# Spring Cloud简介

Spring Cloud是一系列框架的有序集合，利用Springbok的便利性巧妙地简化了分布式系统的开发。因此，Spring Cloud离不开Spring boot，但是Springbok可以离开Spring Cloud进行开发。

Spring Cloud下的组件有：

1. Spring Cloud Config
2. Spring Cloud Netflix（Eureka、Hystrix、Zuul、Archaius…）
3. Spring Cloud Bus
4. Spring Cloud for Cloud Foundry
5. Spring Cloud Cluster
6. Spring Cloud Consul
7. Spring Cloud Security
8. Spring Cloud Sleuth
9. Spring Cloud Data Flow
10. Spring Cloud Stream
11. Spring Cloud Task
12. Spring Cloud Zookeeper
13. Spring Cloud Connectors
14. Spring Cloud Starters
15. Spring Cloud CLI

下面就这些组件进行介绍。

## Spring Cloud Config

俗称配置中心，将配置文件放在一起，进行统一管理，方便日后更新配置等。目前支持本地化存储、Git存储以及Subversion。

## Spring Cloud Netflix

这可是个大boss，地位仅次于老大，老大各项服务依赖与它，与各种Netflix OSS组件集成，组成微服务的核心，它的小弟主要有Eureka, Hystrix, Zuul, Archaius… 太多了。

## Spring Cloud Bus

事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。确保各个组件之间消息保持畅通。

## Spring Cloud Cluster

Spring Cloud Cluster将取代Spring Integration。提供在分布式系统中的集群所需要的基础功能支持，如：选举、集群的状态一致性、全局锁、tokens等常见状态模式的抽象和实现。

## Spring Cloud for Cloud Foundry

Cloud Foundry是VMware推出的业界第一个开源PaaS云平台，它支持多种框架、语言、运行时环境、云平台及应用服务，使开发人员能够在几秒钟内进行应用程序的部署和扩展，无需担心任何基础架构的问题。

## Spring Cloud Consul

Consul 是一个支持多数据中心分布式高可用的服务发现和配置共享的服务软件,由 HashiCorp 公司用 Go 语言开发, 基于 Mozilla Public License 2.0 的协议进行开源. Consul 支持健康检查,并允许 HTTP 和 DNS 协议调用 API 存储键值对。

## Spring Cloud Security

基于spring security的安全工具包，为你的应用程序添加安全控制。专门负责整个应用的安全问题，设置资源的访问权限。

## Spring Cloud Sleuth

日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。

## Spring Cloud Data Flow

Data flow 是一个用于开发和执行大范围数据处理其模式包括ETL，批量运算和持续运算的统一编程模型和托管服务。

对于在现代运行环境中可组合的微服务程序来说，Spring Cloud data flow是一个原生云可编配的服务。使用Spring Cloud data flow，开发者可以为像数据抽取，实时分析，和数据导入/导出这种常见用例创建和编配数据通道 （data pipelines）。

Spring Cloud data flow 是基于原生云对 spring XD的重新设计，该项目目标是简化大数据应用的开发。Spring XD 的流处理和批处理模块的重构分别是基于 Spring Boot的stream 和 task/batch 的微服务程序。这些程序现在都是自动部署单元而且他们原生的支持像 Cloud Foundry、Apache YARN、Apache Mesos和Kubernetes 等现代运行环境。

Spring Cloud data flow 为基于微服务的分布式流处理和批处理数据通道提供了一系列模型和最佳实践。

## Spring Cloud Stream

Spring Cloud Stream是创建消息驱动微服务应用的框架。Spring Cloud Stream是基于Spring Boot创建，用来建立单独的／工业级spring应用，使用spring integration提供与消息代理之间的连接。数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

一个业务会牵扯到多个任务，任务之间是通过事件触发的，这就是Spring Cloud stream要干的事了

## Spring Cloud Task

Spring Cloud Task 主要解决短命微服务的任务管理，任务调度的工作，比如说某些定时任务晚上就跑一次，或者某项数据分析临时就跑几次。

## Spring Cloud Zookeeper

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。ZooKeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

操作Zookeeper的工具包，用于使用zookeeper方式的服务发现和配置管理，抱了Zookeeper的大腿。

## 1.13 Spring Cloud Connectors

Spring Cloud Connectors 简化了连接到服务的过程和从云平台获取操作的过程，有很强的扩展性，可以利用Spring Cloud Connectors来构建你自己的云平台。

便于云端应用程序在各种PaaS平台连接到后端，如：数据库和消息代理服务。

## Spring Cloud Starters

Spring Boot式的启动项目，为Spring Cloud提供开箱即用的依赖管理。

## Spring Cloud CLI

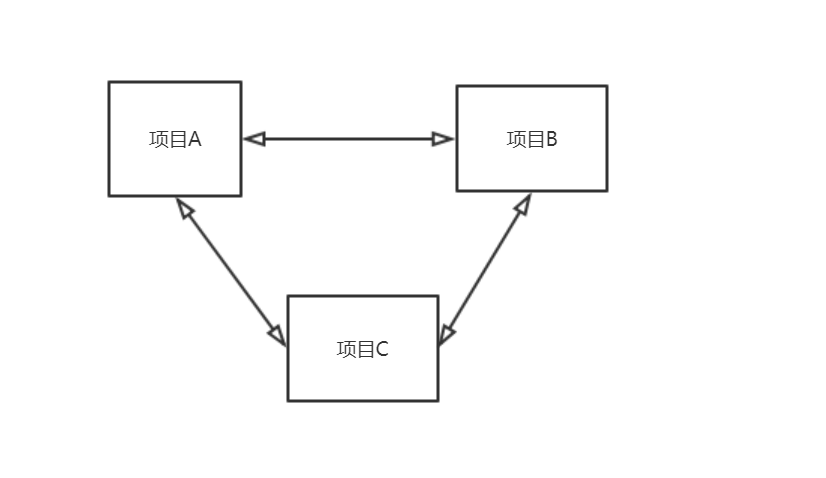
基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。

# Euraka

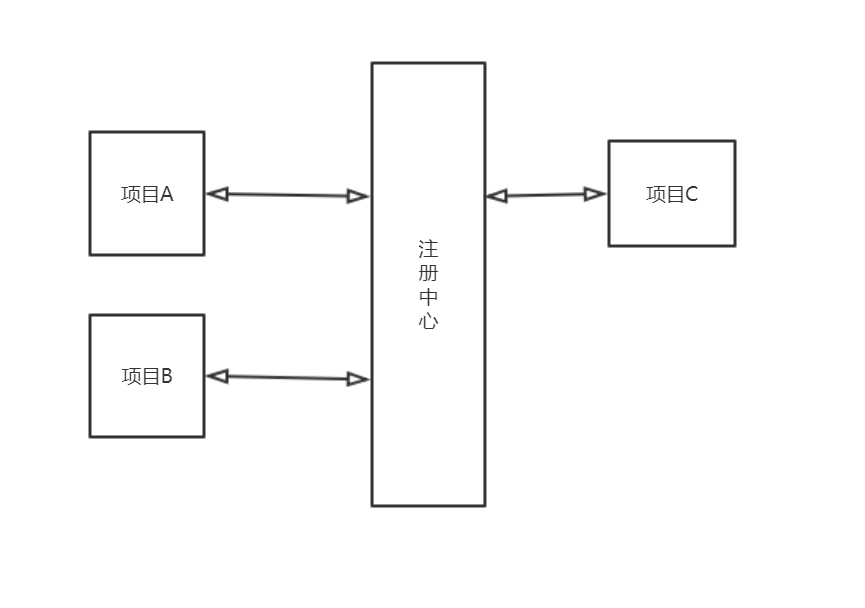
Eureka是Netflix开源的一款提供服务注册和发现的产品，它提供了完整的Service Registry和Service Discovery实现。也是spring cloud体系中最重要最核心的组件之一。

服务中心又称注册中心，管理各种服务功能包括服务的注册、发现、熔断、负载、降级等，比如dubbo admin后台的各种功能。

如果不是用服务注册中心，三个项目的互相调用可能是这样的：



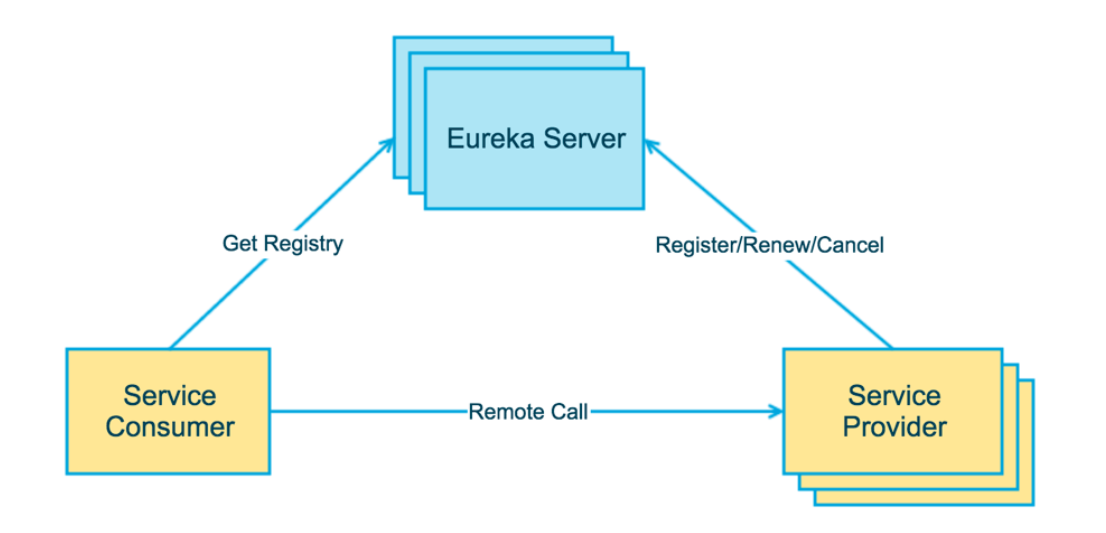
这时候如果想要添加一个项目D嵌入到这个系统去会觉得很困难。因为它们的调用关系比较复杂。如果项目扩展到几十个就更加不可想象了。

