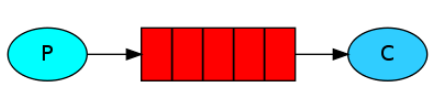
# RabbitMQ的六种模式

简单模式Simple、工作模式Work、发布订阅模式Publish/Subscribe、路由模式Routing、通配符模式Topics、远程调用模式RPC

#### **一 简单模式（Simple / HelloWorld 单生产单消费）**

简单的发送与接收，没有特别的处理。



**RabbitMQ连接（公共的连接方法，其他模式共用此方法）**

public class ConnectionUtil {

public static Connection getConnection() throws IOException, TimeoutException {

//定义连接池

ConnectionFactory connectionFactory = new ConnectionFactory();

//连接地址

connectionFactory.setHost("localhost");

//连接端口

connectionFactory.setPort(5672);

//用户名

connectionFactory.setUsername("han");

//密码

connectionFactory.setPassword("ilovelaohan");

//通过连接工厂获取连接

Connection connection = connectionFactory.newConnection();

//返回连接

return connection;

}

}

**生产者：**

public class Producter {

private final static String QUEUE\_NAME = "hello";

public static void main(String[] args) throws IOException, TimeoutException {

//获取连接

Connection connection = ConnectionUtil.getConnection();

//从连接中声明通道

Channel channel = connection.createChannel();

//队列申明

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

//消息内容

String message = "simple queue hello world !";

//推送发布消息

//basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body)

channel.basicPublish("", QUEUE\_NAME, null, message.getBytes());

//信道关闭

channel.close();

//连接关闭

connection.close();

}

}

**消费者；**

public class Consumer {

private final static String QUEUE\_NAME = "hello\_queue";

public static void main(String[] args) throws IOException, TimeoutException{

//获取连接

Connection connection = ConnectionUtil.getConnection();

//声明信道

Channel channel = connection.createChannel();

//声明队列

channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

//声明消费者

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, byte[] body)

throws IOException {

String message = new String(body,"UTF-8");

System.out.println("customer 消费消息："+message);

}

};

channel.basicConsume(QUEUE\_NAME, true,consumer);

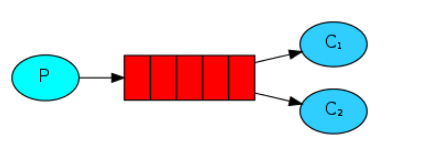
}

}

此代码建立1个生产者2个消费者的话，直接回发生轮询，1:1的概念。而且极有可能讲负载负载在某一台机器上，而另外的电脑处于闲置状态。所以必须使用ack和qos设置。

#### 二 **工作模式（**[Work queues](https://www.rabbitmq.com/tutorials/tutorial-two-python.html)**单发送多接收）**

一个生产者端，多个消费者端。示例中为了保证消息发送的可靠性，不丢失消息，使消息持久化了。同时为了防止接收端在处理消息时down掉，只有在消息处理完成后才发送消息确认。



WorkQueueProducter

public static void main(String[] args) throws IOException, TimeoutException {

String queueName = "work\_queue";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//Queue.DeclareOk queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete,

//Map<String, Object> arguments) throws IOException;

//声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//void basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body) throws IOException;

//发布消息

for (int i = 0; i < 100; i++) {

String message = "iloveyou "+i;

channel.basicPublish("", queueName, MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

}

channel.close();

conn.close();

}

WorkQueueCustomer

public static void main(String[] args) throws IOException, TimeoutException {

String queueName = "work\_queue";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

// Queue.DeclareOk queueDeclare(String queue, boolean durable, boolean

// exclusive, boolean autoDelete,

// Map<String, Object> arguments) throws IOException;

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

//channel.basicQos(1);

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接

}

第二个消费者跟第一个消费者代码完全一样。所以不再赘述。

结果是：同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）。要注意ack和qos的设置。

工作队列

我们将创建一个工作队列，用于在多个工作人员之间分配耗时的任务。

工作队列（又称：任务队列）背后的主要思想是避免立即执行资源密集型任务，并且必须等待它完成。

相反，我们安排任务稍后完成。我们将任务封装 为消息并将其发送到队列。

在后台运行的工作进程将弹出任务并最终执行作业。当您运行许多工作程序时，它们之间将共享任务。

循环调度

使用任务队列的一个优点是能够轻松地并行工作。如果我们正在积压工作积压，我们可以添加更多工人，这样就可以轻松扩展。

首先，让我们尝试同时运行两个worker实例。他们都会从队列中获取消息，但究竟如何呢？让我们来看看。

你需要打开三个控制台。两个将运行工作程序。这些游戏机将成为我们的两个消费者 - C1和C2。

默认情况下，RabbitMQ将按顺序将每条消息发送给下一个消费者。平均而言，每个消费者将获得相同数量的消息。这种分发消息的方式称为循环法。

发送者代码跟以前一样。接受者也一样。但是接受者启动了两个。在这种体系下：

一旦RabbitMQ向消费者发送消息，它立即将其标记为删除。在这种情况下，如果你杀死一个工人，我们将丢失它刚刚处理的消息。我们还将丢失分发给这个特定工作者但尚未处理的所有消息。

但我们不想失去任何任务。如果工人死亡，我们希望将任务交付给另一名工人。

RabbitMQ支持 消息确认。消费者发回ack（nowledgement）告诉RabbitMQ已收到，处理了特定消息，RabbitMQ可以自由删除它。

如果消费者死亡（其通道关闭，连接关闭或TCP连接丢失）而不发送确认，

RabbitMQ将理解消息未完全处理并将重新排队。如果同时有其他在线消费者，

则会迅速将其重新发送给其他消费者。这样你就可以确保没有消息丢失，即使工人偶尔会死亡。

默认情况下，手动消息确认已打开。在前面的示例中，我们通过autoAck = true 标志明确地将它们关闭。一旦我们完成任务，就应该将此标志设置为false并从工作人员发送正确的确认。

channel.basicQos（1）; //一次只接受一条未包装的消息

boolean autoAck = false ;

channel.basicConsume（TASK\_QUEUE\_NAME，autoAck，deliverCallback，consumerTag - > {}）;

确定即使您在处理消息时使用CTRL + C杀死一名工作人员，也不会丢失任何内容。工人死后不久，所有未经确认的消息将被重新传递。

\*\*\*\*\*但是未经确认的消息，当客户端退出时，会被重新传递。这样消息会重新入池。

内存占用会越来愈多。所以必须手工进行应答。channel.basicAck(envelope.getDeliveryTag(), false);

经过手工确认的消息，会进行销毁，未进行手工确认的消息，会重新入池。

确保消息不会丢失。

================

消息持久性

boolean durable = true ;

channel.queueDeclare（“hello”，durable，false，false，null）;

完成队列持久性。

消息持久性

channel.basicPublish（“”，“task\_queue”，

MessageProperties.PERSISTENT\_TEXT\_PLAIN，

message.getBytes（））;

但是：

RabbitMQ接受消息，但未保存消息的时候，仍然有一段时间，可能只是放入缓存

而未存入磁盘，可能仍然未完成持久性。

对于普通任务队列已经足够，如果需要更强的保证，可以使用【发布者确认】。

==============

公平派遣

有时候会发生忙的忙死，闲的闲死。但RabbitMQ对此一无所知，仍然会均匀地发送消息。

原因是RabbitMQ只是在消息进入队列时调度消息，不会查看消费者未确认消息数量。只是盲目的按照次序发送消息。

解决方案：

使用 basicQos方法和 prefetchCount = 1设置

basicQos方法:

指该消费者在接收到队列里的消息但没有返回确认结果之前,它不会将新的消息分发给它。

1:对比测试：两个消费者都订阅同一队列，no\_ack均设置为false即开启acknowledge机制，

且均未设置prefetch\_count，向队列发布5条消息

结果：不管消息是否被ack，rabbitmq会轮流向两个消费者投递消息，

第一个消费者收到"1"，"3"，"5"三条消息， 第二个消费者收到"2"，"4"两条消息。

2）prefetch\_count设置测试：两个消费者都订阅同一队列，

开启acknowledge机制，第一个消费者prefetch\_count设置为1，

另一个消费者未设置prefetch\_count，同样向队列发布5条消息

结果：rabbitmq向第一个消费者投递了一条消息后，

消费者未对该消息进行ack，rabbitmq不会再向该消费者投递消息，剩下的四条消息均投递给了第二个消费者

命令行批量删除

首先定位到 rabbitMQ 安装目录的sbin 目录下。打开cmd窗口。

关闭应用的命令为： rabbitmqctl stop\_app

清除的命令为： rabbitmqctl reset

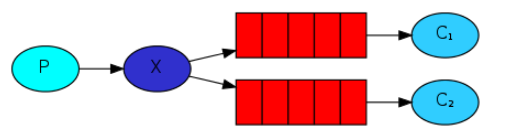
重新启动命令为： rabbitmqctl start\_app

ps

查看所有队列命令： rabbitmqctl list\_queues

#### **三 发布、订阅模式(Publish/Subscribe)**

使用场景：发布、订阅模式，生产者端发送消息，多个消费者同时接收所有的消息



WorkQueueProducter

public static void main(String[] args) throws IOException, TimeoutException {

final String EXECHANGENAME = "publicsubscrible";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为fanout

channel.exchangeDeclare(EXECHANGENAME, "fanout");

for (int i = 0; i < 100; i++) {

String message = "i love laohan "+i;

//生产者端发布消息到交换机，使用“fanout”方式发送，

//即广播消息，不需要使用queue，发送端不需要关心谁接收。

channel.basicPublish(EXECHANGENAME, "", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

}

System.out.println("发布者订阅模式发布消息成功");

channel.close();

conn.close();

}

WorkQueueCustomer

public static void main(String[] args) throws IOException, TimeoutException {

final String EXECHANGENAME = "publicsubscrible";

final String queueName = "publicsubscrible\_queue";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为fanout

channel.exchangeDeclare(EXECHANGENAME, "fanout");

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//绑定队列到交换机

channel.queueBind(queueName, EXECHANGENAME, "");

//注意binding queue的时候，channel.queueBind()的第三个参数Routing key为空，

//即所有的消息都接收。如果这个值不为空，在exchange type为“fanout”方式下该值被忽略！

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

channel.basicQos(1);

//申明消费者

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

}

};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接

}

第二个消费者代码跟第一个消费者完全一样，但是注意使用第二个队列。则每个消息都会被两个消费者都能获取到。

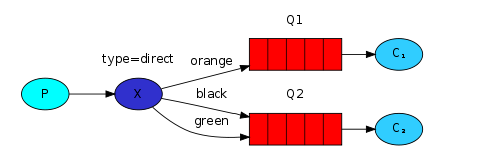
消费者端：

1、声明和生产者端一样的交换机。

2、注意binding queue的时候，channel.queueBind()的第三个参数Routing key为空，即所有的消息都接收。如果这个值不为空，在exchange type为“fanout”方式下该值被忽略！

##### **四：路由模式（Routing）**

生产者按routing key发送消息，不同的消费者端按不同的routing key接收消息。



channel.queueBind（queueName，EXCHANGE\_NAME，“”）;

绑定是交换和队列之间的关系。这可以简单地理解为：队列对来自此交换的消息感兴趣。

绑定可以采用额外的routingKey参数。为了避免与basic\_publish参数混淆，我们将其称为 绑定密钥。这就是我们如何使用键创建绑定：：

channel.queueBind（queueName，EXCHANGE\_NAME，“black”）;

绑定密钥的含义取决于交换类型。我们之前使用的 fanout 交换只是忽略了它的价值。

## 直接交换

发布者订阅模式向所有消费者广播所有消息。我们希望扩展它以允许根据消息的严重性过滤消息。例如，我们可能需要一个程序将日志消息写入磁盘以仅接收严重错误，而不是在警告或信息日志消息上浪费磁盘空间。

Fanout交换只能进行无意识的广播。

直接交换。直接交换背后的路由算法很简单 - 消息进入队列，其 绑定密钥与消息的路由密钥完全匹配。

此设置中，我们可以看到直接交换X与两个绑定到它的队列。第一个队列绑定橙色绑定，第二个绑定有两个绑定，一个绑定密钥为黑色，另一个绑定为绿色。

在这样的设置中，使用路由密钥orange发布到交换机的消息 将被路由到队列Q1。路由键为黑色 或绿色的消息将转到Q2。所有其他消息将被丢弃。

一个生产者，两个消费者：

WorkQueueProducter

final String EXECHANGENAME = "publicsubscrible4";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为direct

channel.exchangeDeclare(EXECHANGENAME, "direct");

