

# 轻松搞定Rabbi tMQ

崔成龙



# 目 录

前言

轻松搞定RabbitMQ (一) —— RabbitMQ基础知识+HelloWorld

轻松搞定RabbitMQ (二) —— 工作队列之消息分发机制

轻松搞定RabbitMQ (三) —— 消息应答与消息持久化

轻松搞定RabbitMQ (四) —— 发布/订阅

轻松搞定RabbitMQ (五) —— 路由选择

轻松搞定RabbitMQ (六) —— 主题

轻松搞定RabbitMQ (七) —— 远程过程调用RPC

# 前言

---

原文出处：[轻松搞定RabbitMQ](#)

作者：[崔成龙](#)

本系列文章经作者授权在看云整理发布，未经作者允许，请勿转载！

## 轻松搞定RabbitMQ

轻松搞定RabbitMQ，翻译的官网教程，添加了自己的理解。

# 轻松搞定RabbitMQ (一) —— RabbitMQ基础知识+HelloWorld

本文是简单介绍一下RabbitMQ，参考官网上的教程。同时加入了一些自己的理解。官网教程详见：["Hello World!"](#)。

## 引言

你是否遇到过多个系统间需要通过定时任务来同步某些数据？

你是否在为异构系统的不同进程间相互调用、通讯的问题而苦恼、挣扎？

如果是，那么恭喜你，消息服务让你可以很轻松地解决这些问题。消息服务擅长于解决多系统、异构系统间的数据交换（消息通知/通讯）问题。

本文将要介绍的RabbitMQ就是当前最主流的消息中间件之一。

## RabbitMQ简介

MQ（Message Queue，消息队列）是一种应用系统之间的通信方法。是通过读写出入队列的消息来通信（RPC则是通过直接调用彼此来通信的）。

AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。

RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

下面通过生产者代码来解释一下RabbitMQ中涉及到的概念。

```
public class MsgSender {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] args) throws IOException {
        /**
         * 创建连接连接到MabbitMQ
        */
    }
}
```

```

        */
        ConnectionFactory factory = new ConnectionFactory();
        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 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!龙轩";
        // 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();
    }
}

```

## ConnectionFactory、Connection、Channel

ConnectionFactory、Connection、Channel，这三个都是RabbitMQ对外提供的API中最基本的对象。不管是服务器端还是客户端都会首先创建这三类对象。

ConnectionFactory为Connection的制造工厂。

Connection是与RabbitMQ服务器的socket链接，它封装了socket协议及身份验证相关部分逻辑。

Channel是我们与RabbitMQ打交道的最重要的一个接口，大部分的业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

## Queue

Queue（队列）是RabbitMQ的内部对象，用于存储消息，用下图表示。

queue\_name



RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。



队列是有Channel声明的，而且这个操作是**幂等**的。同名的队列多次声明也只会创建一次。我们发送消息就是想这个声明的队列里发送消息。

看一下消费者的代码：

```

public class MsgReceiver {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv) throws IOException, Interr
uptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道，与发送端一样
        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, nul
l);
        // 创建队列消费者
        QueueingConsumer consumer = new QueueingConsumer(channel)
;
        System.out.println(" [*] Waiting for messages. To exit pr
ess CTRL+C");
        // 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t
ake方法）
            QueueingConsumer.Delivery delivery = consumer.nex
tDelivery();
            String message = new String(delivery.getBody());
            System.out.println(" [x] Received '" + message +
"'");
        }
    }
}
  
```

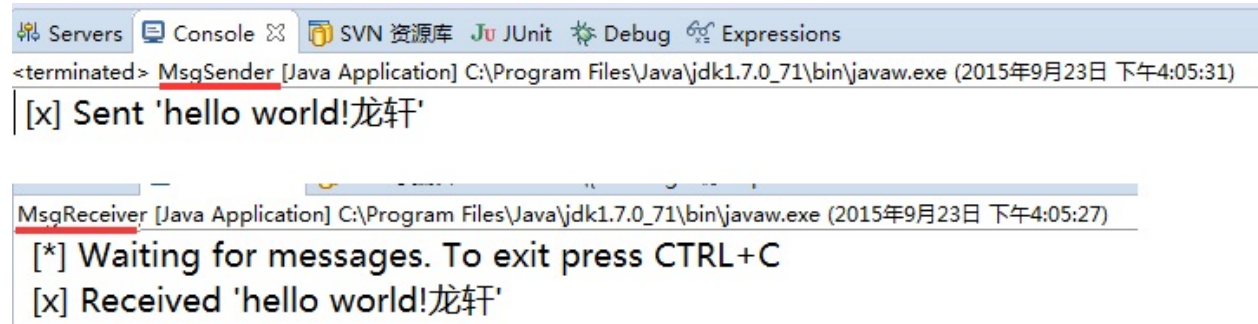
从上述代码中，我们可以看到ConnectionFactory、Connection、Channel这三个对象都还是会创建。而队列在消费者这里又声明了一遍。这是为了防止先启动消费者，当为消费者指定队列时，如果RabbitMQ服务器上未声明过队列，就会抛出IO异常。

## QueueingConsumer

队列消费者，用于监听队列中的消息。调用nextDelivery方法时，内部实现就是调用队列

的take方法。该方法的作用：获取并移除此队列的头部，在元素变得可用之前一直等待（如果有必要）。说白了就是如果没有消息，就处于阻塞状态。

运行结果如下：（生产者、消费者谁先运行都可以）



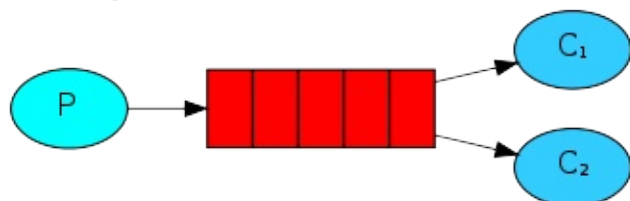
```
Servers Console SVN 资源库 JUnit Debug Expressions
<terminated> MsgSender [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\javaw.exe (2015年9月23日 下午4:05:31)
[x] Sent 'hello world!龙轩'

MsgReceiver [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\javaw.exe (2015年9月23日 下午4:05:27)
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'hello world!龙轩'
```

# 轻松搞定RabbitMQ（二）——工作队列之消息分发机制

上一篇博文中简单介绍了一下RabbitMQ的基础知识，并写了一个经典语言入门程序——HelloWorld。本篇博文中我们将会创建一个工作队列用来在工作者（consumer）间分发耗时任务。同样是翻译的[官网实例](#)。

## 工作队列



在[前一篇博文](#)中，我们完成了一个简单的对声明的队列进行发送和接受消息程序。下面我们将创建一个工作队列，来向多个工作者（consumer）分发耗时任务。

工作队列（又名：任务队列）的主要任务是为了避免立即做一个资源密集型的却又必须等待完成的任务。相反的，我们进行任务调度：将任务封装为消息并发给队列。在后台运行的工作者（consumer）将其取出，然后最终执行。当你运行多个工作者（consumer），队列中的任务被工作进行共享执行。

这样的概念对于在一个HTTP短链接的请求窗口中处理复杂任务的web应用程序，是非常有用的。

## 准备

使用Thread.Sleep()方法来模拟耗时。采用小数点的数量来表示任务的复杂性。每一个点将住哪用1s的“工作”。例如，Hello... 处理完需要3s的时间。

发送端（生产者）：NewTask.java

```
public class NewTask {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] args) throws IOException {
        /**
         * 创建连接连接到MabbitMQ
         */
        ConnectionFactory factory = new ConnectionFactory();
```



```

        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 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(QueueName, 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("", QueueName, null, message.getBytes());

        System.out.println(" [x] Sent '" + message + "'");
        // 关闭频道和连接
        channel.close();
        connection.close();
    }
}

