[一、整体介绍](#5919-1588580316473)

[二、概念学习](#8685-1588577725643)

[2.1 SpringCloud](#4446-1588577970745)

[2.2 Eureka](#5099-1588579363696)

[2.3 Ribbon](#2013-1588577994613)

[2.4 Feign](#2575-1588669478299)

[2.5 Hystrix](#2510-1588580458248)

[2.6 HystrixDashbord](#6254-1588663249581)

[2.7 Config](#5880-1588663191608)

[2.8 Zuul](#8060-1588663204366)

[2.9 Sleuth ZipKin](#6654-1588581296618)

[2.10 RabbitMQ](#5338-1588582097029)

[三、实战](#6218-1588582429604)

[3.1 springboot学习补充](#4038-1588663806468)

[3.11 Springboot配置获取](#1412-1588663986439)

[3.3.2 多环境配置](#6580-1588663986439)

[3.1.3 maven依赖文件下载慢问题的解决](#3324-1588663986439)

[3.1.4 怎样构建一个springboot项](#2980-1588664133915)

[3.2 Eureka](#1560-1588664085773)

[3.2.1 添加依赖](#1244-1588664266977)

[3.2.2 新增配置文件](#5911-1588582369499)

[3.2.3 添加注解](#5765-1588664866021)

[3.3.4 添加对peer1 peer2 的转换](#1333-1588664931796)

[3.3.5 测试](#4017-1588580260938)

[3.3 Ribbon](#8013-1588665635860)

[3.1 新建两个服务提供类](#9958-1588667387826)

[3.2 新建ribbonconsumer](#1620-1588667506985)

[3.4 Feign](#4423-1588667884495)

[3.5 Hystrix](#1026-1588668028887)

[3.5 Config](#9452-1588667647464)

[3.5.1搭建一个配置中心config-cloud](#2198-1588673511024)

[3.5.2 获取配置](#2575-1588673699989)

[3.6 zuul](#7492-1588674753374)

[3.7 Sleuth 和 Zipkin](#3643-1588675194143)

[3.7.1 添加依赖](#5752-1588675133819)

[3.7.2 新建zipkin项目](#2619-1588675517768)

[3.8 RabbitMQ 构建](#5375-1588675821567)

[3.8.1 安装Erlang](#3080-1588733038992)

[3.8.2 安装RabbitMQ](#7785-1588733130568)

[3.8.3 测试登录](#9678-1588733344108)

[3.8.4 测试连接RabbitMQ](#5899-1588731456389)

[3.9 Spring 整合 RabbitMQ](#4074-1588734258190)

**一、整体介绍**

这个项目是学习SpringCloud的项目，首先您应该先从github上(<https://github.com/kzdw/springcloud>)拉取此项目导入到自己的idea中。

下面是对每个moudle介绍，如果你是第一次接触SpringCloud，可以先忽略这部分介绍。

直接进入到概念学习。

1.1 eurekaserver1、eurekaserver2 是eureka注册服务中心，两者形成集群

1.2 serverprovider9001、serverprovider9002 两个提供服务的moudle主要提供给feignconsumer 和 ribbonconsumer 消费使用，两者形成集群

1.3 ribbonconsumer 示例使用负载均衡，失败重试，ping等功能

1.4 feignconsumer 示例怎么样使用feign优雅地调用其他微服务

1.5 hystrix-dashboard 示例怎样搭建一个hystrix的界面化面板

1.6 config-cloud 示例怎样创建一个配置服务中心，连接git，为其他moudle提供配置服务

1.7 config-cloud-client 示例怎样连接配置中心

1.8 zuul 示例怎样使用zuul实现微服务的统一访问

1.9 zipkin-server 示例怎样界面化**展示**微服务调用链（数据是由sleuth提供）

1.10 rabbitmq-demo 示例怎样使用RabbitMQ

1.11 rabbitmq-handler、rabbitmq-message-sender 示例rabbitMq和springboot整合

**二、概念学习**

**2.1 SpringCloud**

微服务就是把原本臃肿的一个项目的所有模块拆分开来并做到互相没有关联，甚至可以不使用同一个数据库。 比如：项目里面有serverprovider9001模块和ribbonconsumer 模块，但是ribbonconsumer 模块和ribbonconsumer 模块并没有直接关系，仅仅只是一些数据需要交互，那么就可以吧这2个模块单独分开来，当ribbonconsumer 需要调用serverprovider9001的时候，serverprovider9001是一个服务方，但是serverprovider9001需要调用ribbonconsumer 的时候，ribbonconsumer 又是服务方了， 所以，他们并不在乎谁是服务方谁是调用方，他们都是2个**独立的服务。**可以理解为一个卖裤子的老板和一个卖衣服的老板，他们都是独立的，只是能提供这个服务而已，谁需要被人的产品（服务），那么自己就是消费者，别人就是服务方

微服务只是一种项目的架构方式，或者说是一种概念，就如同我们的MVC架构一样， 那么Spring-Cloud便是对这种技术的实现。

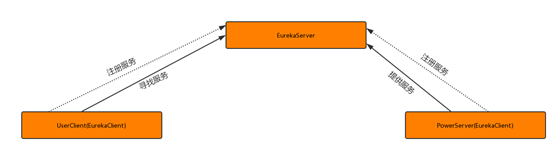
**2.2 Eureka**

2.1 基本概念

Eureka简单的说就是服务的注册于发现。

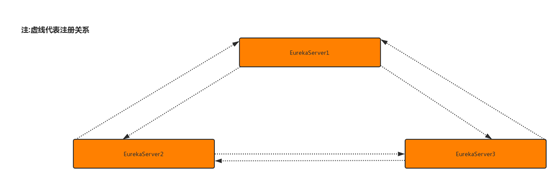
Eureka 采用了 C-S 的设计架构。Eureka Server 作为服务注册功能的服务器，它是服务注册中心。

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。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



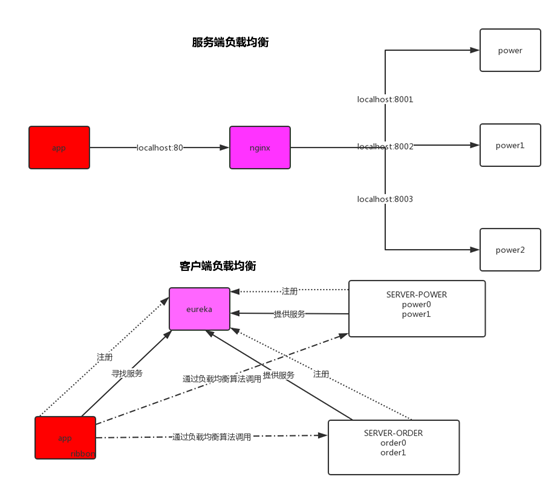
2.2 Eureka 高可用

服务启动后向Eureka注册，Eureka Server会将注册信息向其他Eureka Server进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。



**2.3 Ribbon**

Ribbon 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。





**2.4 Feign**

Feign可以优雅地调用其他微服务。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。

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，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。

**2.5 Hystrix**

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

降级，超时 :

我们先来解释一下降级,降级是当我们的某个微服务响应时间过长，或者不可用了，讲白了也就是那个微服务调用不了了，我们不能把错误信息返回出来，或者让他一直卡在那里，所以要在准备一个对应的策略（一个方法）当发生这种问题的时候我们直接调用这个方法来快速返回这个请求，不让他一直卡在那 。

熔断,限流:

讲完降级，我们来讲讲熔断，其实熔断，就好像我们生活中的跳闸一样， 比如说你的电路出故障了，为了防止出现大型事故 这里直接切断了你的电源以免意外继续发生, 把这个概念放在我们程序上也是如此， 当一个微服务调用多次出现问题时（默认是10秒内20次当然 这个也能配置），hystrix就会采取熔断机制，不再继续调用你的方法（会在默认5秒钟内和电器短路一样，5秒钟后会试探性的先关闭熔断机制，但是如果这时候再失败一次{之前是20次} 那么又会重新进行熔断） 而是直接调用降级方法，这样就一定程度上避免了服务雪崩的问题

**2.6 HystrixDashbord**

Hystrix（注意 是单纯的Hystrix） 提供了对于微服务调用状态的监控（信息）， 但是，需要结合

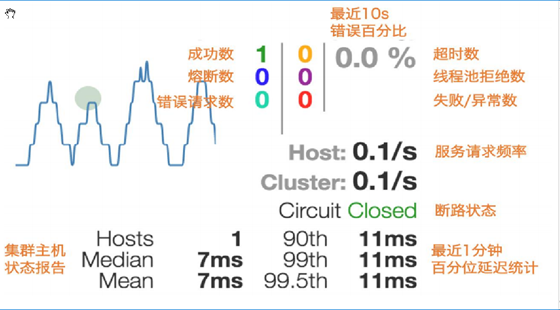
spring-boot-actuator 模块一起使用来通过可视化界面直观查看。HystrixDashbord主要就是进行图形化。

实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，它的健康度从绿色

该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，就可以在大量的实例中快速的发现故障实例和高压力实例。

曲线：用来记录2分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。

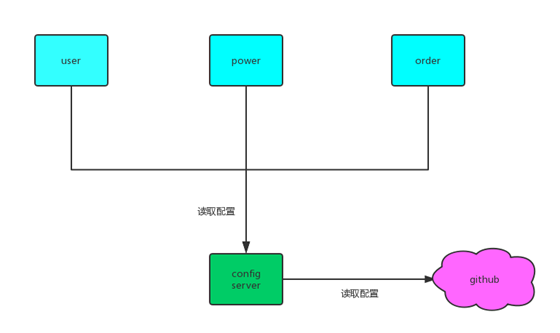
整图解释：



**2.7 Config**

我们既然要做项目， 那么就少不了配置，传统的项目还好，但是我们微服务项目， 每个微服务就要做独立的配置， 这样难免有点复杂， 所以， config项目出来了，它就是为了解决这个问题： 把你所有的微服务配置通过某个平台：

比如 github， gitlib 或者其他的git仓库 进行集中化管理（当然，也可以放在本地）.



看图就很好理解了。

**2.8 Zuul**

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

**2.9 Sleuth ZipKin**

在微服务架构系统中如何串联调用链，快速定位问题，如何厘清微服务之间的依赖关系，如何进行各个服务接口的性能分折，如何跟踪业务流的处理是很大的问题。那么sleuth的出现就是为了解决以上问题。

span（跨度）：基本工作单元。 span用一个64位的id唯一标识。除ID外，span还包含其他数据，例如描述、时间戳事件、键值对的注解（标签）， spanID、span父 ID等。 span被启动和停止时，记录了时间信息。初始化 span被称为"rootspan"，该 span的 id和 trace的 ID相等。

trace（跟踪）：一组共享"rootspan"的 span组成的树状结构称为 traceo trac也用一个64位的 ID唯一标识， trace中的所有 span都共享该 trace的 ID

annotation（标注）： annotation用来记录事件的存在，其中，核心annotation用来定义请求的开始和结束。

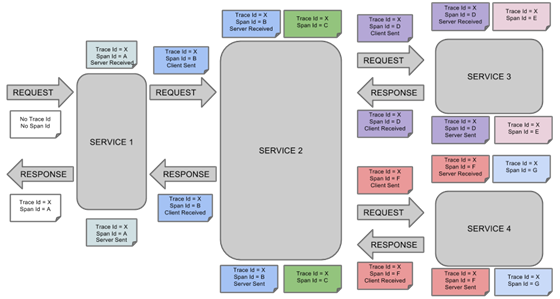
CS（ Client sent客户端发送）：客户端发起一个请求，该 annotation描述了span的开 始。

SR（ server Received服务器端接收）：服务器端获得请求并准备处理它。如果用 SR减去 CS时间戳，就能得到网络延迟。c)

SS（ server sent服务器端发送）：该 annotation表明完成请求处理（当响应发回客户端时）。如果用 SS减去 SR时间戳，就能得到服务器端处理请求所需的时间。

CR（ Client Received客户端接收）： span结束的标识。客户端成功接收到服务器端的响应。如果 CR减去 CS时间戳，就能得到从客户端发送请求到服务器响应的所需的时间

Spring Cloud Sleuth可以追踪10种类型的组件：async、Hystrix，messaging，websocket，rxjava，scheduling，web（Spring MVC Controller，Servlet），webclient（Spring RestTemplate）、Feign、Zuul



因为sleuth对于分布式链路的跟踪仅仅是一些数据的记录， 这些数据我们人为来读取和处理难免会太麻烦了，所以我们一般吧这种数据上交给Zipkin Server 来统一处理.Zipkin UI 来进行可视化展示。

**2.10 RabbitMQ**

消息中间键的作用：

交互系统之间没有直接的调用关系，只是通过消息传输，故系统侵入性不强，耦合度低。

例如原来的一套逻辑，完成支付可能涉及先修改订单状态、计算会员积分、通知物流配送几个逻辑才能完成；通过MQ架构设计，就可将紧急重要（需要立刻响应）的业务放到该调用方法中，响应要求不高的使用消息队列，放到MQ队列中，供消费者处理

通过消息作为整合，大数据的背景下，消息队列还与实时处理架构整合，为数据处理提供性能支持

RabbitMQ就是对于消息中间件的一种实现，市面上还有很多很多实现， 比如RabbitMQ、ActiveMq、ZeroMq、kafka，以及阿里开源的RocketMQ等等。

AMQP 协议中的基本概念：

•Broker: 接收和分发消息的应用，我们在介绍消息中间件的时候所说的消息系统就是Message Broker。

• Virtual host: 出于多租户和安全因素设计的，把AMQP的基本组件划分到一个虚拟的分组中，类似于网络中的namespace概念。当多个不同的用户使用同一个RabbitMQ server提供的服务时，可以划分出多个vhost，每个用户在自己的vhost创建exchange／queue等。

• Connection: publisher／consumer和broker之间的TCP连接。断开连接的操作只会在client端进行，Broker不会断开连接，除非出现网络故障或broker服务出现问题。

• Channel: 如果每一次访问RabbitMQ都建立一个Connection，在消息量大的时候建立TCP Connection的开销将是巨大的，效率也较低。Channel是在connection内部建立的逻辑连接，如果应用程序支持多线程，通常每个thread创建单独的channel进行通讯，AMQP method包含了channel id帮助客户端和message broker识别channel，所以channel之间是完全隔离的。Channel作为轻量级的Connection极大减少了操作系统建立TCP connection的开销。

• Exchange: message到达broker的第一站，根据分发规则，匹配查询表中的routing key，分发消息到queue中去。常用的类型有：direct (point-to-point), topic (publish-subscribe) and fanout (multicast)。

• Queue: 消息最终被送到这里等待consumer取走。一个message可以被同时拷贝到多个queue中。

• Binding: exchange和queue之间的虚拟连接，binding中可以包含routing key。Binding信息被保存到exchange中的查询表中，用于message的分发依据。

