## 介绍与说明

Rabbitmq是一款比较成熟的消息中间件；

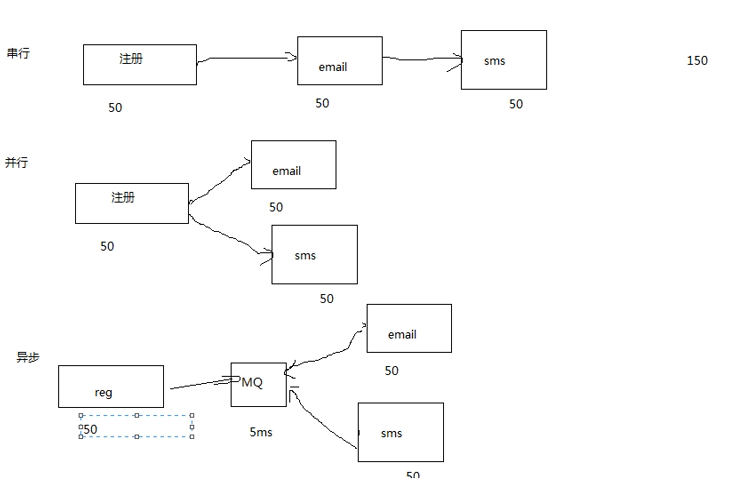
类似的产品还有：

RabbitMQ   
ZeroMQ   
ActiveMQ   
……

### RabbitMQ基本用途：

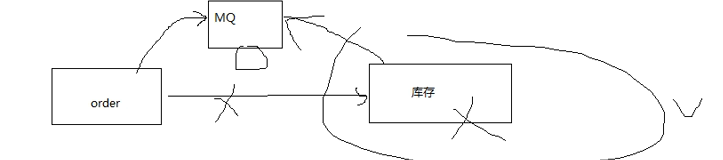
异步处理、应用解耦、流量削峰、日志处理等

1. 异步处理



比如注册一个账号完成后发送email和短信SMS，普通模式可能选择串行或者并行，其本身都有自己的局限，串行是发送完一个再发送另外一个，并行是同步发送两个；但实际发送email和发送SMS之间并没有依赖关系，因此不管串行还是并行的方式都不满足；而加入中间件后，注册成功后向mq发送一段消息，email和sms发送程序作为两个独立的消息消费者绑定了这个mq队列就能获取到这段消息，从而各自处理自己发送注册成功的邮件和短信；

1. 应用解耦

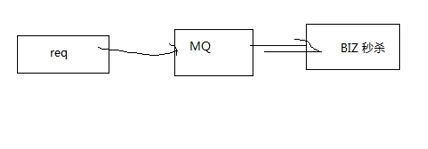


消弱系统间强依赖的情况；

比如订单系统获取到新的订单信息，需要去查询库存系统已确定是否能正常处理新的订单；但如果库存系统挂了，强依赖的情况下，订单系统就会报错，新的订单也会受到影响；而加入mq，订单系统发送消息给mq，再传递消息给库存系统，这样就减少了因为某一系统内部出现问题造成整个系统之间出现问题的风险；

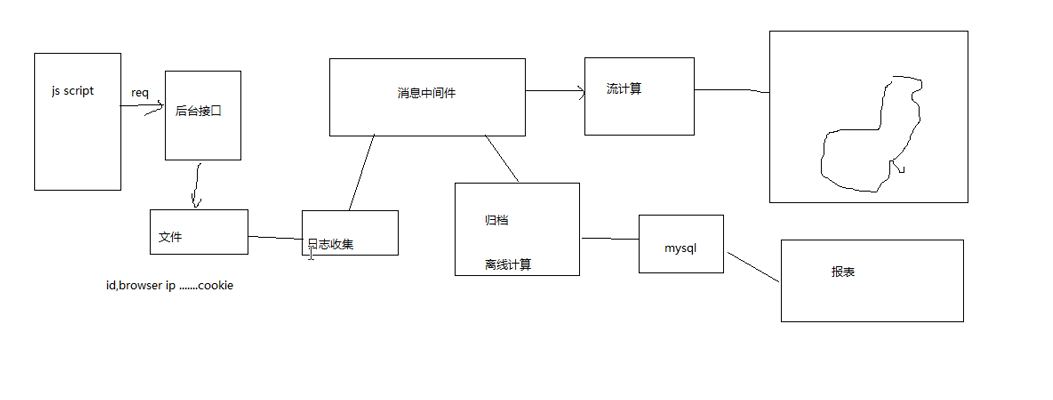
当然mq作为基础服务暂不考虑它挂的情况，实际上如果mq服务挂掉且没有做持久化处理，还是会有整个系统崩溃的危险；

1. 流量削峰



比如淘宝双十一购物节，再某一时间点开始，海量数据如果直接冲击系统，一般情况下服务器根本无法承受会直接宕机；而加入mq可以起到一个缓冲的作用，请求数据可以先存到mq中，小批量的发送到服务器进行处理，处理完一批再发一批，而且可以过滤掉一些无效的请求消息；

1. 日志处理



同样淘宝双十一购物节，可以通过分析处理订单产生的日志，来获取数据信息，统计订单的分类属性等，日志收集系统可以将日志文件中的订单的关键信息比如id、ip、cookie、地址、物品分类等等数据汇总后发送消息到mq，mq再分发到不同的应用服务中，比如可以做成实时数据统计展示再大屏幕上，也可以分发到归档系统，保存到mysql数据库再制作成报表等；

如果你想问这个图的右上角那个框里是什么东西，答案是那是中国地图……

RabbitMQ：erlang语言 开发的。

Python中连接RabbitMQ的模块：pika 、Celery(分布式任务队列) 、haigha   
可以维护很多的队列

RabbitMQ 教程官网：<http://www.rabbitmq.com/getstarted.html>

### **几个概念说明：**

Broker：简单来说就是消息队列服务器实体。 类似于mysql的数据库服务器  
Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。   
Queue：消息队列载体，每个消息都会被投入到一个或多个队列。   
Binding：绑定，它的作用就是把exchange和queue按照路由规则绑定起来。   
Routing Key：路由关键字，exchange根据这个关键字进行消息投递。   
vhost：虚拟主机，一个broker里可以开设多个vhost，用作不同用户的权限分离。 （类似mysql的db）  
producer：消息生产者，就是投递消息的程序。   
consumer：消息消费者，就是接受消息的程序。   
channel：消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务

## RabbitMQ安装

### **ubuntu系统**

install rabbitmq-server # 直接搞定

### **centos系统**

1）Install Erlang

# For EL5:

rpm -Uvh http://download.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rpm

# For EL6:

rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm

# For EL7:

rpm -Uvh http://download.fedoraproject.org/pub/epel/7/x86\_64/e/epel-release-7-8.noarch.rpm

