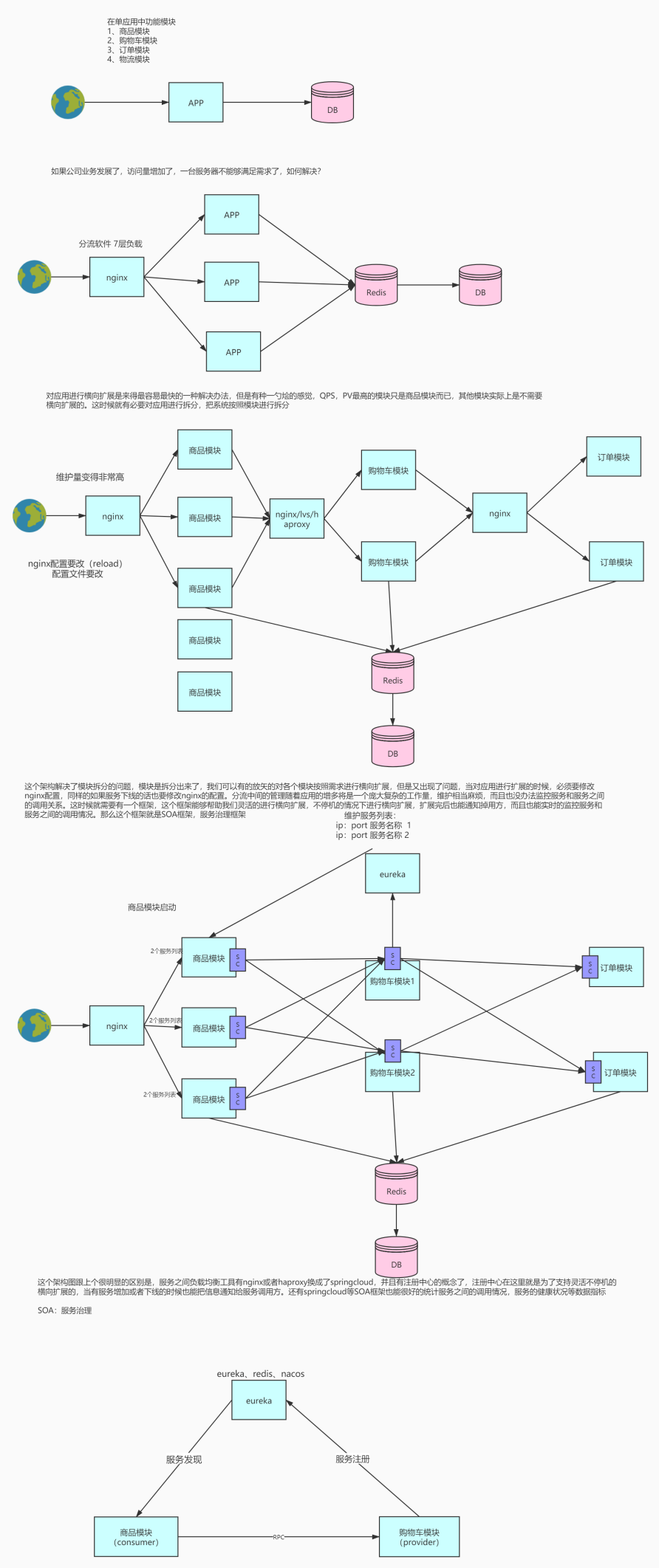
**Springcloud**

# 博客网址

<https://juejin.im/post/5e3d1594518825493c7b4840>

# 架构演变



# 微服务架构的优缺点

缺点:

1. 部署困难
2. 分布式事务问题,接口设计变得讲究，特殊场景，幂等，支持分布式事务
3. 网络抖动带来的超时，

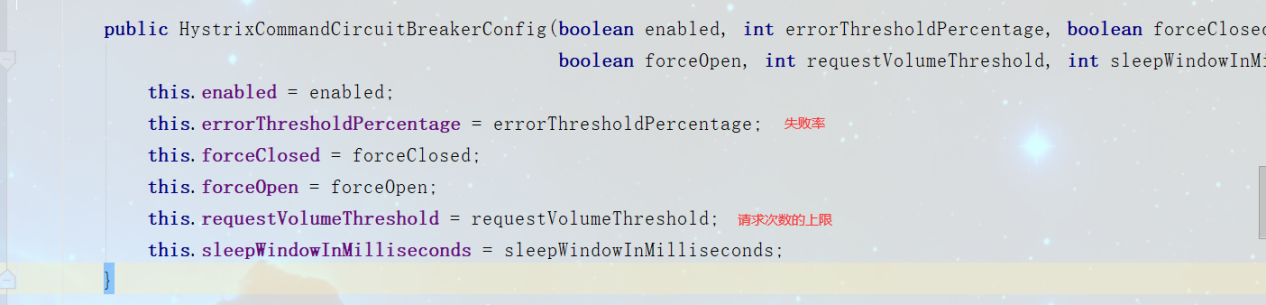
优点：

1. 实时监控
2. 扩容简单
3. 运行时干预：可以手动下线服务

# hystrix

## 配置

是在HystrixCommandProperties中。

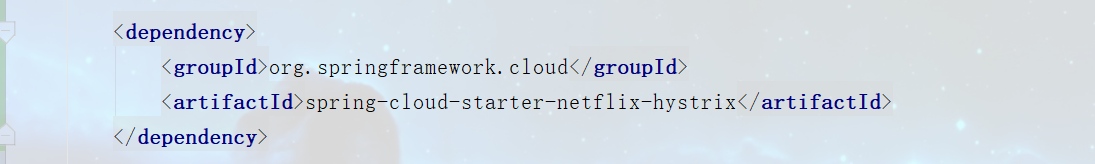


## 场景

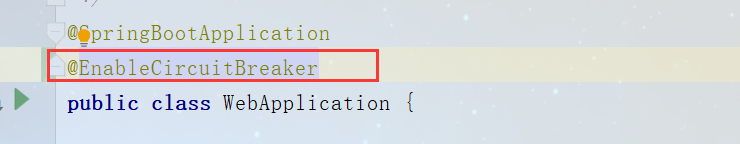
大量的请求请求某台服务器的某个接口，但是这个接口由于某些原因不能正常的相应，由于请求太多，已经耗尽了线程池的中线程资源，这时有其他的请求去访问B接口，B接口是正常的，但是由于线程池资源耗尽，导致请求在等待，这时很有可能发生服务雪崩，一个接口的不能正常访问导致了整台机器的所有接口都不不能正常访问。