```

### 工作者 ( 消费者 ) Worker.java

```

public class Worker {
    private final static String QueueName = "hello";

    public static void main(String[] argv) throws IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道, 与发送端一样
        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(QueueName, false, false, false, null);

        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
                Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");

                System.out.println(" [x] Received '" + message + "'");
                System.out.println(" [x] Processing... at " + new Date().toLocaleString());
            }
        };
    }
}

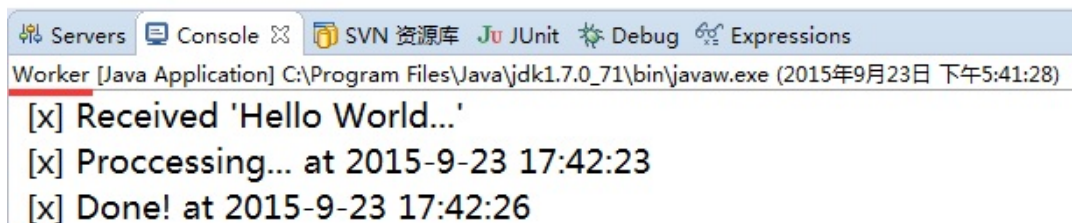
```

```

        try {
            for (char ch: message.toCharArray()) {
                if (ch == '.') {
                    Thread.sleep(1000);
                }
            }
        } catch (InterruptedException e) {
        } finally {
            System.out.println(" [x] Done! at " + new Date().toLocaleString());
        }
    }
};
channel.basicConsume(QUEUE_NAME, true, consumer);
}

```

运行结果如下：

## 任务分发机制

正主来了。。。下面开始介绍各种任务分发机制。

### Round-robin (轮询分发)

使用任务队列的优点之一就是可以轻易的并行工作。如果我们积压了好多工作，我们可以通过增加工作者（消费者）来解决这一问题，使得系统的伸缩性更加容易。

修改一下NewTask，使用for循环模拟多次发送消息的过程：

```

        for (int i = 0; i < 5; i++) {
            // 发送的消息
            String message = "Hello World"+Strings.repeat(".",
            , i);

            // 往队列中发出一条消息
            channel.basicPublish("", QUEUE_NAME, null, message);

            e.getBytes());
            System.out.println(" [x] Sent '" + message + "'");
        }
    }
};
channel.basicConsume(QUEUE_NAME, true, consumer);
}

```

```
;
}
```

我们先启动1个生产者实例，2个工作者实例，看一下如何执行：

```

[terminated] NewTask [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\javaw.exe (2015年9月24日 上午9:44:21)
[x] Sent 'Hello World'
[x] Sent 'Hello World.'
[x] Sent 'Hello World..'
[x] Sent 'Hello World...'
[x] Sent 'Hello World....'

Worker [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\javaw.exe (2015年9月24日 上午9:44:14)
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Hello World' 0个小数点
[x] Processing... at 2015-9-24 9:44:21
[x] Done! at 2015-9-24 9:44:21
[x] Received 'Hello World..' 2个小数点
[x] Processing... at 2015-9-24 9:44:21
[x] Done! at 2015-9-24 9:44:23
[x] Received 'Hello World...' 4个小数点
[x] Processing... at 2015-9-24 9:44:23
[x] Done! at 2015-9-24 9:44:27

Worker [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\javaw.exe (2015年9月24日 上午9:44:17)
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Hello World.' 1个小数点
[x] Processing... at 2015-9-24 9:44:21
[x] Done! at 2015-9-24 9:44:22
[x] Received 'Hello World...' 3个小数点
[x] Processing... at 2015-9-24 9:44:22
[x] Done! at 2015-9-24 9:44:25

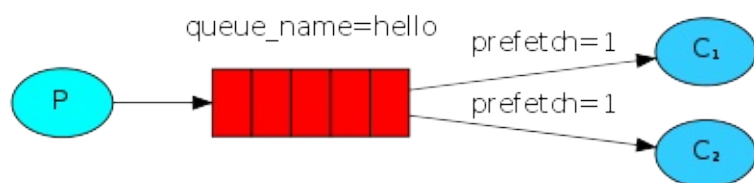
```

从上述的结果中，我们可以得知，在默认情况下，RabbitMQ将逐个发送消息到在序列中的下一个消费者(而不考虑每个任务的时长等等，且是提前一次性分配，并非一个一个分配)。平均每个消费者获得相同数量的消息。这种方式分发消息机制称为Round-Robin (轮询)。

## Fair dispatch (公平分发)

您可能已经注意到，任务分发仍然没有完全按照我们想要的那样。比如：现在有2个消费者，所有的奇数的消息都是繁忙的，而偶数则是轻松的。按照轮询的方式，奇数的任务交给了第一个消费者，所以一直在忙个不停。偶数的任务交给另一个消费者，则立即完成任务，然后闲得不行。而RabbitMQ则是不了解这些的。

这是因为当消息进入队列，RabbitMQ就会分派消息。它不看消费者为应答的数目，只是盲目的将第n条消息发给第n个消费者。



为了解决这个问题，我们使用basicQos( prefetchCount = 1)方法，来限制RabbitMQ只发不超过1条的消息给同一个消费者。当消息处理完毕后，有了反馈，才会进行第二次发送。

```
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。

还有一点需要注意，使用公平分发，必须关闭自动应答，改为手动应答。这些内容会在下篇博文中讲述。

整体代码如下：生产者NewTask.java

```
public class NewTask {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] args) throws IOException {
        /**
         * 创建连接连接到MabbitMQ
         */
        ConnectionFactory factory = new ConnectionFactory();
        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 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);

        int prefetchCount = 1;
        //限制发给同一个消费者不得超过1条消息
        channel.basicQos(prefetchCount);
        for (int i = 0; i < 5; i++) {
            // 发送的消息
            String message = "Hello World"+Strings.repeat(".",
,5-i)+(5-i);

            // 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();
    }
}
```

消费者Worker.java

```

public class Worker {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv) throws IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道, 与发送端一样
        Connection connection = factory.newConnection();
        final 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");

        channel.basicQos(1); // 保证一次只分发一个
        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
            Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");

                System.out.println(" [x] Received '" + message + "'");

                try {
                    for (char ch: message.toCharArray()) {
                        if (ch == '.') {
                            Thread.sleep(1000);
                        }
                    }
                } catch (InterruptedException e) {
                } finally {
                    System.out.println(" [x] Done! at " + new Date().toLocaleString());
                    channel.basicAck(envelope.getDeliveryTag(), false);
                }
            }
        };

        channel.basicConsume(QUEUE_NAME, false, consumer)
    ;
    }
}

```

运行结果如下：

ServersConsoleSVN 资源库JUnitDebugExpressions

<terminated> NewTask [Java Application] C:\Program Files\Java\jdk1.7.0\_71\bin\javaw.exe

[X] Sent 'Hello World.....5' ①

[X] Sent 'Hello World....4' ②

[X] Sent 'Hello World...3' ③ 生产者依次发送消息

[X] Sent 'Hello World..2' ④

[X] Sent 'Hello World.1' ⑤

ServersConsoleSVN 资源库JUnitDebugExpressions

Worker [Java Application] C:\Program Files\Java\jdk1.7.0\_71\bin\javaw.exe (2015年9月2

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

[X] Received 'Hello World.....5' 任务①

[X] Proccessing at 2015-9-24 18:22:06 需要执行5s

[X] Done! at 2015-9-24 18:22:12

[X] Received 'Hello World..2' 任务④

[X] Proccessing at 2015-9-24 18:22:12 当前消费者1空闲

[X] Done! at 2015-9-24 18:22:14 交由消费者1执行任务④

[X] Received 'Hello World.1' 任务⑤

[X] Proccessing at 2015-9-24 18:22:14 当前消费者1空闲

[X] Done! at 2015-9-24 18:22:15 交由消费者1执行任务⑤

消费者1

ServersConsoleSVN 资源库JUnitDebugExpressions

Worker [Java Application] C:\Program Files\Java\jdk1.7.0\_71\bin\javaw.exe (2015年9月2

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

[X] Received 'Hello World....4' 任务②

[X] Proccessing at 2015-9-24 18:22:06 需要执行4s

[X] Done! at 2015-9-24 18:22:11

[X] Received 'Hello World...3' 任务③

[X] Proccessing at 2015-9-24 18:22:11 当前消费者2空闲

[X] Done! at 2015-9-24 18:22:14 所以由其执行任务③

消费者2



# 轻松搞定RabbitMQ ( 三 ) ——消息应答与消息持久化

这个官网的[第二个例子](#)中的消息应答和消息持久化部分。我把它摘出来作为单独的一块儿来分享。

## Message acknowledgment ( 消息应答 )

执行一个任务可能需要花费几秒钟，你可能会担心如果一个消费者在执行任务过程中挂掉了。基于现在的代码，一旦RabbitMQ将消息分发给消费者，就会从内存中删除。在这种情况下，如果杀死正在执行任务的消费者，会丢失正在处理的消息，也会丢失已经分发给这个消费者但尚未处理的消息。

但是，我们不想丢失任何任务，如果有一个消费者挂掉了，那么我们应该将分发给它的任务交付给另一个消费者去处理。

为了确保消息不会丢失，RabbitMQ支持消息应答。消费者发送一个消息应答，告诉RabbitMQ这个消息已经接收并且处理完毕了。RabbitMQ可以删除它了。

如果一个消费者挂掉却没有发送应答，RabbitMQ会理解为这个消息没有处理完全，然后交给另一个消费者去重新处理。这样，你就可以确认即使消费者偶尔挂掉也不会丢失任何消息了。

没有任何消息超时限制；只有当消费者挂掉时，RabbitMQ才会重新投递。即使处理一条消息会花费很长的时间。

消息应答是默认打开的。我们明确地把它们关掉了 ( `autoAck=true` )。现在将应答打开，一旦我们完成任务，消费者会自动发送消息应答。

