# Spring Cloud 入门

## 参考文档

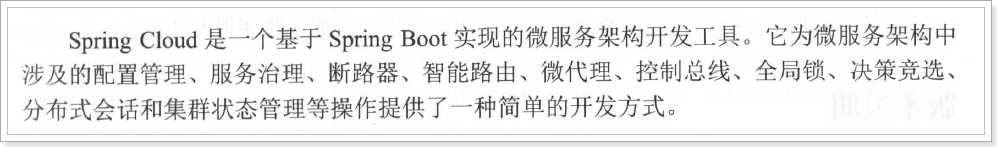
Spring Cloud项目的官方网址

<http://projects.spring.io/spring-cloud/>

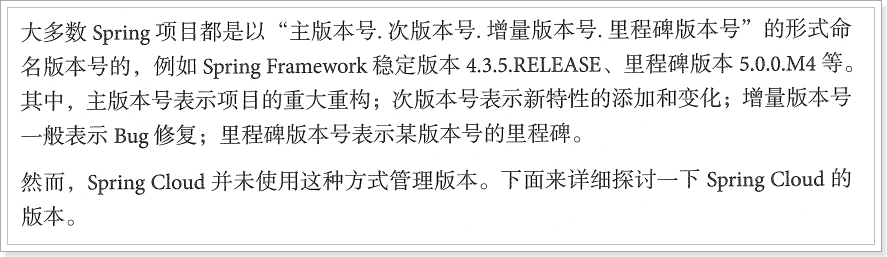
Spring Cloud Dalston 中文文档 参考手册 中文版

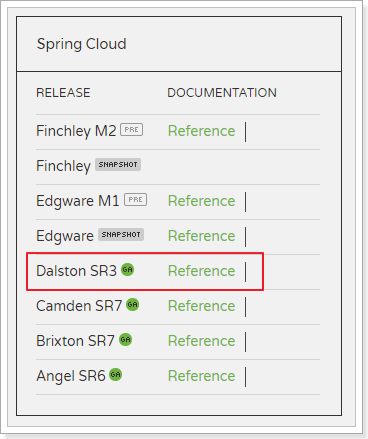
<https://springcloud.cc/spring-cloud-dalston.html>

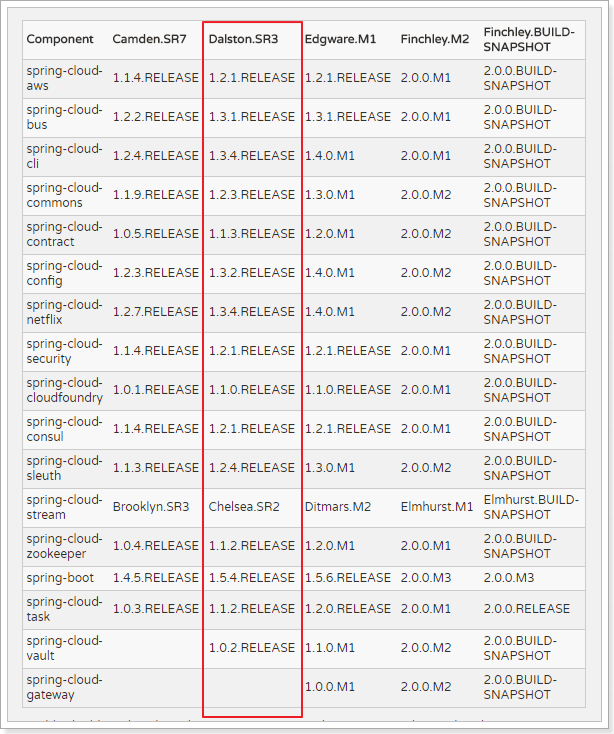
## Spring Cloud简介



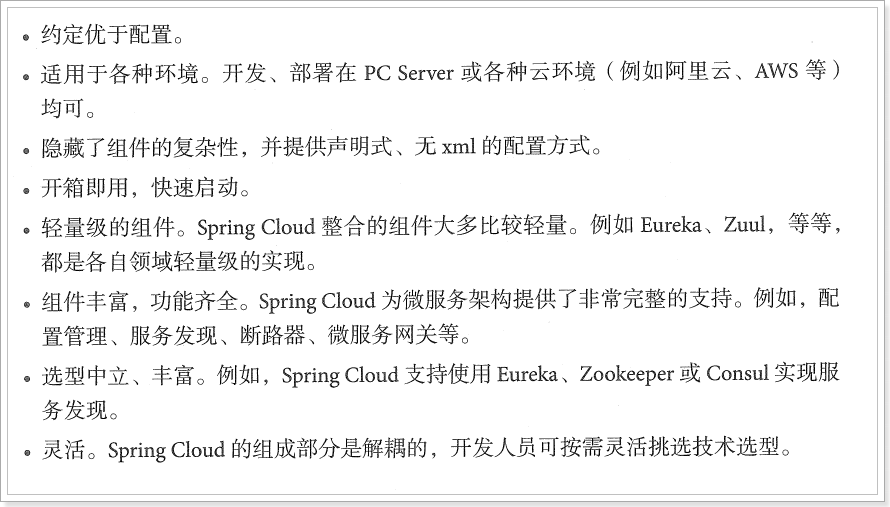
### 关于版本号





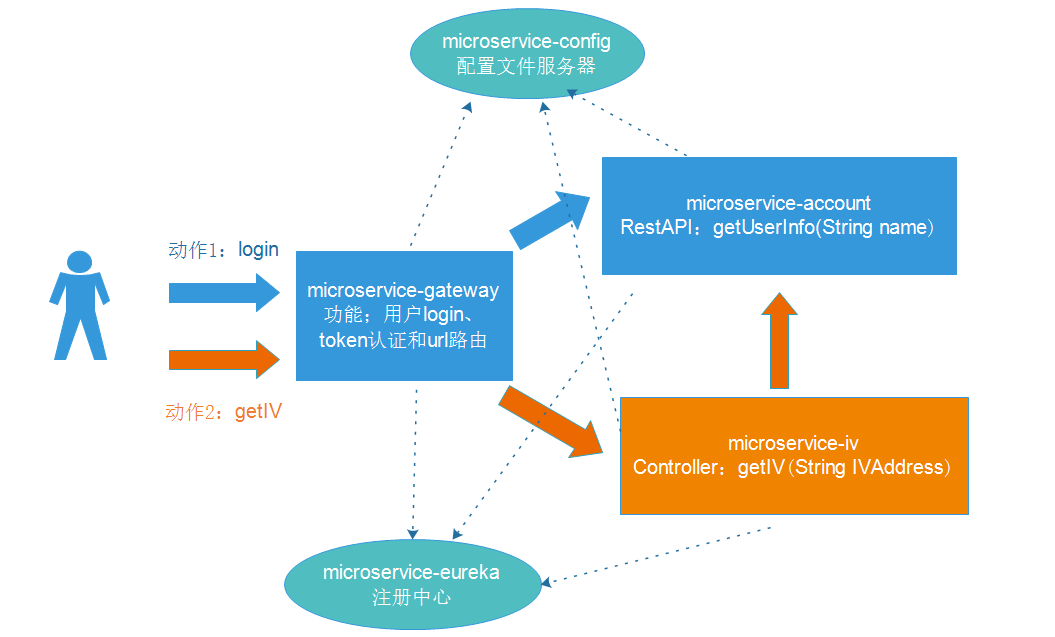


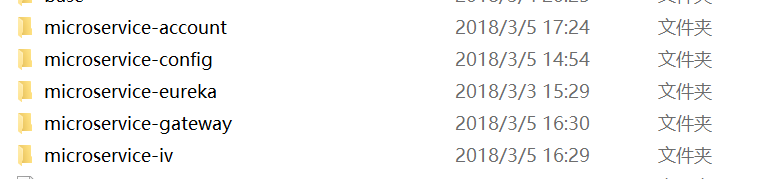
### Spring Cloud框架特点



## 使用Spring Boot实现微服务

### 业务架构





### 环境准备：mysql、redis、rabbitmq



## Eureka注册与发现组件

### 分析硬编码的问题

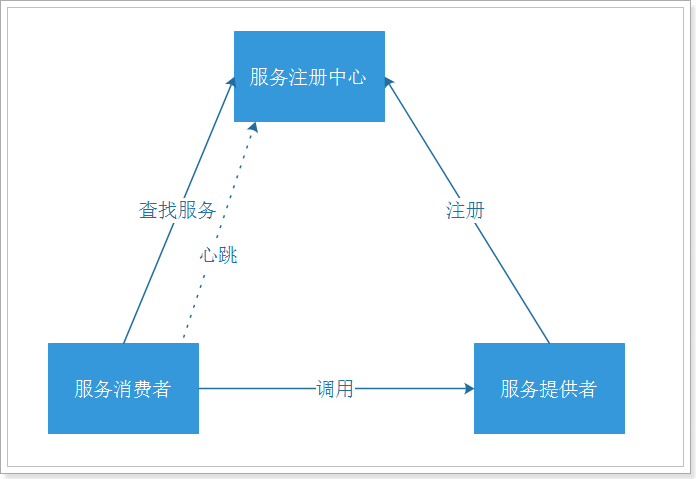
传统的HTTPClinet需要把REST接口的url写在配置文件中，但是我们想想：

1. 如果商品微服务的ip地址发生了变更，订单微服务中的配置文件也需要跟着修改
2. 如果商品微服务有多个，那么在订单微服务中又该如何写地址？

那应该怎么解决呢？ -- 通过服务注册、发现的机制来完成。

### 微服务注册与发现

原理示意图：

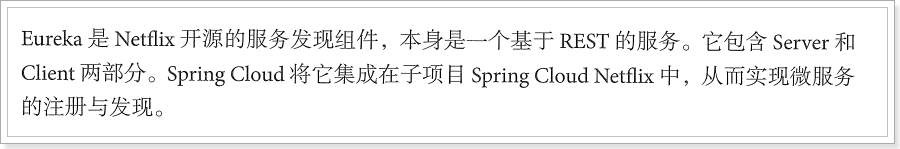


由上图可以看出：

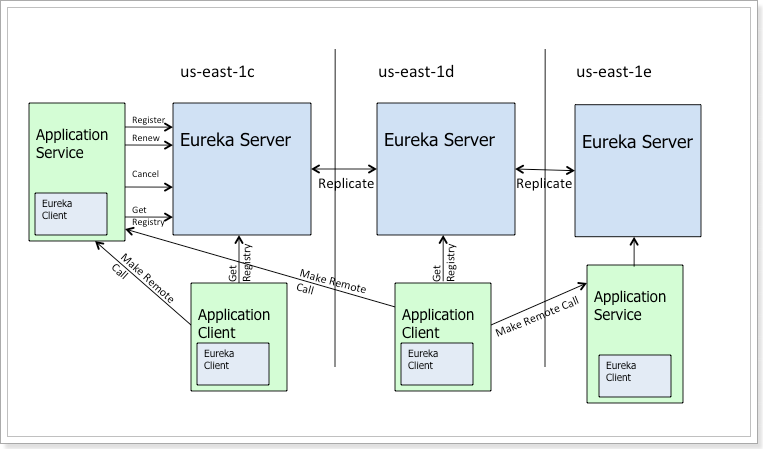
1. 服务提供者将服务注册到注册中心
2. 服务消费者通过注册中心查找服务
3. 查找到服务后进行调用（这里就是无需硬编码url的解决方案）
4. 服务的消费者与服务注册中心保持心跳连接，一旦服务提供者的地址发生变更时，注册中心会通知服务消费者

### 注册中心Eureka

Spring Cloud提供了多种注册中心的支持，如：Eureka、ZooKeeper等。推荐使用Eureka。



### 原理



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Eureka Server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。

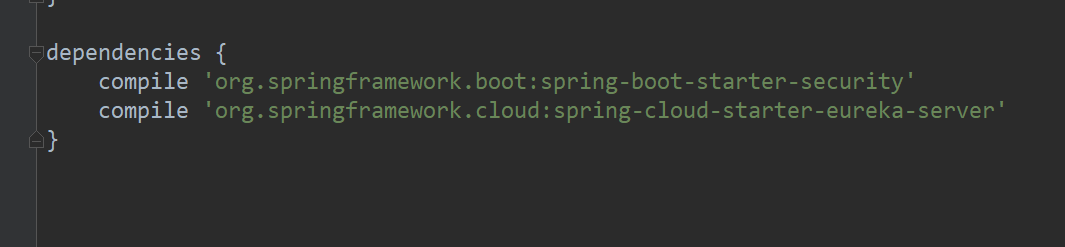
Eureka Client是一个java客户端，用于简化与Eureka Server的交互，客户端同时也就别一个内置的、使用轮询(round-robin)负载算法的负载均衡器。

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，Eureka Server将会从服务注册表中把这个服务节点移除(默认90秒)。

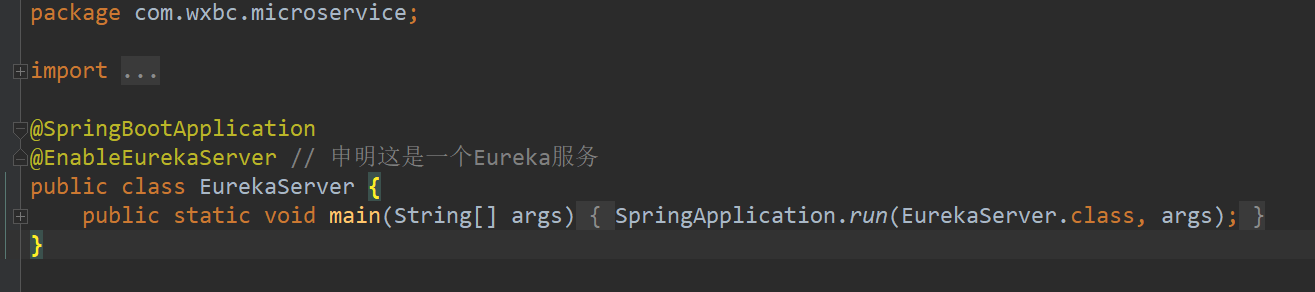
Eureka Server之间通过复制的方式完成数据的同步，Eureka还提供了客户端缓存机制，即使所有的Eureka Server都挂掉，客户端依然可以利用缓存中的信息消费其他服务的API。综上，Eureka通过心跳检查、客户端缓存等机制，确保了系统的高可用性、灵活性和可伸缩性。

