

一 消息队列

1 秒杀功能

秒杀对于系统的特点：瞬间高并发。在瞬间超过系统负荷的请求如何处理---消息队列缓存机制。

2 消息队列概括

2.A 队列

queue--队列

一般队列的结构都类似一个双向链表，头尾操作速度快；
先进先出.优先插队

2.B 消息队列

message queue--消息队列

队列中的内容就是消息.消息对象不是单纯的字符串等.

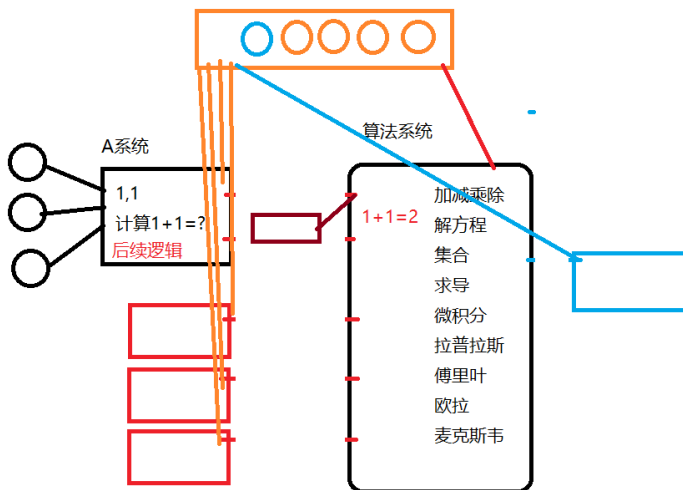
2.C 消息队列的作用

2.C.a 解耦

- 把同步代码换成异步
- 新增商品时(考虑缓存,考虑索引)
 - 添加缓存
 - 增量索引
 - insert 执行之前

2.C.b 消峰

- 消除并发的峰值
- 将超过上限的并发放到队列缓存等待



二 RabbitMQ

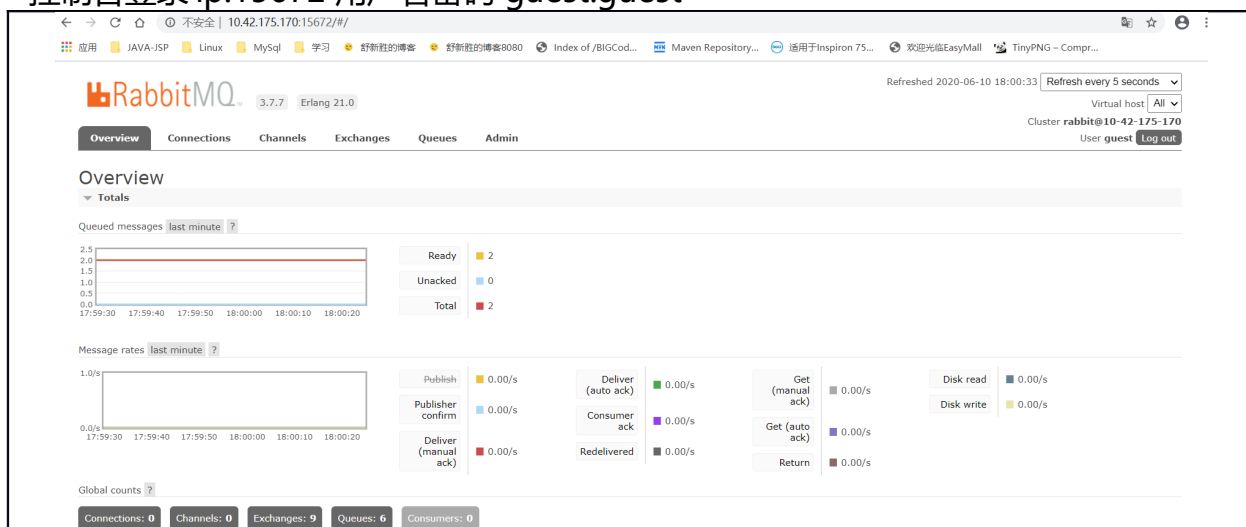
企业级别的常见的消息代理。可以通过使用 rabbitmq 第三方的队列技术实现消息的传递。

1 启动 rabbitmq

```
[root@10-42-175-170 ~]# service rabbitmq-server start
Starting rabbitmq-server:
SUCCESS
rabbitmq-server.
[root@10-42-175-170 ~]#
```

2 观察控制台

控制台登录 ip:15672 用户名密码 guest:guest

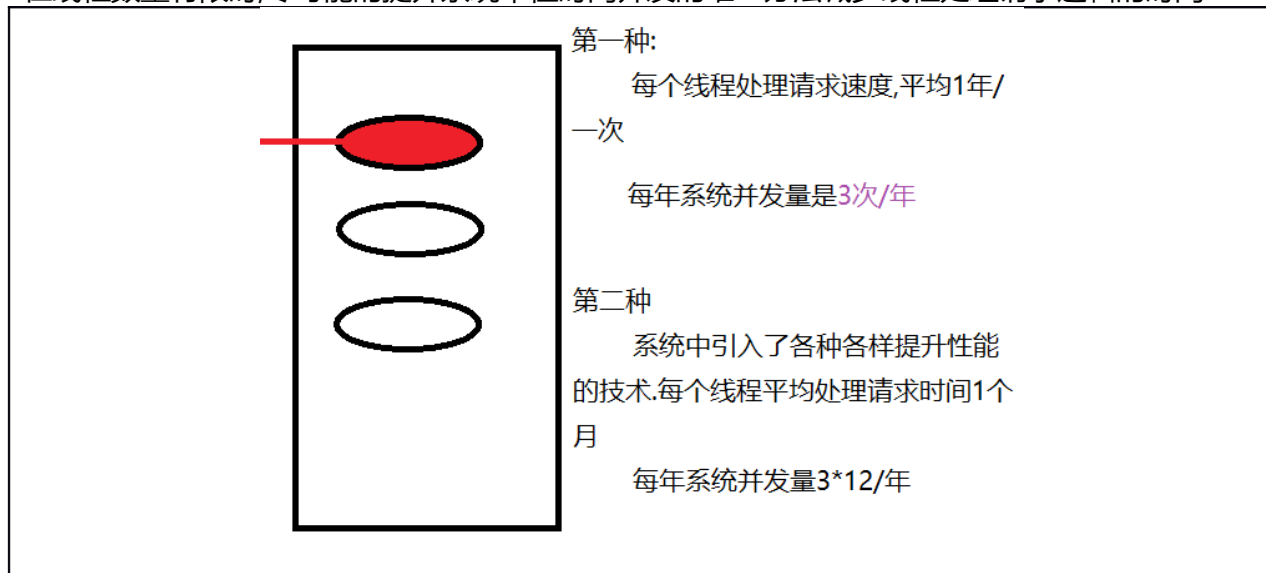


三 为什么引入 rabbitmq 能够提升系统并发

并发:

