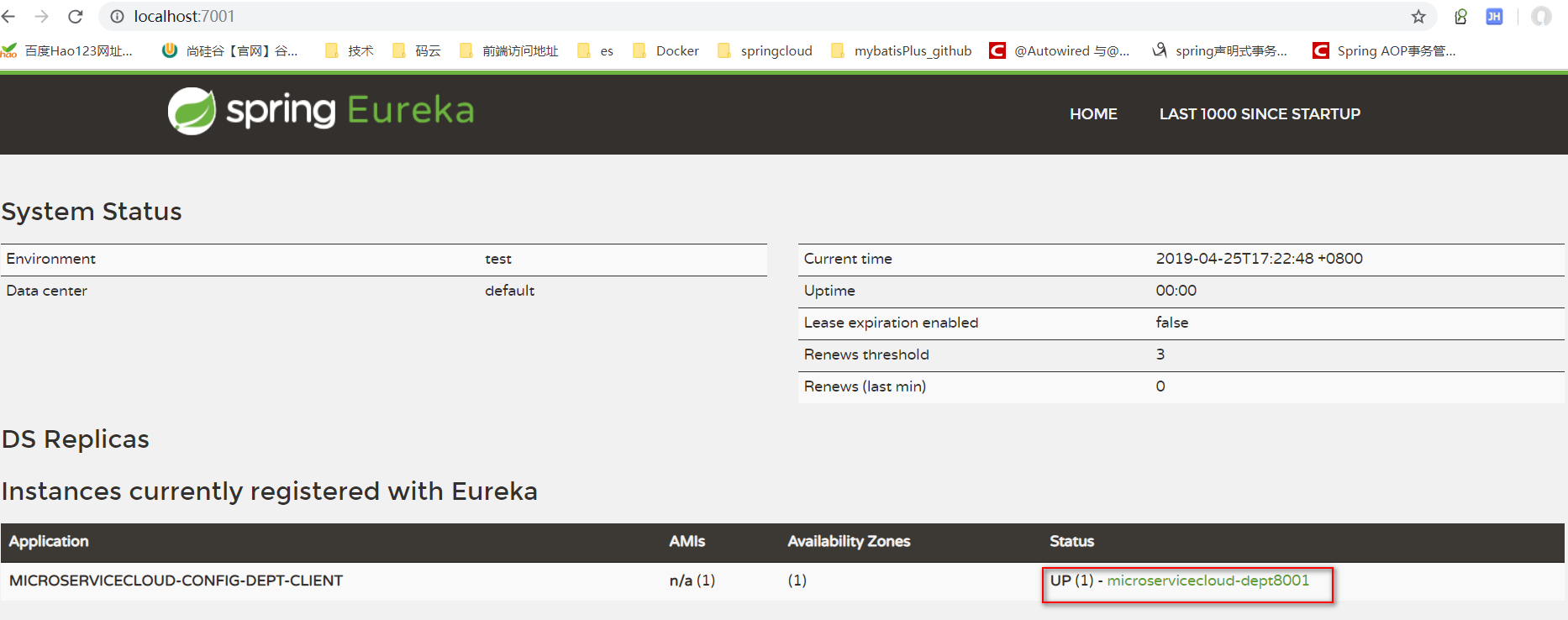
|  |
| --- |
| **package** com.hhf.springcloud; **import** org.mybatis.spring.annotation.MapperScan; **import** org.springframework.boot.SpringApplication; **import** org.springframework.boot.autoconfigure.SpringBootApplication; **import** org.springframework.cloud.netflix.eureka.EnableEurekaClient; **import** org.springframework.context.annotation.ImportResource; @SpringBootApplication @MapperScan(value = {**"com.hhf.springcloud.dao"**,**"com.hhf.springcloud.mapper"**}) @ImportResource(locations = **"classpath:mybatis/mybatis.cfg.xml"**) *//启动时加载config目录下面的xml配置文件* @EnableEurekaClient *//本服务启动后会自动注册进eureka服务中(本服务为客户端)* **public class** ConfigDeptProvider8001\_App {  **public static void** main(String[] args) {  SpringApplication.*run*(ConfigDeptProvider8001\_App.**class**, args);  } } |

**测试类（直接从8001dept复制过来）：**



**测试：**





**访问页面，如下表示配置成功：**