## 使用

1、加入jar包



1. 开启熔断器功能



3、

## 隔离策略

### 线程池

Hystrix会分配新的线程，请求速度快，占用内存。

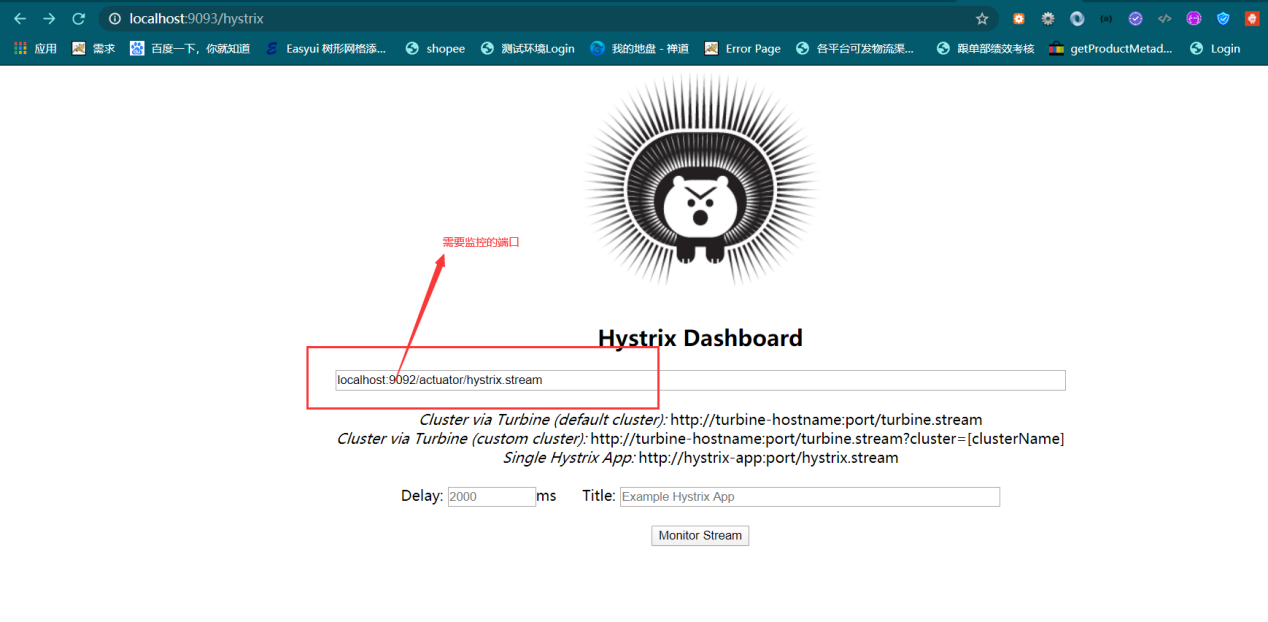
### 信号量

Hystrix不会分配新的线程，请求速度相比上面的策略会相对慢一些。

## 服务降级

如果服务失败了，请求如何处理。

## 服务监控



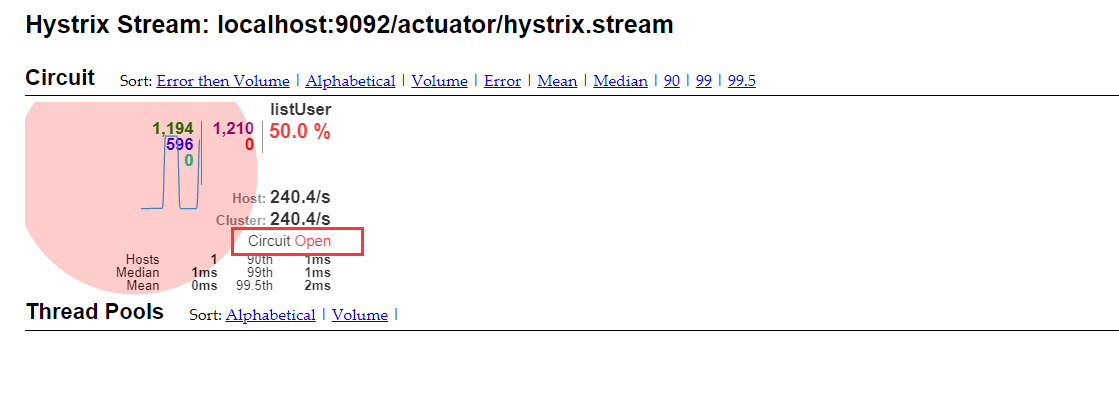
暴露/actuator/hystrix.stream

|  |
| --- |
| @Bean **public** ServletRegistrationBean hystrixMetricsStreamServlet() {  ServletRegistrationBean registration = **new** ServletRegistrationBean(**new** HystrixMetricsStreamServlet());  registration.addUrlMappings(**"/actuator/hystrix.stream"**);  **return** registration; } |