```
boolean autoAck = false;
channel.basicConsume(QUEUE_NAME, autoAck, consumer);
```

修改一下Worker.java

```
channel.basicQos(1); // 保证一次只分发一个
// 创建队列消费者
final Consumer consumer = new DefaultConsumer(channel) {
    @Override
```

```

        public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
            String message = new String(body, "UTF-8");

            System.out.println(" [x] Received '" + message + "'");

            System.out.println(" [x] Proccessing... at " + new Date().toLocaleString());
            try {
                for (char ch: message.toCharArray()) {
                    if (ch == '.') {
                        Thread.sleep(1000);
                    }
                }
            } catch (InterruptedException e) {
            } finally {
                System.out.println(" [x] Done! at " + new Date().toLocaleString());
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        }
    }
};

```

我们还是运行1个生产者，2个消费者，在消息处理过程中，人为让一个消费者挂掉，然后会看到剩下的任务都会被另外的消费者执行。

运行结果如下：

The screenshot displays two Eclipse IDE console windows side-by-side, showing the execution of a RabbitMQ consumer. The left window is for '消费者1' (Consumer 1) and the right window is for '消费者2' (Consumer 2). Both windows show the same sequence of messages: '[x] Sent 'Hello World'', '[x] Sent 'Hello World.', '[x] Sent 'Hello World..', '[x] Sent 'Hello World...', and '[x] Sent 'Hello World....'. The left window shows the first four messages being processed and completed, while the fifth message is interrupted. The right window shows the first message being processed and completed, while the other four messages are interrupted. The text '人为关掉，模拟消费者异常挂掉' (Manually closed, simulating consumer abnormal hang) is written in red below the first window. The text '该任务是第一个消费者执行中断挂掉后，第二个消费者来执行了该任务' (This task is executed by the second consumer after the first consumer is interrupted) is written in red to the right of the second window.

消费者1	消费者2
[x] Received 'Hello World' 0个小数点	[x] Received 'Hello World.' 1个小数点
[x] Proccessing... at 2015-9-24 13:42:06	[x] Proccessing... at 2015-9-24 13:42:06
[x] Done! at 2015-9-24 13:42:06 执行完毕	[x] Done! at 2015-9-24 13:42:07 执行完毕
[x] Received 'Hello World..' 2个小数点	[x] Received 'Hello World...' 3个小数点
[x] Proccessing... at 2015-9-24 13:42:06	[x] Proccessing... at 2015-9-24 13:42:07
[x] Done! at 2015-9-24 13:42:08 执行完毕	[x] Done! at 2015-9-24 13:42:10 执行完毕
[x] Received 'Hello World....' 4个小数点	[x] Received 'Hello World....' 4个小数点
[x] Proccessing... at 2015-9-24 13:42:08	[x] Proccessing... at 2015-9-24 13:42:10
人为关掉，模拟消费者异常挂掉	[x] Done! at 2015-9-24 13:42:14 执行完毕
【执行中断】	该任务是第一个消费者执行中断挂掉后，第二个消费者来执行了该任务

如果你关闭了自动消息应答，手动也未设置应答，这是一个很简单错误，但是后果却是极其严重的。消息在分发出去以后，得不到回应，所以不会在内存中删除，结果RabbitMQ会



越来越占用内存，最终的结果，你懂得。。。

## Message durability (消息持久化)

我们已经了解了如何确保即使消费者死亡，任务也不会丢失。但是如果RabbitMQ服务器停止，我们的任务仍将失去！

当RabbitMQ退出或者崩溃，将会丢失队列和消息。除非你不要队列和消息。两件事儿必须保证消息不被丢失：我们必须把“队列”和“消息”设为持久化。

```
boolean durable = true;  
channel.queueDeclare("hello", durable, false, false, null);
```

尽管这行代码是正确的，但他不会在我们当前的设置中起作用。因为我们已经定义了一个名叫hello的未持久化的队列。RabbitMQ不允许使用不同的参数设定重新定义已经存在的队列，并且会向尝试如此做的程序返回一个错误。一个快速的解决方案——就是声明一个不同名字的队列，比如task\_queue。

(当然，我们也可以登录到RabbitMQ的服务管理页面，RabbitMQ默认的端口是5672，管理页面默认端口是15672，页面地址为：<http://localhost:15672>，使用是用户名和密码登录。RabbitMQ的默认密码和用户名都是guest。点开“queue”那栏，可以看到队列列表，点击“hello”杜列，会展开队列的详细信息。把页面拉到最后，有一项“Delete / purge”，点开，点击“Delete”按钮，就可以把队列删除掉了。然后再运行代码的时候，就会创建一个支持持久化的hello队列。)

上述的代码需要在生产者和消费者都要作出同样的修改。

在这一点上我们确信task\_queue的队列不会丢失，即使RabbitMQ服务重启。现在我们需要将消息标记为持久性的——通过设置 MessageProperties(实现BasicProperties)为 PERSISTENT\_TEXT\_PLAIN。

现在你可以启动RabbitMQ服务器，执行一次生产者NewTask的程序，然后关闭服务器，再重新启动服务器，运行消费者Work做下实验。可以发现消费者依旧可以读出消息来。说明在RabbitMQ服务器关闭后，消息和队列信息都已经做了持久化。再次启动后，会重新加载到服务器中，消费者运行后，就可以正常的从队列中获取消息了。



## 轻松搞定RabbitMQ（四）——发布/订阅

翻译地址：<http://www.rabbitmq.com/tutorials/tutorial-three-java.html>

在前面的教程中，我们创建了一个工作队列，都是假设一个任务只交给一个消费者。这次我们做一些完全不同的事儿——将消息发送给多个消费者。这种模式叫做“发布/订阅”。

为了说明这个模式，我们将构建一个简单日志系统。它包含2段程序：第一个将发出日志消息，第二个接受并打印消息。

如果在日志系统中每一个接受者（订阅者）都会得到消息的拷贝。那样的话，我们可以运行一个接受者（订阅者）程序，直接把日志记录到硬盘。同时运行另一个接受者（订阅者）程序，打印日志到屏幕上。

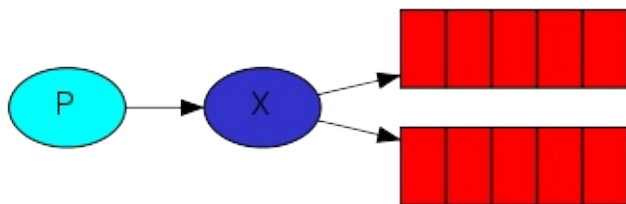
说白了，发表日志消息将被广播给所有的接收者。

### Exchanges（转发器）

前面的博文汇总，我们都是基于一个队列发送和接受消息。现在介绍一下完整的消息传递模式。

RabbitMQ消息模式的核心理念是：生产者没有直接发送任何消费到队列。实际上，生产者都不知道这个消费是发送给哪个队列的。

相反，生产者只能发送消息给转发器，转发器是非常简单的。一方面它接受生产者的消息，另一方面向队列推送消息。转发器必须清楚的知道如何处理接收到的消息。附加一个特定的队列吗？附加多个队列？或者是否丢弃？这些规则通过转发器的类型进行定义。



类型有：Direct、Topic、Headers和Fanout。我们关注最后一个。现在让我们创建一个该类型的转发器，定义如下：

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

fanout转发器非常简单，从名字就可以看出，它是广播接受到的消息给所有的队列。而这正好符合日志系统的需求。

## Nameless exchange ( 匿名转发 )

之前我们对转换器一无所知，却可以将消息发送到队列，那是可能是我们用了默认的转发器，转发器名为空字符串""。之前我们发布消息的代码是：