### 编写Eureka Server

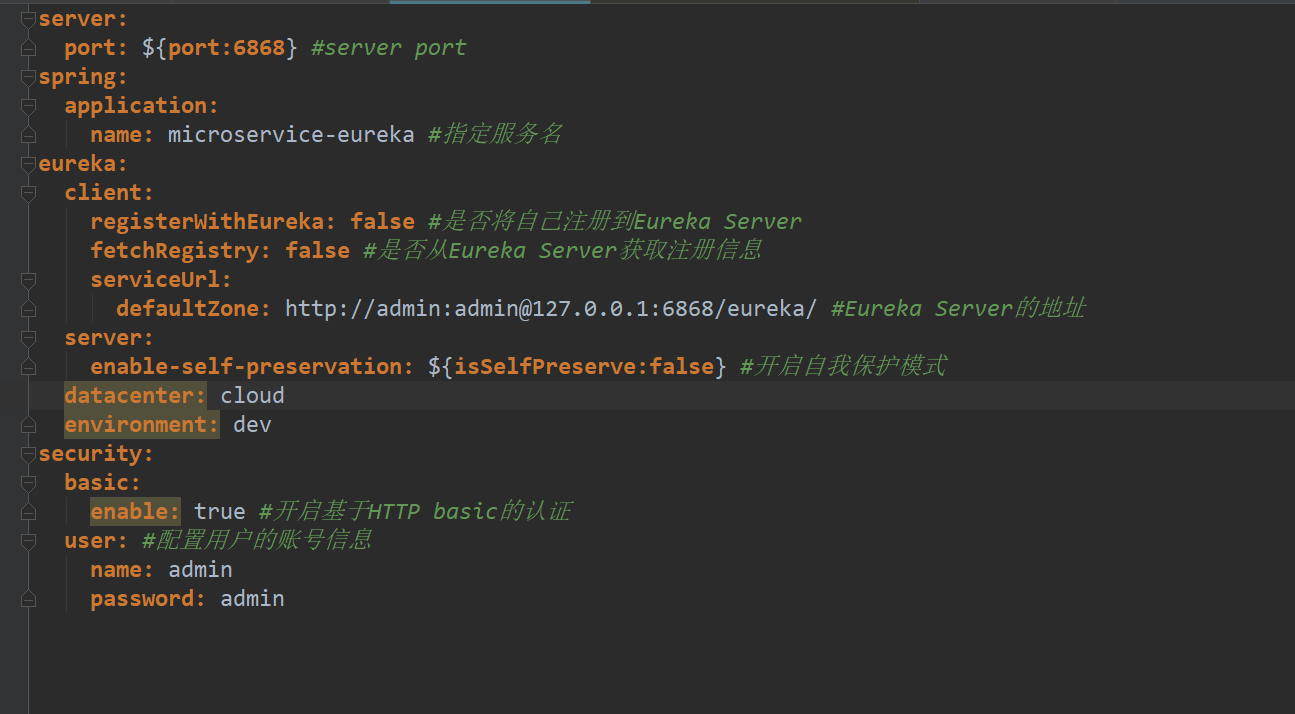
引入依赖



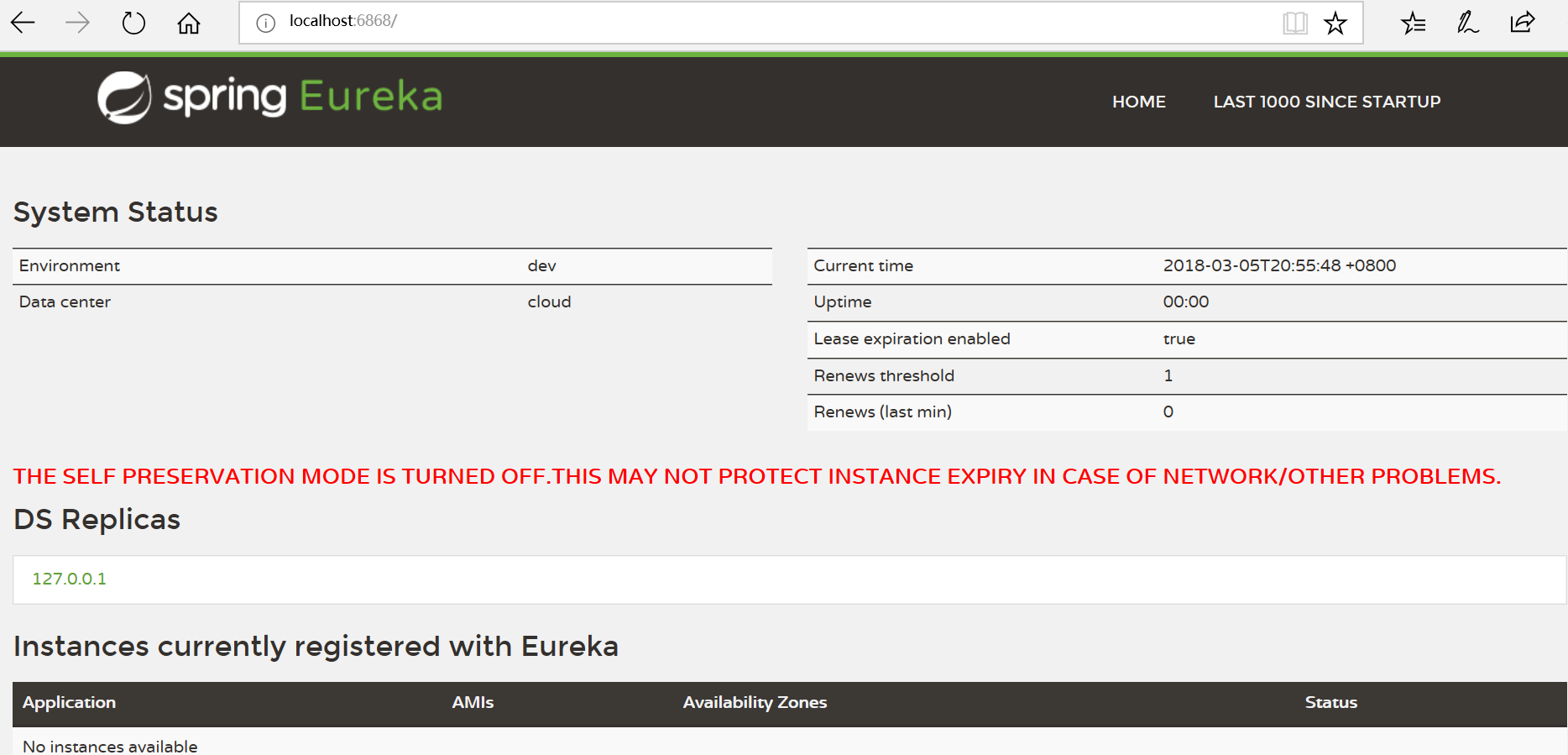
编写程序入口



编写Application.yml



管理信息页面

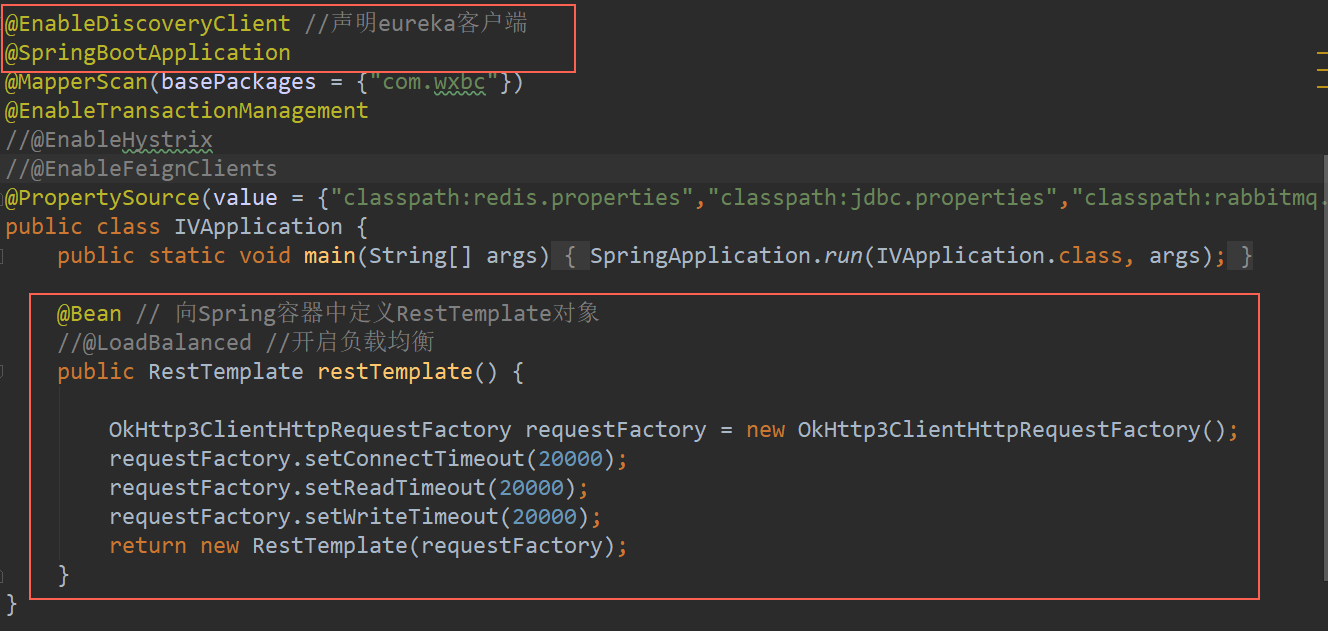


### IV系统从Eureka发现服务

引入依赖



编写入口，注入restTemplate



编写配置文件



### Account系统注册Eureka发现服务





### 测试请求

启动microservice-account和microservice-iv

发起请求<http://localhost:8081/iv/getIV/123456>

获取到serviceID对应的URL



获得结果



### 配置Eureka集群

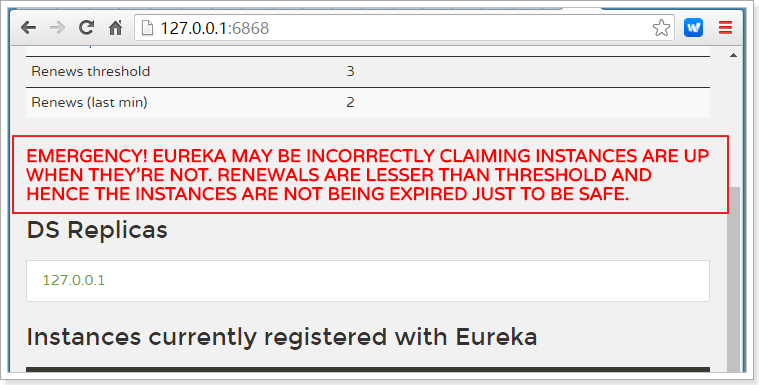
Eureka服务端配置：

**server:  
 port:** ${**port:6868**} *#server port***spring:  
 application:  
 name:** microservice-eureka *#指定服务名***eureka:  
 client:  
 registerWithEureka: true** *#是否将自己注册到Eureka Server* **fetchRegistry: true** *#是否从Eureka Server获取注册信息* **serviceUrl:  
 defaultZone:** http://admin:admin@127.0.0.1:6867/eureka/ *#把自己注册给另一个eureka* **server:  
 enable-self-preservation:** ${**isSelfPreserve:false**} *#开启自我保护模式* **datacenter:** cloud  
 **environment:** dev  
**security:  
 basic:  
 enable:** true *#开启基于HTTP basic的认证* **user:** *#配置用户的账号信息* **name:** admin  
 **password:** admin

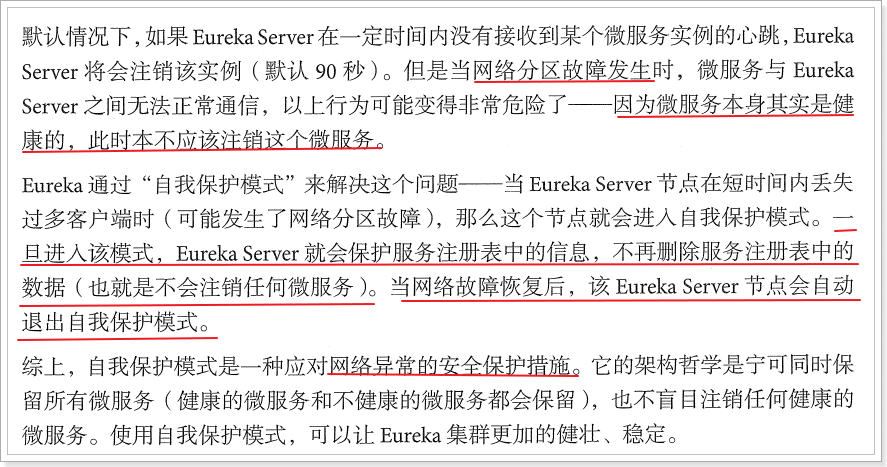
Eureka客户端配置：

**eureka:  
 client:  
 registerWithEureka: true** *#是否将自己注册到Eureka服务中，默认为true* **fetchRegistry: true** *#是否从Eureka中获取注册信息，默认为true* **serviceUrl:** *#Eureka客户端与Eureka服务端进行交互的地址* **defaultZone:** http://admin:admin@127.0.0.1:6868/eureka/ ,http://admin:admin@127.0.0.1:6867/eureka/  
**instance:  
 prefer-ip-address: true** *#将自己的ip地址注册到Eureka服务中* **ip-address:** 127.0.0.1 *#指定IP地址* **instance-id:** ${**spring.application.name**}:${**server.port**} *#指定实例id*

