

1. 消息中间件概述

1.1. MQ概述

MQ全称 Message Queue（消息队列），是在消息的传输过程中保存消息的容器。多用于分布式系统之间进行通信。

应用之间的远程调用

OverviewConnectionsChannelsExchangesQueuesAdmin

Users

Virtual HostsPolicies

All users

Filter: ☐ Regex (?) (?) 1 item, page size up to 100

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/	•

(?)

Add a user

Username:

Password: (confirm)

Tags: (?)

Get Admin | Monitoring | Policymaker | Management | None

Add user

加入MQ后应用之间的调用

OverviewConnectionsChannelsExchangesQueuesAdmin

Users

Virtual Hosts

Policies

All virtual hosts

Filter: ☐ Regex (?) (?) 1 item, page size up to 100

Overview		Messages			Network		Message rates		
Name	Users (?)	Ready	Unacked	Total	From client	To client	publish	deliver	get
/	guest	NaN	NaN	NaN					

Add a new virtual host

Name:

Add virtual host

1.2.MQ的优势：

1、应用解耦

MQ相当于一个中介，生产方通过MQ与消费方交互，它将应用程序进行解耦合。

系统的耦合性越高，容错性就越低，可维护性就越低。

OverviewConnectionsChannelsExchangesQueuesAdmin

Virtual host: All

Virtual Host: / lxs

Users

Virtual Hosts

Policies

No users have permission to access this virtual host. Use "Set Permission" below to grant users permission to access this virtual host.

Overview

Permissions

Current permissions

... no permissions ...

Set permission

User:

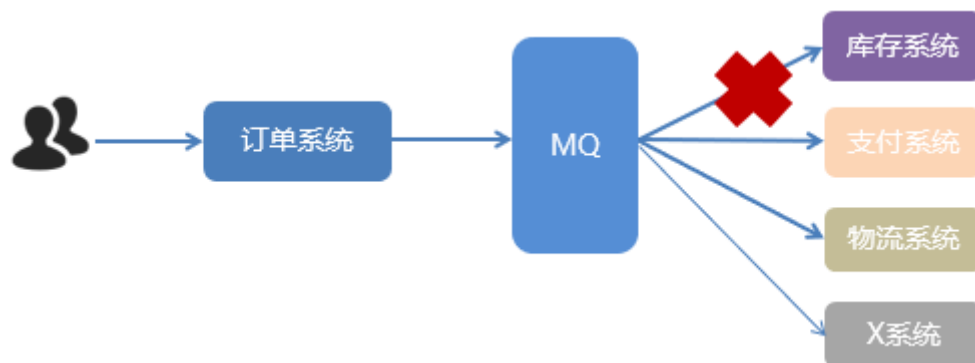
Configure regex:

Write regex:

Read regex:

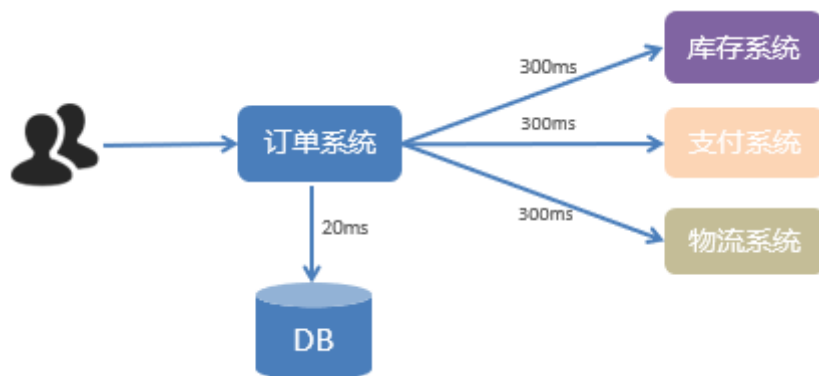
Set permission

使用 MQ 使得应用间解耦，提升容错性和可维护性。



2、任务异步处理

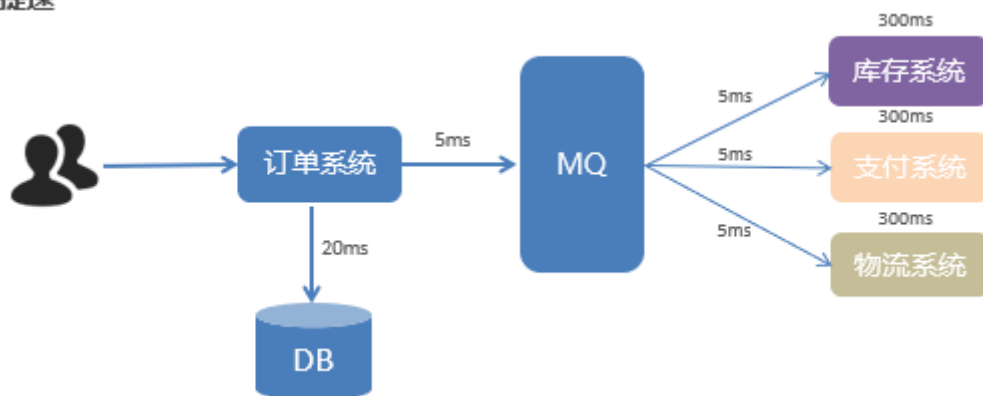
将不需要同步处理的并且耗时长操作由消息队列通知消息接收方进行异步处理。提高了应用程序的响应时间。



一个下单操作耗时： $20 + 300 + 300 + 300 = 920\text{ms}$

用户点击完下单按钮后，需要等待920ms才能得到下单响应，太慢！

改进

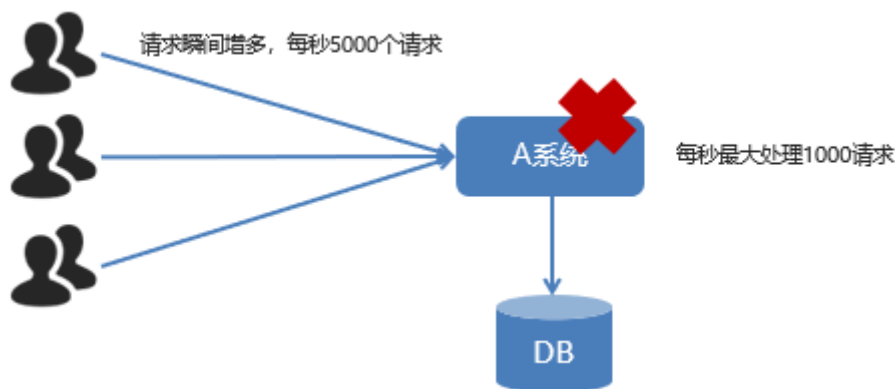


用户点击完下单按钮后，只需等待25ms就能得到下单响应 ($20 + 5 = 25\text{ms}$)。

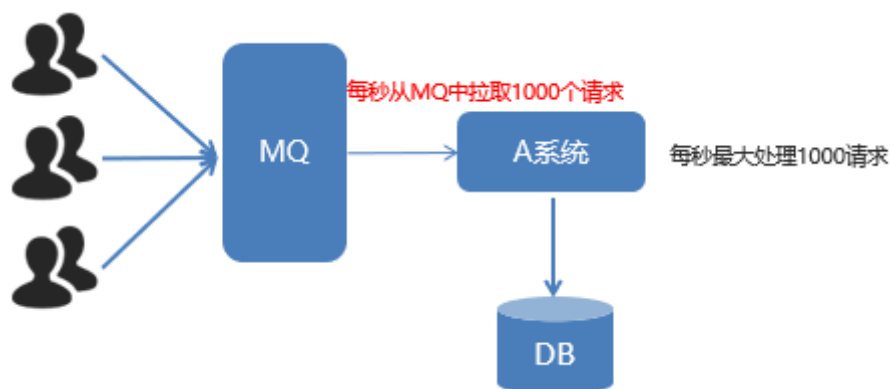
提升用户体验和系统吞吐量（单位时间内处理请求的数目）。

3、削峰填谷

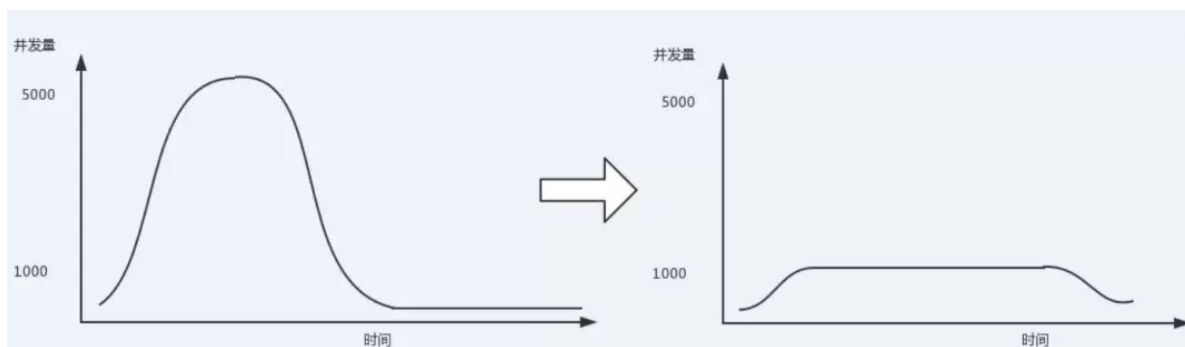
如订单系统，在下单的时候就会往数据库写数据。但是数据库只能支撑每秒1000左右的并发写入，并发量再高就容易宕机。低峰期的时候并发也就100多个，但是在高峰期时候，并发量会突然激增到5000以上，这个时候数据库肯定卡死了。