String message = "red";

channel.basicPublish(EXECHANGENAME, "red", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "green";

channel.basicPublish(EXECHANGENAME, "green", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "black";

channel.basicPublish(EXECHANGENAME, "black", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "white";

channel.basicPublish(EXECHANGENAME, "white", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

channel.close();

conn.close();

WorkQueueCustomer

final String EXECHANGENAME = "publicsubscrible4";

final String queueName = "publicsubscrible\_queue41";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为direct

channel.exchangeDeclare(EXECHANGENAME, "direct");

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//绑定队列到交换机 指定路由routingKey

//结束路由routingKey为green和black的消息

channel.queueBind(queueName, EXECHANGENAME, "green");

channel.queueBind(queueName, EXECHANGENAME, "black");

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

channel.basicQos(1);

//申明消费者

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

}

};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接

WorkQueueCustomer2

final String EXECHANGENAME = "publicsubscrible4";

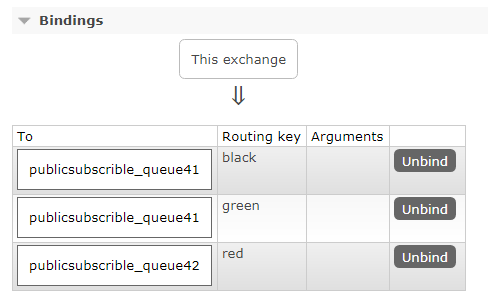
final String queueName = "publicsubscrible\_queue42";

//绑定队列到交换机 指定路由routingKey

//结束路由routingKey为green和black的消息

channel.queueBind(queueName, EXECHANGENAME, "red");

则路由模式根据routing key来判断，接受什么数据。跟消费者端和模式3（发布订阅模式）的区别：在绑定queue和exchange的时候使用了路由routing key，即从该exchange上只接收routing key指定的消息。其他没有建立队列的消息被丢弃掉。



#### 五 **通配符（或主题）模式（Topics ，按topic发送接收)**

将虚拟广播的fanout交换，更换为带路由的 direct的交换方式，可以选择性的接受消息。但它仍然具有局限性，不能基于多个标准进行路由。

通过更为复杂的 topic 交换，可以检测消息的来源。

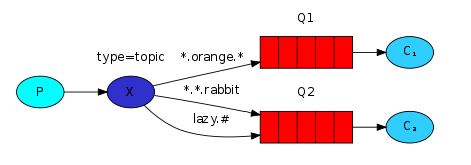
生产者端不只按固定的routing key发送消息，而是按字符串“匹配”发送，消费者端同样如此。

与之前的路由模式相比，它将信息的传输类型的key更加细化，以“key1.key2.keyN…”的模式来指定信息传输的key的大类型和大类型下面的小类型，让消费者端可以更加精细的确认自己想要获取的信息类型。而在消费者端，不用精确的指定具体到哪一个大类型下的小类型的key，而是可以使用类似正则表达式(但与正则表达式规则完全不同)的通配符在指定一定范围或符合某一个字符串匹配规则的key，来获取想要的信息。“通配符交换机”（Topic Exchange）将路由键和某模式进行匹配。此时队列需要绑定在一个模式上。符号“#”匹配一个或多个词，符号“\*”仅匹配一个词。

发送到topic交换的消息不能具有任意的 routing\_key - 它必须是由点分隔的单词列表。单词可以是任何内容，但通常它们指定与消息相关的一些功能。一些有效的路由键示例：“ stock.usd.nyse ”，“ nyse.vmw”，“ quick.orange.rabbit ”。路由密钥中可以包含任意数量的单词，最多可达255个字节。

绑定密钥也必须采用相同的形式。topic交换背后的逻辑 类似于direct交换- 使用特定路由密钥发送的消息将被传递到与匹配的绑定密钥绑定的所有队列。但是绑定键有两个重要的特殊情况

* \*（星号）可以替代一个单词。
* ＃（hash）可以替换零个或多个单词



WorkQueueProducter

final String EXECHANGENAME = "topic\_exchange";

//得到连接

Connection conn = ConnectionUtil.getConnection();

//创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为topic

channel.exchangeDeclare(EXECHANGENAME, "topic");

String message = "send jd order message";

channel.basicPublish(EXECHANGENAME, "order.jd", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "send tm order message";

channel.basicPublish(EXECHANGENAME, "order.tm", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "send jd goods message";

channel.basicPublish(EXECHANGENAME, "goods.jd", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

message = "send tm goods message";

channel.basicPublish(EXECHANGENAME, "goods.tm", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

channel.close();

conn.close();

WorkQueueCustomer

final String EXECHANGENAME = "topic\_exchange";

final String queueName = "topic\_queue1";

// 得到连接

Connection conn = ConnectionUtil.getConnection();

// 创建频道

Channel channel = conn.createChannel();

//声明交换机Exchange类型为topic

channel.exchangeDeclare(EXECHANGENAME, "topic");

// 声明消息队列

boolean durable = true;

channel.queueDeclare(queueName, durable, false, false, null);

//绑定队列到交换机 指定路由routingKey

//结束路由routingKey为green和black的消息

channel.queueBind(queueName, EXECHANGENAME, "order.\*");

//同一时刻服务器只发送1条消息给消费者（能者多劳，消费消息快的，会消费更多的消息）

//保证在接收端一个消息没有处理完时不会接收另一个消息，即消费者端发送了ack后才会接收下一个消息。

//在这种情况下生产者端会尝试把消息发送给下一个空闲的消费者。

channel.basicQos(1);

//申明消费者

Consumer consumer = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body,"UTF-8");

System.out.println("消费======================1消息 "+message);

//手动返回消息

channel.basicAck(envelope.getDeliveryTag(), false);

}

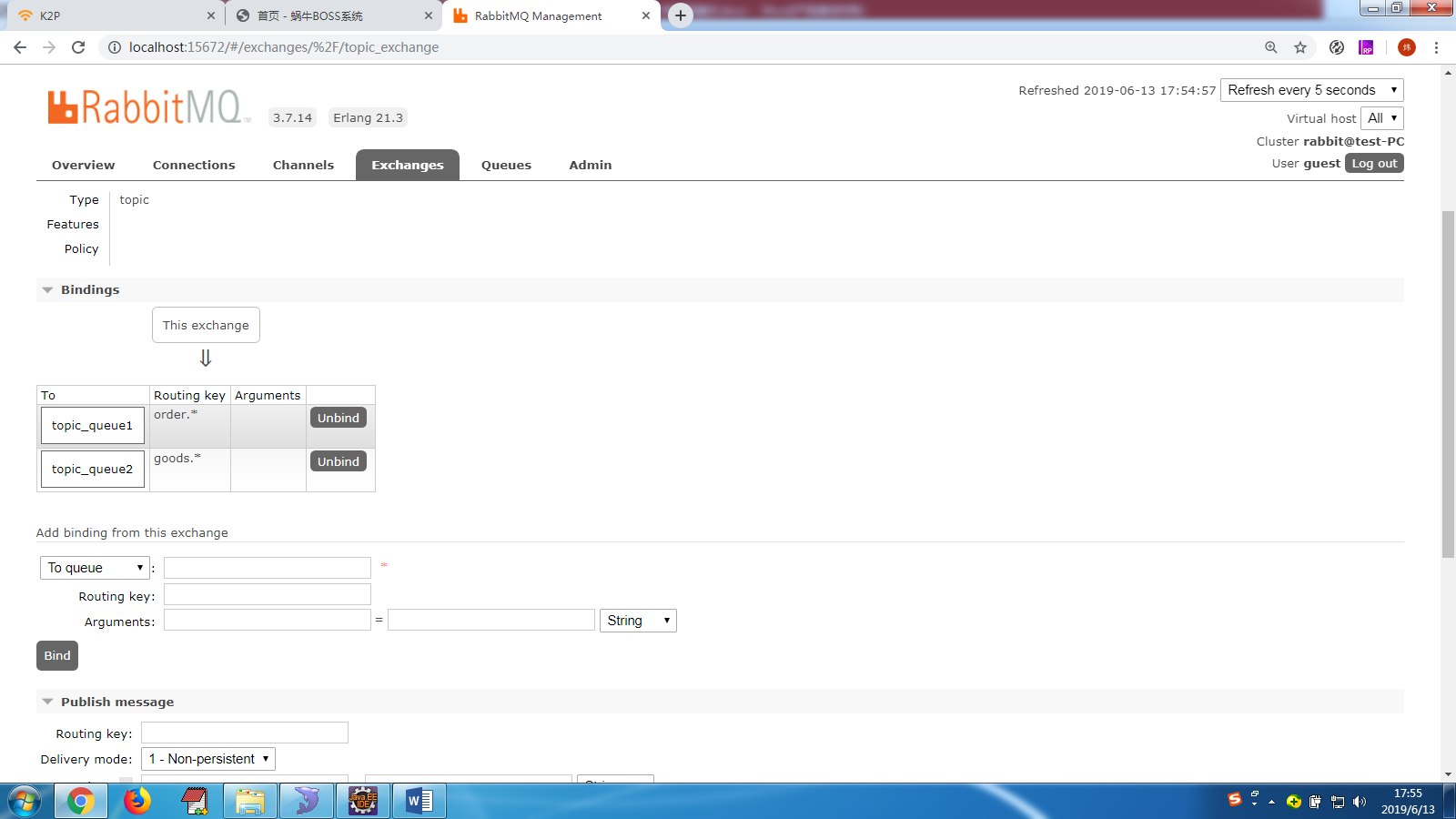
};

//定义的消费者监听队列 autoAck：true自动返回结果，false手动返回

boolean autoAck = false;

channel.basicConsume(queueName,autoAck,consumer);

//注意，此处不要管理频道和连接



生产者和模式4（路由模式）的区别：

1、交换机exchange的type为topic

2、发送消息的routing key不是固定的单词，而是匹配字符串，如"order.#"，\*匹配一个单词，#匹配0个或多个单词。因此如“order.#”能够匹配到“order.jd.buy”，但是“order.\* ”只会匹配到“order.jd”

## 六：远程过程调用（RPC）

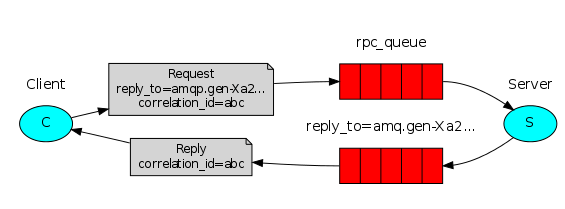
#### 有关RPC的说明

尽管RPC在计算中是一种非常常见的模式，但它经常受到批评。当程序员不知道函数调用是本地的还是慢的RPC时，会出现问题。这样的混淆导致系统不可预测，并增加了调试的不必要的复杂性。错误使用RPC可以导致不可维护的意大利面条代码，而不是简化软件。

考虑到这一点，请考虑以下建议：

* 确保明显哪个函数调用是本地的，哪个是远程的。
* 记录您的系统。使组件之间的依赖关系变得清晰。
* 处理错误案例。当RPC服务器长时间停机时，客户端应该如何反应？

如有疑问，请避免使用RPC。如果可以，您应该使用异步管道 - 而不是类似RPC的阻塞，将结果异步推送到下一个计算阶段。



我们的RPC将这样工作：

* 对于RPC请求，客户端发送带有两个属性的消息： replyTo（设置为仅为请求创建的匿名独占队列）和correlationId（设置为每个请求的唯一值）。
* 请求被发送到rpc\_queue队列。
* RPC worker（aka：server）正在等待该队列上的请求。当出现请求时，它会执行该作业，并使用来自replyTo字段的队列将带有结果的消息发送回客户端。
* 客户端等待回复队列上的数据。出现消息时，它会检查correlationId属性。如果它与请求中的值匹配，则将响应返回给应用程序。

发出客户请求：

RPCClient fibonacciRpc = new RPCClient（）;

System.out.println（“[x] Requesting fib（30）”）;

字符串响应= fibonacciRpc.call（“30”）;

System.out.println（“[。] Got'” + response + “'”）;

fibonacciRpc.close（）;

### 任务分发机制

##### 1Round-robin dispathching循环分发

RabbbitMQ的分发机制非常适合扩展,而且它是专门为并发程序设计的,如果现在load加重,那么只需要创建更多的Consumer来进行任务处理。

##### 2Message acknowledgment消息确认

为了保证数据不被丢失,RabbitMQ支持消息确认机制,为了保证数据能被正确处理而不仅仅是被Consumer收到,那么我们不能采用no-ack，而应该是在处理完数据之后发送ack.