## 服务熔断

熔断开启条件：

1. 在一段时间内
2. 请求次数达到上限
3. 失败率达到一定的比率：默认为50%

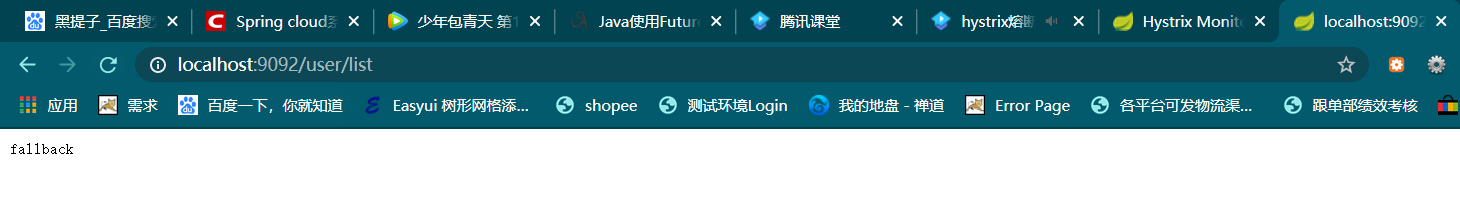


隔一段时间后，熔断器从打开状态------>>>>变为半开状态

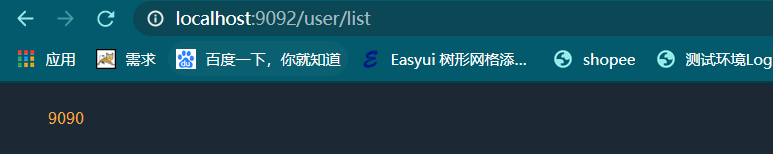
半开状态下，接收用户请求，如果处理成功，由半开状态变为关闭状态；

如果失败，由半开状态变为打开状态。

打开之后马上去请求，直接走的是降级方法：



过一段时间之后（默认5秒），再去请求



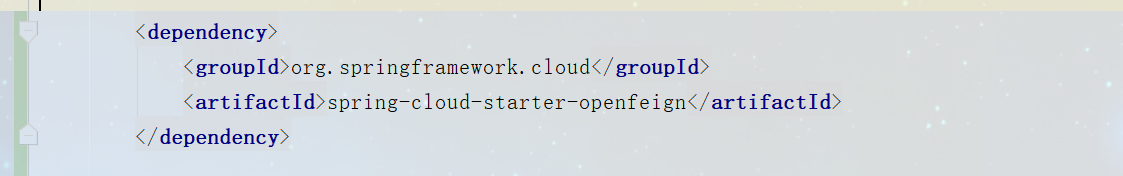
## Hystrix功能如何生效的

HystrixCommandAspect 是一个切面，切面表达式是一个HystrixCommand注解，那么只要有方法加了这个注解，就会为该类生成代理，将通知切入到加注解的方法中去。

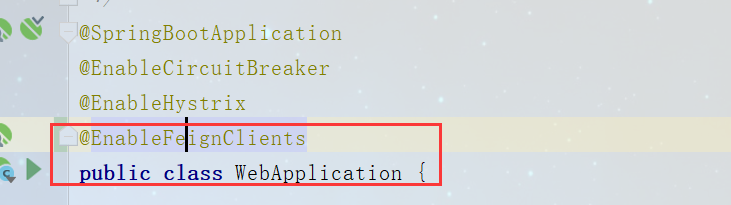
# Feign

## 使用

1、加入jar包



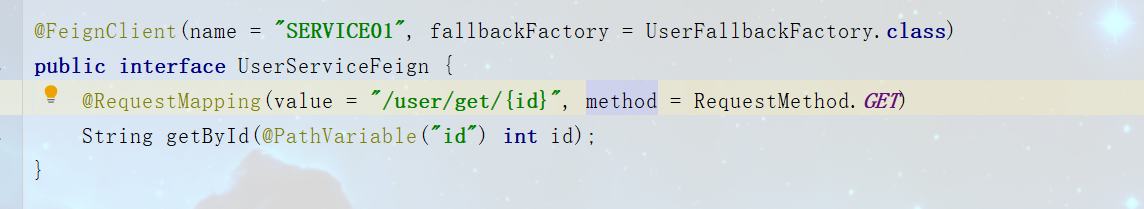
2、开启feign



3、开启hystrix

|  |
| --- |
| **feign**:  **hystrix**:  **enabled**: true |

4、定义接口



5、设置回调工厂

|  |
| --- |
| @Component **public class** UserFallbackFactory **implements** FallbackFactory<UserServiceFeign> {   @Override  **public** UserServiceFeign create(Throwable throwable) {  System.***out***.println(throwable.getCause());  **return new** UserServiceFeign() {  @Override  **public** String getById(**int** id) {  **return** String.*valueOf*(id) + **"===================fallback"**;  }  };  } } |

