### MQ初步 & RabbitMQ & SpringAMQP

#### 前言

这里学习服务异步通讯的实用篇内容，RabbitMQ的基本信息，高级内容在后续学习中补充！

这里分三部分学习：

一是了解一下MQ，看一下这种异步通讯技术和之前学习的同步通讯技术有什么差别，以及各自的优缺点，以及在什么场景下该用社么！

二是了解MQ的一种具体实现，即RabbitMQ，看一下RabbitMQ的具体架构，安装RabbitMQ，通过一个入门案例手写代码体会一下MQ的异步通讯是如何完成的。

最后进入实战阶段，学习SpringAMQP，这是企业使用MQ最常用的方式，然后去整合RabbitMQ，看一下怎么更方便快捷实现消息的处理。

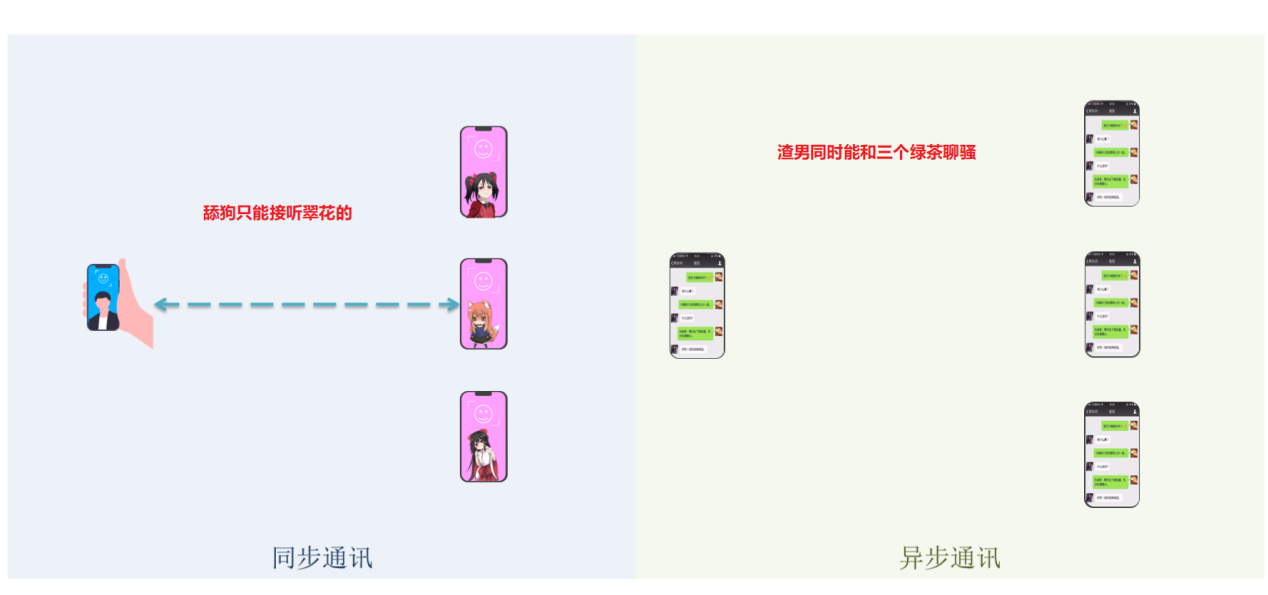
当然还会学习RabbitMQ的五种常见消息模型，以及在使用过程中可能碰到的问题。

#### 什么是同步通讯：

比如双方打电话，优点：等到妹子恢复再聊下一句；缺点：当其他妹子想和你聊天时，不好意思，进不来。

#### 什么是异步通讯：

比如双方微信聊天，优点：可以同时更N个妹子聊天，时间管理大师；缺点：时效性往往不太好，可能不能立即得到妹子的回复

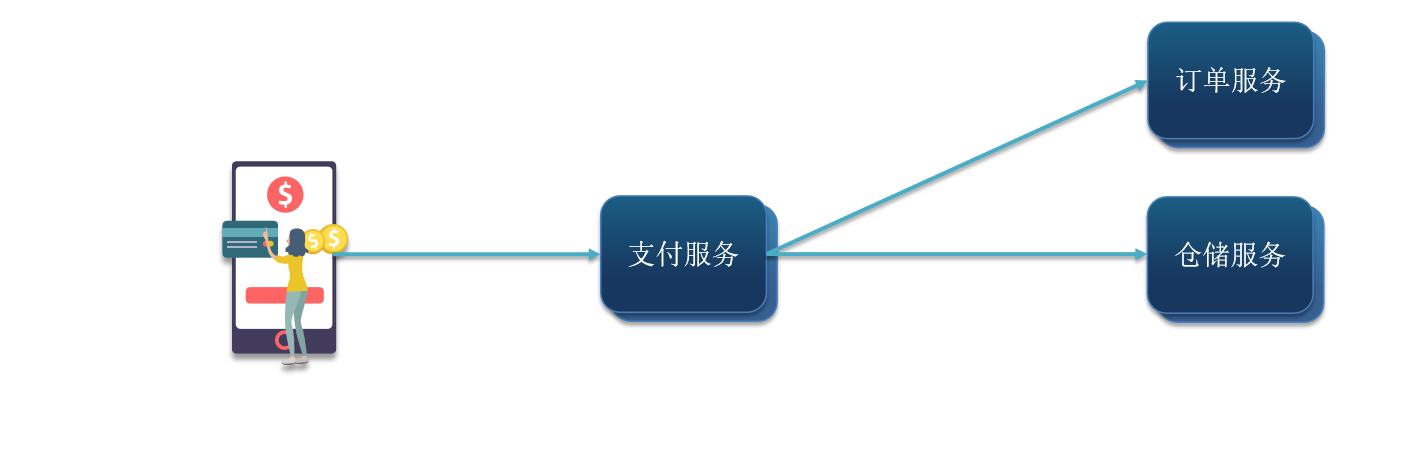


你可能会问，既然异步通讯时效性那么不好，为什么还要用呢？那是因为异步通讯有同步通讯没有的优点，我们在实际开发中，也会有同步通讯、异步通讯的场景，请看下面：

#### 同步调用的问题

之前学习的微服务间基于Feign的调用就是同步调用的情况，存在一些问题：

我们做一个购买商品支付的业务，用户支付调支付服务，支付成功之后需要调用订单服务，因为需要去修改订单状态啊，而后还要去调用仓储服务，因为你要给用户发货啊，这就是我们购物时的基本场景



在这个过程中，支付服务调用订单服务也好，调用仓储服务也好，都需要等待对方的响应，所以这种调用是实时的调用，即同步调用，存在问题：

##### 1：耦合高：

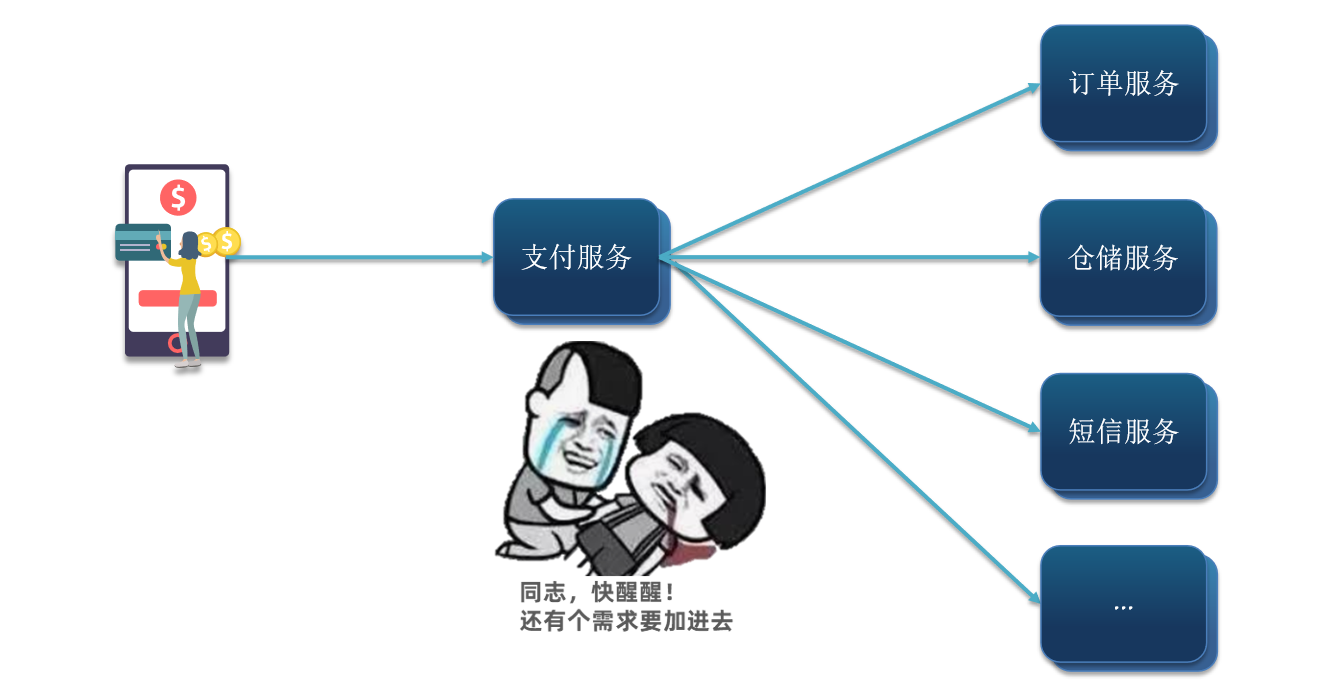
每次加入新的需求，都要修改原来的代码，比如产品经理脑门一拍，他觉得用户支付完了应该给用户发个短信通知一下，你看这就要加业务了，你要价格发短信的业务，得在支付服务里改动代码



你以为这样就结束了吗，产品经理又脑门一拍，他觉得业务还是可以升级嘛，他觉得可以加个优惠卷功能，用户付了款的时候用了优惠券，那么要把优惠券废弃，那又要加业务：

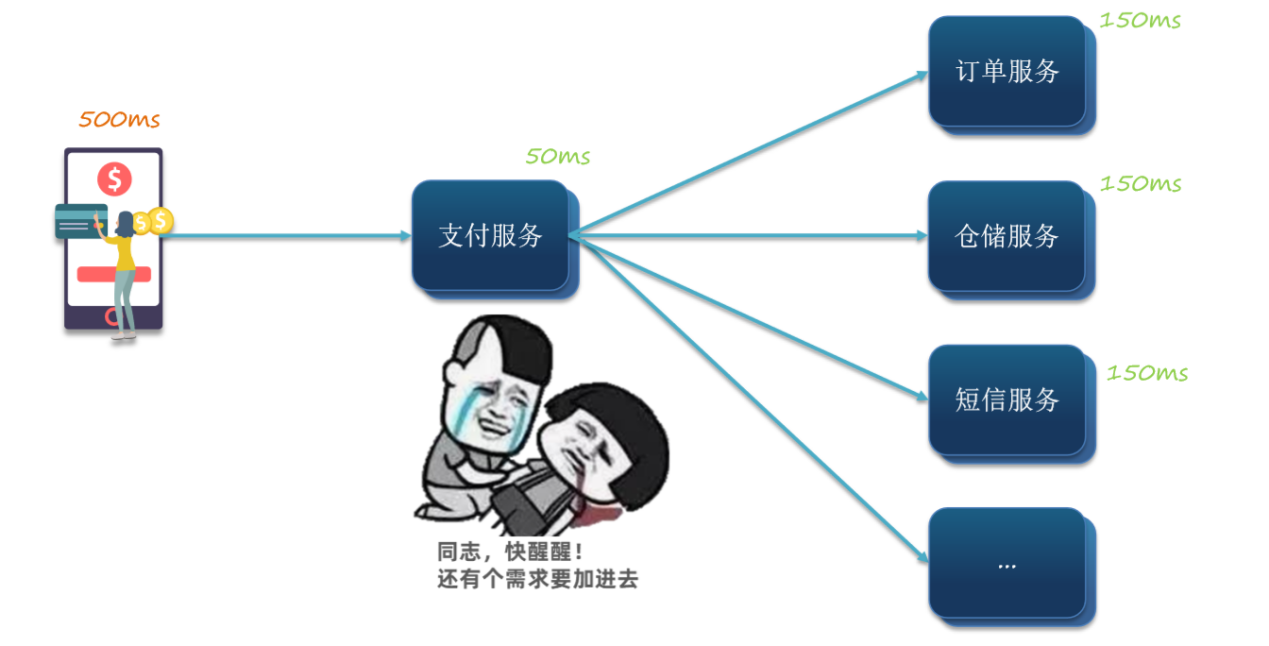


你以为完了吗？产品经理又作妖，他又要加积分系统，用户支付了要给用户加积分，吸引用户下次再来买，你看你看，每加一次业务，你的代码都得改一次，这样需求改来改去，谁受得了啊，这就是耦合带来的很严重的问题！



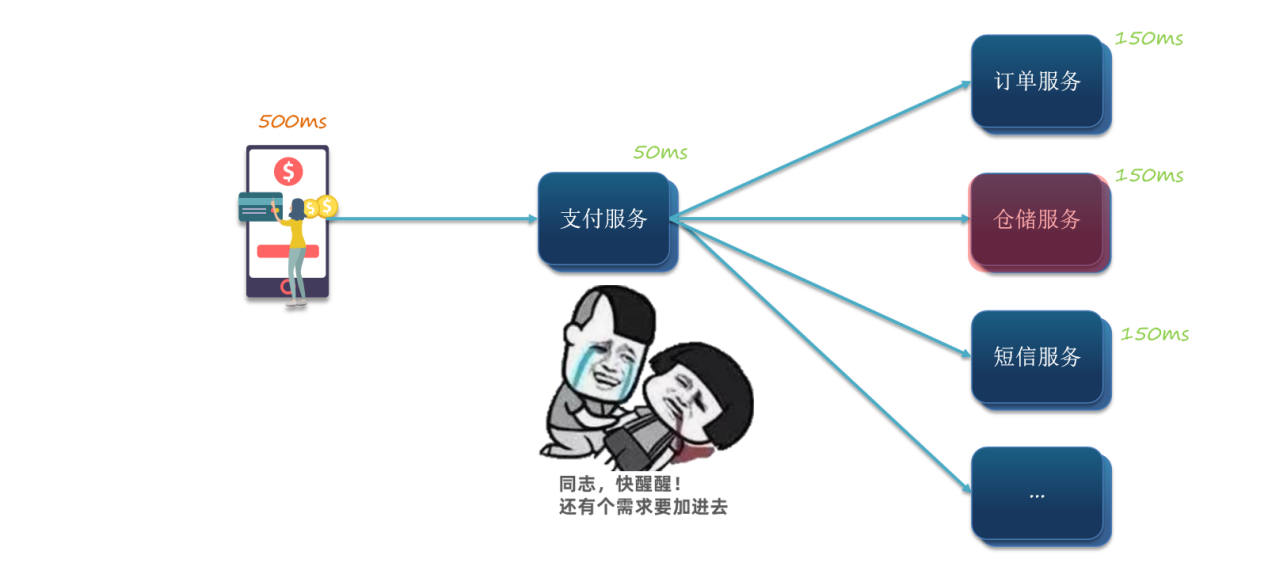
##### 2：性能下降：

调用者需要等待服务提供者响应，如果调用链过长则响应时间等于每次调用的时间之和；比如用户调用支付业务，假设支付业务耗时50ms，紧接着支付服务调用其他服务，都耗时150ms，支付服务调用订单服务是同步调用的，所以必须等待整个订单服务执行完成，才能执行仓储服务，也就是说支付服务等了150ms，然后再调仓储服务，然后又要等150ms，等仓储服务执行完成，再调用短信服务。。。。。因此整个流程下来总耗时就达到了很恐怖的500ms，也就是说你一秒钟只能处理2个请求，你这业务也忒烂了吧，数以十万百万的并发来了，你这业务能抗住吗！？这就是第二个问题；性能下降，吞吐量也下降



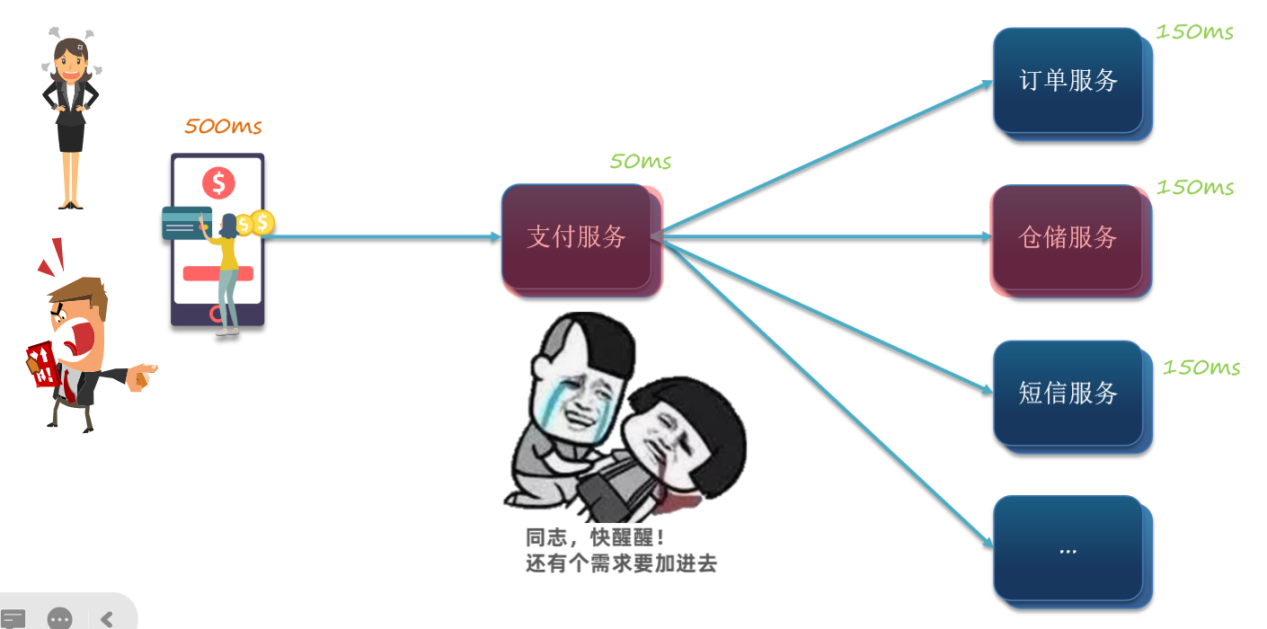
##### 3：资源浪费：

调用链中的每个服务在等待响应过程中，不能释放请求占用的资源，高并发场景下会极度浪费系统资源；比支付服务在等待订单服务执行的过程中，订单服务在干等着，CPU、内存都在占着，订单服务这个时候啥都不干，就占着、等着订单服务执行完毕再往下执行调用仓储服务，所以在整个等待过程中，支付服务有大量资源的浪费。这就是同步调用的第三个问题，资源的利用不够充分，有很多的资源浪费，一次的请求可能没啥，如果请求多了，一百个、一千个、一万个，你得浪费多少资源啊