消息被MQ保存起来了，然后系统就可以按照自己的消费能力来消费，比如每秒1000个消息，这样慢慢写入数据库，这样就不会卡死数据库了。



但是使用了MQ之后，限制消费消息的速度为1000，但是这样一来，高峰期产生的数据势必会被积压在MQ中，高峰就被“削”掉了。但是因为消息积压，在高峰期过后的一段时间内，消费消息的速度还是会维持在1000QPS，直到消费完积压的消息,这就叫做“填谷”



1.3. MQ的劣势

系统可用性降低

系统引入的外部依赖越多，系统稳定性越差。一旦MQ宕机，就会对业务造成影响。如何保证MQ的高可用？

系统复杂度提高

MQ的加入大大增加了系统的复杂度，以前系统间是同步的远程调用，现在是通过MQ进行异步调用。如何保证消息没有被重复消费？怎么处理消息丢失情况？那么保证消息传递的顺序性？

一致性问题

A 系统处理完业务，通过 MQ 给B、C、D三个系统发消息，如果 B 系统、C 系统处理成功，D 系统处理失败。如何保证消息数据处理的一致性？

1.4. 常见的 MQ 产品

目前业界有很多的 MQ 产品，例如 RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、MetaMq 等，也有直接使用 Redis 充当消息队列的案例，而这些消息队列产品，各有侧重，在实际选型时，需要结合自身需求及 MQ 产品特征，综合考虑。

	RabbitMQ	ActiveMQ	RocketMQ	Kafka
公司/社区	Rabbit	Apache	阿里	Apache
开发语言	Erlang	Java	Java	Scala&Java

	RabbitMQ	ActiveMQ	RocketMQ	Kafka
协议支持	AMQP , XMPP, SMTP, STOMP	OpenWire,STOMP, REST,XMPP,AMQP	自定义	自定义协议, 社区封装了 http协议支持
客户端支持语言	官方支持Erlang, Java, Ruby等,社区产出多种API, 几乎支持所有语言	Java, C, C++, Python, PHP, Perl, .net等	Java, C++ (不成熟)	官方支持J ava, 社区产出多种 API, 如PHP, Python等
单机吞吐量	万级 (其次)	万级 (最差)	十万级 (最好)	十万级 (次之)
消息延迟	微妙级	毫秒级	毫秒级	毫秒以内
功能特性	并发能力强, 性能极其好, 延时低, 社区活跃, 管理界面丰富	老牌产品, 成熟度高, 文档较多	MQ功能比较完备, 扩展性佳	只支持主要的 MQ功能, 毕竟是为大数据领域准备的。

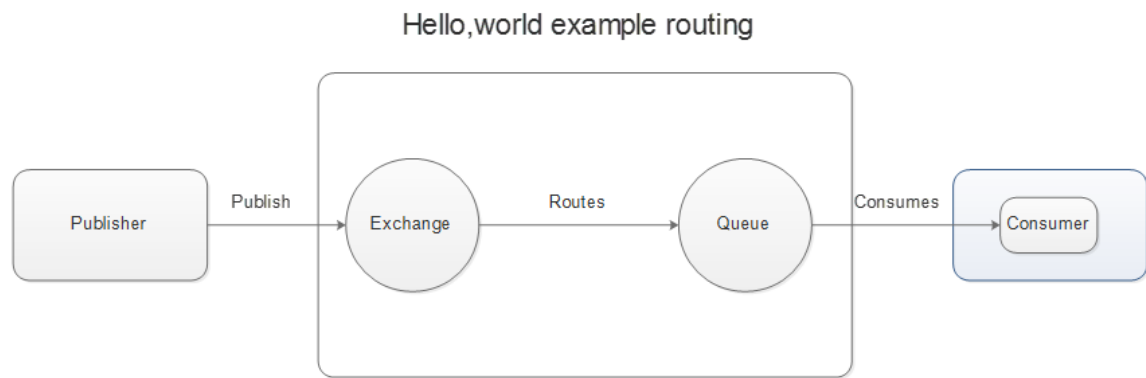
由于 RabbitMQ 综合能力强劲, 所以接下来的课程中, 我们将主要学习 RabbitMQ。

1.5. AMQP 和 JMS

实现MQ的大致有两种主流方式: AMQP、JMS。

AMQP

AMQP, 即 Advanced Message Queuing Protocol (高级消息队列协议), 是一个网络协议, 是应用层协议的一个开放标准, 为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息, 遵循此协议, 不收客户端和中间件产品和开发语言限制。2006年, AMQP 规范发布。类比HTTP。



JMS

JMS 即 Java 消息服务 (JavaMessage Service) 应用程序接口，是一个 Java 平台中关于面向消息中间件的API

JMS 是 JavaEE 规范中的一种，类比JDBC

很多消息中间件都实现了JMS规范，例如：ActiveMQ。RabbitMQ 官方没有提供 JMS 的实现包，但是开源社区有

AMQP 与 JMS 区别

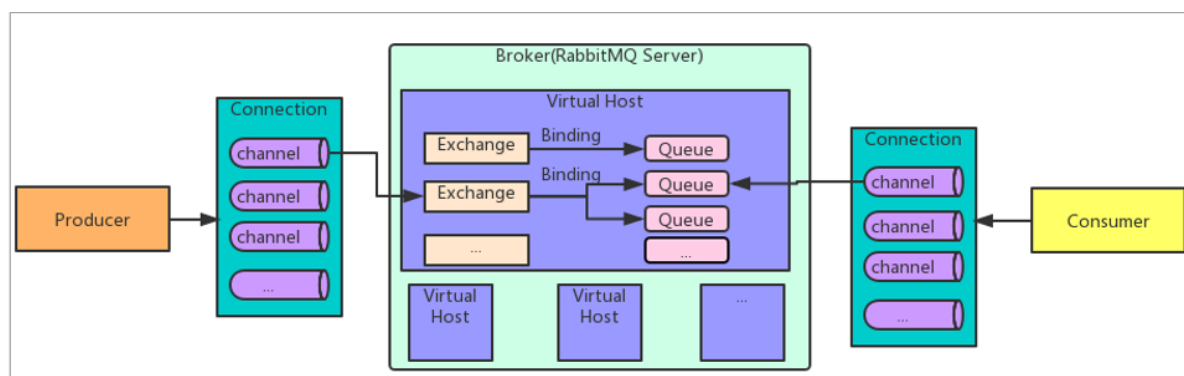
- JMS是定义了统一的接口，来对消息操作进行统一；AMQP是通过规定协议来统一数据交互的格式
- JMS限定了必须使用java语言；AMQP只是协议，不规定实现方式，因此是跨语言的。
- JMS规定了两种消息模式；而AMQP的消息模式更加丰富

1.6. RabbitMQ

RabbitMQ官方地址： <http://www.rabbitmq.com/>

2007年，Rabbit 技术公司基于 AMQP 标准开发的 RabbitMQ 1.0 发布。RabbitMQ 采用 Erlang 语言开发。Erlang 语言专门为开发高并发和分布式系统的一种语言，在电信领域使用广泛。

RabbitMQ 基础架构如下图：

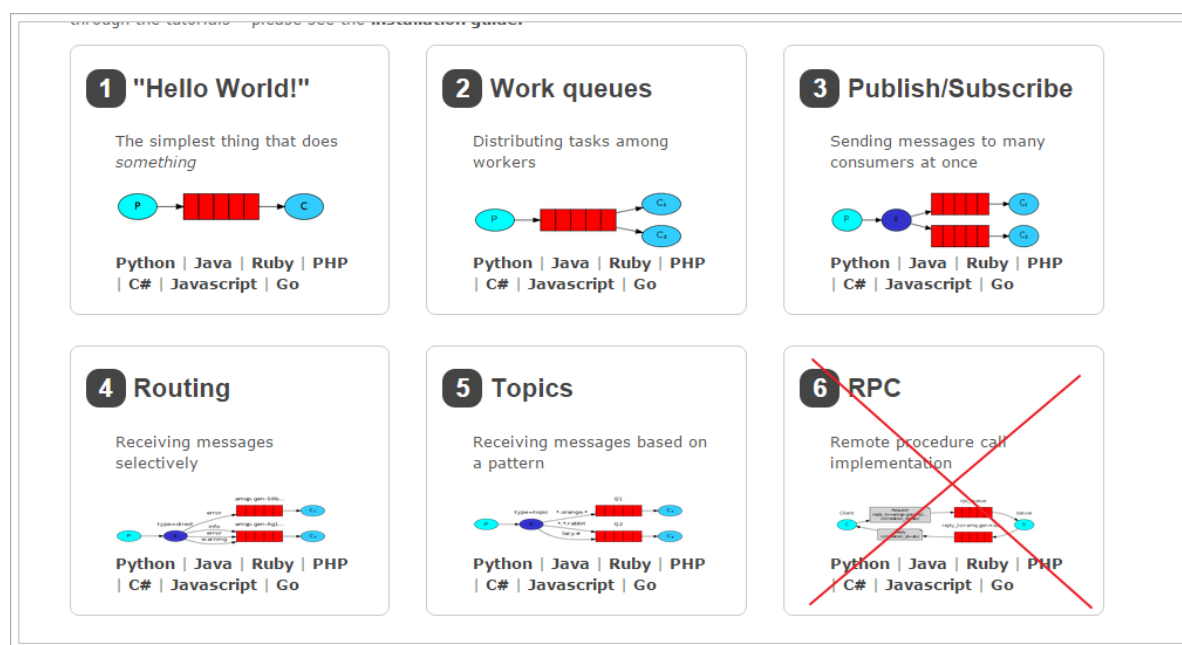


RabbitMQ 中的相关概念：

- Broker: 接收和分发消息的应用, RabbitMQ Server就是 Message Broker
- Virtual host: 出于多租户和安全因素设计的, 把 AMQP 的基本组件划分到一个虚拟的分组中, 类似于网络中的 namespace 概念。当多个不同的用户使用同一个 RabbitMQ server 提供的服务时, 可以划分出多个vhost, 每个用户在自己的 vhost 创建 exchange / queue 等
- Connection: publisher / consumer 和 broker 之间的 TCP 连接
- Channel: 如果每一次访问 RabbitMQ 都建立一个 Connection, 在消息量大的时候建立 TCP Connection的开销将是巨大的, 效率也较低。Channel 是在 connection 内部建立的逻辑连接, 如果应用程序支持多线程, 通常每个thread创建单独的 channel 进行通讯, AMQP method 包含了 channel id 帮助客户端和message broker 识别 channel, 所以 channel 之间是完全隔离的。Channel 作为轻量级的 Connection 极大减少了操作系统建立 TCP connection 的开销
- Exchange: message 到达 broker 的第一站, 根据分发规则, 匹配查询表中的 routing key, 分发消息到queue 中去。常用的类型有: direct (point-to-point), topic (publish-subscribe) and fanout (multicast)
- Queue: 消息最终被送到这里等待 consumer 取走
- Binding: exchange 和 queue 之间的虚拟连接, binding 中可以包含 routing key。Binding 信息被保存到 exchange 中的查询表中, 用于 message 的分发依据