yum install erlang

2）Install RabbitMQ Server

rpm --import https://www.rabbitmq.com/rabbitmq-release-signing-key.asc

yum install rabbitmq-server-3.6.5-1.noarch.rpm

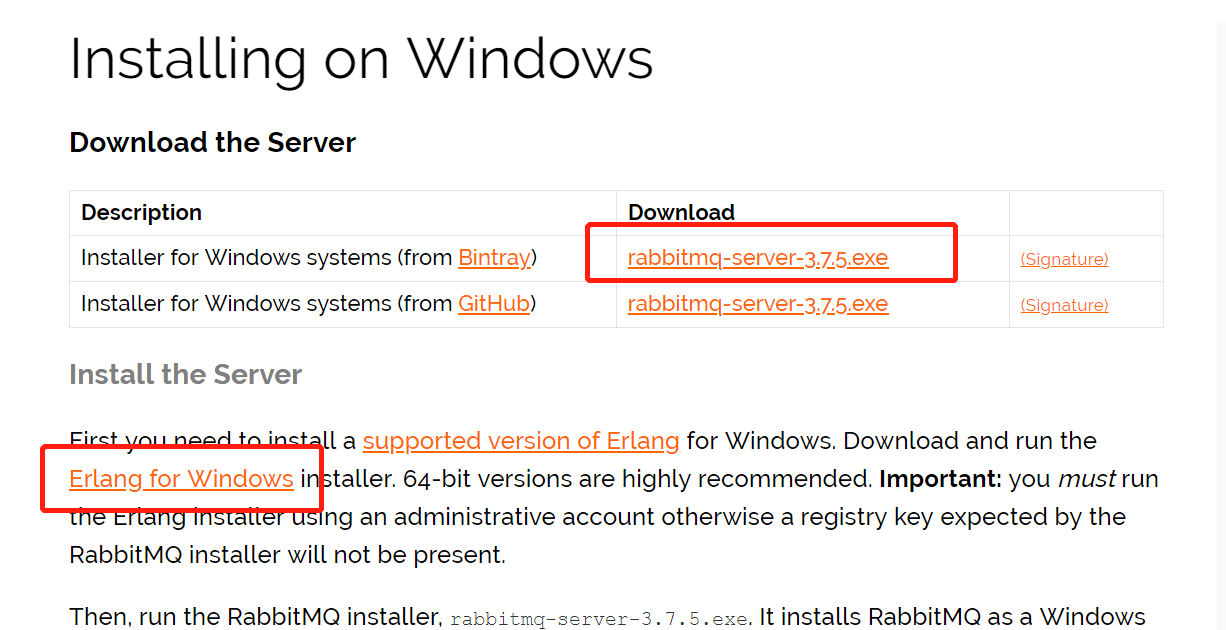
3）use RabbitMQ Server

chkconfig rabbitmq-server on

service rabbitmq-server stop/start

### windows系统

官网：<http://www.rabbitmq.com/>

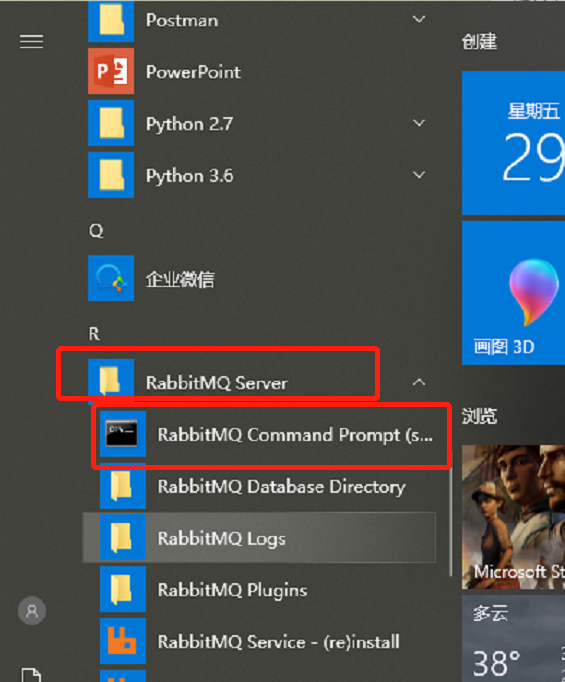


Erlang是rabbitmq的开发语言，需要先安装运行环境



分别下载安装包后，先安装OPT这个，再按rabbitmq的；

安装完后，在windows的开始列表找到



点击打开rabbitmq的控制台，在控制台输入

rabbitmq-plugins enable rabbitmq\_management



之后打开浏览器，输入

<http://localhost:15672>

看看是否能打开rabbitmq的客户端页面；

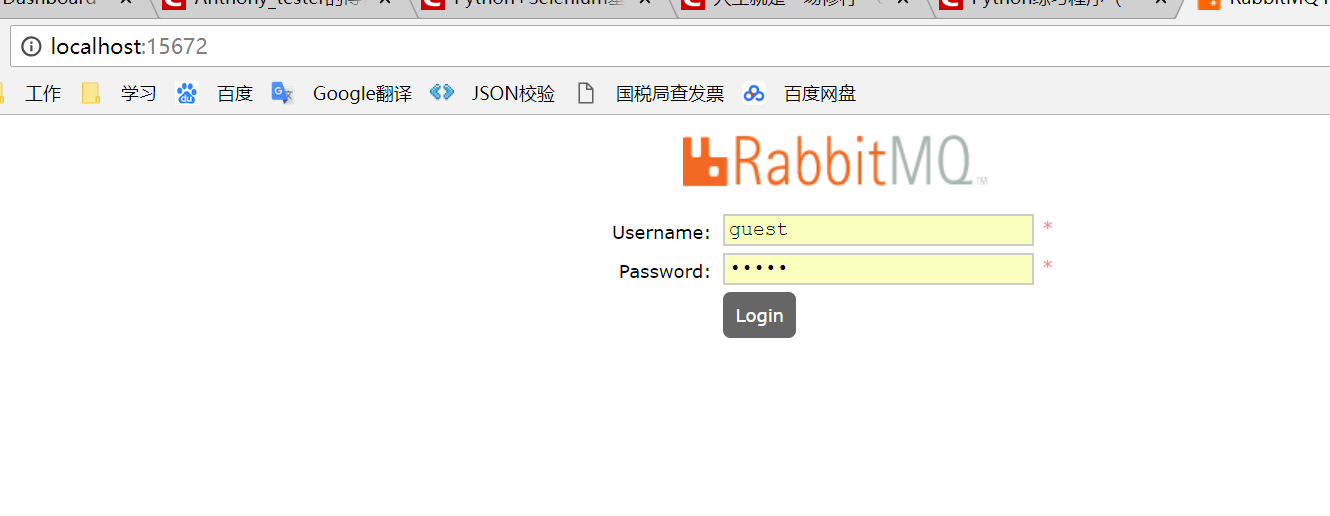
如果无法在web页打开管理界面，则继续按照以下方式操作：

使用管理员打开*cmd执行此命令：*net stop RabbitMQ && net start RabbitMQ



重新打开<http://localhost:15672/> ，你会发现管理页面出来了。

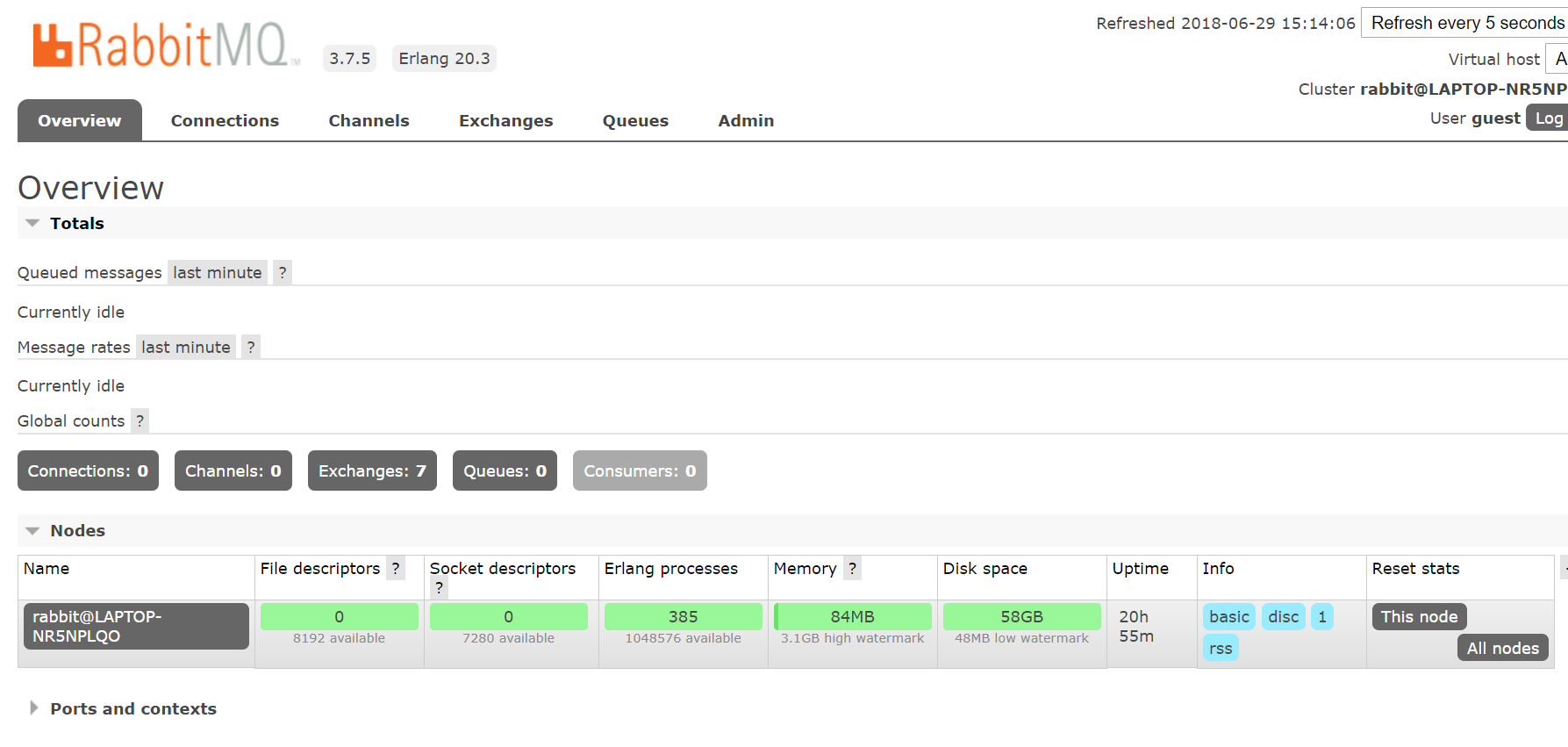
默认账号密码都是 guest/ guest



登陆后，安装就完成了

## 三、客户端页面基本操作

### Overview



是整体的统计与展示，以及一些监控数据；

程序中定义mq相关对象的大体顺序为：

1. 定义connection
2. 通过connection定义channel，一般都是channel干活，类似于mysql的cursor；
3. 通过channel定义exchange并确定如有规则等
4. 通过channel定义queue并与exchange绑定