## 可以对异常信息包装成自定义的异常对象

|  |
| --- |
| @Component **public class** FeignErrorMessageFilter {   @Bean  **public** FeignErrorDecoder feignErrorDecoder() {  **return new** FeignErrorDecoder();  }   **private class** FeignErrorDecoder **implements** ErrorDecoder{   @Override  **public** Exception decode(String s, Response response) {  *// RuntimeException可以替换成自定义的异常 // RuntimeException runtimeException = new RuntimeException(response.getbo)* **try** {  String s1 = Util.*toString*(response.body().asReader());  RuntimeException runtimeException = **new** RuntimeException(s1);  **return** runtimeException;  } **catch** (IOException e) {  e.printStackTrace();  }  **return null**;  }  }  } |

# 超时

Ribbon和hystrix的配置如下：

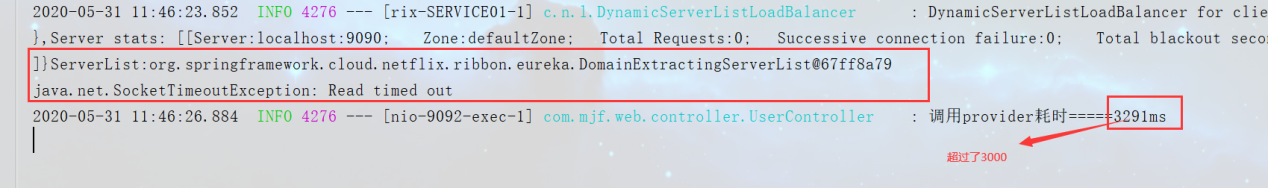
|  |
| --- |
| **ribbon**:  **ConnectTimeout**: 1000 *# 请求连接的超时时间* **ReadTimeout**: 3000 *#请求处理的超时时间* **hystrix**:  **command**:  **default**:  **execution**:  **isolation**:  **thread**:  **timeoutInMilliseconds**: 5000 *#全局超时时间* |

1、当ribbon的重试时间超过了hystrix的超时时间，超时时间就以hystrix为主，因为hystrix进行了方法降级处理。

关闭ribbon的重试策略

|  |
| --- |
| **ribbon**:**OkToRetryOnAllOperations**: false  MaxAutoRetriesNextServer: 0  **MaxAutoRetries**: 0 |