在处理完数据之后发送ack,就是告诉RabbitMQ数据已经被接收,处理完成,RabbitMQ可以安全的删除它了.

如果Consumer退出了但是没有发送ack,那么RabbitMQ就会把这个Message发送到下一个Consumer，这样就保证在Consumer异常退出情况下数据也不会丢失.

RabbitMQ它没有用到超时机制.RabbitMQ仅仅通过Consumer的连接中断来确认该Message并没有正确处理，也就是说RabbitMQ给了Consumer足够长的时间做数据处理。

如果忘记ack,那么当Consumer退出时,Mesage会重新分发,然后RabbitMQ会占用越来越多的内存.

##### 3Message durability消息持久化

要持久化队列queue的持久化需要在声明时指定durable=True;

**这里要注意,队列的名字一定要是Broker中不存在的,不然不能改变此队列的任何属性.**

队列和交换机有一个创建时候指定的标志durable,durable的唯一含义就是具有这个标志的队列和交换机会在重启之后重新建立,它不表示说在队列中的消息会在重启后恢复

消息持久化包括3部分

1. exchange持久化,在声明时指定durable => true

hannel.ExchangeDeclare(ExchangeName, "direct", durable: true, autoDelete: false, arguments: null);//声明消息队列，且为可持久化的

2.queue持久化,在声明时指定durable => true

channel.QueueDeclare(QueueName, durable: true, exclusive: false, autoDelete: false, arguments: null);//声明消息队列，且为可持久化的

3.消息持久化,在投递时指定delivery\_mode => 2(1是非持久化).

channel.basicPublish("", queueName, MessageProperties.PERSISTENT\_TEXT\_PLAIN, msg.getBytes());

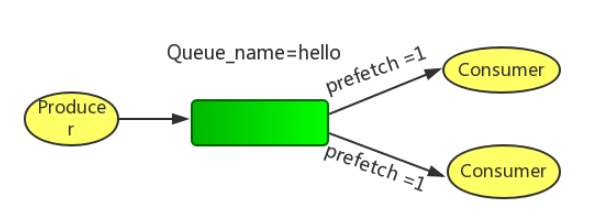
如果exchange和queue都是持久化的,那么它们之间的binding也是持久化的,如果exchange和queue两者之间有一个持久化，一个非持久化,则不允许建立绑定.

**注意**：一旦创建了队列和交换机,就不能修改其标志了,例如,创建了一个non-durable的队列,然后想把它改变成durable的,唯一的办法就是删除这个队列然后重现创建。

##### 4Fair dispatch公平分发

分发机制不是那么优雅,默认状态下,RabbitMQ将第n个Message分发给第n个Consumer。n是取余后的,它不管Consumer是否还有unacked Message，只是按照这个默认的机制进行分发.

那么如果有个Consumer工作比较重,那么就会导致有的Consumer基本没事可做,有的Consumer却毫无休息的机会,那么,Rabbit是如何处理这种问题呢?



通过basic.qos方法设置prefetch\_count=1，这样RabbitMQ就会使得每个Consumer在同一个时间点最多处理一个Message，换句话说,在接收到该Consumer的ack前,它不会将新的Message分发给它

channel.basic\_qos(prefetch\_count=1)

**注意，这种方法可能会导致queue满。当然，这种情况下你可能需要添加更多的Consumer，或者创建更多的virtualHost来细化你的设计。**

##### 5.分发到多个Consumer

##### 5.1Exchange

先来回顾以下交换机路由的几种类型:   
Direct Exchange:直接匹配,通过Exchange名称+RountingKey来发送与接收消息.   
Fanout Exchange:广播订阅,向所有的消费者发布消息,但是只有消费者将队列绑定到该路由器才能收到消息,忽略Routing Key.   
Topic Exchange：主题匹配订阅,这里的主题指的是RoutingKey,RoutingKey可以采用通配符,如:\*或#，RoutingKey命名采用.来分隔多个词,只有消息这将队列绑定到该路由器且指定RoutingKey符合匹配规则时才能收到消息;   
Headers Exchange:消息头订阅,消息发布前,为消息定义一个或多个键值对的消息头,然后消费者接收消息同时需要定义类似的键值对请求头:(如:x-mactch=all或者x\_match=any)，只有请求头与消息头匹配,才能接收消息,忽略RoutingKey.   
默认的exchange:如果用空字符串去声明一个exchange，那么系统就会使用”amq.direct”这个exchange，我们创建一个queue时,默认的都会有一个和新建queue同名的routingKey绑定到这个默认的exchange上去

channel.BasicPublish("", "TaskQueue", properties, bytes)*;*

因为在第一个参数选择了默认的exchange，而我们申明的队列叫TaskQueue，所以默认的，它在新建一个也叫TaskQueue的routingKey，并绑定在默认的exchange上，导致了我们可以在第二个参数routingKey中写TaskQueue，这样它就会找到定义的同名的queue，并把消息放进去。

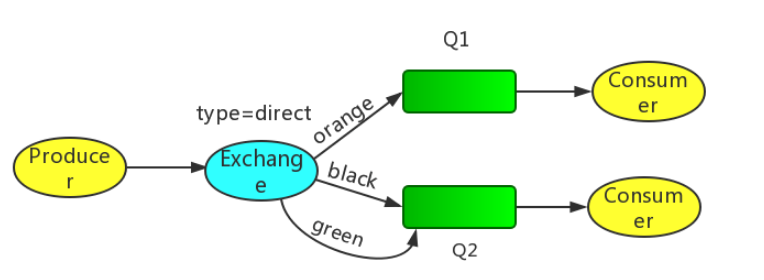
**如果有两个接收程序都是用了同一个的queue和相同的routingKey去绑定direct exchange的话，分发的行为是负载均衡的，也就是说第一个是程序1收到，第二个是程序2收到，以此类推。**   
**如果有两个接收程序用了各自的queue，但使用相同的routingKey去绑定direct exchange的话，分发的行为是复制的，也就是说每个程序都会收到这个消息的副本。行为相当于fanout类型的exchange。**

##### 5.2 Bindings 绑定

绑定其实就是关联了exchange和queue，或者这么说:queue对exchange的内容感兴趣,exchange要把它的Message deliver到queue。

##### 5.3Direct exchange

Driect exchange的路由算法非常简单:通过bindingkey的完全匹配，可以用下图来说明.

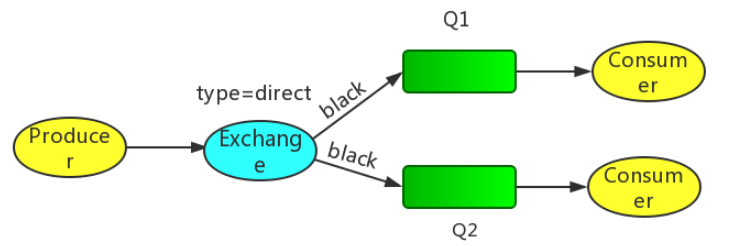


Exchange和两个队列绑定在一起,Q1的bindingkey是orange，Q2的binding key是black和green.

当Producer publish key是orange时,exchange会把它放到Q1上,如果是black或green就会到Q2上,其余的Message被丢弃.

##### 5.4 Multiple bindings

多个queue绑定同一个key也是可以的,对于下图的例子,Q1和Q2都绑定了black,对于routing key是black的Message，会被deliver到Q1和Q2，其余的Message都会被丢弃.



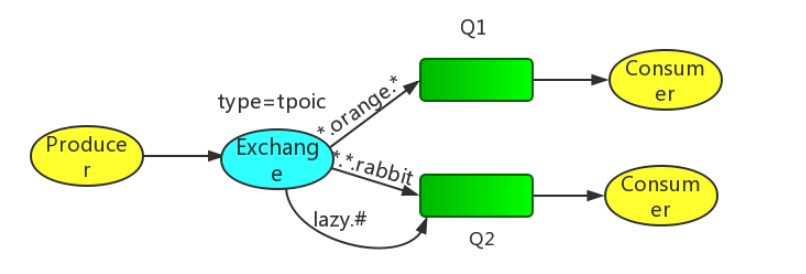
##### 5.5 Topic exchange

对于Message的routing\_key是有限制的，不能使任意的。格式是以点号“.”分割的字符表。比如：”stock.usd.nyse”, “nyse.vmw”, “quick.orange.rabbit”。你可以放任意的key在routing\_key中，当然最长不能超过255 bytes。

对于routing\_key，有两个特殊字符

\*(星号)代表任意一个单词

#(hash)0个或多个单词



Producer发送消息时需要设置routing\_key，routing\_key包含三个单词和连个点号o,第一个key描述了celerity(灵巧),第二个是color(色彩),第三个是物种:

在这里我们创建了两个绑定： Q1 的binding key 是”.orange.“； Q2 是 “..rabbit” 和 “lazy.#”：

Q1感兴趣所有orange颜色的动物

Q2感兴趣所有rabbits和所有的lazy的.

例子:rounting\_key 为 “quick.orange.rabbit”将会发送到Q1和Q2中

rounting\_key 为”lazy.orange.rabbit.hujj.ddd”会被投递到Q2中,#匹配0个或多个单词。

#### 6.消息序列化

RabbitMQ使用ProtoBuf序列化消息,它可作为RabbitMQ的Message的数据格式进行传输,由于是结构化的数据,这样就极大的方便了Consumer的数据高效处理,当然也可以使用XML，与XML相比,ProtoBuf有以下优势:

1.简单

2.size小了3-10倍

3.速度快了20-100倍

4.易于编程

6.减少了语义的歧义.

，ProtoBuf具有速度和空间的优势，使得它现在应用非常广泛

STEP1:simple方式

===============================================================

1:pom.xml

<dependency>

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

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

</dependency>

2:在application.properties添加rabbitmq的相关信息：

spring.application.name=boot-rabbitmq

spring.rabbitmq.host=127.0.0.1

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

spring:

rabbitmq:

host: 127.0.0.1

port: 5672

username: guest

password: guest

3:配置队列：

@Configuration

public class RabbitMqConfig {

@Bean

@ConditionalOnMissingBean(name="queue")//可选

public Queue getQueue() {

return new Queue("laohanqueue");

}

}

4、发送者通过Controller类发送消息：

@RestController

public class SendController {

@Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

String message = "message is "+new Date();

amqpTemplate.convertAndSend("laohanqueue",message);

return message;

}

}

5: 接受消息：

@Component

@RabbitListener(queues="laohanqueue")

public class Reveice {

@RabbitHandler

public void receive(String msg) {

System.out.println("您有新的短消息"+msg);

}

}

注意，发送者和接收者的queue name必须一致，不然不能接收

===============================================================

STEP1.1:simple带ack方式

生产者

yml

server:

port: 9000

spring:

rabbitmq:

addresses: localhost

port: 5672

username: guest

password: guest

配置队列

@Configuration

public class MqConfig {

@Bean

public Queue getQueue() {

return new Queue("directhanqueue");

}

}

发送消息

@RestController

public class SendController {

@Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

amqpTemplate.convertAndSend("directhanqueue2", "今天天气真正好"+i+"!");

return "消息发送成功"+i;

}

}

消费者：

yml

server:

port: 9001

spring:

rabbitmq:

addresses: localhost

port: 5672

username: guest

password: guest

publisher-returns: true

publisher-confirms: true

listener:

simple:

acknowledge-mode: manual

prefetch: 1

direct:

acknowledge-mode: manual

prefetch: 1

@Component

@RabbitListener(queues="directhanqueue2")

public class Receive {

@RabbitHandler

public void rev(String msg,Channel channel,Message message) throws IOException, InterruptedException {

Thread.sleep(10000);

System.out.println("接受到消息1"+msg);

channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

}

}

然后完成一个生产者，2个消费者，则产生对等轮询。并且有消息验证。

===============================================================

STEP5.1:Boot Topic 带exchange ack方式

生产者

server:

port: 9000

spring:

rabbitmq:

addresses: localhost

port: 5672

username: guest

password: guest

配置中心

@Configuration

public class MqConfig {

@Bean

public Queue queue1() {

return new Queue("shanxi.xian");

}

@Bean

public Queue queue2() {

return new Queue("shanxi.hanzhong");

}

@Bean

public TopicExchange createChange() {

return new TopicExchange("bootExchange");

}

@Bean

Binding bandingExchange(Queue queue1,TopicExchange topicExchange) {

return BindingBuilder.bind(queue1).to(topicExchange).with("shanxi.hanzhong");

