1. **RabbitMQ 入门 Helloworld**

**1.1 介绍**

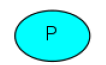
RabbitMQ 是信息传输的中间者。

本质上，从生产者（producers）接收消息，转发这些消息给消费者（consumers）.

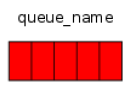
换句话说，能按指定规则进行消息转发、缓冲、和持久化。

常见术语：

* 一个发送消息的程序是一个producer(生产者)。下图表示：



* Queue（队列）类似邮箱。依存于RabbitMQ内部。虽然消息通过RabbitMQ在你的应用中传递，但是它们只能存储在queue中。队列不受任何限制，可存储任何数量的消息—本质上是一个无限制的缓存。很多producers可通过同一个队列发送消息，相同的很多consumers可从同一个队列上接收消息。下图表示：



* Consuming（消费）类似于接收。consumer基本属于等待接收消息的程序。下图表示:



注意：producer（生产者）,consumer（消费者）,broker（RabbitMQ服务）并不需要部署在同一台机器上，实际上在大多数实际的应用中，也不会部署在同一台机器上。



**2、Java入门实例**

一个producer发送消息，一个接收者接收消息，并在控制台打印出来。如下图：

注：需要在官网下载rabbitmq-java-client-bin-\*.zip将jar放入项目的classpath.

* **发送端**：Send.java 连接到RabbitMQ（此时服务需要启动），发送一条数据，然后退出。

|  |
| --- |
| package com.zhy.rabbit.\_01;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;    public class Send  {  //队列名称  private final static String QUEUE\_NAME = "hello";    public static void main(String[] argv) throws java.io.IOException  {  //创建连接连接到MabbitMQ  ConnectionFactory factory = new ConnectionFactory();  //设置MabbitMQ所在主机ip或主机名  factory.setHost("localhost");  //创建一个连接  Connection connection = factory.newConnection();  //创建一个频道  Channel channel = connection.createChannel();    //指定一个队列  channel.queueDeclare(QUEUE\_NAME, false, false, false, null);  //发送的消息  String message = "hello world!";  //往队列中发出一条消息  channel.basicPublish("", QUEUE\_NAME, null, message.getBytes());  System.out.println(" [x] Sent '" + message + "'");  //关闭频道和连接  channel.close();  connection.close();  }  } |

注意:队列只会在它不存在时创建，多次声明并不会重复创建。

信息的内容是字节数组，就意味着可传递任何数据。

* **接收端**：Recv.java 不断等待服务器推送消息，然后在控制台输出。

|  |
| --- |
| package com.zhy.rabbit.\_01;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class Recv  {  //队列名称  private final static String QUEUE\_NAME = "hello";    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  //打开连接和创建频道，与发送端一样  ConnectionFactory factory = new ConnectionFactory();    factory.setHost("localhost");  Connection connection = factory.newConnection();    Channel channel = connection.createChannel();    //声明队列，主要为了防止消息接收者先运行此程序，队列还不存在时创建队列。  channel.queueDeclare(QUEUE\_NAME, false, false, false, null);  System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");    //创建队列消费者  QueueingConsumer consumer = new QueueingConsumer(channel);  //指定消费队列  channel.basicConsume(QUEUE\_NAME, true, consumer);  while (true)  {  //nextDelivery是一个阻塞方法（内部实现其实是阻塞队列的take方法）  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());  System.out.println(" [x] Received '" + message + "'");  }    }  } |

分别运行Send.java和Recv.java 顺序无所谓。前提RabbitMQ服务开启。

运行结果：

|  |
| --- |
| [x]Sent 'hello world!'  ----------------------------------------  [\*] Waiting for messages. To exitpress CTRL+C  [x] Received 'hello world!' |

1. **RabbitMQ （二）工作队列 （使用的匿名转发器）**

上篇博客中写了通过一个命名的队列发送和接收消息。这篇中将会创建一个工作队列用来在工作者（consumer）间分发耗时任务。

工作队列的主要任务：避免立刻执行资源密集型任务，然后必须等待其完成。

相反地，进行任务调度：把任务封装为消息发送给队列。

工作进程在后台运行并不断的从队列中取出任务然后执行。

当你运行了多个工作进程时，任务队列中的任务将会被工作进程共享执行。

这样的概念在web应用中极其有用，当在很短的HTTP请求间需要执行复杂的任务。

**2.1、 准备**

使用Thread.sleep来模拟耗时任务。在发送到队列的消息的末尾添加一定数量的点，每个点代表在工作线程中需耗时1秒，例如hello…将会需要等待3秒。

* 发送端：

NewTask.Java

|  |
| --- |
| package com.zhy.rabbit.\_02\_workqueue;  import java.io.IOException;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  public class NewTask  {  //队列名称  private final static String QUEUE\_NAME = "workqueue";    public static void main(String[] args) throws IOException  {  //创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(QUEUE\_NAME, false, false, false, null);  //发送10条消息，依次在消息后面附加1-10个点  for (int i = 0; i < 10; i++)  {  String dots = "";  for (int j = 0; j <= i; j++)  {  dots += ".";  }  String message = "helloworld" + dots+dots.length();  channel.basicPublish("", QUEUE\_NAME, null, message.getBytes());  System.out.println(" [x] Sent '" + message + "'");  }  //关闭频道和资源  channel.close();  connection.close();  }  } |