RabbitMQ提供了6种模式: 简单模式, work模式, Publish/Subscribe发布与订阅模式, Routing路由模式, Topics主题模式, RPC远程调用模式(远程调用, 不太算MQ; 暂不作介绍);

官网对应模式介绍: <https://www.rabbitmq.com/getstarted.html>



2. 安装及配置RabbitMQ

详细查看 [资料/软件/安装RabbitMQ.md](#) 文档。

3. RabbitMQ入门



在上图的模型中，有以下概念：

- P：生产者，也就是要发送消息的程序
- C：消费者：消息的接收者，会一直等待消息到来
- queue：消息队列，图中红色部分。类似一个邮箱，可以缓存消息；生产者向其中投递消息，消费者从其中取出消息

3.1. 搭建示例工程

3.1.1. 创建工程

Parent:	<None>
Name:	mq-demo
Location:	E:\work\0_lxs\mq-demo
▼ Artifact Coordinates	
GroupId:	com.lxs
The name of the artifact group, usually a company domain	
ArtifactId:	mq-demo
The name of the artifact within the group, usually a module name	
Version:	1.0-SNAPSHOT

3.1.2. 添加依赖

往lxs-rabbitmq的pom.xml文件中添加如下依赖：

```
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.6.0</version>
</dependency>
```

3.2. 编写生产者

编写消息生产者com.lxs.rabbitmq.simple.Producer

```
package com.lxs.rabbitmq.simple;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
```



```

public class Producer {

    static final String QUEUE_NAME = "simple_queue";

    public static void main(String[] args) throws Exception {
        //创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //主机地址;默认为 localhost
        connectionFactory.setHost("localhost");
        //连接端口;默认为 5672
        connectionFactory.setPort(5672);
        //虚拟主机名称;默认为 /
        connectionFactory.setVirtualHost("/1xs");
        //连接用户名;默认为guest
        connectionFactory.setUsername("1xs");
        //连接密码;默认为guest
        connectionFactory.setPassword("1xs");

        //创建连接
        Connection connection = connectionFactory.newConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        // 要发送的信息
        String message = "你好; 小兔子! ";
        /**
         * 参数1: 交换机名称, 如果没有指定则使用默认Default Exchange
         * 参数2: 路由key, 简单模式可以传递队列名称
         * 参数3: 消息其它属性
         * 参数4: 消息内容
         */
        channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
        System.out.println("已发送消息: " + message);

        // 关闭资源
        channel.close();
        connection.close();
    }
}

```

在执行上述的消息发送之后; 可以登录rabbitMQ的管理控制台, 可以发现队列和其消息:

Overview Connections Channels Exchanges **Queues** Admin

Queues

▼ All queues (1)

Pagination

Page 1 ▼ of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates				+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
/itcast	simple_queue	D	idle	1	0	1	0.00/s				

Overview Connections Channels Exchanges **Queues** Admin

Warning: getting messages from a queue is a destructive action. ?

Ack Mode: ▼

Encoding: ▼ ?

Messages:

Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange	(AMQP default)
Routing Key	simple_queue
Redelivered	0
Properties	
Payload	你好; 小兔子!
21 bytes	
Encoding: string	

3.3. 编写消费者

抽取创建connection的工具类com.lxs.rabbitmq.util.ConnectionUtil;

```
package com.lxs.rabbitmq.util;

import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class ConnectionUtil {

    public static Connection getConnection() throws Exception {
        //创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //主机地址;默认为 localhost
        connectionFactory.setHost("localhost");
        //连接端口;默认为 5672
        connectionFactory.setPort(5672);
        //虚拟主机名称;默认为 /
    }
}
```

```

        connectionFactory.setVirtualHost("/lxs");
        //连接用户名：默认为guest
        connectionFactory.setUsername("lxs");
        //连接密码：默认为guest
        connectionFactory.setPassword("lxs");

        //创建连接
        return connectionFactory.newConnection();
    }
}

```

编写消息的消费者com.lxs.rabbitmq.simple.Consumer

```

package com.lxs.rabbitmq.simple;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(Producer.QUEUE_NAME, true, false, false, null);

        //创建消费者：并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签，在channel.basicConsume时候可以指定
             * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
             标志(收到消息失败后是否需要重新发送)
             * properties 属性信息
             * body 消息
             */
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                //路由key
            }
        };
    }
}

```

```

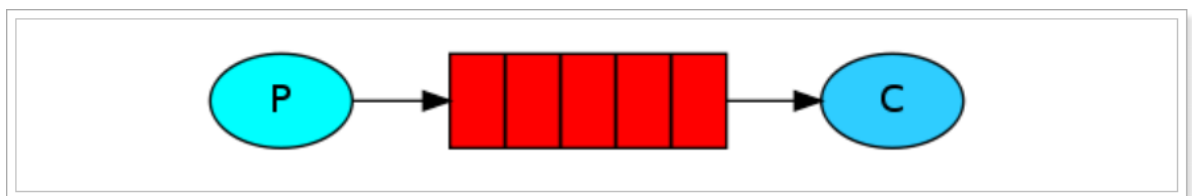
        System.out.println("路由key为: " + envelope.getRoutingKey());
        //交换机
        System.out.println("交换机为: " + envelope.getExchange());
        //消息id
        System.out.println("消息id为: " + envelope.getDeliveryTag());
        //收到的消息
        System.out.println("接收到的消息为: " + new String(body, "utf-8"));
    }
};
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认, 设置为true为表示消息接收到自动向mq回复接收到了, mq接收到回复
会删除消息, 设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.QUEUE_NAME, true, consumer);

//不关闭资源, 应该一直监听消息
//channel.close();
//connection.close();
}
}

```

3.4. 小结

上述的入门案例中其实使用的是如下的简单模式：



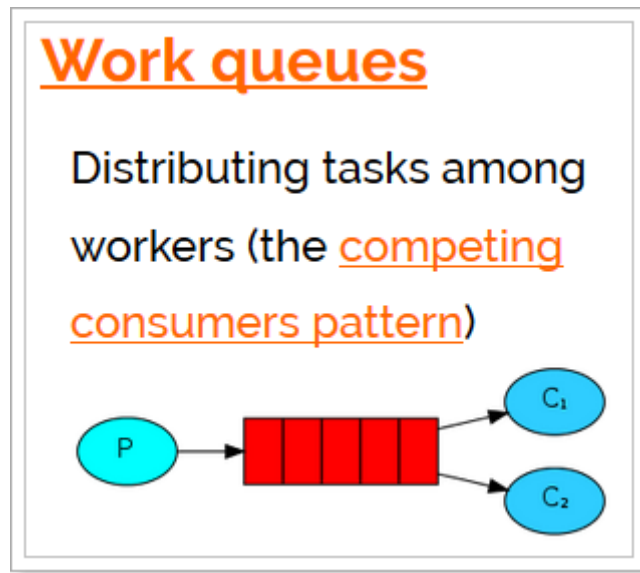
在上图的模型中，有以下概念：

- P：生产者，也就是要发送消息的程序
- C：消费者：消息的接受者，会一直等待消息到来。
- queue：消息队列，图中红色部分。类似一个邮箱，可以缓存消息；生产者向其中投递消息，消费者从其中取出消息。

5. RabbitMQ工作模式

4.1. Work queues工作队列模式

4.1.1. 模式说明



work queues 与入门程序的简单模式相比，多了一个或一些消费端，多个消费端共同消费同一个队列中的消息。

应用场景：对于 任务过重或任务较多情况使用工作队列可以提高任务处理的速度。

4.1.2. 代码

work queues 与入门程序的简单模式的代码是几乎一样的；可以完全复制，并复制多一个消费者进行多个消费者同时消费消息的测试。

1) 生产者

```
package com.lxs.rabbitmq.work;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Producer {

    static final String QUEUE_NAME = "work_queue";

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

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        // 声明（创建）队列
        /**
         * 参数1：队列名称
         * 参数2：是否定义持久化队列
         * 参数3：是否独占本次连接
         * 参数4：是否在不使用的时候自动删除队列
        */
    }
}
```

```

    * 参数5: 队列其它参数
    */
channel.queueDeclare(Queue.NAME, true, false, false, null);

for (int i = 1; i <= 30; i++) {
    // 发送信息
    String message = "你好; 小兔子! work模式--" + i;
    /**
     * 参数1: 交换机名称, 如果没有指定则使用默认Default Exchange
     * 参数2: 路由key, 简单模式可以传递队列名称
     * 参数3: 消息其它属性
     * 参数4: 消息内容
     */
    channel.basicPublish("", Queue.NAME, null, message.getBytes());
    System.out.println("已发送消息: " + message);
}

// 关闭资源
channel.close();
connection.close();
}
}