系统资源(线程),单位时间处理的请求.看成 web 应用的并发能力

在线程数量有限时,尽可能的提升系统单位时间并发的唯一办法减少线程处理请求逻辑的时间



四 RabbitMQ 结构

1 核心组件

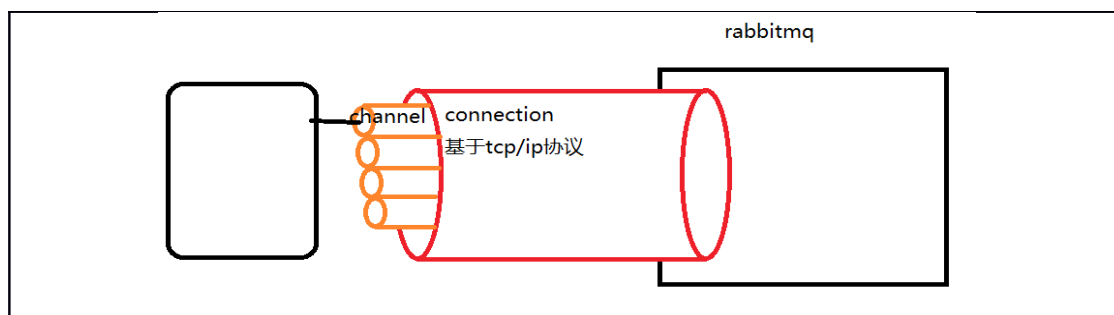
1.A 链接对象

程序链接 rabbitmq, 作为客户端操作消息队列, 需要通过链接对象对接。

每个进程会创建一个底层长链接, 链接 rabbitmq。

操作 rabbitmq 必须拿到一个信道对象--短连接。

长链接频繁销毁创建, 消耗系统资源。

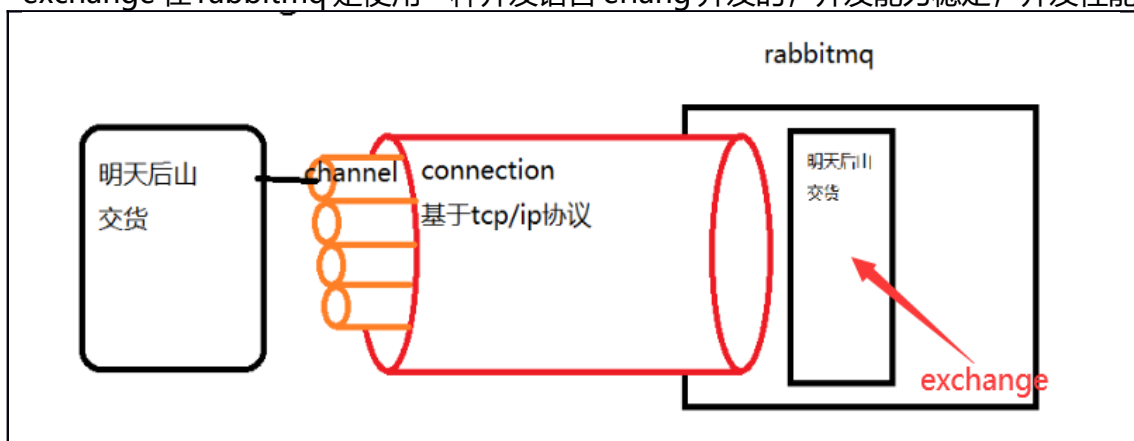


1.B 交换机 exchange

核心组件交换机—exchange。

消息的并发处理，并不会交给客户端去处理，客户端语言不同，并发能力也不同。统一由交换机接受客户端的消息。

exchange 在 rabbitmq 是使用一种并发语言 erlang 开发的，并发能力稳定，并发性能高。



控制台查看：

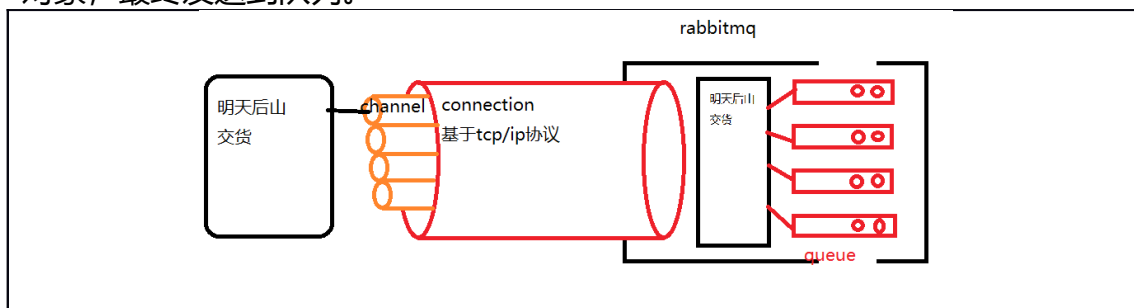
The screenshot shows the 'Exchanges' tab in the RabbitMQ management console. The page title is 'Exchanges' and it shows 'All exchanges (7)'. A red arrow points to the 'exchange' label in the header. Below the pagination controls, a table lists the available exchange types:

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			

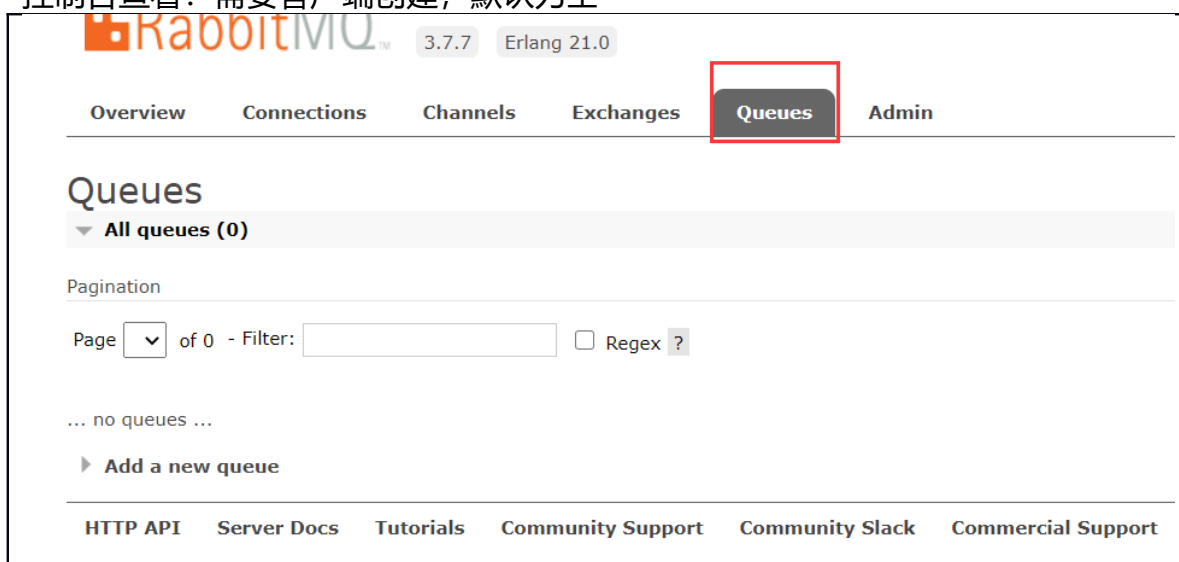
At the bottom, there is a link to 'Add a new exchange'.