下面是将项目（服务）注册到服务注册中心后的调用关系：

这时候我们只需要往注册中心添加我们的项目即可，剩下的交给注册中心去管理。

Netflix是一家做流媒体播放的美国公司。Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现。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。

Eureka由两个组件组成：Eureka服务器和Eureka客户端。Eureka服务器用作服务注册服务器。Eureka客户端是一个java客户端，用来简化与服务器的交互、作为轮询负载均衡器，并提供服务的故障切换支持。Netflix在其生产环境中使用的是另外的客户端，它提供基于流量、资源利用率以及出错状态的加权负载均衡。



上图有三个角色：

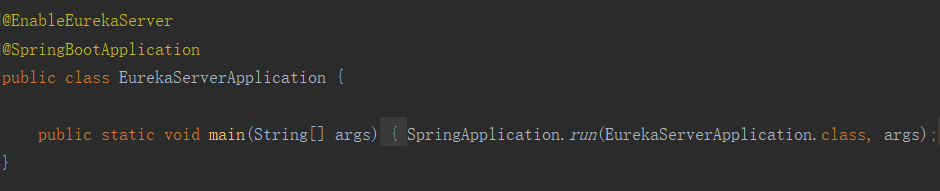
1. **Eureka Server**
   1. 提供服务注册与发现
2. **Service Provider**
   1. 服务提供方。将自身服务注册到Eureka,并被Service Consumer找到。
3. **Service Consumer**
   1. 服务消费方。从Eureka获取注册服务列表，从而能够消费服务。

## Eureka Server

Pom依赖：



启动类中添加@EnableEurekaServer注解：



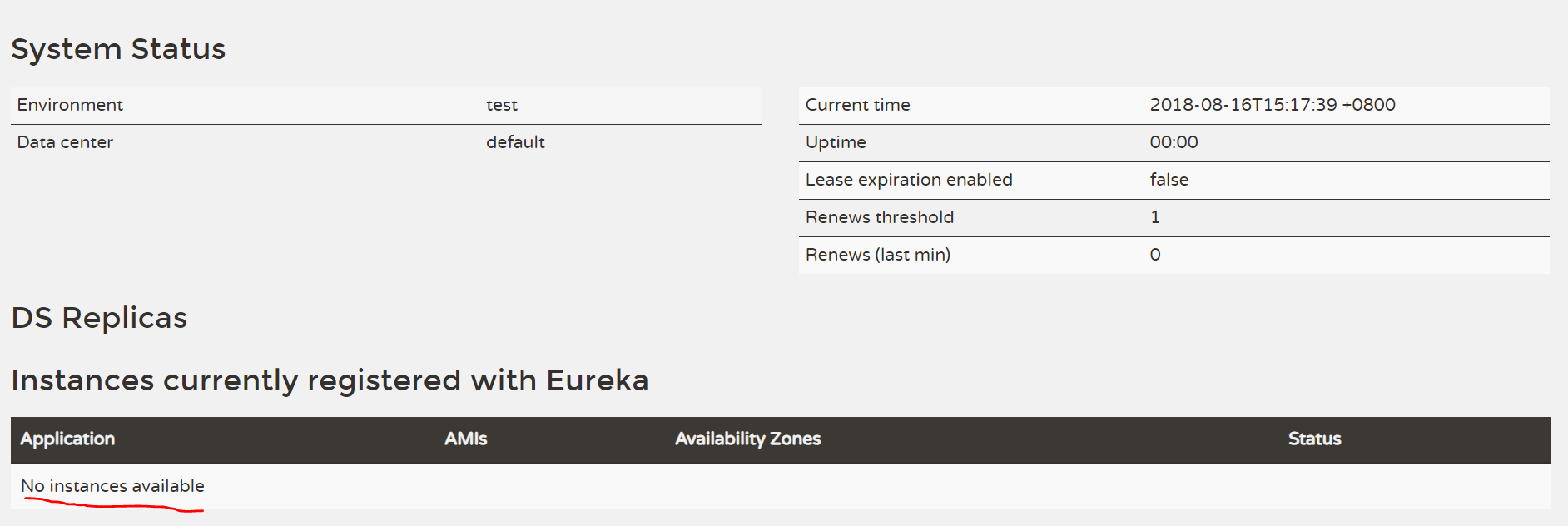
application.properties配置文件下配置name、ip：

*修改：-eureka.client.register-with-eureka=false*

*表面不检索服务，因为注册中心的职责是维护服务实例。*



打开<http://localhost:7080/>显示页面：

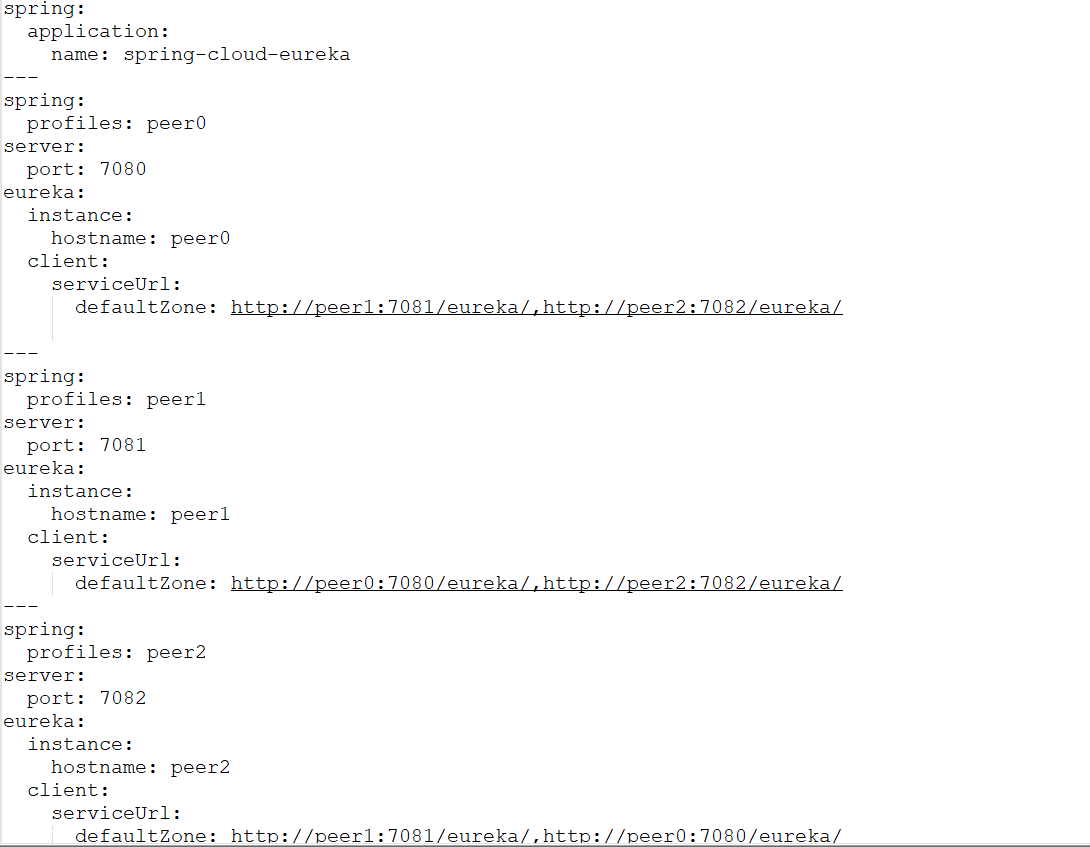


这时候是没有任何服务注册进来的。

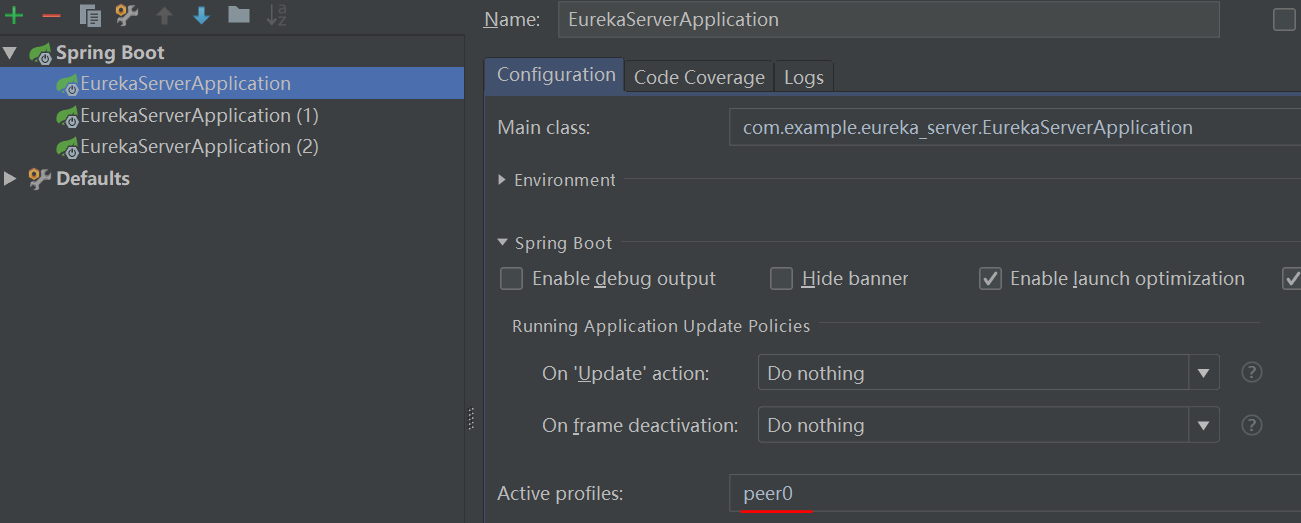
如果注册中心是单点的话，一旦出现故障，所有服务都不可使用。在一个分布式系统中，服务注册中心是最重要的基础部分，理应随时处于可以提供服务的状态。为了维持其可用性，使用集群是很好的解决方案。Eureka通过互相注册的方式来实现高可用的部署。