}

@Bean

Binding bandingExchanges(Queue queue2,TopicExchange topicExchange) {

return BindingBuilder.bind(queue2).to(topicExchange).with("shanxi.#");

}

}

发送数据

@Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

i++;

amqpTemplate.convertAndSend("bootExchange", "shanxi.xian", "西安今天天气真正好"+i+"!");

return "消息发送成功"+i;

}

@RequestMapping("/send2")

public String send2() {

i++;

amqpTemplate.convertAndSend("bootExchange", "shanxi.hanzhong", "汉中今天天气真正好"+i+"!");

return "消息发送成功"+i;

}

消费者

yml

server:

port: 9001

spring:

rabbitmq:

addresses: localhost

port: 5672

username: guest

password: guest

publisher-returns: true

publisher-confirms: true

listener:

simple:

acknowledge-mode: manual

prefetch: 1

direct:

acknowledge-mode: manual

prefetch: 1

消费者1：

@Component

@RabbitListener(queues="shanxi.xian")

public class Receive {

@RabbitHandler

public void rev(String msg,Channel channel,Message message) throws IOException, InterruptedException {

Thread.sleep(5000);

System.out.println("西安接受到消息1"+msg);

channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

}

}

消费者2：

@Component

@RabbitListener(queues="shanxi.hanzhong")

public class Receive2 {

@RabbitHandler

public void rev(String msg,Channel channel,Message message) throws IOException, InterruptedException {

Thread.sleep(5000);

System.out.println("汉中接受到消息2"+msg);

channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

}

}

===============================================================

===============================================================

spring boot 发送mq:

1:pom.xml

<dependency>

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

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

</dependency>

2:在application.properties添加rabbitmq的相关信息：

spring.application.name=boot-rabbitmq

spring.rabbitmq.host=127.0.0.1

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

3:配置队列：

@Configuration

public class RabbitMqConfig {

@Bean

@ConditionalOnMissingBean(name="queue")//可选

public Queue getQueue() {

return new Queue("laohanqueue");

}

}

4、发送者通过Controller类发送消息：

@RestController

public class SendController {

@Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

String message = "message is "+new Date();

amqpTemplate.convertAndSend("laohanqueue",message);

return message;

}

}

5: 接受消息：

@Component

@RabbitListener(queues="laohanqueue")

public class Reveice {

@RabbitHandler

public void receive(String msg) {

System.out.println("您有新的短消息"+msg);

}

}

注意，发送者和接收者的queue name必须一致，不然不能接收

Topic Exchange的使用配置

Topic Exchange是RabbitMQ中最灵活的一种方式，它能够根据routing\_key自由的绑定不同的队列，

可以适用绝大部分的项目需求

1:

@Configuration

public class RabbitMqConfig {

final static String message = "topic.message";

final static String messages = "topic.messages";

@Bean

public Queue queue() {

return new Queue(message);

}

@Bean

public Queue queues() {

return new Queue(messages);

}

@Bean

TopicExchange getExchange() {

return new TopicExchange("exchange");

}

@Bean

Binding bindingExchangeMessage(Queue queue,TopicExchange topicExchange) {

return BindingBuilder.bind(queue).to(topicExchange).with("topic.message");

}

@Bean

Binding bindingExchangeMessages(Queue queues,TopicExchange topicExchange) {

return BindingBuilder.bind(queues).to(topicExchange).with("topic.#");

}

}

2：@RestController

public class SendController {

@Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

String message = "message is "+new Date();

amqpTemplate.convertAndSend("exchange","topic.message",message);

return message;

}

@RequestMapping("/send2")

public String send2() {

String message = "message2 is "+new Date();

amqpTemplate.convertAndSend("exchange","topic.messages",message);

return message;

}

}

3：@Component

@RabbitListener(queues="topic.message")

public class Reveice {

@RabbitHandler

public void receive(String msg) {

System.out.println("server 1 您有新的短消息"+msg);

}

}

4：@Component

@RabbitListener(queues="topic.messages")

public class Reveice2 {

@RabbitHandler

public void receive(String msg) {

System.out.println("server 2 您有新的短消息"+msg);

}

}

Topic Exchange 转发消息主要是根据通配符，队列topic.message只能匹配topic.message的路由。而topic.messages匹配路由规则是topic.#，所以它可以匹配topic.开头的全部路由。而topic.#发送的消息也只能是topic.#的接受者才能接收。

结果：

按/send发送请求，则两个消息队列全部命中，发一条消息，会有两个接受者。

按/send2发送请求，则只匹配messages的消息队列，只有一个接受者。

[**RabbitMq 队列的一些常见方法及参数**](https://www.cnblogs.com/fanqisoft/p/10389044.html)

方法：

　　1、QueueDeclare　　声明队列

1 public static QueueDeclareOk QueueDeclare(String queue, Boolean durable, Boolean exclusive, Boolean autoDelete, IDictionary<String, Object> arguments);

queue:声明的队列名称

durable：是否持久化，是否将队列持久化到mnesia数据库中，有专门的表保存我们的队列声明。

exclusive：排外，①当前定义的队列是connection的channel是共享的，其他的connection是访问不到的。②当connection关闭的时候，队列将被删除。

autoDelete：自动删除，当最后一个consumer（消费者）断开之后，队列将自动删除。

arguments：参数是rabbitmq的一个扩展，功能非常强大，基本是AMPQ中没有的。  
  
2、QueueDeclareNoWait　　声明队列无需等待

3、QueueDeclarePassive　　消极的队列声明

　　被用来检测队列是否存在，不存在则抛出异常，存在则不做任何操作。

参数：

　　1.x-message-ttl：Number

　　1个发布的消息在队列中存在多长时间后被取消（单位毫秒）

　　\*可以对单个消息设置过期时间

　　2.x-expires：Number

　　当Queue（队列）在指定的时间未被访问，则队列将被自动删除。

　　3.x-max-length：Number

　　队列所能容下消息的最大长度。当超出长度后，新消息将会覆盖最前面的消息，类似于Redis的LRU算法。

　　4.x-max-length-bytes：Number

　　限定队列的最大占用空间，当超出后也使用类似于Redis的LRU算法。

　　5.x-overflow：String

　　设置队列溢出行为。这决定了当达到队列的最大长度时，消息会发生什么。有效值为Drop Head或Reject Publish。

　　6.x-dead-letter-exchange：String

　　有时候我们希望当队列的消息达到上限后溢出的消息不会被删除掉，而是走到另一个队列中保存起来。

　　7.x-dead-letter-routing-key：String

　　如果不定义，则默认为溢出队列的routing-key，因此，一般和6一起定义。

　　8.x-max-priority：Number

　　如果将一个队列加上优先级参数，那么该队列为优先级队列。

　　　　1）、给队列加上优先级参数使其成为优先级队列

　　　　x-max-priority=10【值不要太大，本质是一个树结构】

　　　　2）、给消息加上优先级属性

[复制代码](javascript:void(0);)

　　通过优先级特性，将一个队列编程stack（堆栈）

　　9.x-queue-mode：String

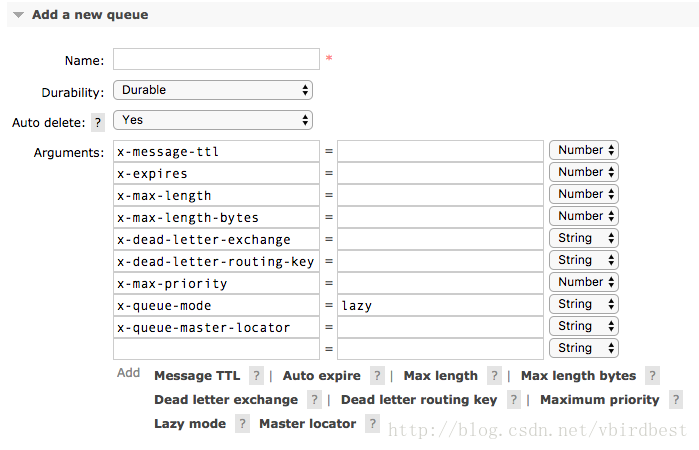
　　队列类型　　x-queue-mode=lazy　　懒队列，在磁盘上尽可能多地保留消息以减少RAM使用；如果未设置，则队列将保留内存缓存以尽可能快地传递消息。

　　10.x-queue-master-locator：String

　　将队列设置为主位置模式，确定在节点集群上声明时队列主位置所依据的规则。

**RabbitMq各个参数含义**

论队列声明的各个参数



1. queueDeclare(String queue,
2. boolean durable,
3. boolean exclusive,
4. Map<String, Object> arguments);

* 1
* 2
* 3
* 4
* queue: 队列名称
* durable： 是否持久化, 队列的声明默认是存放到内存中的，如果rabbitmq重启会丢失，如果想重启之后还存在就要使队列持久化，保存到Erlang自带的Mnesia数据库中，当rabbitmq重启之后会读取该数据库
* exclusive：是否排外的，有两个作用，一：当连接关闭时connection.close()该队列是否会自动删除；二：该队列是否是私有的private，如果不是排外的，可以使用两个消费者都访问同一个队列，没有任何问题，如果是排外的，会对当前队列加锁，其他通道channel是不能访问的，如果强制访问会报异常：com.rabbitmq.client.ShutdownSignalException: channel error; protocol method: #method<channel.close>(reply-code=405, reply-text=RESOURCE\_LOCKED - cannot obtain exclusive access to locked queue 'queue\_name' in vhost '/', class-id=50, method-id=20)一般等于true的话用于一个队列只能有一个消费者来消费的场景
* autoDelete：是否自动删除，当最后一个消费者断开连接之后队列是否自动被删除，可以通过RabbitMQ Management，查看某个队列的消费者数量，当consumers = 0时队列就会自动删除
* arguments：   
  队列中的消息什么时候会自动被删除？
  + Message TTL(x-message-ttl)：设置队列中的所有消息的生存周期(统一为整个队列的所有消息设置生命周期), 也可以在发布消息的时候单独为某个消息指定剩余生存时间,单位毫秒, 类似于redis中的ttl，生存时间到了，消息会被从队里中删除，注意是消息被删除，而不是队列被删除， 特性Features=TTL, 单独为某条消息设置过期时间AMQP.BasicProperties.Builder properties = new AMQP.BasicProperties().builder().expiration(“6000”);   
    channel.basicPublish(EXCHANGE\_NAME, “”, properties.build(), message.getBytes(“UTF-8”));
  + Auto Expire(x-expires): 当队列在指定的时间没有被访问(consume, basicGet, queueDeclare…)就会被删除,Features=Exp
  + Max Length(x-max-length): 限定队列的消息的最大值长度，超过指定长度将会把最早的几条删除掉， 类似于mongodb中的固定集合，例如保存最新的100条消息, Feature=Lim
  + Max Length Bytes(x-max-length-bytes): 限定队列最大占用的空间大小， 一般受限于内存、磁盘的大小, Features=Lim B
  + Dead letter exchange(x-dead-letter-exchange)： 当队列消息长度大于最大长度、或者过期的等，将从队列中删除的消息推送到指定的交换机中去而不是丢弃掉,Features=DLX
  + Dead letter routing key(x-dead-letter-routing-key)：将删除的消息推送到指定交换机的指定路由键的队列中去, Feature=DLK
  + Maximum priority(x-max-priority)：优先级队列，声明队列时先定义最大优先级值(定义最大值一般不要太大)，在发布消息的时候指定该消息的优先级， 优先级更高（数值更大的）的消息先被消费,
  + Lazy mode(x-queue-mode=lazy)： Lazy Queues: 先将消息保存到磁盘上，不放在内存中，当消费者开始消费的时候才加载到内存中
  + Master locator(x-queue-master-locator)

**注意**

关于队列的声明，如果使用同一套参数进行声明了，就不能再使用其他参数来声明，要么删除该队列重新删除，可以使用命令行删除也可以在RabbitMQ Management上删除，要么给队列重新起一个名字。

**队列持久化**

重启RabbitMQ服务器(可以通过rabbitmqctl stop\_app关闭服务器，rabbitmqctl start\_app重启服务器)，可以登录RabbitMQ Management—> Queues中可以看到之前声明的队列还存在

1. boolean durable = true;
2. channel.queueDeclare(QUEUE\_NAME, durable, false, false, arguments);

* 1
* 2

**消息持久化**

设置消息持久化必须先设置队列持久化，要不然队列不持久化，消息持久化，队列都不存在了，消息存在还有什么意义。消息持久化需要将交换机持久化、队列持久化、消息持久化，才能最终达到持久化的目的