1.C 队列 queue

队列是 rabbitmq 里接收、存储消息对象的组件，必须绑定一个交换机，当交换机接收到客户端发送的消息时，会根据路由逻辑判断当前消息发送给哪个/哪些队列，消息封装成对象，最终发送到队列。



控制台查看：需要客户端创建，默认为空



2 客户端角色

客户端：所有能连接 rabbitmq 的软件、代码、插件都可以是 rabbitmq 的客户端。

根据客户端不同的功能区分角色

2.A 生产者 productor

连接上 rabbitmq 后，将客户端准备好的消息发送到 rabbitmq，这个客户端就是生产者，而生产者从来不会把消息发给 queue 而是发给交换机。

A 系统发送消息 "hello", "hello" 最终到队列，A 就是生产者

2.B 消费者 consumer

监听队列,从队列中拿到已经存储的消息对象,进行客户端的消费逻辑.

2.C 无角色

如果客户端连接上 rabbitmq 没有发送消息,不需要接收消费消息,那这个客户端就是无角色的客户端。

可以实现通过连接来声明组件。

exchange 不是自动生成给你使用,不同的系统、不同的客户端可以有不同的自定义的交换机;
queue 也是一样,组件的创建过程也是由客户端完成的。

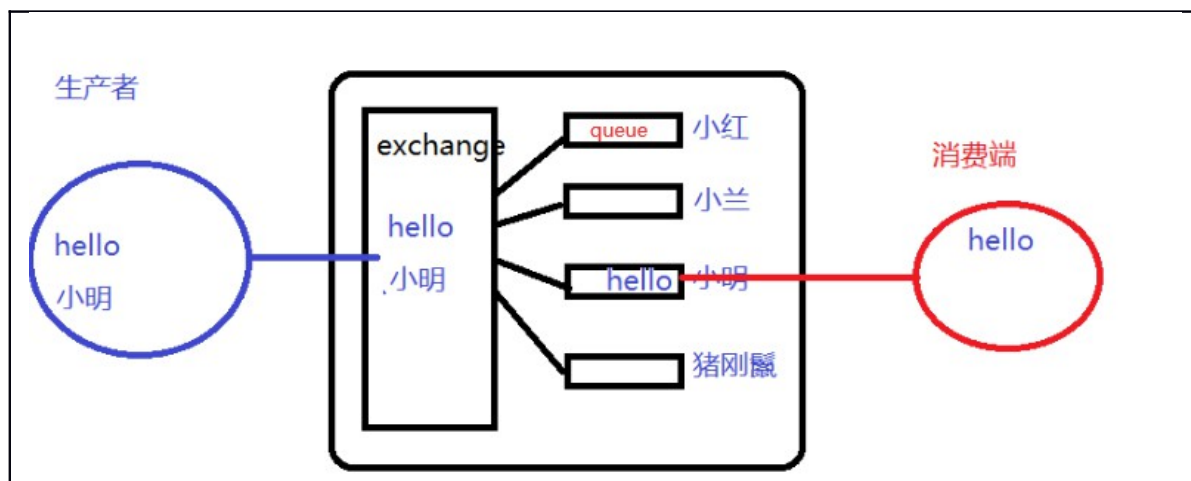
五 RabbitMQ 五种工作模式

前两种工作模式,并没有讨论交换机转发消息的逻辑,只讨论了消费端监听消息队列,消费消息的逻辑.