##### 4：级联失败：

如果服务提供者出现问题了，所有的调用方都会跟着出问题，如同多米诺骨牌一样，迅速导致整个微服务故障。假设说仓储服务扛不住压力挂了，这个时候请求来访问仓储服务必然会阻塞，支付服务调用仓储服务的时候调不通，会卡住，卡住这个支付请求就不会释放，一个请求卡住，假以时日卡得越来越多请求，这个时候支付服务资源耗尽，请求再来就进不去支付服务了，支付服务等于也挂了，于是再有别的服务来调支付服务，别的服务也调不通，那么挂掉的服务越来越多，用户就要骂街了：什么垃圾网站，访问一下就挂了，速度还这么慢！



##### 总结：

1：同步调用的优点：

时效性较强，可以立即得到结果

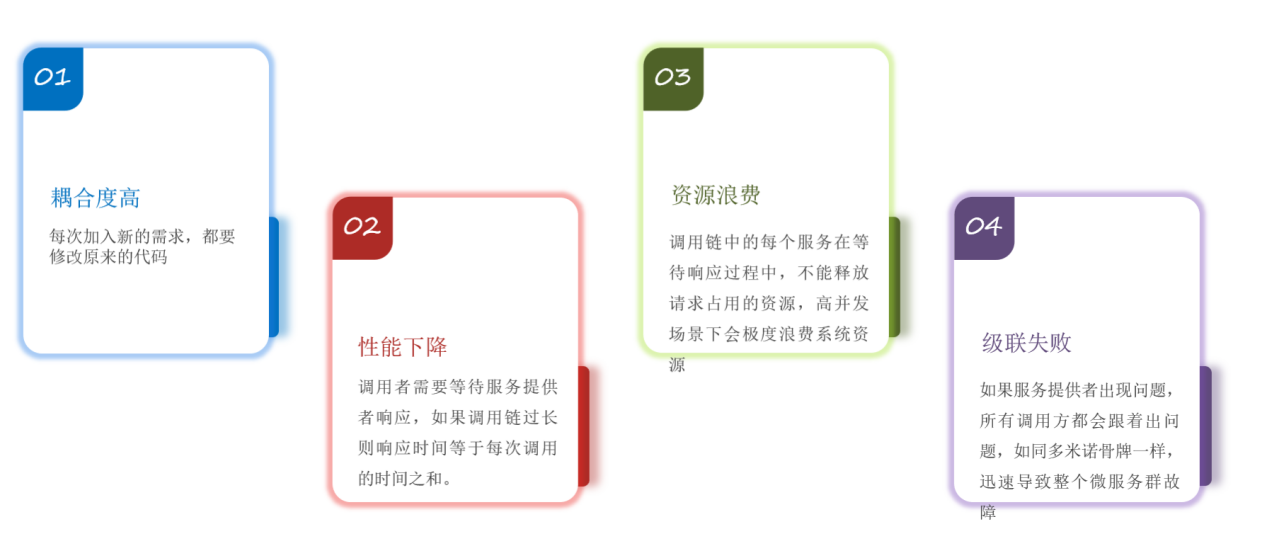
2：同步调用的问题：

耦合度高；

性能和吞吐能力下降；

有额外的资源浪费；

有级联失败问题。



#### 异步调用方案

上节了解了微服务间同步调用，并且了解了同步调用得优势和不足，这节继续学习微服务间得异步调用方案，异步调用比较常见实现就是事件驱动模式：

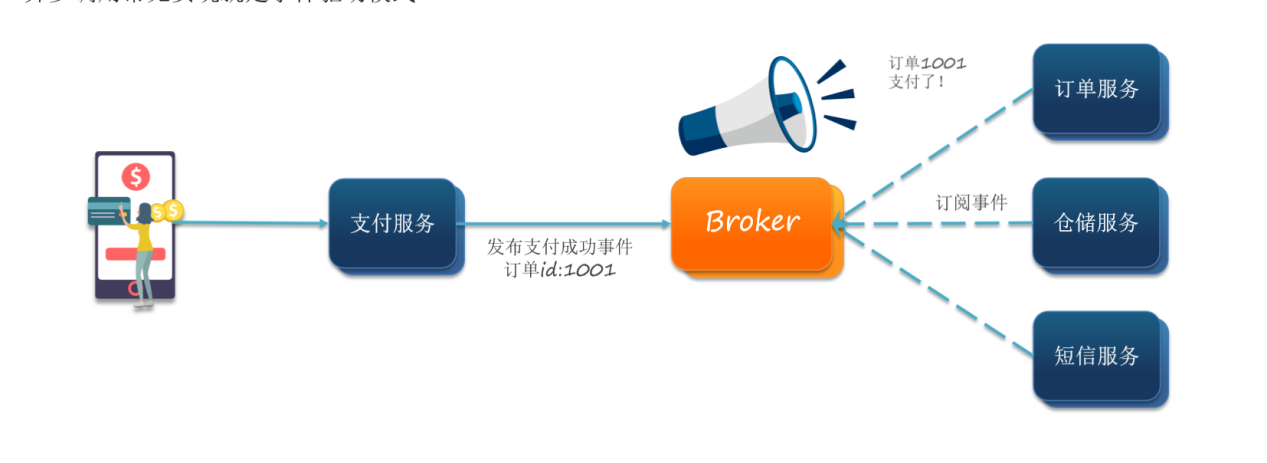
##### 事件驱动模式

###### Broker

通过案例说明：用户支付时，会调用支付服务，而支付服务在完成支付以后，后续需要订单服务、仓储服务、短信服务等各自完成对应自己的业务，而现在是事件驱动（异步调用），不能像同步调用一样由支付服务去调用订单服务、仓储服务、短信服务这些服务，所以引入一个概念叫Broker，Broker即事件代理者（也就是今天要学习的MQ）**。**

###### 事件订阅

**在我们这个业务中，一旦有人支付成功，就是一个事件**，那么这个事件就会交给Broker去管理，订单服务、仓储服务、短信服务就会去找Broker，当有人支付成功了，Borker要通知订单服务、仓储服务、短信服务这些服务，即事件订阅；而支付服务发现有人支付成功，发布一个事件，Borker就会通知订单服务、仓储服务、短信服务这些服务，订单服务就会去更新订单状态，仓储服务就回去完成库存扣减、发货，短信服务就会去完成短信发送。在整个过程中，支付服务完成事件发布之后就立即结束了自己的业务，可以返回结果给用户了，不需要等待订单服务、仓储服务、短信服务这些服务完成业务，因为支付服务已经把事件发不出去了，Broker负责通知订单服务、仓储服务、短信服务这些服务，支付服务不用管订单服务、仓储服务、短信服务这些服务什么时候执行业务、什么时候执行完业务。这就是异步得方式！



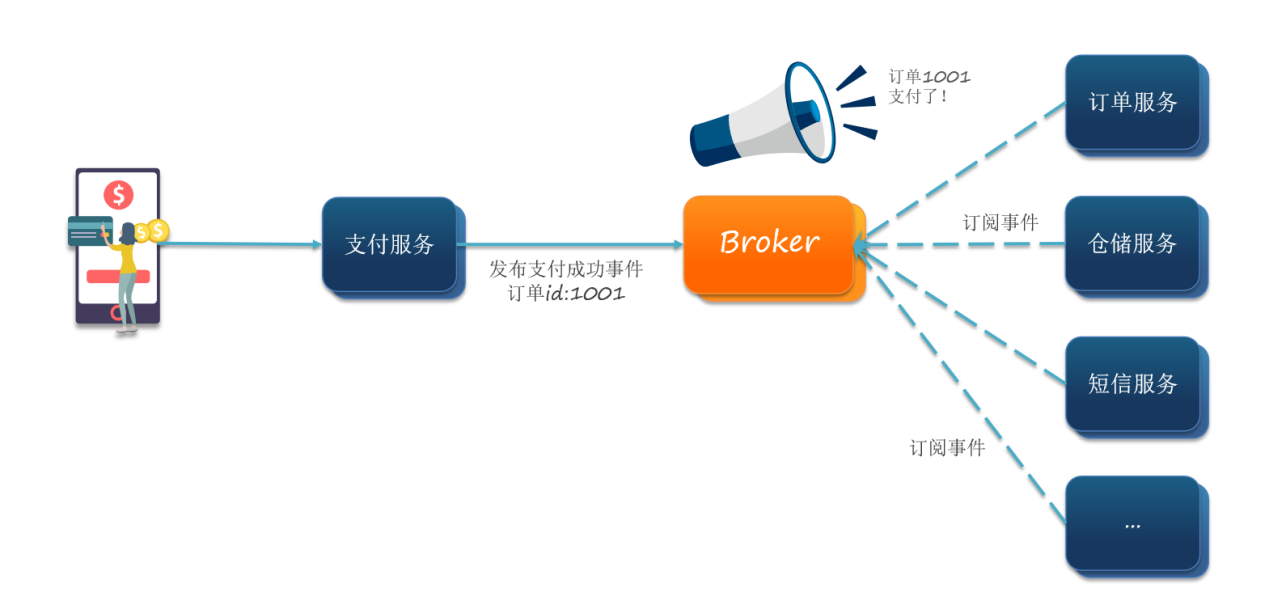
###### 总结：事件驱动模型实现得异步调用方案

A发送事件给Broker，其他人在Broker订阅这个事件，当Broker有事件时，这些人就能收到事件进行对应得处理逻辑，A发送完、发布完事件，就立即结束自己的业务，余下的工作由其他人完成。

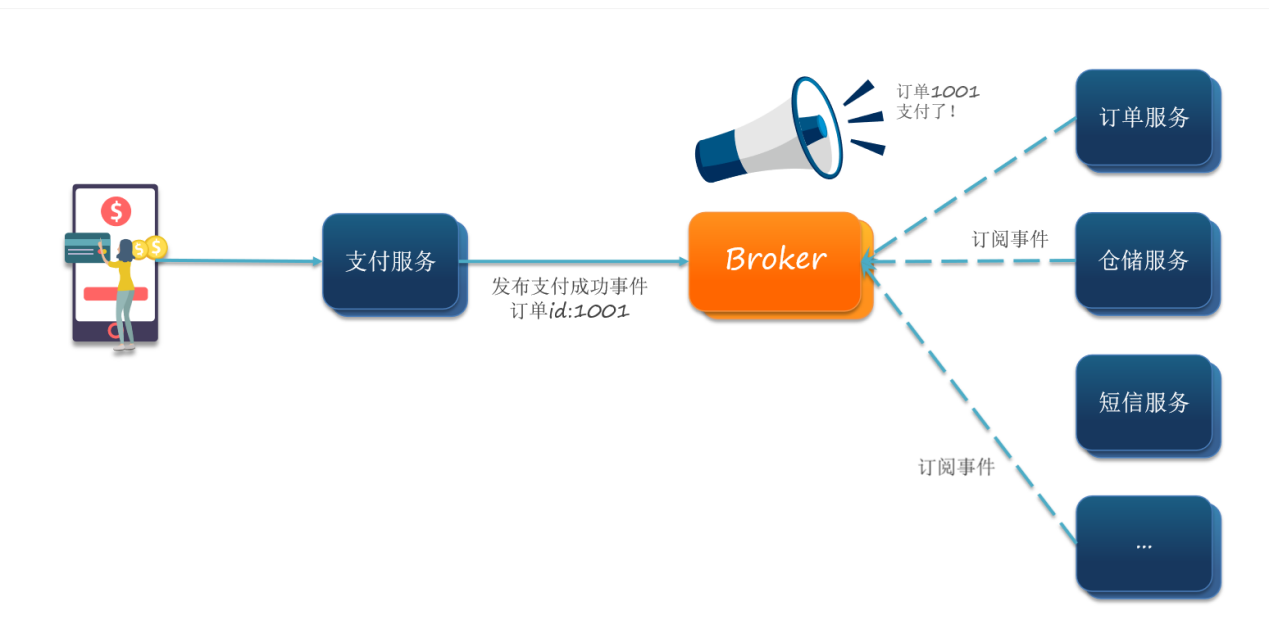
#### 异步调用方案优势：

##### 优势一：服务解耦；

同步调用当有新的需求，比如要新增积分服务，那么要改动支付服务，加上调用积分服务的代码，现在不用了，因为支付服务不负责调用这些服务，而是只发一个订单支付了的事件到Broker，至于谁接收事件，什么时候完成，跟支付服务没关系，支付服务压根不管。所以一旦有新的业务（服务）出现，只需要去订阅Broker事件就可以了，于是就解除了服务与服务之间的耦合了。

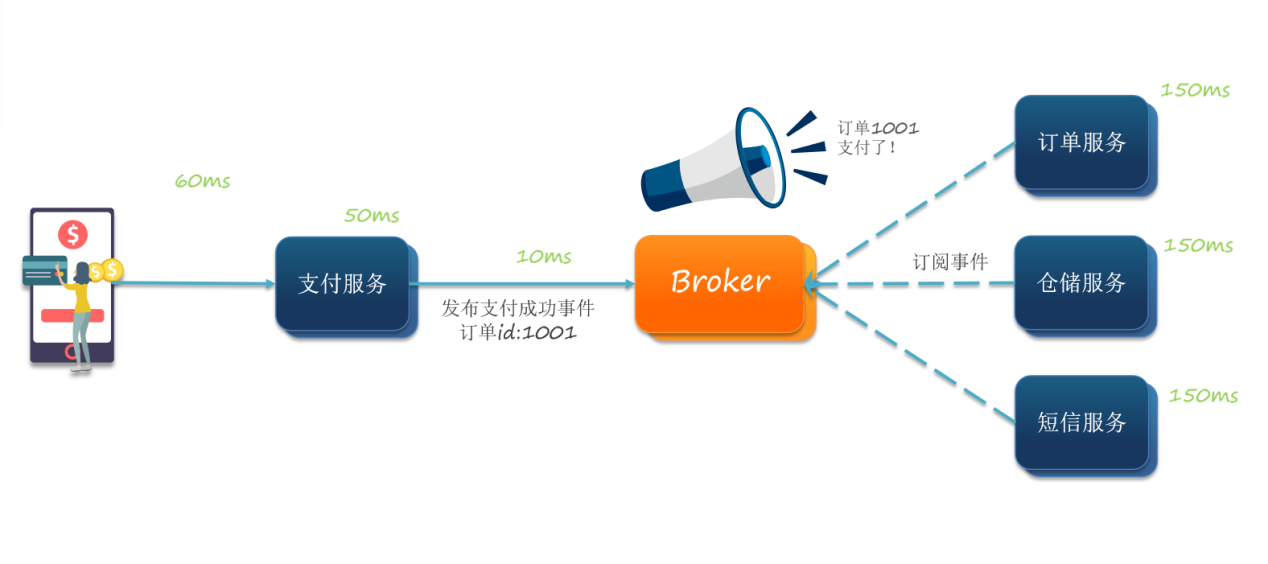


一样的，当有天不需要短信服务了，以前的同步调用方案需要去支付服务里删除代码，而现在的异步调用方案则只需要让短信服务取消Broker事件，将来支付事件通知就通知不到短信服务，那么短信就不发了。以上，无论是增加业务还是取消业务，都不需要修改支付服务的代码，你看你看你看，这不于就接触了服务之间的耦合了吗。



##### 优势二：性能提升，吞吐量提高：

以前的同步调用方案支付服务支付完成后，是要来调用订单服务、仓储服务、短信服务的，每个服务都有一定的耗时，因此总耗时就是每个服务的耗时之和；而异步调用方案则是支付服务支付完成后，向Broker发布事件，这个时候，支付服务就可以立即结束告诉用户支付成功了，因为确实是支付成功了，后续的订单、仓储、短信跟支付是没什么关系的（这里的意思就是，用户只关心支付成功没有，也就是说只关心第一步成功没有，至于后面的订单或者是短信通知相对来说不重要了，因为用户知道钱已经给了，如果有问题就再说，有问题再找客服，而且一般有问题的概率很小），这些动作由Broker去通知订单服务、仓储服务、短信服务去完成，这些动作什么时候完成、耗时多久跟支付服务没关系，只要这些动作最终能做完就行，因此业务总耗时就变成了支付服务里的耗时加发布事件的耗时，耗时缩短了，性能就提升了，吞吐量提高了！



##### 优势三：故障隔离，服务没有强依赖关系，不用担心级联失败问题：

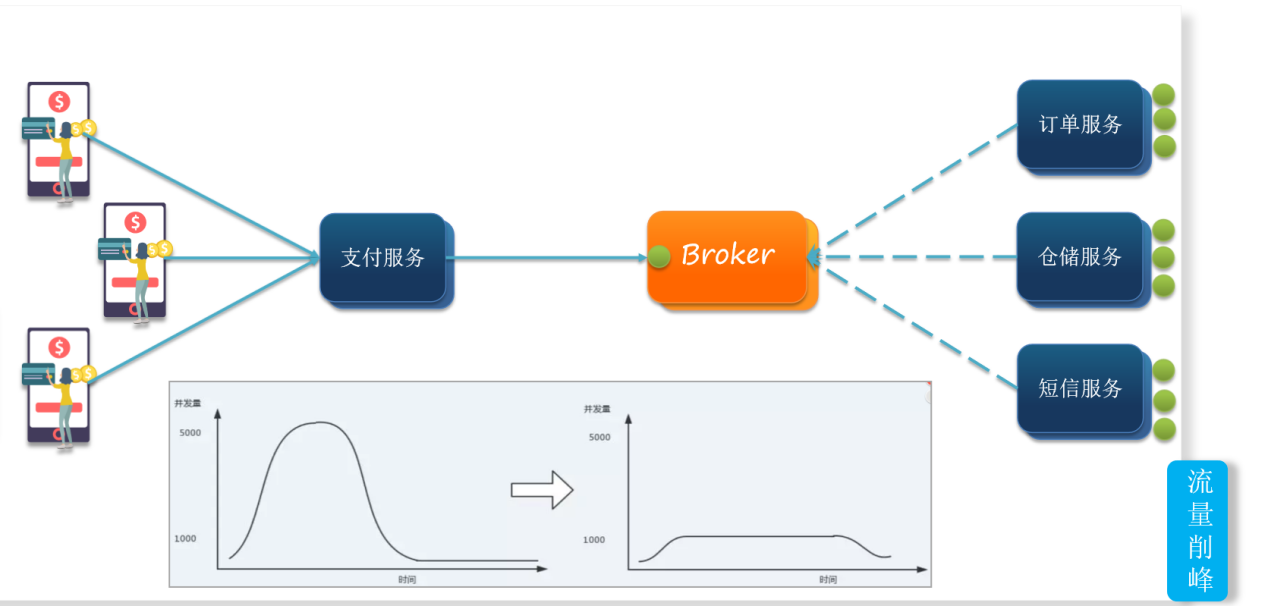
服务没有强依赖关系，同步通讯的级联失败问题也就不存在了，比如，现在仓储服务挂了，跟支付服务没关系，支付服务发完事件就结束了，支付服务又不调用仓储服务，仓储服务大不了重启以下不就好了，因此不会出现因为仓储服务挂了导致支付服务跟着挂了的情况，所以级联失败问题也得到解决。同时因为没有强依赖关系，即不在支付服务里面调用其他服务，那么也就不用等待其他服务执行完，那么也就没有了资源浪费的问题，于是资源浪费问题也解决了。