所以知道这个流程，在看页面中的tab页就能清晰一些；

分别都有connections、channels、exchanges、queues的管理页面；

### Admin

1. 添加用户

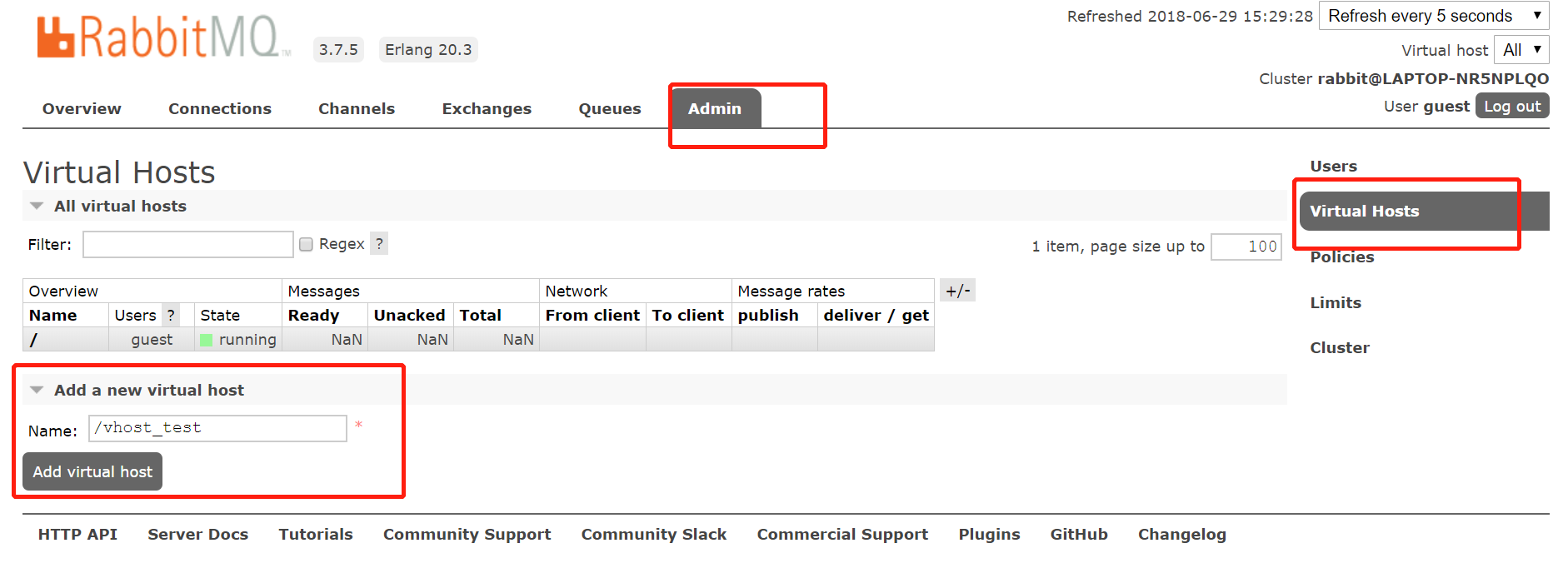


一般我们不用安装完成后默认给的超级管理员账号操作一些东西，可以新建一个用后超级管理员权限的用户；

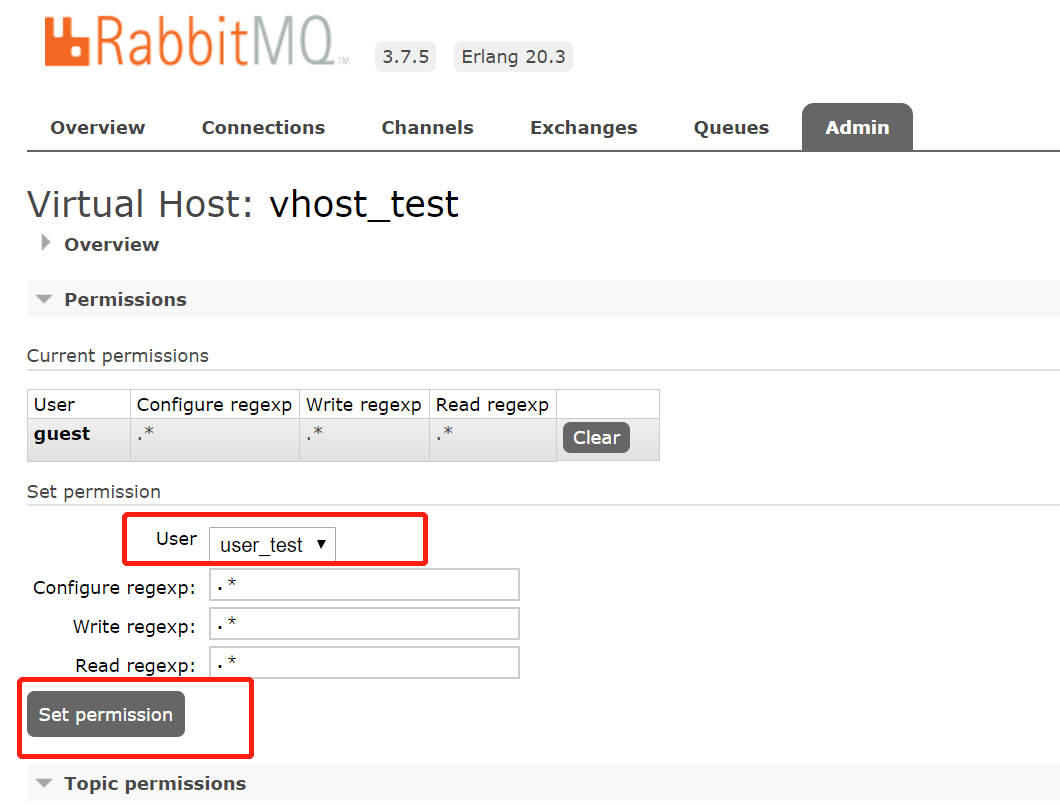


创建完成后，在列表中可以看到，Can access virtual hosts项为No access；上面说过，virtual hosts类似于mysql中的db，要建表需要先创建db；在这里要建queue，要先建virtual hosts；

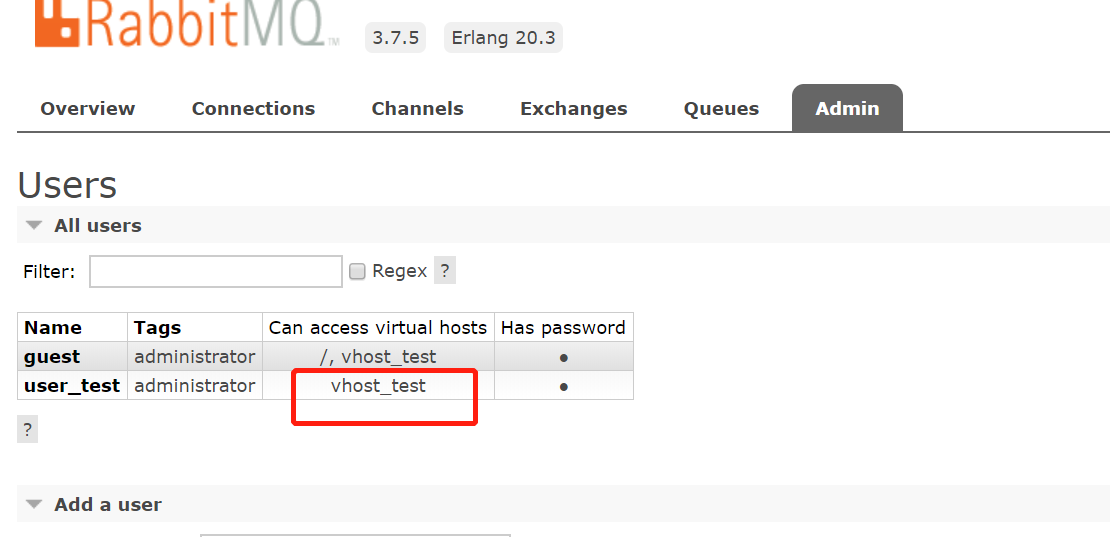
1. 创建virtual host



创建完成后，列表中点击名称，打开该virtual host操作管理页面，选择之前创建的用户，set permission，



就能赋予用户该virtual host的访问权限；

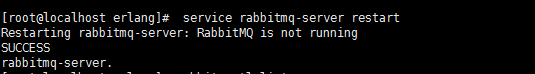


创建connections、channels、exchanges、queues等都可以在此创建，而且比较简单，在此不在缀文；

## 四、rabbitmq 管理 及 常用命令

安装完服务就用这个来启动：

service rabbitmq-server restart



RPM安装的，那就启动进程，启动控制台插件就可以了

**Rabbitmq服务器**的主要通过rabbitmqctl和rabbimq-plugins两个工具来管理，以下是一些

### 常用功能

1）. 服务器启动与关闭

启动: rabbitmq-server –detached

关闭:rabbitmqctl stop

若单机有多个实例，则在rabbitmqctlh后加–n 指定名称

注：

如果执行命令，报一下错误：

Error: unable to perform an operation on node 'rabbit@DESKTOP-6JT7D2H'. Please see diagnostics information and suggestions below.

解决方法：

C:\Users\[自己本地用户目录]\.erlang.cookie 文件拷贝到

C:\Windows\System32\config\systemprofile替换掉.erlang.cookie文件

重启rabbitMQ服务：net stop RabbitMQ && net start RabbitMQ

2）. 插件管理

开启某个插件：rabbitmq-pluginsenable xxx

关闭某个插件：rabbitmq-pluginsdisablexxx

注意：重启服务器后生效。

3）.virtual\_host管理

新建virtual\_host: rabbitmqctladd\_vhost xxx

撤销virtual\_host:rabbitmqctl delete\_vhost xxx

4）. 用户管理

新建用户：rabbitmqctl add\_user xxxpwd

删除用户: rabbitmqctl delete\_user xxx

改密码: rabbimqctlchange\_password {username} {newpassword}

设置用户角色：rabbitmqctlset\_user\_tags {username} {tag ...}

Tag可以为 administrator,monitoring, management

5）. 权限管理

权限设置：set\_permissions [-pvhostpath] {user} {conf} {write} {read}

Vhostpath

Vhost路径

user

用户名

Conf

一个正则表达式match哪些配置资源能够被该用户访问。

Write

一个正则表达式match哪些配置资源能够被该用户读。

Read

一个正则表达式match哪些配置资源能够被该用户访问。

6）. 获取服务器状态信息

服务器状态：rabbitmqctl status

队列信息：rabbitmqctl list\_queues[-p vhostpath] [queueinfoitem ...]

Queueinfoitem可以为：name，durable，auto\_delete，arguments，messages\_ready，

messages\_unacknowledged，messages，consumers，memory

Exchange信息：rabbitmqctllist\_exchanges[-p vhostpath] [exchangeinfoitem ...]

Exchangeinfoitem有：name，type，durable，auto\_delete，internal，arguments.

Binding信息：rabbitmqctllist\_bindings[-p vhostpath] [bindinginfoitem ...]

Bindinginfoitem有：source\_name，source\_kind，destination\_name，destination\_kind，routing\_key，arguments

Connection信息：rabbitmqctllist\_connections [connectioninfoitem ...]

Connectioninfoitem有：recv\_oct，recv\_cnt，send\_oct，send\_cnt，send\_pend等。

Channel信息：rabbitmqctl list\_channels[channelinfoitem ...]

Channelinfoitem有consumer\_count，messages\_unacknowledged，messages\_uncommitted，acks\_uncommitted，messages\_unconfirmed，prefetch\_count，client\_flow\_blocked

### **常用命令：**

查看所有队列信息

rabbitmqctl list\_queues

关闭应用

rabbitmqctl stop\_app

启动应用，和上述关闭命令配合使用，达到清空队列的目的

rabbitmqctl start\_app

清除所有队列

rabbitmqctl reset

更多用法及参数，可以执行如下命令查看

rabbitmqctl

rabbitmq-server start 或者 service rabbitmq-server start #启动rabbitmq

rabbitmqctl list\_exchanges

rabbitmqctl list\_bindings

rabbitmqctl list\_queues #分别查看当前系统种存在的Exchange和Exchange上绑定的Queue信息。

rabbitmqctl status #查看运行信息

rabbitmqctl stop #停止运行rabbitmq

rabbitmq-plugins enable rabbitmq\_management

启动rabbitmq的图形管理界面，这个操作必须重启rabbitmq， 然后在web中 http://127.0.0.1:15672 用户名和密码都是guest guest。如果局域网无法访问设置防火墙过滤规则或关闭防火墙。

### **rabbitmq安装Management Plugin**

这个插件实际上已经随rabbitmq一道安装了，要做的是，enable它：

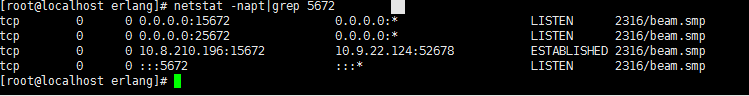
rabbitmq-plugins enable rabbitmq\_management

然后，重启rabbitmq：

service rabbitmq-server stop

service rabbitmq-server start

之后，netstat -napt|grep 5672 就会看到



通过 [http://rabbitmq-server-ip:15672](http://rabbitmq-server-ip:15672/)，和guest:guest的用户名密码就能登录管理页面了