1 简单模式

- ★ 强调的是消费端结构,每个队列,只被一个消费端监听。

1.A 结构



生产者: 发送消息到交换机,明确表示路由目的地

交换机: 接收消息,根据路由目的地,将消息转发

消费端: 一个消费端监听这个队列,一旦发现消息获取消息执行消费逻辑

1.B 应用场景

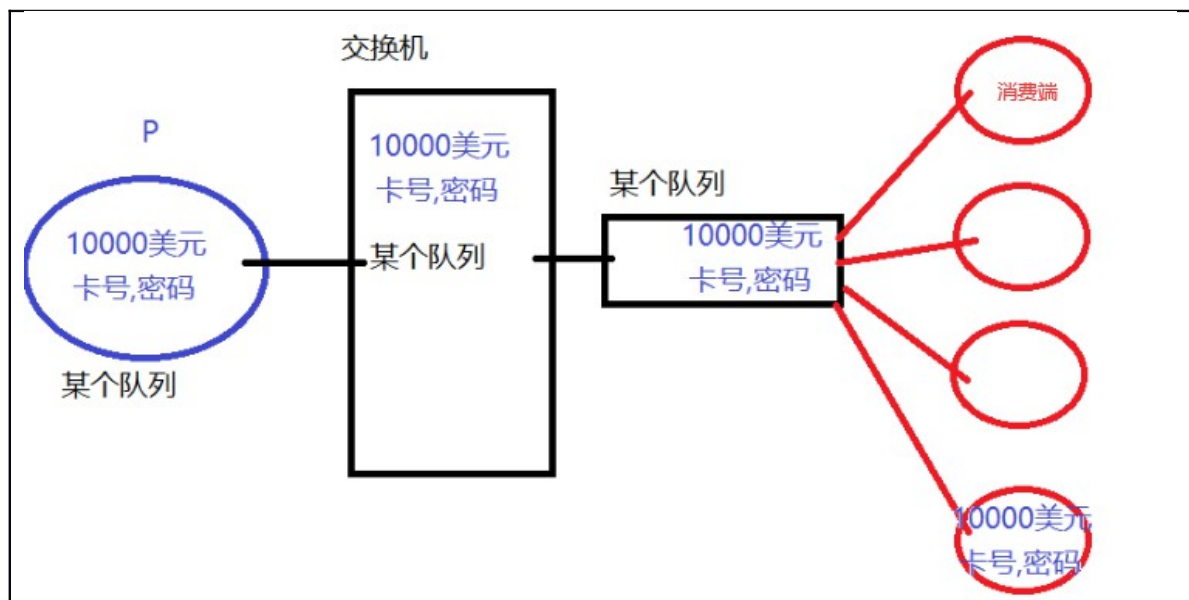
- 发短信;一发,一接

- 。 邮件;一发,一接

2 争抢模式（工作模式）

强调的消费端结构，队列可以由多个消费端同时监听形成争抢。

2.A 结构



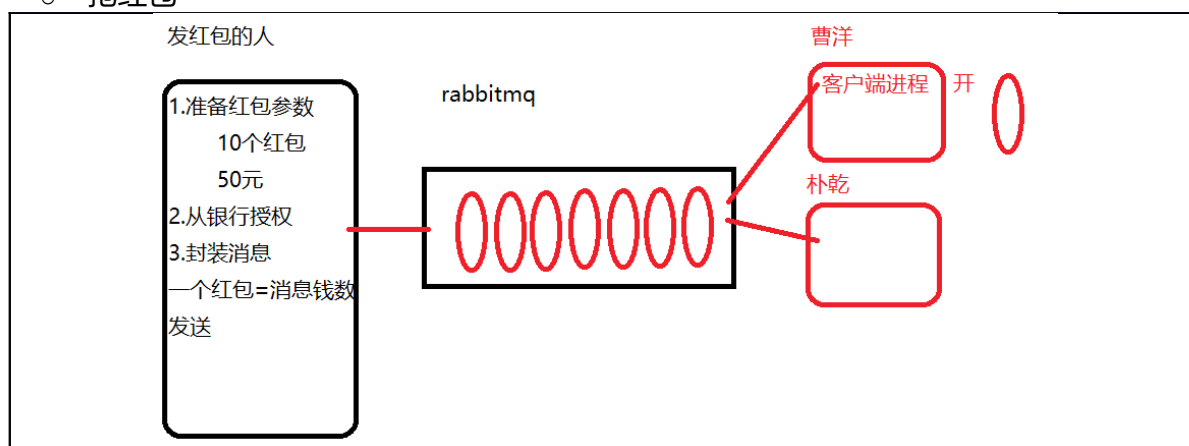
生产者：发送消息到交换机

交换机：根据目的地，转发消息到队列

消费端：多个消费端同时监听一个队列，每一条消息都会在这些消费者中产生争抢的逻辑。

2.B 应用场景

- 。 抢红包

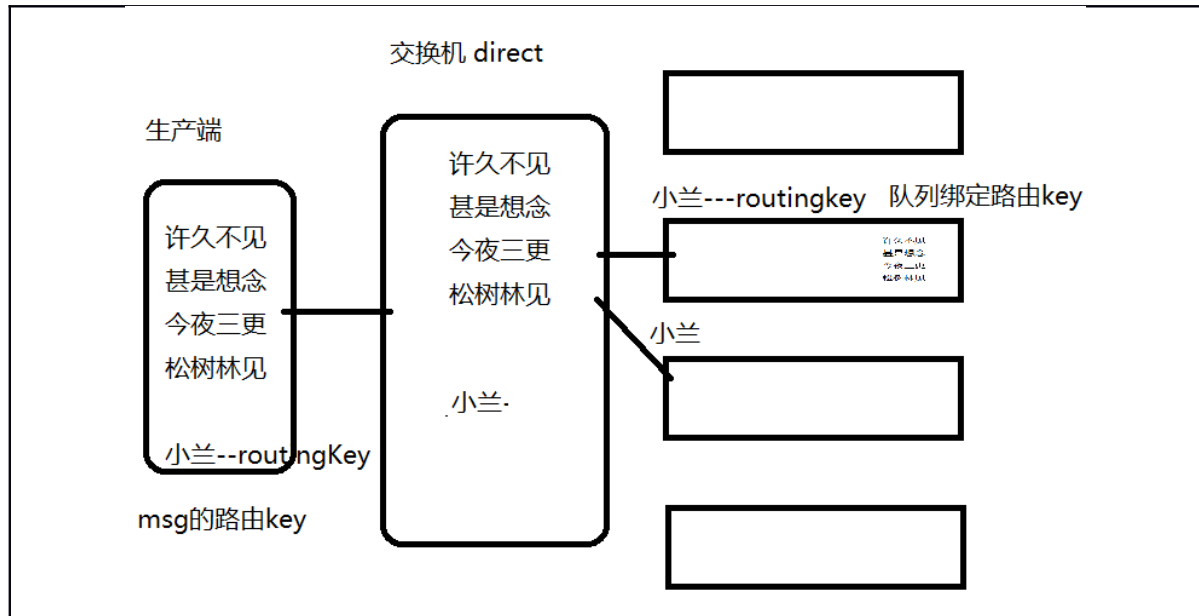


- 。 资源分配

3 路由模式

强调交换机,将会按照消息的路由 key 转发消息.