### Eureka的自我保护模式



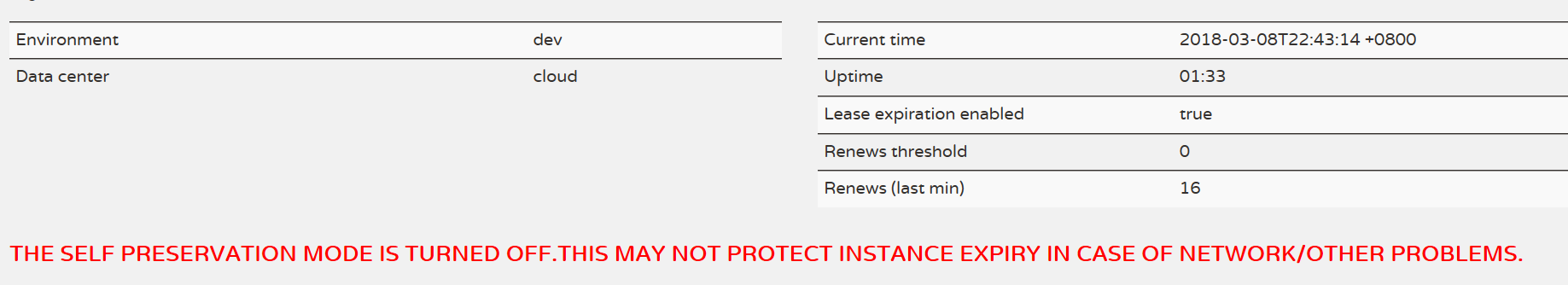
如图，当前Eureka进入了自我保护模式。



所以，一般进入自我保护模式，无需处理。如果，需要禁用自我保护模式，只需要在配置文件中添加配置即可：

**eureka:  
 server:  
 enable-self-preservation:** ${**isSelfPreserve:false**} *#开启自我保护模式*

重新启动服务查看效果：



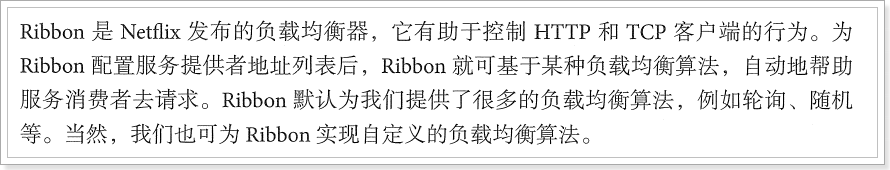
## 使用Ribbon实现负载均衡

如果为同一个的提供者在Eureka中注册了多个服务，那么客户端该如何选择服务呢？

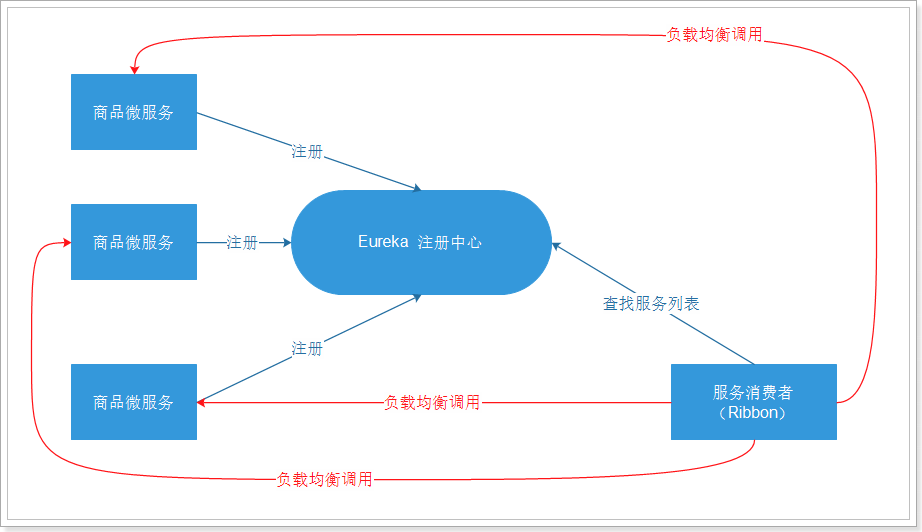
这时，就需要在客户端实现服务的负载均衡。

在Spring Cloud中推荐使用Ribbon来实现负载均衡。

### Ribbon简介



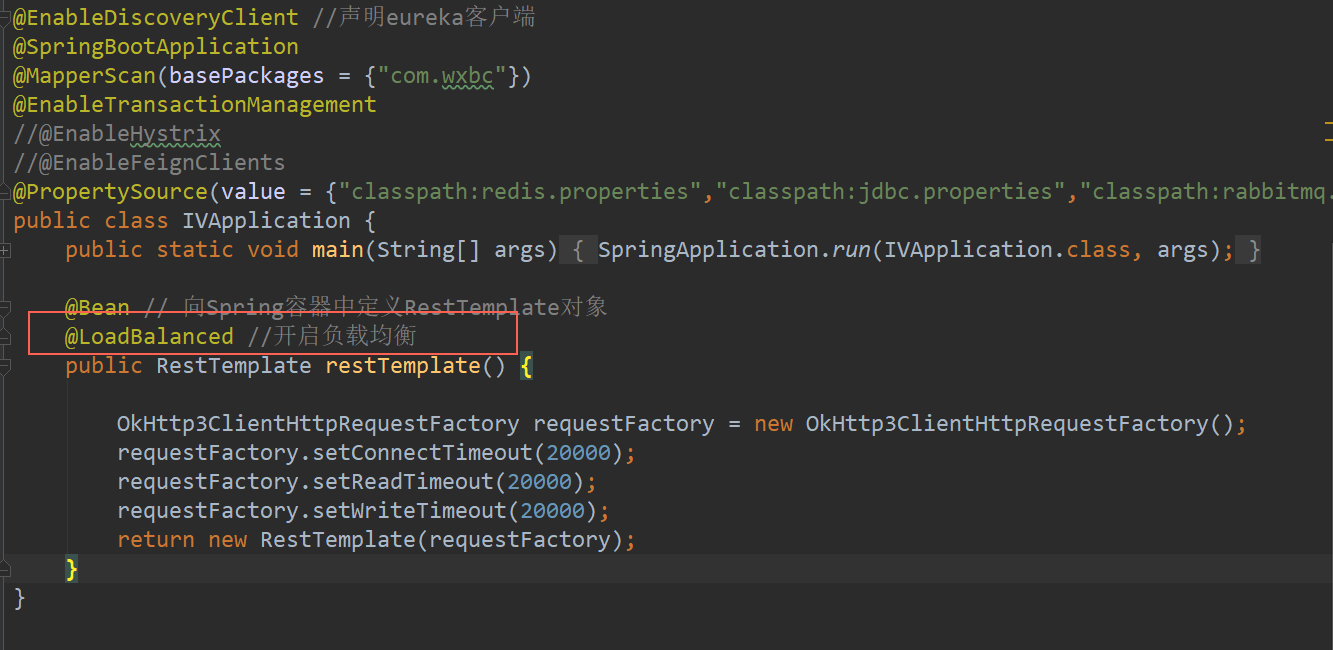
### 架构



### 开始使用Ribbon

该依赖是可以省略的，因为>spring-cloud-starter-eureka-server中已经包含了spring-cloud-starter-ribbon

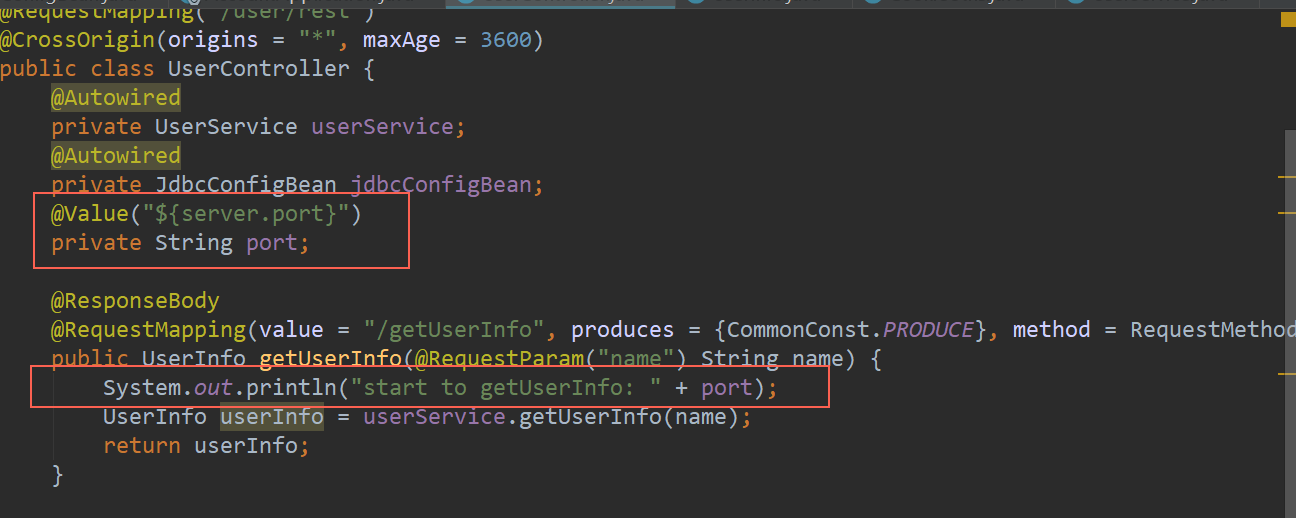
#### 为RestTemplate设置@LoadBalanced注解



#### 改造IVService的实现



#### 在getUserInfo接口中打印microservice-account 的端口号



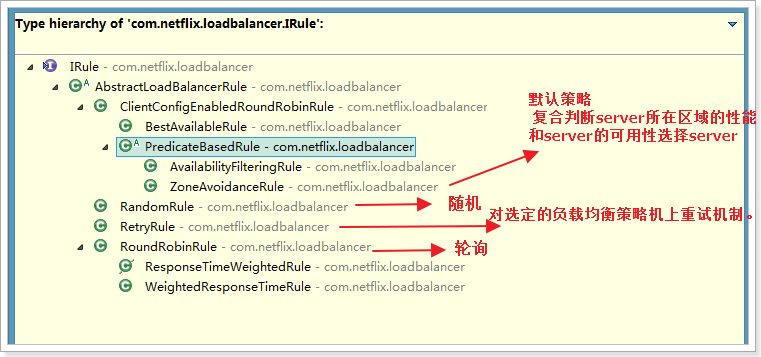
#### 启动两台microservice-account 端口号8182，8181

发起请求<http://localhost:8081/iv/getIV/123456>

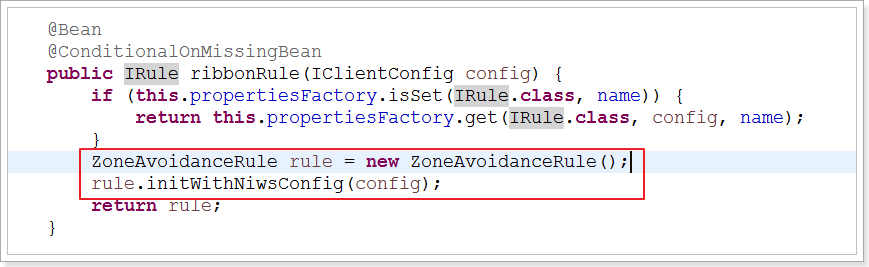
查看两台microservice-account控制台轮询打印结果

### 负载均衡策略

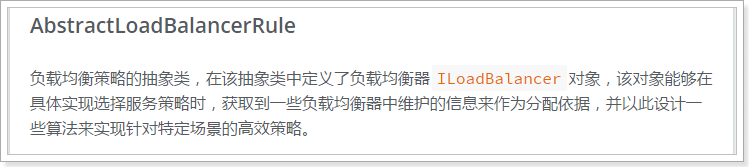
接口：com.netflix.loadbalancer.IRule，其实现类：

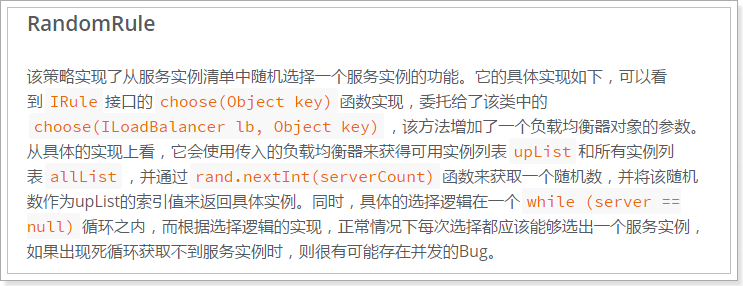


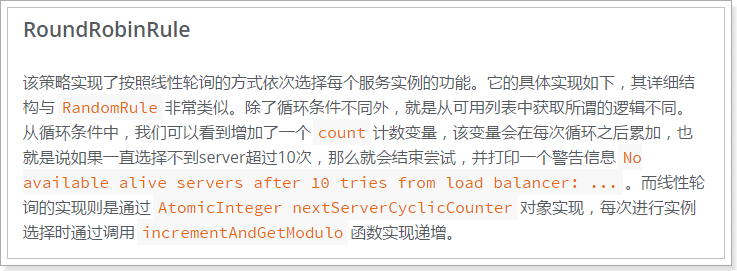
默认策略：



其它策略：









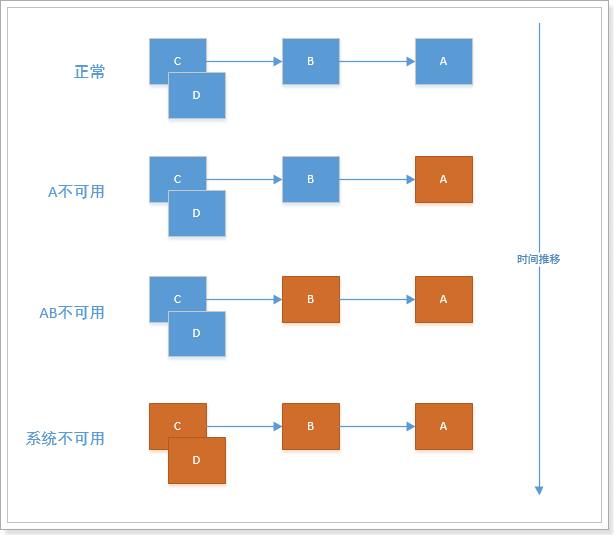
建议：采用默认策略即可。

## 容错保护：Hystrix

### 雪崩效应

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。

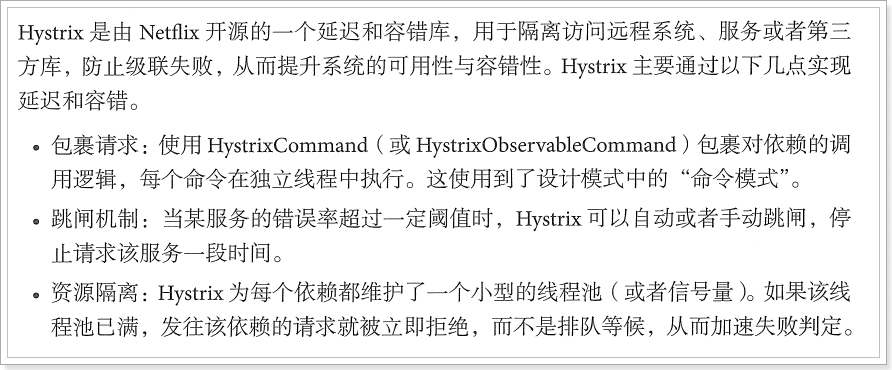
如果下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。

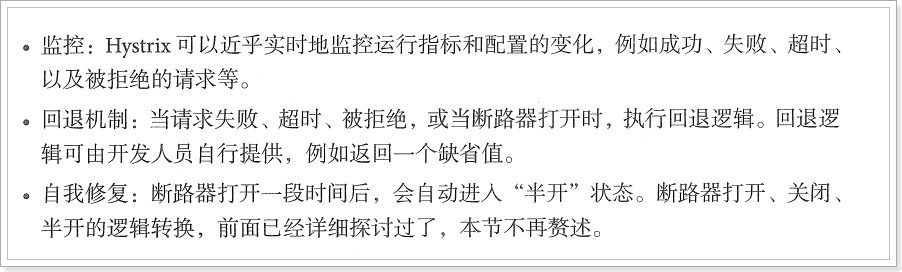


### Hystrix简介

主页：<https://github.com/Netflix/Hystrix/>

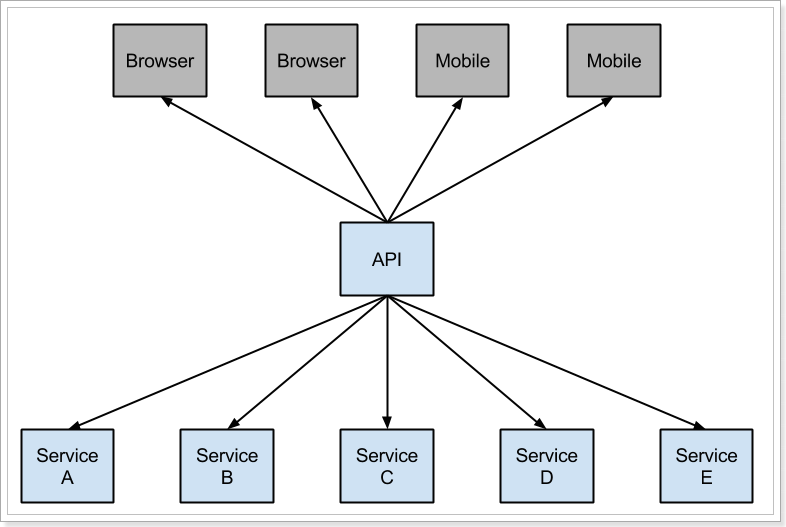




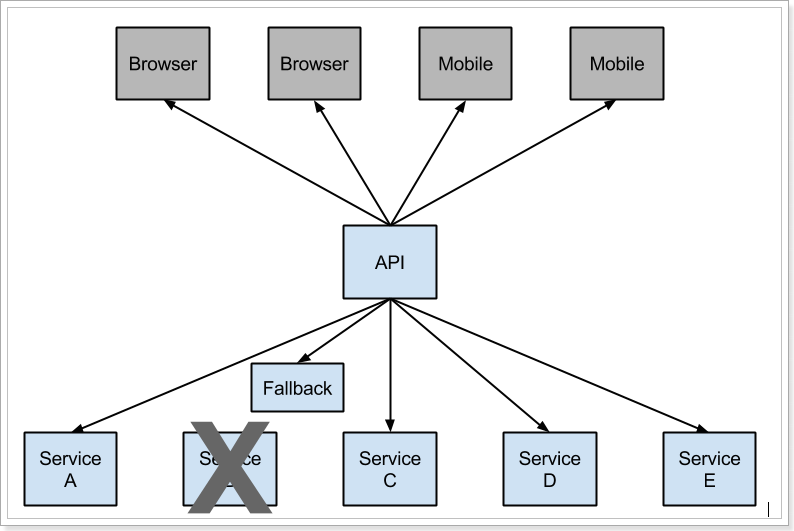


### 原理说明

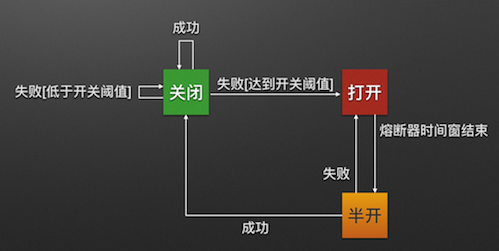
正常情况：



当对特定服务的呼叫达到一定阈值时，电路打开，不进行通讯。并且是一个隔离的线程中进行的。



熔断器开关相互转换的逻辑:

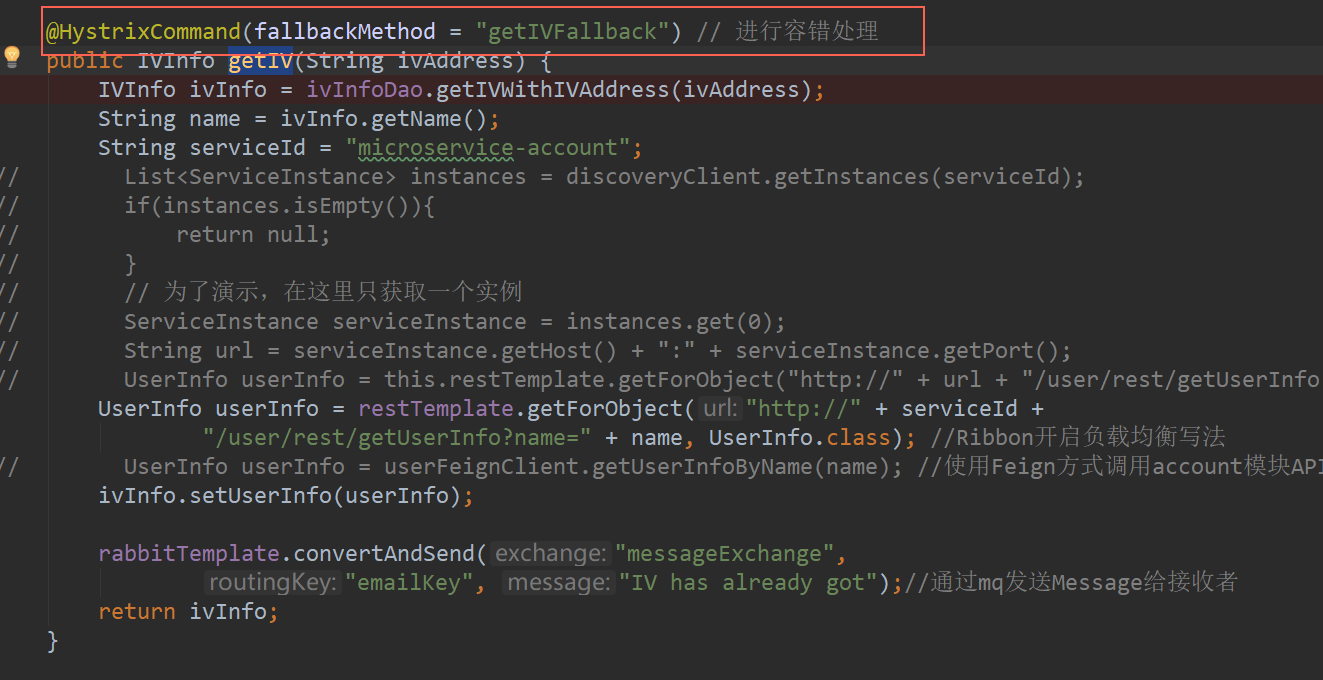


### 在microservice-iv系统中增加Hystrix实现容错。

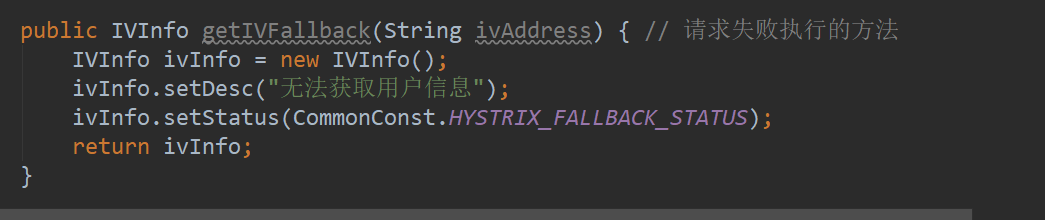
#### 导入依赖



### 修改IVService的getIV方法



#### 添加请求失败的方法



#### 开启Hystrix



启动模块，正常请求如下



停掉microservice-account，再次发起请求



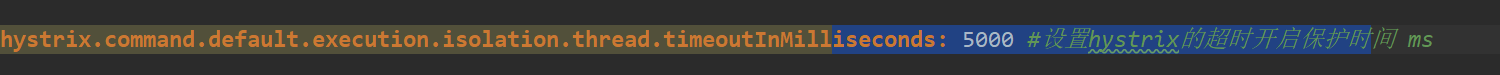
可以看到，IV服务正常，但是查询用户服务已停止服务，查询到的是错误信息。

由此可见，用户服务的宕机并没有影响IV服务的正常工作，起到的容错效果。

### 常见问题：

1.由于Spring的懒加载，通常首次调用其他服务会比较缓慢，由于hystrix的开启保护的接口调用超时时间默认为1s，所以正常调用也会出现fallback。

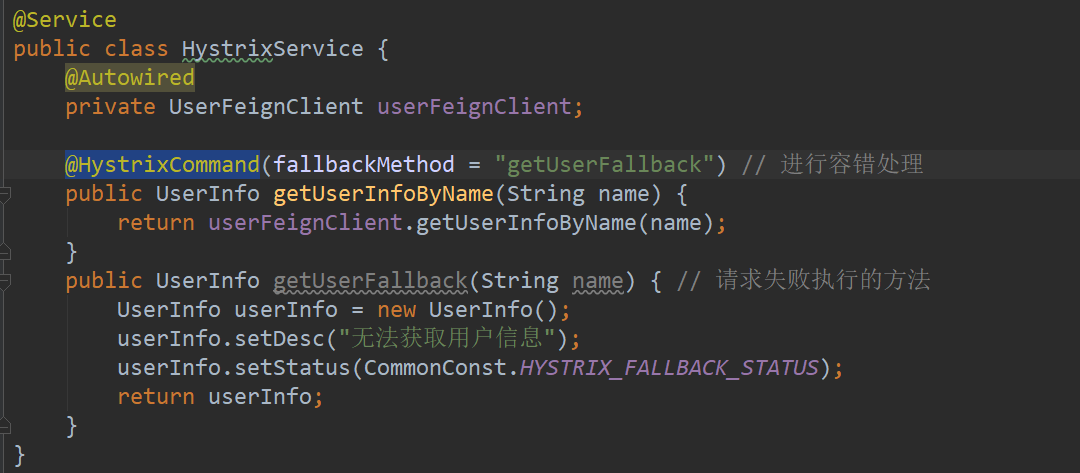
因此推荐将hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds设置为5s。

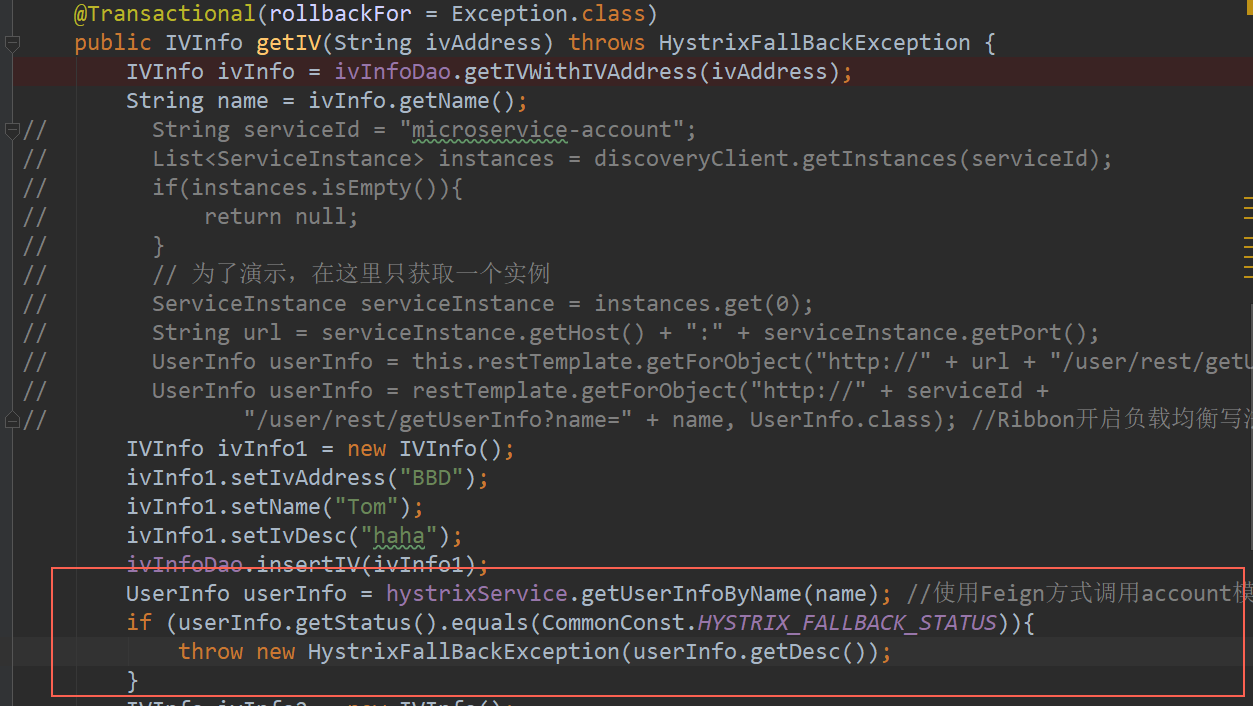


1. @Transactional和@HystrixCommand不能共用问题

如果方法上有@HystrixCommand，则事务控制会失效，原因是因为Hystrix是另起线程包裹被注解方法，这个方法不会被事务的aop增强。

解决方法，这两个注解拆开，写到2个service类里，用被事务控制的方法调用被Hystrix包裹的方法即可。



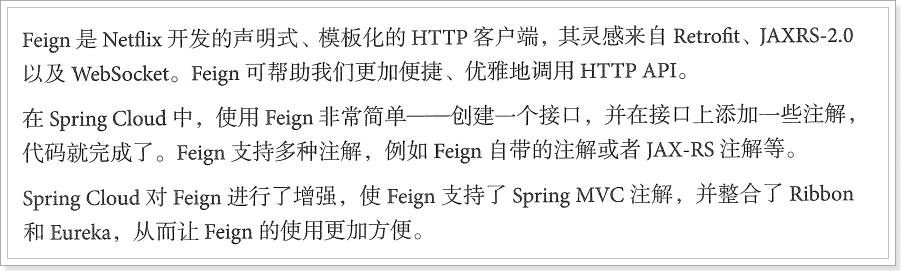


## 使用Feign实现声明式的REST调用

之前我们通过RestTemplate调用REST服务，虽然使用了Ribbon和Hystrix可以实现负载均衡和容错处理，但是这个编码在实现大量业务时会显得太过于冗余（如，多参数的URL拼接）。有没有更加优雅的实现呢？