```

2) 消费者1

```

package com.lxs.rabbitmq.work;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        // 声明(创建)队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(Producer.Queue.NAME, true, false, false, null);

        //一次只能接收并处理一个消息
        channel.basicQos(1);

        //创建消费者; 并设置消息处理
    }
}

```

```

DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签, 在channel.basicConsume时候可以指定
     * envelope 消息包的内容, 可从中获取消息id, 消息routingkey, 交换机, 消息和重传
    标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        try {
            //路由key
            System.out.println("路由key为: " + envelope.getRoutingKey());
            //交换机
            System.out.println("交换机为: " + envelope.getExchange());
            //消息id
            System.out.println("消息id为: " + envelope.getDeliveryTag());
            //收到的消息
            System.out.println("消费者1-接收到的消息为: " + new String(body,
"utf-8"));

            Thread.sleep(1000);

            //确认消息
            channel.basicAck(envelope.getDeliveryTag(), false);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认, 设置为true为表示消息接收到自动向mq回复接收到了, mq接收到回复
    会删除消息, 设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.QUEUE_NAME, false, consumer);
}
}

```

3) 消费者2

```

package com.lxs.rabbitmq.work;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
    }
}

```

```

// 创建频道
channel channel = connection.createChannel();

// 声明（创建）队列
/**
 * 参数1: 队列名称
 * 参数2: 是否定义持久化队列
 * 参数3: 是否独占本次连接
 * 参数4: 是否在不使用的时候自动删除队列
 * 参数5: 队列其它参数
 */
channel.queueDeclare(Producer.QUEUE_NAME, true, false, false, null);

//一次只能接收并处理一个消息
channel.basicQos(1);

//创建消费者；并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，在channel.basicConsume时候可以指定
     * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        try {
            //路由key
            System.out.println("路由key为: " + envelope.getRoutingKey());
            //交换机
            System.out.println("交换机为: " + envelope.getExchange());
            //消息id
            System.out.println("消息id为: " + envelope.getDeliveryTag());
            //收到的消息
            System.out.println("消费者2-接收到的消息为: " + new String(body,
"utf-8"));

            Thread.sleep(1000);

            //确认消息
            channel.basicAck(envelope.getDeliveryTag(), false);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};

//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.QUEUE_NAME, false, consumer);

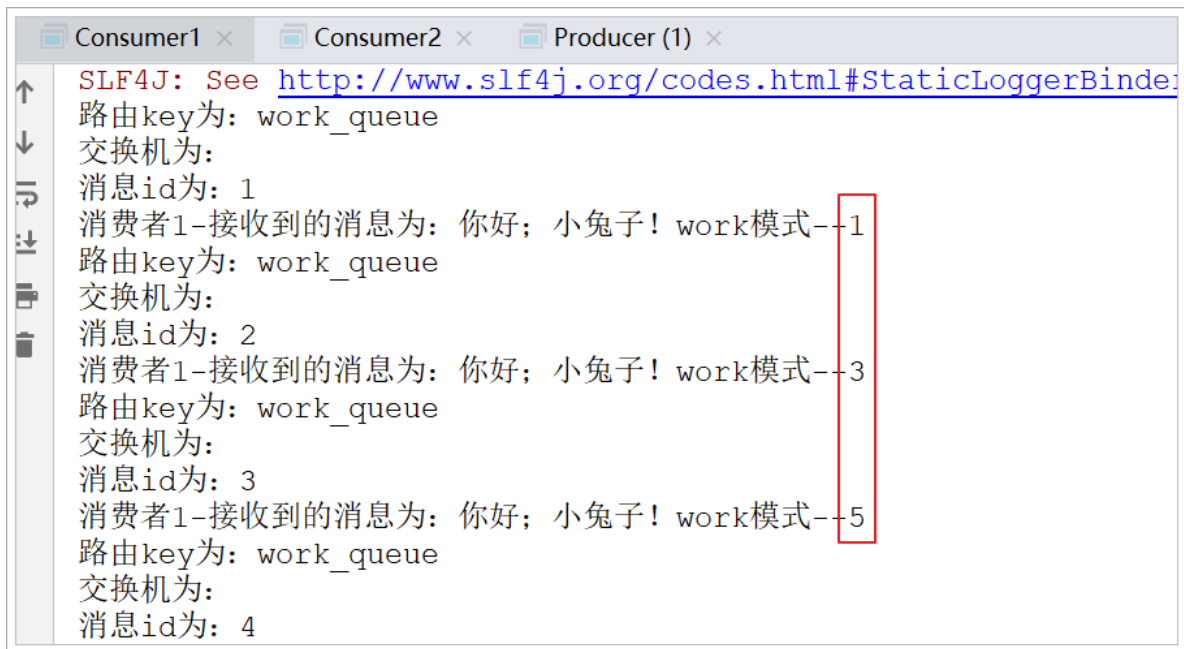
```



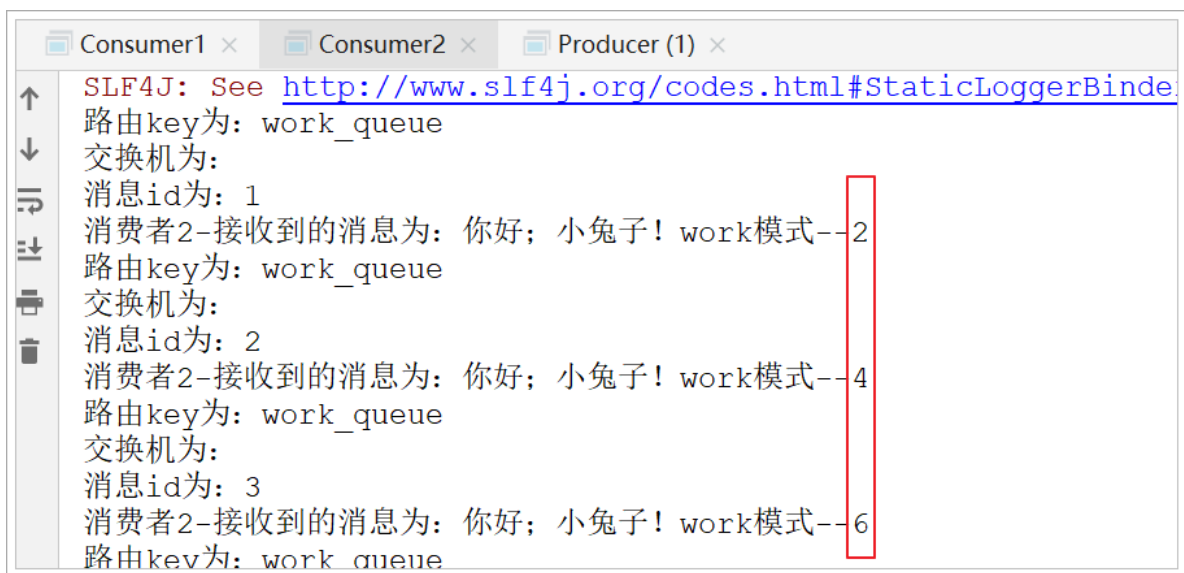
```
}  
}
```

4.1.3. 测试

启动两个消费者，然后再启动生产者发送消息；到IDEA的两个消费者对应的控制台查看是否竞争性的接收到消息。



```
Consumer1 x Consumer2 x Producer (1) x  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder  
路由key为: work_queue  
交换机为:  
消息id为: 1  
消费者1-接收到的消息为: 你好; 小兔子! work模式--1  
路由key为: work_queue  
交换机为:  
消息id为: 2  
消费者1-接收到的消息为: 你好; 小兔子! work模式--3  
路由key为: work_queue  
交换机为:  
消息id为: 3  
消费者1-接收到的消息为: 你好; 小兔子! work模式--5  
路由key为: work_queue  
交换机为:  
消息id为: 4
```



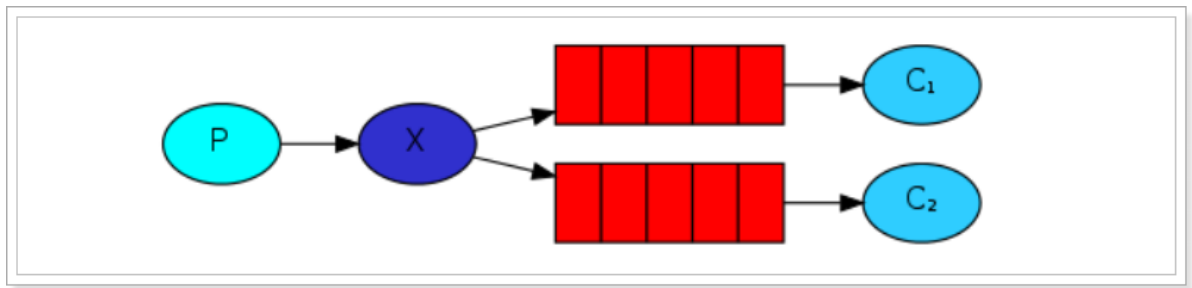
```
Consumer1 x Consumer2 x Producer (1) x  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder  
路由key为: work_queue  
交换机为:  
消息id为: 1  
消费者2-接收到的消息为: 你好; 小兔子! work模式--2  
路由key为: work_queue  
交换机为:  
消息id为: 2  
消费者2-接收到的消息为: 你好; 小兔子! work模式--4  
路由key为: work_queue  
交换机为:  
消息id为: 3  
消费者2-接收到的消息为: 你好; 小兔子! work模式--6  
路由key为: work_queue
```