3.A 结构



生产端：将消息携带路由 key 目的地发送到交换机

交换机：类型 direct,根据消息路由 key 匹配队列绑定路由 key,一旦匹配上将消息转发出去.交换机不会存储消息,直接消息就删除了

生产端：可以一对 1 监听,一对多监听

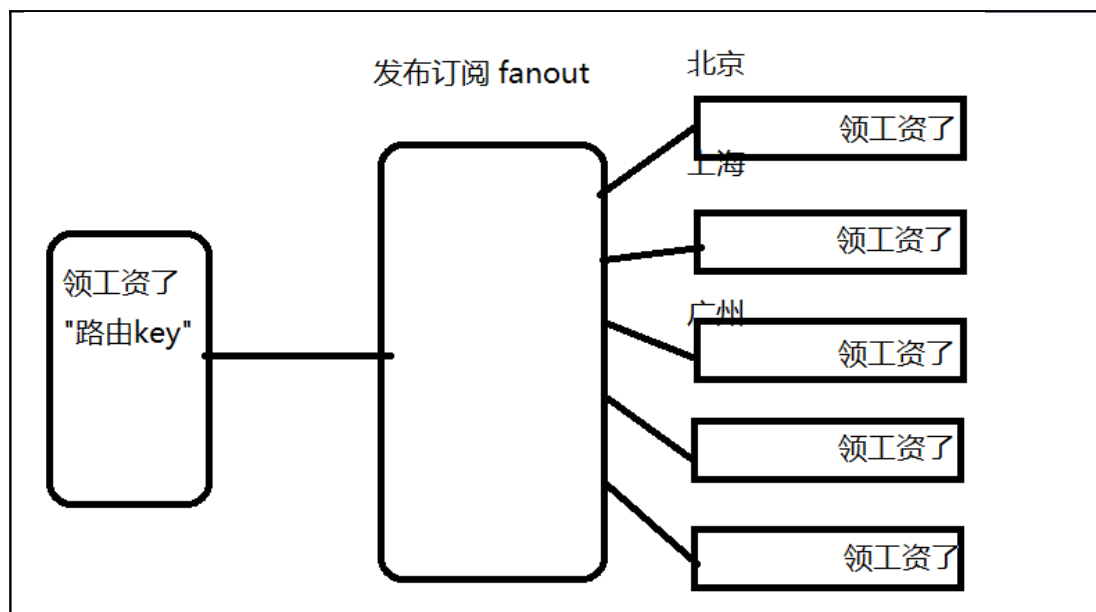
3.B 应用场景

- 短信
可以使用手机号作为路由 key
- 邮件
邮件地址
- 实现错误的自动接收

4 发布订阅模式

强调的是个交换机,会将消息发送给后端所有的队列（绑定的）queue.群发

4.A 结构



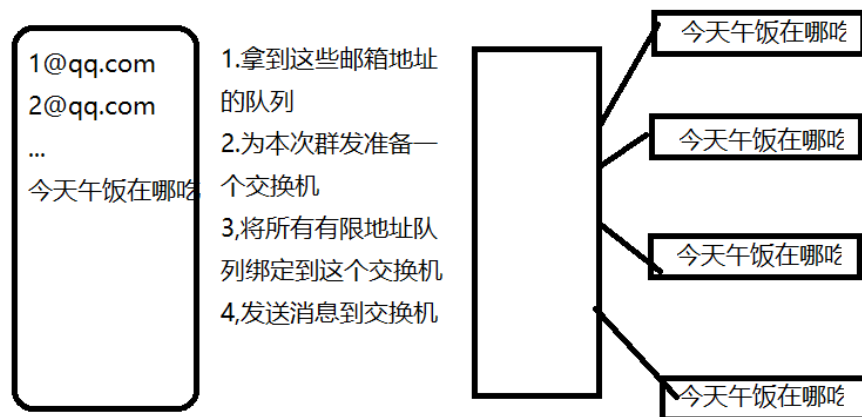
生成端：照常发送消息

交换机：不再判断路由，直接将消息发送给后端所有绑定到他的队列

消费端：简单消费,争抢消费

4.B 应用场景

- 邮件群发
邮件客户端



- 广告

5 主题模式

和路由很类似，区别在于路由模式中，队列绑定交换机时使用的是明确的详细的路由 key，但是主题模式是一个范围的值。

关系到路由 key

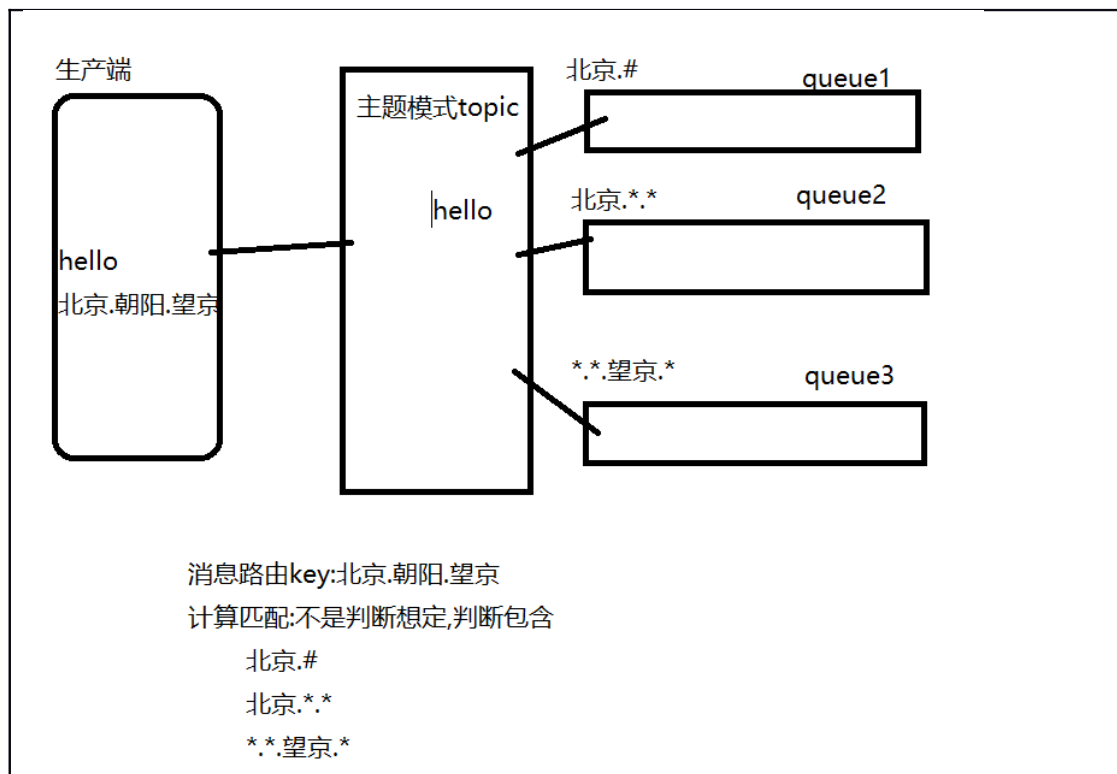
小兰
18877779987

1@qq.com

有的时候路由 key 是一个多级字符串,比如全国地址

中国.北京.朝阳.望京.13 里街道.28 号.11 门.要实现多级传递时,一般可以由主题模式完成

5.A 结构



后端队列绑定交换机使用特殊符号代替路由 key 字符串

#: 任意多级的任意长度字符串

*: 表示一级的任意字符串

#=haha.kaka.wawa.lalala,中国.北京

*=alsdjflasjdf,中国

北京.#: 可以匹配路由 key

北京.*: 可以匹配

.望京.:消息路由 key 没有第四级

5.B 应用场景

物流的分拣.

多级传递.