##### 优势四：流量削峰：

异步调用除了以上讲的能够解决同步调用的问题以外，还有一些自己的优势。

比如，现在一秒内有一万个支付请求来到支付服务，但是订单服务、仓储服务、短信服务每秒只能处理一百个，此时Broker就能起到缓冲的作用，就好像洪水来了有个大坝拦住了，然后订单服务、仓储服务、短信服务能处理多少请求就处理多少请求，处理完了，再从Broker中取，这样订单服务、仓储服务、短信服务处理业务的速度是一直按照自己的能力来的，能处理几个就处理几个，压力都由Broker扛着，于是一个高度的并发就被砍平了：**短时间内的海量请求变成较长时间处理完，起到了保护微服务的作用**。这就是流量削峰，特别是在如秒杀等高并发场景下可以用这个技术去实现



##### 总结：

异步通信优点：

1：耦合度低，因为异步通讯不需要去调用对方，而是通过发布事件，几乎没有耦合；

2：吞吐量提升，因为不存在调用，所以也就不需要等待对方执行完成，就会耗时更短，因此吞吐量就更高；

3：故障隔离：因为没有调用，对方挂了跟自己没影响，自然就把故障隔离了；

4：流量削峰：这不是解决同步通讯的问题，这是异步通讯特有的功能，高并发的流量来了之后，通过broker做一个缓存，微服务可以基于自己能力从Broker中获取事件、处理事件，这样就可以起到对微服务的保护！

以上，会发现异步通信过程中，所有东西都依赖于Broker去实现，因此，一旦Broker挂了，那么整个微服务也完蛋了；而且流量削峰的过程中，高并发的流量用Broker去缓存，微服务“慢慢”从Broker中去取，因此，如果Broker的高并发能力不行，扛不住这么海量的高并发流量，就像大坝修的不好，洪水来了，可能冲毁大坝。

以上，所以对Broker的可用性、并发能力要求都很高！

异步通信缺点：

1：依赖于Broker的可靠性、安全性、吞吐能力；

2：架构复杂了，业务没有明显的流程线，出了问题不好追踪管理

#### 什么时候使用同步？什么时候使用异步？

事实上，大多数情况下都会去使用同步，因为大多数情况下对并发并没有很高的要求，相反对时效性要求很高，比如我调用你去查询一个信息，而这个信息又立马在接下来的业务中用到，必须要用同步调用；

而如果你不需要等调用的结果，同时对吞吐量的要求、并发的要求高，还希望接触服务间的耦合，那么就应该用异步方案，这样的场景相对少一点。

#### 异步通讯的实现：MQ

上面学习了异步通讯中的事件模型驱动架构，我们知道这个架构中最重要的就是Broker，因为Broker是用来做事件的管理！因此Broker的可用性、稳定性、并发能力即吞吐量就成为了事件驱动模型架构中最重要的问题！其中MQ（消息队列）就是这个模型架构优秀的、常见的解决方案！

##### 什么是MQ

MQ（Message Queue）消息队列，队列我们不陌生，不就是存东西东的吗！消息队列字面上看就是存放消息的队列，消息就是事件驱动模型架构的事件，消息队列MQ就是上面说的事件驱动架构中的Broker。消息就是事件。

##### MQ有很多实现，常见的MQ有：

###### RabbitMQ：

基于Erlang语言 可用性高 吞吐一般 消息延迟微秒级 消息可靠性高。RabbitMQ稳定性好，可靠性高，吞吐量也不差，所以适合用于对稳定性要求较高的场景：比如业务之间的通信！

###### ActiveMQ：

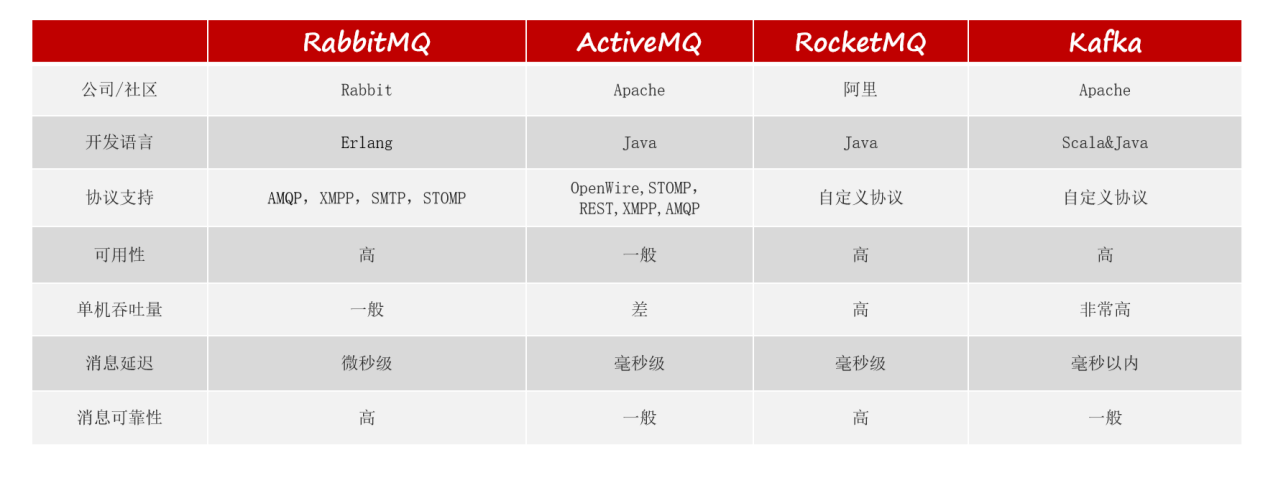
基于Java语言 可用性一般 吞吐量最低 消息延迟毫秒级 消息可靠性一般

###### RocketMQ：

基于Java语言 可用性一般 吞吐量高 消息延迟毫秒级 消息可靠性高。RocketMQ稳定性好，可靠性高，吞吐量也不差，所以适合用于对稳定性要求较高的场景：比如业务之间的通信！

###### Kafka：

基于Scala&Java 可用性高 吞吐量最高 消息延迟低毫秒级 消息可靠性一般，kafka吞吐能力见长，稳定性较差，可靠性较低，所以更适合那种海量数据的传输，但是对于数据安全要求不高的场景：比如日志数据的传输。



###### 技术选型

判断MQ的性能，要从消息吞吐量、消息延迟、可用性、消息可靠性这四个方面考量！没有完美的解决方案，技术选型要综合考虑！这里我们选用RabbitMQ，因为作为中小型企业来讲，你也没有对MQ深度定制的需求，更强调MQ稳定性和社区活越度，选用RabbitMQ是没有问题的！如果你是大型企业，需要对MQ做深度定制，那么你可以选用RocketMQ，基于Java语言做自定义开发！

一般衡量MQ的吞吐量是按照MQ每秒能发送的数据大小来分析的！

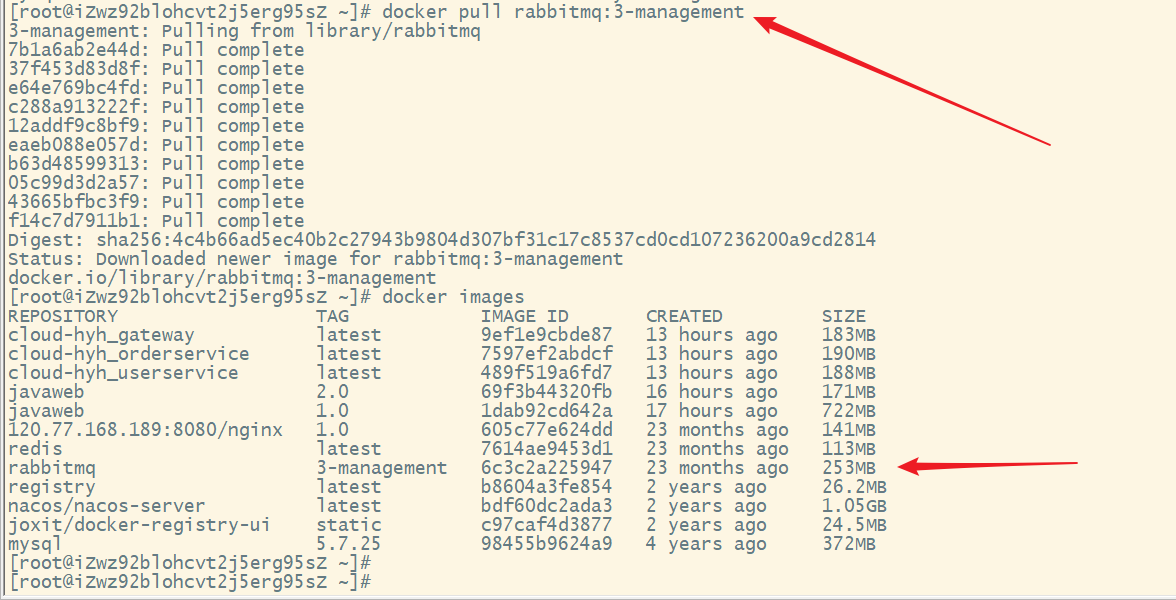
#### RabbitMQ安装：

RabbitMQ是基于Erlang语言开发的开源消息通信中间件，因为Erlang是面向并发的编程语言，天生就是为了分布式系统设计的，而即然RabbitMQ是基于Erlang开发，那么自然就具备了这些特征，同时，对消息可靠性、稳定性、高可用非常优异。RabbitMQ最擅厂的就是消息的可靠性、消息的稳定性、整个系统的高可用！

RabbitMQ官网：<https://www.rabbitmq.com/>

##### Docker安装RabbitM：

###### 1：拉取镜像：docker pull rabbitmq:3-management



###### 2：docker run安装MQ：

--hostname mq1 是配置主机名，为了以后集群部署，单机不配也没问题；开放了两个端口，15672是管理平台的端口；5672是做消息通信的端口，也就是说将来发消息、收消息都需要通过这个端口去建立连接