```
channel.basicPublish("", "hello", null, message.getBytes());
```

第一个参数就是转发器的名字，空字符串表示模式或者匿名的转发器。消息通过队列的routingKey路由到指定的队列中去，如果存在的话。

现在我们可以指定转发器的名字了：

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

## Temporary queues ( 临时队列 )

你可能还记得之前我们用队列时，会指定一个名字。队列有名字对我们来说是非常重要的——我们需要为消费者指定同一个队列。

但这并不是我们的日志系统所关心的。我们要监听所有日志消息，而不仅仅是一类日志。我们只对当前流动的消息感兴趣。解决这些问题，我盟需要完成两件事。

首先，每当我盟连接到RabbitMQ时，需要一个新的空队列。为此我们需要创建一个随机名字的空队列，或者更好的，让服务器选好一个随机名字的空队列给我们。

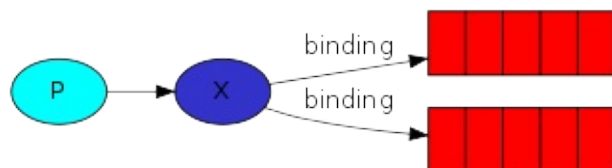
其次，一旦消费者断开连接，队列将自动删除。

我们提供一个无参的queueDeclare()方法，创建一个非持久化、独立的、自动删除的队列，且名字是随机生成的。

```
String queueName = channel.queueDeclare().getQueue();
```

queueName是一个随机队列名。看起来会像amq.gen-JzTY20BRgKO-HjmUJj0wLg。

## Bindings ( 绑定 )

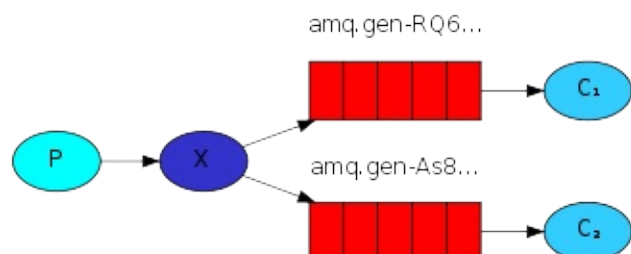


我们已经创建了一个广播的转发器和一个随机队列。现在需要告诉转发器转发消息到队列。这个关联转发器和队列的我们叫它Binding。

```
channel.queueBind(queueName, "logs", "");
```

这样，日志转发器将附加到日志队列上去。

## 完整的例子：



### 发送端代码（生产者）EmitLog.java

```
public class EmitLog {
    private final static String EXCHANGE_NAME = "logs";

    public static void main(String[] args) throws IOException {
        /**
         * 创建连接连接到MabbitMQ
         */
        ConnectionFactory factory = new ConnectionFactory();
        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 创建一个连接
        Connection connection = factory.newConnection();
        // 创建一个频道
        Channel channel = connection.createChannel();
        // 指定转发—广播
        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");

        for(int i=0;i<3;i++){
            // 发送的消息
            String message = "Hello World!";
            channel.basicPublish(EXCHANGE_NAME, "", null, mes
            sage.getBytes());
        }
    }
}
```

```

        System.out.println(" [x] Sent '" + message + "'");
    }

    // 关闭频道和连接
    channel.close();
    connection.close();
}

```

## 消费者1 ReceiveLogs2Console.java

```

public class ReceiveLogs2Console {
    private static final String EXCHANGE_NAME = "logs";

    public static void main(String[] argv) throws IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道, 与发送端一样
        Connection connection = factory.newConnection();
        final Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
        // 声明一个随机队列
        String queueName = channel.queueDeclare().getQueue();
        channel.queueBind(queueName, EXCHANGE_NAME, "");
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received '" + message + "'");
            }
        };
        channel.basicConsume(queueName, true, consumer);
    }
}

```

## 消费者2 ReceiveLogs2File.java

```

public class ReceiveLogs2File {
    private static final String EXCHANGE_NAME = "logs";

    public static void main(String[] argv) throws IOException, InterruptedException {

```

```

uptedException {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("127.0.0.1");
    // 打开连接和创建频道, 与发送端一样
    Connection connection = factory.newConnection();
    final Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
    // 声明一个随机队列
    String queueName = channel.queueDeclare().getQueue();
    channel.queueBind(queueName, EXCHANGE_NAME, "");
    System.out.println(" [*] Waiting for messages. To exit pr
ess CTRL+C");

    // 创建队列消费者
    final Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IO
Exception {
            String message = new String(body, "UTF-8");
            print2File(message);
            System.out.println(" [x] Received '" + messag
e + "'");
        }
    };
    channel.basicConsume(queueName, true, consumer);

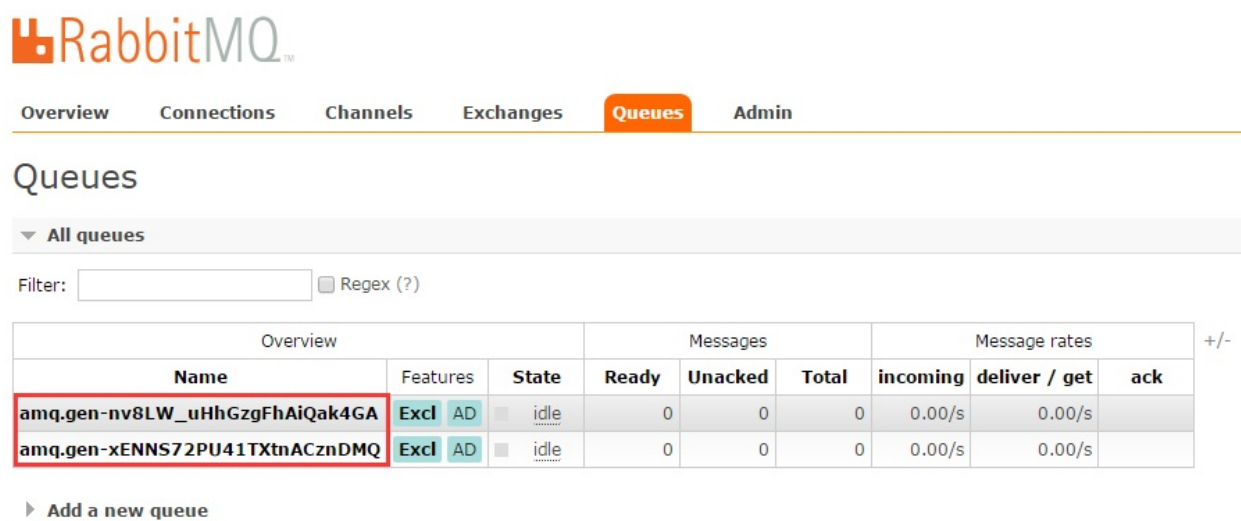
    private static void print2File(String msg) {
        try {
            String dir = ReceiveLogs2File.class.getClassLoad
er().getResource("").getPath();
            String logFileName = new SimpleDateFormat("yyyy-M
M-dd").format(new Date());
            File file = new File(dir, logFileName + ".log");
            FileOutputStream fos = new FileOutputStream(file,
true);
            fos.write(((new SimpleDateFormat("HH:mm:ss")).form
at(new Date())+" - "+msg + "\r\n").getBytes());
            fos.flush();
            fos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