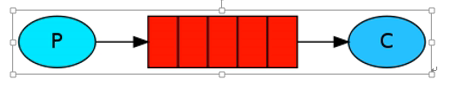
## 五、python操作rabbitmq基本示例

首先安装pika模块：

pip pika

### 简单的队列

#### 1.1 模型



P： producer，消息的生产者

红色的： 消息队列

C： consumer，消息的消费者

3个对象--生产者、队列、消费者

#### 1.2 定义消息生产者

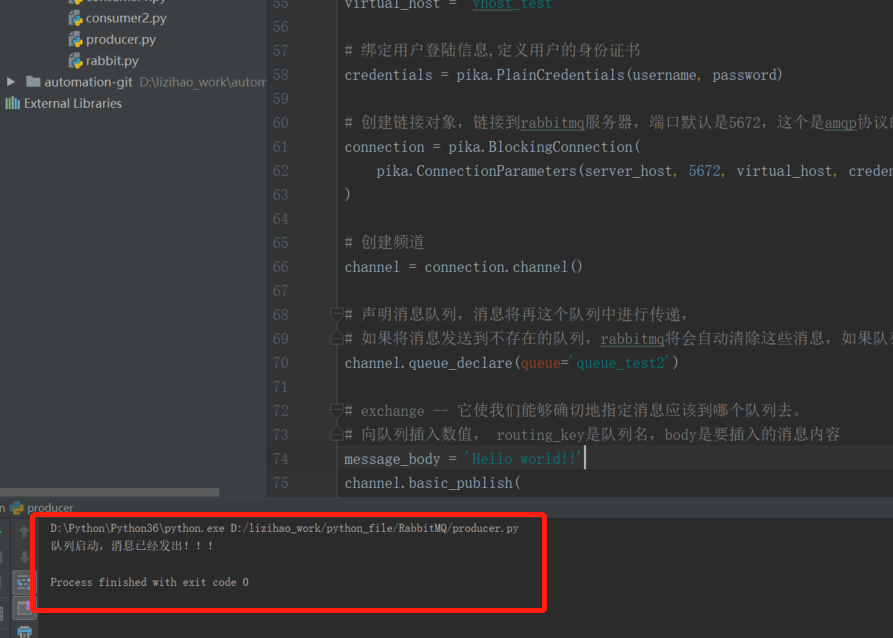
|  |
| --- |
| **import** pika  # 配置信息，可以定义在配置文件中 username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  # 绑定用户登陆信息,定义用户的身份证书 credentials = pika.PlainCredentials(username, password)  # 创建链接对象，链接到rabbitmq服务器，端口默认是5672，这个是amqp协议的端口 connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) )  # 创建频道 channel = connection.channel()  # 声明消息队列，消息将再这个队列中进行传递， # 如果将消息发送到不存在的队列，rabbitmq将会自动清除这些消息，如果队列不存在，则自动创建 channel.queue\_declare(queue='queue\_test2')  # exchange -- 它使我们能够确切地指定消息应该到哪个队列去。 # 向队列插入数值， routing\_key是队列名，body是要插入的消息内容 message\_body = 'Hello world!!' channel.basic\_publish(  exchange='',  routing\_key='queue\_test2',  body=message\_body )  # 代码是串行的，发送成功后会执行到该打印语句 print('队列启动，消息已经发出！！！')  # 缓冲区已经flush，而且消息已经确认发送到了rabbitmq中，关闭链接 connection.close() |

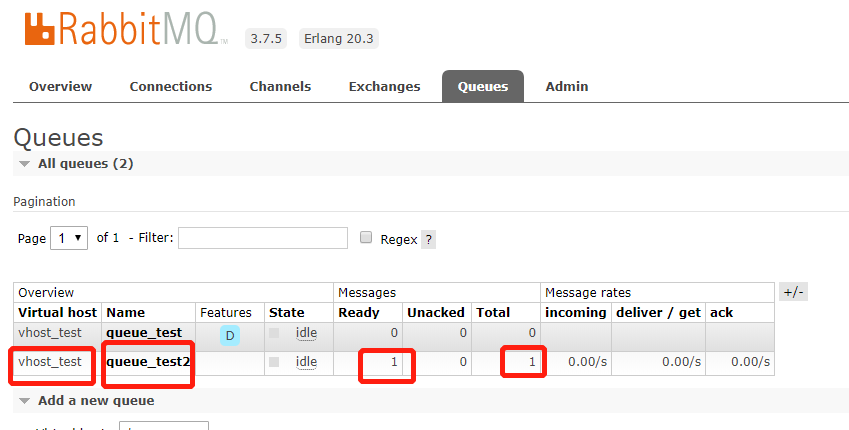
#### 1.3 定义消息消费者

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) )  channel = connection.channel()  # 声明消息队列，消息将在这个队列中进行传递，如果队列不存在，则创建 # 生产者和消费者都要声明一个相同的队列，用来防止某一方挂了，另外一方能正常运行 channel.queue\_declare(queue='queue\_test2')  # 定义一个回调函数来处理，这边的回调函数就是将信息打印出来，具体的参数用途后面再说 **def callback**(ch, method, properties, body):  print("[x] Recevied %s" % body)  # 告诉rabbitmq使用callback函数来接收信息 channel.basic\_consume(  callback,  queue='queue\_test2',  no\_ack=**True** # no\_ack=True表示回调函数中不需要发送确认标识 )  # mq中没有消息的话给出提示按ctrl+c退出。 print("[\*] Waiting for message. To exit press CTRL+C")  # 开始接收信息，并进入阻塞状态，队列里有信息才会调用callback函数进行处理 channel.start\_consuming() |

#### 1.4 效果

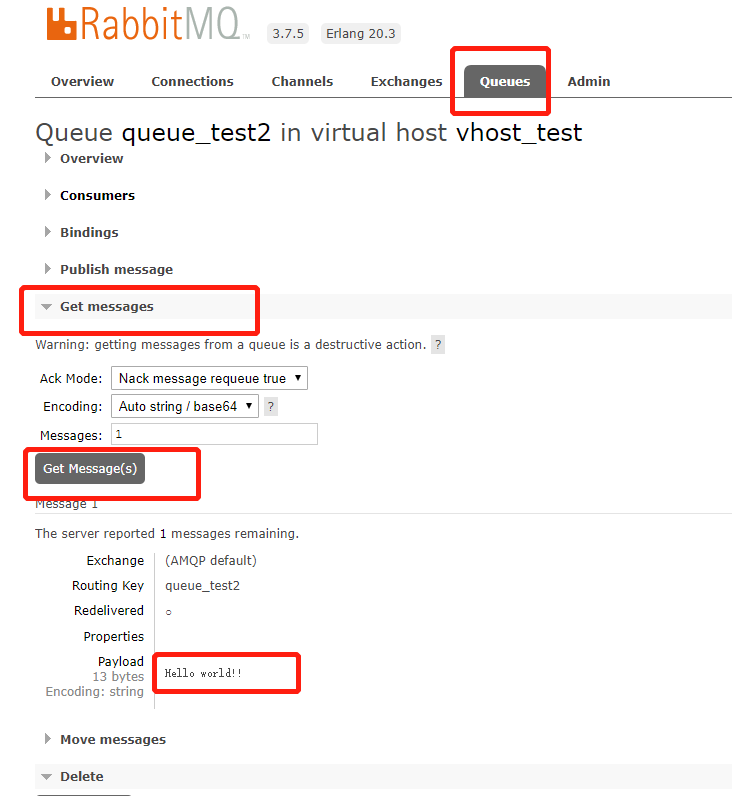
先不启动消费者，先启动生产者，生产者向mq中发送消息，mq会暂存消息：



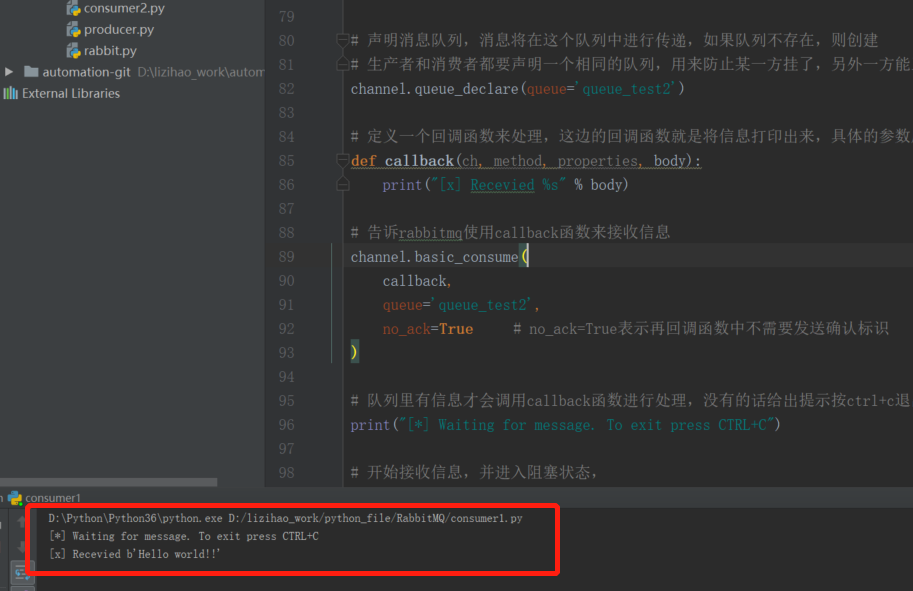


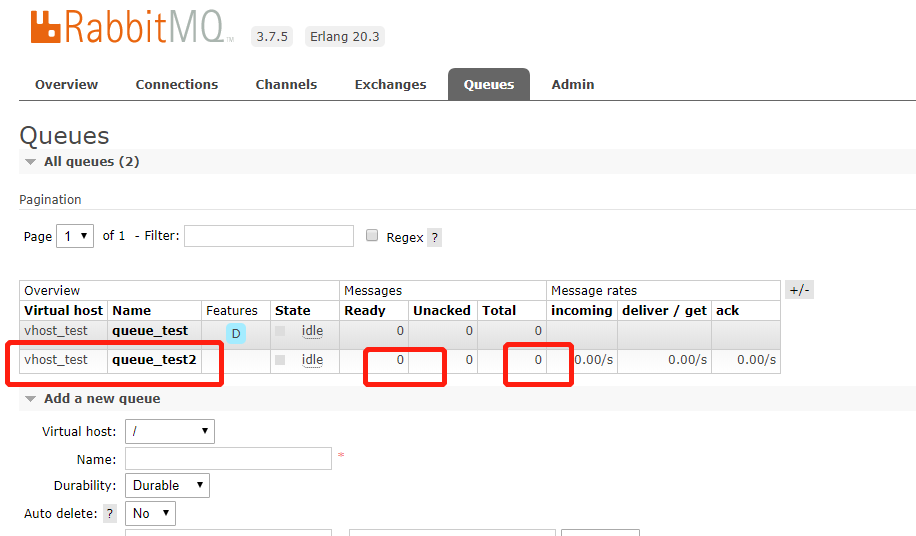
在定义的队列下，可以看到有一条消息待消费；

点击进入该队列，可以查看具体消息内容：



此时，如果启动消费者，mq中的消息将会被消费，





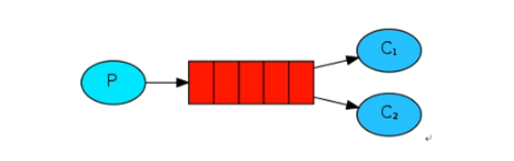
#### 1.5 简单队列的不足

耦合性高，消费者和生产者是一一对应的关系，如果想要多个消费者来获取消息就不足

队列名称变更时，每个都得相应的变更；

### 工作队列（轮询分发Round-robin）

#### 2.1 模型



为什么会出现工作队列：

Simple队列是一一对应的，在实际应用中，生产者发送消息是毫不费力的，而消费者通常跟实际的业务程序结合，处理消息可能要花费一定的时间，这样如果只用一个消费这来处理数据，mq会积压很多消息，影响系统的运行效率；

#### 定义消息生产者

将普通队列中定义的消费者利用for循环稍加修改，模拟发送多条数据；

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test2')  message\_body = 'Hello world!!'  # 通过for循环调用20次channel.basic\_publish()函数，模拟发送20条消息数据 **for** i **in** range(20):  channel.basic\_publish(  exchange='',  routing\_key='queue\_test2',  body="[%s]" % i + message\_body  )  print('队列启动，消息已经发出！！！') connection.close() |