docker run \

-e RABBITMQ\_DEFAULT\_USER=hyh \

-e RABBITMQ\_DEFAULT\_PASS=123456 \

--name mq \

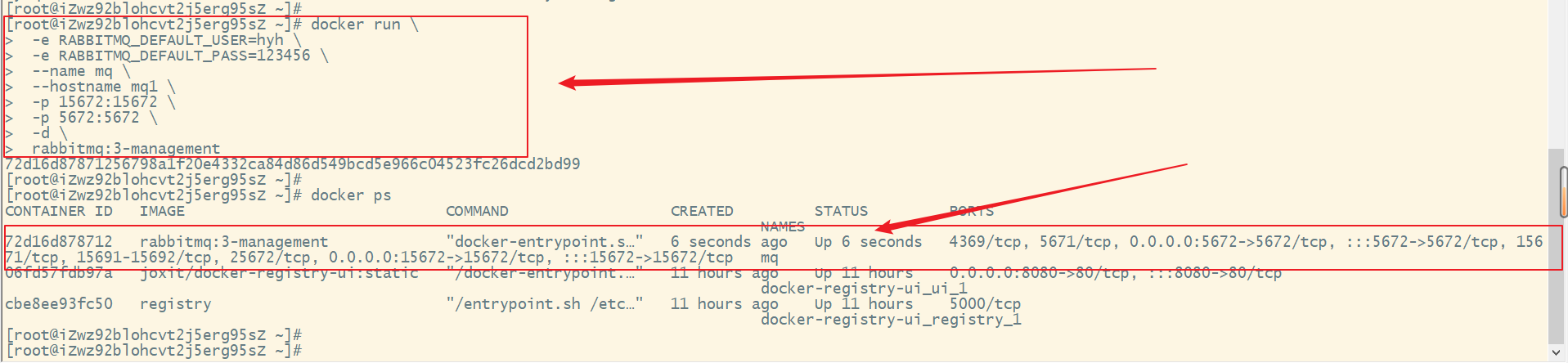
--hostname mq1 \

-p 15672:15672 \

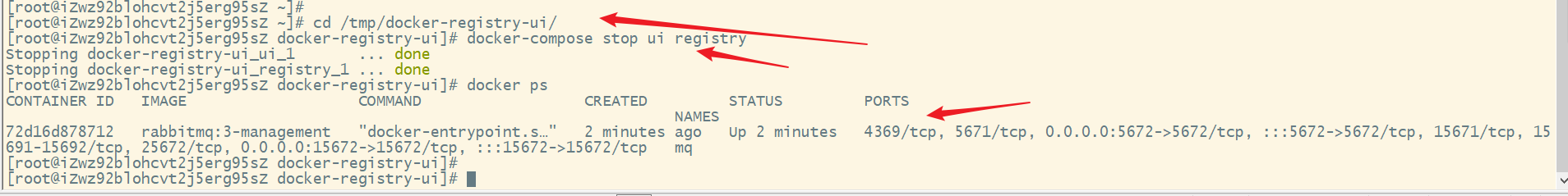
-p 5672:5672 \

-d \

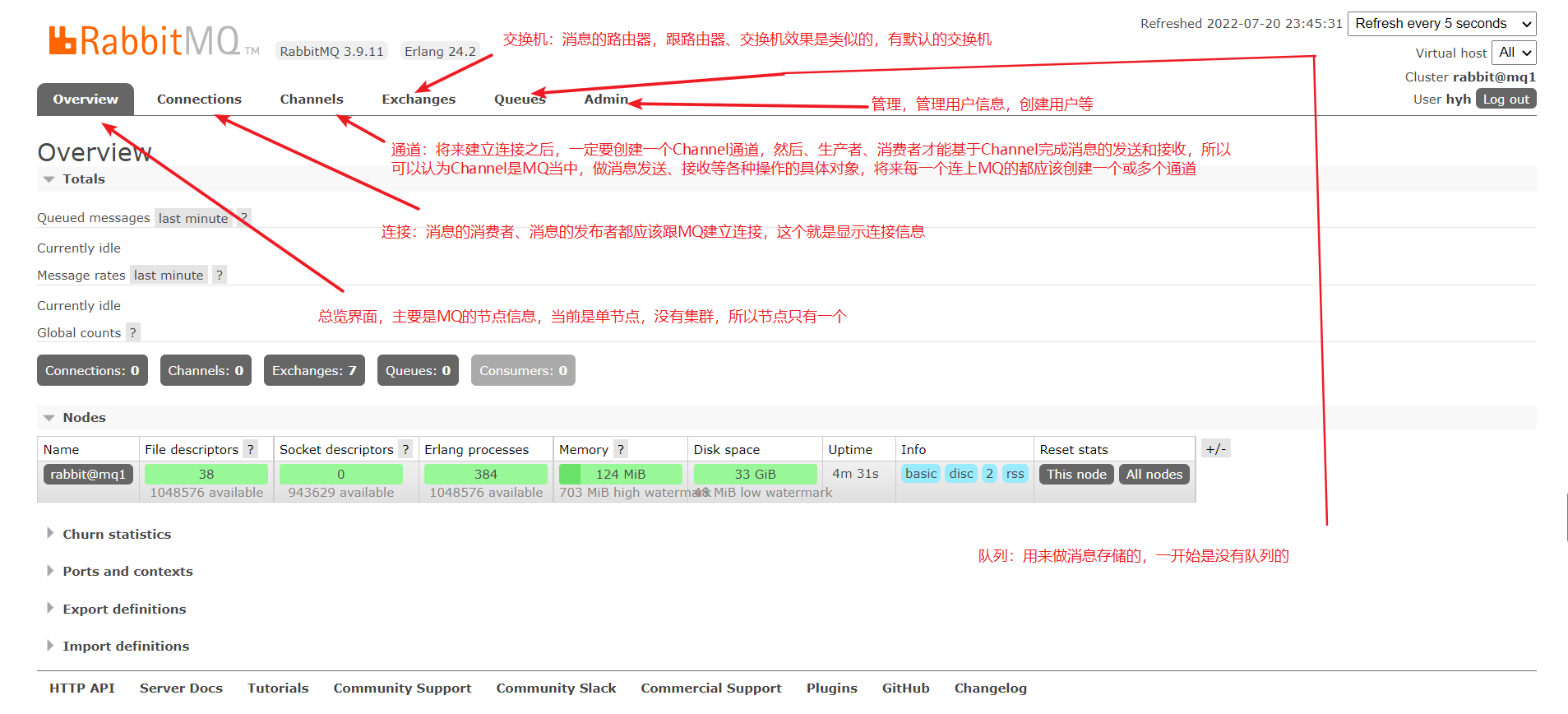
rabbitmq:3-management



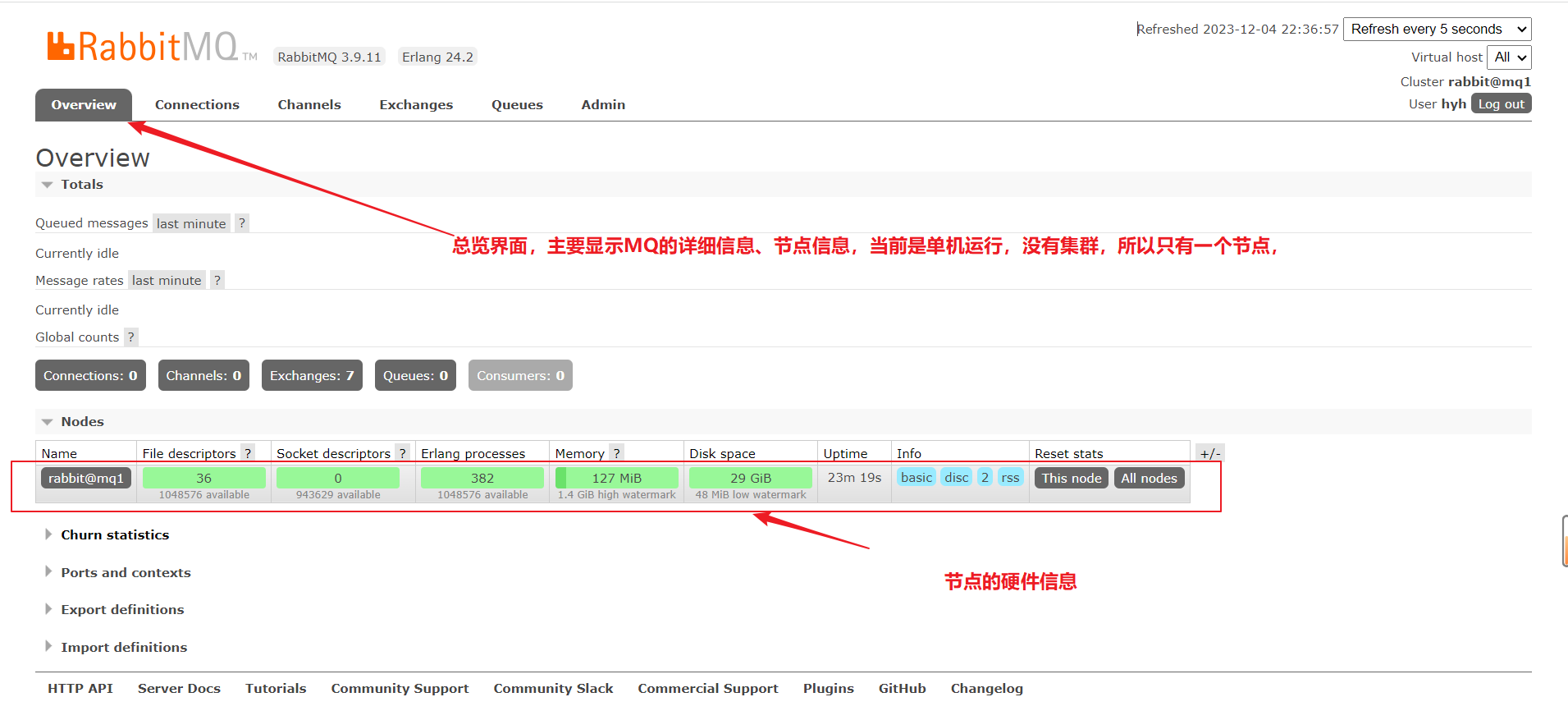
###### Tips：停止私有镜像仓库服务：



###### 3：访问管理平台：<http://120.77.168.189:15672/> 账号、密码是上面配的：



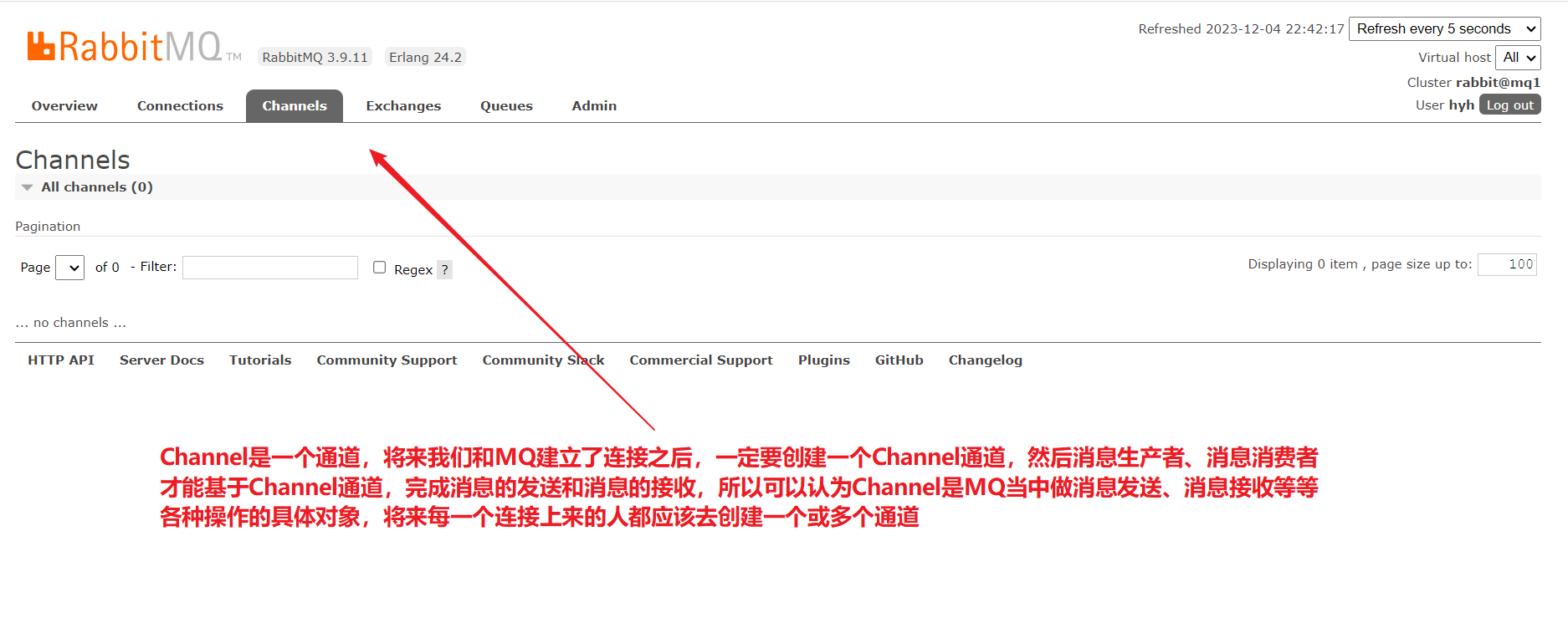
Overviews总览



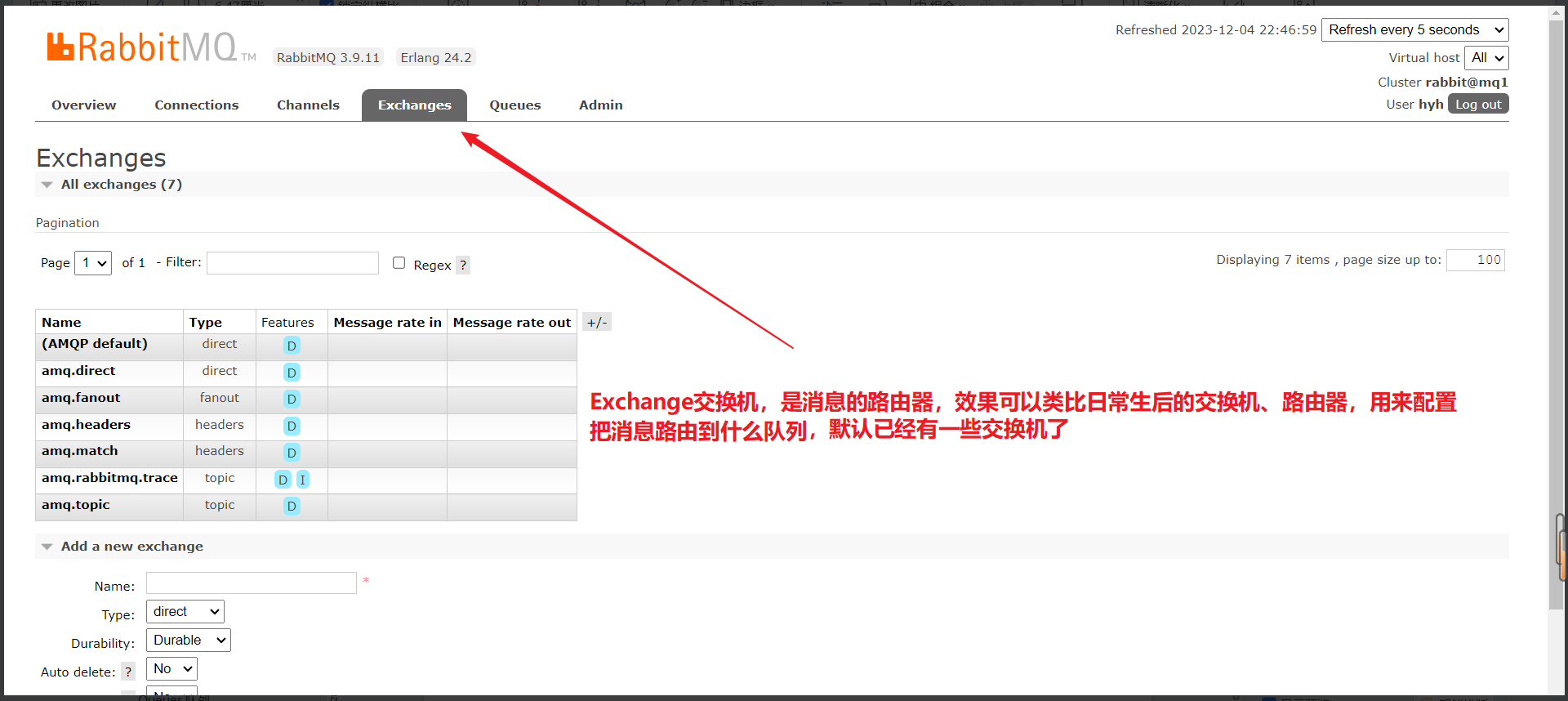
Connections连接



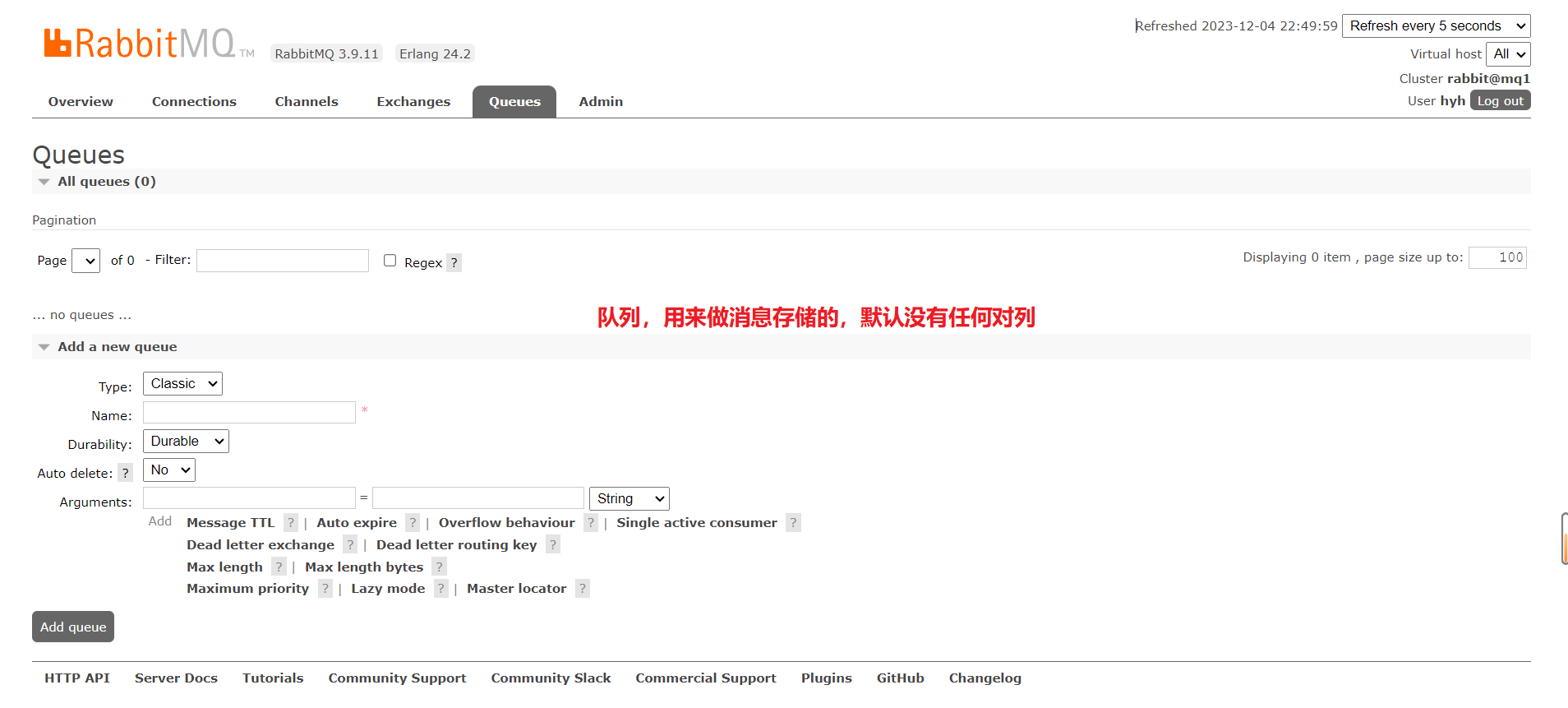
Channels通道



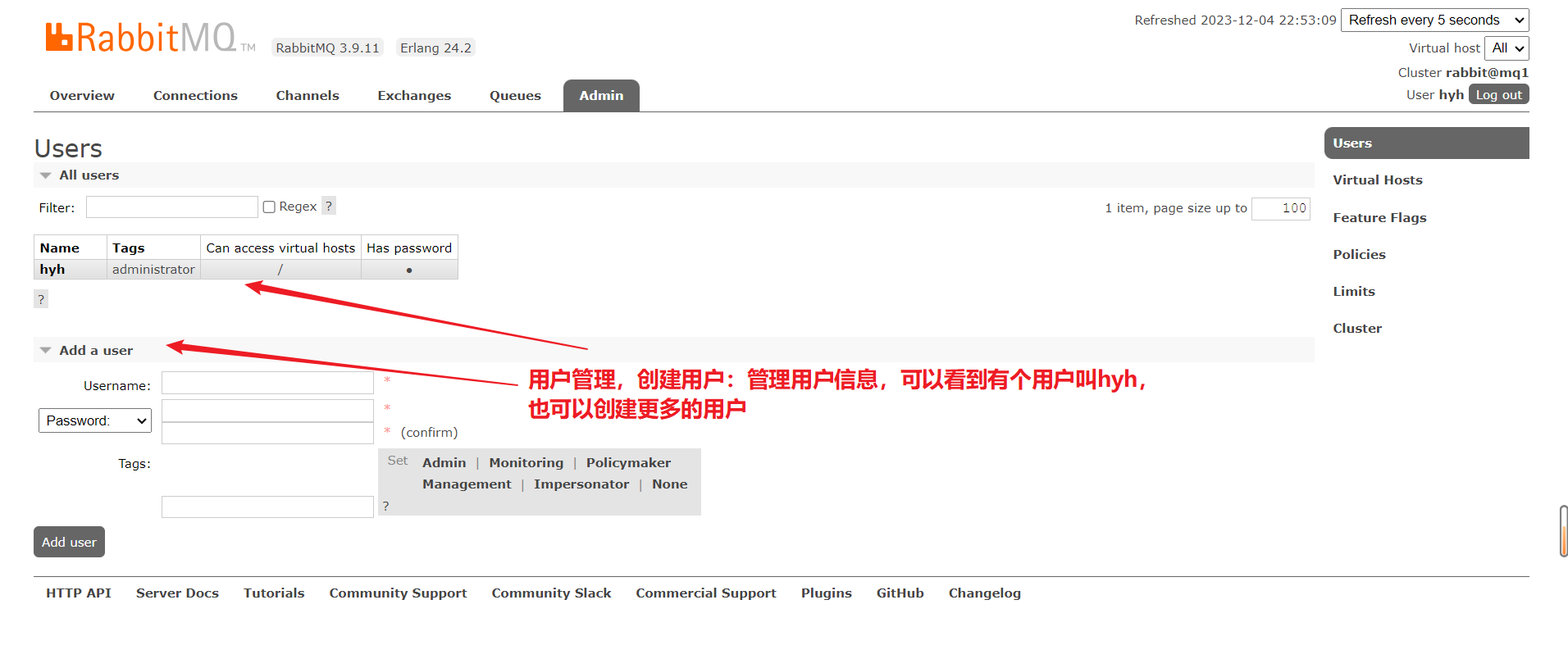
Excahnges交换机



Queues队列

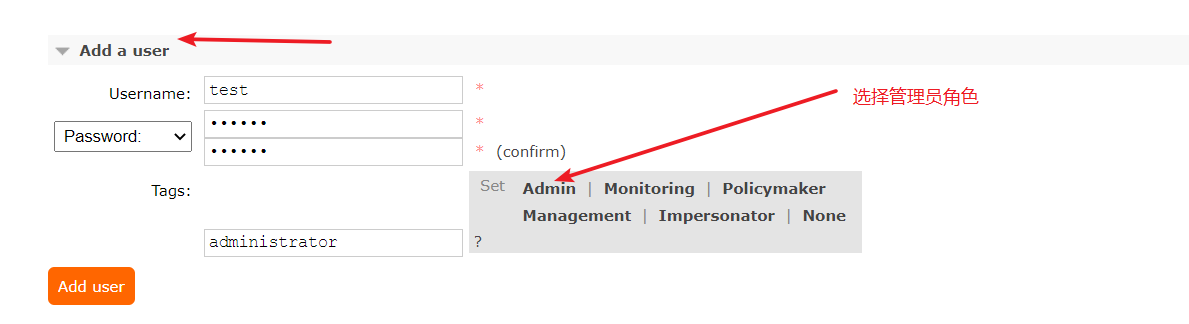