**2.1.1 例子: 关闭了消息应答方式**

* 接收端：

Work.java

|  |
| --- |
| package com.zhy.rabbit.\_02\_workqueue;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class Work  {  //队列名称  private final static String QUEUE\_NAME = "workqueue";    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  //区分不同工作进程的输出  int hashCode = Work.class.hashCode();    //创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(QUEUE\_NAME, false, false, false, null);  System.out.println(hashCode + " [\*] Waiting for messages. To exit press CTRL+C");    QueueingConsumer consumer = new QueueingConsumer(channel);  // 指定消费队列  //【注：此处通过autoAsk=true关闭了消息应答方式】  channel.basicConsume(QUEUE\_NAME, true, consumer);  while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());    System.out.println(hashCode + " [x] Received '" + message + "'");  doWork(message);  System.out.println(hashCode + " [x] Done");  }  }    /\*\*  \* 每个点耗时1s  \* @param task  \* @throws InterruptedException  \*/  private static void doWork(String task) throws InterruptedException  {  for (char ch : task.toCharArray()) //返回字符数组，存放了当前字符串中的所有字符  {  if (ch == '.')  Thread.sleep(1000);  }  }  } |

**Round-robin 转发**

使用任务队列的好处是能够很容易的并行工作。

如果积压了很多工作，仅通过增加更多的工作者就可解决问题，使系统的伸缩性更容易。

下面先运行3个工作者（Work.java）实例，然后运行NewTask.java，3个工作者实例都会得到信息。但是如何分配呢？让我们来看输出结果：

[x] Sent 'helloworld.1'

[x] Sent 'helloworld..2'

[x] Sent 'helloworld...3'

[x] Sent 'helloworld....4'

[x] Sent 'helloworld.....5'

[x] Sent 'helloworld......6'

[x] Sent 'helloworld.......7'

[x] Sent 'helloworld........8'

[x] Sent 'helloworld.........9'

[x] Sent 'helloworld..........10'

工作者1：

605645 [\*] Waiting for messages. To exit press CTRL+C

605645 [x] Received 'helloworld.1'

605645 [x] Done

605645 [x] Received 'helloworld....4'

605645 [x] Done

605645 [x] Received 'helloworld.......7'

605645 [x] Done

605645 [x] Received 'helloworld..........10'

605645 [x] Done

工作者2：

18019860 [\*] Waiting for messages. To exit press CTRL+C

18019860 [x] Received 'helloworld..2'

18019860 [x] Done

18019860 [x] Received 'helloworld.....5'

18019860 [x] Done

18019860 [x] Received 'helloworld........8'

18019860 [x] Done

工作者3：

18019860 [\*] Waiting for messages. To exit press CTRL+C

18019860 [x] Received 'helloworld...3'

18019860 [x] Done

18019860 [x] Received 'helloworld......6'

18019860 [x] Done

18019860 [x] Received 'helloworld.........9'

18019860 [x] Done

可看到，默认的，RabbitMQ会一个一个的发送信息给下一个消费者(consumer)，而不考虑每个任务的时长等等，且是一次性分配，并非一个一个分配。

平均的每个消费者将会获得相等数量的消息。这样分发消息的方式叫做round-robin。

**2.2、 消息应答（message acknowledgments）**

执行一个任务需花费几秒钟。可能会担心当一个工作者在执行任务时发生中断。

上面代码，一旦RabbItMQ交付了一个信息给消费者，会马上从内存中移除该信息。

在这种情况下，如杀死正在执行任务的某个工作者，会丢失它正在处理的信息。

也会丢失已转发给这个工作者且它还未执行的消息。

上面例子，首先开启两个任务，然后执行发送任务的代码（NewTask.java），然后立即关闭第二个任务，结果为：

工作者2：

31054905 [\*] Waiting for messages. To exit press CTRL+C

31054905 [x] Received 'helloworld..2'

31054905 [x] Done

31054905 [x] Received 'helloworld....4'

工作者1：

18019860 [\*] Waiting for messages. To exit press CTRL+C

18019860 [x] Received 'helloworld.1'

18019860 [x] Done

18019860 [x] Received 'helloworld...3'

18019860 [x] Done

18019860 [x] Received 'helloworld.....5'

18019860 [x] Done

18019860 [x] Received 'helloworld.......7'

18019860 [x] Done

18019860 [x] Received 'helloworld.........9'

18019860 [x] Done

可看到，第二个工作者至少丢失了6，8，10号任务，且4号任务未完成。

但不希望丢失任何任务（信息）。当某个工作者（接收者）被杀死，希望将任务传递给另一个工作者。

为保证消息永远不丢失，RabbitMQ支持消息应答（message acknowledgments）。

消费者发送应答给RabbitMQ，告诉它信息已被接收和处理，然后RabbitMQ可自由的进行信息删除。

如消费者被杀死而没有发送应答，RabbitMQ会认为该信息没有被完全的处理，然后将会重新转发给别的消费者。通过这种方式，可确认信息不会被丢失，即使消者偶尔被杀死。

这种机制并没有超时时间这么一说，RabbitMQ只有在消费者连接断开时重新转发此信息。如果消费者处理一个信息需要耗费特别特别长的时间是允许的。

消息应答默认打开。

**2.2.1 开启消息应答**

【保证即使消费者被杀死，消息也不会被丢失，但如果RabbitMQ服务被停止，消息仍然会丢失。】

上面代码中通过显示的设置autoAsk=true关闭了这种机制。下面我们修改代码（Work.java）：

|  |
| --- |
| boolean ack = false ; //打开应答机制  channel.basicConsume(QUEUE\_NAME, ack, consumer);  //另外需要在每次处理完成一个消息后，手动发送一次应答。  channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false); |