#### 2.3 定义消息消费者1

消息消费者1，与普通队列定义的消费者相同：

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test2')   **def callback**(ch, method, properties, body):  print("Recevied %s" % body)  channel.basic\_consume(  callback,  queue='queue\_test2',  no\_ack=**True** )  print("[\*] Waiting for message. To exit press CTRL+C") channel.start\_consuming() |

#### 2.4 定义消息消费者2

消费者2中，callback()函数中增加一个time.sleep(3)，模拟该消费者处理消息需要耗时3s；

|  |
| --- |
| **import** pika **import** time  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test2')  **def callback**(ch, method, properties, body):  time.sleep(3) # 处理消息时，sleep 3秒，模拟耗时  print("Recevied %s" % body)  channel.basic\_consume(  callback,  queue='queue\_test2',  no\_ack=**True** )  print("[\*] Waiting for messages. To exit press CTRL+C") channel.start\_consuming() |

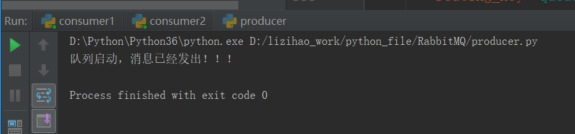
#### 2.5 效果

这样修改后的结果是，生产者生产20条数据发送到mq，消费者1和消费者2都从mq中获取消息；但消费者1处理消息快，消费者2处理消息慢，处理每条消息需要3秒；

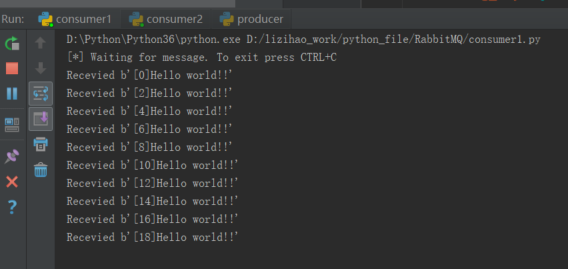
先启动消费者1和消费者2，让他们处于等待消息的状态，再启动生产者；

否则如果先启动生产者，消息会立即发送到mq，再启动消费者时，不管先启动哪一个，都会先开始处理消息，如果时先启动消费者1，mq中的20条数据会立刻被消费光；

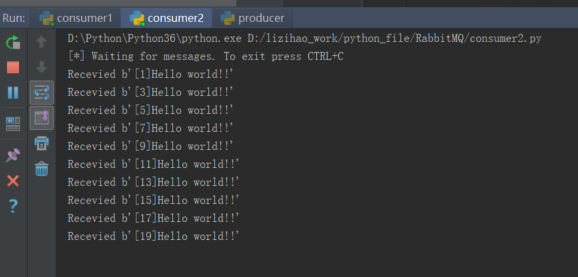
启动消费者1和2后，启动生产者：



消费者1的结果：



消费者2的结果：



分析：

从执行的结果看，消费者1处理了从0到18的偶数的10条消息，消费者2处理了从1到19的奇数的10条消息，虽然消费者2处理时，耗时比较久，但最后还是处理了10条数据；

其原理是，再生产者将20条消息发送到mq后，mq确定消息队列绑定了两个消费者，于是就不管什么情况，一人一条先分发出去，让消费者自己去处理，这样的分发方式叫做轮询分发；

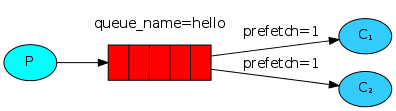
#### 2.6 轮询分发的不足

相信从过程中已经能够看出其中的不足，消费者2处理数据慢，却还是硬生生的塞了10条数据，等他慢慢处理；而消费者1迅速处理完10条数据后，处于闲置的状态；这样同样会引起整个系统效率的降低；

而且mq发送10条消息给消费者2，消费者2没有能迅速处理，此时如果消费者2挂掉，没处理的消息都会丢失

### 工作队列（公平分发Fair dispatch）

#### 3.1 模型



基本模型与轮询分发的一致，但是不同点是消费者处理消息的方式不一样了；

之前mq发送消息给消费者后，消费者开启了自动应答，会自动返回一个反馈给mq，而mq收到反馈后立即再发送消息给这个消费者；

而现在需要定义消费者时，将消费者的自动应答关闭，手动定义回执，处理完消息后返回一个回执给mq；这样相当于消费者处理完一个，mq才再发送一个；

这样的方式也一定程度上避免了轮询分发中消息容易丢失的问题，因为消息都在mq中，消费者处理完了并且mq获取到消费者处理完成的反馈消息，才会给新的消息出去；不会因为消费者挂了而导致的消息丢失；

#### 3.2 定义消息生产者

与轮询分发的生产者一致，模拟发送20条数据

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test')  message\_body = 'Hello world!!'  # 通过for循环调用20次channel.basic\_publish()函数，模拟发送20条消息数据 **for** i **in** range(20):  channel.basic\_publish(  exchange='',  routing\_key='queue\_test',  body="[%s] " % i + message\_body  )  print('队列启动，消息已经发出！！！') connection.close() |

#### 3.3 定义消息消费者1

定义的处理消息的函数callback()底部定义反馈确认的方法，再处理完消息（即这里的print）后，发送确认消息给mq，mq接受到该消息之后才会再次发送消息给该消费者；同时定义了手动发送确认，就得把channel.basic\_consume()中的自动应答关掉，即no\_ack置为False；

再调用callback()函数之前，定义channel.basic\_qos(prefetch\_count=1)，即每次最多处理1条数据，正在处理这1条数据时，mq不再发送消息给这个消费者，处理完后接收到确认反馈，再发送；

|  |
| --- |
| **import** pika **import** time  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test')   **def callback**(ch, method, properties, body):  print("Recevied %s" % body)  time.sleep(1)  ch.basic\_ack(delivery\_tag=method.delivery\_tag) # 给mq发送确认反馈，收到该反馈mq再发消息  channel.basic\_qos(prefetch\_count=1) # 每次最多处理1条消息，处理时不再接受消息 channel.basic\_consume(  callback,  queue='queue\_test',  no\_ack=**False** # 关闭自动应答，如果定义了手动反馈，这边还为True，会报错 )  print("[\*] Waiting for message. To exit press CTRL+C") channel.start\_consuming() |

#### 3.4 定义消息消费者2

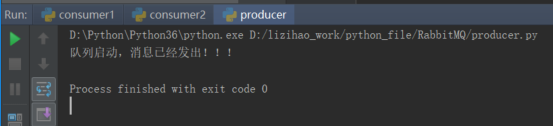
与消费者1的定义方法一致，也是关闭自动应答，定义手动反馈确认消息，并且定义最多处理1条消息；唯一与消费者1不同的时sleep()的时间，消费者1设置了sleep(1)，如果不设置，消费者2在sleep()的时候，消费者1就把20条数据处理完了；消费者2设置sleep 3s，造成一个场景是消费者2处理消息耗费的时间比消费者1要多；

|  |
| --- |
| **import** pika **import** time  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test')   **def callback**(ch, method, properties, body):  print("Recevied %s" % body)  time.sleep(3)  ch.basic\_ack(delivery\_tag=method.delivery\_tag) # 定义反馈机制，在处理完消息后反馈给mq  channel.basic\_qos(prefetch\_count=1) # 每次最多处理1条消息，处理时不再接受消息 channel.basic\_consume(  callback,  queue='queue\_test',  no\_ack=**False** # 自动应答置为False )  print("[\*] Waiting for messages. To exit press CTRL+C") channel.start\_consuming() |

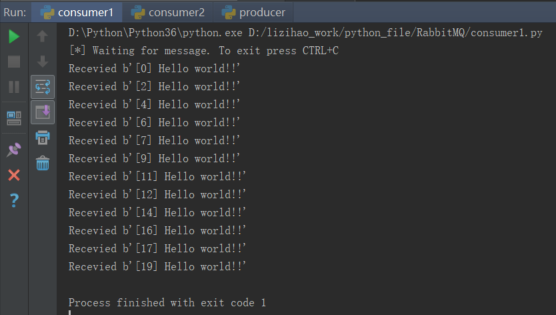
#### 3.5 效果

现在设置的效果是，两个消费者都是1次处理1条数据，处理完1条数据后，再接受mq发送的另一条消息；这样也就是说同样的时间下20条数据，处理数据快的能多处理，处理数据慢的会少处理，用较短的时间把20条数据处理完；

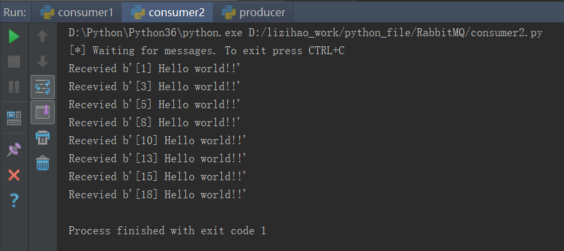
先启动消费者1和2，再启动生产者：



消费者1：



消费者2：



分析：

由结果可以看出，消费者1处理了12条数据，而且有奇数也有偶数；消费者2处理了8条数据，也是既有奇数也有偶数；这样就打破了轮询分发的限制，能者多劳，处理的快的就多处理一些，从而提高一些整体的效率；

#### 3.6 公平分发的不足

其实这个不足不是单独公平分发的不足，是以上几种形式的队列共有的不足；

即消息有可能会丢失，rabbitmq默认是采用内存存储，若在处理数据时，mq挂掉了，保存在内存中的消息数据肯定会清空的，仍然是消息丢失的问题，解决这个问题就引入了消息的持久化；

### 消息持久化

这边其实讲的是消息的持久化存储，既然rabbitmq把消息存储在内存中，mq挂机的话消息会丢失，那就想办法让mq把消息存储在硬盘中，这样mq即使挂了，重新启动后消息还是存在的，仍能继续分发消息；

声明消息持久化，其实是两部分的内容：

1. 声明消息队列queue持久化，即定义queue\_declare(queue=’xxx’, durable=True)，将队列保存在硬盘中；
2. 声明消息的持久化，定义生产者basic\_publish()函数时，加上消息持久化的声明；
3. 实际mq的持久化还有一个exchange交换机的持久化，后面介绍exchange的用法；

注意：

不能直接在已经定义好的消息队列中修改，使其变成持久化的方式，这样会保存；必须创建新的消息队列，或者删除已存在的队列重新创建；包括上面讲的自动应答方式的开启和关闭，不能直接修改，在创建的时候就得定义好；

#### 定义消息生产者

|  |
| --- |
| import pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  channel.queue\_declare(queue='queue\_test2', durable=True) # durable置为True，将queue声明为持久化  message\_body = 'Hello world!!!'  for i in range(20):  channel.basic\_publish(  exchange='',  routing\_key='queue\_test2',  body="[%s] " % i + message\_body,  properties=pika.BasicProperties( # 将具体消息声明为持久化  delivery\_mode=2,  )  )  print('队列启动，消息已经发出！！！') connection.close() |