4.1.4. 小结

在一个队列中如果有多个消费者，那么消费者之间对于同一个消息的关系是**竞争**的关系。

4.2. 订阅模式概述

订阅模式示例图：



前面2个案例中，只有3个角色：

- P：生产者，也就是要发送消息的程序
- C：消费者：消息的接受者，会一直等待消息到来。
- queue：消息队列，图中红色部分

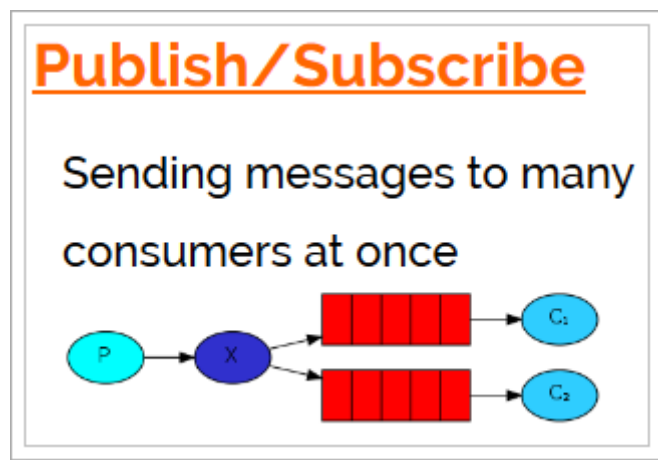
而在订阅模型中，多了一个exchange角色，而且过程略有变化：

- P：生产者，也就是要发送消息的程序，但是不再发送到队列中，而是发给X（交换机）
- C：消费者，消息的接受者，会一直等待消息到来。
- Queue：消息队列，接收消息、缓存消息。
- Exchange：交换机，图中的X。一方面，接收生产者发送的消息。另一方面，知道如何处理消息，例如递交给某个特别队列、递交给所有队列、或是将消息丢弃。到底如何操作，取决于Exchange的类型。Exchange有常见以下3种类型：
 - Fanout：广播，将消息交给所有绑定到交换机的队列
 - Direct：定向，把消息交给符合指定routing key 的队列
 - Topic：通配符，把消息交给符合routing pattern（路由模式）的队列

Exchange（交换机）只负责转发消息，不具备存储消息的能力，因此如果没有任何队列与Exchange绑定，或者没有符合路由规则的队列，那么消息会丢失！

4.3. Publish/Subscribe发布与订阅模式

4.3.1. 模式说明



发布订阅模式：

- 1、每个消费者监听自己的队列。
- 2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收

4.3.2. 代码

1) 生产者

```
package com.lxs.rabbitmq.ps;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;

/**
 * 发布与订阅使用的交换机类型为: fanout
 */
public class Producer {

    //交换机名称
    static final String FANOUT_EXCHANGE = "fanout_exchange";
    //队列名称
    static final String FANOUT_QUEUE_1 = "fanout_queue_1";
    //队列名称
    static final String FANOUT_QUEUE_2 = "fanout_queue_2";

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

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        /**
         * 声明交换机
         * 参数1: 交换机名称
         * 参数2: 交换机类型, fanout、topic、direct、headers
         */
        channel.exchangeDeclare(FANOUT_EXCHANGE, BuiltinExchangeType.FANOUT);

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(FANOUT_QUEUE_1, true, false, false, null);
        channel.queueDeclare(FANOUT_QUEUE_2, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(FANOUT_QUEUE_1, FANOUT_EXCHANGE, "");
    }
}
```

```

channel.queueBind(FANOUT_QUEUE_2, FANOUT_EXCHANGE, "");

for (int i = 1; i <= 10; i++) {
    // 发送信息
    String message = "你好，小兔子！发布订阅模式--" + i;
    /**
     * 参数1: 交换机名称，如果没有指定则使用默认Default Exchange
     * 参数2: 路由key,简单模式可以传递队列名称
     * 参数3: 消息其它属性
     * 参数4: 消息内容
     */
    channel.basicPublish(FANOUT_EXCHANGE, "", null, message.getBytes());
    System.out.println("已发送消息: " + message);
}

// 关闭资源
channel.close();
connection.close();
}
}

```

2) 消费者1

```

package com.lxs.rabbitmq.ps;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.FANOUT_EXCHANGE,
            BuiltinExchangeType.FANOUT);

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(Producer.FANOUT_QUEUE_1, true, false, false, null);

        //队列绑定交换机
    }
}

```

```

channel.queueBind(Producer.FANOUT_QUEUE_1, Producer.FANOUT_EXCHANGE, "");

//创建消费者：并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，在channel.basicConsume时候可以指定
     * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        //路由key
        System.out.println("路由key为: " + envelope.getRoutingKey());
        //交换机
        System.out.println("交换机为: " + envelope.getExchange());
        //消息id
        System.out.println("消息id为: " + envelope.getDeliveryTag());
        //收到的消息
        System.out.println("消费者1-接收到的消息为: " + new String(body,
"utf-8"));
    }
};
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.FANOUT_QUEUE_1, true, consumer);
}
}

```

3) 消费者2

```

package com.lxs.rabbitmq.ps;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机

```

```

channel.exchangeDeclare(Producer.FANOUT_EXCHANGE,
BuiltinExchangeType.FANOUT);

// 声明（创建）队列
/**
 * 参数1: 队列名称
 * 参数2: 是否定义持久化队列
 * 参数3: 是否独占本次连接
 * 参数4: 是否在不使用的时候自动删除队列
 * 参数5: 队列其它参数
 */
channel.queueDeclare(Producer.FANOUT_QUEUE_2, true, false, false, null);

//队列绑定交换机
channel.queueBind(Producer.FANOUT_QUEUE_2, Producer.FANOUT_EXCHANGE, "");

//创建消费者；并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，在channel.basicConsume时候可以指定
     * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        //路由key
        System.out.println("路由key为: " + envelope.getRoutingKey());
        //交换机
        System.out.println("交换机为: " + envelope.getExchange());
        //消息id
        System.out.println("消息id为: " + envelope.getDeliveryTag());
        //收到的消息
        System.out.println("消费者2-接收到的消息为: " + new String(body,
"utf-8"));
    }
};

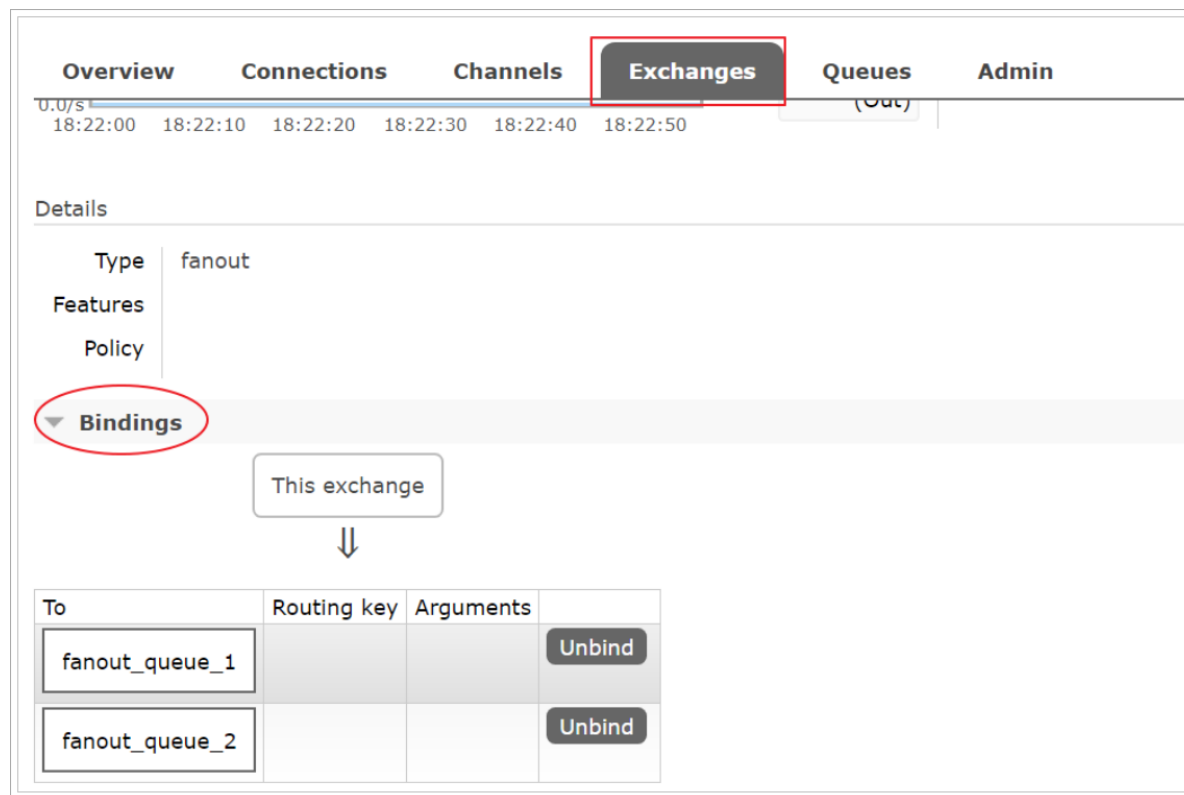
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.FANOUT_QUEUE_2, true, consumer);
}
}

```

4.3.3. 测试

启动所有消费者，然后使用生产者发送消息；在每个消费者对应的控制台可以查看到生产者发送的所有消息；到达广播的效果。

在执行完测试代码后，其实到RabbitMQ的管理后台找到 Exchanges 选项卡，点击 fanout_exchange 的交换机，可以查看到如下的绑定：



4.3.4. 小结

交换机需要与队列进行绑定，绑定之后；一个消息可以被多个消费者都收到。

发布订阅模式与工作队列模式的区别

- 1、工作队列模式不用定义交换机，而发布/订阅模式需要定义交换机。
- 2、发布/订阅模式的生产方是面向交换机发送消息，工作队列模式的生产方是面向队列发送消息(底层使用默认交换机)。
- 3、发布/订阅模式需要设置队列和交换机的绑定，工作队列模式不需要设置，实际上工作队列模式会将队列绑定到默认的交换机。