Exchange的类型:

direct :

这种类型的交换机的路由规则是根据一个routingKey的标识，交换机通过一个routingKey与队列绑定 ，在生产者生产消息的时候 指定一个routingKey 当绑定的队列的routingKey 与生产者发送的一样 那么交换机会吧这个消息发送给对应的队列。

fanout:

这种类型的交换机路由规则很简单，只要与他绑定了的队列， 他就会吧消息发送给对应队列（与routingKey没关系）

topic:(因为\*在这个笔记软件里面是关键字，所以下面就用星替换掉了)

这种类型的交换机路由规则也是和routingKey有关 只不过 topic他可以根据:星,#（ 星号代表过滤一单词，#代表过滤后面所有单词， 用.隔开）来识别routingKey 我打个比方 假设 我绑定的routingKey 有队列A和B A的routingKey是：星.user B的routingKey是: #.user

那么我生产一条消息routingKey 为： error.user 那么此时 2个队列都能接受到， 如果改为 topic.error.user 那么这时候 只有B能接受到了

headers:

这个类型的交换机很少用到，他的路由规则 与routingKey无关 而是通过判断header参数来识别的， 基本上没有应用场景，因为上面的三种类型已经能应付了。

**三、实战**

**3.1 springboot学习补充**

**3.11 Springboot配置获取**

properties中的配置：

#搜索服务地址
solrURL=http://localhost:8080/solr

可以使用以下方式读取：

**@Value("#{constants.solrURL}")**

public String testUrl;

@RequestMapping(value = "/test", method = RequestMethod.GET)

@ResponseBody

public Result queryTest() {

System.out.println("testUrl:" + testUrl);

}

<https://www.cnblogs.com/Jason-Xiang/p/6396235.html>

**3.3.2 多环境配置**

我们开发的时候可能会有多个环境比如 开发 测试 线上，但是每个环境的配置可能不一样，为了方便springboot有默认的方式可以实现。

添加properties文件命名为 application-xxx.properties,例如：

application-dev.properties

application-test.properties

application-produce.properties

公共的配置任然放在application.properties里面，并且写上 spring.profile.active=dev来激活相应配置文件。

**3.1.3 maven依赖文件下载慢问题的解决**

大多数jar包来自国外 使用maven默认配置下载很费劲，这个时候可以在maven的配置文件（maven/config/setting.xml）里面加入以下代码（阿里云镜像），重新下载jar包，速度很快

<mirrors>

<mirror>

<id>nexus-aliyun</id>

<mirrorOf>\*</mirrorOf>

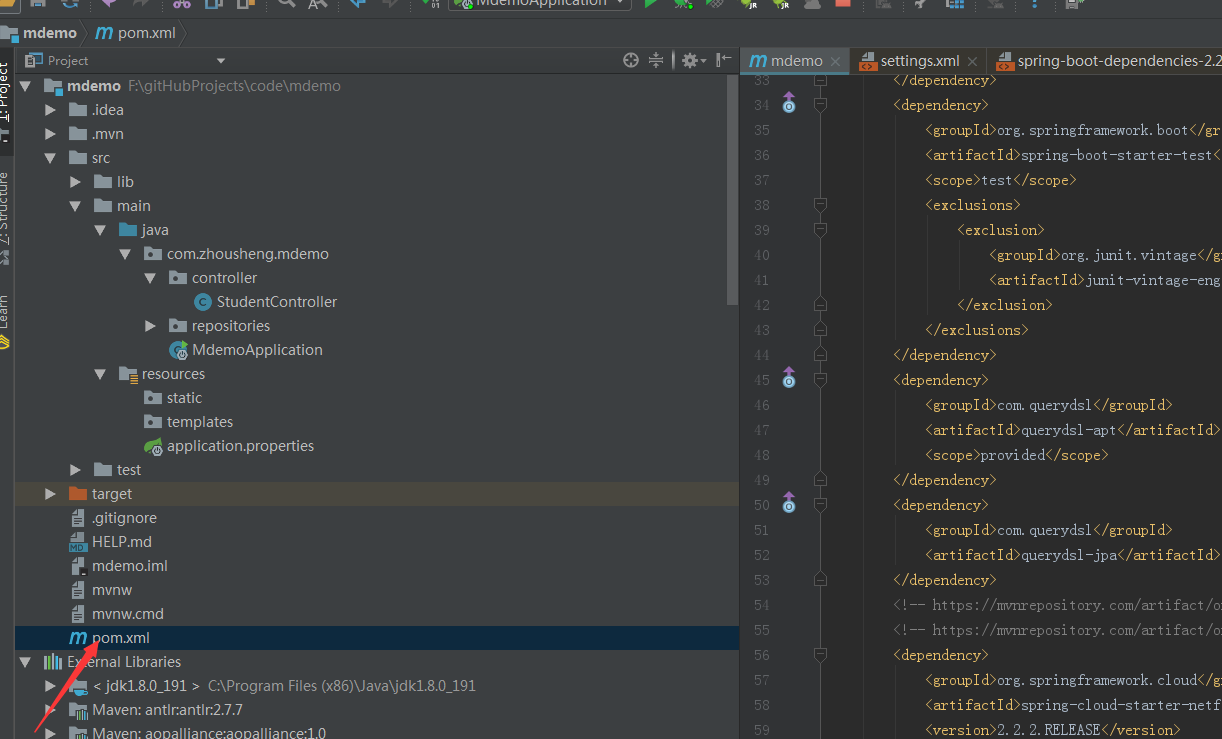
<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>;

</mirror>

</mirrors>

可能有些人的maven安装太久自己都找不到在哪里了，idea中添加setting很方便。



鼠标右键点击--->找到maven--->找到Creat"setting.xml"就可以在idea中编辑了。可能需要重启一下才能生效。妈妈再也不担心我的maven包下载慢了。

**3.1.4 怎样构建一个springboot项**

使用idea构建

这种方式最简单参考别人的 <https://baijiahao.baidu.com/s?id=1608301601154521261&wfr=spider&for=pc>

到官网去构建然后下载导入

<https://start.spring.io/>

先构建maven项目手动改造，这种方式太麻烦就不介绍了

**3.2 Eureka**

新建两个Springboot项目 eurekaserver1、eurekaserver2。

注意看springboot的版本，我这里使用的是 <version>2.0.3.RELEASE</version>，他可以对应springCloud的班本含有 <version>Finchley.SR2</version>。

注意springboot与springcloud的问题，遇到莫名其妙的错误的时候可以访问以下网址查看版本对应 <https://start.spring.io/actuator/info>（必须用Firefox打开才友好可读）

**3.2.1 添加依赖**

两个项目一样的依赖

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<!-- 引入eureka依赖-->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

</dependencies>

**3.2.2 新增配置文件**

新增peer1 和peer2 配置文件

spring.application.name=eureka-server

server.port=1111

eureka.instance.hostname=peer1

eureka.client.register-with-eureka=**true**

eureka.client.fetch-registry=**true**

eureka.client.serviceUrl.defaultZone=http://peer2:1112/eureka/

spring.application.name=eureka-server

server.port=1112

eureka.instance.hostname=peer2

eureka.client.register-with-eureka=**true**

eureka.client.fetch-registry=**true**

eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/

这里可以在两个eureka项目中都加入这两个配置然后通过spring.profiles.active=peer1这种形式来指定使用的配置，也可以分别放在两个项目中。关于以上配置可以阅读这篇文章来学习<https://www.cnblogs.com/xishuai/p/spring-cloud-eureka-safe.html>