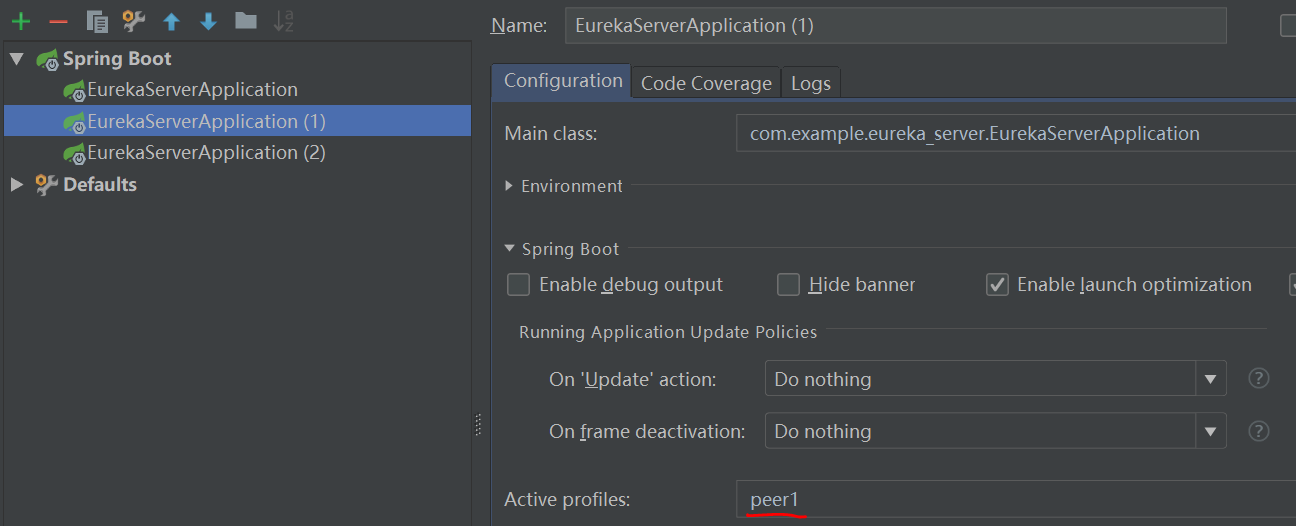
### 三节点的注册中心实现高可用

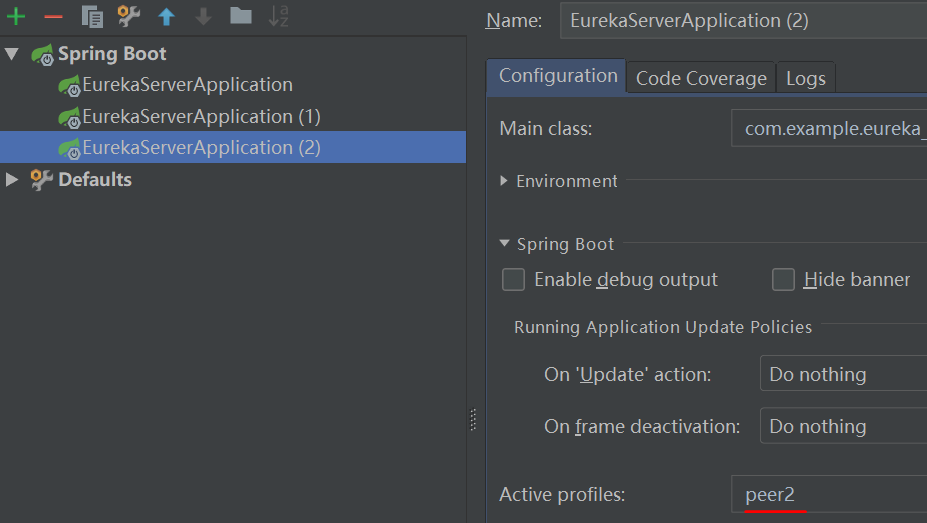
1) 删除application.properties，将application.yml修改如下：



2) 编辑IDEA启动配置





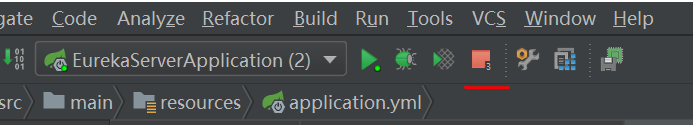


3） 添加到windows/System32/drivers/etc/hosts文件：

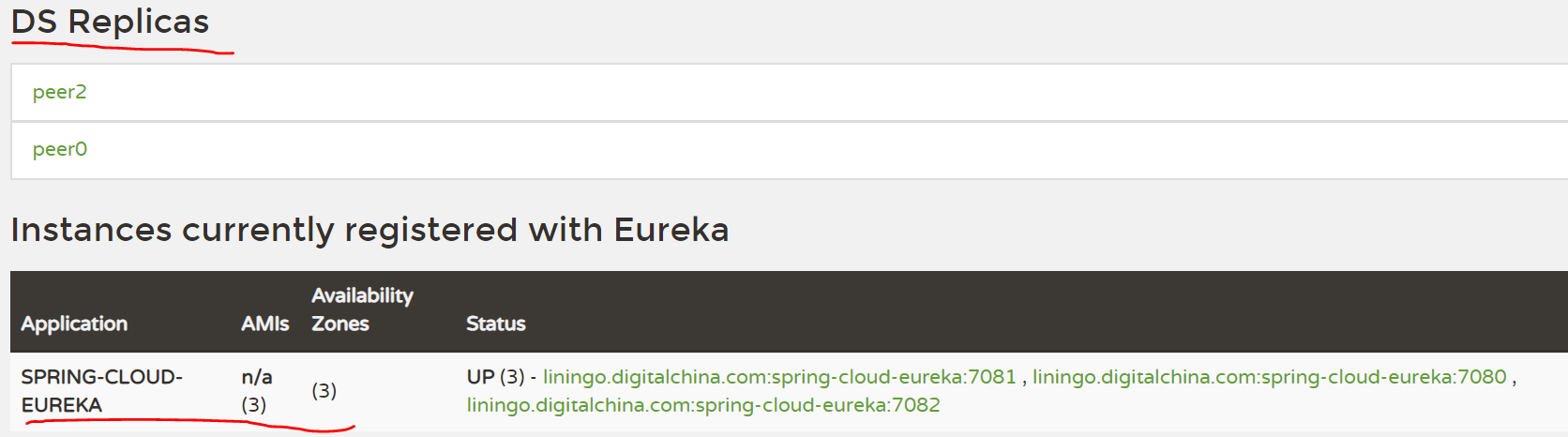


4）分别启动

EurekaServerApplication、EurekaServerApplication(1)、EurekaServerApplication(2)



5）页面访问http://localhost:7081/

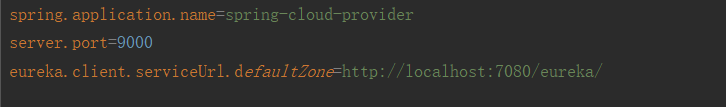


DS Replicas表示DS副本。即表示具有这三个注册中心是一样的。

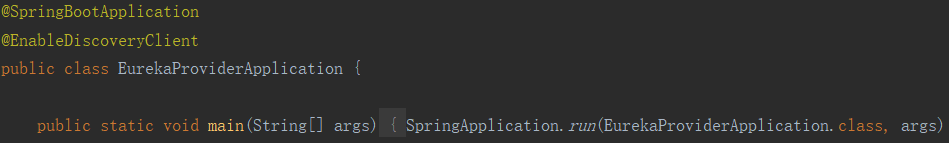
下面分别介绍服务提供者和消费者。

## Service Provider

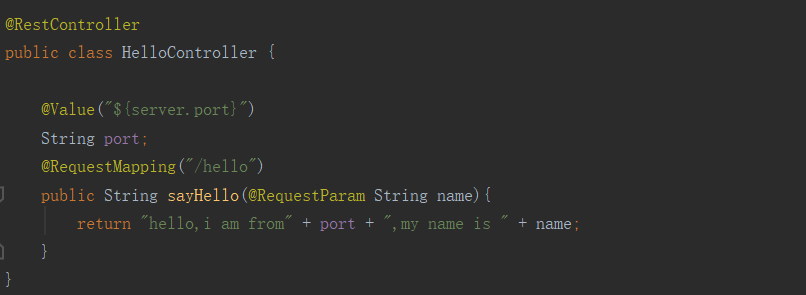
1. 配置application.properties



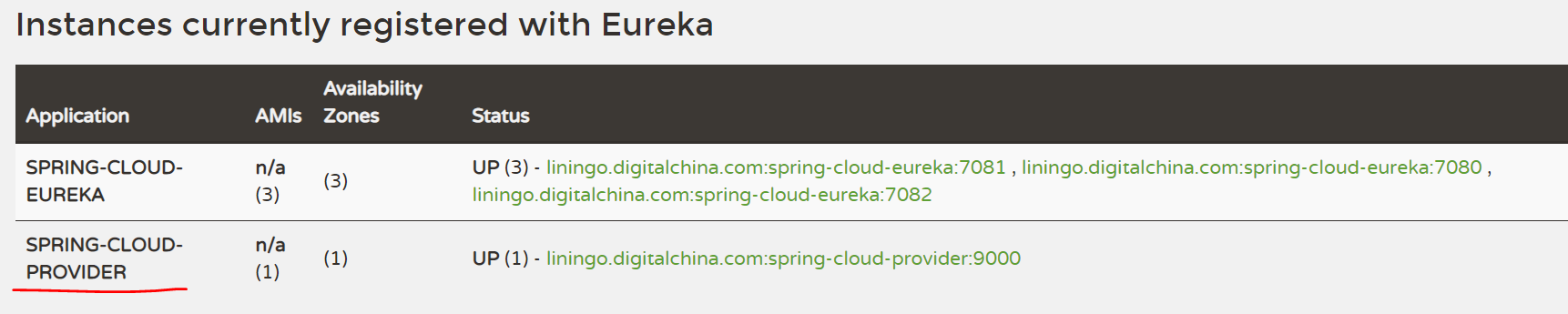
1. 添加@EnableDiscoveryClient到入口类。添加@EnableDiscoveryClient注解后，项目就具有了服务注册的功能。