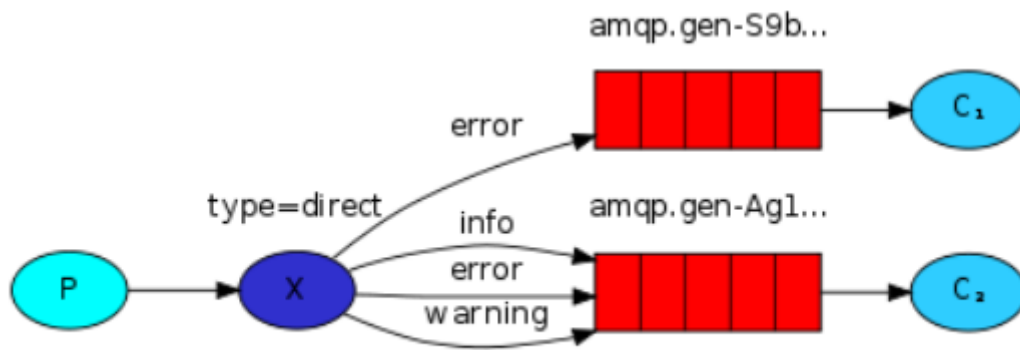
4.4. Routing路由模式

4.4.1. 模式说明

路由模式特点：

- 队列与交换机的绑定，不能是任意绑定了，而是要指定一个 `RoutingKey`（路由key）
- 消息的发送方在向 Exchange 发送消息时，也必须指定消息的 `RoutingKey`。
- Exchange 不再把消息交给每一个绑定的队列，而是根据消息的 `Routing Key` 进行判断，只有队列的 `Routingkey` 与消息的 `Routing key` 完全一致，才会接收到消息

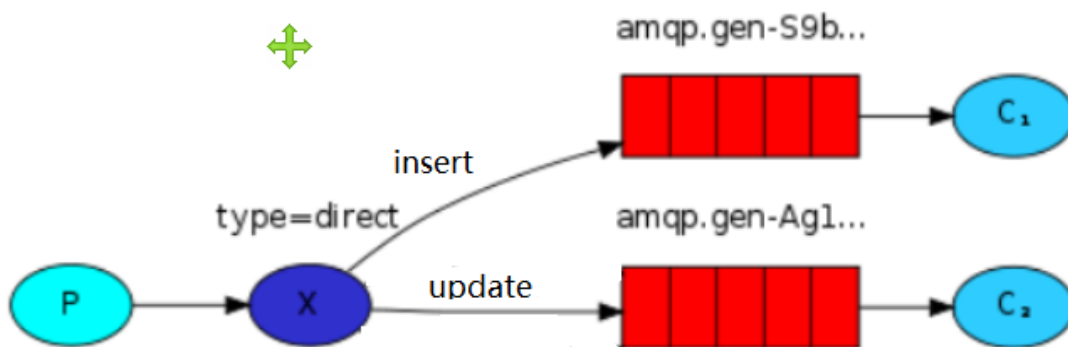
Receiving messages selectively



图解：

- P: 生产者，向Exchange发送消息，发送消息时，会指定一个routing key。
- X: Exchange（交换机），接收生产者的消息，然后把消息递交给与routing key完全匹配的队列
- C1: 消费者，其所在队列指定了需要routing key 为 error 的消息
- C2: 消费者，其所在队列指定了需要routing key 为 info、error、warning 的消息

4.4.2. 代码



在编码上与 Publish/Subscribe发布与订阅模式 的区别是交换机的类型为：Direct，还有队列绑定交换机的时候需要指定routing key。

1) 生产者

```
package com.lxs.rabbitmq.routing;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
```



```

import com.rabbitmq.client.Connection;

/**
 * 路由模式的交换机类型为: direct
 */
public class Producer {

    //交换机名称
    static final String DIRECT_EXCHANGE = "direct_exchange";
    //队列名称
    static final String DIRECT_QUEUE_INSERT = "direct_queue_insert";
    //队列名称
    static final String DIRECT_QUEUE_UPDATE = "direct_queue_update";

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

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        /**
         * 声明交换机
         * 参数1: 交换机名称
         * 参数2: 交换机类型, fanout、topic、direct、headers
         */
        channel.exchangeDeclare(DIRECT_EXCHANGE, BuiltinExchangeType.DIRECT);

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(DIRECT_QUEUE_INSERT, true, false, false, null);
        channel.queueDeclare(DIRECT_QUEUE_UPDATE, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(DIRECT_QUEUE_INSERT, DIRECT_EXCHANGE, "insert");
        channel.queueBind(DIRECT_QUEUE_UPDATE, DIRECT_EXCHANGE, "update");

        // 发送信息
        String message = "新增了商品。路由模式: routing key 为 insert ";
        /**
         * 参数1: 交换机名称, 如果没有指定则使用默认Default Exchange
         * 参数2: 路由key, 简单模式可以传递队列名称
         * 参数3: 消息其它属性
         * 参数4: 消息内容
         */
        channel.basicPublish(DIRECT_EXCHANGE, "insert", null,
message.getBytes());
        System.out.println("已发送消息: " + message);
    }
}

```

```

// 发送信息
message = "修改了商品。路由模式: routing key 为 update" ;
/**
 * 参数1: 交换机名称, 如果没有指定则使用默认Default Exchange
 * 参数2: 路由key, 简单模式可以传递队列名称
 * 参数3: 消息其它属性
 * 参数4: 消息内容
 */
channel.basicPublish(DIRECT_EXCHANGE, "update", null,
message.getBytes());
System.out.println("已发送消息: " + message);

// 关闭资源
channel.close();
connection.close();
}
}

```

2) 消费者1

```

package com.lxs.rabbitmq.routing;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.DIRECT_EXCHANGE,
BuiltinExchangeType.DIRECT);

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
        channel.queueDeclare(Producer.DIRECT_QUEUE_INSERT, true, false, false,
null);

        //队列绑定交换机
        channel.queueBind(Producer.DIRECT_QUEUE_INSERT, Producer.DIRECT_EXCHANGE,
"insert");
    }
}

```

```

//创建消费者：并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，在channel.basicConsume时候可以指定
     * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        //路由key
        System.out.println("路由key为: " + envelope.getRoutingKey());
        //交换机
        System.out.println("交换机为: " + envelope.getExchange());
        //消息id
        System.out.println("消息id为: " + envelope.getDeliveryTag());
        //收到的消息
        System.out.println("消费者1-接收到的消息为: " + new String(body,
"utf-8"));
    }
};
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.DIRECT_QUEUE_INSERT, true, consumer);
}
}

```

3) 消费者2

```

package com.lxs.rabbitmq.routing;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机

```

```

channel.exchangeDeclare(Producer.DIRECT_EXCHANGE,
BuiltinExchangeType.DIRECT);

// 声明（创建）队列
/**
 * 参数1: 队列名称
 * 参数2: 是否定义持久化队列
 * 参数3: 是否独占本次连接
 * 参数4: 是否在不使用的时候自动删除队列
 * 参数5: 队列其它参数
 */
channel.queueDeclare(Producer.DIRECT_QUEUE_UPDATE, true, false, false,
null);

//队列绑定交换机
channel.queueBind(Producer.DIRECT_QUEUE_UPDATE, Producer.DIRECT_EXCHANGE,
"update");

//创建消费者：并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，在channel.basicConsume时候可以指定
     * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        //路由key
        System.out.println("路由key为: " + envelope.getRoutingKey());
        //交换机
        System.out.println("交换机为: " + envelope.getExchange());
        //消息id
        System.out.println("消息id为: " + envelope.getDeliveryTag());
        //收到的消息
        System.out.println("消费者2-接收到的消息为: " + new String(body,
"utf-8"));
    }
};

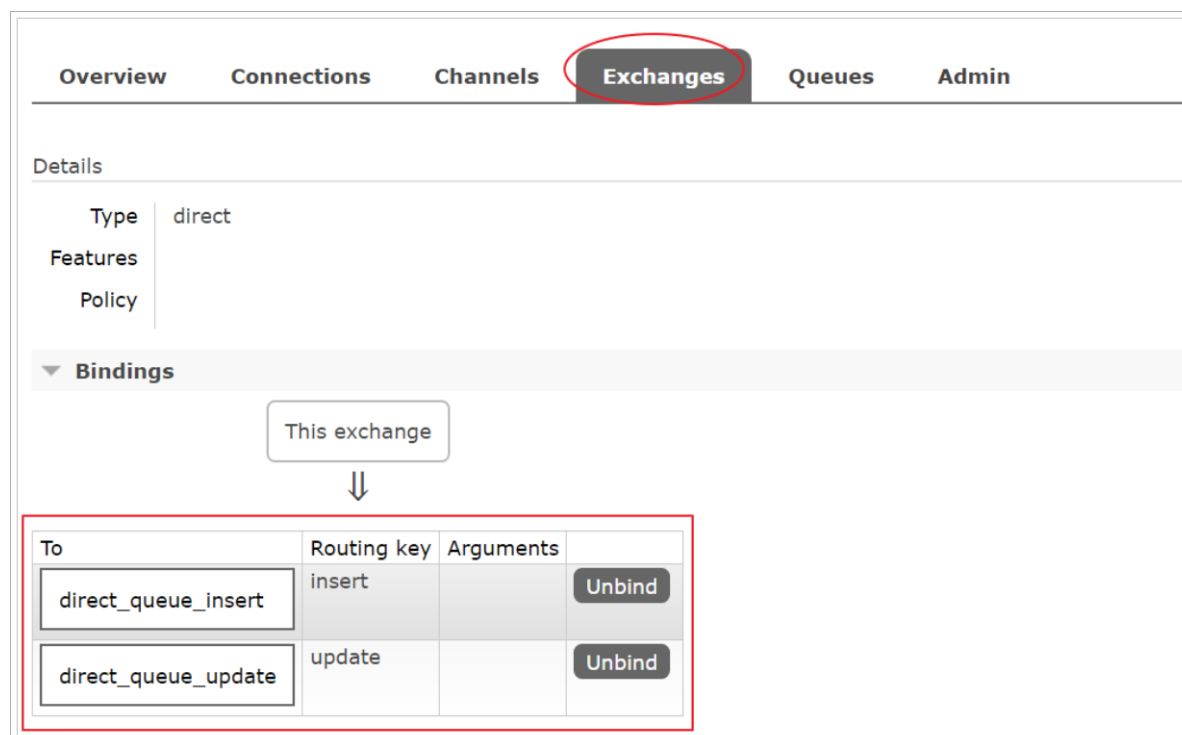
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.DIRECT_QUEUE_UPDATE, true, consumer);
}
}