完整修改后的Work.java

|  |
| --- |
| package com.zhy.rabbit.\_02\_workqueue.ack;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class Work  {  //队列名称  private final static String QUEUE\_NAME = "workqueue";    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  //区分不同工作进程的输出  int hashCode = Work.class.hashCode();  //创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(QUEUE\_NAME, false, false, false, null);  System.out.println(hashCode + " [\*] Waiting for messages. To exit press CTRL+C");  QueueingConsumer consumer = new QueueingConsumer(channel);  // 指定消费队列  boolean ack = false ; //打开应答机制  channel.basicConsume(QUEUE\_NAME, ack, consumer);  while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());    System.out.println(hashCode + " [x] Received '" + message + "'");  doWork(message);  System.out.println(hashCode + " [x] Done");  //发送应答  channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);  }  }  } |

测试：

把消息数量改为5，然后先打开两个消费者（Work.java），然后发送任务（NewTask.java），立即关闭一个消费者，观察输出：

[x] Sent 'helloworld.1'

[x] Sent 'helloworld..2'

[x] Sent 'helloworld...3'

[x] Sent 'helloworld....4'

[x] Sent 'helloworld.....5'

工作者2

18019860 [\*] Waiting for messages. To exit press CTRL+C

18019860 [x] Received 'helloworld..2'

18019860 [x] Done

18019860 [x] Received 'helloworld....4'

工作者1

31054905 [\*] Waiting for messages. To exit press CTRL+C

31054905 [x] Received 'helloworld.1'

31054905 [x] Done

31054905 [x] Received 'helloworld...3'

31054905 [x] Done

31054905 [x] Received 'helloworld.....5'

31054905 [x] Done

31054905 [x] Received 'helloworld....4'

31054905 [x] Done

可以看到工作者2没有完成的任务4，重新转发给工作者1完成了。

**3、 消息持久化（Message durability）**

当RabbitMQ退出或异常退出，将会丢失所有的队列和信息，除非告诉它不要丢失。

需做两件事来确保信息不会丢失：需要给所有的队列和消息设置持久化的标志。

第一， 需确认RabbitMQ永远不会丢失我们的**队列**。为了这样，需声明它为**持久化**的。

|  |
| --- |
| boolean durable = true;  channel.queueDeclare("task\_queue", durable, false, false, null); |

注：RabbitMQ不允许使用不同的参数重新定义一个队列，所以已存在的队列，无法修改其属性。

第二， 需要标识**信息**为**持久化**的。通过设置MessageProperties（implements BasicProperties）值为PERSISTENT\_TEXT\_PLAIN。

|  |
| --- |
| channel.basicPublish("","task\_queue",MessageProperties.PERSISTENT\_TEXT\_PLAIN,message.getBytes()); |

现在你可以执行一个发送消息的程序，然后关闭服务，再重新启动服务，运行消费者程序做下实验。

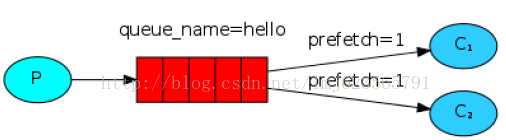
**4、公平转发（Fair dispatch）**

或许会发现，目前的消息转发机制（Round-robin）并非是我们想要的。例如，这样一种情况，对于两个消费者，有一系列的任务，奇数任务特别耗时，而偶数任务却很轻松，这样造成一个消费者一直繁忙，另一个消费者却很快执行完任务后等待。

造成这样的原因是因为RabbitMQ仅是当消息到达队列进行转发消息。并不在乎有多少任务消费者并未传递一个应答给RabbitMQ。仅仅盲目转发所有的奇数给一个消费者，偶数给另一个消费者。

为了解决这样的问题，可以使用basicQos方法，传递参数为prefetchCount = 1。这样告诉RabbitMQ不要在同一时间给一个消费者超过一条消息。换句话说，只有在消费者空闲的时候会发送下一条信息。

|  |
| --- |
| int prefetchCount = 1;  channel.basicQos(prefetchCount); |



注：如果所有的工作者都处于繁忙状态，你的队列有可能被填充满。你可能会观察队列的使用情况，然后增加工作者，或者使用别的什么策略。

测试：改变发送消息的代码，将消息末尾点数改为6-2个，然后首先开启两个工作者，接着发送消息：

[x] Sent 'helloworld......6'

[x] Sent 'helloworld.....5'

[x] Sent 'helloworld....4'

[x] Sent 'helloworld...3'

[x] Sent 'helloworld..2'

工作者1：

18019860 [\*] Waiting for messages. To exit press CTRL+C

18019860 [x] Received 'helloworld......6'

18019860 [x] Done

18019860 [x] Received 'helloworld...3'

18019860 [x] Done

工作者2：

31054905 [\*] Waiting for messages. To exit press CTRL+C

31054905 [x] Received 'helloworld.....5'

31054905 [x] Done

31054905 [x] Received 'helloworld....4'

31054905 [x] Done

31054905 [x] Received 'helloworld..2'

31054905 [x] Done

可以看出此时并没有按照之前的Round-robin机制进行转发消息，而是当消费者不忙时进行转发。且这种模式下支持动态增加消费者，因为消息并没有发送出去，动态增加了消费者马上投入工作。而默认的转发机制会造成，即使动态增加了消费者，此时的消息已经分配完毕，无法立即加入工作，即使有很多未完成的任务。