调用服务的时间超过了ribbon的设置的超时时间，就会打印出超时的异常信息。



# Config

见博客：<https://blog.csdn.net/qq_31706095/article/details/106178001>。

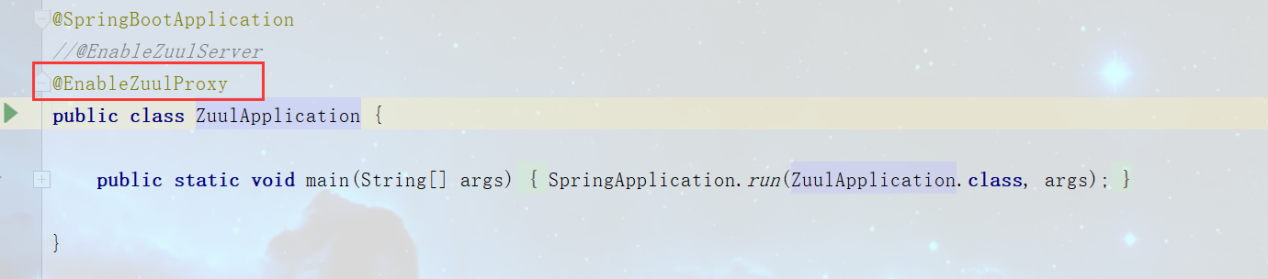
# Zuul

## 搭建

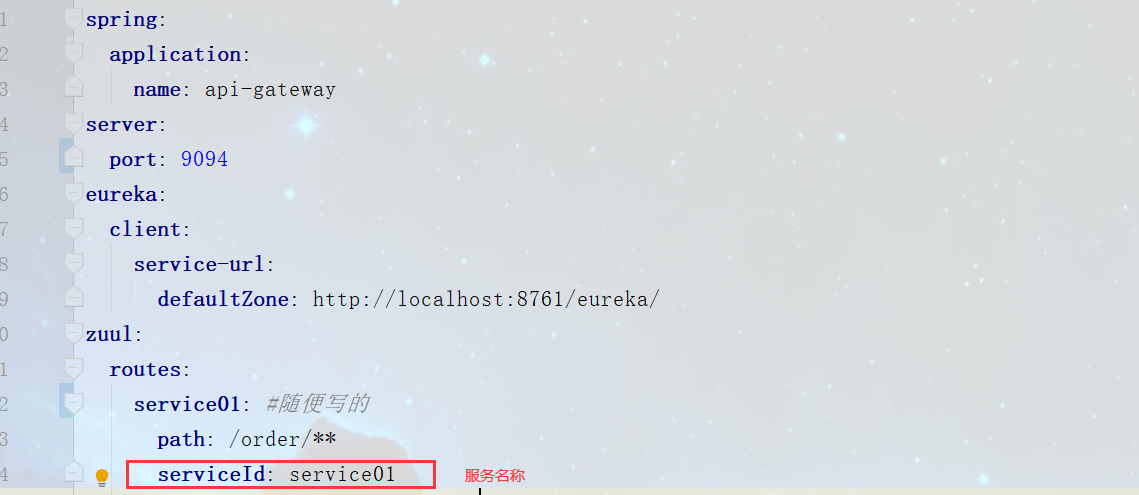
1、Jar包导入

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-zuul</**artifactId**> </**dependency**> |

2、开启zuul路由功能。



3、配置



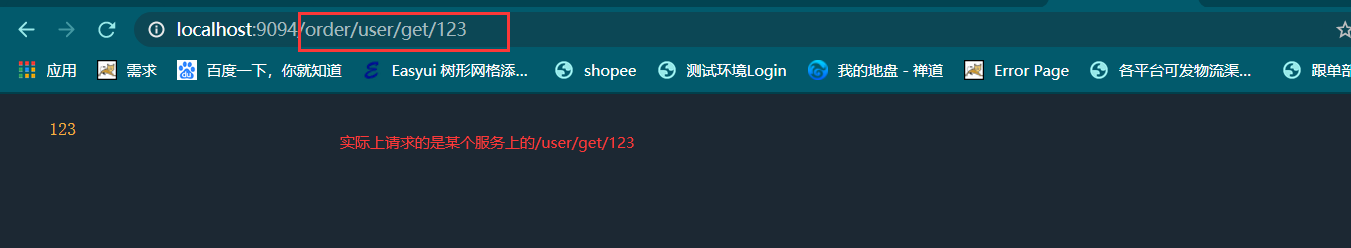
## 动态路由

需要使用配置中心，将zuul的配置放到git上面，实现自动刷新。

需要将zuulproperties上面加上@refrescope注解，这样，每次在修改git上面的配置文件的时候，就会动态的刷新zuulproperties的属性，其实就是重新创建了一个对象，然后重新读取了配置文件中的属性值。

|  |
| --- |
| @Bean @Primary @ConfigurationProperties(**"zuul"**) @RefreshScope **public** ZuulProperties zuulProperties() {  **return new** ZuulProperties(); } |