**3.2.3 添加注解**

在启动类上添加注解

@SpringBootApplication

@EnableEurekaServer

public class Eurekaserver1Application {

public static void main(String[] args) {

SpringApplication.*run*(Eurekaserver1Application.class, args);

}

}

**3.3.4 添加对peer1 peer2 的转换**

找到 C:\Windows\System32\drivers\etc 的host文件

添加：

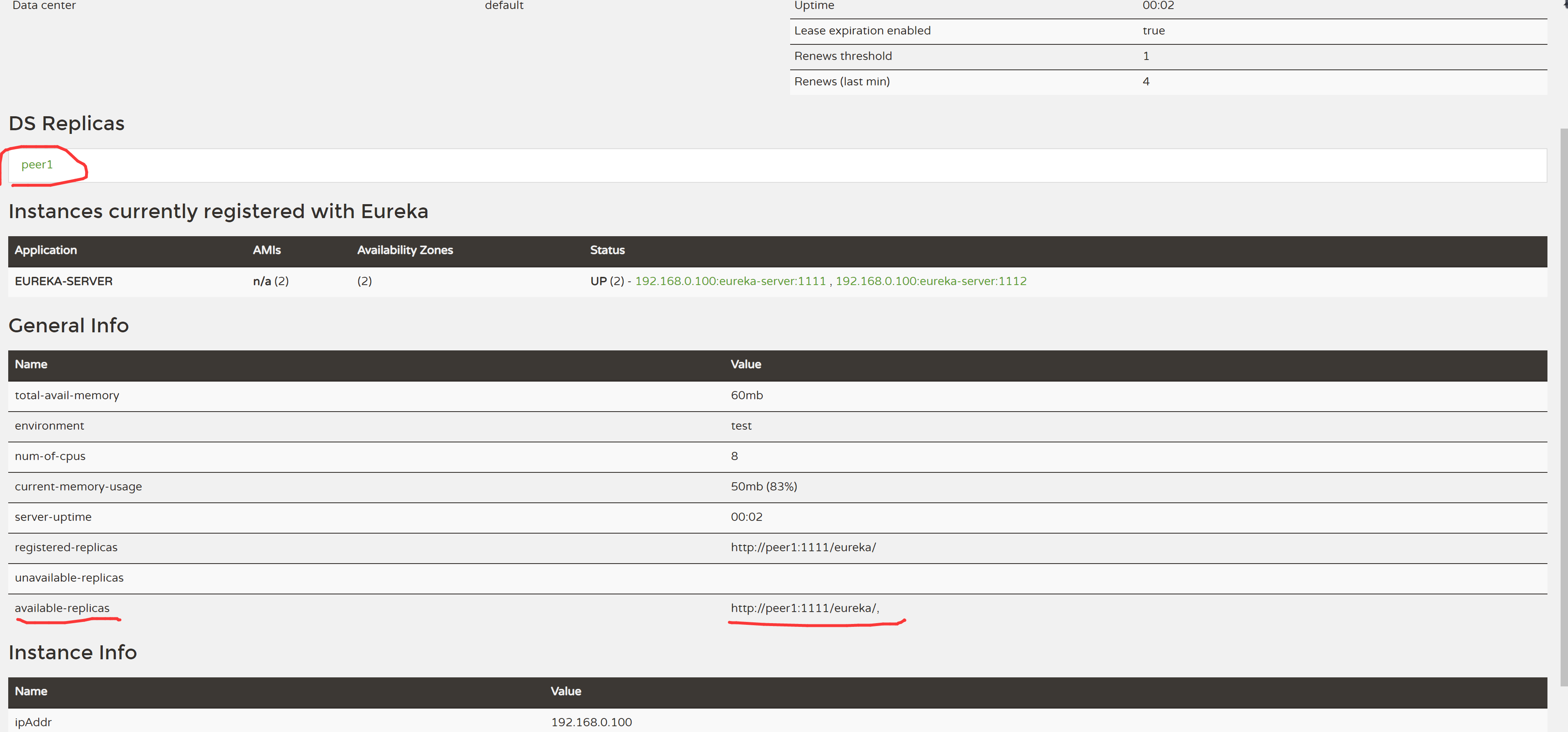
127.0.0.1 peer1

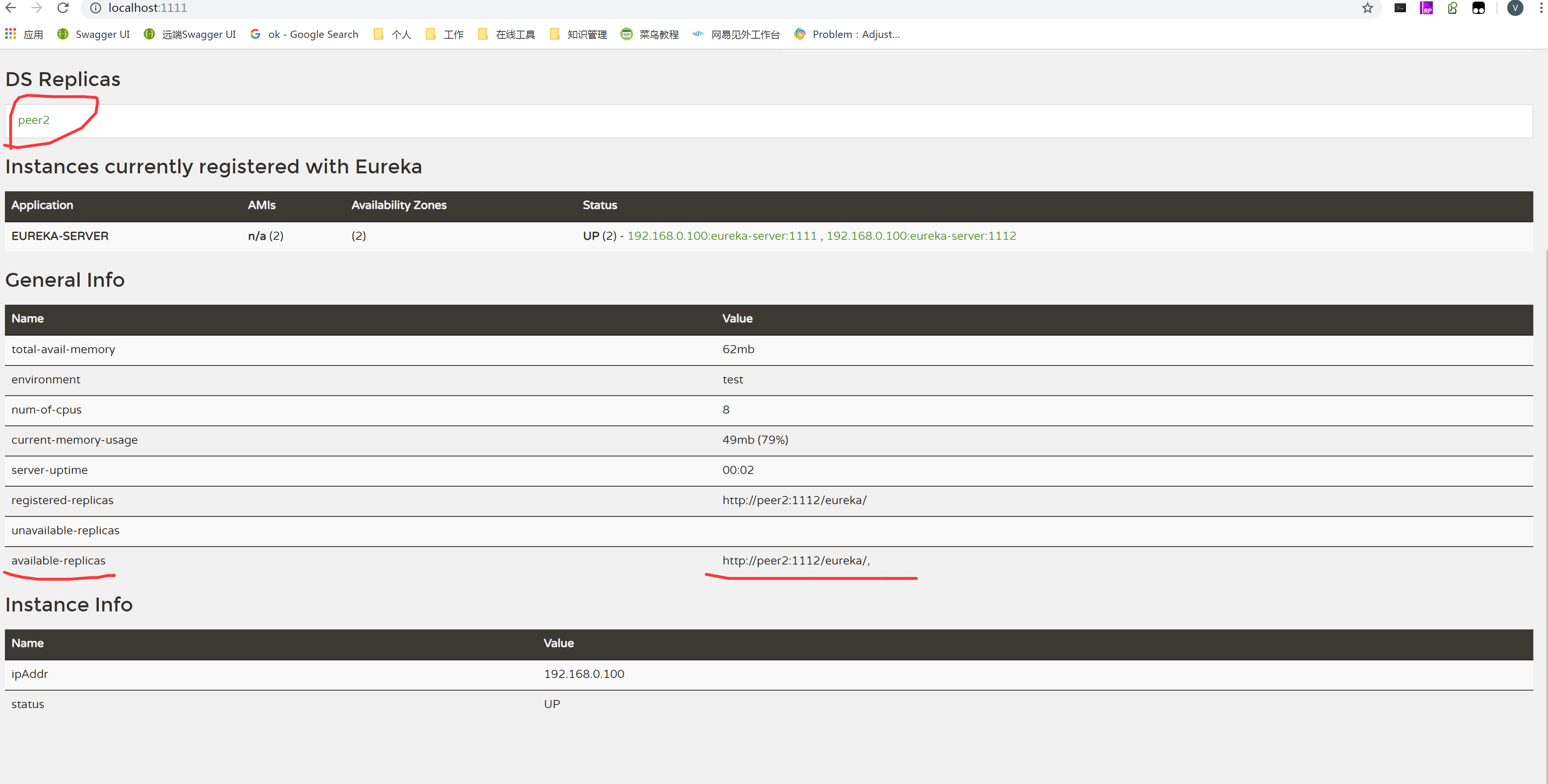
127.0.0.1 peer2

**3.3.5 测试**

分别启动两个项目

看到以下效果说明就成功了





**3.3 Ribbon**

**3.3.1 新建两个服务提供类**

首先新建两个服务提供类，注意最好两个API返回的值不一样，我们后面测试的时候才能看出效果。

具体操作省略了，请看serverprovider9001、serverprovider9002两个moudle即可

**3.3.2 新建ribbonconsumer**

**3.3.2.1新建ribbonconsumer**

新建moudle，其他配置什么的直接看项目，其实需要修改的就三个地方，**Pom文件、properties文件、启动类**，后面的项目除了特殊说明，关于这三个地方也不赘述了。

**3.3.2.2负载均衡**

@SpringBootApplication

@EnableDiscoveryClient

@EnableCircuitBreaker//开启熔断机制

//@SpringCloudApplication 也可以使用这个 里面包含了 @EnableDiscoveryClient @EnableCircuitBreaker

@EnableHystrix

public class ServerconsumerApplication {

@Bean

@LoadBalanced

RestTemplate restTemplate() {