Admin用户管理



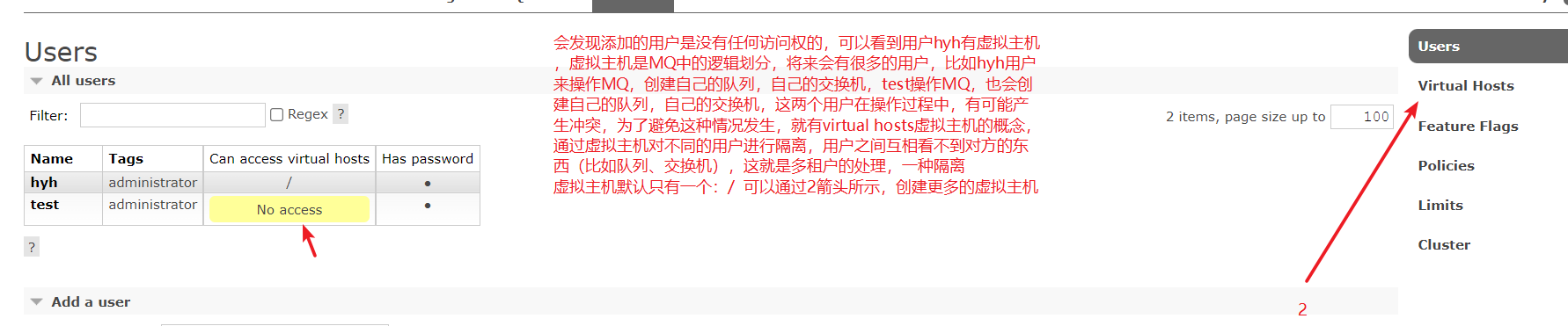
###### 4：添加用户：



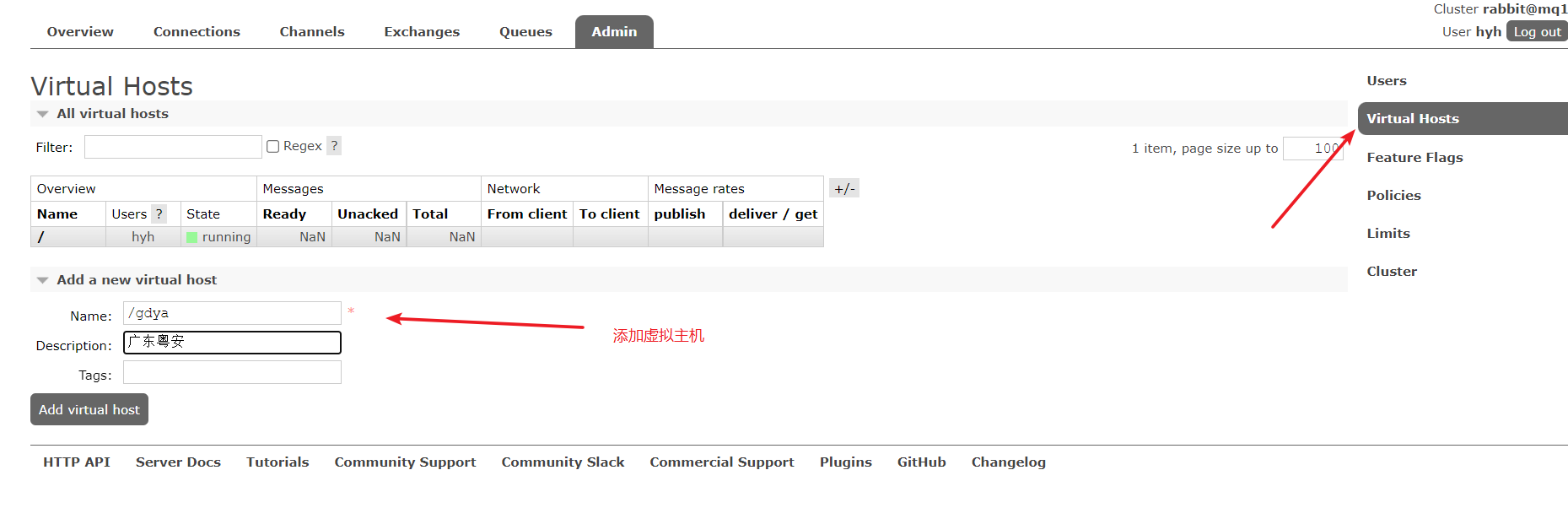


虚拟主机Virtual Host：

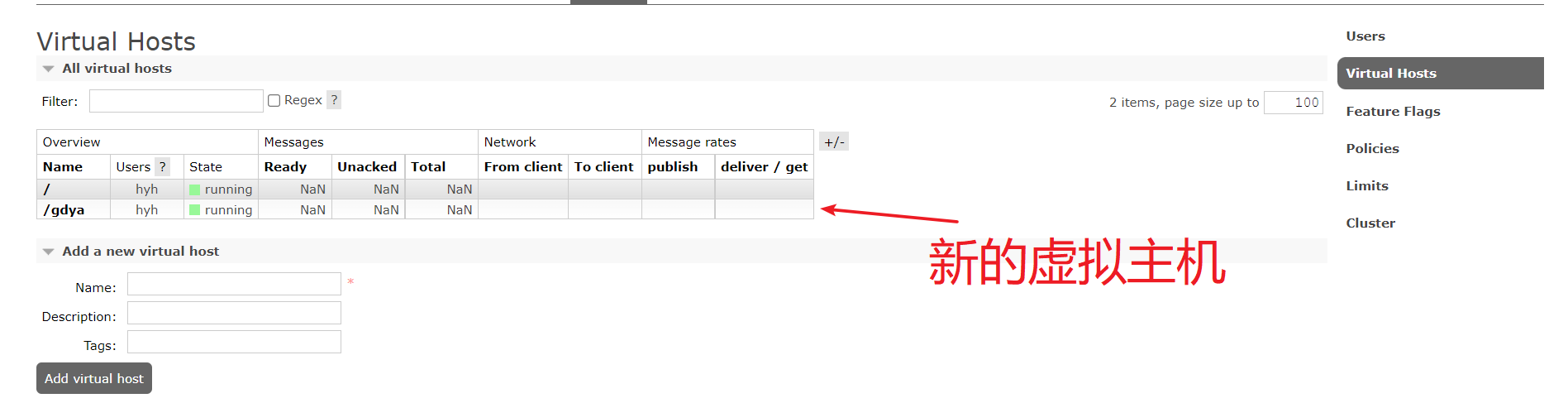
上面创建完用户后，可以看到新增用户是没有访问权的：no access，这是有个概念：Virtual Host虚拟主机，虚拟主机是MQ中的逻辑划分，将来可能会有很多用户，比如这个hyh用户来操作RabbitMQ，创建自己的队列、自己的交换机等等，test用户也来操作RabbitMQ，这两个用户在操作过程中有可能产生冲突，为了避免这种问题的发生，就有一个Virtual Host虚拟主机的概念，通过虚拟主机对不同的用户进行一个隔离，大家互相看不到对方的东西（比如队列、交换机），这就是称为多租户的处理！是一种隔离，这就是虚拟主机的概念，虚拟主机默认只有一个：/，可以通过Virtual Host菜单创建更多的而虚拟主机！



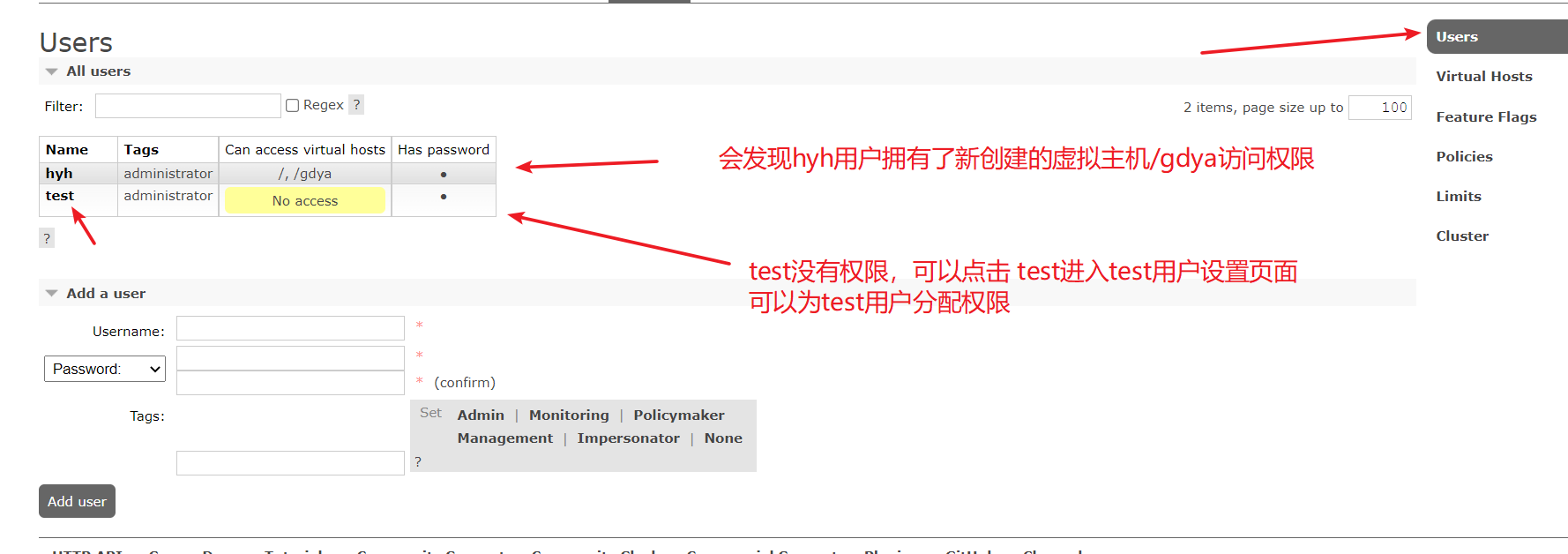
1：添加虚拟主机：



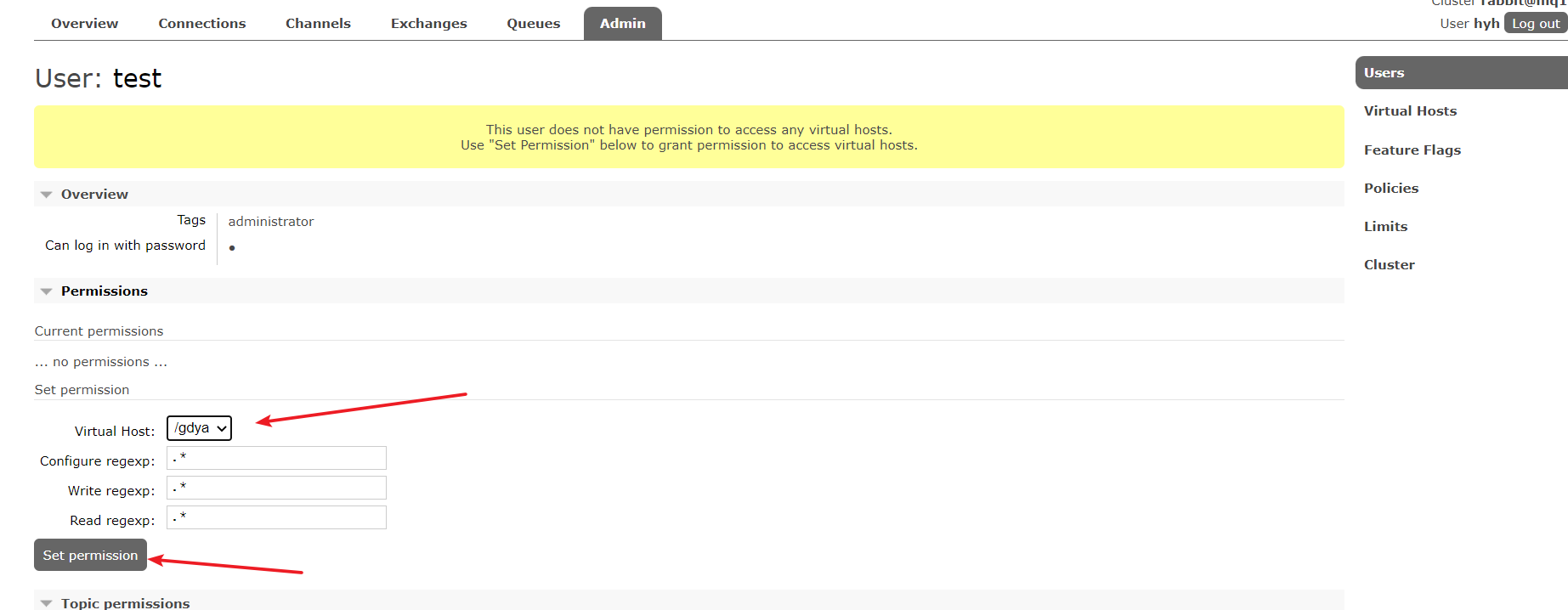
2：看到创建出来的新的虚拟主机：



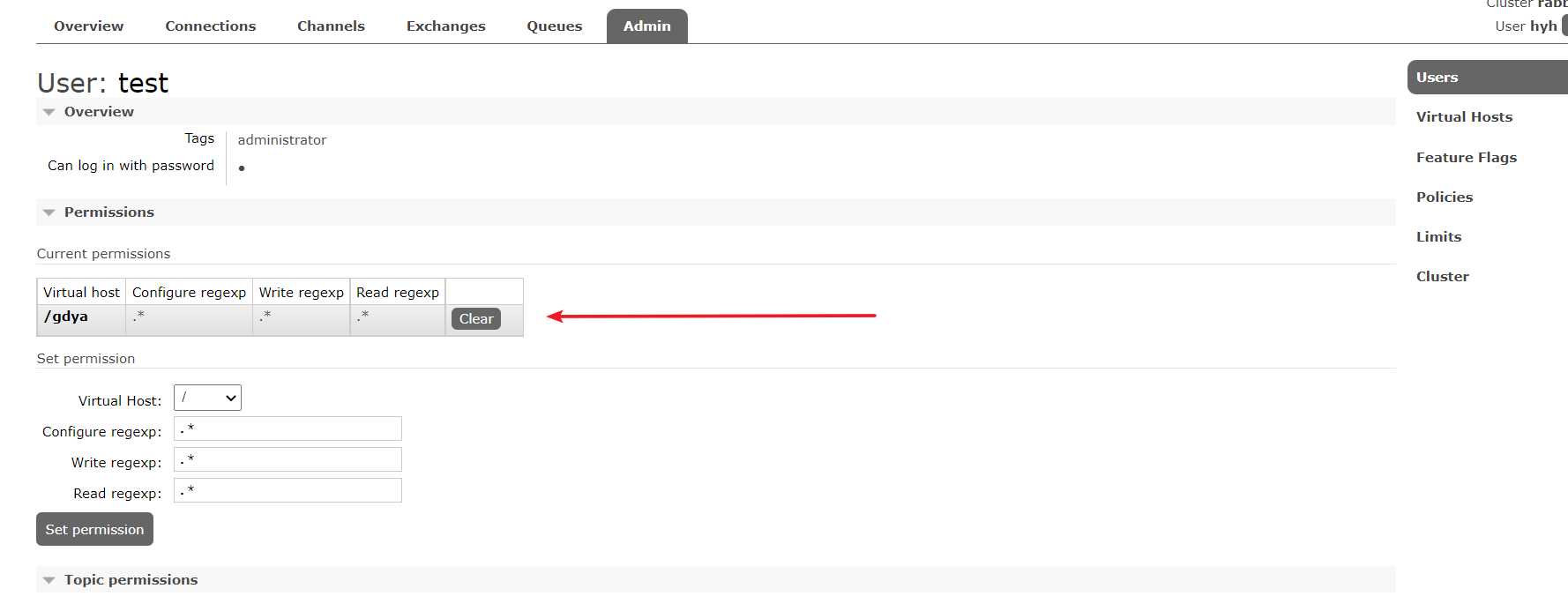
3：点击再次回到Users界面，可以看到hyh用户拥有了新创建的虚拟主机/gdya的访问权限：