**5、完整的代码**

NewTask.java

|  |
| --- |
| package com.zhy.rabbit.\_02\_workqueue.ackandpersistence;  import java.io.IOException;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.MessageProperties;    public class NewTask  {  // 队列名称  private final static String QUEUE\_NAME = "workqueue\_persistence";    public static void main(String[] args) throws IOException  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  // 声明队列  boolean durable = true;// 1、设置队列持久化  channel.queueDeclare(QUEUE\_NAME, durable, false, false, null);  // 发送10条消息，依次在消息后面附加1-10个点  for (int i = 5; i > 0; i--)  {  String dots = "";  for (int j = 0; j <= i; j++)  {  dots += ".";  }  String message = "helloworld" + dots + dots.length();  // MessageProperties 2、设置消息持久化  channel.basicPublish("", QUEUE\_NAME,  MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());  System.out.println(" [x] Sent '" + message + "'");  }  // 关闭频道和资源  channel.close();  connection.close();  }  } |

Work.java

|  |
| --- |
| package com.zhy.rabbit.\_02\_workqueue.ackandpersistence;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;    public class Work  {  // 队列名称  private final static String QUEUE\_NAME = "workqueue\_persistence";    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  // 区分不同工作进程的输出  int hashCode = Work.class.hashCode();  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  // 声明队列  boolean durable = true;  channel.queueDeclare(QUEUE\_NAME, durable, false, false, null);  System.out.println(hashCode  + " [\*] Waiting for messages. To exit press CTRL+C");  //设置最大服务转发消息数量  int prefetchCount = 1;  channel.basicQos(prefetchCount);  QueueingConsumer consumer = new QueueingConsumer(channel);  // 指定消费队列  boolean ack = false; // 打开应答机制  channel.basicConsume(QUEUE\_NAME, ack, consumer);  while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());    System.out.println(hashCode + " [x] Received '" + message + "'");  doWork(message);  System.out.println(hashCode + " [x] Done");  //channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);  channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);  }  }    /\*\*  \* 每个点耗时1s  \* @param task  \* @throws InterruptedException  \*/  private static void doWork(String task) throws InterruptedException  {  for (char ch : task.toCharArray())  {  if (ch == '.')  Thread.sleep(1000);  }  }  } |

1. **【ok】RabbitMQ （三） 发布/订阅 [**fanout **](** 忽略routingKey**)**

上一篇中，实现了工作队列，且工作队列中的一个任务只会发给一个工作者，除非某个工作者未完成任务意外被杀死，会转发给另外的工作者。

这篇中，会做一些改变，就是把一个消息发给多个消费者，这种模式称为发布/订阅（类似观察者模式）。

为了验证这种模式，准备构建一个简单的日志系统。

系统包含两类程序，一类程序发动日志，另一类程序接收和处理日志。

在我们的日志系统中，每一个运行的接收者程序都会收到日志。然后实现，一个接收者将接收到的数据写到硬盘上，同时另一个接收者把接收到的消息展现在屏幕上。

本质上来说，就是发布的日志消息会转发给所有接收者。

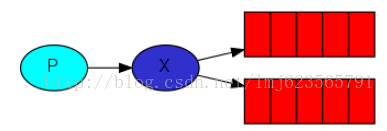
**3.1 转发器（Exchanges）**

前面主要介绍都是发送者发送消息给队列，接收者从队列接收消息。

下面会引入Exchanges，展示RabbitMQ的完整的消息模型。

RabbitMQ消息模型的核心理念是生产者永远不会直接发送任何消息给队列，一般生产者甚至不知道消息应该发送到哪些队列。生产者只能发送消息给转发器（Exchange）。

转发器一边接收从生产者发来的消息，另一边把消息推送到队列中。转发器必须清楚的知道消息如何处理它收到的每一条消息。是否应该追加到一个指定的队列？多个队列？或是否应该丢弃？这些规则通过转发器的类型进行定义。



下面列出一些可用的转发器类型：

Direct

Topic

Headers

Fanout

目前关注最后一个fanout，声明转发器类型的代码：

channel.exchangeDeclare("logs","fanout");

fanout类型转发器把所有接收到的消息，广播到所有它所知的队列。

不过这正是我们前述的日志系统所需要的。

**2、匿名转发器（nameless exchange）**

生产者只能发送消息给转发器（Exchange），但前两篇中的例子并没使用到转发器，仍可发送和接收消息。是因为使用了一个默认的转发器，它的标识符为””。之前发送消息的代码：

|  |
| --- |
| channel.basicPublish("", QUEUE\_NAME,MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes()); |