可以看到我们1个生产者用于发送log消息, 2个消费者, 一个用于显示, 一个用于记录文件。

生产者声明了一个广播模式的转换器，订阅这个转换器的消费者都可以收到每一条消息。可以看到在生产者中，没有声明队列。这也验证了之前说的。生产者其实只关心exchange，至于exchange会把消息转发给哪些队列，并不是生产者关心的。

2个消费者，一个打印日志，一个写入文件，除了这2个地方不一样，其他地方一模一样。也是声明一下广播模式的转换器，而队列则是随机生成的，消费者实例启动后，会创建一个随机实例，这个在管理页面可以看到(如图)。而实例关闭后，随机队列也会自动删除。最后将队列与转发器绑定。



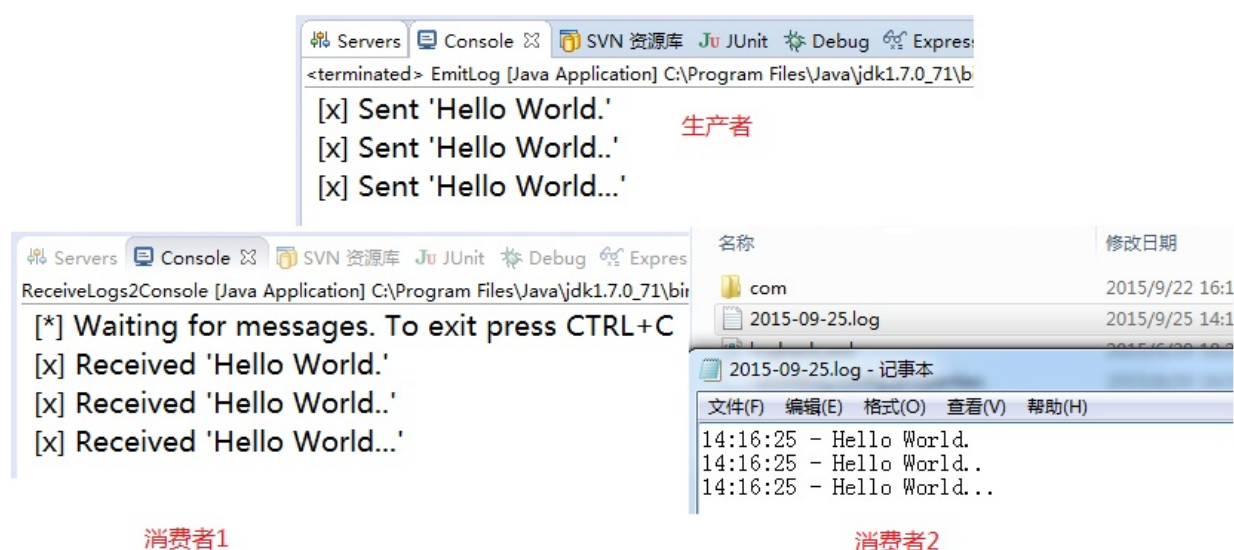
The screenshot shows the RabbitMQ Management interface. The 'Queues' tab is selected. A table lists the queues:

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
amq.gen-nv8LW_uHhGzgFhAiQak4GA	Excl AD	idle	0	0	0	0.00/s	0.00/s	
amq.gen-xENNS72PU41TXtnACznDMQ	Excl AD	idle	0	0	0	0.00/s	0.00/s	

Below the table, there is a link: [Add a new queue](#).

注：运行的时候要先运行2个消费者实例，然后在运行生产者实例。否则获取不到实例。

看看最终的结果吧：



The screenshot shows two terminal windows and a file explorer. The top terminal window, labeled '生产者' (Producer), shows three messages being sent: '[x] Sent 'Hello World.', [x] Sent 'Hello World.', [x] Sent 'Hello World...'. The bottom terminal window, labeled '消费者1' (Consumer 1), shows three messages being received: '[x] Received 'Hello World.', [x] Received 'Hello World.', [x] Received 'Hello World...'. The file explorer shows a folder named 'com' and a file named '2015-09-25.log'.





# 轻松搞定RabbitMQ ( 五 ) ——路由选择

翻译地址：<http://www.rabbitmq.com/tutorials/tutorial-four-java.html>

在[前篇博文](#)中，我们建立了一个简单的日志系统。可以广播消息给多个消费者。本篇博文，我们将添加新的特性——我们可以只订阅部分消息。比如：我们可以接收Error级别的消息写入文件。同时仍然可以在控制台打印所有日志。

## Bindings ( 绑定 )

在上一篇博客中我们已经使用过绑定。类似下面的代码：

```
channel.queueBind(queueName, EXCHANGE_NAME, "");
```

绑定表示转换器与队列之间的关系。可以简单的人为：队列对该转发器上的消息感兴趣。

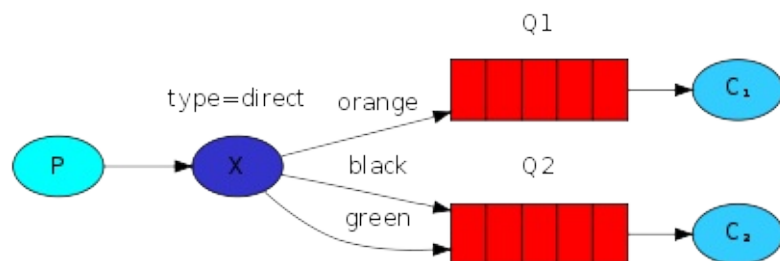
绑定可以设定额外的routingKey参数。为了与避免basicPublish方法（发布消息的方法）的参数混淆，我们准备把它称作绑定键（binding key）。下面展示如何使用绑定键（binding key）来创建一个绑定：

```
channel.queueBind(queueName, EXCHANGE_NAME, "black");
```

绑定键关键取决于转换器的类型。对于fanout类型，忽略此参数。

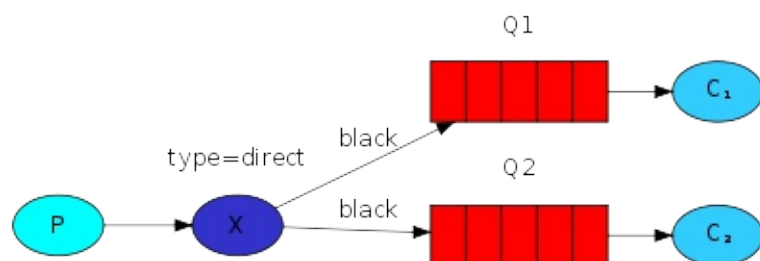
## Direct exchange ( 直接转发 )

前面讲到我们的日志系统广播消息给所有的消费者。我们想对其扩展，根据消息的严重性来过滤消息。例如：我们希望将致命错误的日志消息记录到文件，而不是把磁盘空间浪费在warn和info类型的日志上。我们使用的fanout转发器，不能给我们太多的灵活性。它仅仅只是盲目的广播而已。我们使用direct转发器进行代替，其背后的算法很简单——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。



在上图中，我们可以看到direct类型的转发器与2个队列进行了绑定。第一个队列使用的绑定键是orange，第二个队列绑定键为black和green。这样当消息发布到转发器是，附带orange绑定键的消息将被路由到队列Q1中去。附带black和green绑定键的消息被路由到Q2中去。其他消息全部丢弃。

## Multiple bindings ( 多重绑定 )



使用一个绑定键绑定多个队列是完全合法的。如上图，绑定键black绑定了2个队列——Q1和Q2。

## Emitting logs(发送日志)

我们将这种模式用于日志系统，发送消息给direct类型的转发器。我们将 提供日志严重性做为绑定键。那样，接收程序可以选择性的接收严重性的消息。首先关注发送日志的代码：