### Feign的简介

项目主页：<https://github.com/OpenFeign/feign>

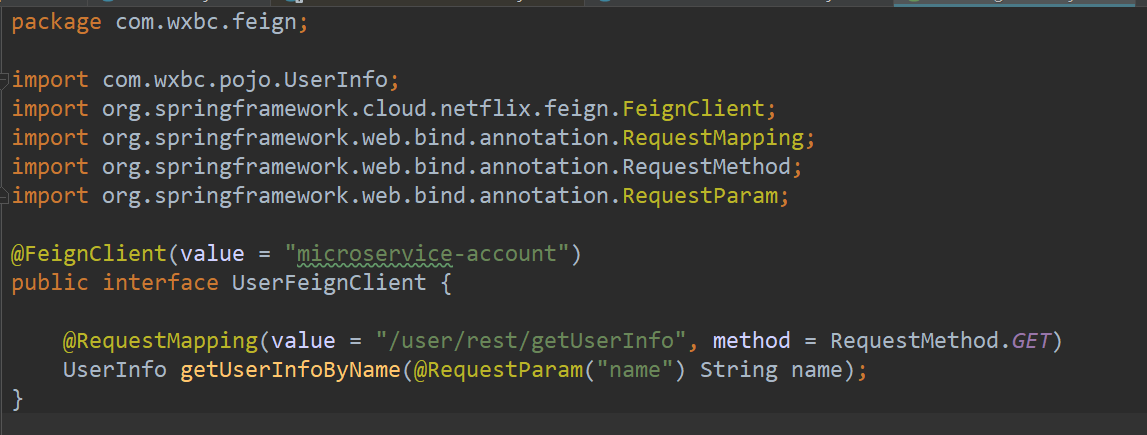


### 在IV微服务中增加Feign的支持

#### 导入依赖

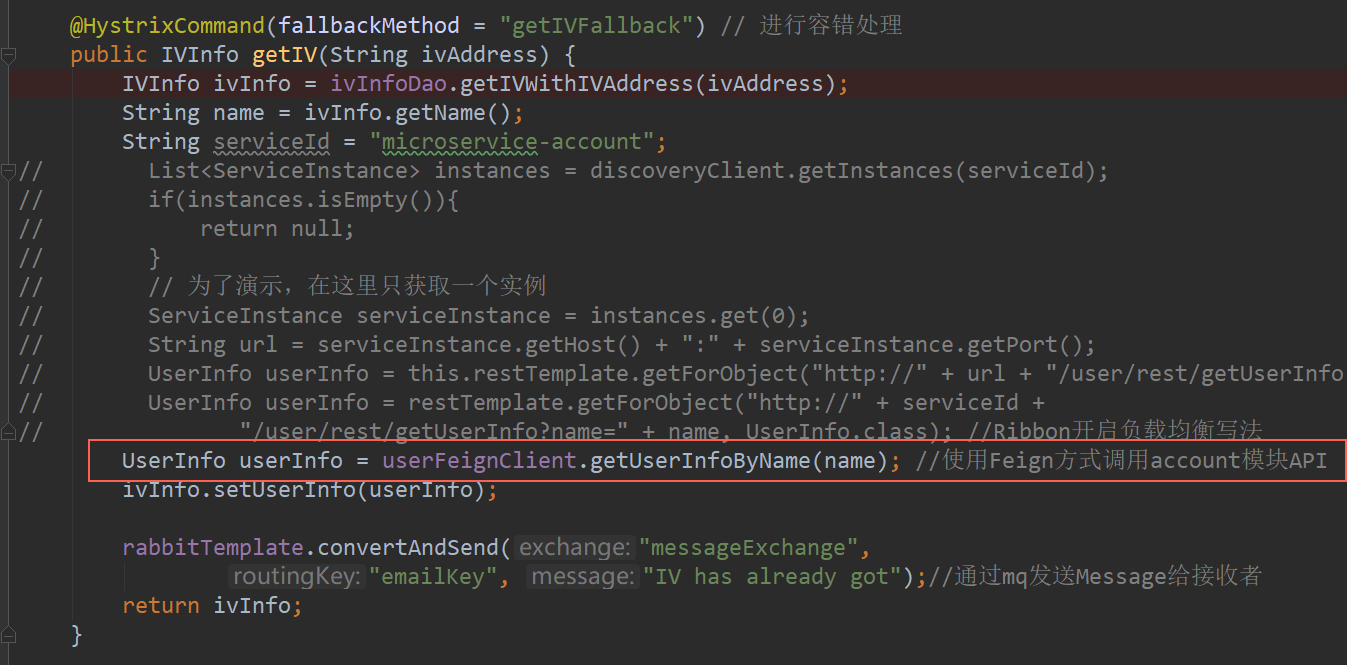


#### 创建一个UserFeignClient接口

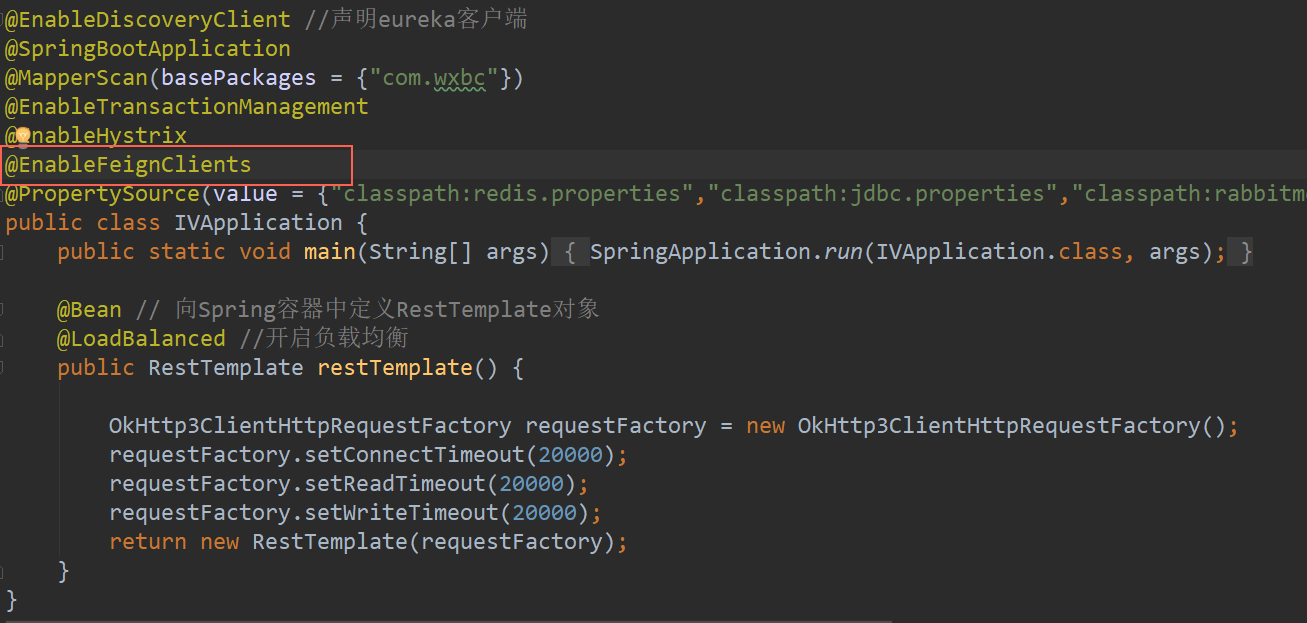


#### 改造IVservice类





#### 在启动类中添加@EnableFeignClients注解



### 原理解释：

1. 由于我们在入口@EnableFeignClients注解，Spring启动后会扫描标注了@FeignClient注解的接口，然后生成代理类
2. 我们在@FeignClient接口中指定了value，其实就是指定了在Eureka中的服务名称
3. 在FeignClient中的定义方法以及使用了SpringMVC的注解，Feign就会根据注解中的内容生成对应的URL，然后基于Ribbon的负载均衡去调用REST服务

## 服务网关 Spring Cloud Zuul

再看一下微服务架构的设计：



这种架构设计是一种封闭式的，唯一开放的出入口是Gateway，而传统的开放型服务是没有Gateway的。

传统的开放型服务每个模块都会有各自的权限校验和参数校验逻辑，这种方式存在的不足：

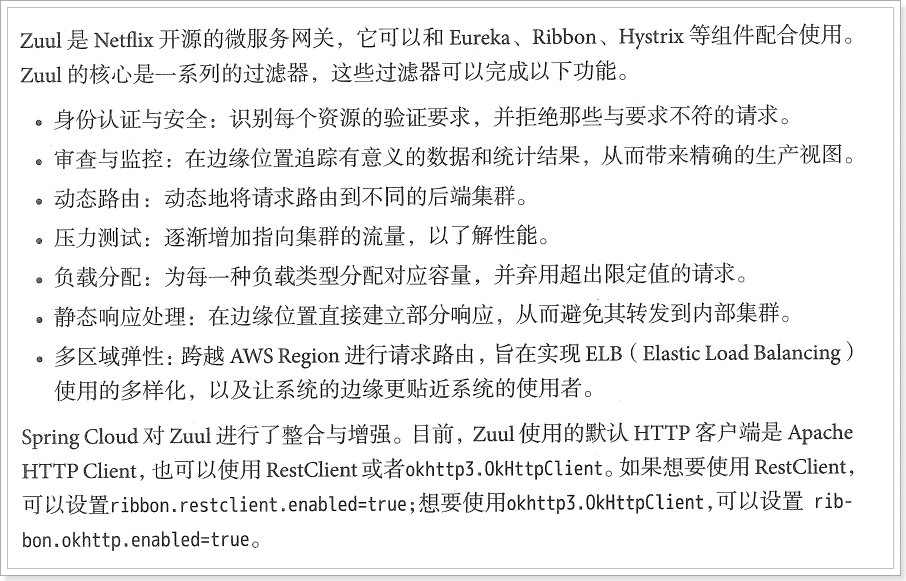
* 首先，破坏了服务无状态特点。
  + 为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。
  + 从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。
* 其次，无法直接复用既有接口。
  + 当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。

为了解决上面这些问题，我们需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器 🡺 服务网关。

服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix中的Zuul就担任了这样的一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。

### Zuul的简介

官网：<https://github.com/Netflix/zuul>



### 使用Zuul组件搭建Gateway

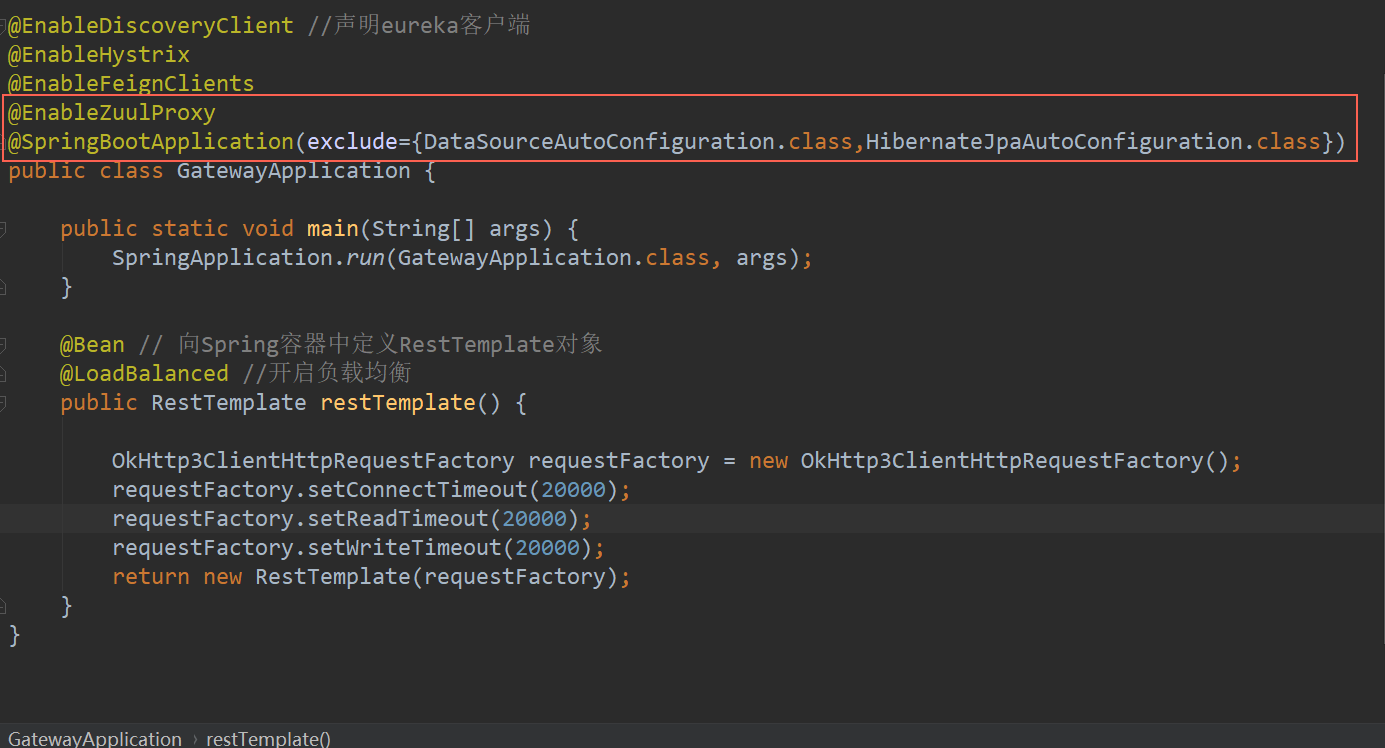
#### 创建工程microsrvice-gateway

引入zuul依赖



#### 编写启动类GatewayApplication

添加@EnableZuulProxy，并且在@SpringBootApplication注解中排除DataSourceAutoConfiguration.class,HibernateJpaAutoConfiguration.class，因为Zuul不使用数据库，不排除的话会报错。

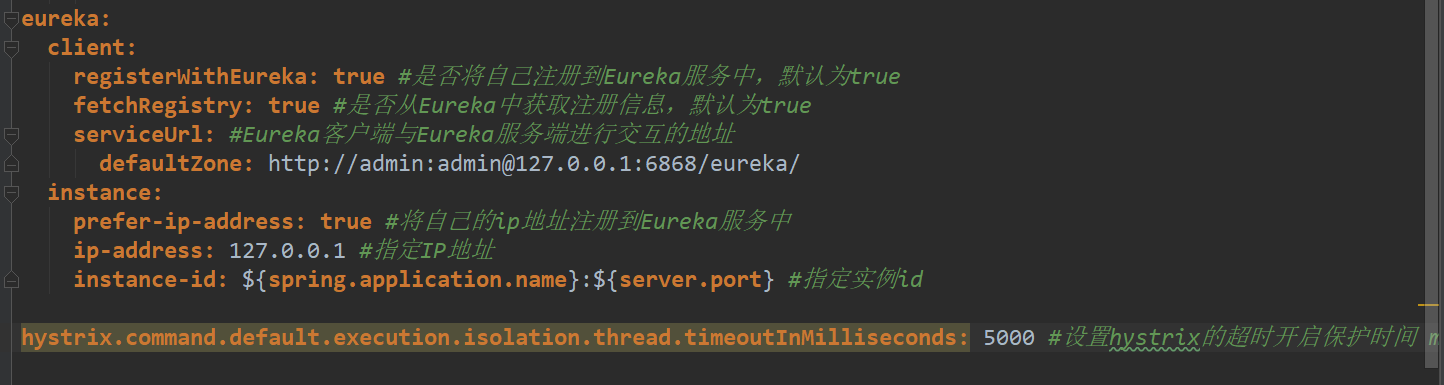


#### 编写application.yml文件

Zuul.routes.\*为制定路由分配地址，如果path配置为 /microservice-account/\*\*，serviceId为microservice-account ，则会将/microservice-account/映射成serviceId对应的url，如http://localhost:8080