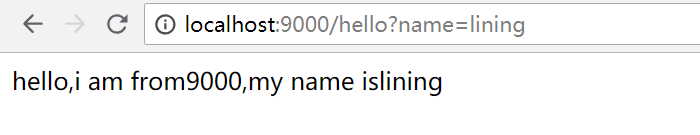
1. Controller



1. 查看是否注册到服务中心



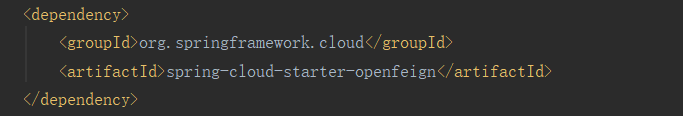
1. 访问<http://localhost:9000/hello?name=lining>结果



## Server Consumer

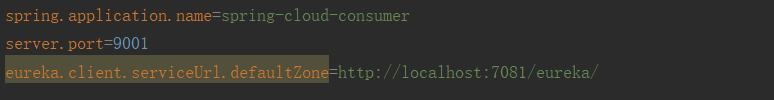
### 基本使用

1）pom依赖





2）配置application.properties

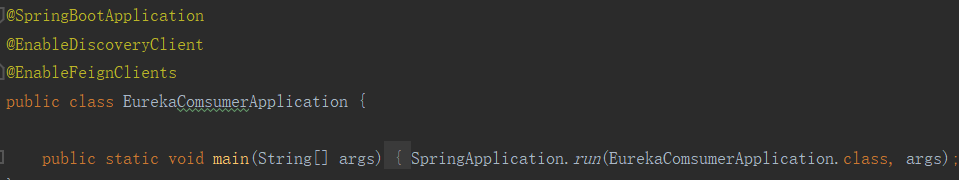


3）添加注解到启动类

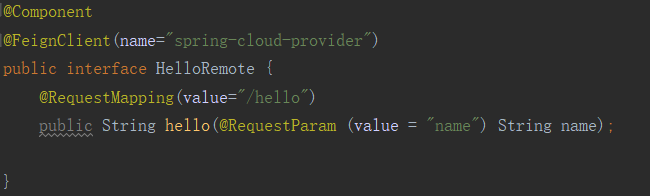
@EnableDiscoveryClient：启用服务注册与发现；

@EnableFeignClients: 启用feign进行远程调用

Feign是一个声明式Web Service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。



4）feign调用实现

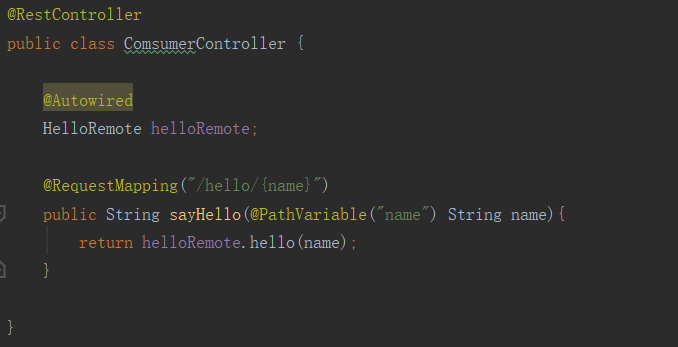


name:远程服务名，即服务提供者spring.application.name配置的名称。如果需要配置负载均衡，需要将多个同名的服务提供者注册到注册中心。

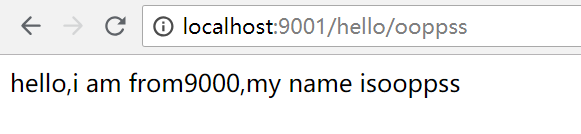
此类中的方法和远程服务中contoller中的方法名和参数需保持一致。

1. web层调用远程服务

将HelloRemote注入到controller层，像普通方法一样去调用即可。

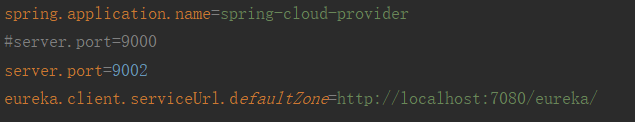


1. 测试

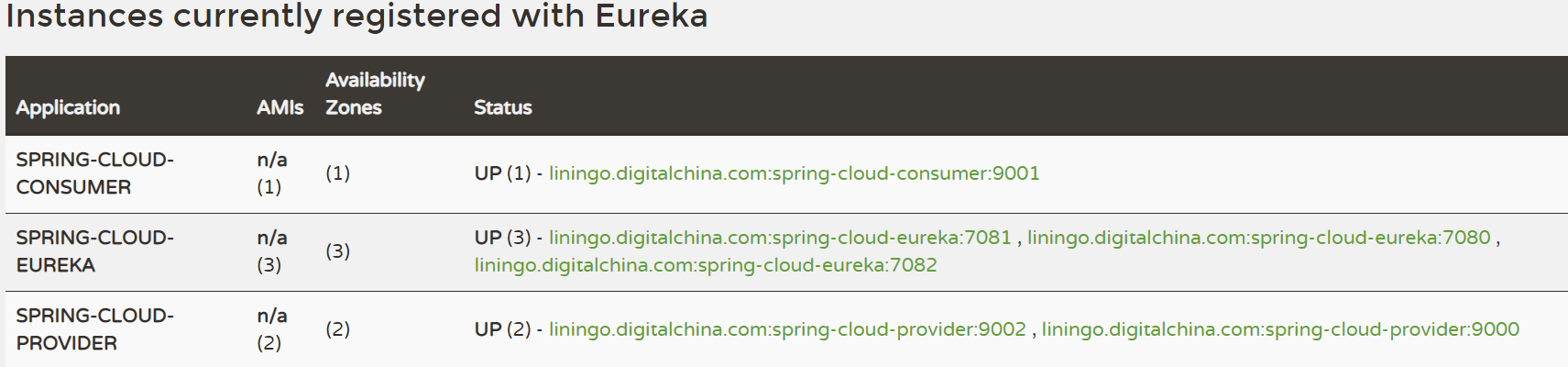


### 增加负载均衡功能

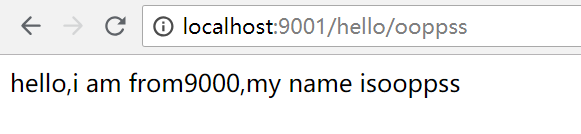
将配置文件的端口改为9002，再多启动一个Service Provider。

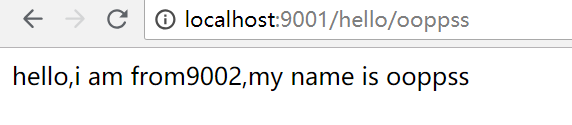


可以看到注册中心有两个服务提供者。



1）测试：<http://localhost:9001/hello/ooppss，不断刷新页面，发现会轮流访问端口9000>、9002。



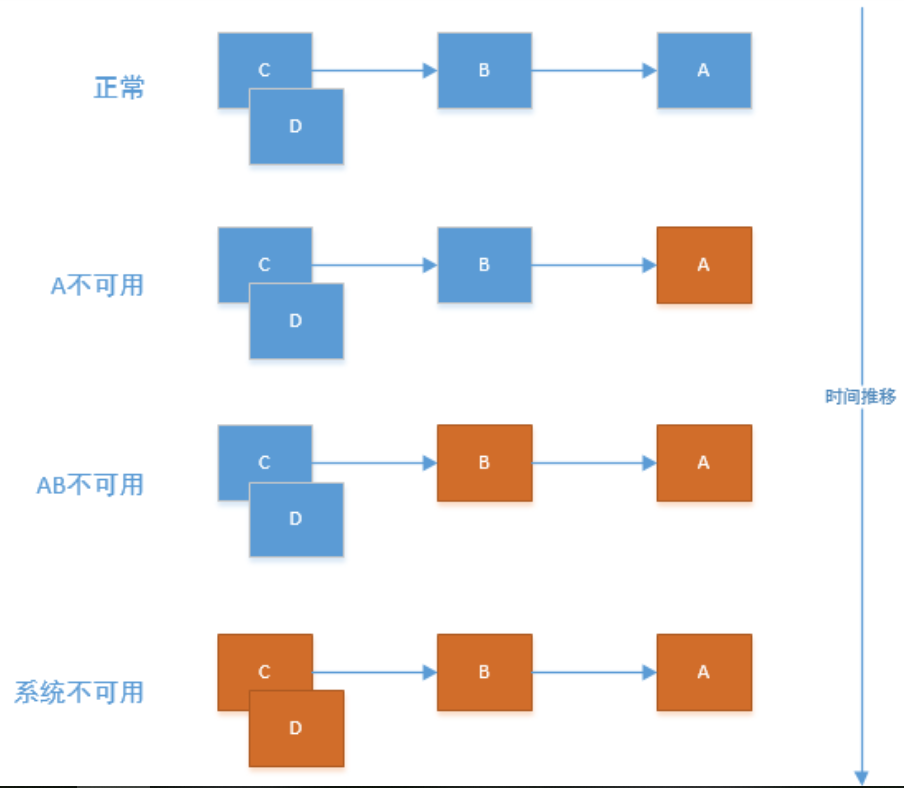


# 熔断器Hystrix

## 熔断器

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

如果下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。



### 熔断器特性

#### 1）熔断器机制

1. 当客户端请求后端服务失败率超过一定比例（默认50%）时，熔断器切换到开路（OPEN）状态，后续的后端请求会被熔断器截取并返回错误信息。这时候的请求是没有到达后端的。

2. 熔断器在一段时间后（默认五秒）会自动切换到半开路（HALF-OPEN）状态，判断下一个请求能否成功访问到后端服务，如果可以，熔断器切换到闭路状态（CLOSE），否则切换到开路状态。

3. 熔断器是后端服务的保险丝，并且拥有自我修复能力。如果发现后端服务不可用，它会直接截断客户端的请求，保护后端。如果发现后端服务恢复正常，它会恢复客户端的正常访问。

#### 2）Fallback

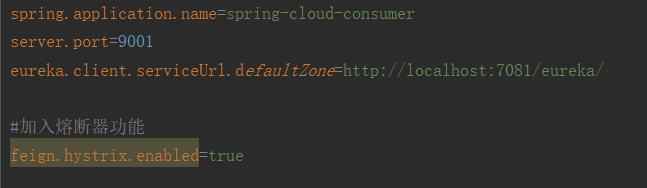
1. 要实现熔断器功能，需要在@FeignClient中指定熔断器的处理类。当熔断器处于开路状态时，会去调用该类的方法，返回给客户端。

#### 3）资源隔离策略

在Hystrix中, 主要通过线程池来实现资源隔离. 通常在使用的时候我们会根据调用的远程服务划分出多个线程池. 例如调用产品服务的Command放入A线程池, 调用账户服务的Command放入B线程池. 这样做的主要优点是运行环境被隔离开了. 这样就算调用服务的代码存在bug或者由于其他原因导致自己所在线程池被耗尽时, 不会对系统的其他服务造成影响. 但是带来的代价就是维护多个线程池会对系统带来额外的性能开销. 如果是对性能有严格要求而且确信自己调用服务的客户端代码不会出问题的话, 可以使用Hystrix的信号模式(Semaphores)来隔离资源.