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.*run*(ServerconsumerApplication.class, args);

}

}

在RestTemplate上添加 @LoadBalanced可以实现负载均衡。

如果需要使用其他负载均衡策略，可以在启动类中添加类似如下的代码

@Bean

public IRule iRule(){

    return  new RoundRobinRule();

}

我们测试如果能看到不一样的返回值就说明成功了。

**3.3.2.3 Hystrix降级**

编写服务类

@Service

public class RibbinconsumerService {

@Autowired

RestTemplate template;

@HystrixCommand(fallbackMethod = "findRibbonConsumerFallback")

public String findRibbonConsumer() {

String studentQuery = template.getForObject("http://serverprovider1/findStudent", String.class);

return studentQuery;

}

public String findRibbonConsumerFallback() {

return new String("student not found");

}

}

注意这个地方示例了怎样服务超时或不可用的时候怎么降级。findRibbonConsumerFallback()就是降级方法，说白了就是调用失败的时候掉用的。

**3.4 Feign**

@FeignClient("SERVERPROVIDER1")//直接指定那个服务提供

public interface IService {

@RequestMapping("/findStudent")

String findStudent();

}

feign的具体作用不说了，看代码就知道了，现在调用明显比ribbon更加优雅了，只需要定义接口，指定提供服务的类和mapping链接就可以了。

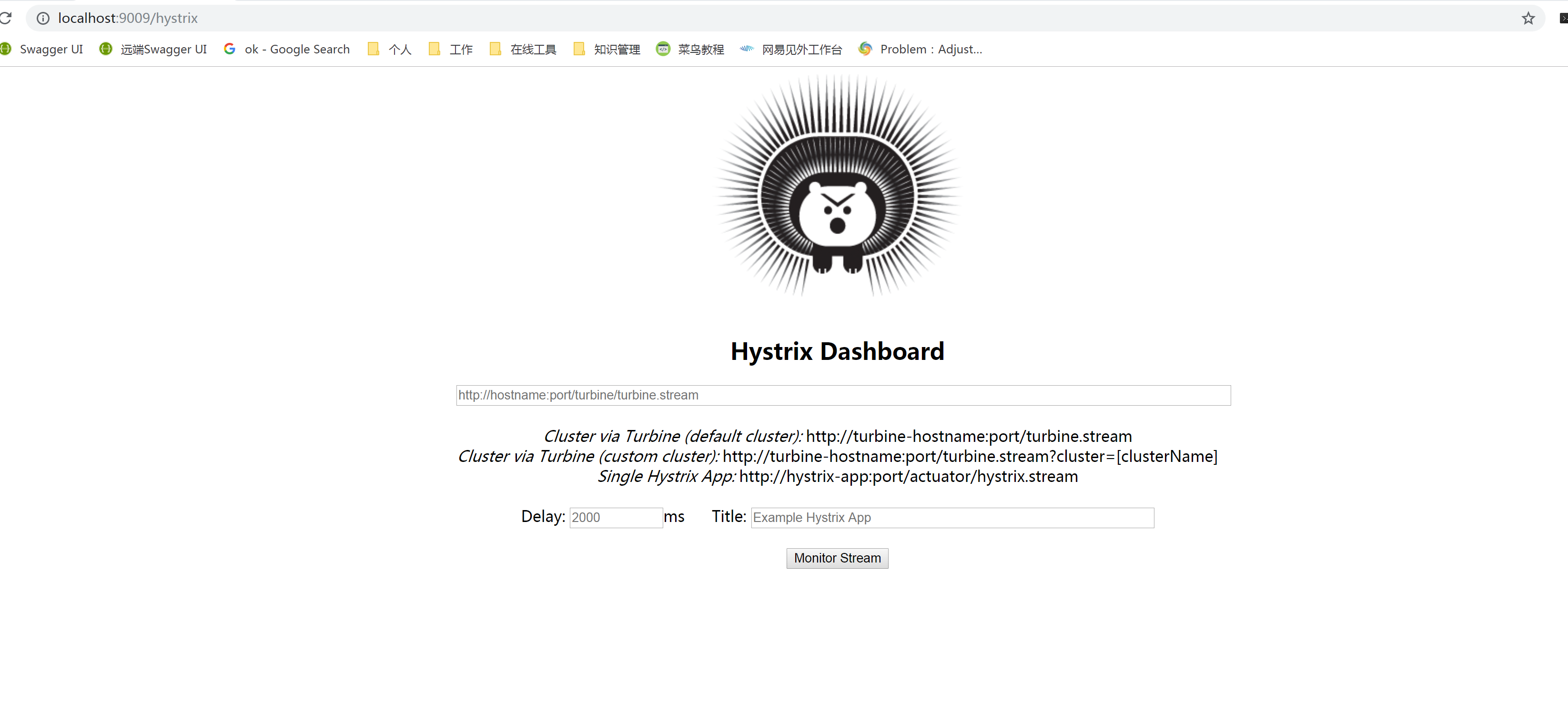
通过访问<http://localhost:9004/findStudent>，返回了不同的值，我们知道负载均衡起效了。