方式一：设置deliveryMode=2

1. channel.exchangeDeclare(EXCHANGE\_NAME, BuiltinExchangeType.DIRECT, true);
2. channel.queueDeclare(QUEUE\_NAME, true, false, false, arguments);
3. channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "");
5. String message = "Hello RabbitMQ: ";
6. *// 设置消息持久化*
7. AMQP.BasicProperties.Builder properties = new AMQP.BasicProperties().builder();
8. properties.deliveryMode(2); *// 设置消息是否持久化，1： 非持久化 2：持久化*
10. channel.basicPublish(EXCHANGE\_NAME, "", properties.build(), message.getBytes("UTF-8"));

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

方式二：设置BasicProperties为MessageProperties.PERSISTENT\_TEXT\_PLAIN

1. channel.queueDeclare(QUEUE\_NAME, true, false, false, arguments);
2. channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "");
4. String message = "Hello RabbitMQ: ";
5. channel.basicPublish(EXCHANGE\_NAME, "", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes("UTF-8"));

* 1
* 2
* 3
* 4
* 5

**Message TTL消息剩余生存时间**

统一设置队列中的所有消息的过期时间，例如设置10秒，10秒后这个队列的消息清零

方式一：为该队列的所有消息统一设置相同的声明周期

1. Map<String, Object> arguments = new HashMap<String, Object>();
2. arguments.put("x-message-ttl", 10000);
4. *// 声明队列时指定队列中的消息过期时间*
5. channel.queueDeclare(QUEUE\_NAME, false, false, false, arguments);
6. channel.basicPublish(EXCHANGE\_NAME, "", null, message.getBytes("UTF-8"));

* 1
* 2
* 3
* 4
* 5
* 6

方式二：单独为某条消息单独设置时间

// expiration: 设置单条消息的过期时间

1. channel.queueDeclare(QUEUE\_NAME, false, false, false, null);
2. channel.basicPublish(EXCHANGE\_NAME, "", null, message.getBytes("UTF-8"));
4. for(int i = 1; i <= 5; i++) {
5. AMQP.BasicProperties.Builder properties = new AMQP.BasicProperties()
6. .builder().expiration( i \* 1000 + "");
8. channel.basicPublish(EXCHANGE\_NAME, "", properties.build(), (message + i).getBytes("UTF-8"));
9. }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

**Auto Expire自动过期**

x-expires用于当多长时间没有消费者访问该队列的时候，该队列会自动删除，可以设置一个延迟时间，如仅启动一个生产者，10秒之后该队列会删除，或者启动一个生产者，再启动一个消费者，消费者运行结束后10秒，队列也会被删除

1. Map<String, Object> arguments = new HashMap<String, Object>();
2. arguments.put("x-expires", 10000);
4. channel.queueDeclare(QUEUE\_NAME, false, false, false, arguments);
5. channel.basicPublish(EXCHANGE\_NAME, "", null, message.getBytes("UTF-8"));

* 1
* 2
* 3
* 4
* 5

**Max Length最大长度**

x-max-length:用于指定队列的长度，如果不指定，可以认为是无限长，例如指定队列的长度是4，当超过4条消息，前面的消息将被删除，给后面的消息腾位

1. Map<String, Object> arguments = new HashMap<String, Object>();
2. arguments.put("x-max-length", 4);
4. channel.queueDeclare(QUEUE\_NAME, false, false, false, arguments);
5. for(int i = 1; i <= 5; i++) {
6. channel.basicPublish(EXCHANGE\_NAME, "", MessageProperties.PERSISTENT\_TEXT\_PLAIN, (message + i).getBytes("UTF-8"));
7. }

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**Max Length Bytes代码片段**

x-max-length-bytes: 用于指定队列存储消息的占用空间大小，当达到最大值是会删除之前的数据腾出空间

1. Map<String, Object> arguments = new HashMap<String, Object>();
2. rguments.put("x-max-length-bytes", 1024);
4. channel.queueDeclare(QUEUE\_NAME, false, false, false, arguments);
5. channel.basicPublish(EXCHANGE\_NAME, "", null, message.getBytes("UTF-8"));

* 1
* 2
* 3
* 4
* 5

**Maximum priority最大优先级**

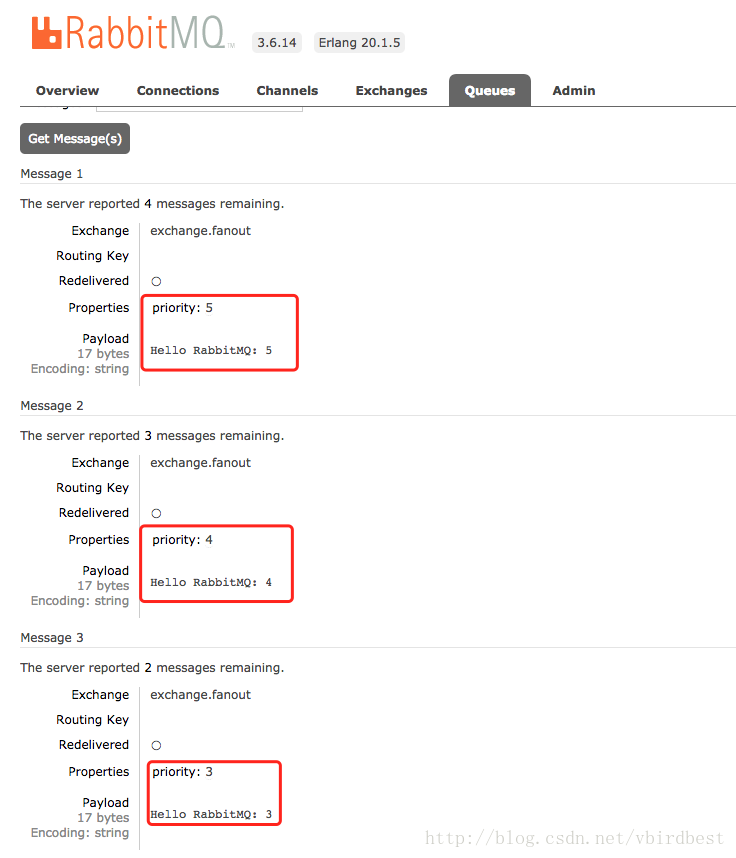
x-max-priority: 设置消息的优先级，优先级值越大，越被提前消费。

正常情况下不适用优先级   
Hello RabbitMQ: 1   
Hello RabbitMQ: 2   
Hello RabbitMQ: 3   
Hello RabbitMQ: 4   
Hello RabbitMQ: 5

使用优先级顺序正好相反   
Hello RabbitMQ: 5   
Hello RabbitMQ: 4   
Hello RabbitMQ: 3   
Hello RabbitMQ: 2   
Hello RabbitMQ: 1

1. Map<String, Object> arguments = new HashMap<String, Object>();
2. arguments.put("x-max-priority", 10);
4. channel.queueDeclare(QUEUE\_NAME, false, false, false, arguments);
5. for(int i = 1; i <= 5; i++) {
6. AMQP.BasicProperties.Builder properties = new AMQP.BasicProperties()
7. .builder().priority(i);
8. channel.basicPublish(EXCHANGE\_NAME, "", properties.build(), (message + i).getBytes("UTF-8"));
9. }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9



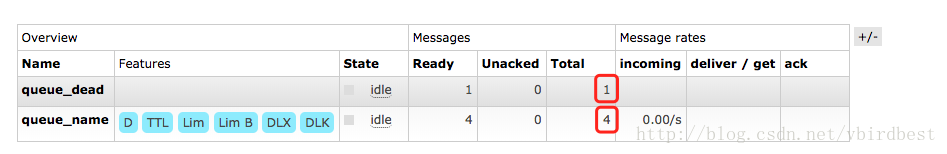
**Dead letter exchange(死亡交换机) 和 Dead letter routing key(死亡路由键)**

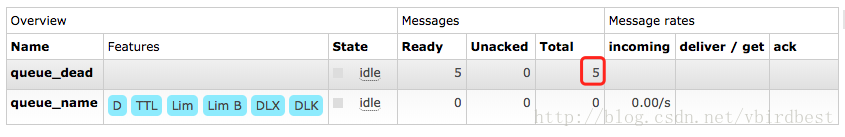
当队列中的消息过期，或者达到最大长度而被删除，或者达到最大空间时而被删除时，可以将这些被删除的信息推送到其他交换机中，让其他消费者订阅这些被删除的消息，处理这些消息

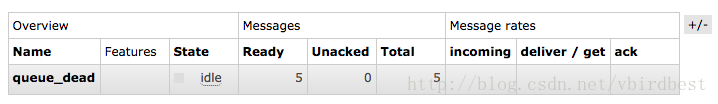
1. public void testBasicPublish() throws IOException, TimeoutException, InterruptedException {
2. ConnectionFactory factory = new ConnectionFactory();
3. factory.setHost("127.0.0.1");
4. factory.setPort(AMQP.PROTOCOL.PORT);
5. factory.setUsername("mengday");
6. factory.setPassword("mengday");
8. Connection connection = factory.newConnection();
9. Channel channel = connection.createChannel();
11. *// 声明一个接收被删除的消息的交换机和队列*
12. String EXCHANGE\_DEAD\_NAME = "exchange.dead";
13. String QUEUE\_DEAD\_NAME = "queue\_dead";
14. channel.exchangeDeclare(EXCHANGE\_DEAD\_NAME, BuiltinExchangeType.DIRECT);
15. channel.queueDeclare(QUEUE\_DEAD\_NAME, false, false, false, null);
16. channel.queueBind(QUEUE\_DEAD\_NAME, EXCHANGE\_DEAD\_NAME, "routingkey.dead");
18. String EXCHANGE\_NAME = "exchange.fanout";
19. String QUEUE\_NAME = "queue\_name";
20. channel.exchangeDeclare(EXCHANGE\_NAME, BuiltinExchangeType.FANOUT);
22. Map<String, Object> arguments = new HashMap<String, Object>();
23. arguments.put("x-message-ttl", 15000);
24. arguments.put("x-max-length", 4);
25. arguments.put("x-max-length-bytes", 1024);
26. arguments.put("x-expires", 30000);
28. arguments.put("x-dead-letter-exchange", "exchange.dead");
29. arguments.put("x-dead-letter-routing-key", "routingkey.dead");
30. channel.queueDeclare(QUEUE\_NAME, true, false, false, arguments);
31. channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "");
33. String message = "Hello RabbitMQ: ";
34. for(int i = 1; i <= 5; i++) {
35. channel.basicPublish(EXCHANGE\_NAME, "", null, (message + i).getBytes("UTF-8"));
36. }

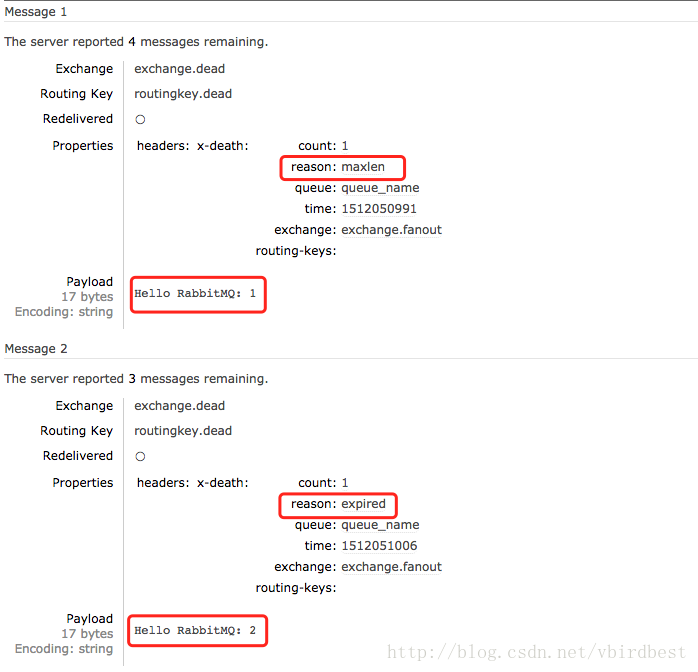
39. channel.close();
40. connection.close();
41. }

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

刚开始由于队列长度是4，总共发送5条消息，所以最早进入队列的消息1将被删除掉，被推送到“死亡队列”中，所以看到普通队列的消息为4条，死亡队列的消息为1条，是消息1   