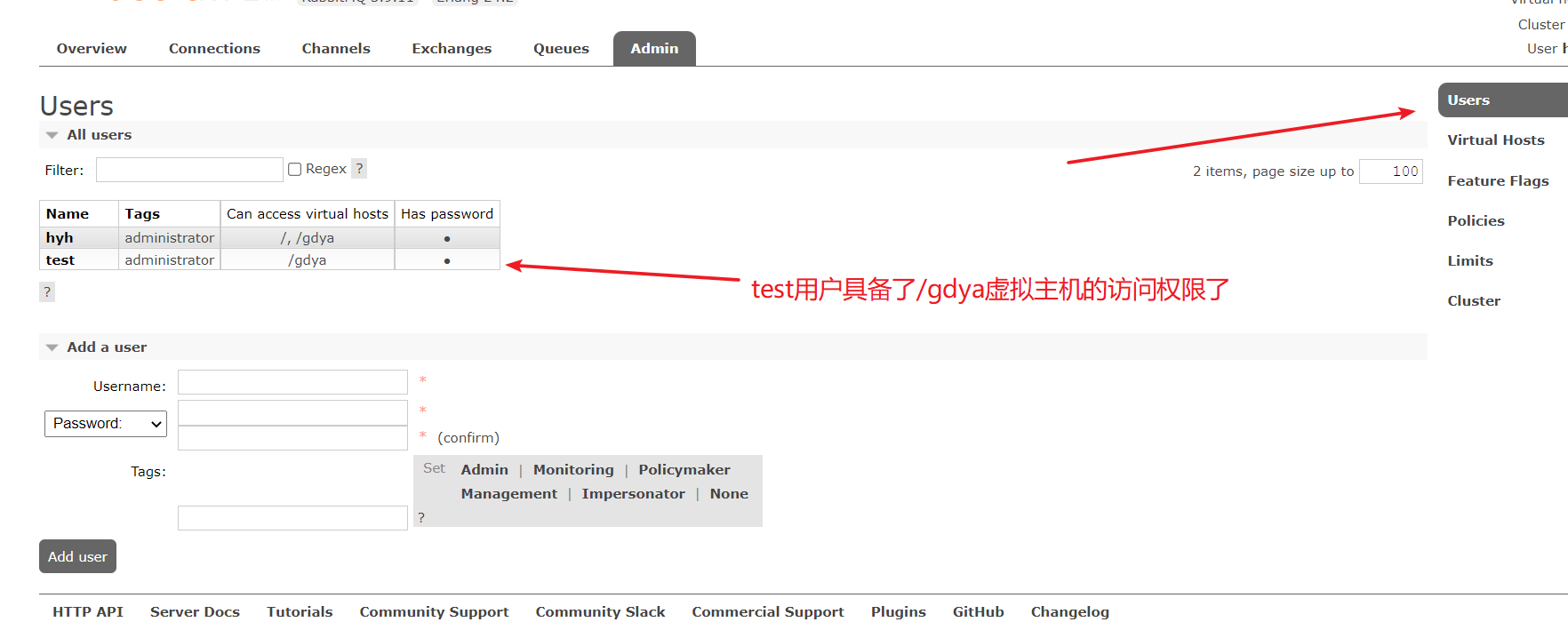
4：点击用户test，可以为test用户分配虚拟主机访问权限



5：发现test用户具备了/gdya虚拟主机的访问权限了：



6：再次点击返回User,可以看到test用户具备了/gdya虚拟主机的访问权限了：

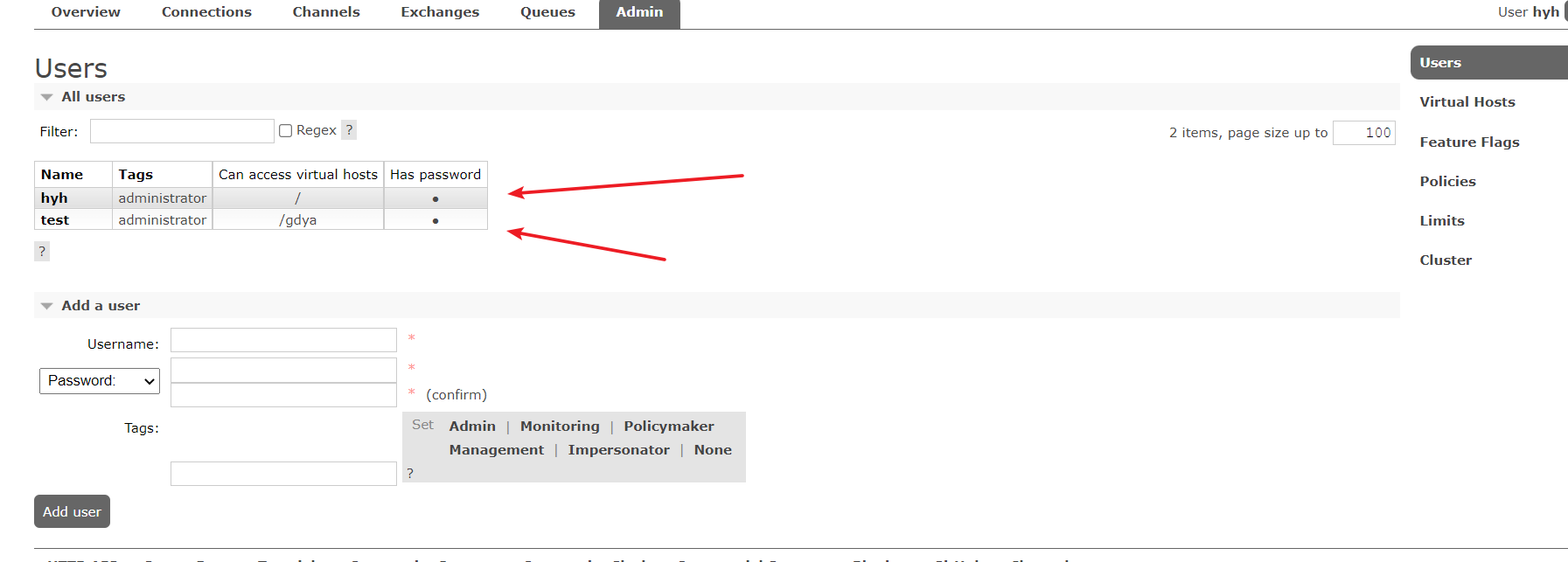


7：在exchanges查看交换机：

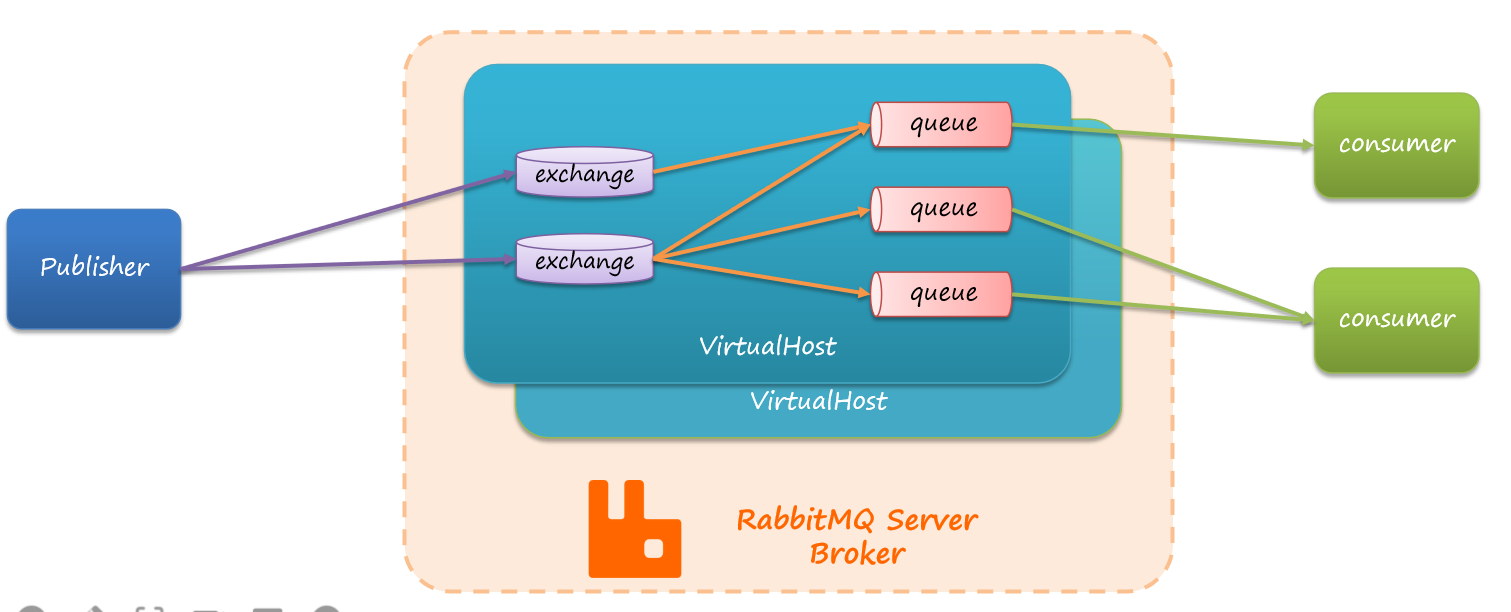
在交换机界面你会发现，虚拟主机/和虚拟主机/gdya这两个虚拟主机所具备的交换机信息是一样的，也就是说，虽然这里的交换机名称name是冲突的，但是因为虚拟主机不同，他们是被互相隔离开的！



8：一般情况下，每个用户应该有自己独享的虚拟主机，以隔离不同用户的业务，互相看不到对方的内容，回到User，把hyh用户的/gdya虚拟主机的访问权限清除：



##### RabbitMQ的结构和概念：



Publisher：消息发送着

Consumer：消息消费者

Publisher和Consumer和MQ建立连接Connection，互相和MQ有一个通道Channel;

Publisher会把消息发送到exchange（交换机），交换机负责路由，再把消息投递到对应Queue（队列），队列负责暂存消息，而后消费者从队列中获取消息，然后处理消息。

会看到图中有VirtualHost（虚拟主机），将来创建一个用户，用户应该有自己的虚拟主机，各个虚拟主机之间是相互隔离的、看不到的，这样可以避免干扰。这里可以理解成一台电脑装了多个系统！

##### 总结：

RabbitMQ中的几个概念：

Channel：操作MQ的工具，可以理解成通道

Exchange：路由消息到队列中

Queue：缓存消息

Virtual host：虚拟主机，是对queue、exchange等资源的逻辑分组，进行不同用户的隔离

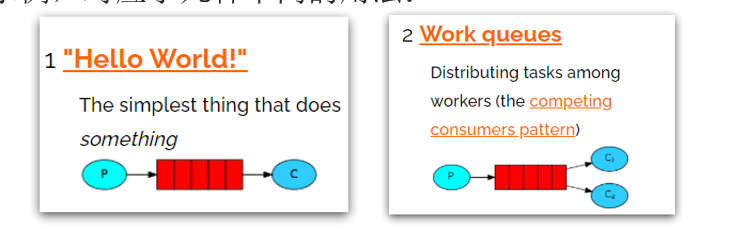
#### RabbitMQ中五种常见消息模型

以上安装了RabbitMQ，并且了解了RabbitMQ的整体架构，接下来看看MQ中常见的消息模型，并且动手实践一下。

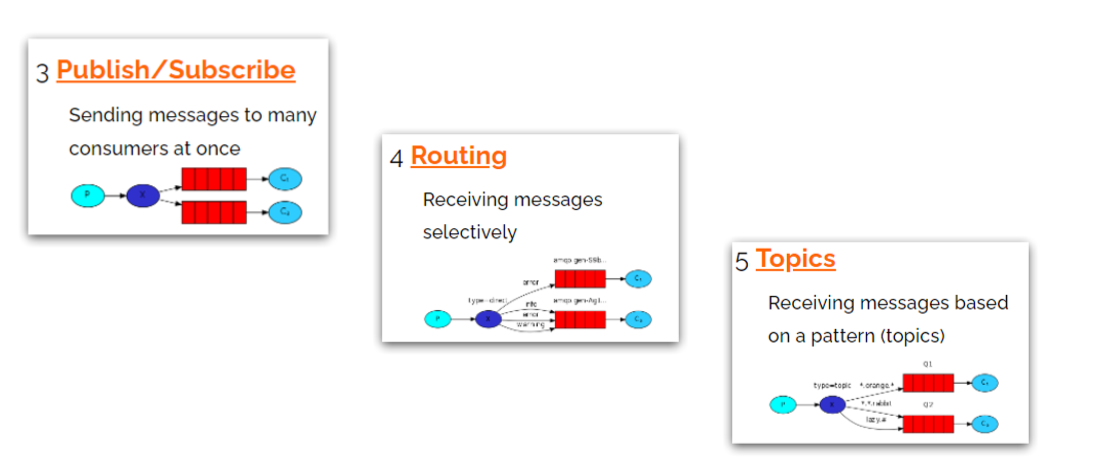
MQ的官方文档<https://rabbitmq.com/getstarted.html>中给出了5个MQ的Demo实例，这1-5是跟消息发送和接收有关的，对应了几种不同的用法。



其中前两个是基本消息队列（BasicQueue）和工作消息队列（WorkQueue），这两种有共同的特征，就是消息的发送和接收都是直接基于队列来完成的，见下图（p是publisher，c是consumer，红色部分就是队列），所以是基于队列来完成消息发送和接收，并没有出现前面RabbitMQ架构学习中出现的交换机exchange，不是一个完整的消息驱动的模型



后三种都是发布订阅（Publish、Subscribe），只是根据用到的交换机类型不同划分成具体三种：广播（Fanout Exchange），路由（Direct Exchange）、主题（Topic Exchange），见下图，紫色的东东就是交换机Exchange



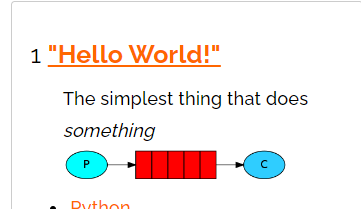
##### 1：基本消息队列（BasicQueue）：

这个队列和下面的工作消息队列一样，消息的发送和接收都是直接基于队列来完成的，而并没有出现我们上面在架构中出现的交换机，不是一个完整的消息驱动模型。如图所示：一个生产者，一个消费者，一对一

Publisher：消息发布者，将消息发送到队列Queue

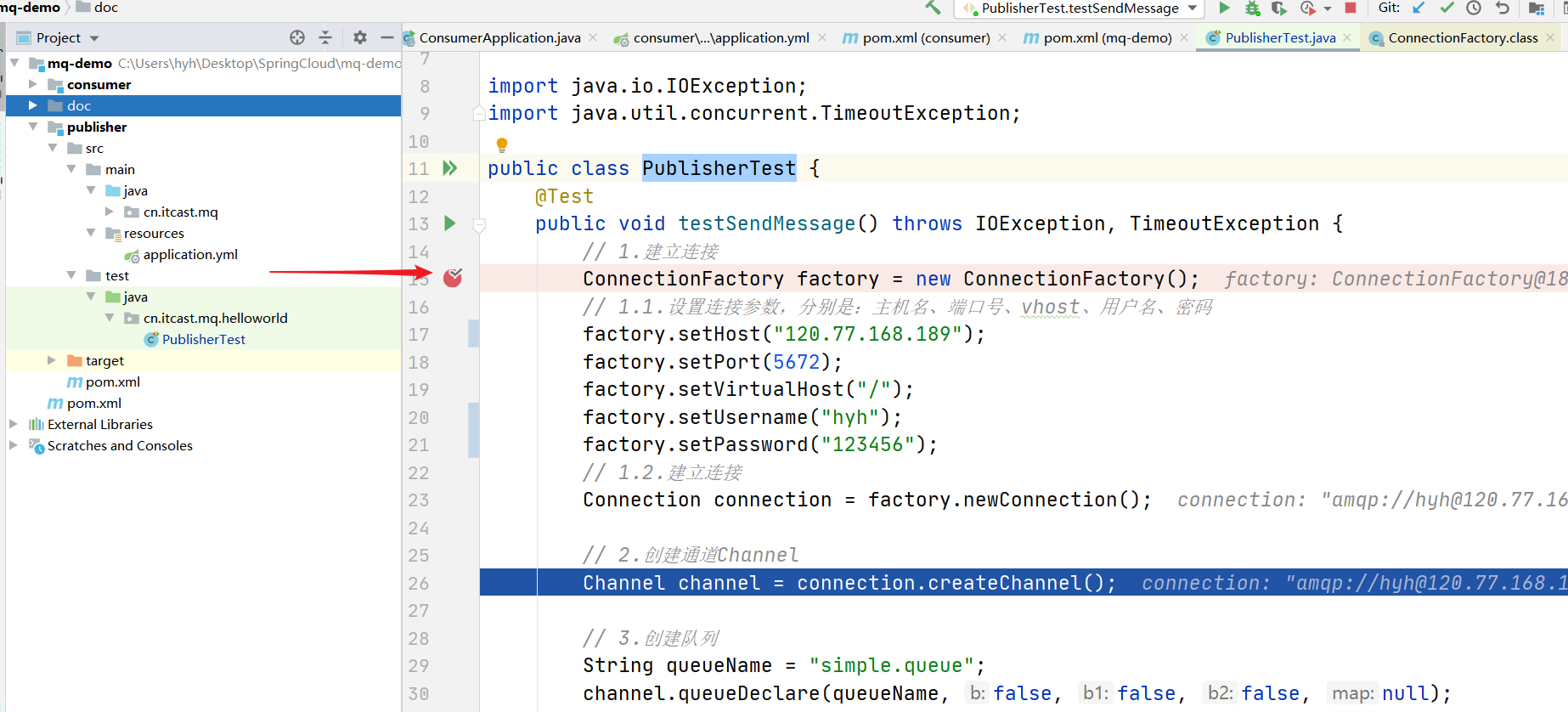
Queue：消息队列，负责接收并缓存消息

Consumer：订阅队列，获取队列中的消息、处理队列中的消息



基本消息队列只包括三个角色：Queue是RabbitMQ去管理的，Publisher和Consumer则需要我们编写代码实现！

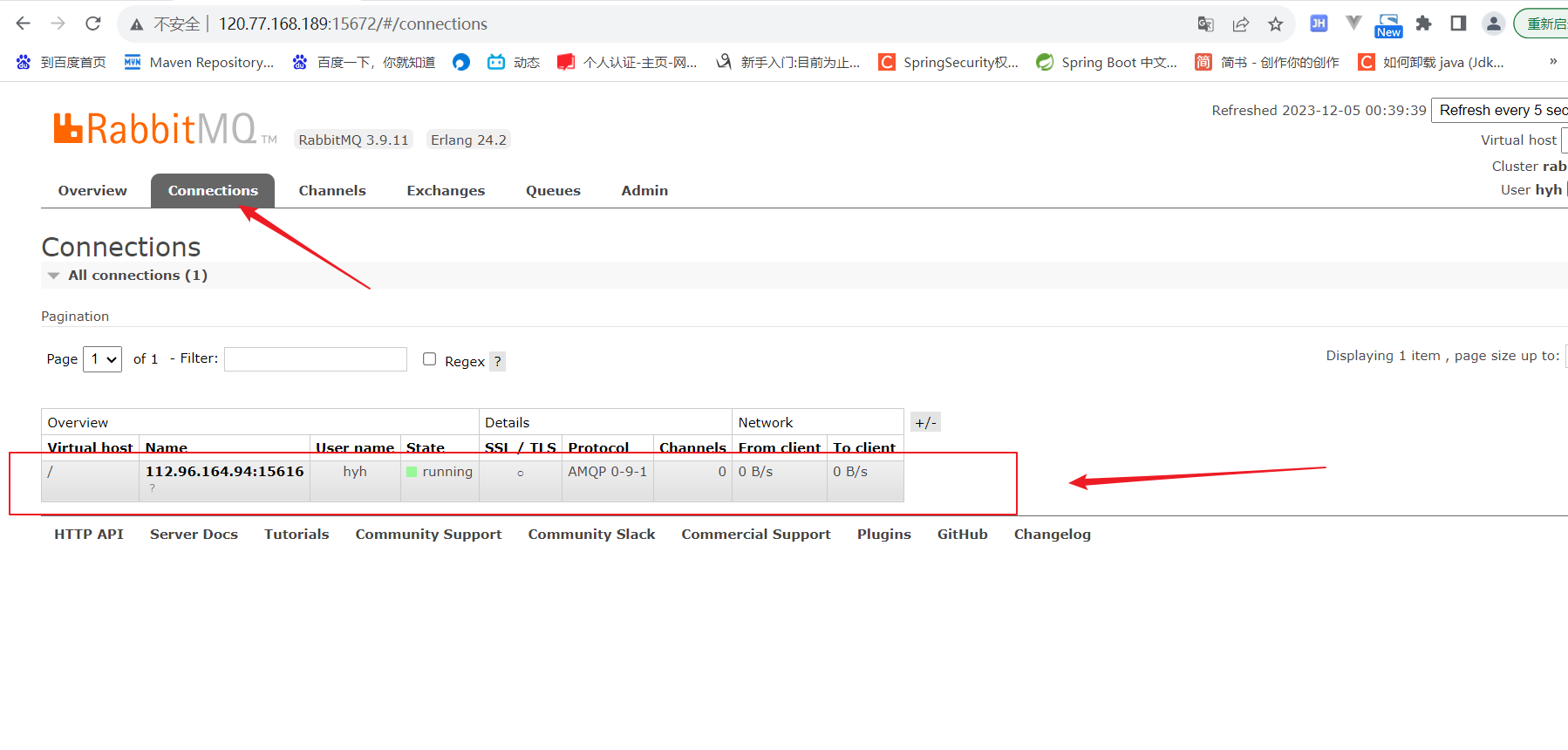
###### 1：导入资料中的mq-demo项目到IDEA，对测试类PublisherTest打断点：



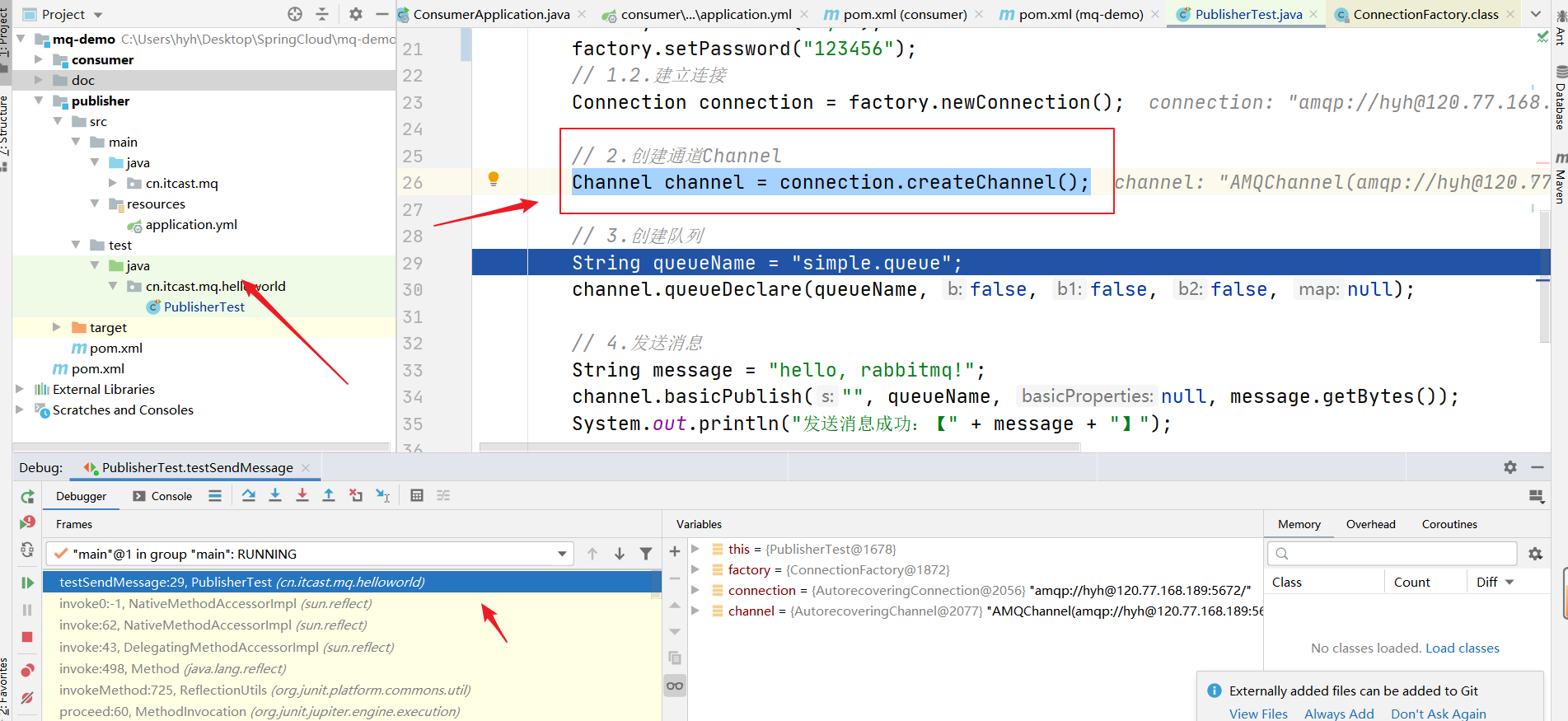
###### 2：打断点代码往下执行，让代码执行完建立与RabbitMQ连接这行代码：Connection connection = factory.newConnection();