```

4.4.3. 测试

启动所有消费者，然后使用生产者发送消息；在消费者对应的控制台可以查看到生产者发送对应routing key对应队列的消息；到达**按照需要接收**的效果。

在执行完测试代码后，其实到RabbitMQ的管理后台找到 Exchanges 选项卡，点击 `direct_exchange` 的交换机，可以查看到如下的绑定：



4.4.4. 小结

Routing模式要求队列在绑定交换机时要指定routing key，消息会转发到符合routing key的队列。

4.5. Topics通配符模式

4.5.1. 模式说明

Topic 类型与 Direct 相比，都是可以根据 RoutingKey 把消息路由到不同的队列。只不过 Topic 类型 Exchange 可以让队列在绑定 Routing key 的时候**使用通配符**！

Routingkey 一般都是有一个或多个单词组成，多个单词之间以"."分割，例如：`item.insert`

通配符规则：

#：匹配一个或多个词

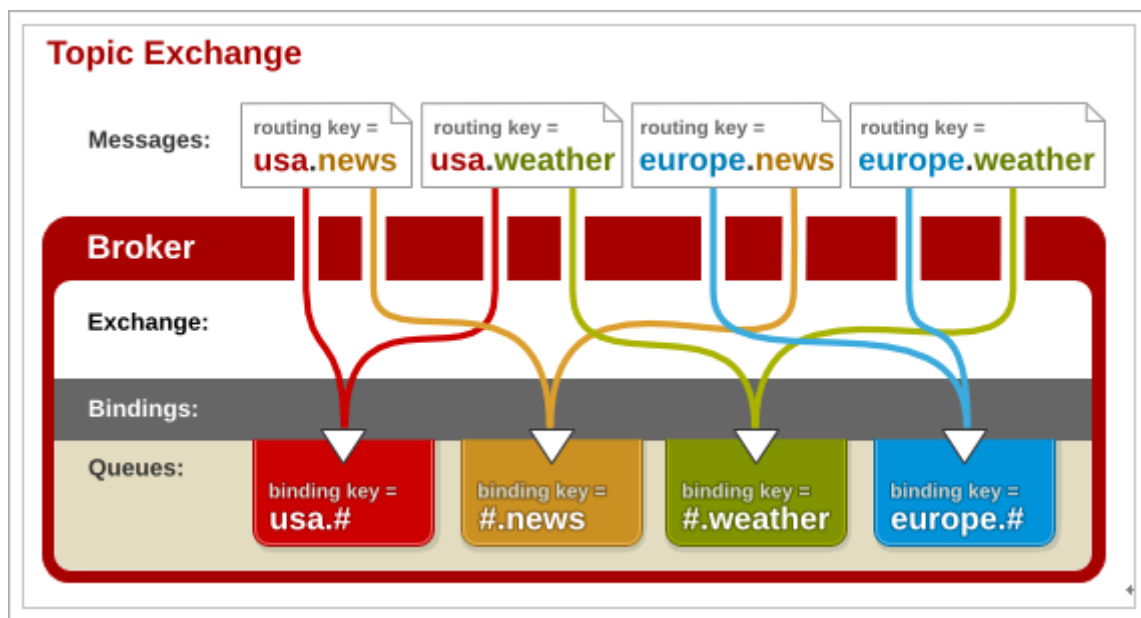
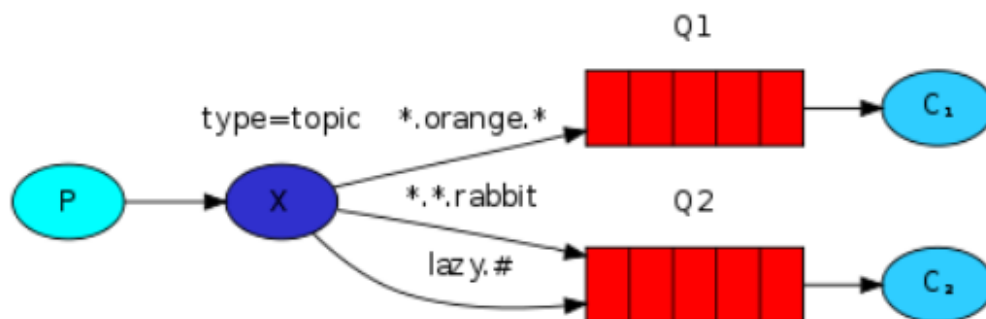
*：匹配不多不少恰好1个词

举例：

`item.#`：能够匹配 `item.insert.abc` 或者 `item.insert`

`item.*`：只能匹配 `item.insert`

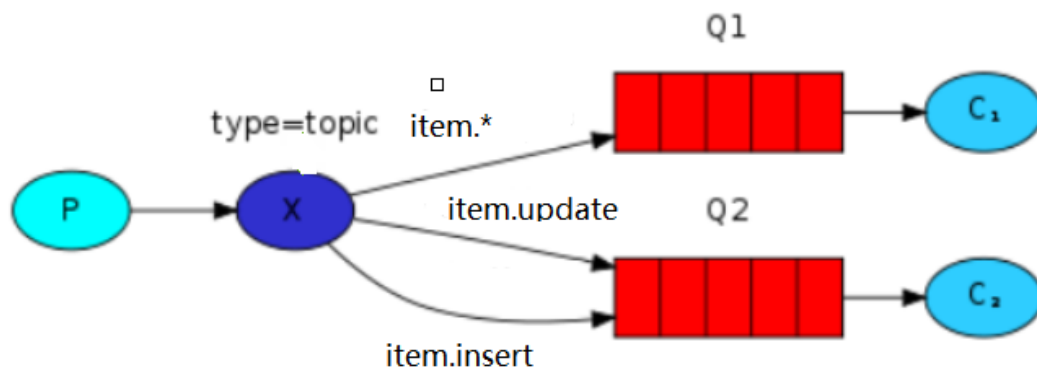
Receiving messages based on a pattern (topics)



图解：

- 红色Queue：绑定的是 `usa.#`，因此凡是以 `usa.` 开头的 `routing key` 都会被匹配到
- 黄色Queue：绑定的是 `#.news`，因此凡是以 `.news` 结尾的 `routing key` 都会被匹配

4.5.2. 代码



1) 生产者

使用topic类型的Exchange，发送消息的routing key有3种：`item.insert`、`item.update`、`item.delete`：

```

package com.lxs.rabbitmq.topic;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;

/**
 * 通配符Topic的交换机类型为: topic
 */
public class Producer {

    //交换机名称
    static final String TOPIC_EXCHANGE = "topic_exchange";
    //队列名称
    static final String TOPIC_QUEUE_1 = "topic_queue_1";
    //队列名称
    static final String TOPIC_QUEUE_2 = "topic_queue_2";

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

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        /**
         * 声明交换机
         * 参数1: 交换机名称
         * 参数2: 交换机类型, fanout、topic、topic、headers
         */
        channel.exchangeDeclare(TOPIC_EXCHANGE, BuiltinExchangeType.TOPIC);

        // 发送信息
    }
  
```

```

String message = "新增了商品。Topic模式: routing key 为 item.insert ";
channel.basicPublish(TOPIC_EXCHANGE, "item.insert", null,
message.getBytes());
System.out.println("已发送消息: " + message);

// 发送信息
message = "修改了商品。Topic模式: routing key 为 item.update";
channel.basicPublish(TOPIC_EXCHANGE, "item.update", null,
message.getBytes());
System.out.println("已发送消息: " + message);

// 发送信息
message = "删除了商品。Topic模式: routing key 为 item.delete";
channel.basicPublish(TOPIC_EXCHANGE, "item.delete", null,
message.getBytes());
System.out.println("已发送消息: " + message);

// 关闭资源
channel.close();
connection.close();
}
}

```

2) 消费者1

接收两种类型的消息：更新商品和删除商品

```

package com.lxs.rabbitmq.topic;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.TOPIC_EXCHANGE,
        BuiltinExchangeType.TOPIC);

        // 声明（创建）队列
        /**
         * 参数1: 队列名称
         * 参数2: 是否定义持久化队列
         * 参数3: 是否独占本次连接
         * 参数4: 是否在不使用的时候自动删除队列
         * 参数5: 队列其它参数
         */
    }
}

```



```

channel.queueDeclare(Producer.TOPIC_QUEUE_1, true, false, false, null);

//队列绑定交换机
channel.queueBind(Producer.TOPIC_QUEUE_1, Producer.TOPIC_EXCHANGE,
"item.update");
channel.queueBind(Producer.TOPIC_QUEUE_1, Producer.TOPIC_EXCHANGE,
"item.delete");

//创建消费者；并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，在channel.basicConsume时候可以指定
     * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
标志(收到消息失败后是否需要重新发送)
     * properties 属性信息
     * body 消息
     */
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        //路由key
        System.out.println("路由key为: " + envelope.getRoutingKey());
        //交换机
        System.out.println("交换机为: " + envelope.getExchange());
        //消息id
        System.out.println("消息id为: " + envelope.getDeliveryTag());
        //收到的消息
        System.out.println("消费者1-接收到的消息为: " + new String(body,
"utf-8"));
    }
};

//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.TOPIC_QUEUE_1, true, consumer);
}
}