像往常一样首先创建一个转换器：

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
```

然后为发送消息做准备：

```
channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes());
```

为了简化代码，我们假定日志的严重性是 'info' , 'warning' , 'error' 中之一。

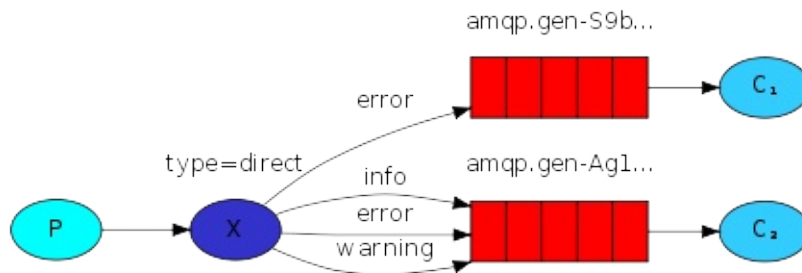
## Subscribing(订阅)

接收消息跟前面博文中的一样。我们仅需要修改一个地方：为每一个我们感兴趣的严重性的消息，创建一个新的绑定。

```
String queueName = channel.queueDeclare().getQueue();

for(String severity : argv){
    channel.queueBind(queueName, EXCHANGE_NAME, severity);
}
```

## 完整的例子



### 发送端代码 ( EmitLogDirect.java )

```
public class EmitLogDirect {
    private final static String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] args) throws IOException {
        /**
         * 创建连接连接到MabbitMQ
         */
        ConnectionFactory factory = new ConnectionFactory();
        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 创建一个连接
        Connection connection = factory.newConnection();
        // 创建一个频道
        Channel channel = connection.createChannel();
        // 指定转发—广播
        channel.exchangeDeclare(EXCHANGE_NAME, "direct");

        //所有日志严重性级别
        String[] severities={"error","info","warning"};
        for(int i=0;i<3;i++){
            String severity = severities[i%3]; //每一次发送一条不
```

同严重性的日志

```

        // 发送的消息
        String message = "Hello World"+Strings.repeat(".",
, i+1);

        //参数1:exchange name
        //参数2:routing key
        channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes());
        System.out.println(" [x] Sent '" + severity + "': '"
        + message + "'");
    }
    // 关闭频道和连接
    channel.close();
    connection.close();
}
}

```

### 消费者1 ( ReceiveLogs2Console.java )

```

public class ReceiveLogs2Console {
    private static final String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] argv) throws IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道, 与发送端一样
        Connection connection = factory.newConnection();
        final Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "direct");
        // 声明一个随机队列
        String queueName = channel.queueDeclare().getQueue();

        //所有日志严重性级别
        String[] severities={"error","info","warning"};
        for (String severity : severities) {
            //关注所有级别的日志(多重绑定)
            channel.queueBind(queueName, EXCHANGE_NAME, severity);
        }

        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received '" + envelope.getRoutingKey() + "': '" + message + "'");
            }
        };
    }
}

```

```

        }
    };
    channel.basicConsume(queueName, true, consumer);
}
}

```

## 消费者2 ( ReceiveLogs2File.java )

```

public class ReceiveLogs2File {
    private static final String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] argv) throws IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道, 与发送端一样
        Connection connection = factory.newConnection();
        final Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "direct");
        // 声明一个随机队列
        String queueName = channel.queueDeclare().getQueue();

        String severity="error";//只关注error级别的日志, 然后记录到文件中去
        channel.queueBind(queueName, EXCHANGE_NAME, severity);

        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                //记录日志到文件:
                print2File( "["+ envelope.getRoutingKey() + "
] "+message);
            }
        };
        channel.basicConsume(queueName, true, consumer);
    }

    private static void print2File(String msg) {
        try {
            String dir = ReceiveLogs2File.class.getClassLoader().getResource("").getPath();
            String logFileName = new SimpleDateFormat("yyyy-MM-dd").format(new Date());
            File file = new File(dir, logFileName + ".log");
            FileOutputStream fos = new FileOutputStream(file,

```

```

true);
        fos.write((new SimpleDateFormat("HH:mm:ss").format(new Date()))+" - "+msg + "\r\n").getBytes());
        fos.flush();
        fos.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

最终结果：



罗哩罗嗦的说这么多，其实就是说了这么一件事：我们可以使用Direct exchange+routingKey来过滤自己感兴趣的消息。一个队列可以绑定多个routingKey。这就是我们今天的主题——路由选择。

## 轻松搞定RabbitMQ ( 六 ) ——主题

翻译地址：<http://www.rabbitmq.com/tutorials/tutorial-five-java.html>

在上一篇博文中，我们进一步改良了日志系统。使用Direct类型的转换器，使得接收者有能力进行选择性的接收日志，而非fanout那样，只能无脑的转发，如果你还不了解，请阅读：[轻松搞定RabbitMQ \( 四 \) ——发布/订阅](#)。

虽然使用Direct类型的转换器改进了日志系统。但它仍然有一定的局限性——不能根据多重条件进行路由选择。

在我们的日志系统中，我们可能不仅仅根据日志严重性订阅日志，也想根据发送源订阅。你可能从unix工具syslog了解过这个概念，它可以根据严重性 ( info/warning/crit... ) 和设备 ( auth/cron/kern... ) 转发日志。

这将给我们更多的灵活性——我们可以订阅来自元 ‘cron’ 的致命错误日志，同时也可以订阅 ‘kern’ 的所有日志。

为了在我们的日志系统中实现上述需求，我们需要了解更复杂的主题类型的转发器——Topic Exchange。

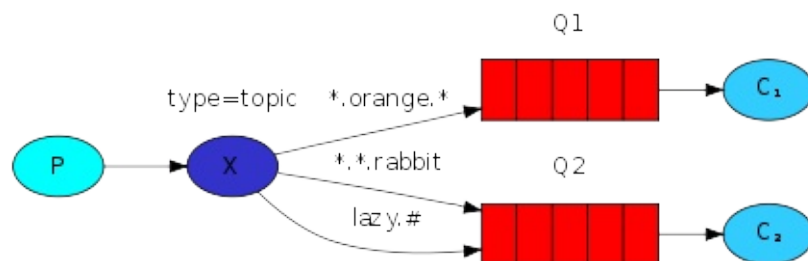
### Topic exchange ( 主题转发器 )

发送给主题转发器的消息不能是任意设置的选择键，必须是用小数点隔开的一系列的标识符。这些标识符可以是随意，但是通常跟消息的某些特性相关联。一些合法的路由选择键比如 “socket.usd.nyse” , “nyse.vmw” , “quick.orange.rabbit” , 你愿意用多少单词都可以，只要不超过上限的255个字节。

绑定键也必须以相同的格式。主题转发器的逻辑类似于direct类型的转发器。消息通过一个特定的路由键发送到所有与绑定键匹配的队列中。需要注意的是，关于绑定键有两种特殊的情况：\* ( 星号 ) 可以代替任意一个标识符；# ( 井号 ) 可以代替零个或多个标识符。

举一个简单例子，如下图：





在上图例子中，我们发送描述动物的消息。消息会转发给包含3个单词（2个小数点）的路由键绑定的队列中。绑定键中的第一个单词描述的是速度，第二个是颜色，第三个是物种：“速度.颜色.物种”。

我们创建3个绑定：Q1绑定键是 “.orange.”，Q2绑定键是 “..rabbit”，Q3绑定键是 “lazy.#”。这些绑定可以概括为：Q1只对橙色的动物感兴趣。Q2则是关注兔子和所有懒的动物。

路由键为 “quick.orange.rabbit” 的消息会被路由到2个队列中去。而 “lazy.orange.elephant” 的消息同样会发往2个队列。另外 “quick.orange.fox” 仅仅发往第一个队列，而 “lazy.brown.fox” 则只发往第二个队列。“quick.brown.fox” 则所有的绑定键都不匹配而被丢弃。

如果我们违反约定，发送了只带1个或者4个标识符的选择键，像 “orange” 或者 “quick.orange.male.rabbit”，会发生什么呢？这些消息都不匹配任何绑定，所以将被丢弃。