## 请求路径



## 自定义过滤器

Pre：还没有调用服务之前调用

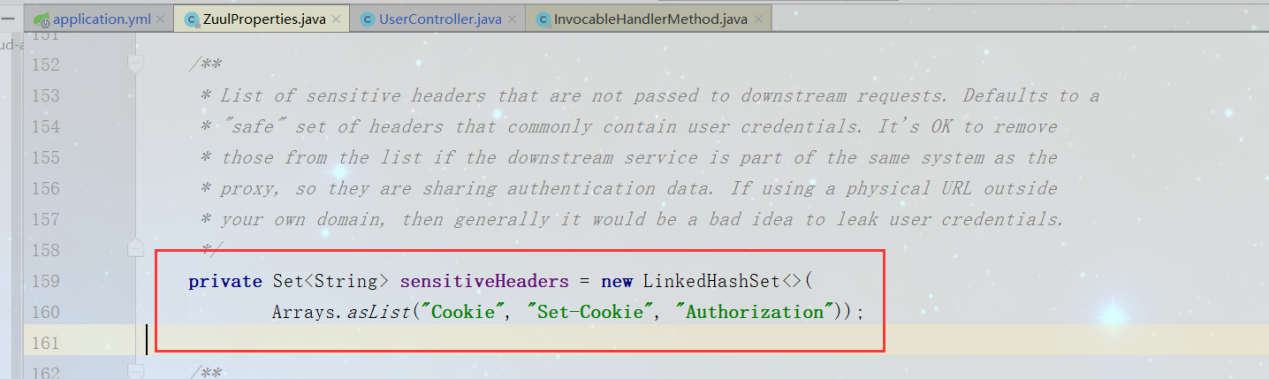
Post：调用完服务之后调用的。

route：

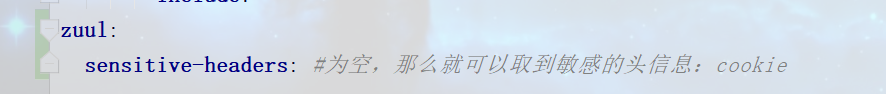
Error：调用前几个过滤器出现了异常，就进入了error过滤器。

### 敏感信息

默认是取不到Cookie，Set-Cookie，Authorization的信息的。



如果要想取到这些信息，那就就需要配置sensitiveHeaders为空。



此时就可以取到cookie的信息了。



# 授权

## 获取token的方式

### 授权码模式

需要向平台申请AppId和AppSecret，才具备平台资质。

### 客户端模式

将申请到AppId和AppSecret，再加上grant\_type=client给认证服务器，认证服务器会检测AppId和AppSecret是否存在，如果存在，就会返回一个token给客户端。

如果不同的用户，用同一个token都是可以访问的。

### 密码模式

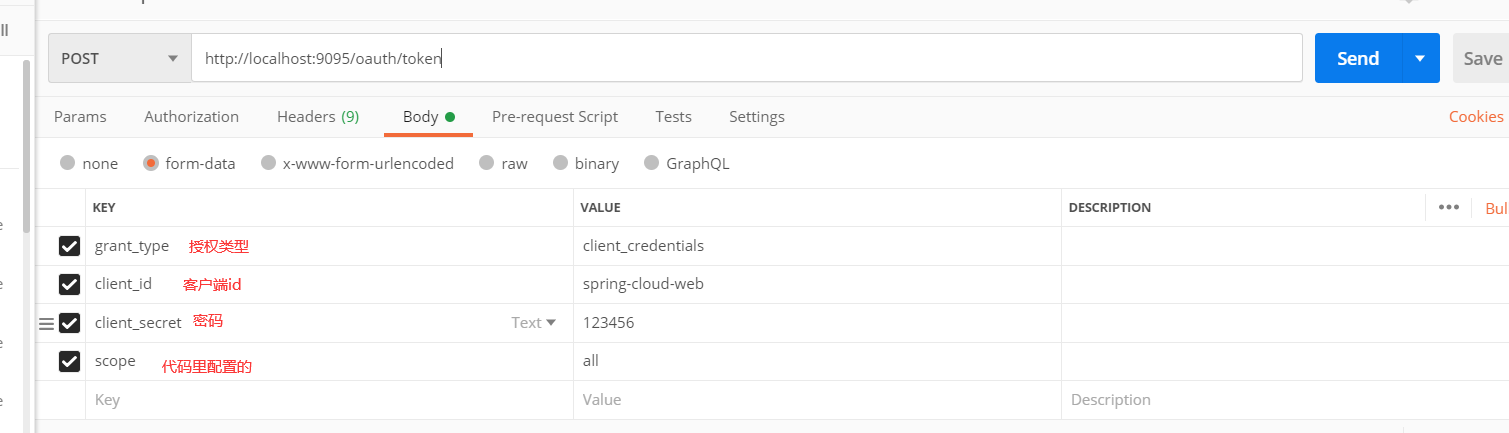
粒度更细，将token与用户绑定。不同的用户使用同一个token不能访问，每个用户都由一个token。