**3.5 Hystrix**

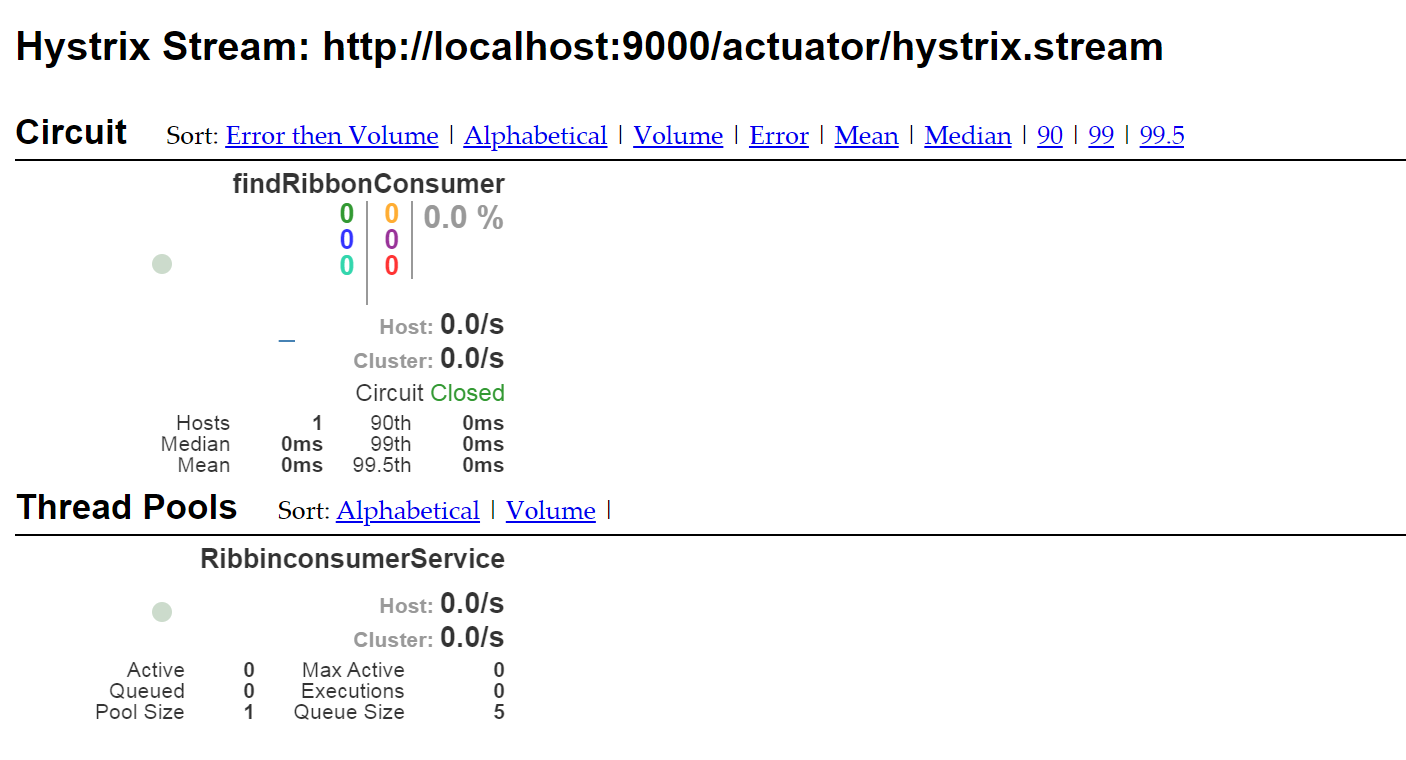
上面ribbonconsumer已经使用到了Hystrix的降级功能。如果需要可视话的监测需要使用Hystrix dashboard。

新建hystrix-dashboard moudle，配置按照项目里面写就可以了。然后启动项目。访问<http://localhost:9009/hystrix>

可以看到以下界面



然后输入需要监控的项目，<http://localhost:9000/actuator/hystrix.stream> 就可以看到以下结果



**3.5 Config**

**3.5.1搭建一个配置中心config-cloud**

需要说明的是这个配置中心的目的是去连接git，获取里面的配置

server.port=9009

spring.application.name=config-cloud

spring.cloud.config.server.git.uri=https://github.com/kzdw/springCloudConfigCenter.git

#如果是私有项目，那么就需要配置账户及密码

#spring.cloud.config.server.git.username=

#spring.cloud.config.server.git.password=

启动项目后，通过访问<http://localhost:9010/test-config.yml>，后面直接跟的是文件名，如果能过获取到配置，那么这一步就成功了

**3.5.2 获取配置**

新建项目，然后需要添加bootstrap.yml配置

**spring:**

**cloud:**

**config:**

**name:** test-config

**profile:** dev

**label:** master

**uri:** http://localhost:9010/

我们可以在原来的配置里面写上端口9100，以测试启动后的端口是不是任然是9100

启动项目，打印日志为Tomcat started on port(s): 8201 (http) with context path ''

说明没有使用本地配置了，使用了git上的配置

**3.6 zuul**

新建zuul moudle，新增配置

zuul.routes.api-a.path=/api-a/\*\*

zuul.routes.api-a.serviceId=feign-consumer

zuul.routes.api-b.path=/api-b/\*\*

zuul.routes.api-b.serviceId=ribbon-consumer

zuul.ignored-services="\*"

eureka.client.service-url.defaultZone=http://localhost:1111/eureka/

启动项目，我就可以以统一的方式去访问其他微服务了。<http://localhost:9006/api-a/findStudent>

**3.7 Sleuth 和 Zipkin**

**3.7.1 添加依赖**

首先，sleuth是从其他微服务中提取数据，所以我们的依赖应该加载需要监控的项目中，serverprovider9001、serverprovider9002、feignconsumer

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

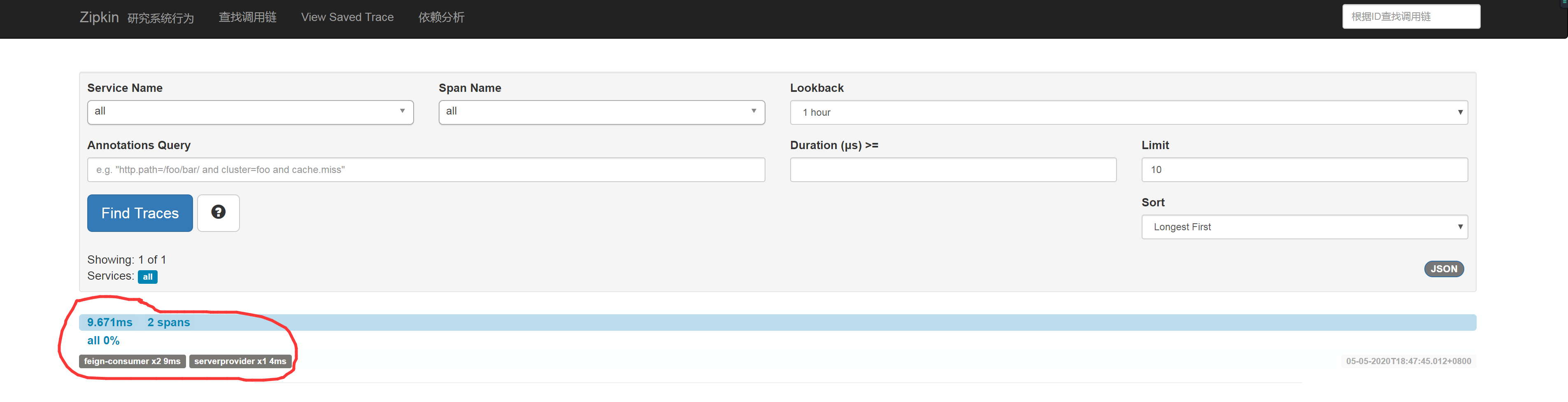
<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

**3.7.2 新建zipkin项目**

然后我们新建一个项目，zipkin-server，这个项目是可以单独启动的，<http://localhost:9019/zipkin/>，我们访问以下feignconsumer <http://localhost:9004/findStudent>。