另外，“lazy.orange.male.rabbit”，尽管有4个标识符，但是仍然匹配最后一个绑定键，所以会发送到第二个队列中。

注：主题类型的转发器非常强大，可以实现其他类型的转发器。当队列绑定#绑定键，可以接受任何消息，类似于fanout转发器。当特殊字符\*和#不包含在绑定键中，这个主题转发器就像一个direct类型的转发器。

## 完整实例

我们将主题类型的转发器应用到日志系统中，路由格式为：“<设备>.<严重级别>”。

发送端 (EmitLogTopic.java)

```
public class EmitLogDirect {
    private final static String EXCHANGE_NAME = "topic_logs";
```

本文档使用 [看云](#) 构建

```

public static void main(String[] args) throws IOException {
    /**
     * 创建连接连接到MabbitMQ
     */
    ConnectionFactory factory = new ConnectionFactory();
    // 设置MabbitMQ所在主机ip或者主机名
    factory.setHost("127.0.0.1");
    // 创建一个连接
    Connection connection = factory.newConnection();
    // 创建一个频道
    Channel channel = connection.createChannel();
    // 指定转发—广播
    channel.exchangeDeclare(EXCHANGE_NAME, "topic");

    //所有设备和日志级别
    String[] facilities = {"auth", "cron", "kern", "auth.A"};
    String[] severities = {"error", "info", "warning"};

    for(int i=0; i<4; i++){
        for(int j=0; j<3; j++){
            //每一个设备，每种日志级别发送一条日志消息
            String routingKey = facilities[i] + "." + severities[j%3];

            // 发送的消息
            String message = " Hello World!" + Strings.repeat(".", i+1);

            //参数1: exchange name
            //参数2: routing key
            channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes());
            System.out.println(" [x] Sent [" + routingKey + "] : '" + message + "'");
        }
    }
    // 关闭频道和连接
    channel.close();
    connection.close();
}

```

### 消费者1 ( ReceiveLogs2Console.java )

```

public class ReceiveLogs2Console {
    private static final String EXCHANGE_NAME = "topic_logs";

    public static void main(String[] argv) throws IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道，与发送端一样
        Connection connection = factory.newConnection();
    }
}

```

```

        final Channel channel = connection.createChannel();

        //声明exchange
        channel.exchangeDeclare(EXCHANGE_NAME, "topic");
        // 声明一个随机队列
        String queueName = channel.queueDeclare().getQueue();

        String[] routingKeys = {"auth.*", ".*.info", "#.warning"}; //
        关注所有的授权日志、所有info和waring级别的日志
        for (String routingKey : routingKeys) {
            //关注所有级别的日志（多重绑定）
            channel.queueBind(queueName, EXCHANGE_NAME, routi
ngKey);
        }
        System.out.println(" [*] Waiting for messages. To exit pr
ess CTRL+C");

        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws I
OException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received [" + envel
ope.getRoutingKey() + "] :'" + message + "'");
            }
        };
        channel.basicConsume(queueName, true, consumer);
    }
}

```

## 消费者2 ( ReceiveLogs2File.java )

```

public class ReceiveLogs2File {
    private static final String EXCHANGE_NAME = "topic_logs";

    public static void main(String[] argv) throws IOException, Interr
uptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("127.0.0.1");
        // 打开连接和创建频道，与发送端一样
        Connection connection = factory.newConnection();
        final Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "topic");
        // 声明一个随机队列
        String queueName = channel.queueDeclare().getQueue();
        channel.queueBind(queueName, EXCHANGE_NAME, "");

        String severity="kern.error";//只关注核心错误级别的日志，然后记录到
        文件中去。
        channel.queueBind(queueName, EXCHANGE_NAME, severity);
    }
}

```

```

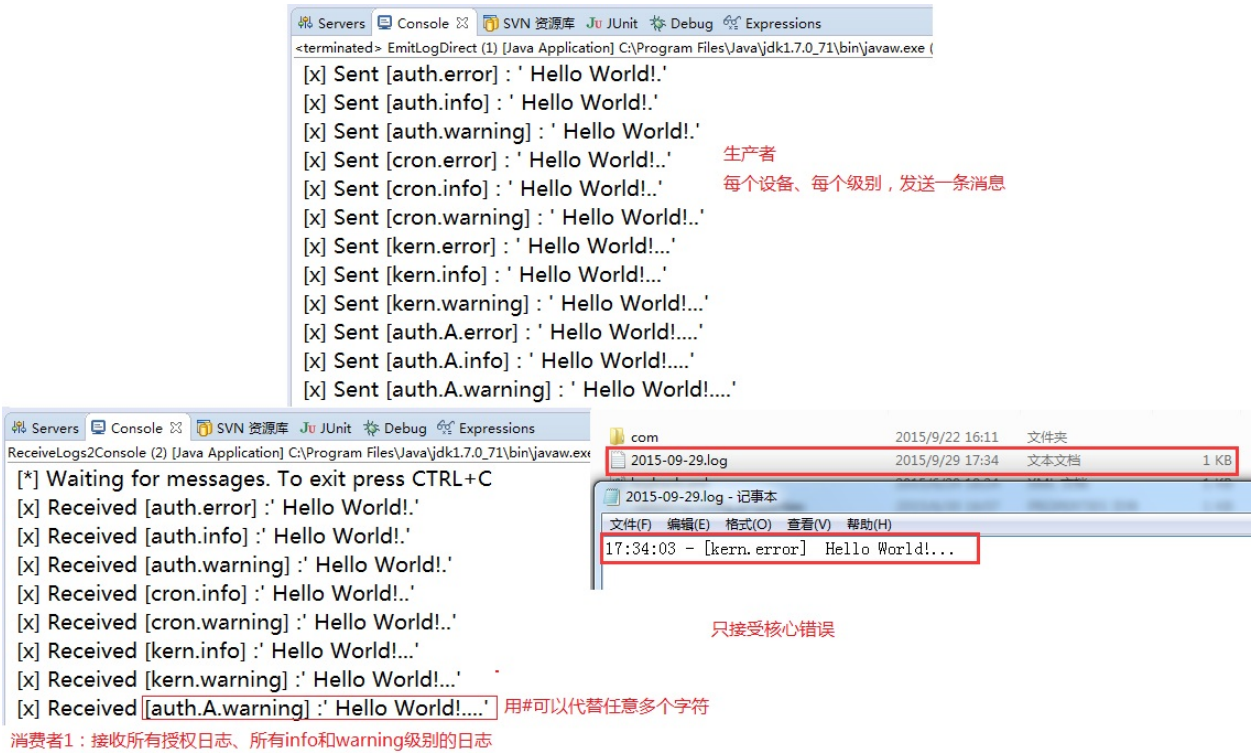
        System.out.println(" [*] Waiting for messages. To exit pr
ess CTRL+C");

        // 创建队列消费者
        final Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag,
Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws I
OException {
                String message = new String(body, "UTF-8");
                //记录日志到文件：
                print2File( "["+ envelope.getRoutingKey() + "
] "+message);
            }
        };
        channel.basicConsume(queueName, true, consumer);
    }

    private static void print2File(String msg) {
        try {
            String dir = ReceiveLogs2File.class.getClassLoad
er().getResource("").getPath();
            String logFileName = new SimpleDateFormat("yyyy-M
M-dd").format(new Date());
            File file = new File(dir, logFileName + ".log");
            FileOutputStream fos = new FileOutputStream(file,
true);
            fos.write((new SimpleDateFormat("HH:mm:ss").forma
t(new Date()))+" - "+msg + "\r\n").getBytes());
            fos.flush();
            fos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

最终结果：



# 轻松搞定RabbitMQ ( 七 ) —— 远程过程调用RPC

翻译：<http://www.rabbitmq.com/tutorials/tutorial-six-java.html>

在第二篇博文中，我们已经了解到了如何使用工作队列来向多个消费者分散耗时任务。

但是付过我们需要在远程电脑上运行一个方法然后等待结果，该怎么办？这是不同的需求。这个模式通常叫做RPC。

本文我们将使用RabbitMQ构建一个RPC系统：一个客户端和一个可扩展的RPC服务器端。由于我们没有任何真实的耗时任务需要分配，所以我们将创建一个虚拟的RPC服务，可以返回斐波纳契数列。

## Client interface ( 客户端接口 )

为了说明RPC服务可以使用，我们创建一个简单的客户端类。暴露一个方法——发送RPC请求，然后阻塞直到获得结果。