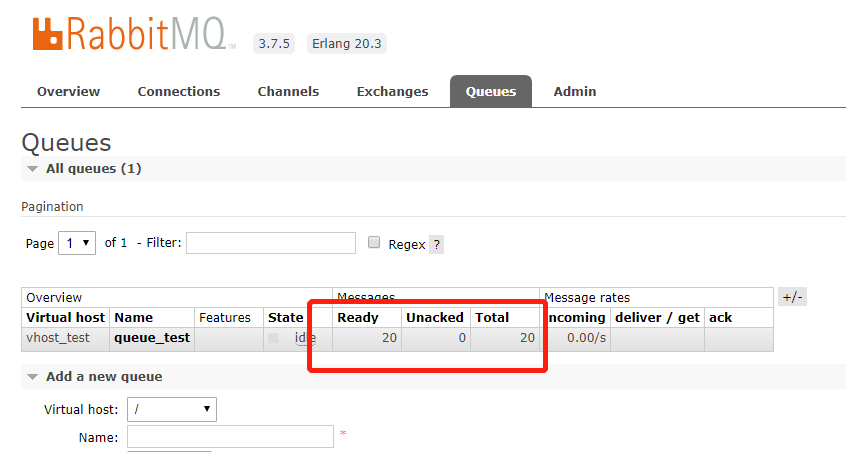
#### 定义消息消费者

|  |
| --- |
| import pika import time  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel() channel.queue\_declare(queue='queue\_test')   def callback(ch, method, properties, body):  print("Recevied %s" % body)  time.sleep(1)  ch.basic\_ack(delivery\_tag=method.delivery\_tag) # 给mq发送确认反馈，收到该反馈mq再发消息  channel.basic\_qos(prefetch\_count=1) # 每次最多处理1条消息，处理时不再接受消息 channel.basic\_consume(  callback,  queue='queue\_test',  no\_ack=False # 关闭自动应答，如果定义了手动反馈，这边还为True，会报错 )  print("[\*] Waiting for message. To exit press CTRL+C") channel.start\_consuming() |

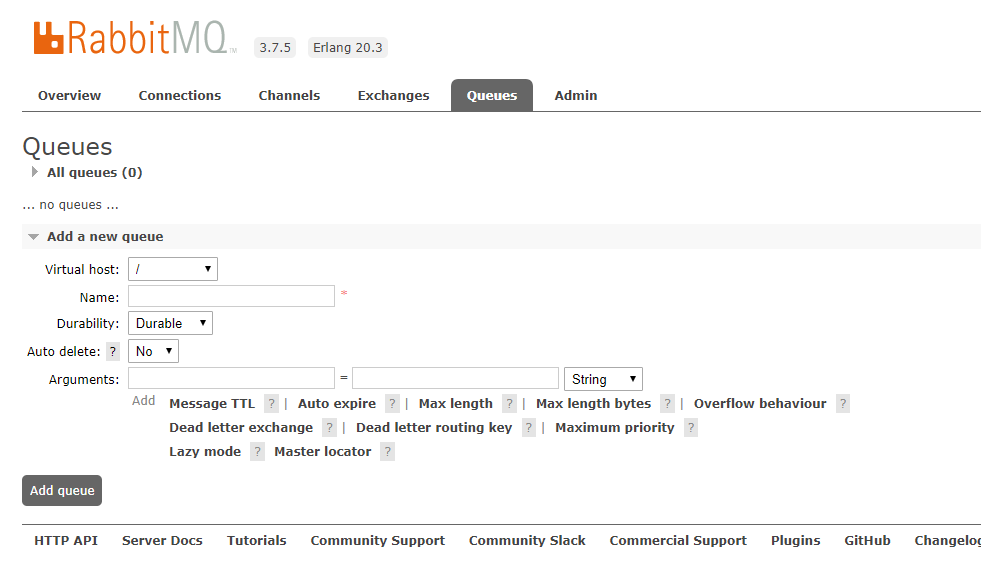
#### 4.3 效果

声明消息持久化主要是在定义生产者的时候声明，先不启动消费者，让生产者的消息发送到mq；

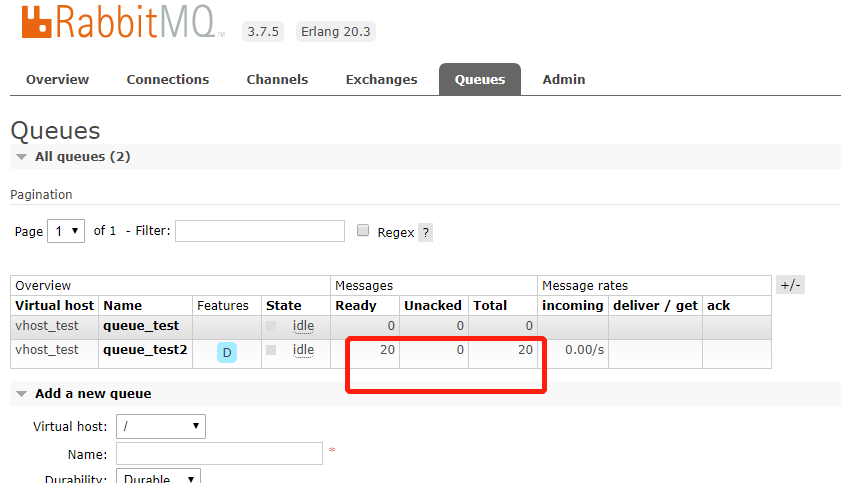
1. 未设置消息持久化的mq，启动消息生产者，消息发送到mq；



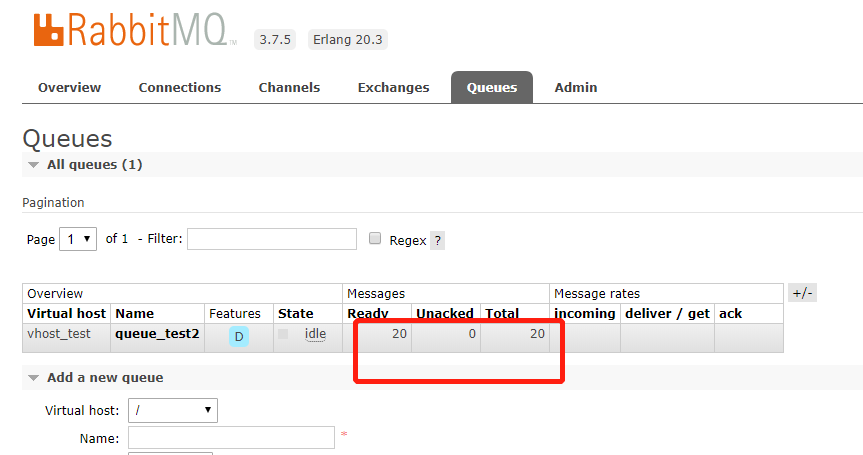
停掉mq后重启，队列和消息都不存在了；



1. 设置了消息持久化的队列，启动生产者，消息发送到mq：



停掉mq的服务，然后重启，可以看到队列和消息都还存在，而且蓝色的D表示durable是True的状态；



此时再启动消费者，消息同样能够被消费掉；

### 交换机exchange

以上谈到的mq的定义和使用方式都只是利用一个mq队列进行数据的分发和处理；但实际的工作需求可能会需要处理比较大量的消息数据，或者考虑业务逻辑的复杂程度，单一的mq队列满足不了实际需要，就得考虑多个消息队列的协同使用；

而实现多个消息队列协同工作，就需要用到exchange（交换机）来实现；

交换机的工作原理：

定义生产者时，定义exchange，并绑定queues到exchange，并且定义一套路由规则，消息发送时，生产者先将消息发送到exchange，exchange通过定义好的路由规则来确定消息分别需要发送到哪个mq，之后mq中的消息由各自绑定的消费者去处理；

而根据不同的路由规则，exchange大概分为3种模式：

模式1：Fanout 任何发送到Fanout exchange的消息都会被发到与该exchange绑定的所有queue上；

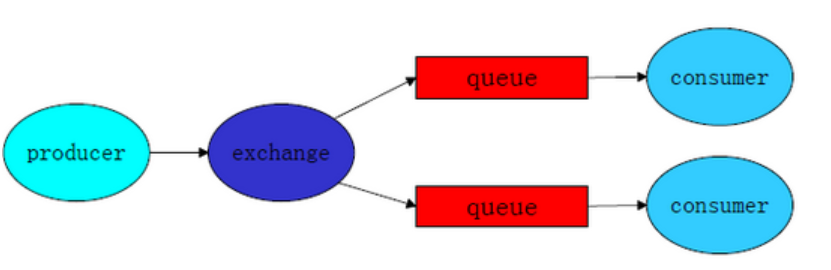
模式2：Direct exchange通过判定定义好的路由键来确定消息发送到哪个queue上；

模式3：Topic 路由键模糊匹配，通过通配符等来定义路由键，exchange模糊匹配路由键，匹配中后就将消息发送到相应的queue上；

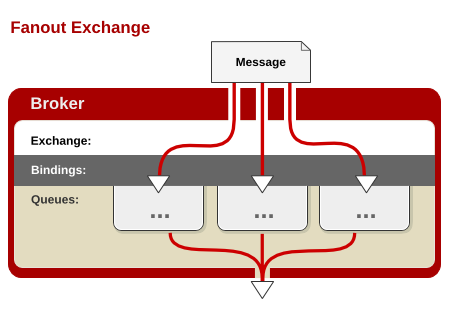
下面分别介绍不同模式的exchange的用法与特点；

### Fanout模式

#### 6.1 模型



匹配模式：



#### 6.2 特点

任何发送到Fanout exchange的消息都会被发到与该exchange绑定的所有queue上；

除此之外，Fanout exchange还有以下特点：

1. 可以理解为路由表模式
2. 这种模式不需要routing\_key，即使指定，也不生效；
3. 这种模式需要提前将exchange与queue绑定，一个exchange可以与多个queue进行绑定，一个queue也可以与多个exchange绑定；
4. 如果接受到消息的exchange没有与queue进行绑定，则消息被抛弃；

#### 6.3 定义消息生产者

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义交换机，exchange表示交换机名称，type表示类型 channel.exchange\_declare(  exchange='exchange\_test',  exchange\_type='fanout' )  message\_body = 'Hello World!!!'  # 将消息发送到交换机，而不是发送到queue了 channel.basic\_publish(  exchange='exchange\_test', # 指定exchange  routing\_key='', # fanout模式为广播模式，不需要路由键，配置了也不生效  body=message\_body )  print('队列启动，消息已经发出！！！') connection.close() |