随着时间的流逝，普通队列中的消息都该过期了，所以消息2、3、4、5都被推送到死亡队列，所以死亡队列消息是5条，普通队列的消息条数为0   


再随着时间的流逝，普通队列过了指定时间没有被消费者访问，这个队列自动被删除了，所以看不到普通队列了，只有死亡队列   


查看死亡队列的消息可以得知，消息一死亡的原因是maxlen达到了最大长度，消息2、3、4、5都是因为生存时间到了导致死亡的   
   


**一个比较杂的综合示例**

关于消费者就不用代码来获取消息了，直接在RabbitMQ Management点击某个队列的名字，然后Get Message(s) 即可获取

该示例使用很多参数配置，可能实际使用不会像这样用，因为这样好像不太配套。

1. public class Producer {
2. @Test
3. public void testBasicPublish() throws IOException, TimeoutException, InterruptedException {
4. ConnectionFactory factory = new ConnectionFactory();
5. factory.setHost("127.0.0.1");
6. factory.setPort(AMQP.PROTOCOL.PORT);
7. factory.setUsername("mengday");
8. factory.setPassword("mengday");
10. Connection connection = factory.newConnection();
11. Channel channel = connection.createChannel();
13. *// 声明一个接收被删除的消息的交换机和队列*
14. String EXCHANGE\_DEAD\_NAME = "exchange.dead";
15. String QUEUE\_DEAD\_NAME = "queue\_dead";
16. channel.exchangeDeclare(EXCHANGE\_DEAD\_NAME, BuiltinExchangeType.DIRECT);
17. channel.queueDeclare(QUEUE\_DEAD\_NAME, false, false, false, null);
18. channel.queueBind(QUEUE\_DEAD\_NAME, EXCHANGE\_DEAD\_NAME, "routingkey.dead");
20. String EXCHANGE\_NAME = "exchange.fanout";
21. String QUEUE\_NAME = "queue\_name";
22. channel.exchangeDeclare(EXCHANGE\_NAME, BuiltinExchangeType.FANOUT);

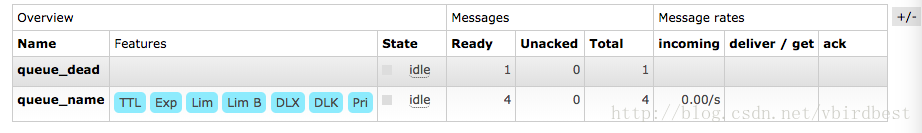
25. Map<String, Object> arguments = new HashMap<String, Object>();
26. *// 统一设置队列中的所有消息的过期时间*
27. arguments.put("x-message-ttl", 30000);
28. *// 设置超过多少毫秒没有消费者来访问队列，就删除队列的时间*
29. arguments.put("x-expires", 20000);
30. *// 设置队列的最新的N条消息，如果超过N条，前面的消息将从队列中移除掉*
31. arguments.put("x-max-length", 4);
32. *// 设置队列的内容的最大空间，超过该阈值就删除之前的消息*
33. arguments.put("x-max-length-bytes", 1024);
34. *// 将删除的消息推送到指定的交换机，一般x-dead-letter-exchange和x-dead-letter-routing-key需要同时设置*
35. arguments.put("x-dead-letter-exchange", "exchange.dead");
36. *// 将删除的消息推送到指定的交换机对应的路由键*
37. arguments.put("x-dead-letter-routing-key", "routingkey.dead");
38. *// 设置消息的优先级，优先级大的优先被消费*
39. arguments.put("x-max-priority", 10);
40. channel.queueDeclare(QUEUE\_NAME, false, false, false, arguments);
41. channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "");

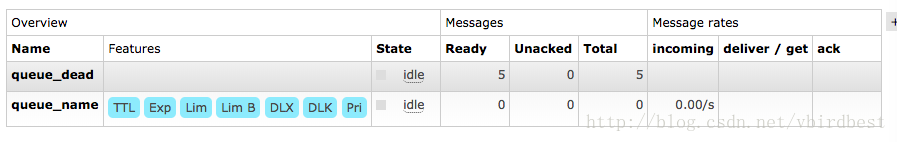
44. String message = "Hello RabbitMQ: ";
45. for(int i = 1; i <= 5; i++) {
46. *// expiration: 设置单条消息的过期时间*
47. AMQP.BasicProperties.Builder properties = new AMQP.BasicProperties().builder().priority(i).expiration( i \* 1000 + "");
48. channel.basicPublish(EXCHANGE\_NAME, "", properties.build(), (message + i).getBytes("UTF-8"));
49. }

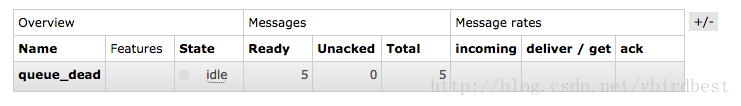
52. channel.close();
53. connection.close();
54. }
55. }

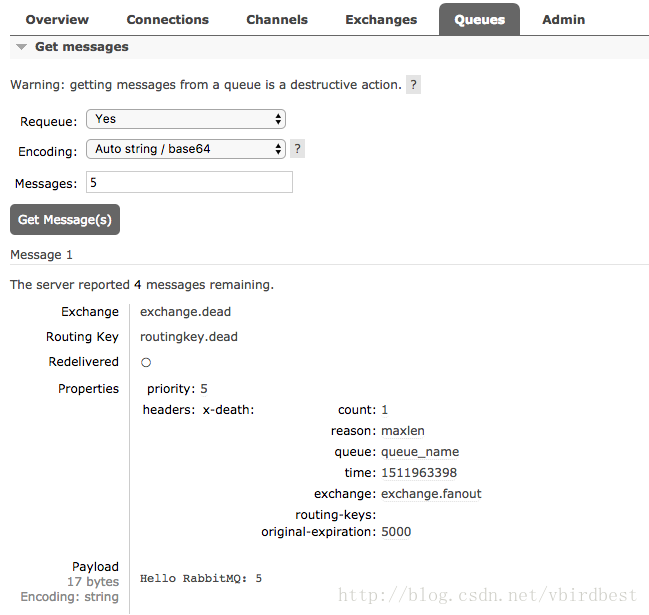
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56

**运行效果**





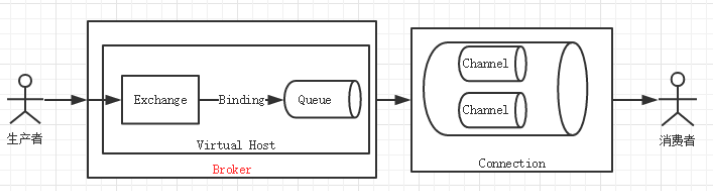








[RabbitMQ事务和Confirm发送方消息确认](https://www.cnblogs.com/vipstone/p/9350075.html)



## 一、事务使用

正常情况下，如果消息经过交换器进入队列就可以完成消息的持久化，但如果消息在没有到达broker之前出现意外，那就造成消息丢失，有没有办法可以解决这个问题？

RabbitMQ有两种方式来解决这个问题：

通过AMQP提供的事务机制实现；

使用发送者确认模式实现；

事务的实现主要是对信道（Channel）的设置，主要的方法有三个：

channel.txSelect()声明启动事务模式；

channel.txComment()提交事务；

channel.txRollback()回滚事务；

从上面的可以看出事务都是以tx开头的，tx应该是transaction extend（事务扩展模块）的缩写.

===CODE

// 创建连接

ConnectionFactory factory = new ConnectionFactory();

factory.setUsername("guest");

factory.setPassword("guest");

factory.setHost("localhost");

factory.setPort(5672);

Connection conn = factory.newConnection();

// 创建信道

Channel channel = conn.createChannel();

// 声明队列

channel.queueDeclare("tq1", true, false, false, null);

String message = String.format("时间 => %s", new Date().getTime());

try {

channel.txSelect();// 声明事务

// 发送消息

channel.basicPublish("", "tq1", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes("UTF-8"));

channel.txCommit(); // 提交事务

System.out.println("提交成功");

} catch (Exception e) {

// TODO: handle exception

channel.txRollback();

System.out.println("提交失败"+e);

}finally {

channel.close();

conn.close();

}

上就完成了事务的交互流程，如果其中任意一个环节出现问题，就会抛出IoException移除，这样用户就可以拦截异常进行事务回滚，或决定要不要重复消息。

那么，既然已经有事务了，没什么还要使用发送方确认模式呢，原因是因为事务的性能是非常差的。事务模式的性能要差很多，那有没有既能保证消息的可靠性又能兼顾性能的解决方案呢？那就是接下来要讲的**Confirm发送方确认模式**。

## 二、Confirm发送方确认模式

Confirm发送方确认模式使用和事务类似，也是通过设置Channel进行发送方确认的。

Confirm的三种实现方式：

方式一：channel.waitForConfirms()普通发送方确认模式；

方式二：channel.waitForConfirmsOrDie()批量确认模式；

方式三：channel.addConfirmListener()异步监听发送方确认模式；

### 方式一：普通Confirm模式

channel.confirmSelect();// 开启发送方确认模式

// 发送消息

channel.basicPublish("", "tq1", null, message.getBytes("UTF-8"));

if(channel.waitForConfirms()) {

System.out.println("发送成功");

}else {

System.out.println("发送失败");

}

我们只需要在推送消息之前，channel.confirmSelect()声明开启发送方确认模式，再使用channel.waitForConfirms()等待消息被服务器确认即可。

### 方式二：批量Confirm模式

channel.confirmSelect();// 开启发送方确认模式

// 发送消息

for (int i = 0; i < 10000; i++) {

channel.basicPublish("", "tq1", null, message.getBytes("UTF-8"));

}

channel.waitForConfirmsOrDie();

////直到所有信息都发布，只要有一个未确认就会IOException

System.out.println("全部执行完成"+(end-start));

channel.waitForConfirmsOrDie()，使用同步方式等所有的消息发送之后才会执行后面代码，只要有一个消息未被确认就会抛出IOException异常。

### 方式三：异步Confirm模式

channel.confirmSelect();

// 发送消息

long start = System.currentTimeMillis();

for (int i = 0; i <10000; i++) {

channel.basicPublish("", "tq1", null, message.getBytes("UTF-8"));

}

channel.addConfirmListener(new ConfirmListener() {

public void handleNack(long deliveryTag, boolean multiple) throws IOException {

// TODO Auto-generated method stub

System.out.println("消息未确认"+deliveryTag);

}

public void handleAck(long deliveryTag, boolean multiple) throws IOException {

// TODO Auto-generated method stub

System.out.println("消息已经确认"+deliveryTag+" "+multiple);

}

});

异步模式的优点，就是执行效率高，不需要等待消息执行完，只需要监听消息即可.

可以看出，代码是异步执行的，消息确认有可能是批量确认的，是否批量确认在于返回的multiple的参数，此参数为boolean值，如果true表示批量执行了deliveryTag这个值以前的所有消息，如果为false的话表示单条确认。

Confirm性能测试

测试前提：与事务一样，我们发送1w条消息。

方式一：Confirm普通模式

执行花费时间：2253s

执行花费时间：2018s

执行花费时间：2043s

方式二：Confirm批量模式

执行花费时间：1576s

执行花费时间：1400s

执行花费时间：1374s

方式三：Confirm异步监听方式

执行花费时间：1498s

执行花费时间：1368s

执行花费时间：1363s

总结

综合总体测试情况来看：Confirm批量确定和Confirm异步模式性能相差不大，Confirm模式要比事务快10倍左右。

RabbitMq:TTL Time to Live。

可以对消息和队列设置TTL。

1：设置消息的TTL.

两种方式设置TTL:

1:通过队列属性设置，队列中的所有消息都有相同的过期时间。

Map map = new HashMap();

map.put("x-message-ttl", 6000);

channel.queueDeclare(QUEUENAME, false, false, false, map);

如果不设置TTL，则表示此消息不会过期；

如果将TTL设置为0，则表示除非此时能够直接将消息投递到消费者，否则改消息

会被立即丢弃。可以部分替代3.0之前版本的immediate参数。

2：对消息本身进行单独设置。每条消息的TTL可以不同。

如果两种方法一起使用，则消息TTL以两者之间较小的那个数值为准。

消息在队列中的生存时间一旦超过设置的TTL值，就会变成死信。

针对每条消息设置TTL的方法是channel.basicPublish方法中加入expiration的属性参数，

单位是毫秒。

AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties().builder();

builder.deliveryMode(2);//消息持久化

builder.expiration("6000");//设置TTL=6000毫秒

AMQP.BasicProperties properties = builder.build();

channel.basicPublish("", QUEUENAME, false, properties, msg.getBytes("UTF-8"));

DLX Dead-Letter-Exchange 死信交换器

当消息在一个队列中编程死信之后，它能被重新被发送到另一个交换器中，这个

交换器就是DLX,绑定DLX的队列就是死信队列。

消息变成死信一般是由于一下几种情况：

1：消息被拒绝 Basic.Reject/Basic.Nack 并且设置requeue参数为false

2: 消息过期

3：队列达到最大长度

DLX也是一个正常的交换器，和一般的交换器没有区别，它能在任何的队列上被指定，实际上

就是设置某个队列的属性。当这个队列中存在死信时，RabbitMQ就会自动地将这个消息重新发布

到设置的DLX上去，进而被路由到另一个队列，即死信队列。

通过在channel.queueDeclare方法中设置x-dead-letter-exchange参数来为这个队列添加DLX.