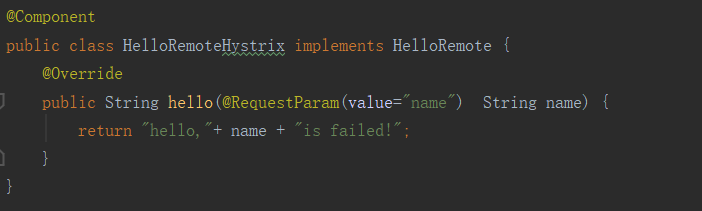
### 开启熔断器功能

在服务消费者端的配置文件内开启熔断器功能。



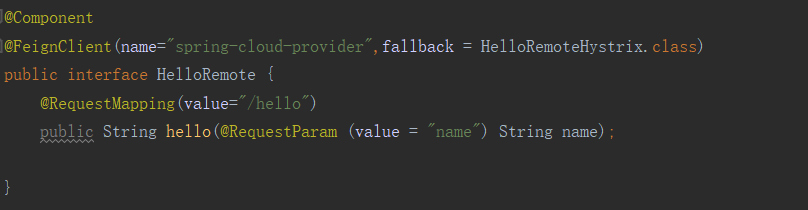
### 创建回调类

当熔断器处于OPEN状态时，会返回回调类里的对应方法。



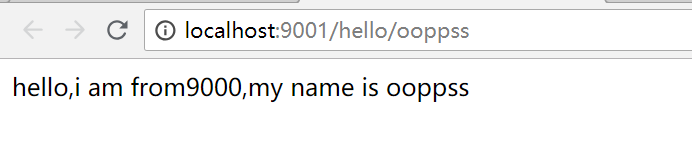
### 给远程调用服务添加fallback功能

在@FeignClient内配置fallback属性。

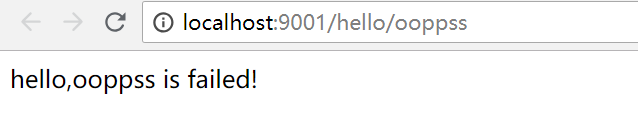


### 测试

1. 开启Service Provider服务时，访问<http://localhost:9001/hello/ooppss>，返回正常结果。



1. 关闭Service Provider服务后，再次访问<http://localhost:9001/hello/ooppss>，返回fallback方法定义的错误信息。



## 熔断器监控Hystrix Dashboard

使用Hystrix Dashboard对熔断器进行监控，可以很直观的得到当前服务的监控状态。Hystrix Dashboard统计了成功率、最近十秒的错误比例、请求频率等。

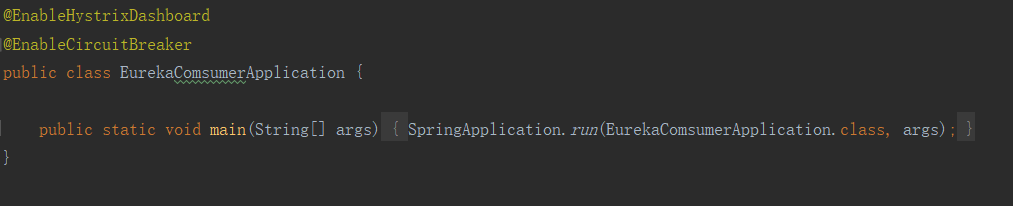
### 添加pom依赖



### 启动类添加注解

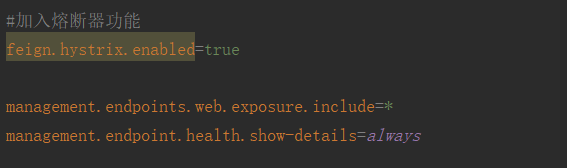
在启动类添加@EnableCircuitBreaker开启断路器功能。

在启动类添加@EnableHystrixDashboard开启断路器监控功能。

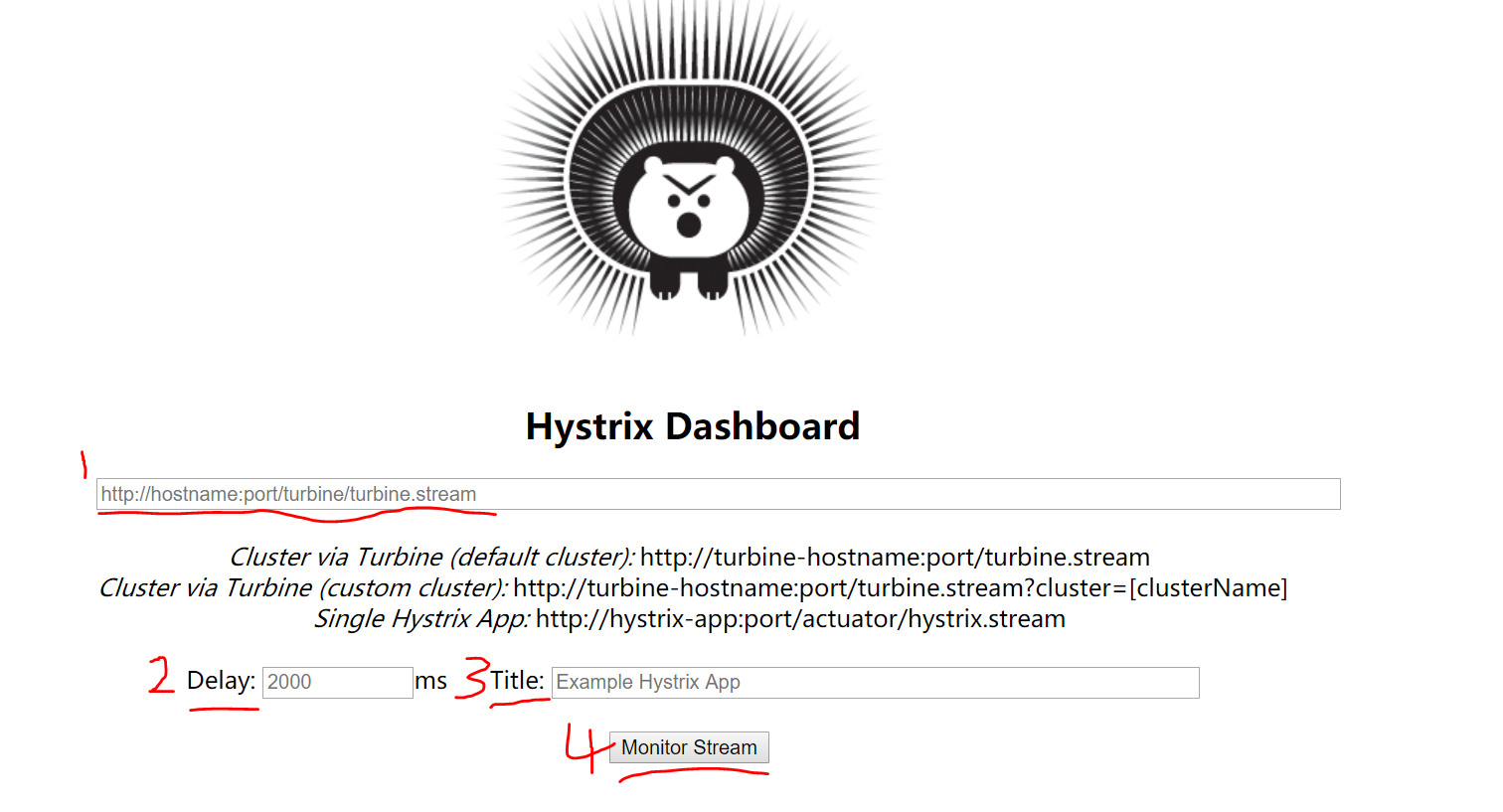


### 配置文件

如果使用的Spring boot 为 2.0 +，为了安全，默认 Actuator 只暴露了2个端点，heath 和 info。我们还要配置暴露所有端点，配置文件新增配置：