#### 6.4 定义消息消费者1

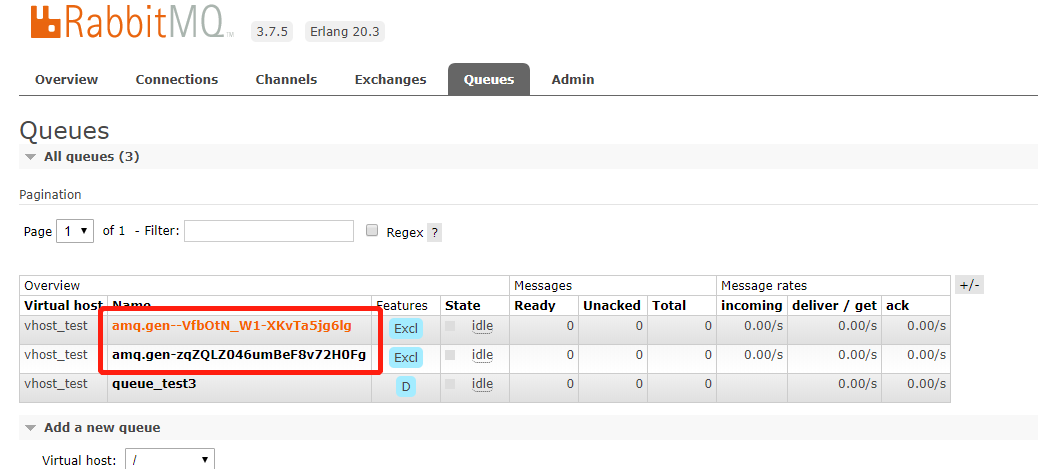
|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  channel.exchange\_declare(  exchange='exchange\_test',  exchange\_type='fanout' )  # 随机创建队列 result = channel.queue\_declare(exclusive=**True**) # exclusive=True表示建立临时队列，consumer关闭后，该队列就会被删除 queue\_name = result.method.queue  # 将queue与exchange进行绑定 channel.queue\_bind(  exchange='exchange\_test',  queue=queue\_name )   **def callback**(ch, method, properties, body):  print("Recevied message %s" % body)  channel.basic\_consume(  callback,  queue=queue\_name,  no\_ack=**True** )  print("[\*] Waiting for message. To exit press CTRL+C") channel.start\_consuming() |

#### 6.5 定义消息消费者2

|  |
| --- |
| import pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  channel.exchange\_declare(  exchange='exchange\_test',  exchange\_type='fanout' )  # 随机创建队列 result = channel.queue\_declare(exclusive=**True**) # exclusive=True表示建立临时队列，consumer关闭后，该队列就会被删除 queue\_name = result.method.queue  # 将queue与exchange进行绑定 channel.queue\_bind(  exchange='exchange\_test',  queue=queue\_name )   **def callback**(ch, method, properties, body):  print("Recevied message %s" % body)  channel.basic\_consume(  callback,  queue=queue\_name,  no\_ack=**True** )  print("[\*] Waiting for message. To exit press CTRL+C") channel.start\_consuming() |

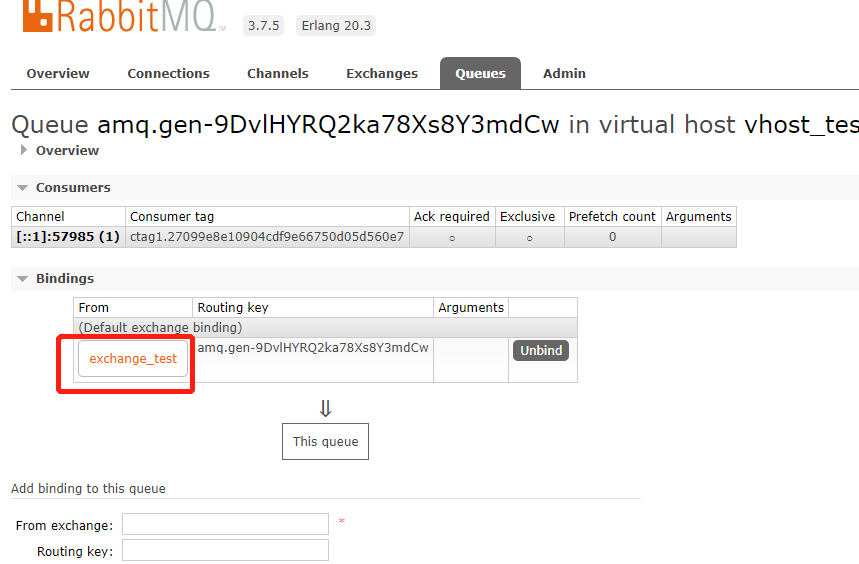
#### 6.6 效果

由代码可以看到，生产者中只声明了exchange，并没有声明queue，如果先启动了生产者，生产者向exchange中发送消息，而exchange如果没有绑定的queue，就会将消息抛弃；所以应该先启动消费者，消费者定义了相同的exchange，而且也定义了queue，只是这里的queue是随机创建的临时队列；之后再启动生产者，就能保证消息传递的准性；



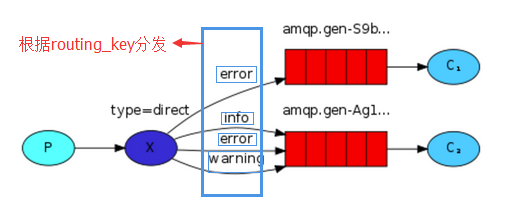
Queue列表中会有一个随机队列，其Features是Excl，表示exclusive=True随机创建；

点击queue名称，可以看到绑定的exchange：

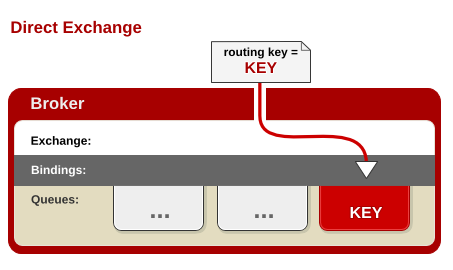


### Direct模式

#### 7.1 模型



匹配模式：



#### 7.2 特点

路由键的工作原理：

每个接收端的消息队列在绑定交换机的时候，可以设定相应的路由键，发送端通过交换机发送信息时，可以指明路由键，交换机会根据路由键把消息发送到相应的消息队列，这样接收端就可以接收到消息了；

任何发送到Direct Exchange的消息都会被转发到routing\_key中指定的queue：

1. 一般情况下可以使用rabbitMQ自带的Exchange:”” （该Exchange的名字为空字符串），也可以自定义Exchange
2. 可以将不同的routing\_key与不同的queue进行绑定，不同的queue与不同的exchange进行绑定；
3. 如果一个exchange声明为direct，并且bind中指定了routing\_key，那么发送消息时需要同时指明该exchange和routing\_key。
4. 如果消息中不存在routing\_key中绑定的队列名，则该消息会被抛弃；

#### 7.3 定义消息生产者

定义生产者时，除了定义交换机还定义了两个不同的路由键，一个info，一个error；并且定义了两个basic\_publish()函数来发送消息，一个发送带有info路由键的消息，另一个发送带有error路由键的消息；

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义交换机名称及类型 channel.exchange\_declare(  exchange='direct\_test',  exchange\_type='direct' )  severity1 = 'info' severity2 = 'error' message\_body = 'Hello World!!!'  # 发布消息至交换机direct\_test, 且发布的消息携带的关键字routing\_key是info channel.basic\_publish(  exchange='direct\_test',  routing\_key=severity1,  body=message\_body )  print("[x] Sent %s: %s" % (severity1, message\_body))  # 再定义一个basic\_publish发布消息至交换机direct\_test, 且发布的消息携带的routing\_key是error channel.basic\_publish(  exchange='direct\_test',  routing\_key=severity2,  body=message\_body )  print("[x] Sent %s: %s" % (severity2, message\_body))  connection.close() |

#### 7.4 定义消息消费者1

消费者1声明了同生产者同样的exchange，并且绑定exchange和随机queue时的routing\_key，声明为info，即消费者1应该接受到带有info路由键的消息；

|  |
| --- |
| **import** pika **import** time  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义exchange和类型 channel.exchange\_declare(  exchange='direct\_test',  exchange\_type='direct' )  # 生成随机队列 result = channel.queue\_declare(exclusive=**True**) queue\_name = result.method.queue severities = 'info'  # 将随机队列与routing\_key关键字以及exchange进行绑定 channel.queue\_bind(  exchange='direct\_test',  queue=queue\_name,  routing\_key=severities # 定义绑定随机队列与exchange的routing\_key是info )  print("[\*] Waiting for message. To exit press CTRL+C")   **def callback**(ch, method, properties, body):  time.sleep(3)  print("Recevied %s: %s" % (method.routing\_key, body))  channel.basic\_consume(  callback,  queue=queue\_name,  no\_ack=**True** )  channel.start\_consuming() |

#### 7.5 定义消息消费者2

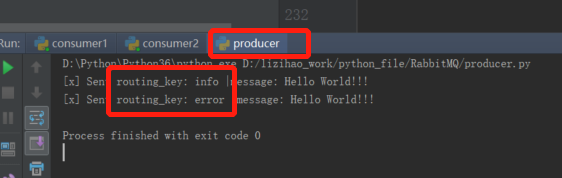
消费者2声明绑定queue和exchange的routing\_key为error，即消费者2应该能接受到带有error路由键的消息；

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义exchange和类型 channel.exchange\_declare(  exchange='direct\_test',  exchange\_type='direct' )  # 生成随机队列 result = channel.queue\_declare(exclusive=**True**) queue\_name = result.method.queue severities = 'error'  # 将随机队列与routing\_key关键字以及exchange进行绑定 channel.queue\_bind(  exchange='direct\_test',  queue=queue\_name,  routing\_key=severities # 定义绑定随机队列与exchange的routing\_key是error )  print("[\*] Waiting for message. To exit press CTRL+C")   **def callback**(ch, method, properties, body):  print("Recevied %s: %s" % (method.routing\_key, body))  channel.basic\_consume(  callback,  queue=queue\_name,  no\_ack=**True** )  channel.start\_consuming() |

#### 7.6 效果

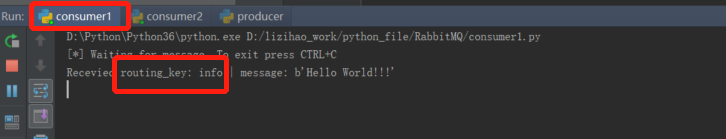
也是先启动消费者1和消费者2，否则生产者声明的exchange会没有queue与之绑定，消息会丢失；