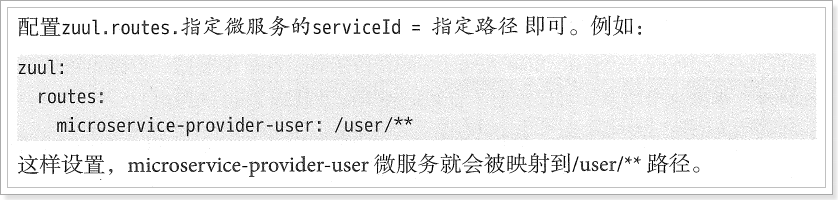


获取路由的url走的是通过Eureka注册中心获取地址，需要把自己注册到Eureka上。

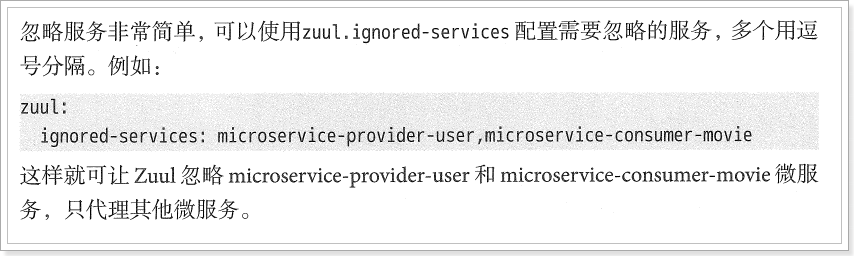


### zuul配置详解

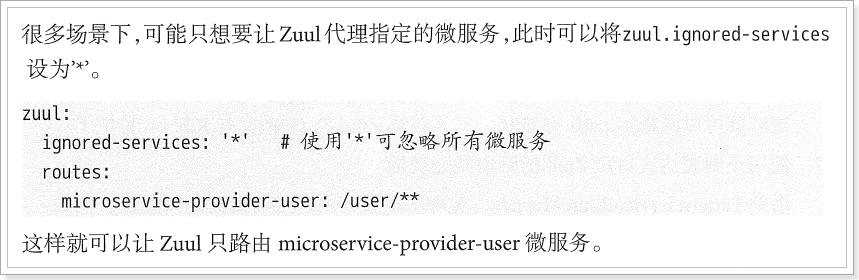
#### 指定服务id



#### 忽略指定服务



#### 忽略所有服务，只是有路由指定



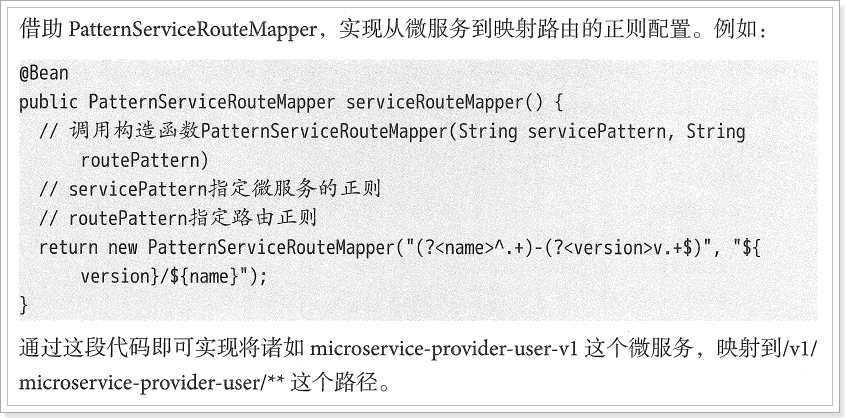
#### 同时配置path和url



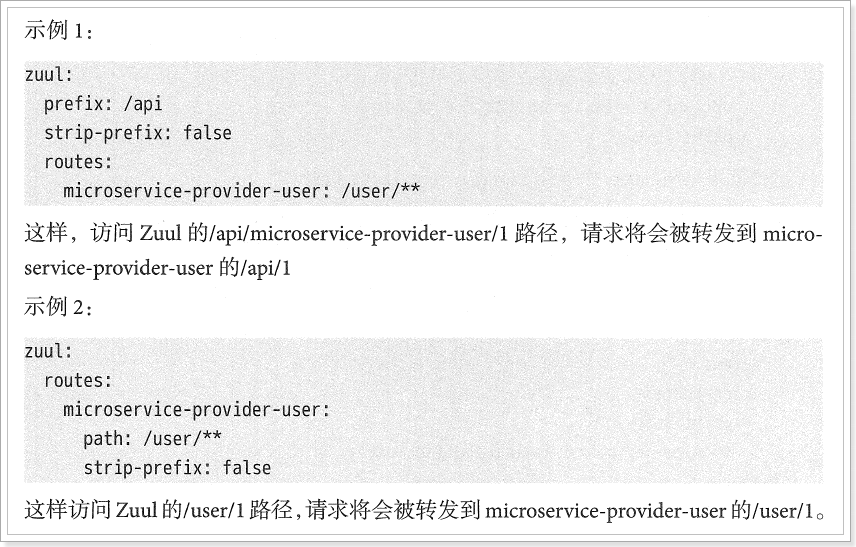
#### 面向服务配置，不破坏Hystrix、Ribbon特性



### 使用正则表达式指定路由规则



### 路由前缀



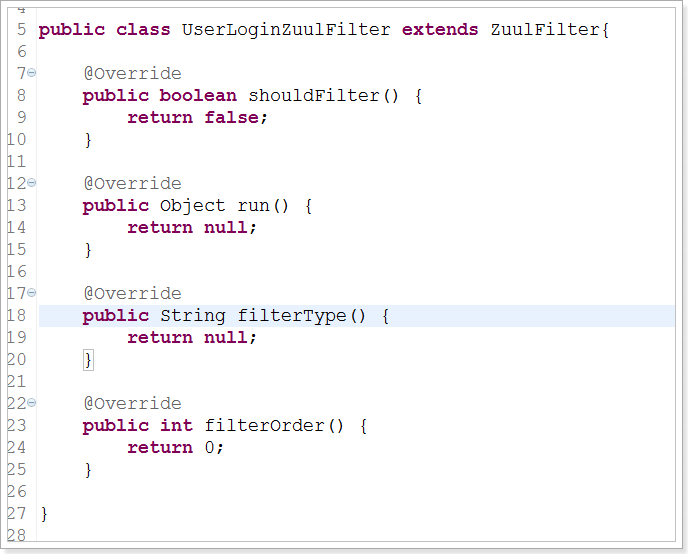
### 忽略某些路径



### 过滤器

过滤器是Zuul的重要组件。

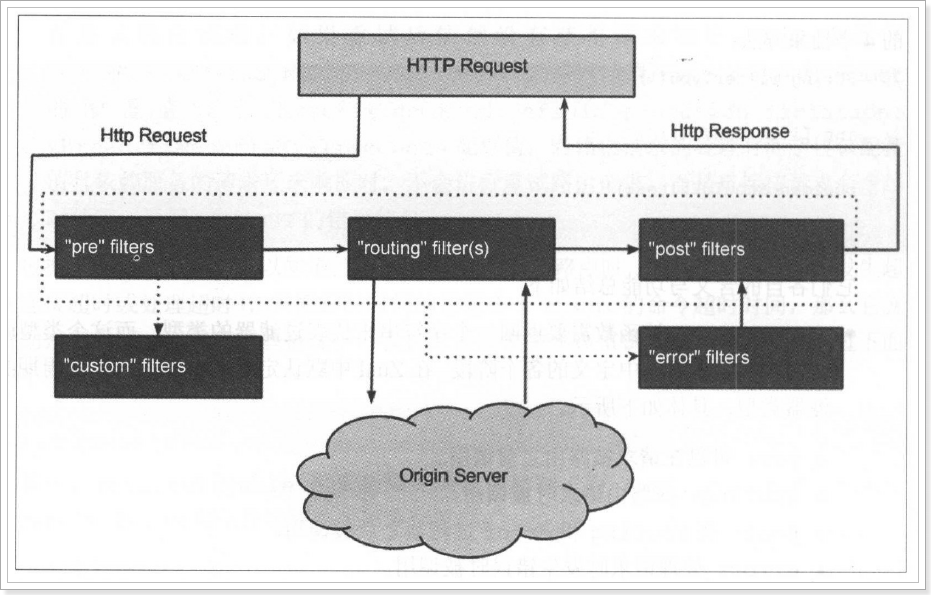
#### 过滤器ZuulFilter

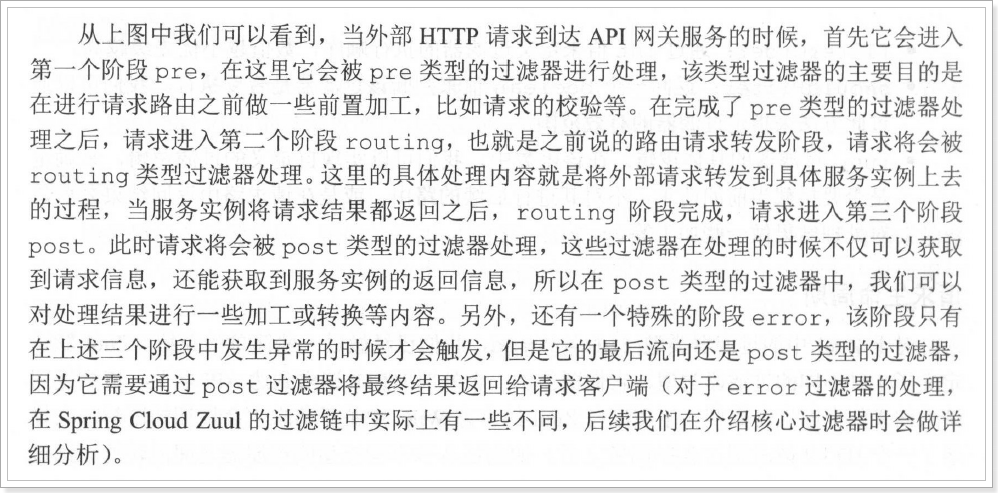


ZuulFilter是一个抽象类，其实现类需要实现4个方法：

1. shouldFilter：返回一个Boolean值，判断该过滤器是否需要执行。返回true执行，返回false不执行。
2. run：过滤器的具体业务逻辑。
3. filterType：返回字符串代表过滤器的类型
   1. pre：请求在被路由之前执行
   2. routing：在路由请求时调用
   3. post：在routing和errror过滤器之后调用
   4. error：处理请求时发生错误调用
4. filterOrder：通过返回的int值来定义过滤器的执行顺序，数字越小优先级越高。