死信代码：

消息的生产者：

=====================================================

public static void main(String[] args) throws Exception {

Connection connection = ConnectionUtil.getConnection();

// 3. 创建channel

Channel channel = connection.createChannel();

channel.exchangeDeclare("exchange.dlx", "direct",true);//死信交换机

channel.exchangeDeclare("exchange.normal", "fanout",true);//普通交换机

Map map = new HashMap();

// map.put("x-message-ttl", 10000);

map.put("x-max-length",20);

map.put("x-dead-letter-exchange", "exchange.dlx");

map.put("x-dead-letter-routing-key", "routingkey");

//声明队列

channel.queueDeclare("queue.normal",true,false,false,map);

channel.queueBind("queue.normal", "exchange.normal", "");

channel.queueDeclare("queue.dlx",true,false,false,null);

channel.queueBind("queue.dlx", "exchange.dlx", "routingkey");

for (int i = 0; i < 100; i++) {

String msg = "how are you"+i;

channel.basicPublish("exchange.normal", "rk", null, msg.getBytes());

}

// 关闭连接

channel.close();

connection.close();

}

=====================================================

普通消息接收

public static void main(String[] args) throws Exception {

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

channel.queueBind("queue.normal", "exchange.normal", "");

DefaultConsumer consume = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String msg = new String(body, "UTF-8");

System.out.println("接受到普通消息" + msg);

channel.basicAck(envelope.getDeliveryTag(), false);

}

};

channel.basicConsume("queue.normal", false, consume);

}

=====================================================

死信接收

public static void main(String[] args) throws Exception {

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

channel.queueDeclare("queue.dlx", true, false, false, null);

channel.queueBind("queue.dlx", "exchange.dlx", "routingkey");

DefaultConsumer consume = new DefaultConsumer(channel) {

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,

byte[] body) throws IOException {

String msg = new String(body, "UTF-8");

System.out.println("接受到死信" + msg);

channel.basicAck(envelope.getDeliveryTag(), false);

}

};

channel.basicConsume("queue.dlx", false, consume);

}

=====================================================

=====================================================

=====================================================

main方法没有默认的消息补偿

在消费者消费产生异常的时候，消息会重新进入消息队列，等待新的消费者。

Rabbitmq 默认情况下 如果消费者程序出现异常情况 会自动实现补偿机制 也就是 重试机制：

==========================

建立Boot测试环境：

1：yml

server:

port: 8000

spring:

rabbitmq:

host: 127.0.0.1

port: 5672

username: guest

password: guest

2:pom.xml

<dependency>

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

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

</dependency>

3:@Configuration

public class RabbitMqConfig {

@Bean

public Queue getQueue() {

return new Queue("laohanqueue");

}

}

4:@Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

String message = "message is "+new Date();

amqpTemplate.convertAndSend("laohanqueue",message);

return message;

}

5:@Component

@RabbitListener(queues="laohanqueue")

public class Reveice {

@RabbitHandler

public void receive(String msg) {

System.out.println("您有新的短消息"+msg);

int i = 9/0;

}

}

==========================

测试结果：

当生产者投递消息后：

消费者会不停的进行打印: 消息一直没有被消费

原因 Rabbitmq 默认情况下 如果消费者程序出现异常情况 会自动实现补偿机制 也就是 重试机制

@RabbitListener底层使用AOP进行拦截，如果程序没有抛出异常，自动提交事务。

如果Aop使用异常通知 拦截获取异常信息的话 ，

自动实现补偿机制，该消息会一直缓存在Rabbitmq服务器端进行重放，一直重试到不抛出异常为准。

可以修改重试策略

一般来说默认5s重试一次，

消费者配置：

spring:

rabbitmq:

host: 127.0.0.1

port: 5672

username: guest

password: guest

listener:

simple:

retry:

####开启消费者重试

enabled: true

####最大重试次数（默认无数次）

max-attempts: 5

####重试间隔次数

initial-interval: 3000ms

重试机制都是间隔性的 每次都是一个线程 单线程重试

效果： 重试5次 不行就放弃了。

=================================================================

消费者如果保证消息幂等性，不被重复消费

消息幂等性，其实就是保证同一个消息不被消费者重复消费两次。

当消费者消费完消息之后，通常会发送一个ack应答确认信息给生产者，

但是这中间有可能因为网络中断等原因，导致生产者未能收到确认消息，

由此这条消息将会被 重复发送给其他消费者进行消费，

实际上这条消息已经被消费过了

解决办法:

消费者端实现幂等性，意味着我们的消息永远不会消费多次，即使我们收到了多条一样的消息。通常有两种方式来避免消费重复消费：

消息全局ID或者写个唯一标识(如时间戳、UUID等) ：每次消费消息之前根据消息id去判断该消息是否已消费过，如果已经消费过，则不处理这条消息，否则正常消费消息，并且进行入库操作。(消息全局ID作为数据库表的主键，防止重复)

利用Redis的setnx 命令：给消息分配一个全局ID，只要消费过该消息，将 < id,message>以K-V键值对形式写入redis，消费者开始消费前，先去redis中查询有没消费记录即可。

代码如下：

=================================================================

1：pom.xml

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

2:application.yml

spring:

rabbitmq:

host: 127.0.0.1

port: 5672

username: guest

password: guest

listener:

simple:

retry:

####开启消费者重试

enabled: true

####最大重试次数（默认无数次）

max-attempts: 5

####重试间隔次数

initial-interval: 3000ms

3:@Configuration

public class RabbitMqConfig {

@Bean

public Queue getQueue() {

return new Queue("laohanqueue");

}

}

4: @Resource

private AmqpTemplate amqpTemplate;

@RequestMapping("/send")

public String send() {

MessageProperties messageProperties = new MessageProperties();

messageProperties.setMessageId(UUID.randomUUID().toString());

messageProperties.setContentType("text/plain");

messageProperties.setContentEncoding("utf-8");

Message message = new Message("hello,woniu!".getBytes(), messageProperties);

amqpTemplate.convertAndSend("laohanqueue",message);

return "ok";

}

5:@Component

@RabbitListener(queues="laohanqueue")

public class Reveice {

@Resource

private RedisTemplate redisTemplate;

@RabbitHandler

public void receive(String value,Message message, Channel channel) throws Exception {

System.out.println("=====>"+value+" "+message);

String messageid = message.getMessageProperties().getMessageId();

String msg = new String(message.getBody(), "UTF-8"); //消息内容获取之

System.out.println("您有新的短消息"+msg+" "+messageid);

if(messageid==null||redisTemplate.opsForValue().get(messageid)!=null){

System.out.println("已经消费过了，不要重复进行消费");

return;

}else{

System.out.println("处理完业务数据");

redisTemplate.opsForValue().set(messageid, "ok");

}

int i = 9/0;

}

}

则即便发生网络故障，即便强制报错，则改消息也只会被处理1次，而且消息补偿也只有一次，不会造成

5次消息补偿。

削峰：

削峰填谷的应用场景：

举个业务场景的栗子，秒杀业务：

上游发起下单操作

下游完成秒杀业务逻辑（库存检查，库存冻结，余额检查，余额冻结，订单生成，余额扣减，库存扣减，生成流水，余额解冻，库存解冻）

上游下单业务简单，每秒发起了10000个请求，下游秒杀业务复杂，每秒只能处理2000个请求，很有可能上游不限速的下单，导致下游系统被压垮，引发雪崩。

为了避免雪崩，常见的优化方案有两种：

1. 业务上游队列缓冲，限速发送
   1. 问答，
   2. 分层过滤
2. 业务下游队列缓冲，限速执行
   1. 排队 MQ

rabbitmq提供了一种服务质量保障功能，即在非自动确认消息的前提下，如果一定数目的消息未被确认，不进行消费新的消息。

使用 basicqos方法：

在消费端进行使用。 0 1 false

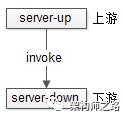
prefetSize：0

prefetCount：这个值一般在设置为非自动ack的情况下生效，一般大小为1

global： true是channel级别， false是消费者级别

注意：我们要使用非自动ack

问：站点与服务，服务与服务上下游之间，一般如何通讯？  
  
答：有两种常见的方式

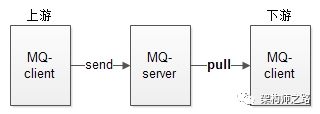


上游将消息发给MQ，MQ将消息推送给下游

问：为什么会有流量冲击？  
  
答：不管采用“直接调用”还是“MQ推送”，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

问：MQ怎么改能缓冲流量？

答：由MQ-server推模式，升级为MQ-client拉模式。



如果你看过秒杀系统的流量监控图的话，你会发现它是一条直线，就在秒杀开始那一秒是一条很直很直的线，这是因为秒杀请求在时间上高度集中于某一特定的时间点。这样一来，就会导致一个特别高的流量峰值，它对资源的消耗是瞬时的。

但是对秒杀这个场景来说，最终能够抢到商品的人数是固定的，也就是说100人和10000人发起请求的结果都是一样的，并发度越高，无效请求也越多。

但是从业务上来说，秒杀活动是希望更多的人来参与的，也就是开始之前希望有更多的人来刷页面，但是真正开始下单时，秒杀请求并不是越多越好。因此我们可以设计一些规则，让并发的请求更多地延缓，而且我们甚至可以过滤掉一些无效请求。

为什么要削峰

为什么要削峰呢？或者说峰值会带来哪些坏处？

我们知道服务器的处理资源是恒定的，你用或者不用它的处理能力都是一样的，所以出现峰值的话，很容易导致忙到处理不过来，闲的时候却又没有什么要处理。但是由于要保证服务质量，我们的很多处理资源只能按照忙的时候来预估，而这会导致资源的一个浪费。

这就好比因为存在早高峰和晚高峰的问题，所以有了错峰限行的解决方案。

削峰的存在，一是可以让服务端处理变得更加平稳，二是可以节省服务器的资源成本。

针对秒杀这一场景，削峰从本质上来说就是更多地延缓用户请求的发出，以便减少和过滤掉一些无效请求，它遵从“请求数要尽量少”的原则。

今天，我就来介绍一下流量削峰的一些操作思路：**排队、答题、分层过滤**。

这几种方式都是无损（即不会损失用户的发出请求）的实现方案，当然还有些有损的实现方案，包括我们后面要介绍的关于稳定性的一些办法，比如限流和机器负载保护等一些强制措施也能达到削峰保护的目的，当然这都是不得已的一些措施，因此就不归类到这里了。

**排队**

要对流量进行削峰，最容易想到的解决方案就是用消息队列来缓冲瞬时流量，把同步的直接调用转换成异步的间接推送，中间通过一个队列在一端承接瞬时的流量洪峰，在另一端平滑地将消息推送出去。在这里，消息队列就像“水库”一样，拦蓄上游的洪水，削减进入下游河道的洪峰流量，从而达到减免洪水灾害的目的。