启动生产者：



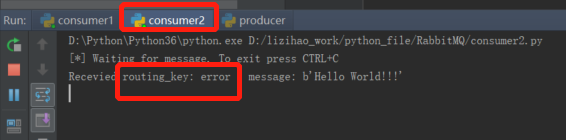
两条带有不同路由键的消息发送成功；

查看消费者1：



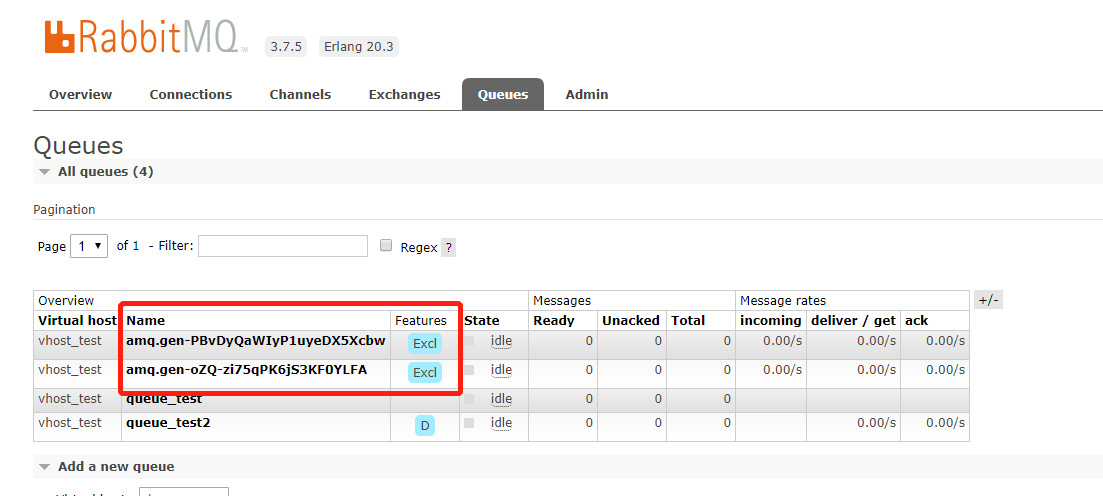
消费者1收到路由键为info的这条消息；

查看消费者2：



消费者2收到路由键为error的这条消息；

查看控制台：

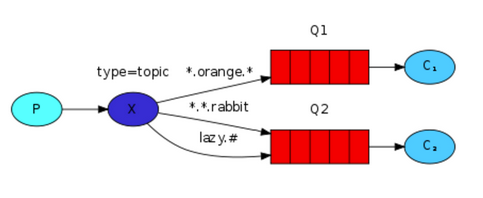


两个随机创建的临时消息队列，点击详情可查看与之绑定的exchange以及对应的routing\_key；

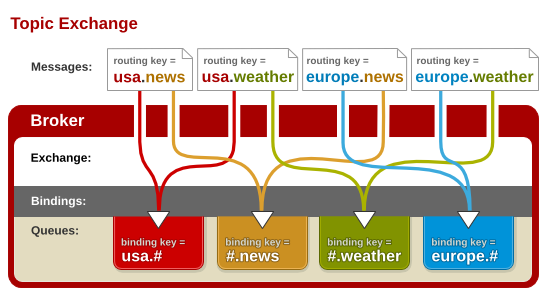
当然停止消费者1和消费者2，这两个queue会从列表中消失；

### Topic模式

#### 8.1 模型



匹配模式：



#### 8.2 特点

路由键模糊匹配：

其实是路由键的扩展，就是可以使用正则表达式；和常用的正则表达式不同，这里“#”表示所有、全部的意思；“\*”只匹配到一个词；

任何发送到Topic Exchange的消息都会被转发到所有关心routing\_key中指定主题的queue上；

1. 这种模式较为复杂，简单说就是每个队列都有其关心的主题，所有的消息都带有一个“标题”(routing\_key)，Exchange会将消息转发到所有关注主题能与routing\_key模糊匹配的队列；
2. 这种模式需要routing\_key，也需要提前绑定Exchange和queue；
3. 在进行绑定时，要提供一个该队列关心的主题，如“#.log.#”表示该队列关心所有涉及log的消息（routing\_key为“MQ.log.error“之类的消息会被转发到该队列）
4. “#“表示0个或者若干个关键字，”\*“表示一个关键字。如”log.warn“匹配，无法与”log.warn.timeout”匹配；但是”log.#“能与上述两者匹配；
5. 同样，如果Exchange没有发现能够与routing\_key匹配的queue，则会抛弃次消息；

Topic模式需要尤其注意的一点是如果一条消息的routing\_key能被两个queue匹配中，则这两个queue都会获取到这一条消息，分别给他们各自的消费者消费；

#### 8.3 定义消息生产者

生产者发送两条消息，带有的routing\_key分别为‘order.log.info’和’product.log.error’；

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义交换机名称及类型 channel.exchange\_declare(  exchange='topic\_test',  exchange\_type='topic' )  severity1 = 'order.log.info' severity2 = 'product.log.error' message\_body = 'Hello World!!!'  # 发布消息至交换机topic\_test, 且发布的消息携带的关键字routing\_key是order.log.info channel.basic\_publish(  exchange='topic\_test',  routing\_key=severity1,  body=message\_body )  print("[x] Sent routing\_key: %s |message: %s" % (severity1, message\_body))  # 再定义一个basic\_publish发布消息至交换机topic\_test, 且发布的消息携带的routing\_key是product.log.error channel.basic\_publish(  exchange='topic\_test',  routing\_key=severity2,  body=message\_body )  print("[x] Sent routing\_key: %s |message: %s" % (severity2, message\_body))  connection.close() |

#### 8.4 定义消息消费者1

声明同生产者同样的exchange，创建随机临时队列，绑定queue与exchange，routing\_key匹配规则为“#.info“，也就是说消费者1能匹配中第一条消息的routing\_key，可以拿到这条消息；注意这里说的拿到消息是queue拿到，而不是消费者拿到；

|  |
| --- |
| **import** pika **import** time  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义exchange和类型 channel.exchange\_declare(  exchange='topic\_test',  exchange\_type='topic' )  # 生成随机队列 result = channel.queue\_declare(exclusive=**True**) queue\_name = result.method.queue severities = '#.info'  # 将随机队列与routing\_key关键字以及exchange进行绑定 channel.queue\_bind(  exchange='topic\_test',  queue=queue\_name,  routing\_key=severities # 定义绑定随机队列与exchange的routing\_key是info )  print("[\*] Waiting for message. To exit press CTRL+C")   **def callback**(ch, method, properties, body):  time.sleep(3)  print("Recevied routing\_key: %s | message: %s" % (method.routing\_key, body))  channel.basic\_consume(  callback,  queue=queue\_name,  no\_ack=**True** )  channel.start\_consuming() |

#### 8.5 定义消息消费者2

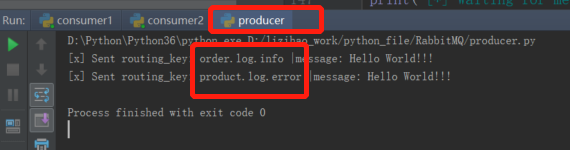
声明同生产者同样的exchange，创建随机临时队列，绑定queue与exchange，routing\_key匹配规则为“\*.log.\*“，也就是说消费者1能匹配中这两条消息的routing\_key，可以拿到这两条消息；注意这里说的拿到消息是queue拿到，而不是消费者拿到；

|  |
| --- |
| **import** pika  username = 'user\_test' password = '123456' server\_host = 'localhost' virtual\_host = 'vhost\_test'  credentials = pika.PlainCredentials(username, password) connection = pika.BlockingConnection(  pika.ConnectionParameters(server\_host, 5672, virtual\_host, credentials) ) channel = connection.channel()  # 定义exchange和类型 channel.exchange\_declare(  exchange='topic\_test',  exchange\_type='topic' )  # 生成随机队列 result = channel.queue\_declare(exclusive=**True**) queue\_name = result.method.queue severities = '\*.log.\*'  # 将随机队列与routing\_key关键字以及exchange进行绑定 channel.queue\_bind(  exchange='topic\_test',  queue=queue\_name,  routing\_key=severities # 定义绑定随机队列与exchange的routing\_key是error )  print("[\*] Waiting for message. To exit press CTRL+C")   **def callback**(ch, method, properties, body):  print("Recevied routing\_key: %s | message: %s" % (method.routing\_key, body))  channel.basic\_consume(  callback,  queue=queue\_name,  no\_ack=**True** )  channel.start\_consuming() |

#### 8.6 效果

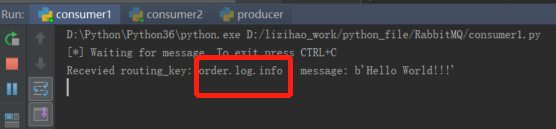
同样的也是需要先启动两个消费者，在启动生产者；

启动生产者：



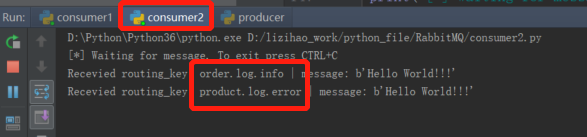
两条消息发送成功；

查看消费者1：



匹配规则为“#.info“，匹配中”order.log.info”的消息；

查看消费者2：



匹配规则为“\*.log.\*“，能匹配中两条消息的routing\_key，所以两条消息都能进入到消费者2的queue，能被消费者2处理掉；

## 六、其他

Rabbitmq的集群部署：

<https://blog.csdn.net/jxdl6655/article/details/78194191>

消息确认机制：事务机制、confirm机制

网络分区配置、节点容量配置、内存管理配置等等，有兴趣的可以自己学习一下；

## 七、测试rabbitmq

### 7.1 测试rabbitmq功能的测试方案

整体的测试思路大概分为5步：

1. 消息发送后能不能找到正确的mq消息队列
2. Mq获取到的消息是否正确；
3. Mq的消息能否被正确的消费者找到并正常处理
4. 消息被处理后是否有正确的回调和入库等操作；
5. 对MQ进行压测，灌入一定量的数据，看处理的消息是否准确，处理的性能是否可以接受；
6. 再有的话就是一些异常场景的测试

异常场景：

1、消费者启动后，未显示在队列的Consumers

2、消费者启动后，消费入库时报错

3、消费者启动后，输入正确的json，重复入库

4、消费者启动后，消费但未入库

5、消费者启动后，消费者刚开始显示，但后来消失(消费者假死)

6、消费者启动后，输入错误的json，消费失败

7、消费者启动后，消费者堵塞（队列阻塞，无法继续添加数据，可能导致服务挂掉）

## 八、 常见问题汇总

1. RabbitMQ队列超长导致QueueingConsumerJVM移除：

现象：

有消息队列中的消息积累很多条待分配的消息，并且卡住不再分发；而消费者的java服务也卡住不再处理消息；经过分析定位是QueueingConsumer持有大量对象导致JVM内存溢出，所以消费者不再消费也不再接收mq的消息，造成消息堵塞；

解决方法：

定义消费者是，调用channel的basicQos()方法，设置临时内存中最多保存的消息数，即每次最多能处理多少消息，这样就不会造成消费者一下接收过多的消息处理不过来，导致内存溢出卡死的结果；

当然这样的添加是解决了消费者的问题，但同时可能造成mq中堆积过多待处理的消息，这时候就应该考虑增加mq服务的数量，或提高mq的处理性能；