```
FibonacciRpcClient fibonacciRpc = new FibonacciRpcClient();
String result = fibonacciRpc.call("4");
System.out.println( "fib(4) is " + result);
```

## Callback queue ( 回调队列 )

一般在RabbitMQ中做RPC是很简单的。客户端发送请求消息，服务器回复响应的消息。为了接受响应的消息，我们需要在请求消息中发送一个回调队列。可以用默认的队列（仅限java客户端）。试一试吧：

```
BasicProperties props = new BasicProperties
    .Builder()
    .replyTo(callbackQueueName)
    .build();

channel.basicPublish("", "rpc_queue", props, message.getBytes());

// ... then code to read a response message from the callback_queue ...
```

## Message properties ( 消息属性 )

AMQP协议为消息预定义了一组14个属性。大部分的属性是很少使用的。除了一下几种：

- `deliveryMode`：标记消息传递模式，2-消息持久化，其他值-瞬态。在第二篇文章中还提到过。
- `contentType`：内容类型，用于描述编码的mime-type。例如经常为该属性设置JSON编码。
- `replyTo`：应答，通用的回调队列名称
- `correlationId`：关联ID，方便RPC响应与请求关联

我们需要添加一个新的导入

```
import com.rabbitmq.client.AMQP.BasicProperties;
```

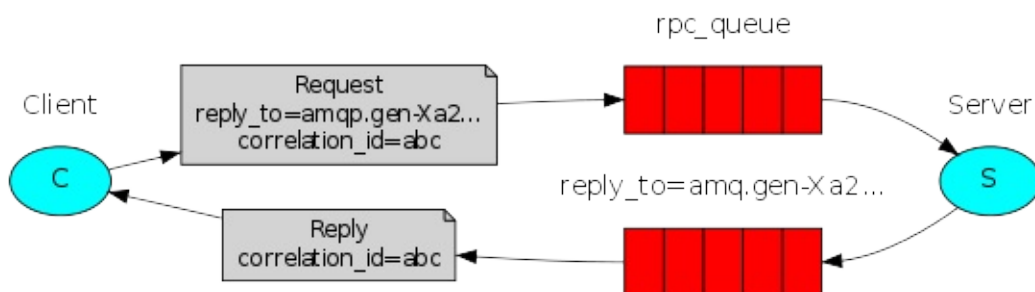
## Correlation Id

在上述方法中为每个RPC请求创建一个回调队列。这是很低效的。幸运的是，一个解决方案：可以为每个客户端创建一个单一的回调队列。

新的问题被提出，队列收到一条回复消息，但是不清楚是那条请求的回复。这就需要使用了`correlationId`属性了。我们要为每个请求设置唯一的值。然后，在回调队列中获取消息，看看这个属性，关联`response`和`request`就是基于这个属性值的。如果我们看到一个未知的`correlationId`属性值的消息，可以放心的无视它——它不是我们发送的请求。

你可能问道，为什么要忽略回调队列中未知的信息，而不是当作一个失败？这是由于在服务器端竞争条件的导致的。虽然不太可能，但是如果RPC服务器在发送给我们结果后，发送请求反馈前就挂掉了，这有可能会发送未知`correlationId`属性值的消息。如果发生了这种情况，重启RPC服务器将会重新处理该请求。这就是为什么在客户端必须很好的处理重复响应，RPC应该是幂等的。

## Summary ( 总结 )



我们的RPC的处理流程：

1. 当客户端启动时，创建一个匿名的回调队列。
2. 客户端为RPC请求设置2个属性：replyTo，设置回调队列名字；correlationId，标记request。
3. 请求被发送到rpc\_queue队列中。
4. RPC服务器端监听rpc\_queue队列中的请求，当请求到来时，服务器端会处理并且把带有结果的消息发送给客户端。接收的队列就是replyTo设定的回调队列。
5. 客户端监听回调队列，当有消息时，检查correlationId属性，如果与request中匹配，那就是结果了。

## 完整的实例

RPC服务器端 ( RPCServer.java )

```
/**
 * RPC服务器端
 *
 * @author arron
 * @date 2015年9月30日 下午3:49:01
 * @version 1.0
 */
public class RPCServer {

    private static final String RPC_QUEUE_NAME = "rpc_queue";

    public static void main(String[] args) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 创建一个连接
        Connection connection = factory.newConnection();
        // 创建一个频道
        Channel channel = connection.createChannel();

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

        //限制：每次最多给一个消费者发送1条消息
        channel.basicQos(1);

        //为rpc_queue队列创建消费者，用于处理请求
        QueueingConsumer consumer = new QueueingConsumer(channel)
;

        channel.basicConsume(RPC_QUEUE_NAME, false, consumer);

        System.out.println(" [x] Awaiting RPC requests");

        while (true) {
            QueueingConsumer.Delivery delivery = consumer.next
```



```

tDelivery();

//获取请求中的correlationId属性值, 并将其设置到结果消息
的correlationId属性中
BasicProperties props = delivery.getProperties();
BasicProperties replyProps = new BasicProperties.
Builder().correlationId(props.getCorrelationId()).build();
//获取回调队列名字
String callQueueName = props.getReplyTo();

String message = new String(delivery.getBody(), "U
TF-8");

System.out.println(" [.] fib(" + message + ")");

//获取结果
String response = "" + fib(Integer.parseInt(messa
ge));

//先发送回调结果
channel.basicPublish("", callQueueName, replyProp
s, response.getBytes());
//后手动发送消息反馈
channel.basicAck(delivery.getEnvelope().getDelive
ryTag(), false);
    }
}

/**
 * 计算斐波列其数列的第n项
 *
 * @param n
 * @return
 * @throws Exception
 */
private static int fib(int n) throws Exception {
    if (n < 0)
        throw new Exception("参数错误, n必须大于等于0");
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
}

```

RPC客户端 ( RPCClient.java ) :

```

/**
 *
 * @author arron
 * @date 2015年9月30日 下午3:44:43
 * @version 1.0
 */

```

```

public class RPCClient {

    private static final String RPC_QUEUE_NAME = "rpc_queue";

    private Connection connection;
    private Channel channel;
    private String replyQueueName;
    private QueueingConsumer consumer;

    public RPCClient() throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        // 设置MabbitMQ所在主机ip或者主机名
        factory.setHost("127.0.0.1");
        // 创建一个连接
        connection = factory.newConnection();
        // 创建一个频道
        channel = connection.createChannel();

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

        //为每一个客户端获取一个随机的回调队列
        replyQueueName = channel.queueDeclare().getQueue();
        //为每一个客户端创建一个消费者（用于监听回调队列，获取结果）
        consumer = new QueueingConsumer(channel);
        //消费者与队列关联
        channel.basicConsume(replyQueueName, true, consumer);
    }

    /**
     * 获取斐波列其数列的值
     *
     * @param message
     * @return
     * @throws Exception
     */
    public String call(String message) throws Exception{
        String response = null;
        String corrId = java.util.UUID.randomUUID().toString();

        //设置replyTo和correlationId属性值
        BasicProperties props = new BasicProperties.Builder().correla
tionId(corrId).replyTo(replyQueueName).build();

        //发送消息到rpc_queue队列
        channel.basicPublish("", RPC_QUEUE_NAME, props, message.getBo
tes());

        while (true) {
            QueueingConsumer.Delivery delivery = consumer.nextDeliver
y();
            if (delivery.getProperties().getCorrelationId().equals(co
rrId)) {
                response = new String(delivery.getBody(),"UTF-8");
                break;
            }
        }
    }
}

```

```
        }  
    }  
    return response;  
}  
  
public static void main(String[] args) throws Exception {  
    RPCClient fibonacciRpc = new RPCClient();  
    String result = fibonacciRpc.call("4");  
    System.out.println( "fib(4) is " + result);  
}  
}
```

输出结果：

```
fib(4) is 3
```