第一个参数为转发器的名称，我们设置为”” : 如果存在routingKey（第二个参数），消息由routingKey决定发送到哪个队列。

现在可指定消息发送到的转发器：

|  |
| --- |
| channel.basicPublish( "logs","", null, message.getBytes()); |

**3、临时队列（Temporary queues）**

前面都为队列指定了特定名称。

能为队列命名很关键，需指定消费者为某个队列。

当希望在生产者和消费者间共享队列时，为队列命名是很重要。

不过，对于我们的日志系统并不关心队列的名称。我们想要接收到所有的消息，且只对当前正在传递的数据感兴趣。为了满足我们的需求，需要做两件事：

第一， 无论何时连接到Rabbit我们都需要一个新的空的队列。为了实现，可使用随机数创建队列，或者更好的，让服务器给我们提供一个随机的名称。

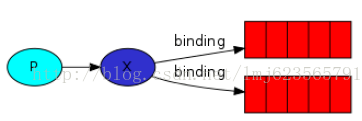
第二， 一旦消费者与Rabbit断开，消费者所接收的那个队列应被自动删除。

Java中可使用queueDeclare()方法，不传递任何参数，来创建一个非持久的、唯一的、自动删除的队列且队列名称由服务器随机产生。

|  |
| --- |
| String queueName = channel.queueDeclare().getQueue(); |

一般情况这个名称与amq.gen-JzTY20BRgKO-HjmUJj0wLg 类似。

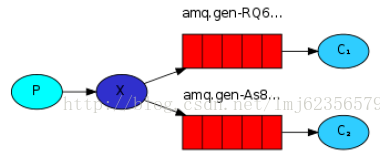
**4、绑定（Bindings）**



已创建了一个fanout转发器和队列，现需通过binding告诉转发器把消息发送给我们的队列。

|  |
| --- |
| channel.queueBind(queueName, “logs”, ””)参数1：队列名称 ；参数2：转发器名称 |

**5、完整的例子**



日志发送端：

|  |
| --- |
| package com.zhy.rabbit.\_03\_bindings\_exchanges;  import java.io.IOException;  import java.util.Date;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;    public class EmitLog  {  private final static String EXCHANGE\_NAME = "ex\_log";    public static void main(String[] args) throws IOException  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();    // 声明转发器和类型  channel.exchangeDeclare(EXCHANGE\_NAME, "fanout" );    String message = new Date().toLocaleString()+" : log something";  // 往转发器上发送消息  channel.basicPublish(EXCHANGE\_NAME, "", null, message.getBytes());    System.out.println(" [x] Sent '" + message + "'");    channel.close();  connection.close();  }  } |

没什么太大改变，声明队列的代码，改为声明转发器了，消息的传递也交给了转发器。

接收端1 :ReceiveLogsToSave.java：

|  |
| --- |
| package com.zhy.rabbit.\_03\_bindings\_exchanges;  import java.io.File;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.text.SimpleDateFormat;  import java.util.Date;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class ReceiveLogsToSave  {  private final static String EXCHANGE\_NAME = "ex\_log";    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();    channel.exchangeDeclare(EXCHANGE\_NAME, "fanout");  // 创建一个非持久的、唯一的且自动删除的队列  String queueName = channel.queueDeclare().getQueue();  // 为转发器指定队列，设置binding  channel.queueBind(queueName, EXCHANGE\_NAME, "");    System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");    QueueingConsumer consumer = new QueueingConsumer(channel);  // 指定接收者，第二个参数为自动应答，无需手动应答  channel.basicConsume(queueName, true, consumer);    while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());    print2File(message);  }    }    private static void print2File(String msg)  {  try  {  String dir = ReceiveLogsToSave.class.getClassLoader().getResource("").getPath();  String logFileName = new SimpleDateFormat("yyyy-MM-dd")  .format(new Date());  File file = new File(dir, logFileName+".txt");  FileOutputStream fos = new FileOutputStream(file, true);  fos.write((msg + "\r\n").getBytes());  fos.flush();  fos.close();  } catch (FileNotFoundException e)  {  e.printStackTrace();  } catch (IOException e)  {  e.printStackTrace();  }  }  } |

随机创建一个队列，将队列与转发器绑定，然后将消费者与该队列绑定，然后写入日志文件。

接收端2：ReceiveLogsToConsole.java

|  |
| --- |
| package com.zhy.rabbit.\_03\_bindings\_exchanges;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class ReceiveLogsToConsole  {  private final static String EXCHANGE\_NAME = "ex\_log";    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();    channel.exchangeDeclare(EXCHANGE\_NAME, "fanout");  // 创建一个非持久的、唯一的且自动删除的队列  String queueName = channel.queueDeclare().getQueue();  // 为转发器指定队列，设置binding  channel.queueBind(queueName, EXCHANGE\_NAME, "");    System.out.println(" [\*] Waiting for messages. To exit press CTRL+C");    QueueingConsumer consumer = new QueueingConsumer(channel);  // 指定接收者，第二个参数为自动应答，无需手动应答  channel.basicConsume(queueName, true, consumer);    while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());  System.out.println(" [x] Received '" + message + "'");  }  }    } |

随机创建一个队列，然后将队列与转发器绑定，然后将消费者与该队列绑定，然后打印到控制台。

现在把两个接收端运行，然后运行3次发送端：

输出结果：

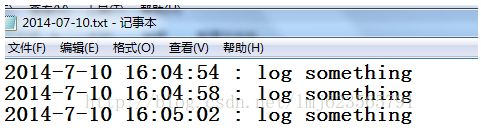
发送端：

[x] Sent '2014-7-10 16:04:54 : log something'

[x] Sent '2014-7-10 16:04:58 : log something'

[x] Sent '2014-7-10 16:05:02 : log something'

接收端1：



接收端2：

[\*] Waiting for messages. To exit press CTRL+C

[x] Received '2014-7-10 16:04:54 : log something'

[x] Received '2014-7-10 16:04:58 : log something'

[x] Received '2014-7-10 16:05:02 : log something'

这个例子实现了文章开头所描述的日志系统，利用了转发器的类型：fanout。

本篇说明了，生产者将消息发送至转发器，转发器决定将消息发送至哪些队列，消费者绑定队列获取消息。

1. **【ok】RabbitMQ （四） 路由选择 (Routing) [direct]**

上一篇建立了简单的日志系统，能够广播日志消息给所有你的接收者。

本篇准备给日志系统添加新的特性，让日志接收者能够订阅部分消息。例如，可仅仅将致命的错误写入日志文件，仍然在控制面板上打印出所有的其他类型的日志消息。

**4.1 绑定（Bindings）**

在上一篇中已使用过绑定。类似下面的代码：

channel.queueBind(queueName, EXCHANGE\_NAME, "");