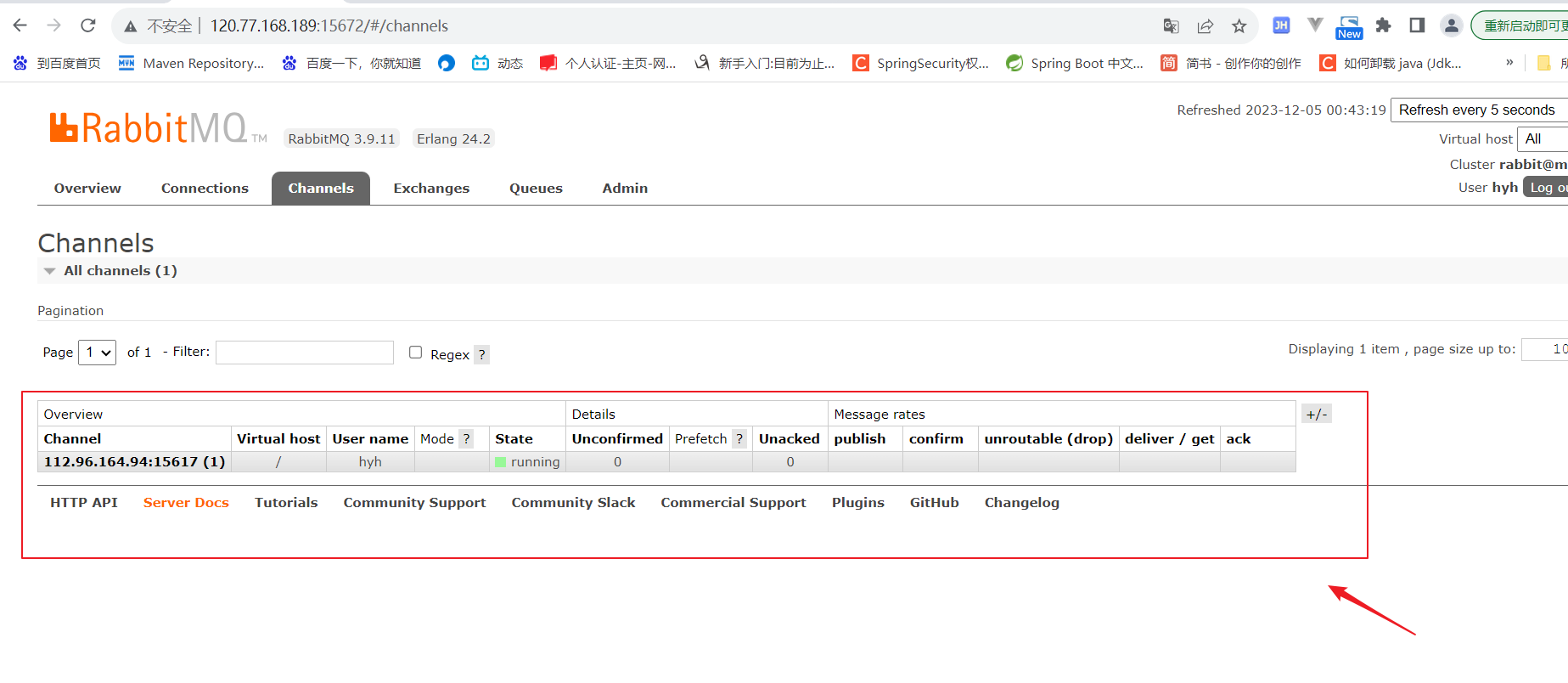
###### 3：在RabbitMQ管理后台查看Connections菜单，看到有连接建立，说明有人连接上来了：



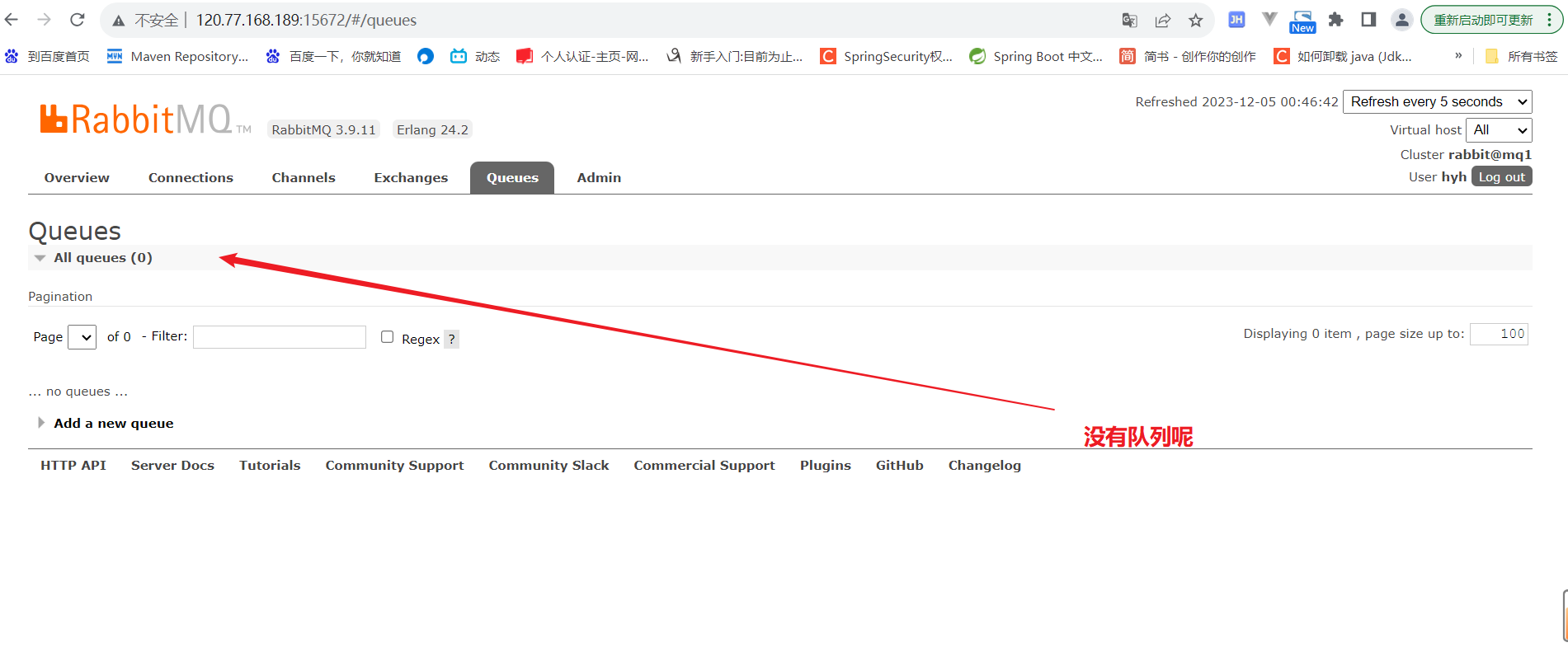
###### 4：继续打断点，代码往下执行，让代码执行完创建通道Channel的这行代码：Channel channel = connection.createChannel();



###### 5：在RabbitMQ管理后台查看Channels菜单，看到有通道创建：

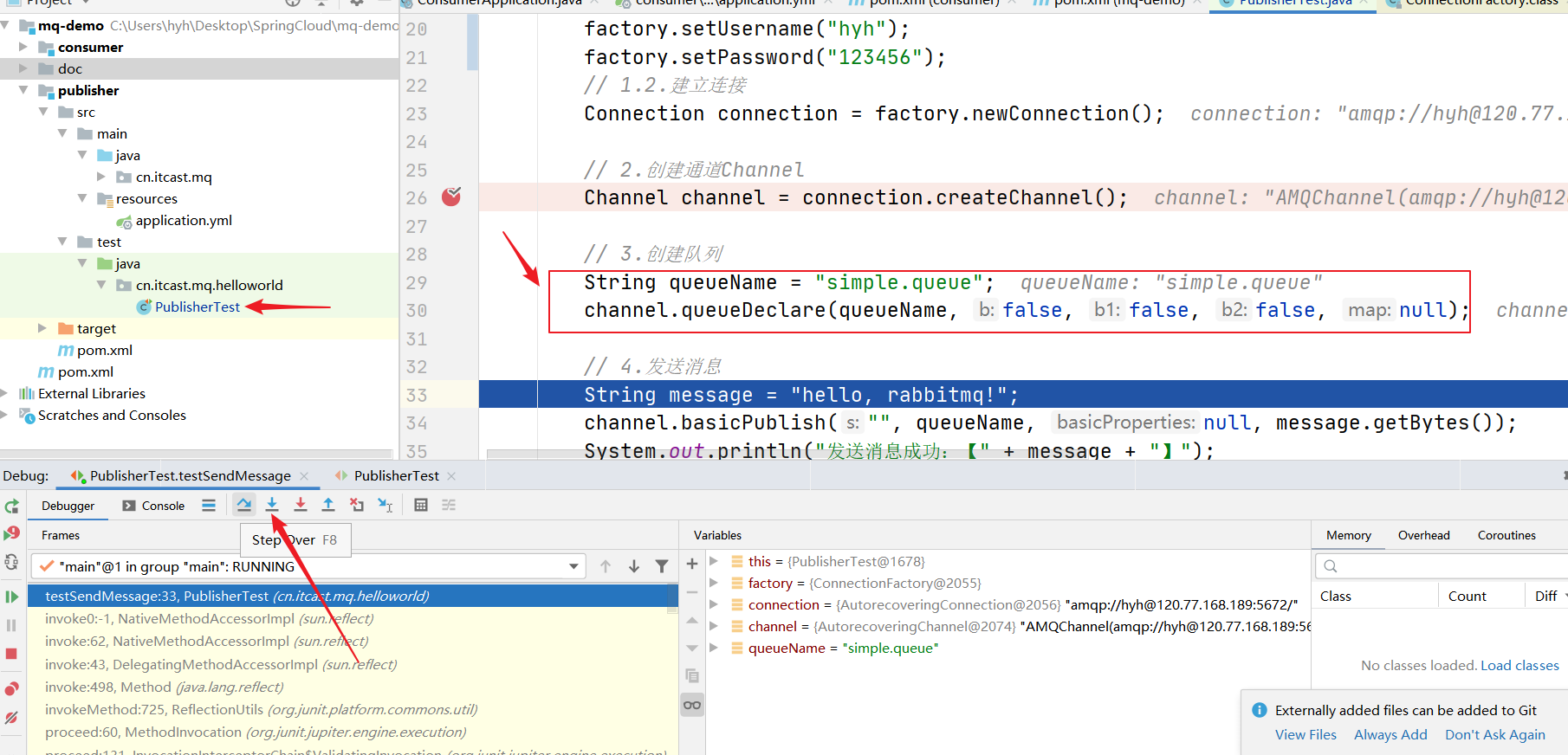


###### 6：通道有了，就可以基于通道往队列当中发送消息了，但是在RabbitMQ管理后台的Queues，发现没有队列呢！

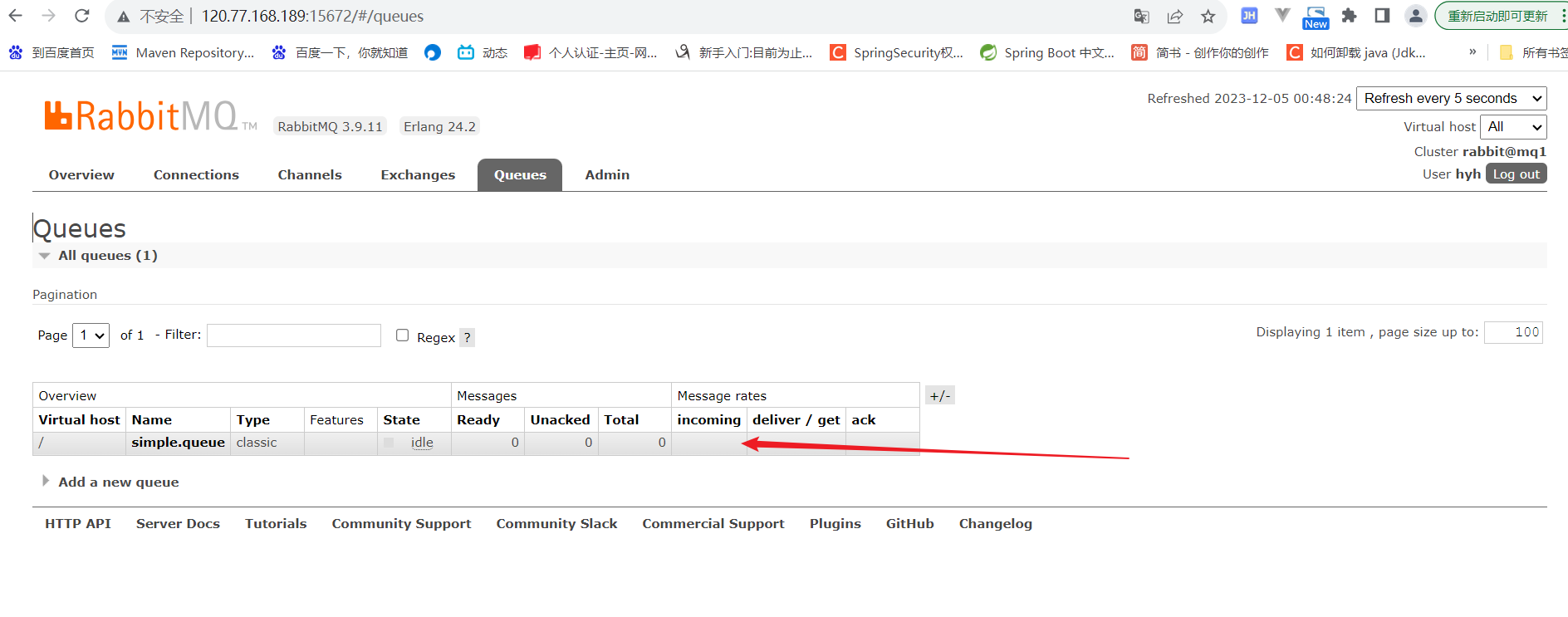


###### 7：继续打断点，代码往下执行，让代码执行完创建队列Queue的代码（注意；RabbitMQ没有队列才创建）：

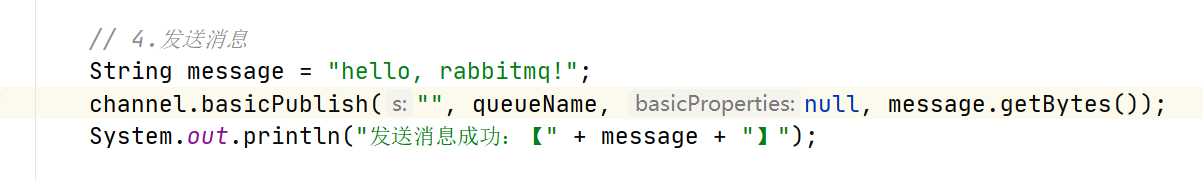
channel.queueDeclare(“simple.queue”, false, false, false, null);