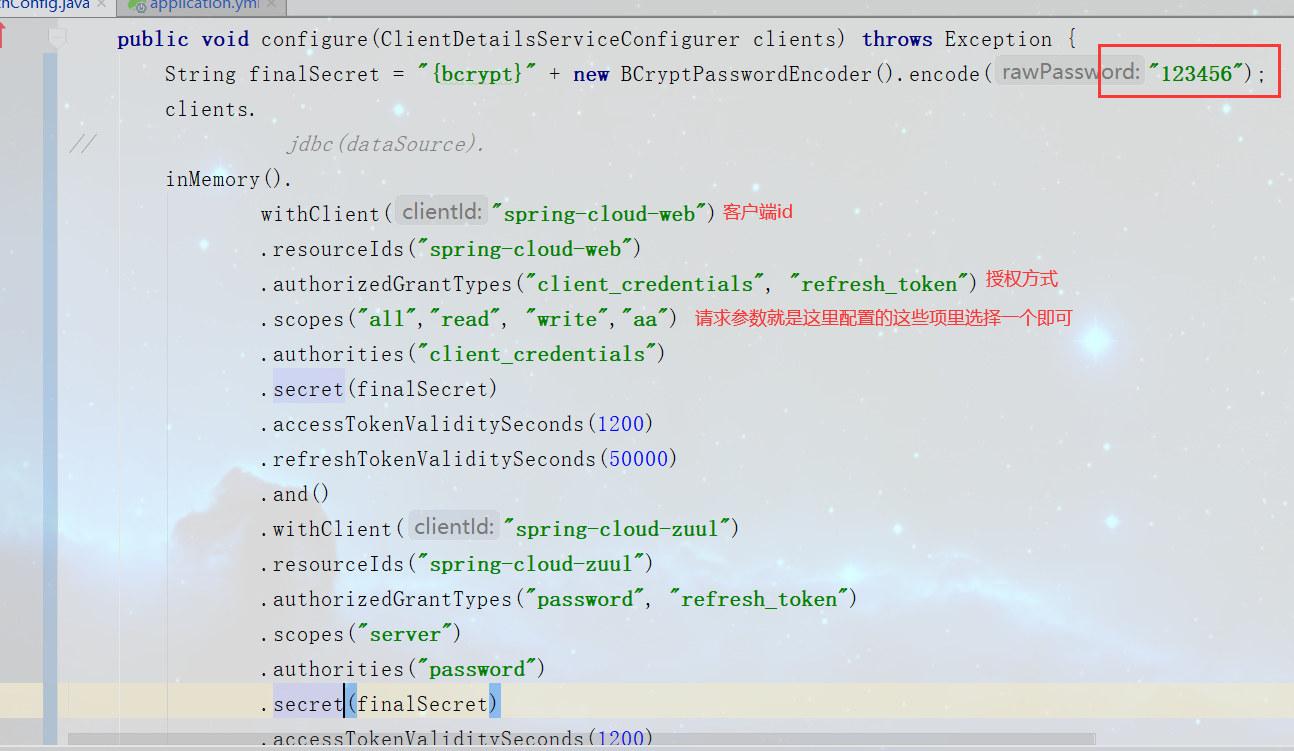
## 认证服务器

### 搭建

### 获取token

**请求地址为：<http://localhost:9095/oauth/token>**





请求的结果如下：