#### 执行流程

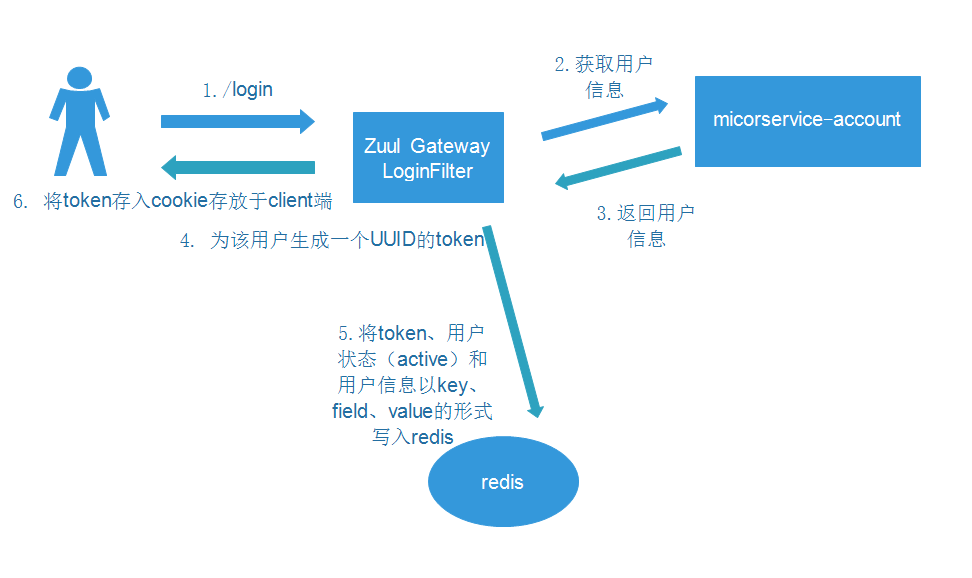




#### 过滤器实战

需求：通过编写过滤器实现两个功能：1用户登陆功能；2是否登录的检查。

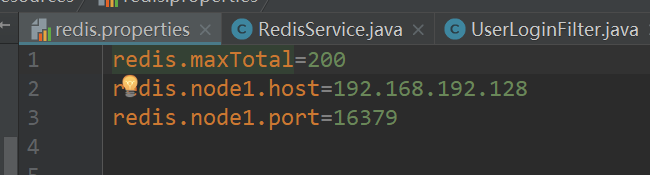
#### Gateway处理用户login的流程



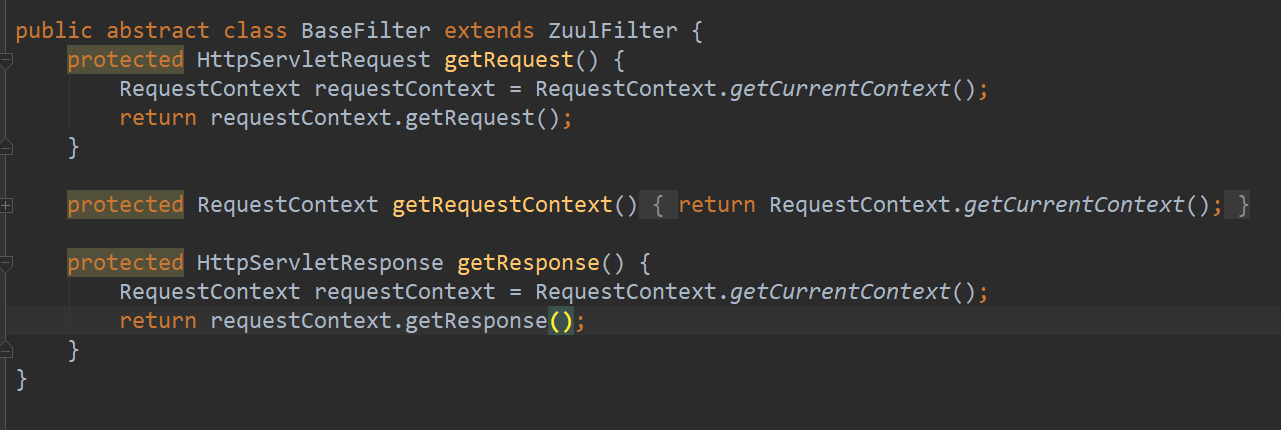
导入Redis依赖



Redis.properties配置

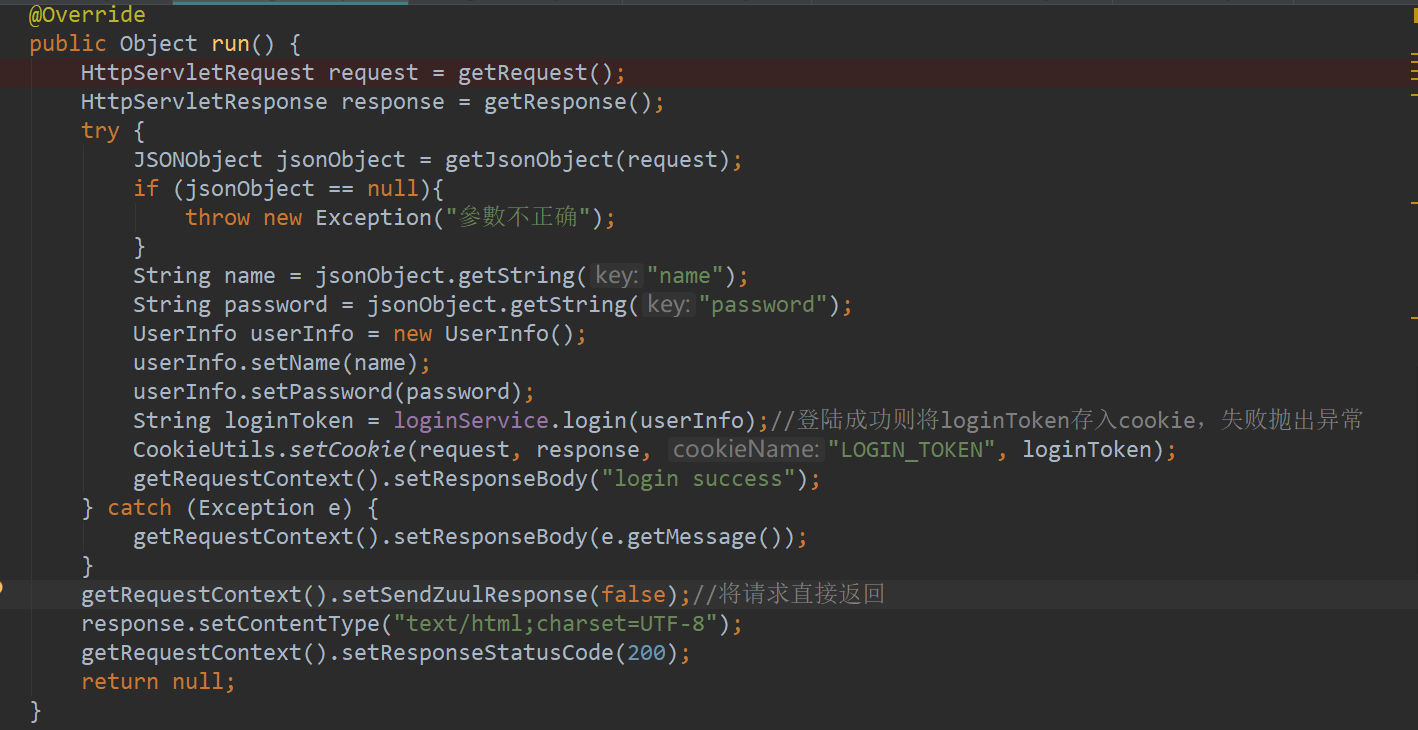


编写BaseFilter过滤器



编写UserLoginFilter过滤器

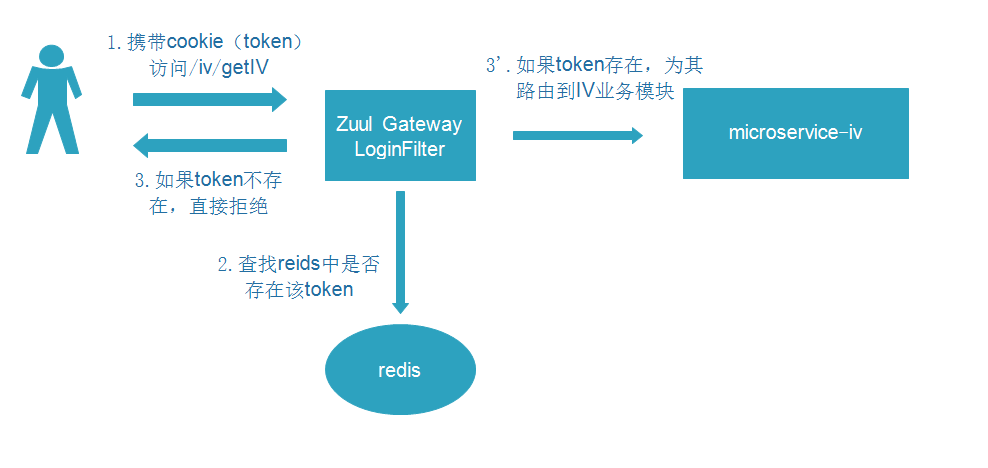


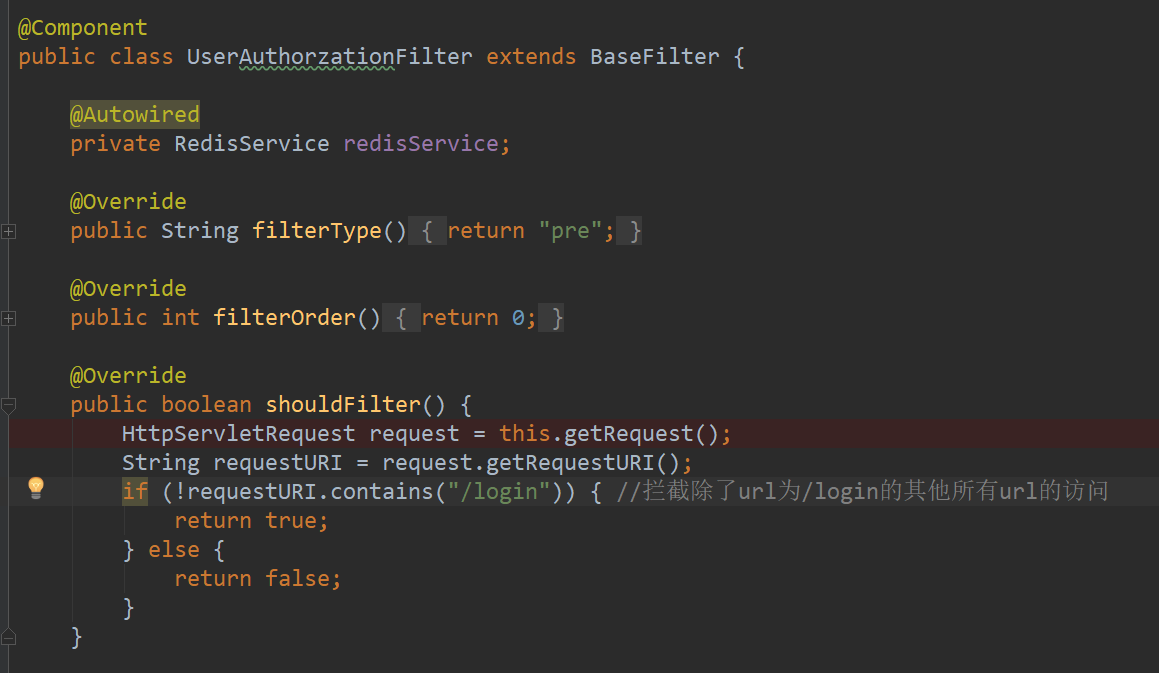


LoginService的编写



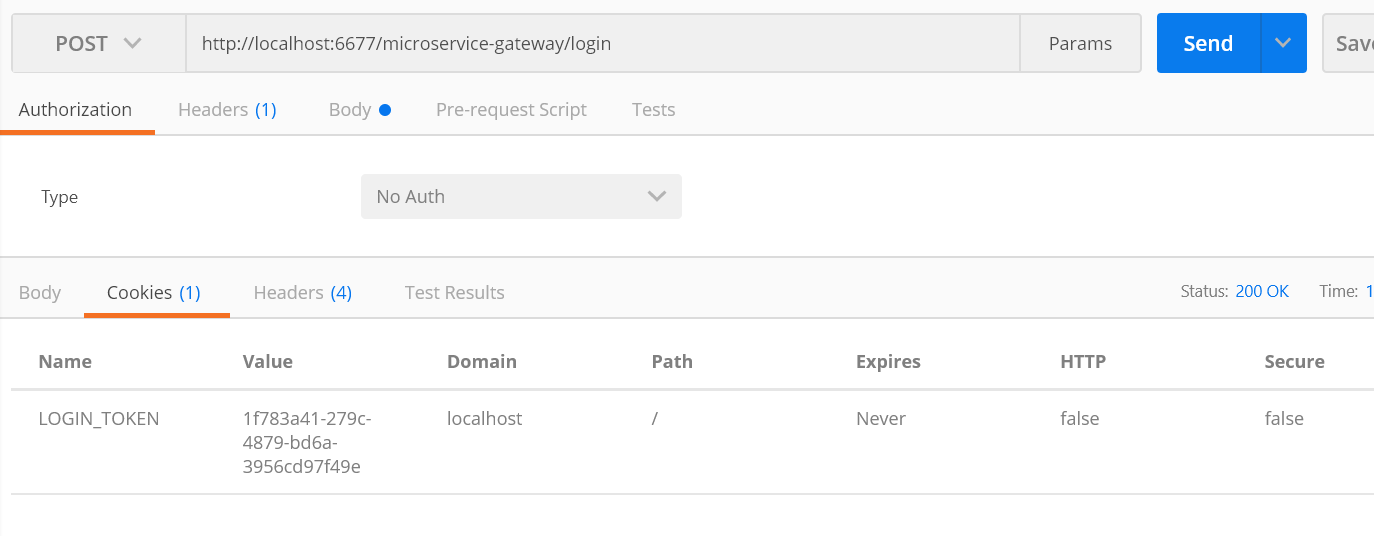
#### Gateway处理用户是否登陆检查的流程

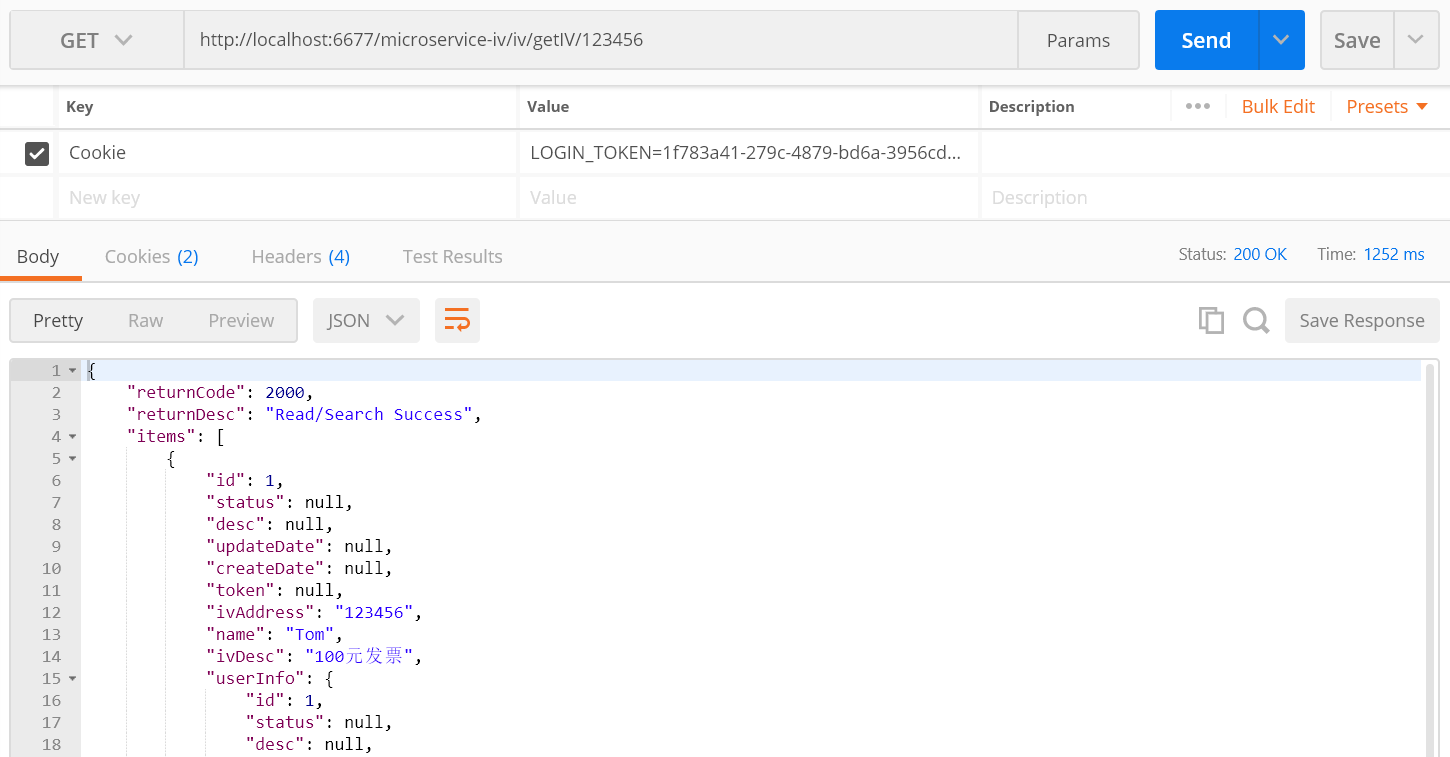






#### 启动测试





### 转发超时问题解决

采用Zuul的默认设置容易产生转发时的socket timeout，解决方案：

**zuul:  
 host:** *#防止转发超时* **connect-timeout-millis:** 10000  
 **socket-timeout-millis:** 50000

**hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:** 5000 *#设置hystrix的超时开启保护时间 ms*

## 使用Spring Cloud Config统一管理微服务的配置

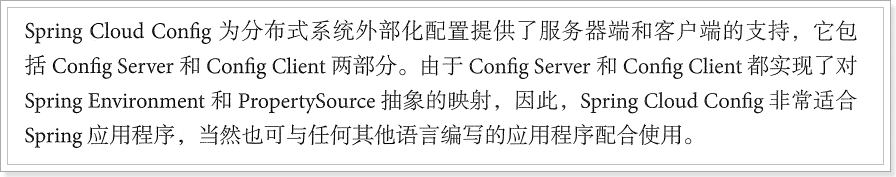
在我们开发项目时，需要有很多的配置项需要写在配置文件中，如：数据库的连接信息等。

这样看似没有问题，但是我们想一想，如果我们的项目已经启动运行，那么数据库服务器的ip地址发生了改变，我们该怎么办？

如果真是这样，我们的应用需要重新修改配置文件，然后重新启动，如果应用数量庞大，那么这个维护成本就太大了！

有没有好的办法解决呢？当然是有的，Spring Cloud Config提供了这样的功能，可以让我们统一管理配置文件，以及实时同步更新，并不需要重新启动应用程序。

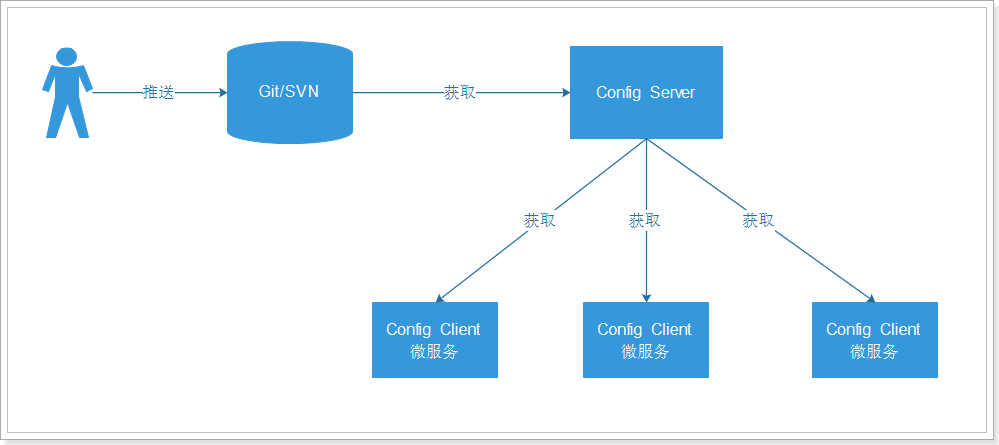
### Spring Cloud Config 简介



Config Server是一个可横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用Git存储配置文件内容，也可以使用SVN存储，或者是本地文件存储。

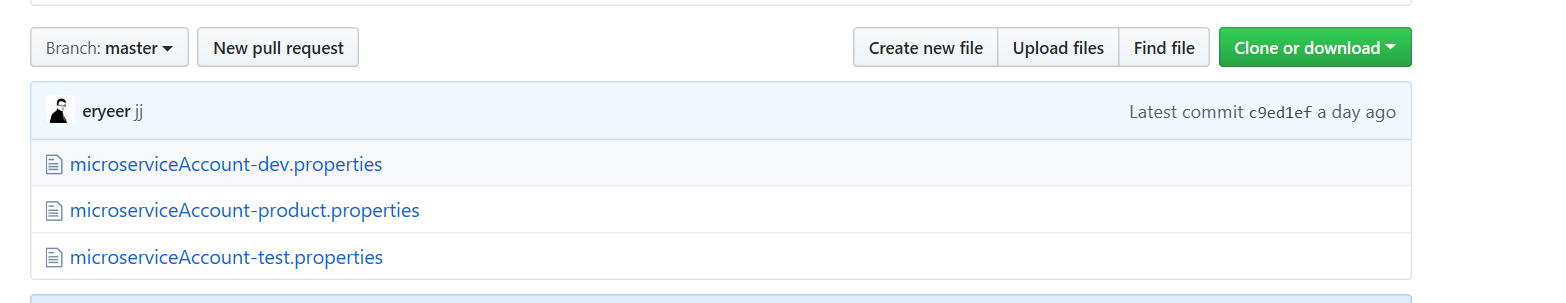
Config Client是Config Server的客户端，用于操作存储在Config Server中的配置内容。微服务在启动时会请求Config Server获取配置文件的内容，请求到后再启动容器。