六 JAVA 客户端实现 5 种工作模式

1 添加 rabbitmq 客户端依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2 创建长连接（Connection）短连接（channel）

下面的五种模式，统一使用该对象

```
//创建连接对象
private Channel channel;
@Before
public void init() throws IOException, TimeoutException {

    //1、创建一个 connection 连接工厂
    ConnectionFactory factory = new ConnectionFactory();
    //2、设置属性
    //提供一些连接参数 host port username password
    factory.setUsername("guest");
    factory.setPassword("guest");
    factory.setHost("10.42.175.170");
    factory.setPort(5672);
    //3、拿到一个连接对象
    Connection connection = factory.newConnection();
    //4、获取 channel
    channel = connection.createChannel();
}
```

3 简单模式

3.A 生产者（发送消息）

```
public void producer() throws IOException {
    //声明准备发送的消息
    String msg="hello01";
    /*① String queue:声明的创建的这个队列的名字 如果已经存在相同名字的,则声明无效
    ② boolean durable（持久的）:队列是否持久化
        持久化队列会在重启 rabbitmq、宕机后恢复 rabbitmq 之后恢复这个队列
        ,没有持久化的就没了
    ③ boolean exclusive（独有的）: 是否专属于当前连接.true 表示专属,
        除了创建声明这个队列的连接可以操作这个队列以外,别人不能用
    ④ boolean autoDelete:最后一个 channel 连接完 queue 是否自动删除
    ⑤ Map args:表示队列声明时的各种属性
```

例如:消息存活时间
最大存储的消息个数
...

默认这个队列绑定 (AMQP default) 交换机,

特性:任意当前 rabbitmq 声明的队列 并绑定路由 key 为队列名,这里为 queue01

```
*/  
//声明队列 queue01  
channel.queueDeclare("queue01",false,  
    false,false,null);  
  
/*  
发送消息到指定队列  
①String exchange:交换机名称 ""表示默认发送给(AMQP default)交换机  
②String routingKey:当前消息发送时携带的路由 key  
③BasicProperties props: 表示封装消息对象 message 时的各种属性  
属性的添加,可以丰富消息的信息,使得处理消费逻辑变得灵活  
④byte[] body:消息体的二进制数据,java 所有对象都可以成为消息体.  
由于这种消息数据占用网络带宽传输,存储在 queue 一段时间,  
消息的封装 遵循精简准确  
*/  
channel.basicPublish("", "queue01", null, msg.getBytes());  
}
```

```
channel.queueDeclare("queue01",false, false,false,null)
```

- ① String queue:声明的创建的这个队列的名字 如果已经存在相同名字的,则声明无效
- ② boolean durable (持久的): 队列是否持久化
持久化队列会在重启 rabbitmq、宕机后恢复 rabbitmq 之后恢复这个队列
没有持久化的就没了
- ③ boolean exclusive (独有的): 是否专属于当前连接.true 表示专属,
除了创建声明这个队列的连接可以操作这个队列以外,别人不能用
- ④ boolean autoDelete: 最后一个 channel 连接完 queue 是否自动删除
- ⑤ Map args: 表示队列声明时的各种属性
例如: 消息存活时间
最大存储的消息个数
...

```
channel.basicPublish("", "queue01", null, msg.getBytes());
```

发送消息到指定队列

- ①String exchange: 交换机名称 ""表示默认发送给 (AMQP default) 交换机
- ②String routingKey: 当前消息发送时携带的路由 key
- ③BasicProperties props: 表示封装消息对象 message 时的各种属性
属性的添加,可以丰富消息的信息,使得处理消费逻辑变得灵活
- ④byte[] body: 消息体的二进制数据,java 所有对象都可以成为消息体.
由于这种消息数据占用网络带宽传输,存储在 queue 一段时间,
消息的封装 遵循精简准确

3.B 消费者 (接收处理消息)

```
/**  
 * 消费端逻辑
```

```

*/
@Test
public void consumer01() throws IOException, InterruptedException {
    //1、生成客户端消费对象
    QueueingConsumer consumer=new QueueingConsumer(channel);
    //将 consumer 绑定监听队列
    // channel.basicConsume("queue01",consumer);
    //① 队列名称, ② autoAck 自动确认 false, 消费对象
    channel.basicConsume("queue01",false,consumer);
    //通过连接调用消息消费
    //delivery 可以看出是一个包含消息的对象
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();

    //获取消息
    byte[] body = delivery.getBody();
    //获取消息相关配置
    AMQP.BasicProperties properties = delivery.getProperties();
    /* properties.getAppId();
    properties.getUserId();*/
    String contentEncoding = properties.getContentEncoding();
    String s = new String(body, "utf-8");
    //打印路由 key
    System.out.println(delivery.getEnvelope().getRoutingKey());
    //打印交换机
    System.out.println(delivery.getEnvelope().getExchange());
    System.out.println(s);
    //手动确认 ①消息标记, ②是否确认多条 false
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(),false);
}

```

消费端的确认机制

为什么要有确认机制：消费端拿到消息后,一定需要存在确认机制，否则容易造成消息错误消费,丢失数据。

确认逻辑：当消费者拿到消息，rabbitmq 记录消息状态为 unacked（未确认），直到消费者返回确认，消息才会删除消失。

自动确认：确认机制可以选择自动确认，消息只要到达消费者端，消息无论正确消费与否，都会自动返回一个确认信息。

上面的例子中关闭了自动确认，用的手动确认。

4 争抢模式（工作模式）

4.A 生产者

```

@Test
public void producter() throws IOException {
    //创建一个workqueue 队列测试用
    //默认这个队列绑定(AMQP default)交换机，并同时绑定 routingKey 为当前队列名
    channel.queueDeclare("workqueue",false,false,false,null);
    //准备发送的消息
    //循环发送 100 个
    for (int i = 0; i < 100; i++) {
        String msg="hello:"+i;
        channel.basicPublish("", "workqueue", null,msg.getBytes());
    }
    System.out.println("发送消息成功");
}

```

4.B 消费者（模拟 2 个）

```

/**
 * 消费端逻辑
 */
@Test

```

```

public void consumer01() throws IOException, InterruptedException {
    channel.queueDeclare("workqueue", false, false, false, null);
    //生成客户端消费对象
    QueueingConsumer consumer=new QueueingConsumer(channel);
    //autoAck 自动
    channel.basicConsume("workqueue", true, consumer);
    while(true){
        //通过连接调用消息消费
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();
        //deliver 可以看出是一个包含消息的对象
        byte[] body = delivery.getBody();
        String s = new String(body, "utf-8");
        System.out.println("消费者 01 接收到消息: "+s);
    }
}

@Test
public void consumer02() throws IOException, InterruptedException {
    channel.queueDeclare("workqueue", false, false, false, null);
    //生成客户端消费对象
    QueueingConsumer consumer=new QueueingConsumer(channel);
    //autoAck 自动
    channel.basicConsume("workqueue", true, consumer);
    while(true){

        //通过连接调用消息消费
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();
        //deliver 可以看出是一个包含消息的对象
        byte[] body = delivery.getBody();
        String s = new String(body, "utf-8");
        System.out.println("消费者 02 接收到消息: "+s);
    }
}
}