### 访问<http://localhost:9001/hystrix>



通过Hystrix Dashboard主页面的文字介绍，我们可以知道，Hystrix Dashboard共支持三种不同的监控方式：

1.默认的集群监控：通过URL:http://turbine-hostname:port/turbine.stream开启，实现对默认集群的监控。

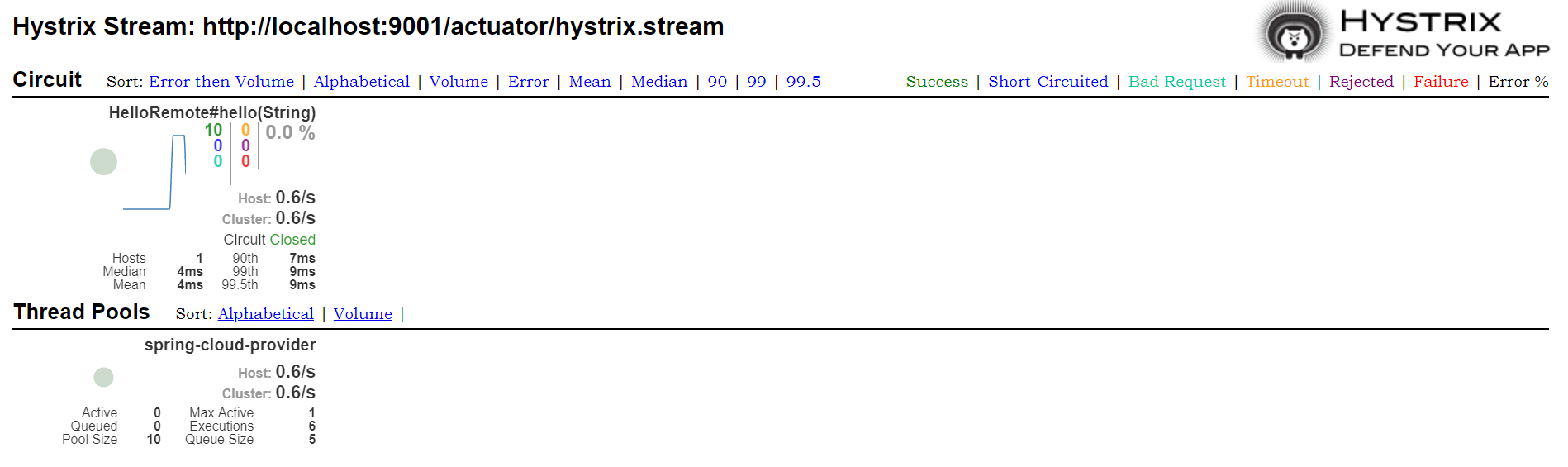
2.指定的集群监控：通过URL:http://turbine-hostname:port/turbine.stream?cluster=[clusterName]开启，实现对clusterName集群的监控。

3.单体应用的监控：通过URL:http://hystrix-app:port/hystrix.stream开启，实现对具体某个服务实例的监控。

4.Delay：控制服务器上轮询监控信息的延迟时间，默认为2000毫秒，可以通过配置该属性来降低客户端的网络和CPU消耗。

5.Title:该参数可以展示合适的标题。

在1处输入监控地址，在2处输入延迟时间，在3处输入标题，最后点击4进入监控页面：



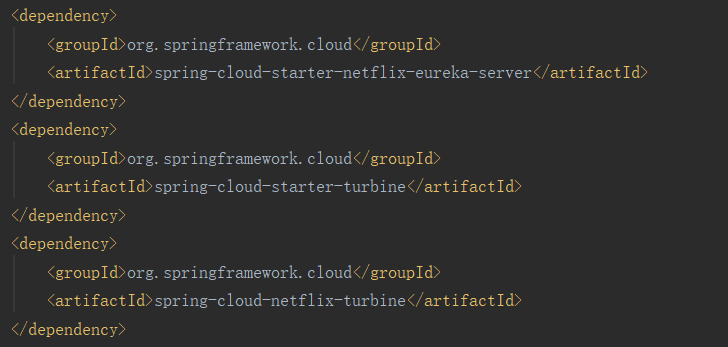
## 监控多个具有断路由功能的节点Turbine

由于网上多数使用1.x版本，所以有些配置并不适用本版本（2.x），仅此说明。

### 新建SpringBoot项目TurbineApplication

#### 1）pom依赖

除了Server Comsumer使用的jar包以外，还需要一下jar包：



#### 2）入口类配置

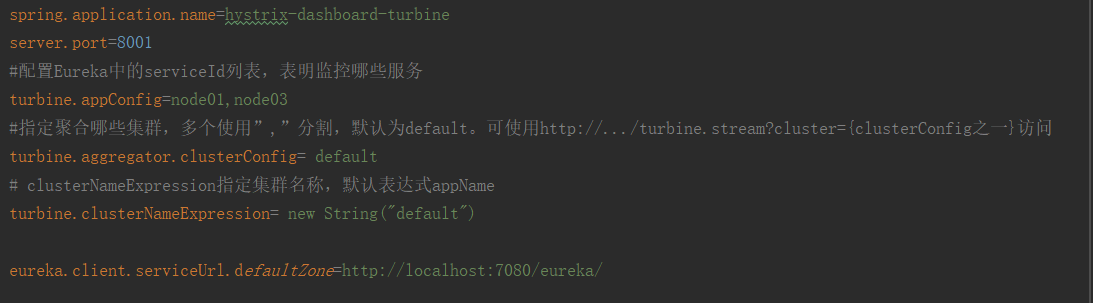
由于需要发现其它节点，所以需要引入@EnableEurekaServer注解。

由于需要注册到注册中心，所以需要@EnableDiscoveryClient注解

如果需要通过Turbine打开小熊界面，需要@EnableHystrixDashboard和@EnableClientBreaker注解。



#### 3）配置文件

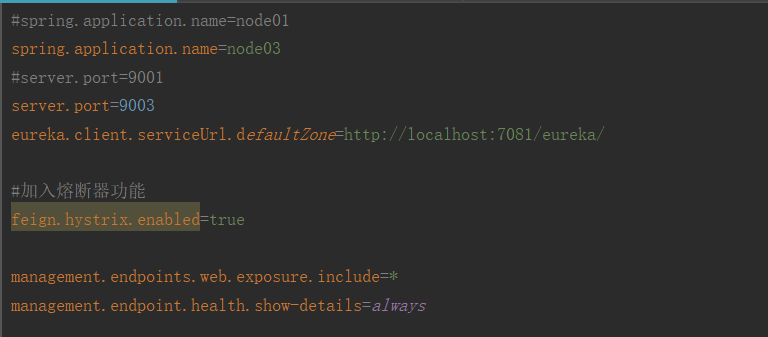


### 更改ServerComsumer

需要启动两个服务消费者，配置如下。

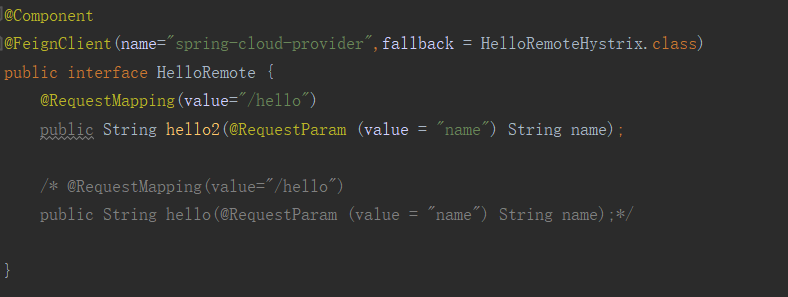
#### 1）配置文件

分别创建node01、node03：



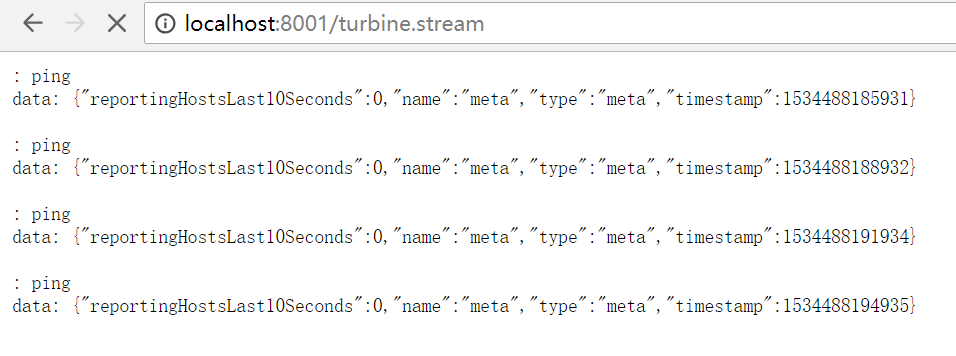
#### 2）修改远程服务调用接口

Node01的方法名为hello；node02的方法名为hello2。其它地方需要做相应修改。



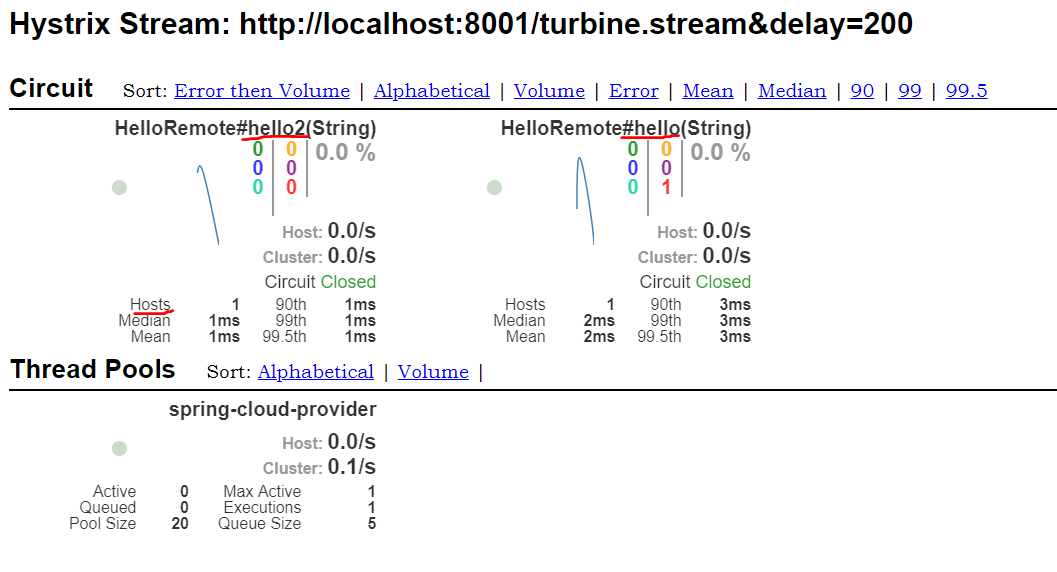
### 访问<http://localhost:8001/turbine.stream>

返回页面如图：



### 访问<http://localhost:8001/hystrix>

在出现的小熊界面输入监控地址：<http://localhost:8001/turbine.stream>，返回界面如下。通过比较发现，监控集群和监控单个实例是差不多的。