用消息队列来缓冲瞬时流量的方案，如下图所示：

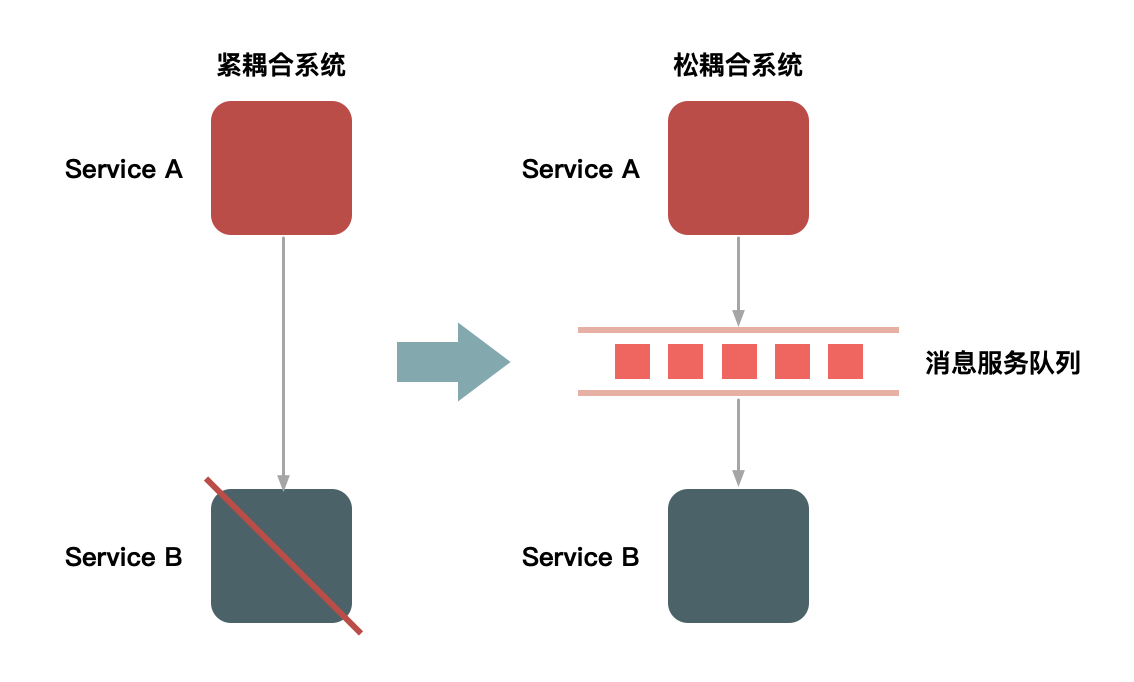


图1 用消息队列来缓冲瞬时流量

但是，如果流量峰值持续一段时间达到了消息队列的处理上限，例如本机的消息积压达到了存储空间的上限，消息队列同样也会被压垮，这样虽然保护了下游的系统，但是和直接把请求丢弃也没多大的区别。就像遇到洪水爆发时，即使是有水库恐怕也无济于事。

除了消息队列，类似的排队方式还有很多，例如：

利用线程池加锁等待也是一种常用的排队方式；

先进先出、先进后出等常用的内存排队算法的实现方式；

把请求序列化到文件中，然后再顺序地读文件（例如基于MySQL binlog的同步机制）来恢复请求等方式。

可以看到，这些方式都有一个共同特征，就是把“一步的操作”变成“两步的操作”，其中增加的一步操作用来起到缓冲的作用。

说到这里你可能会说，这样一来增加了访问请求的路径啊，并不符合我们介绍的“4要1不要”原则。没错，的确看起来不太合理，但是如果不增加一个缓冲步骤，那么在一些场景下系统很可能会直接崩溃，所以最终还是需要你做出妥协和平衡。

**答题**

你是否还记得，最早期的秒杀只是纯粹地刷新页面和点击购买按钮，它是后来才增加了答题功能的。那么，为什么要增加答题功能呢？

这主要是为了增加购买的复杂度，从而达到两个目的。

第一个目的是防止部分买家使用秒杀器在参加秒杀时作弊。2011年秒杀非常火的时候，秒杀器也比较猖獗，因而没有达到全民参与和营销的目的，所以系统增加了答题来限制秒杀器。增加答题后，下单的时间基本控制在2s后，秒杀器的下单比例也大大下降。

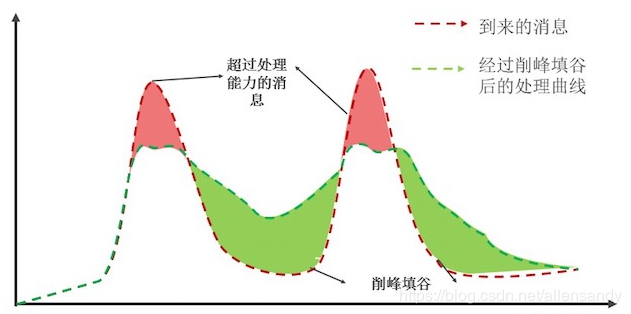
**分层过滤**

前面介绍的排队和答题要么是少发请求，要么对发出来的请求进行缓冲，而针对秒杀场景还有一种方法，就是对请求进行分层过滤，从而过滤掉一些无效的请求。分层过滤其实就是采用“漏斗”式设计来处理请求的，如下图所示。

**削峰填谷**

请求的到来，往往是没有规律的。

例如，某应用的处理能力是每秒 10 个请求。在某一秒，突然到来了 30 个请求，而接下来两秒，都没有请求到达。在这种情况下，如果直接拒绝 20 个请求，应用在接下来的两秒就会空闲。所以，需要把请求突刺均摊到一段时间内，让系统负载保持在请求处理水位之内，同时尽可能地处理更多请求，从而起到“削峰填谷”的效果。



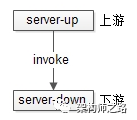
上图中，红色的部分代表超出消息处理能力的部分。观察得出，消息突刺往往都是瞬时的、不规律的，其后一段时间系统往往都会有空闲资源。把红色的那部分消息平摊到后面空闲时去处理，这样既可以保证系统负载处在一个稳定的水位，又可以尽可能地处理更多消息。通过配置流控规则，可以达到消息匀速处理的效果。

**分析问答**

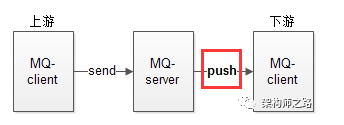
  1、问：站点与服务，服务与服务上下游之间，一般如何通讯？

        答：有两种常见的方式

       一种是“直接调用”，通过RPC框架，上游直接调用下游：



       还有一种，在某些业务场景之下（具体哪些业务场景，见《到底什么时候该使用MQ？》），可以采用“MQ推送”，上游将消息发给MQ，MQ将消息推送给下游。



    这两种都有问题：不管采用“直接调用”还是“MQ推送”，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

**2 问：以上两种为什么会有流量冲击？**

        答：不管采用“直接调用”还是“MQ推送”，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

      举个栗子，秒杀业务：

      上游发起高并发的下单操作。

      下游完成秒杀业务逻辑（库存检查，库存冻结，余额检查，余额冻结，订单生成，余额扣减，库存扣减，生成流水，余额解冻，库存解冻）。

       上游下单业务简单，每秒发起了10000个请求，下游秒杀业务复杂，每秒只能处理2000个请求，很有可能上游不限速的下单，导致下游系统被压垮，引发雪崩。

**3、问：MQ怎么改能缓冲流量**？

         答：由MQ-server推模式，升级为MQ-client拉模式。MQ-client根据自己的处理能力，每隔一定时间，或者每次拉取若干条消息，实施流控，达到保护自身的效果。并且这是MQ提供的通用功能，无需上下游修改代码。

**4、问：如果上游发送流量过大，MQ提供拉模式确实可以起到下游自我保护的作用，会不会导致消息在MQ中堆积？**  
         答：下游MQ-client拉取消息，消息接收方能够批量获取消息，需要下游消息接收方进行优化，方能够提升整体吞吐量，例如：批量写。

# RabbitMQ 队列消息的条数限制、队列字节长度限制、队列溢出行为方式

# 策略地址：<https://www.rabbitmq.com/parameters.html#policies>

# //管理策略

# // 得到mq的连接

# Connection conn = RabbitMqUtil.getConnection();

# // 通过连接得到一个通道

# Channel channel = conn.createChannel();

# //管理策略

# // 声明了一个消息队列

# HashMap<String, Object> map = new HashMap<>();

# // 设置队列最大的条数 10条

# map.put("x-max-length", 10);

# // 设置队列溢出方式 保留前10条

# //定义溢出行为-是从头上删除消息还是拒绝新发布

# map.put("x-overflow", "reject-publish");

# RabbitMQ有两种对队列长度的限制方式

# 对队列中消息的条数进行限制 x-max-length

# 对队列中消息的总量进行限制 x-max-length-bytes

# MQ性能问题

一． 要避免流控机制触发

# 服务端默认配置是当内存使用达到40%，磁盘空闲空间小于50M，即启动内存报警，磁盘报警；报警后服务端触发流控（flow control）机制。一般地，当发布端发送消息速度快于订阅端消费消息的速度时，队列中堆积了大量的消息，导致报警，就会触发流控机制。

# 触发流控机制后，RabbitMQ服务端接收发布来的消息会变慢，使得进入队列的消息减少；与此同时RabbitMQ服务端的消息推送也会受到极大的影响，测试发现，服务端推送消息的频率会大幅下降，等待下一次推送的时间，有时等1分钟，有时5分钟，甚至30分钟。

# 一旦触发流控，将导致RabbitMQ服务端性能恶化，推送消息也会变得非常缓慢；因此要做好数据设计，使得发送速率和接收速率保持平衡，而不至于引起服务器堆积大量消息，进而引发流控。通过增加服务器集群节点，增加消费者，来避免流控发生，治标不治本，而且成本高。

二． 从底层取数据一定要非常及时

# 订阅端每隔500MS调用一次amqp\_consume\_message接口函数从socket上获取数据，正常情况下，服务器每次会推送几百条消息，而且推送的频率会比较高；导致订阅端的本机socket缓冲区会很快存满，导致很多消息无法进行缓存，而被丢掉

# 

# 三． 消息大小不要超过4MB

# 客户端与RabbitMQ服务端的最大帧是128K，但消息大小却可支持数MB，这是可能是因为底层做了拆包组包的，目前我还未查看底层代码。

# 用线程来模拟50个发布者和50个订阅者；消息包大小由1K到10MB，当包大小达到4.5MB时，服务器的性能出现明显的异常，传输率尤其是每秒订阅消息的数量，出现波动，不稳定；同时有一部分订阅者的TCP连接出现断开的现象。可能是客户端底层或者RabbitMQ服务端在进行拆包，组包的时候，出现了明显的压力，而导致异常的发生。

# 超过4MB的消息，最好先进行分包。

# 第一次修改本节

# 由于用线程模拟大量发布者，且是服务器单节点，受客户端主机网卡的限制，发布线程没有速度控制，导致有大量数据发送，服务器带宽下行速率也满负荷，上行带宽却明显低于下行速率，导致服务器内存有大量消息堆积，进而触发RabbitMQ服务器paging操作，才出现了上述不稳定和订阅者断开现象。对发布端做适当流量控制，断开连接现象不再出现，但每秒消息数仍然不稳定。

# //管理策略

# // 声明了一个消息队列

# HashMap<String, Object> map = new HashMap<>();

# //标志队列中的消息存活时间，也就是说队列中的消息超过了指定时间会被删除(数字类型，标志时间，以豪秒为单位)

# map.put("x-message-ttl", 1 \* 24 \* 60 \* 60 \* 10000);

# //队列自身的空闲存活时间，当前的queue在指定的时间内，没有consumer、basic.get也就是未被访问，就会被删除。(数字类型，标志时间，以豪秒为单位)

# map.put("x-expires",60 \* 60 \* 10000);

# //最大长度和最大占用空间，设置了最大长度的队列，在超过了最大长度后进行插入会删除之前插入的消息为本次的留出空间（默认操作是如此，我们可以设置overflow来改变，例如用在并发缓冲时）,相应的最大占用大小也是这个道理，当超过了这个大小的时候，会删除之前插入的消息为本次的留出空间。

# map.put("x-max-length-bytes",100 \* 1024 \* 1024);

# // 设置队列最大的条数 10条

# map.put("x-max-length", 10);

# //Queue上附加优先级属性

# map.put("x-max-priority", 5);

# // 设置队列溢出方式 保留前10条

# //定义溢出行为-是从头上删除消息还是拒绝新发布

# ////队列超出最大长度的处理方案 ，队列溢出的默认处理方案：drop-head (default) 或者拒绝消息 reject-publish 我们做并发限流的时候需要设置为超出队列拒绝

# map.put("x-overflow", "reject-publish");

# 

# /\*

# 消息因为超时或超过限制在队列里消失，这样我们就丢失了一些消息，也许里面就有一些是我们做需要获知的。而rabbitmq的死信功能则为我们带来了解决方案。设置了dead letter exchange与dead letter routingkey（要么都设定，要么都不设定）那些因为超时或超出限制而被删除的消息会被推动到我们设置的exchange中，再根据routingkey推到queue中.

# \*/

# map.put("x-dead-letter-exchange","publicsubscrible");

# map.put("x-dead-letter-routing-key","publicsubscrible\_queue");

# 

# channel.queueDeclare(queueName, true, false, false, map);