使用Spring Cloud Config的架构：



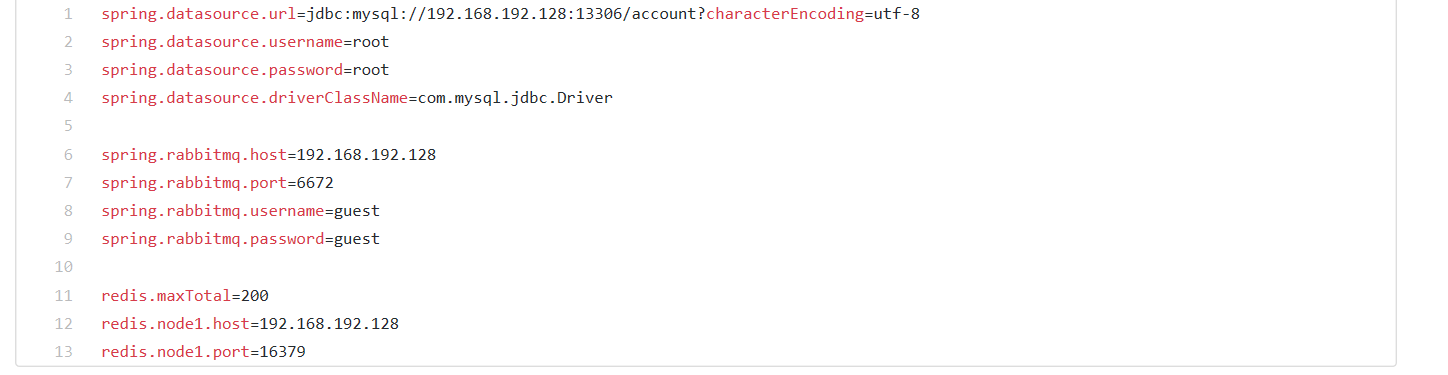
### 快速搭建Config Server

### 准备3个配置文件，推送到Git服务器



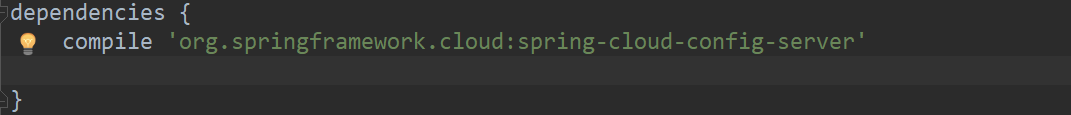
该文件的命名规则是：{application}-{profile}.properties

其内容是（另外2个文件内容稍有不同即可）：

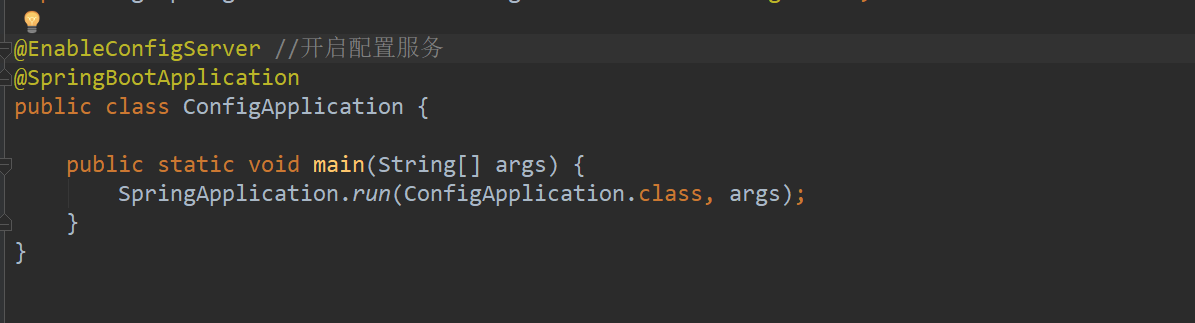


### 创建工程microservice-config

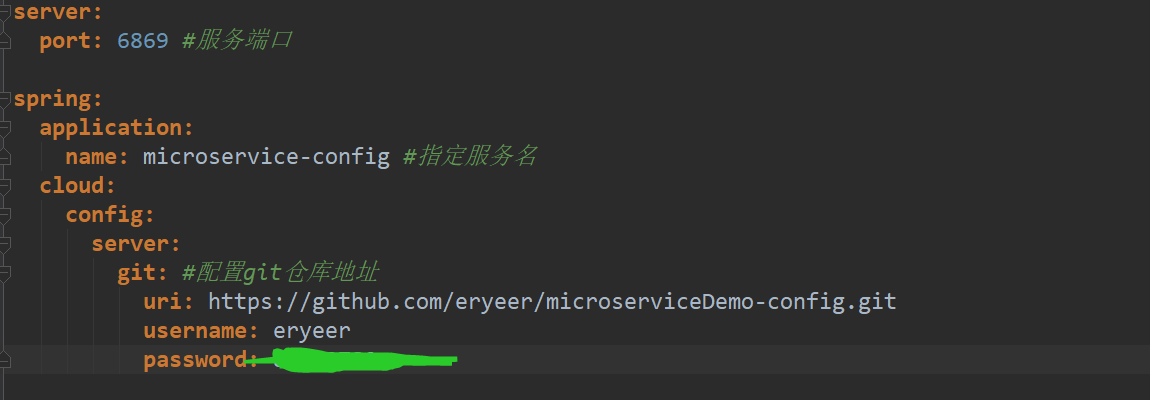
#### 导入依赖



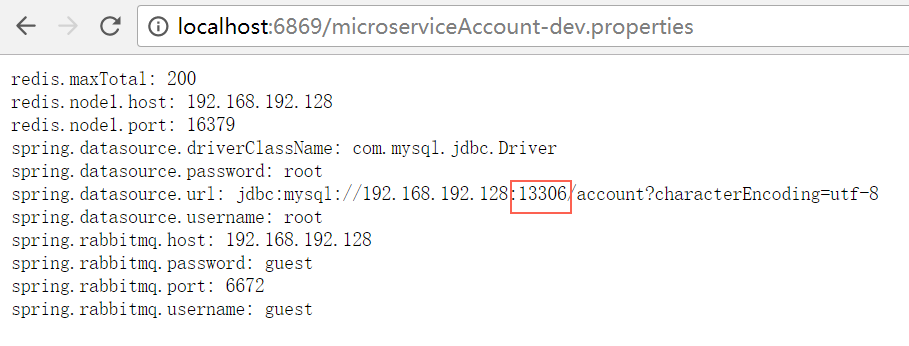
#### 编写入口ConfigApplication

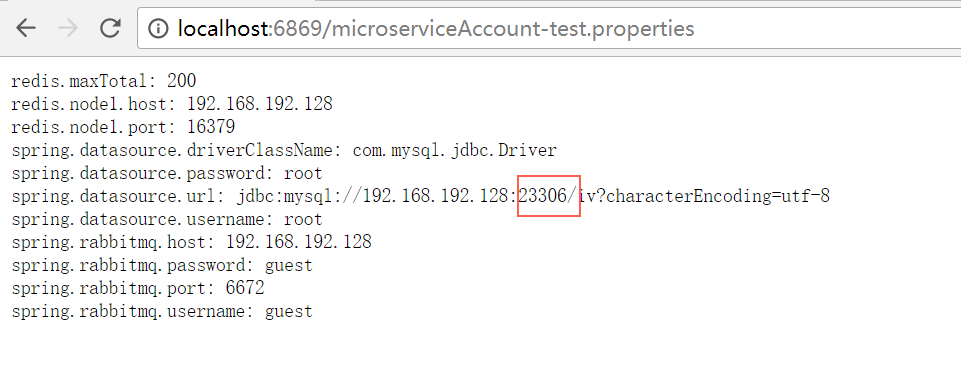


#### 编写配置文件application.yml



#### 启动测试





请求配置文件的规则如下：

/{application}/{profile}/[label]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

其中{label}是指分支，默认是master。

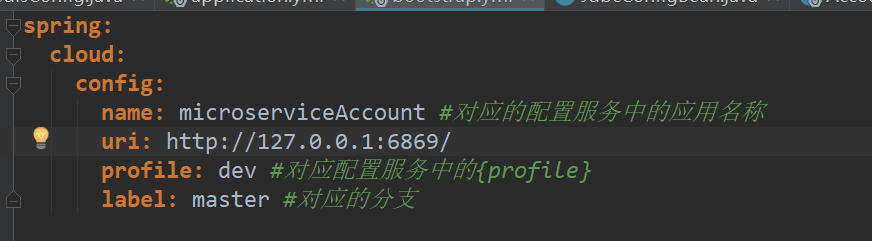
### 搭建Config Client

我们在microservice-account项目中添加Config Client的支持。来读取JDBC的配置文件的内容。

#### 导入依赖



#### 创建配置文件bootstrap.yml



看到日志打印，说明加载config server的配置成功

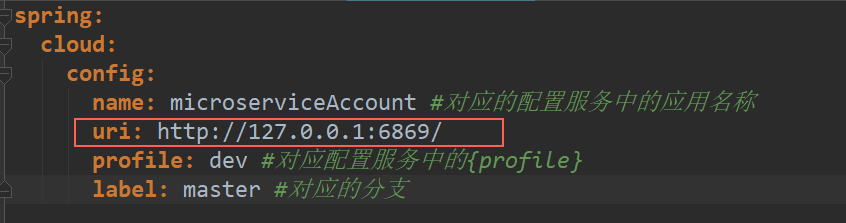
2018-03-06 22:47:22.955 INFO 23128 --- [ main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://127.0.0.1:6869/

2018-03-06 22:47:29.281 INFO 23128 --- [ main] c.c.c.ConfigServicePropertySourceLocator : Located environment: name=microserviceAccount, profiles=[dev], label=master, version=c9ed1ef45b09d7f51f22993da139a5d5c2f71a96, state=null

2018-03-06 22:47:29.281 INFO 23128 --- [ main] b.c.PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource [name='configService', propertySources=[MapPropertySource {name='configClient'}, MapPropertySource {name='https://github.com/eryeer/microserviceDemo-config.git/microserviceAccount-dev.properties'}]]

### Config Client配置的优化

在microservice-account中作为Config Client，在配置文件中了配置了Config Server的地址：



这样的硬编码是不好的，如果配置中心的ip地址发生了改变，那么就需要重新修改并且重启应用了。

如果将Config Server作为一个微服务，并且将其注册的Eureka中，就可以不用硬编码了。

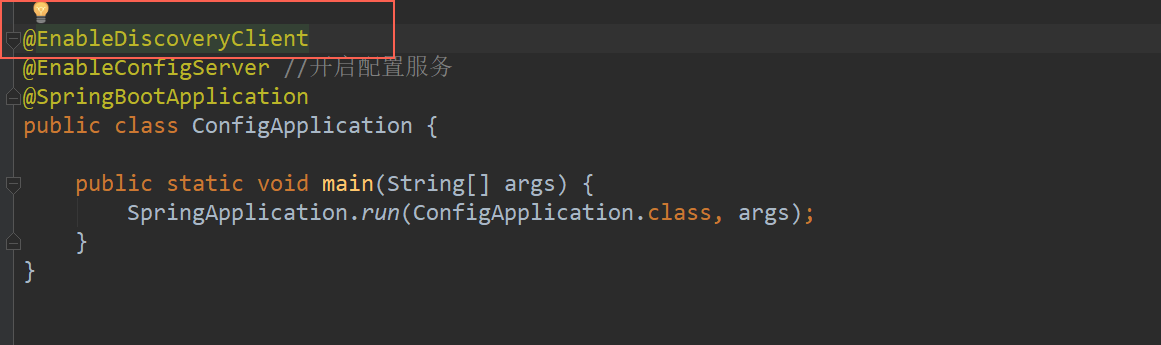
#### 在microservice-config中添加Eureka的依赖



#### 修改application.yml文件

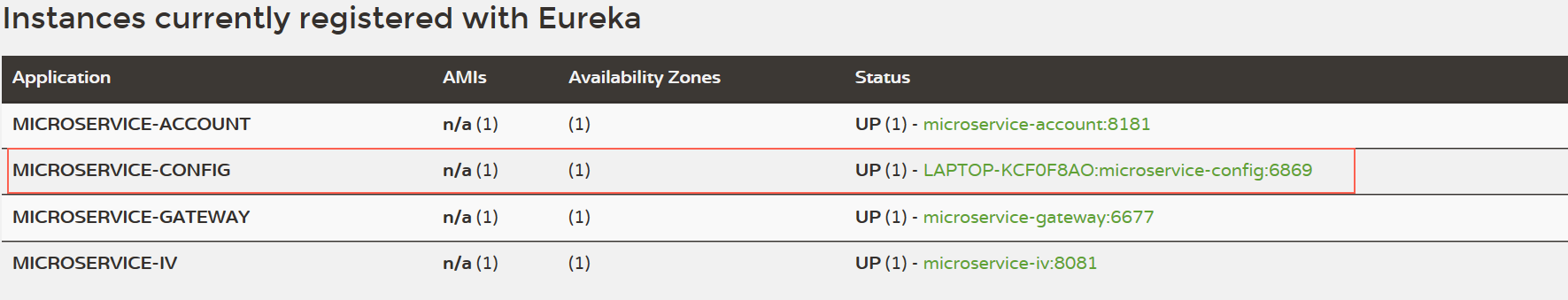


#### ConfigApplication添加@EnableDiscoveryClient注解



#### 重新启动

发现microservice-config已经注册到eureka



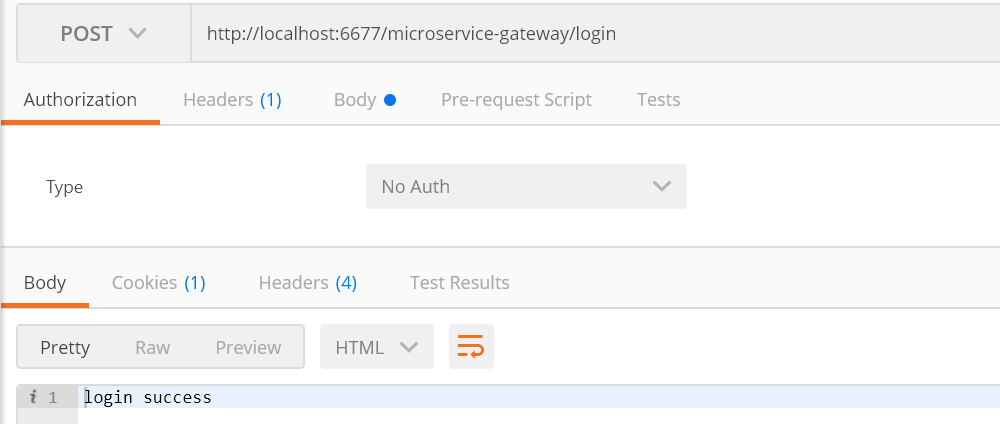
#### 在microservice-account中修改bootstrap.yml配置



疑问：在application.yml中以及配置Eureka的信息，为什么在bootstrap.yml还需要配置？

因为在Spring Boot中bootstrap.yml在application.yml之前加载，所以即使在application.yml中以及配置Eureka的信息，是使用不了的，所以需要在bootstrap.yml中配置Eureka的信息。

#### 测试



一切正常。

2018/3/6

编辑by赵辰