绑定表转发器与队列间关系。也可简单认为：队列对该转发器上的消息感兴趣。

绑定可附带一个额外参数routingKey。

为避免与basicPublish方法（发布消息的方法）参数混淆，准备把它称作绑定键（binding key）。

下面展示如何使用绑定键（binding key）来创建一个绑定：

|  |
| --- |
| channel.queueBind(queueName, EXCHANGE\_NAME, "black"); |

绑定键的意义依赖于转发器的类型。对于fanout类型，忽略此参数。

**4.2 直接转发（Direct exchange）**

上一篇的日志系统广播所有的消息给所有的消费者。

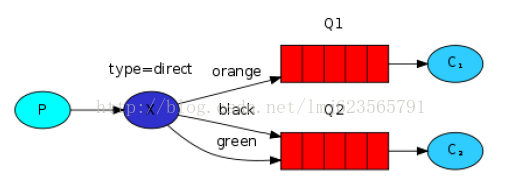
希望可对其扩展，来允许根据日志的严重性进行过滤日志。例如：可能希望把致命类型的错误写入硬盘，而不把硬盘空间浪费在警告或消息类型的日志上。

之前使用fanout类型的转发器，但并没有带来更多的灵活性：仅可愚蠢的转发。

将使用direct类型的转发器进行替代。

direct类型的转发器背后的路由转发算法很简单：消息会被推送至绑定键（binding key）和消息发布附带的选择键（routing key）完全匹配的队列。

图解：



上图，可看到direct类型的转发器与两个队列绑定。

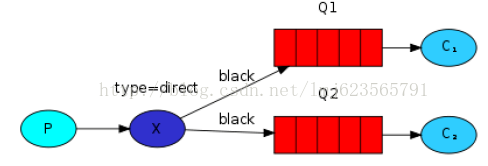
第一个队列与绑定键orange绑定，

第二个队列与转发器间有两个绑定，一个与绑定键black绑定，另一个与green绑定键绑定。

这样的话，当一个消息附带一个选择键（routing key） orange发布至转发器将会被导向到队列Q1。

消息附带一个选择键（routing key）black或者green将会被导向到Q2.所有的其他的消息将会被丢弃。

**4.3、多重绑定（multiple bindings）**



使用一个绑定键（binding key）绑定多个队列是完全合法的。

如上图，一个附带选择键（routing key）的消息将会被转发到Q1和Q2。

**4.4 、发送日志（Emittinglogs）**

准备将这种模式用于我们的日志系统。将消息发送到direct类型的转发器而不是fanout类型。将把日志的严重性作为选择键（routing key）。这样的话，接收程序可以根据严重性来选择接收。我们首先关注发送日志的代码：

像以前一样，需先创建一个转发器：

|  |
| --- |
| channel.exchangeDeclare(EXCHANGE\_NAME,"direct"); |

然后我们准备发送一条消息：

|  |
| --- |
| channel.basicPublish(EXCHANGE\_NAME,severity, null, message.getBytes()); |

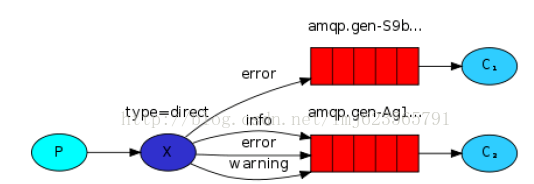
为了简化代码，我们假定‘severity’是‘info’，‘warning’，‘error’中的一个。

**4.5、订阅**

接收消息的代码和前面的博客的中类似，只有一点不同：给我们所感兴趣的严重性类型的日志创建一个绑定。

|  |
| --- |
| String queueName = channel.queueDeclare().getQueue();  for(String severity : argv)  {  channel.queueBind(queueName, EXCHANGE\_NAME, severity);  } |

**4.6、完整实例**



发送端：EmitLogDirect.java

|  |
| --- |
| package com.zhy.rabbit.\_04\_binding\_key;  import java.util.Random;  import java.util.UUID;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  public class EmitLogDirect  {  private static final String EXCHANGE\_NAME = "ex\_logs\_direct";  private static final String[] SEVERITIES = { "info", "warning", "error" };    public static void main(String[] argv) throws java.io.IOException  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  // 声明转发器的类型  channel.exchangeDeclare(EXCHANGE\_NAME, "direct");    //发送6条消息  for (int i = 0; i < 6; i++)  {  String severity = getSeverity();  String message = severity + "\_log :" + UUID.randomUUID().toString();  // 发布消息至转发器，指定routingkey  channel.basicPublish(EXCHANGE\_NAME, severity, null, message.getBytes());  System.out.println(" [x] Sent '" + message + "'");  }    channel.close();  connection.close();  }    /\*\*  \* 随机产生一种日志类型  \* @return  \*/  private static String getSeverity()  {  Random random = new Random();  int ranVal = random.nextInt(3);  return SEVERITIES[ranVal];  }  } |