```

4.C 争抢结果

WorkMode.consumer01 x

Tests passed: 0 of 1 test

消费者01接收到消息: hello:80

消费者01接收到消息: hello:82

消费者01接收到消息: hello:84

消费者01接收到消息: hello:86

消费者01接收到消息: hello:88

消费者01接收到消息: hello:90

消费者01接收到消息: hello:92

消费者01接收到消息: hello:94

消费者01接收到消息: hello:96

消费者01接收到消息: hello:98

WorkMode.consumer02 x

Tests passed: 0 of 1 test

消费者02接收到消息: hello:81

消费者02接收到消息: hello:83

消费者02接收到消息: hello:85

消费者02接收到消息: hello:87

消费者02接收到消息: hello:89

消费者02接收到消息: hello:91

消费者02接收到消息: hello:93

消费者02接收到消息: hello:95

消费者02接收到消息: hello:97

消费者02接收到消息: hello:99

5 路由模式

除了交换机类型 (**direct**) ,绑定 routingKey 不同,其他结构和发布订阅一致

5.A 先创建队列和交换机用于测试

```
// 声明所有组件 一个交换机，一个队列和绑定关系
@Test
public void bind() throws IOException {
    //准备一批静态常量
    String type="direct";//自定义交换机类型
    String exName=type+"EX";
    //准备若干个队列
    String q1=type+"q01";
    String q2=type+"q02";
    //声明队列
    channel.queueDeclare(q1,false,false,false,null);
    channel.queueDeclare(q2,false,false,false,null);
    //声明交换机
    channel.exchangeDeclare(exName,type);
    //声明绑定关系
    //Q1 绑定 exname 使用路由 key 上海
    channel.queueBind(q1,exName,"上海");
    channel.queueBind(q2,exName,"北京");
}
```

5.B 生产者

发送消息时指定交换机和路由 key，交换机将发送到对应的队列

```
@Test
public void producter() throws IOException {
    //准备发送的消息
    //交换机 exName 和路由 key 北京绑定的队列是 q2
    //该消息将发给 Q2
    channel.basicPublish(exName,"北京",null,"hello".getBytes());
}
```

5.C 消费者

路由模式强调交换机，这里消费者和其它模式一样

6 发布订阅模式

强调的是交换机，会将消息发送给后端所有的队列 queue.群发

发布订阅的结构是自定义一个 **fanout** 类型的交换机，执行发布订阅，多个队列绑定这个交换机
测试时无论 routingKey 为多少都会发给绑定的所有队列，
绑定的不会发

6.A 创建订阅模式类型的交换机

```
//声明所有组件 一个交换机，一个队列和绑定关系
@Test
public void bind() throws IOException {
    String type="fanout";//自定义交换机类型
```

```

String exName=type+"EX";
//准备若干个队列
String q1=type+"q01";
String q2=type+"q02";

//声明队列
channel.queueDeclare(q1,false,false,false,null);
channel.queueDeclare(q2,false,false,false,null);

//声明交换机 类型为 fanout
channel.exchangeDeclare(exName,type);

//声明绑定关系
//Q1 绑定 exname 使用路由 key 上海
channel.queueBind(q1,exName,"上海");
channel.queueBind(q2,exName,"北京");
}

```

6.B 生产者

```

@Test
public void producer() throws IOException {
    //准备发送的消息
    //生产者只关心,消息发送到的交换机是谁,
    // routingKey 对我发送消息没关系,因为这个模式会发送给所有队列
    channel.basicPublish("fanoutEX","asdasd",null,"hello".getBytes());
}

```

6.C 消费者

7 主题模式

和路由很类似，区别在于路由模式中，队列绑定交换机时使用的是明确的详细的路由 key，但是主题模式是一个范围的值。

主题模式交换机类型为 **topic**

7.A 创建交换机绑定队列

```

@Test
public void bind() throws IOException {
    //准备一批静态常量
    String type="topic";//自定义交换机类型
    String exName=type+"EX";

    //准备若干个队列
    String q1=type+"q01";
    String q2=type+"q02";

    //声明队列
    channel.queueDeclare(q1,false,false,false,null);
    channel.queueDeclare(q2,false,false,false,null);

    //声明交换机
    channel.exchangeDeclare(exName,type);

    //声明绑定关系
    //Q1 绑定 exname 使用路由 key 上海
    channel.queueBind(q1,exName,"中国.#");
    channel.queueBind(q1,exName,"中国.天津.*");
    channel.queueBind(q2,exName,"*.北京.*");
}

```


交换机绑定到了 q1, 路由 key 为中国.#, #表示匹配多级任意字符串, 发送消息时路由 key 指定为 “中国.重庆.江北” 等等...也能发送消息到该队列

交换机绑定到了 q2, 路由 key 为 “ *.北京.* ”, *表示匹配一级任意字符串, 发送消息时路由 key 指定为 “中国.北京.江北” 、 “上海.北京.江北” 等等...也能发送消息到该队列。

7.B 生产者

```
@Test
public void producter() throws IOException {
    //准备发送的消息
    //不会发送到任何队列
    channel.basicPublish(exName, "北京", null, "hello".getBytes());
    //会匹配 Q1
    channel.basicPublish(exName, "中国.北京.", null, "hello".getBytes());
    //会匹配 q1 和 q2 , 即发送二个消息
    channel.basicPublish(exName, "中国.北京.HH", null, "hello".getBytes());
}
```