再回去看<http://localhost:9019/zipkin/>



出现这样的效果就算成功了，点击进去就可以看到完整的调用链了。

**3.8 RabbitMQ 构建**

**3.8.1 安装Erlang**

首先我们要理解RabbitMQ是Erlang写的，所以在安装RabbitMQ之前，我们需要安装erlang环境。

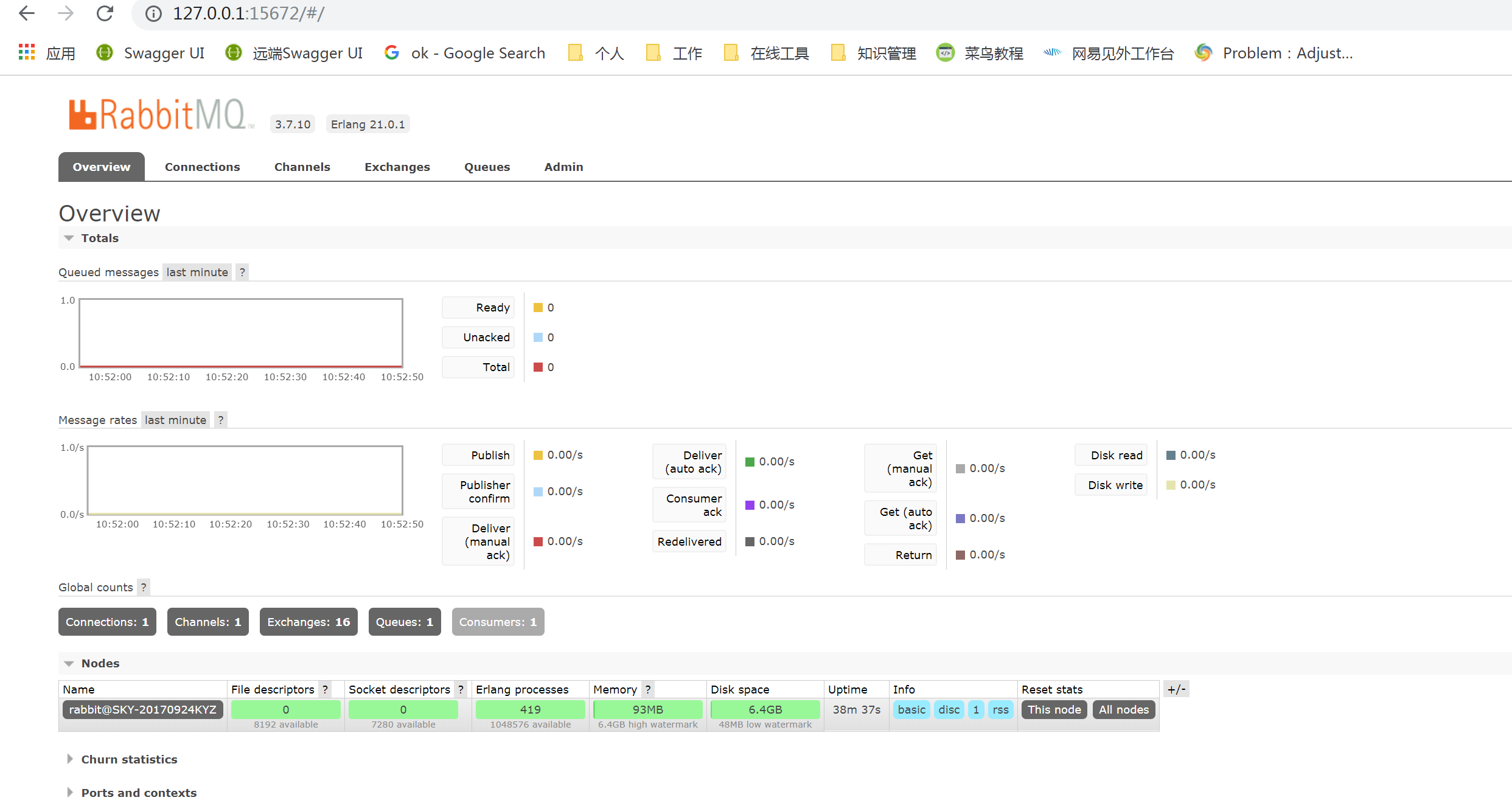
项目com/zhousheng/rabbitmqtest/erlangFile/otp\_win64\_21.0.1.exe已经上传Erlang安装文件。

**3.8.2 安装RabbitMQ**

安装RabbitMQ之前需要注意，Erlang和RabbitMQ有版本对应。请参见<https://blog.csdn.net/qq_36481502/article/details/105834879>

**3.8.3 测试登录**

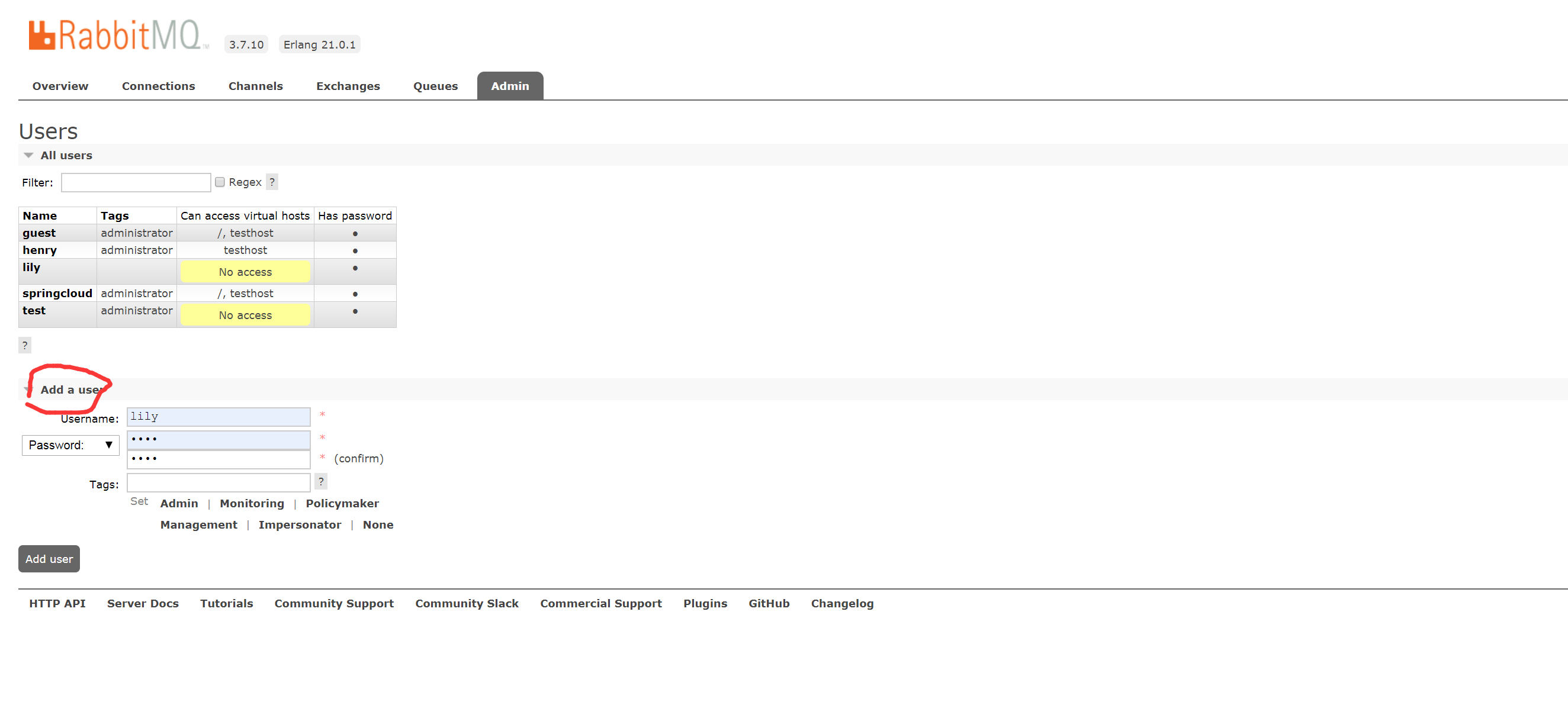
访问 <http://127.0.0.1:15672/#/>，使用默认的guest guest 登录