# 配置中心

## 配置中心具备的能力

1）提供服务端和客户端支持

2）集中管理各环境的配置文件

3）配置文件修改之后，可以快速的生效

4）可以进行版本管理

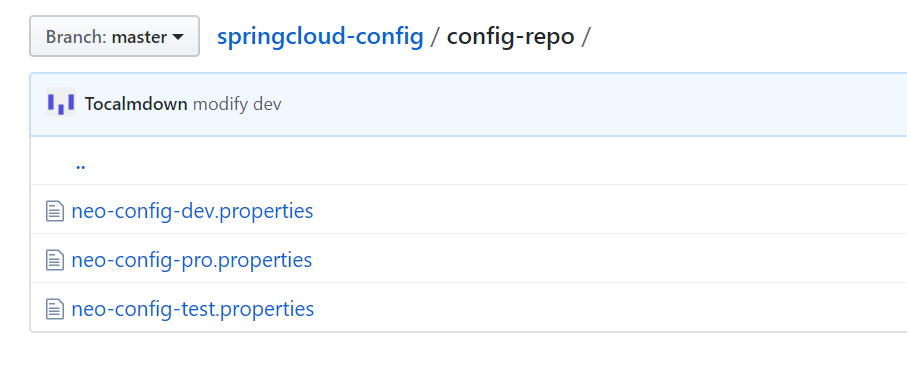
5）支持大的并发查询

6）支持各种语言

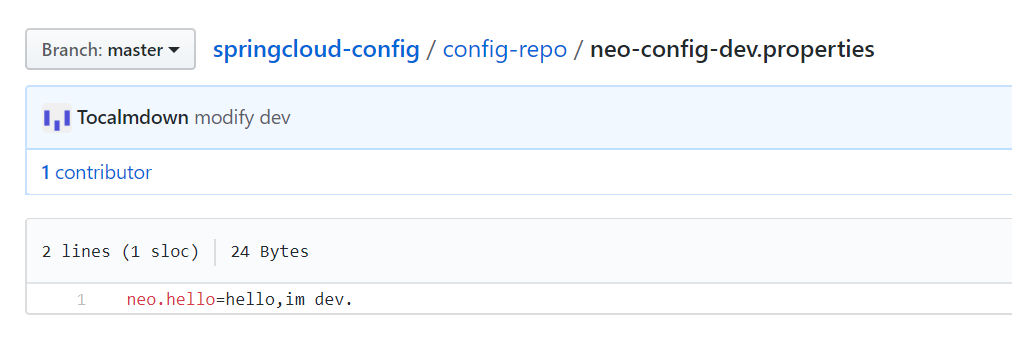
Spring Cloud是配置文件管理的解决方案。配置中心分为服务端和客户端，服务端保存配置文件，并通过接口（url和文件名的映射关系）向客户端提供数据支持。客户端拿到配置文件数据后可以初始化自己的应用。Spring Cloud可以将配置文件放在本地、git或svn。

## 服务端

### 在git上创建文件



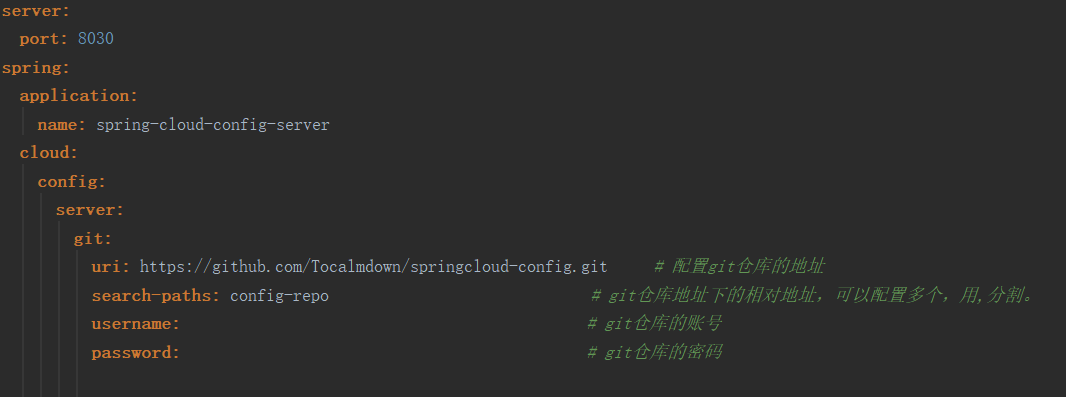
其中neo-config-dev.properties下的配置为下图，其它两个文件以此类推。



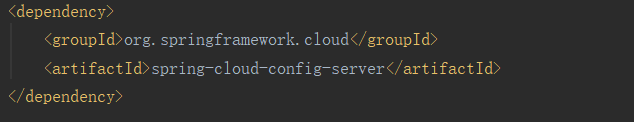
### 配置文件

Uri：git仓库地址

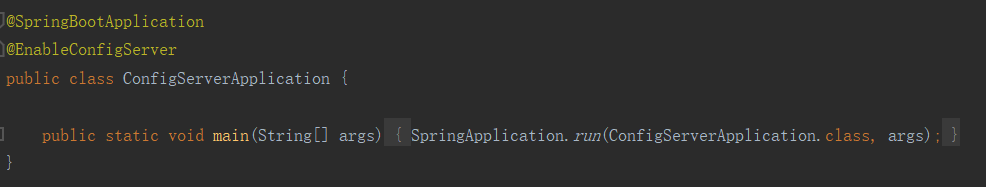
Search-paths：config-repo，表示从该仓库的config-repo文件夹下找（根据url地址映射到某个具体文件）。



### pom依赖

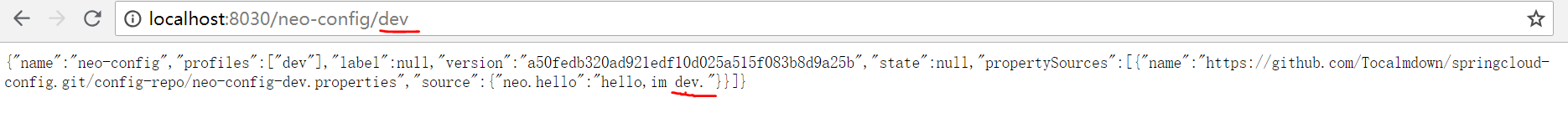


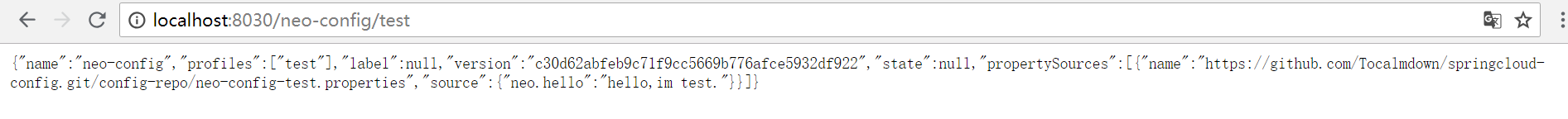
### 入口类开启配置中心服务端注解



### 测试

url的neo-config+dev对应neo-config-dev文件。



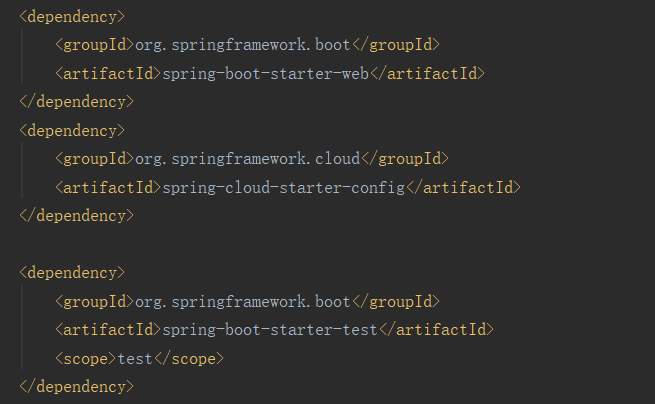


### url和文件名的对应规则

1. /{application}/{profile}[/{label}]
2. /{application}-{profile}.yml
3. /{label}/{application}-{profile}.yml
4. /{application}-{profile}.properties
5. /{label}/{application}-{profile}.properties

## 客户端

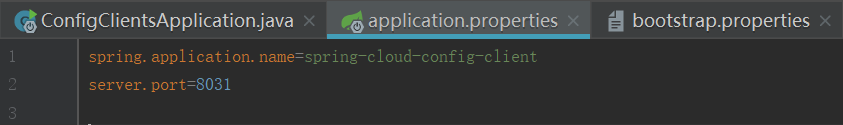
### 添加pom依赖

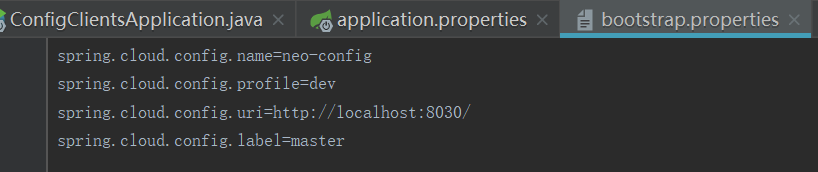


### 配置文件

需要配置两个配置文件，一个application.properties,提供接口给浏览器访问。一个是bootstrap.properties,提供接口给config-client访问。

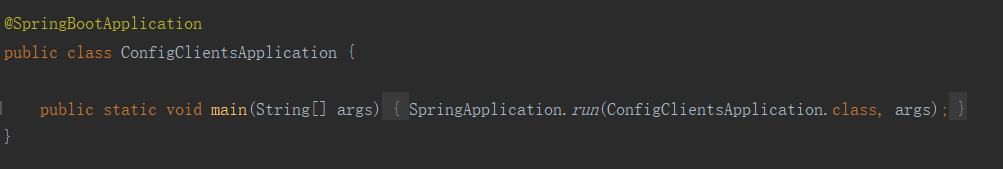
配置中心服务端的接口必须写在bootstrap.properties下，因为它会先application.properties执行。