###### 8：在RabbitMQ管理后台查看Channels菜单，看到有通道创建

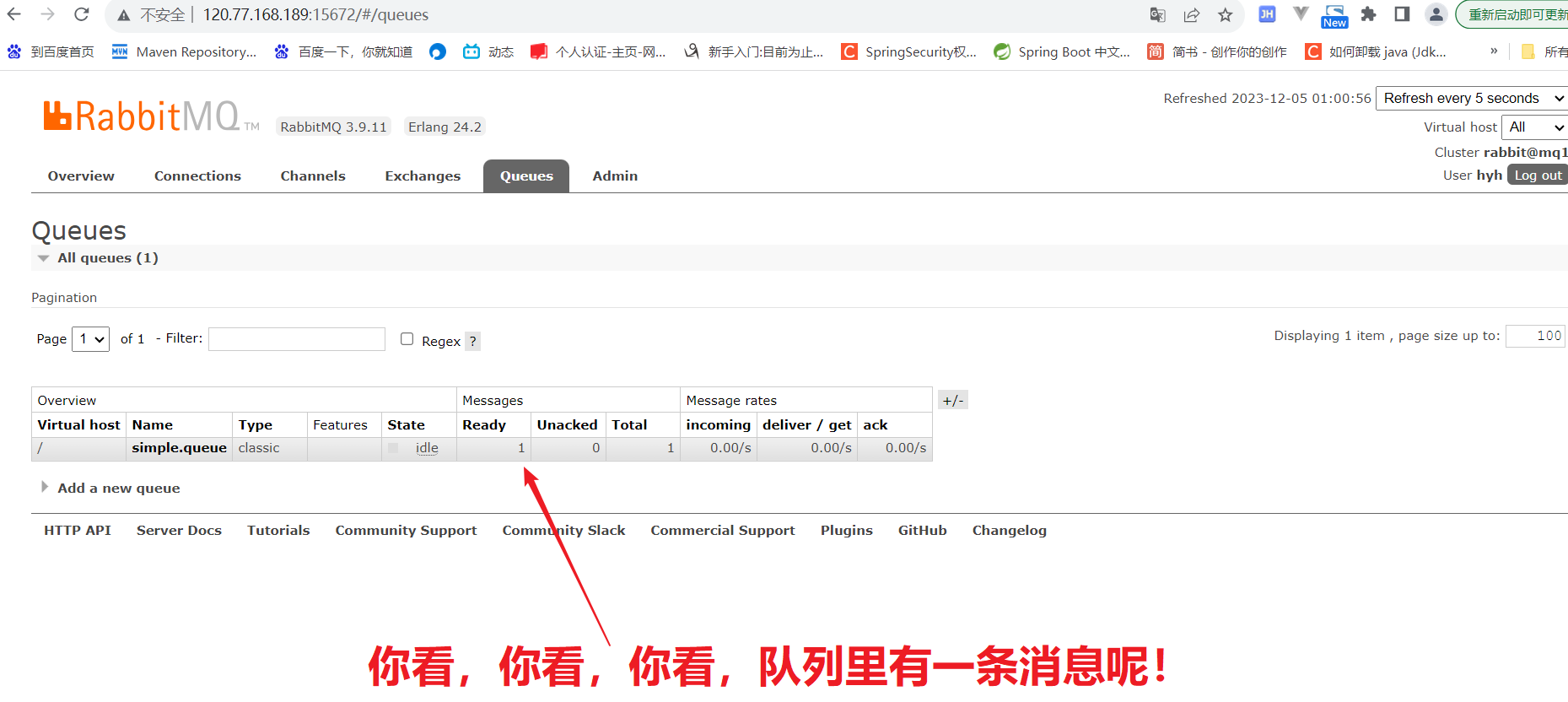


###### 9：有了队列，下一步生产者就可以向队列中发送消息了，继续打断点，代码往下执行，让代码执行完发送消息的代码：

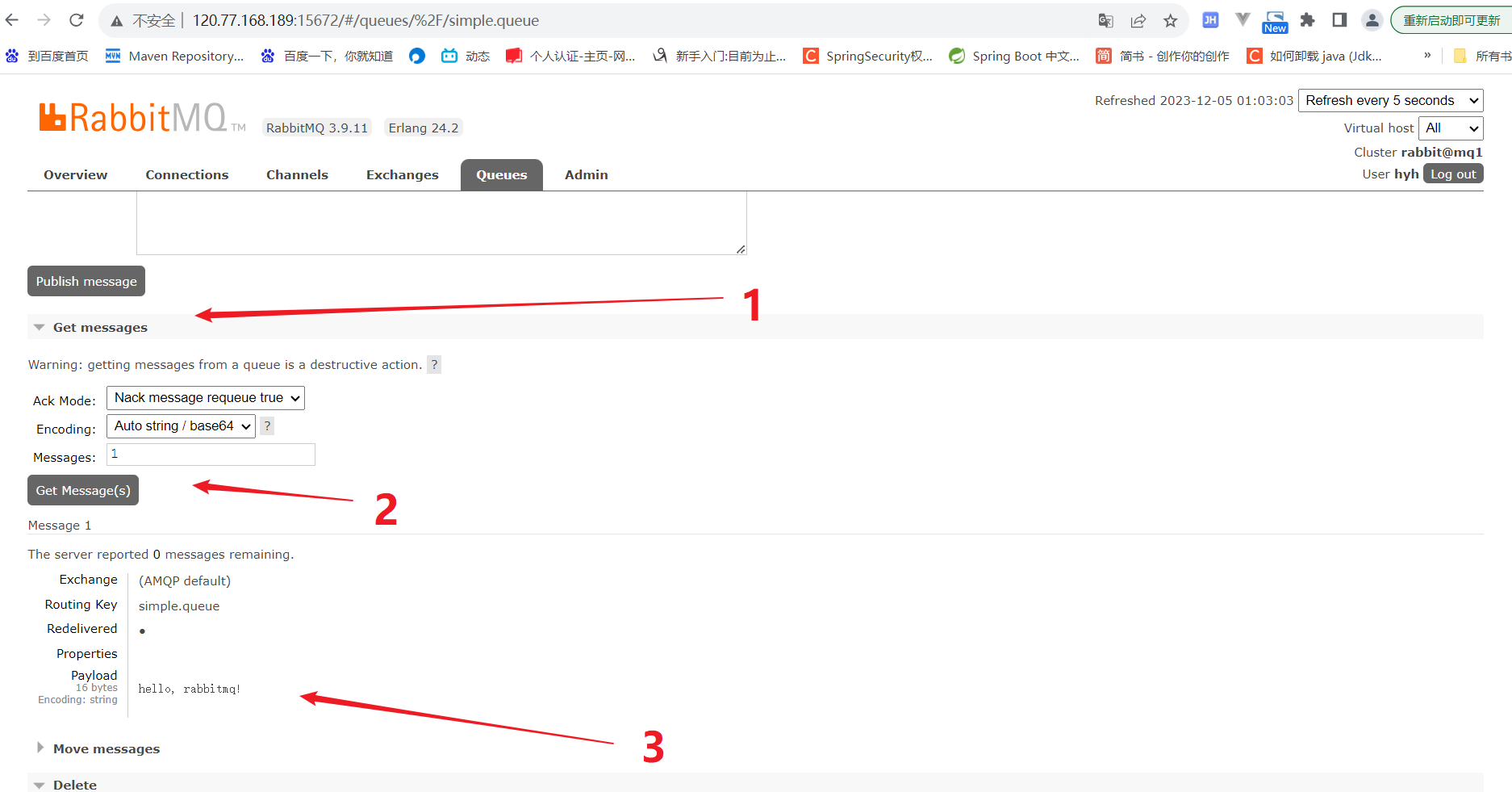


一样的操作就不截图了！

###### 10：在RabbitMQ管理后台查看发送到队列中的消息，说明消息成功发送到队列中去了

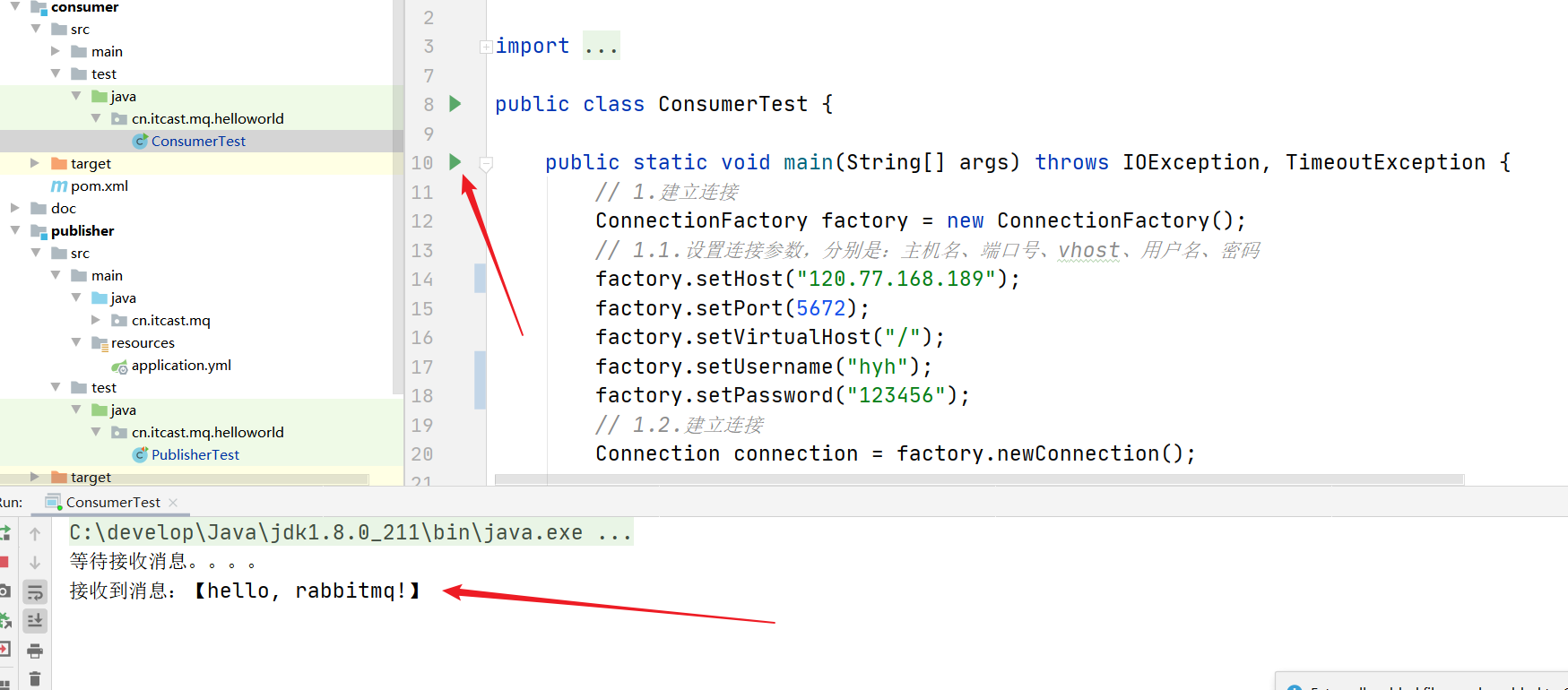


点击队列名，可以查看消息详细信息！



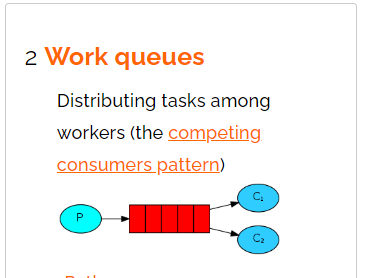
###### 11：执行剩下代码，注意了，发送者已经结束了，连接都断开了，也就是说发送者的消息发完了，发送者的事就没了，我管你谁收到了没有！我不管，你看，这不就是解除了耦合了吗！这不就是异步吗！

###### 12：消费者接收消息，运行消费者代码，看到接收消费了消息



##### 2：工作消息队列（WorkQueue）：不止一个消费者

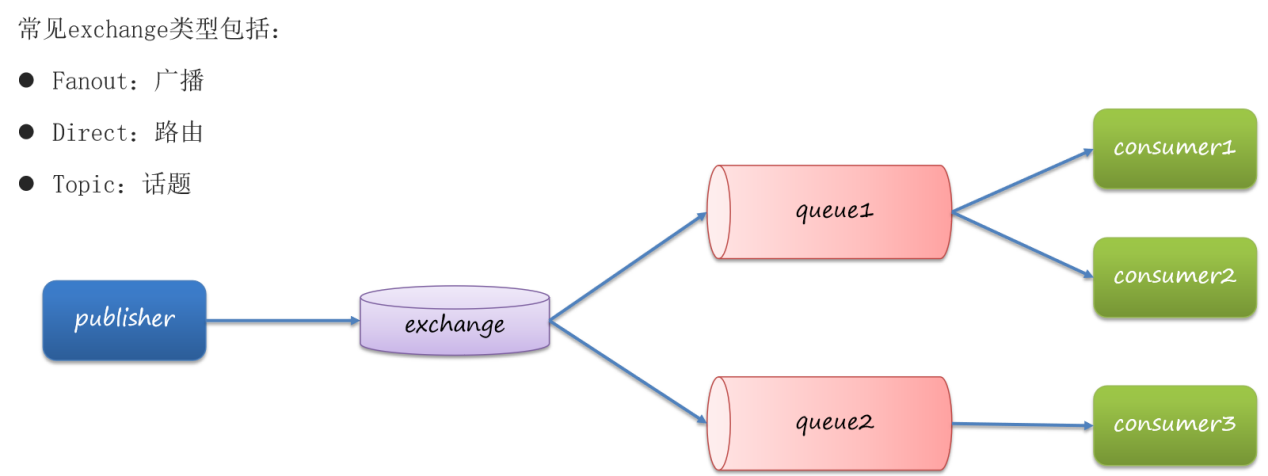
这个队列和上面的基本消息队列一样，消息的发送和接收都是直接基于队列来完成的，而并没有出现我们上面在架构中出现的交换机，不是一个完整的消息驱动模型。如图所示。



##### 基本消息队列和工作消息对列存在的问题：

上面的基本消息队列和工作消息对列，有一个共同的特点，那就是生产者发出的消息只能被一个消费者消费，因为消息一旦消费完，就会从队列中删除，这样的特点，往往无法满足实际生产中的要求，比如当用户支付完成后，得去通知订单服务，仓储服务，短信服务等，让这些服务去完成各自的业务，也就是说，发送的用户支付成功的消息，要被这些服务都收到，以上的基本消息队列和工作消息对列是不符合的，那么就需要用到发布订阅模式：

**发布订阅模式与之前基本消息队列和工作消息对列的区别就是允许将同一个消息发送给多个消费者。实现的方式是加入了交换机exchange!**



可以看到，会有消息发布者publisher 队列queue 消息消费者consumer，消费者跟队列之间依然会去做绑定，可以两个消费者绑定一个队列，也可以一个消费者绑定一个队列，这一点和之前学的没什么变化，所以在发布订阅模型中，我们不用关心消费者这块怎么去绑定，以前怎么做现在就怎么做，我们只要关心消息从发布者这里到达队列的过程，消息发送时，以前是直接到达队列，现在不行了，现在要先发给交换机exchange，再由交换机把消息转发到队列当中，因此，消息发布者不需要知道队列的存在，将来这个消息到底投递给哪个队列，还是投递多个队列，都是由交换机决定的，那么，如果消息实现转发给了多个队列，就实现了一个消息被多个消费者消费了！

消息通过交换机发送给一个队列还是多个队列，是由交换机的类型决定的！交换机（exchange）常见类型：

Fanout：广播

Direct：路由

Topic：主题

注意：exchange交换机虽然有不同类型，当时有一个巩同特点，就是只负责消息路由（即转发），而不是存储，只有队列才是存储消息的，路由失败则消息丢失！

发布订阅（Publish、Subscribe），剩下三种都是发布订阅，只不过这三种用到的交换机类型不同，所以根据交换机类型不同分为三种：

##### 3：Fanout Exchange：广播交换机

特点：会将接收到的消息路由到每一个与其绑定的queue，即只要队列和Fanout Exchange绑定了，那么消息就会被Fanout Exchange路由到与其绑定的队列！



##### 4：Direct Exchange：路由交换机

Direct Exchange会将接收到的消息根据规则路由到指定的queue，注意：不是所有的queue，是指定的queue，因此称为路由模式（routes）。

1：每一个queue都与Exchange设置一个BindingKey；

2：发布者publisher发送消息时，指定消息的RoutingKey；

3：Exchange将消息路由到BindingKey与消息RoutingKey一致的队列；

4：一个Queue队列与交换机Exchange绑定的时候可以指定多个BindingKey，如果不同queue的BindingKey一样，那么此时的效果和Fanout Exchange一样；

**5：利用@RabbitListener声明Exchange、Queue、RoutingKey，不用额外在配置类创建Bean声明，运行项目，会发现RabbitMQ管理平台会自动创建好对应的交换机、队列、以及交换机和队列的绑定关系，同时也证明了，交换机、队列、交换机和队列的绑定关系为啥更适合在消费者出声明！**

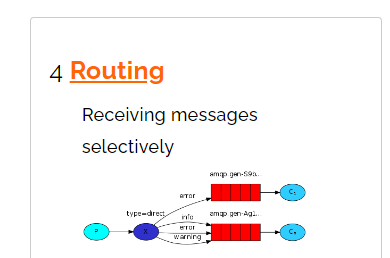
总结：

Direct交换机和Fanout交换机的差异：

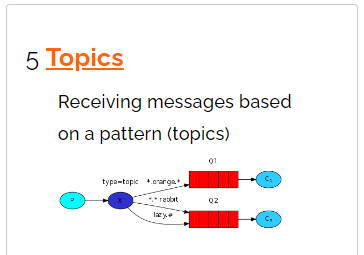
1：Fanout交换机将消息路由给每一个与之绑定的队列；

2：Direct交换机根据RoutingKey判断路由给哪个队列；

3：如果多个队列具有相同的RoutingKey，则与Fanout功能相同；



##### 5：Topic Exchange：主题交换机/话题交换机



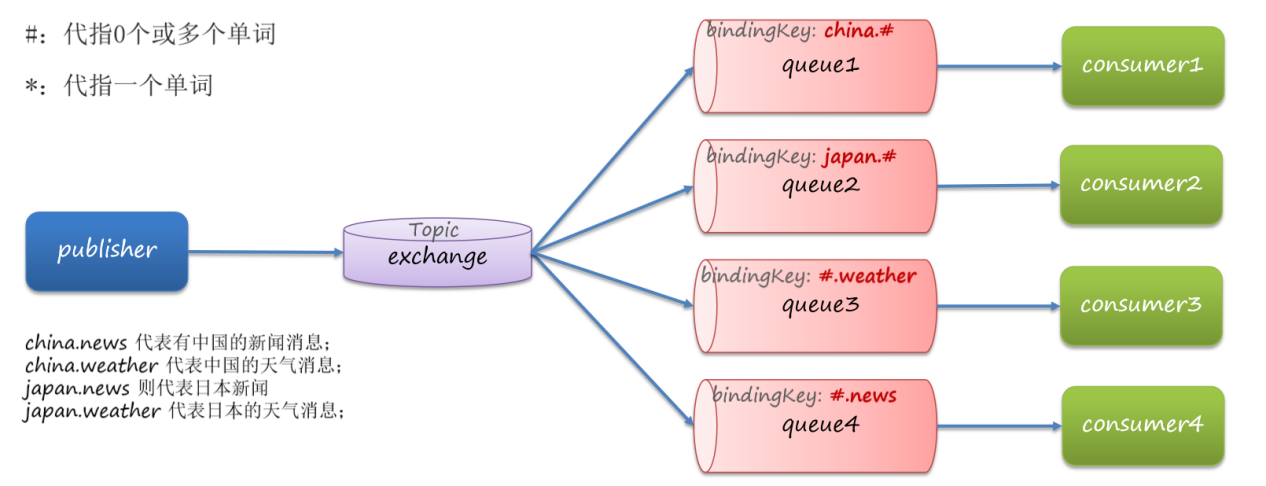
TopicExchange与DirectExchange类似，区别在于routingKey必须是多个单词的列表，并且以点.分割；

Queue与Exchange指定BindingKey时可以使用通配符：

#：代指0个或者多个单词

\*：代指一个单词

可以看出，TopicExchange与DirectExchange相比，没有大的变化，主要就是用通配符来简化了绑定关系的书写，原来需要用多个BindingKey的，现在可以用一个BindingKey表示，见下图：



#### 消息转换器

消息转换器：其实这个消息转换器，一直贯穿我们之前的学习！

说明：

在SpringAMQP的发送消息的方法中，接收消息的类型是Object，也就是说我们可以发送任意对象类型的消息，SpringAMQP会帮我们序列化为字节后发送SpringAMQP默认用的序列化方式是jdk的序列化：application/x-java-serialized-object，这种序列化有缺点：一是序列化性能比较差；二是安全性有问题，容易出现注入问题；三是序列化后数据长度太长了，消息体越大，消息传输的速度越慢，占用内存越大。所以说，我们非常不推荐用默认的序列化方式！

Spring的对消息对象的处理是由org.springframework.amqp.converter.MessageConverter来处理的。而默认实现是SimpleMessageConverter，基于JDK的ObjectOutputStream完成序列化。如果要修改只需要定义一个MessageConverter类型的Bean即可，推荐用Json方式序列化，步骤如下：

1：在publisher服务、consumer服务引入jackson-dataformat-xml依赖；

2：在publisher服务、consumer服务中声明MessageConverter：使用Jackson2JsonMessageConverter覆盖默认的；

3：consumer服务的监听器中消费消息（发送时什么对象，消费时就用什么对象接收）。

总结：

SpringAMQP中消息的序列化和反序列化是怎么实现的？

1：利用MessageConverter实现，默认是JDK的序列化；

2：注意发送发与接收方必须使用相同的MessageConverter