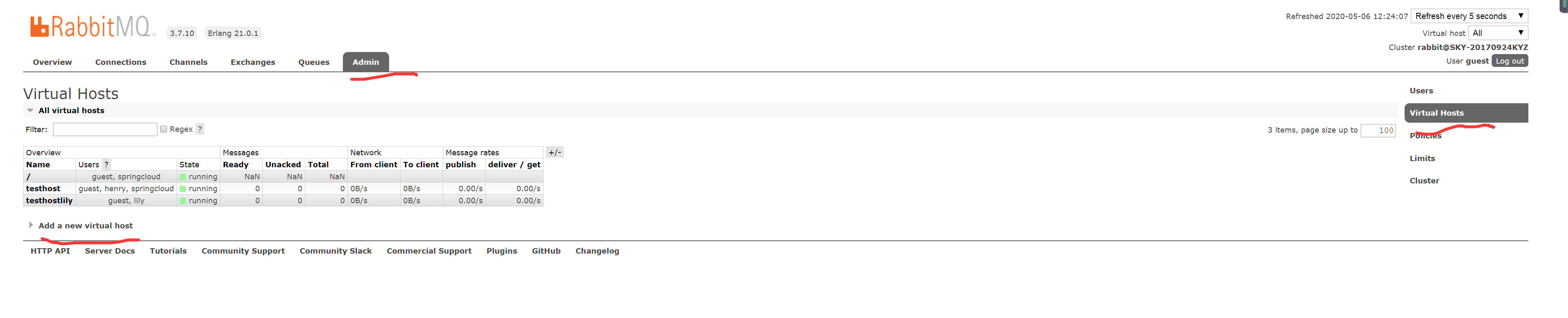
**3.8.4 测试连接RabbitMQ**

新建一个项目，老规矩引入pom文件，这里就不贴了。

首先新建一个用户

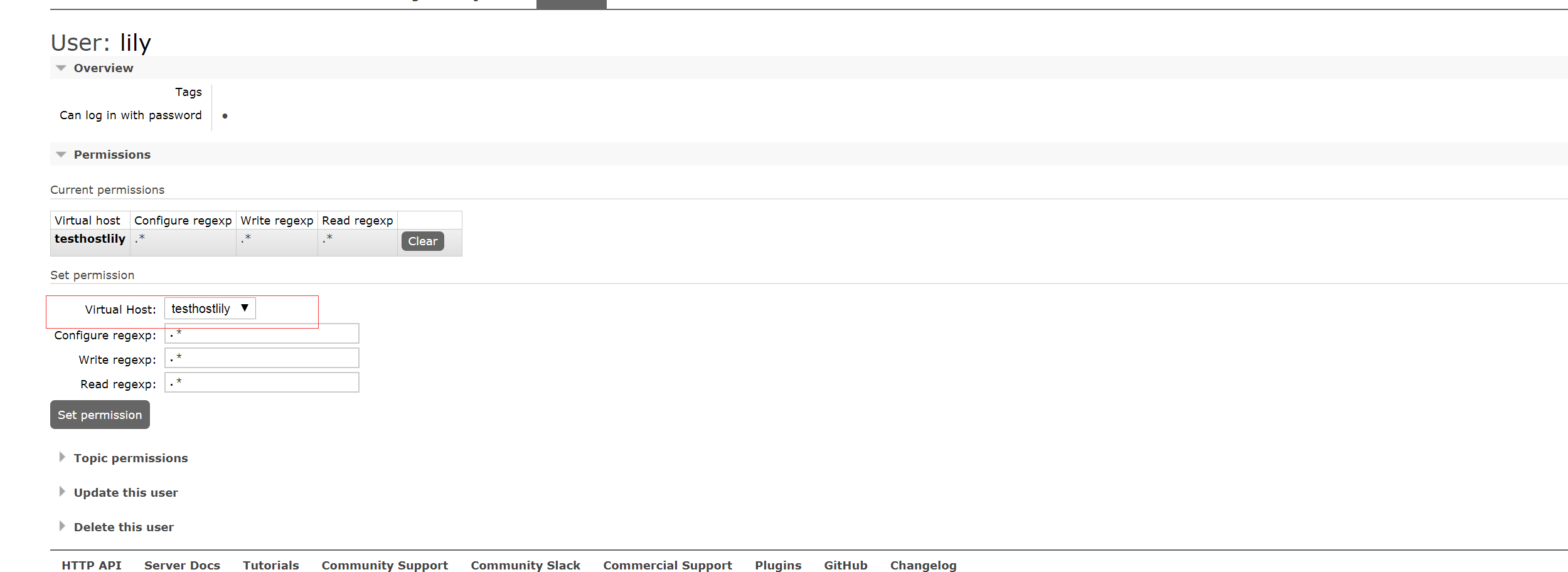


再创建一个vhost



设置权限

然后点击当前用户，设置当前用户的使用新vhost的权限



然后，建立连接

public static Connection getConnection() throws Exception{

//创建一个连接工厂

ConnectionFactory connectionFactory = new ConnectionFactory();

//设置rabbitmq 服务端所在地址 我这里在本地就是本地

connectionFactory.setHost("127.0.0.1");

//设置端口号，连接用户名，虚拟地址等

connectionFactory.setPort(5672);

connectionFactory.setUsername("lily");

connectionFactory.setPassword("lily");

connectionFactory.setVirtualHost("testhostlily");

return connectionFactory.newConnection();

}

}

然后启动producer main方法，再启动consumer main方法之后，能看到持续消费消息就说明成功了。

this is the message that i want to pass to you !71

消息消费成功

this is the message that i want to pass to you !72

消息消费成功

this is the message that i want to pass to you !73

消息消费成功

this is the message that i want to pass to you !74

**3.9 Spring 整合 RabbitMQ**

做了上面的工作之后，接下来的整合就比较简单了。老规矩，新建一个rabbitmq-message-sender

引入pom

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

连接配置

**spring:**

**rabbitmq:**

**host:** 127.0.0.1

**port:** 5672

**username:** lily

**password:** lily

**virtual-host:** testhostlily

**server:**

**port:** 9013

@Configuration

public class RabbitmqConfig {

@Bean

public DirectExchange defaultExchange() {

return new DirectExchange("directExchangeLily");

}

@Bean

public Queue queue() {

//名字 是否持久化

return new Queue("testQueueLily", true);

}

@Bean

public Binding binding() {

//绑定一个队列 to: 绑定到哪个交换机上面 with：绑定的路由建（routingKey）

return BindingBuilder.*bind*(queue()).to(defaultExchange()).with("studentInfoLily");

}

}

这样就可以了

至于消息接收端更简单，配置不说了。

消费直接通过注解就可以了

@Component

public class RabbitmqHandler {

@RabbitListener(queues = "testQueueLily")

public void get(String message) throws Exception{

System.*out*.println(message);

}

}