7.C 消费者

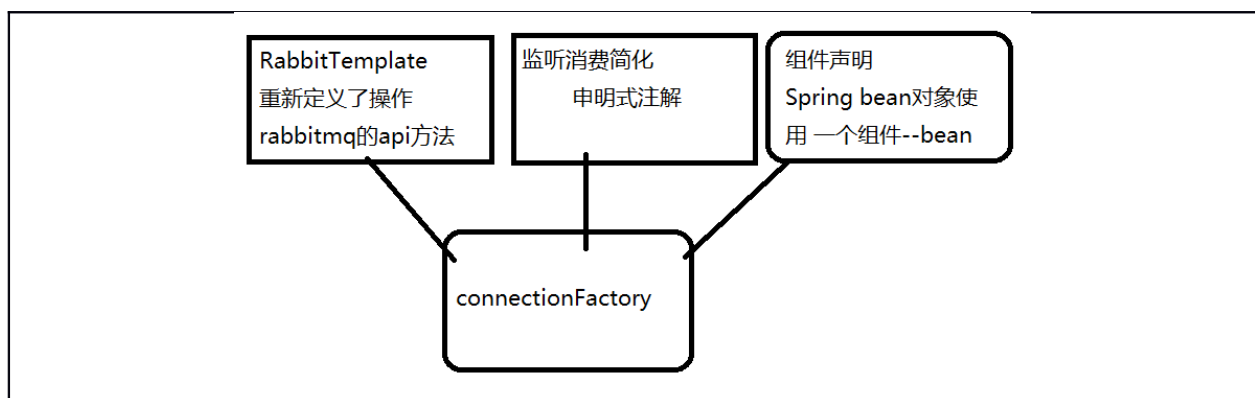
和其它模式一样

七 SpringBoot 整合 RabbitMQ

1 手动配置

思路:通过配置类实现 connectionFactory 封装初始化,由容器管理

2 自动配置



2.A 导入依赖

```
<!--rabbitmq-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2.B 配置属性

```
spring.rabbitmq.host=10.42.175.170
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

2.C 生产者

```
@Service
public class TestService {
    @Autowired
    private RabbitTemplate rabbitTemplate;
    @Test
    public void test(){
        /*
         *send(方法发送)
         * 客户端关心自定义封装消息过程
         */
        //消息配置
        MessageProperties messageProperties = new MessageProperties();
        messageProperties.setPriority(100);
        //封装消息
        Message message = new Message("TEST".getBytes(),messageProperties);
        //发送消息
        rabbitTemplate.send("topicEX","中国.北京",message);
        /*
         *convertAndSend(方法发送)
         */
        rabbitTemplate.convertAndSend("topicEX","中国.北京","你好");
    }
}
```

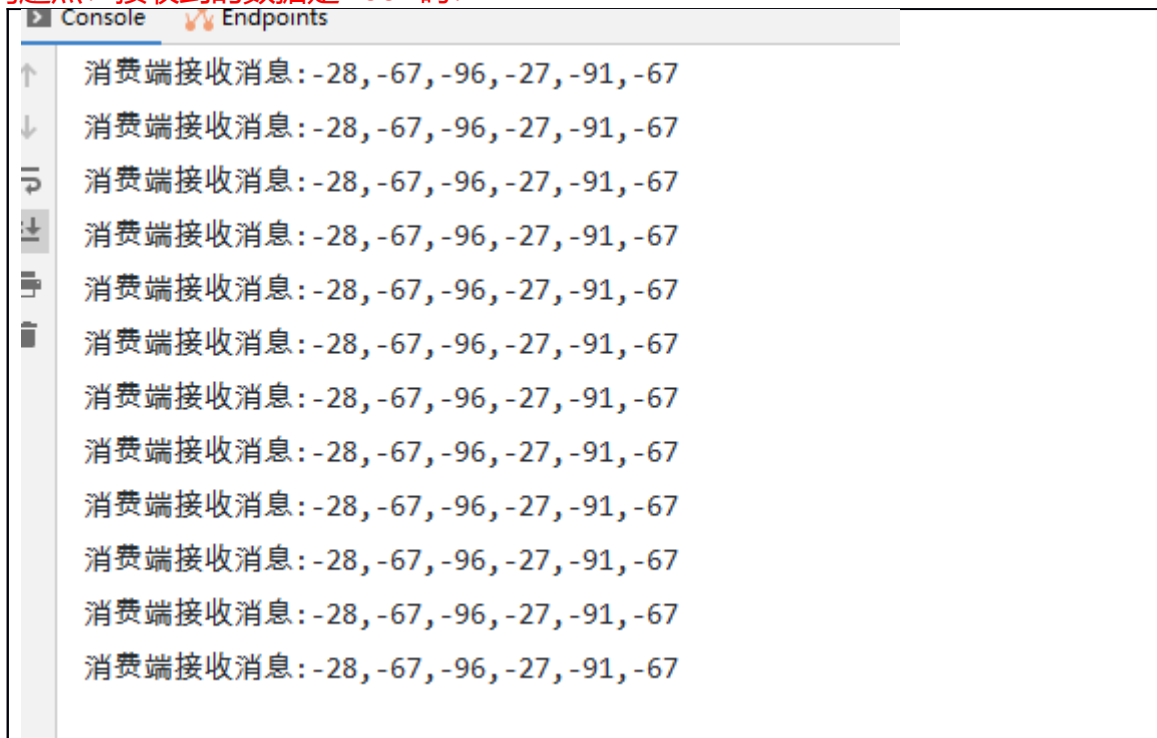
2.D 消费者

关键点: @Component

@RabbitListener(queues = {"topicq01","topicq02"})

```
@Component//Spring
public class Consumer {
    // 监听 队列 topicq01 和 02
    @RabbitListener(queues = {"topicq01","topicq02"})
    public void consume(String msg) throws UnsupportedEncodingException {
        // 当前方法就是消费逻辑调用的消费逻辑，底层链接
        // 拿到消息之后会调用这个吧消息传递给方法参数
        System.out.println("消费端接收消息："+msg);
    }
}
```

问题点：接收到的数据是 ASCII 码？



3 自定义声明组件

- 创建自定义的组件的对象

queue 的组件对应 **Queue** 类

exchange 根据不同类型对应 **DirectExchange** **FanoutExchange** **TopicExchange** 类

绑定关系对应类型 **Binding** 类

- 通过在配置类中声明这些对象创建组件

spring 容器的内存对象不会立刻到 rabbitmq 创建组件，而是当前程序第一次连接(发送消息，程序监听队列)rabbitmq 时自动利用这些对象创建声明组件。

可以在任何一个配置类中通过 @Bean 创建组件，这里用的 SpringBoot 启动类

```
@SpringBootApplication
@EnableEurekaClient
@MapperScan("cn.shu.seckill.mapper")
public class SeckillStarter {
    public static void main(String[] args) {
        SpringApplication.run(SeckillStarter.class, args);
    }
    //启动类当成声明组件配置类

    @Bean//声明一个队列
    public Queue queue01(){
        //对象包装的属性,会在第一次程序链接 rabbitmq 时
        //调用 queueDeclare
        return new Queue("testqueue", false, false, false, null);
    }

    @Bean//声明交换机
    public DirectExchange ex01(){
        return new DirectExchange("testEX");
    }

    @Bean//绑定关系
    public Binding bind01(){
        return BindingBuilder.bind(queue01()).to(ex01()).with("test");
    }
}
```

--