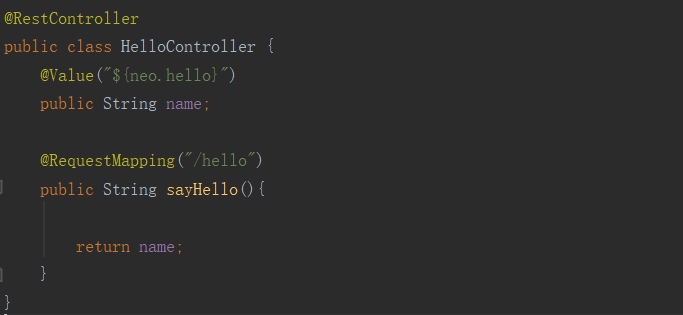


### 启动类

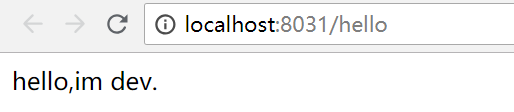
仅需要一个注解。



### Controller



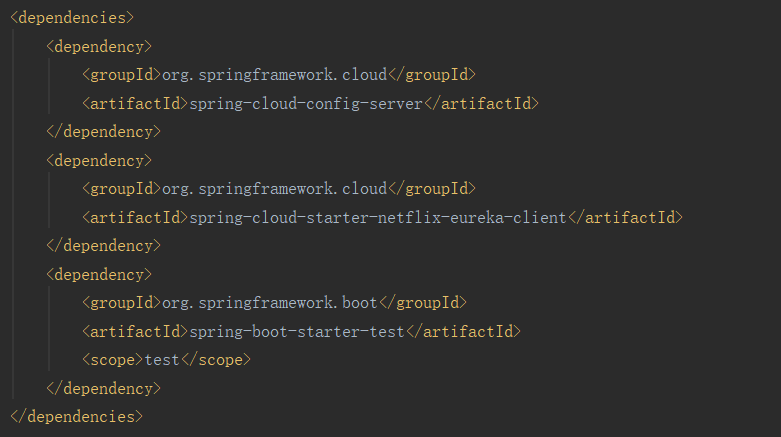
### 测试



# 高可用的配置中心

## 改造配置中心服务端

### 修改pom

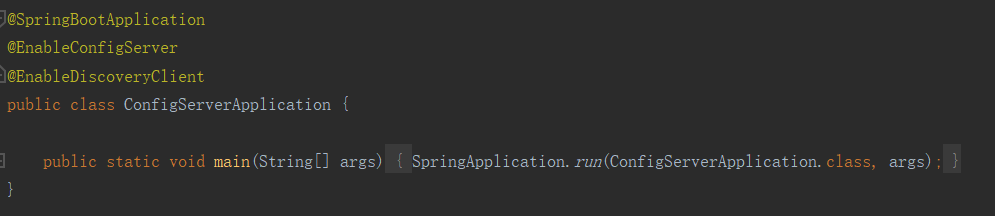


### 修改application.yml

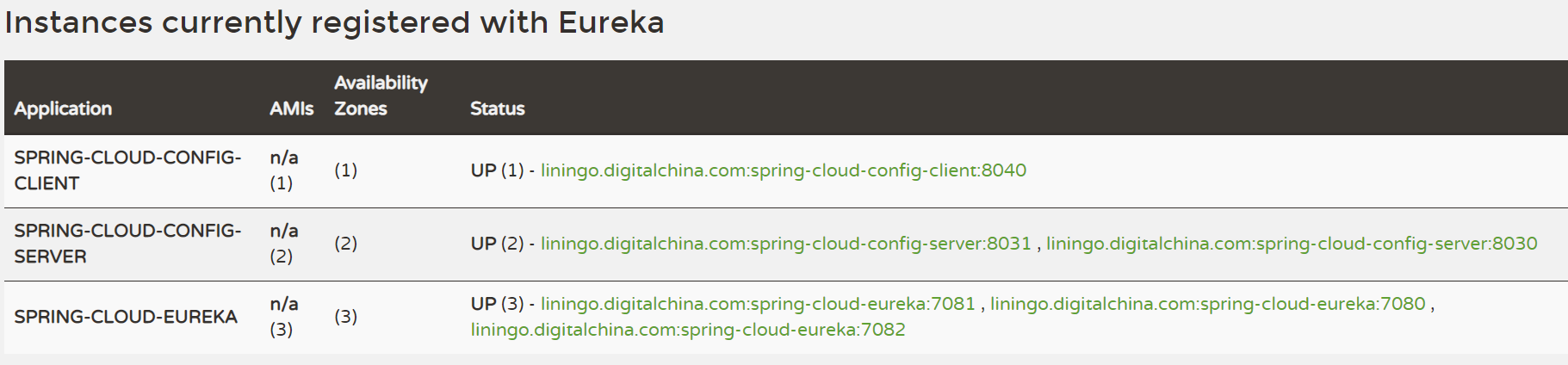
为验证高可用，启动8030和8031两个端口的配置中心服务器。



### 入口类注解

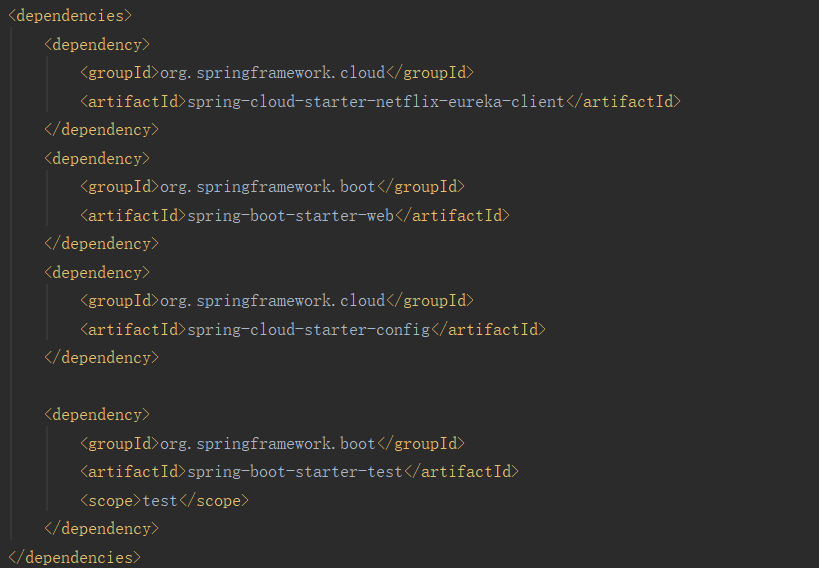


### 验证注册到注册中心



## 修改配置中心客户端

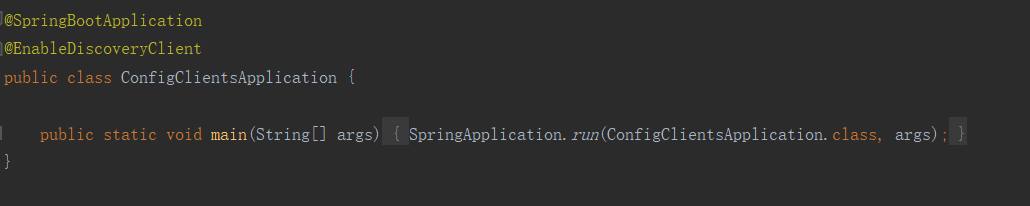
### 配置pom依赖



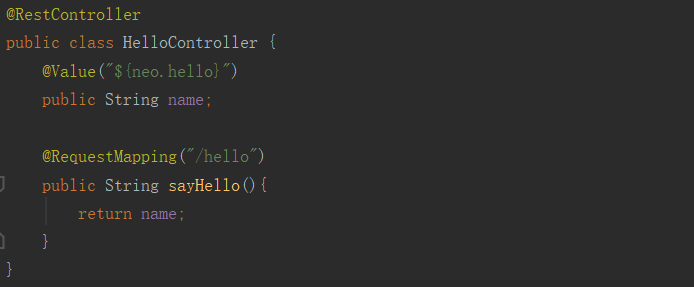
### 修改bootstrap.properties



### 入口类注解

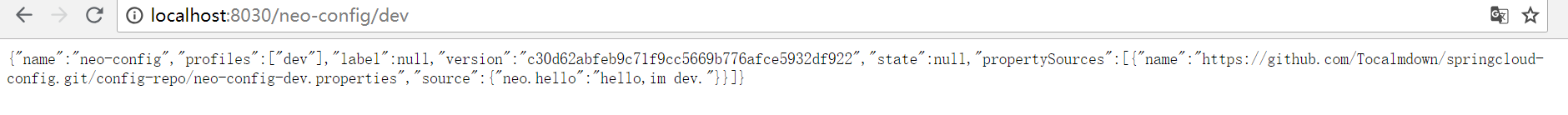


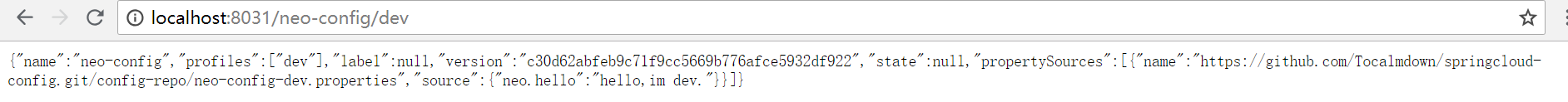
### Controller类

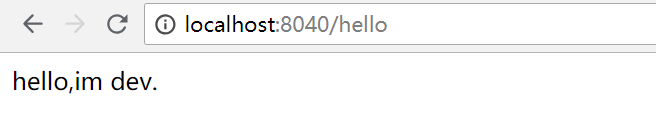


### 测试

测试配置中心服务端连接git是否正常、配置中心客户端是否成功连接服务端。并且当关掉8030端口的服务端时，客户端依然可以成功从配置文件拿值，说明高可用。







## Webhook

当我们修改配置文件后，再次去访问，发现获得的值是更新前的，这是因为springboot项目只有在启动的时候才会获取配置文件的值，修改github信息后，client端并没有再次去获取。这时候需要有人告诉我们，配置信息发生更新了。这个人就是git的webhook机制。

# 配置中心和消息总线

如果需要客户端获取到最新的配置信息需要执行refresh，我们可以利用webhook的机制每次提交代码发送请求来刷新客户端，当客户端越来越多的时候，需要每个客户端都执行一遍，这种方案就不太适合了。使用Spring Cloud Bus可以完美解决这一问题。