随机发送6条随机类型（routing key）的日志给转发器~~

接收端：ReceiveLogsDirect.java

|  |
| --- |
| package com.zhy.rabbit.\_04\_binding\_key;  import java.util.Random;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class ReceiveLogsDirect  {    private static final String EXCHANGE\_NAME = "ex\_logs\_direct";  private static final String[] SEVERITIES = { "info", "warning", "error" };    public static void main(String[] argv) throws java.io.IOException,  java.lang.InterruptedException  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  // 声明direct类型转发器  channel.exchangeDeclare(EXCHANGE\_NAME, "direct");    String queueName = channel.queueDeclare().getQueue();  String severity = getSeverity();  // 指定binding\_key  channel.queueBind(queueName, EXCHANGE\_NAME, severity);  System.out.println(" [\*] Waiting for "+severity+" logs. To exit press CTRL+C");    QueueingConsumer consumer = new QueueingConsumer(channel);  channel.basicConsume(queueName, true, consumer);    while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());    System.out.println(" [x] Received '" + message + "'");  }  }    /\*\*  \* 随机产生一种日志类型  \* @return  \*/  private static String getSeverity()  {  Random random = new Random();  int ranVal = random.nextInt(3);  return SEVERITIES[ranVal];  }  } |

接收端随机设置一个日志严重级别（binding\_key）。。。

开启了3个接收端程序，两个准备接收error类型日志，一个接收info类型日志，然后运行发送端程序

运行结果：

[x] Sent 'error\_log :d142b096-46c0-4380-a1d2-d8b2ac136a9c'

[x] Sent 'error\_log :55ee1fc4-c87c-4e5e-81ba-49433890b9ce'

[x] Sent 'error\_log :d01877d6-87c7-4e0a-a109-697d122bc4c9'

[x] Sent 'error\_log :b42471b1-875c-43f1-b1ea-0dd5b49863f3'

[x] Sent 'info\_log :a6c1bc87-efb0-43eb-8314-8a74c345ed05'

[x] Sent 'info\_log :b6a84b6a-353e-4e88-8c23-c791d93b44be'

------------------------------------------------------------------------------------

[\*] Waiting for error logs. To exit press CTRL+C

[x] Received 'error\_log :d142b096-46c0-4380-a1d2-d8b2ac136a9c'

[x] Received 'error\_log :55ee1fc4-c87c-4e5e-81ba-49433890b9ce'

[x] Received 'error\_log :d01877d6-87c7-4e0a-a109-697d122bc4c9'

[x] Received 'error\_log :b42471b1-875c-43f1-b1ea-0dd5b49863f3'

------------------------------------------------------------------------------------

[\*] Waiting for error logs. To exit press CTRL+C

[x] Received 'error\_log :d142b096-46c0-4380-a1d2-d8b2ac136a9c'

[x] Received 'error\_log :55ee1fc4-c87c-4e5e-81ba-49433890b9ce'

[x] Received 'error\_log :d01877d6-87c7-4e0a-a109-697d122bc4c9'

[x] Received 'error\_log :b42471b1-875c-43f1-b1ea-0dd5b49863f3'

------------------------------------------------------------------------------------

[\*] Waiting for info logs. To exit press CTRL+C

[x] Received 'info\_log :a6c1bc87-efb0-43eb-8314-8a74c345ed05'

[x] Received 'info\_log :b6a84b6a-353e-4e88-8c23-c791d93b44be'

可看到我们实现了博文开头所描述的特性，接收者可自定义感兴趣类型的日志。

其实文章这么长就在说：发送消息时可以设置routing\_key，接收队列与转发器间可以设置binding\_key，接收者接收与binding\_key与routing\_key相同的消息。

1. **【ok】RabbitMQ （五）主题（Topic）**

上一篇中，改良了日志系统。使用direct类型转发器，使得接收者有能力进行选择性的接收日志,，而非fanout那样，只能够无脑的转发

虽使用direct类型改良了系统，但仍存在一些局限性：它不能够基于多重条件进行路由选择。

在我们的日志系统中，有可能希望不仅根据日志的级别且想根据日志的来源进行订阅。

这个概念类似unix工具：syslog，它转发日志基于严重性（info/warning/crit…）和设备（auth/cron/kern…）这样可能给我们更多的灵活性：可能只想订阅来自’cron’的致命错误日志，而不是来自’kern’的。

为了在系统中实现上述的需求，需要学习稍微复杂的主题类型的转发器（topic exchange）。

**5.1 主题转发（Topic Exchange）**

发往主题类型的转发器的消息不能随意的设置选择键（routing\_key），必须是由点隔开的一系列的标识符组成。

标识符可是任何东西，但一般都与消息的某些特性相关。

一些合法的选择键的例子："stock.usd.nyse", "nyse.vmw","quick.orange.rabbit".

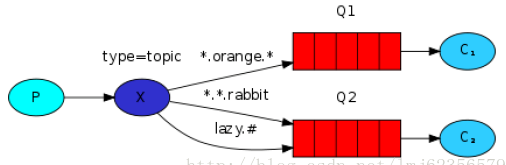
可定义任何数量的标识符，上限为255个字节。

绑定键和选择键的形式一样。

主题类型的转发器背后的逻辑和直接类型的转发器很类似：一个附带特殊的选择键将会被转发到绑定键与之匹配的队列中。需注意：关于绑定键有两种特殊的情况。

|  |
| --- |
| \*可以匹配一个标识符。  #可以匹配0或多个标识符。 |

**5.2 图解：**



准备发送关于动物的消息。

消息会附加一个选择键包含3个标识符（两个点隔开）。

第一个标识符描述动物的速度，

第二个标识符描述动物的颜色，

第三个标识符描述动物的物种：

<speed>.<color>.<species>。

创建3个绑定键：Q1与\*.orange.\*绑定Q2与\*.\*.rabbit和lazy.#绑定。

可简单的认为:

Q1对所有的橙色动物感兴趣。

Q2想要知道关于兔子的一切以及关于懒洋洋的动物的一切。

一个附带quick.orange.rabbit的选择键的消息将会被转发到两个队列。

附带lazy.orange.elephant的消息也会被转发到两个队列。

另一方面quick.orange.fox只会被转发到Q1

lazy.brown.fox将会被转发到Q2。

lazy.pink.rabbit虽然与两个绑定键匹配，但是也只会被转发到Q2一次。

quick.brown.fox不能与任何绑定键匹配，所以会被丢弃。

如违反约定，发送一个或四个标识符的选择键，类似：orange，quick.orange.male.rabbit，这些选择键不能与任何绑定键匹配，所以消息将会被丢弃。

另一方面，lazy.orange.male.rabbit，虽是四个标识符，也可与lazy.#匹配，从而转发至Q2。

注：主题类型的转发器非常强大，可实现其他类型的转发器。

当一个队列与绑定键#绑定，将会收到所有消息，类似fanout类型转发器。

当绑定键中不包含任何#与\*时，类似direct类型转发器。

**5.3 完整的例子**

发送端EmitLogTopic.Java：

|  |
| --- |
| package com.zhy.rabbit.\_05\_topic\_exchange;  import java.util.UUID;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  public class EmitLogTopic  {  private static final String EXCHANGE\_NAME = "topic\_logs";  public static void main(String[] argv) throws Exception  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  channel.exchangeDeclare(EXCHANGE\_NAME, "topic");  String[] routing\_keys = new String[] { "kernal.info", "cron.warning",  "auth.info", "kernel.critical" };  for (String routing\_key : routing\_keys)  {  String msg = UUID.randomUUID().toString();  channel.basicPublish(EXCHANGE\_NAME, routing\_key, null, msg .getBytes());  System.out.println(" [x] Sent routingKey = "+routing\_key+" ,msg = " + msg + ".");  }  channel.close();  connection.close();  }  } |

我们发送了4条消息，分别设置了不同的选择键。

接收端1，ReceiveLogsTopicForKernel.java

|  |
| --- |
| package com.zhy.rabbit.\_05\_topic\_exchange;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class ReceiveLogsTopicForKernel  {  private static final String EXCHANGE\_NAME = "topic\_logs";    public static void main(String[] argv) throws Exception  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  // 声明转发器  channel.exchangeDeclare(EXCHANGE\_NAME, "topic");  // 随机生成一个队列  String queueName = channel.queueDeclare().getQueue();    //接收所有与kernel相关的消息  channel.queueBind(queueName, EXCHANGE\_NAME, "kernel.\*");  System.out.println(" [\*] Waiting for messages about kernel. To exit press CTRL+C");    QueueingConsumer consumer = new QueueingConsumer(channel);  channel.basicConsume(queueName, true, consumer);    while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());  String routingKey = delivery.getEnvelope().getRoutingKey();    System.out.println(" [x] Received routingKey = " + routingKey  + ",msg = " + message + ".");  }  }  } |

直接收和Kernel相关的日志消息。

接收端2，ReceiveLogsTopicForCritical.java

|  |
| --- |
| package com.zhy.rabbit.\_05\_topic\_exchange;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Connection;  import com.rabbitmq.client.ConnectionFactory;  import com.rabbitmq.client.QueueingConsumer;  public class ReceiveLogsTopicForCritical  {  private static final String EXCHANGE\_NAME = "topic\_logs";    public static void main(String[] argv) throws Exception  {  // 创建连接和频道  ConnectionFactory factory = new ConnectionFactory();  factory.setHost("localhost");  Connection connection = factory.newConnection();  Channel channel = connection.createChannel();  // 声明转发器  channel.exchangeDeclare(EXCHANGE\_NAME, "topic");  // 随机生成一个队列  String queueName = channel.queueDeclare().getQueue();    // 接收所有与kernel相关的消息  channel.queueBind(queueName, EXCHANGE\_NAME, "\*.critical");    System.out .println(" [\*] Waiting for critical messages. To exit press CTRL+C");    QueueingConsumer consumer = new QueueingConsumer(channel);  channel.basicConsume(queueName, true, consumer);    while (true)  {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = new String(delivery.getBody());  String routingKey = delivery.getEnvelope().getRoutingKey();    System.out.println(" [x] Received routingKey = " + routingKey  + ",msg = " + message + ".");  }  }  } |

只接收致命错误的日志消息。

运行结果：

[x] Sent routingKey = kernal.info ,msg = a7261f0d-18cc-4c85-ba80-5ecd9283dae7.

[x] Sent routingKey = cron.warning ,msg = 0c7e4484-66e0-4846-a869-a7a266e16281.

[x] Sent routingKey = auth.info ,msg = 3273f21f-6e6e-42f2-83df-1f2fafa7a19a.

[x] Sent routingKey = kernel.critical ,msg = f65d3e1a-0619-4f85-8b0d-59375380ecc9.

--------------------------------------------------------------------------------------------------------------------

[\*] Waiting for messages about kernel. To exit press CTRL+C

[x] Received routingKey = kernel.critical,msg = f65d3e1a-0619-4f85-8b0d-59375380ecc9.

--------------------------------------------------------------------------------------------------------------------

[\*] Waiting for critical messages. To exit press CTRL+C

[x] Received routingKey = kernel.critical,msg = f65d3e1a-0619-4f85-8b0d-59375380ecc9.

可看到，通过使用topic类型的转发器，成功实现了多重条件选择的订阅。