```

3) 消费者2

接收所有类型的消息：新增商品，更新商品和删除商品。

```

package com.lxs.rabbitmq.topic;

import com.lxs.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

```

```

public static void main(String[] args) throws Exception {
    Connection connection = ConnectionUtil.getConnection();

    // 创建频道
    Channel channel = connection.createChannel();

    //声明交换机
    channel.exchangeDeclare(Producer.TOPIC_EXCHANGE,
        BuiltinExchangeType.TOPIC);

    // 声明（创建）队列
    /**
     * 参数1: 队列名称
     * 参数2: 是否定义持久化队列
     * 参数3: 是否独占本次连接
     * 参数4: 是否在不使用的时候自动删除队列
     * 参数5: 队列其它参数
     */
    channel.queueDeclare(Producer.TOPIC_QUEUE_2, true, false, false, null);

    //队列绑定交换机
    channel.queueBind(Producer.TOPIC_QUEUE_2, Producer.TOPIC_EXCHANGE,
        "item.*");

    //创建消费者：并设置消息处理
    DefaultConsumer consumer = new DefaultConsumer(channel){
        @Override
        /**
         * consumerTag 消息者标签，在channel.basicConsume时候可以指定
         * envelope 消息包的内容，可从中获取消息id，消息routingkey，交换机，消息和重传
         标志(收到消息失败后是否需要重新发送)
         * properties 属性信息
         * body 消息
         */
        public void handleDelivery(String consumerTag, Envelope envelope,
            AMQP.BasicProperties properties, byte[] body) throws IOException {
            //路由key
            System.out.println("路由key为: " + envelope.getRoutingKey());
            //交换机
            System.out.println("交换机为: " + envelope.getExchange());
            //消息id
            System.out.println("消息id为: " + envelope.getDeliveryTag());
            //收到的消息
            System.out.println("消费者2-接收到的消息为: " + new String(body,
                "utf-8"));
        }
    };

    //监听消息
    /**
     * 参数1: 队列名称
     * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
     会删除消息，设置为false则需要手动确认
     * 参数3: 消息接收到后回调
     */
}

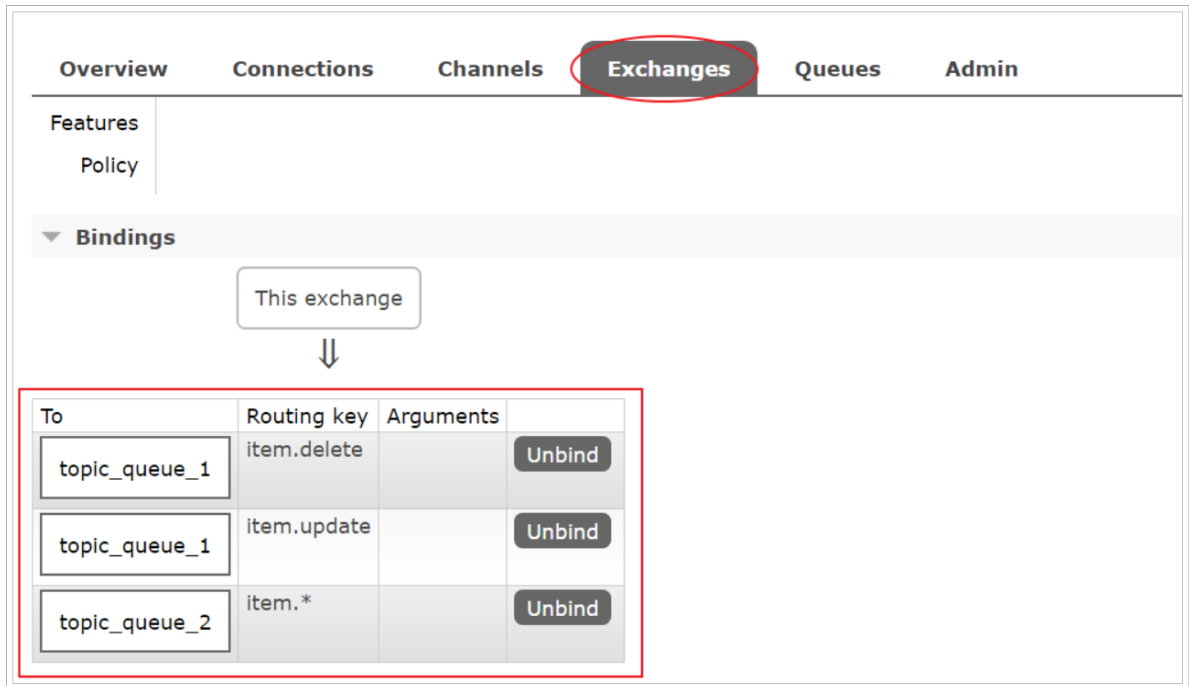
```

```
channel.basicConsume(Producer.TOPIC_QUEUE_2, true, consumer);  
}  
}
```

4.5.3. 测试

启动所有消费者，然后使用生产者发送消息；在消费者对应的控制台可以查看到生产者发送对应routing key对应队列的消息；到达按照需要接收的效果；并且这些routing key可以使用通配符。

在执行完测试代码后，其实到RabbitMQ的管理后台找到 Exchanges 选项卡，点击 topic_exchange 的交换机，可以查看到如下的绑定：



4.5.4. 小结

Topic主题模式可以实现 Publish/Subscribe发布与订阅模式和 Routing路由模式的功能；只是Topic在配置routing key的时候可以使用通配符，显得更加灵活。

4.6. 模式总结

RabbitMQ工作模式：

1、简单模式 HelloWorld

一个生产者、一个消费者，不需要设置交换机（使用默认的交换机）

2、工作队列模式 Work Queue

一个生产者、多个消费者（竞争关系），不需要设置交换机（使用默认的交换机）

3、发布订阅模式 Publish/subscribe

需要设置类型为fanout的交换机，并且交换机和队列进行绑定，当发送消息到交换机后，交换机会将消息发送到绑定的队列

4、路由模式 Routing

需要设置类型为direct的交换机，交换机和队列进行绑定，并且指定routing key，当发送消息到交换机后，交换机会根据routing key将消息发送到对应的队列

5、通配符模式 Topic

需要设置类型为topic的交换机，交换机和队列进行绑定，并且指定通配符方式的routing key，当发送消息到交换机后，交换机会根据routing key将消息发送到对应的队列

5. Spring 整合RabbitMQ

5.1. 搭建生产者工程

5.1.1. 创建工程

Parent:	<None>
Name:	spring-rabbitmq-producer
Location:	E:\work\w1\spring-rabbitmq-producer
▼ Artifact Coordinates	
GroupId:	com.lxs
The name of the artifact group, usually a company domain	
ArtifactId:	spring-rabbitmq-producer
The name of the artifact within the group, usually a module name	
Version:	1.0-SNAPSHOT

5.1.2. 添加依赖

修改pom.xml文件内容为如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.lxs</groupId>
    <artifactId>spring-rabbitmq-producer</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.amqp</groupId>
            <artifactId>spring-rabbit</artifactId>
            <version>2.1.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

```

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
    </dependencies>

</project>

```

5.1.3. 配置整合

1. 创建 `spring-rabbitmq-producer\src\main\resources\properties\rabbitmq.properties` 连接参数等配置文件;

```

rabbitmq.host=192.168.56.110
rabbitmq.port=5672
rabbitmq.username=lx
rabbitmq.password=lx
rabbitmq.virtual-host=/lx

```

2. 创建 `spring-rabbitmq-producer\src\main\resources\spring\spring-rabbitmq.xml` 整合配置文件;

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:rabbit="http://www.springframework.org/schema/rabbit"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/rabbit
        http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">

    <!--加载配置文件-->
    <context:property-placeholder
        location="classpath:properties/rabbitmq.properties"/>

    <!-- 定义rabbitmq connectionFactory -->
    <rabbit:connection-factory id="connectionFactory" host="${rabbitmq.host}"
        port="${rabbitmq.port}"
        username="${rabbitmq.username}"
        password="${rabbitmq.password}"
        virtual-host="${rabbitmq.virtual-host}"/>

    <!--定义管理交换机、队列-->
    <rabbit:admin connection-factory="connectionFactory"/>

    <!--定义持久化队列，不存在则自动创建；不绑定到交换机则绑定到默认交换机
    默认交换机类型为direct，名字为：""，路由键为队列的名称

```

```

-->
<rabbit:queue id="spring_queue" name="spring_queue" auto-declare="true"/>

<!-- ~~~~~广播：所有队列都能收到消息
~~~~~ -->
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_fanout_queue_1" name="spring_fanout_queue_1" auto-
declare="true"/>

<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_fanout_queue_2" name="spring_fanout_queue_2" auto-
declare="true"/>

<!--定义广播类型交换机；并绑定上述两个队列-->
<rabbit:fanout-exchange id="spring_fanout_exchange"
name="spring_fanout_exchange" auto-declare="true">
    <rabbit:bindings>
        <rabbit:binding queue="spring_fanout_queue_1"/>
        <rabbit:binding queue="spring_fanout_queue_2"/>
    </rabbit:bindings>
</rabbit:fanout-exchange>

<!-- ~~~~~通配符：*匹配一个单词，#匹配多个单词
~~~~~ -->
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_topic_queue_star" name="spring_topic_queue_star"
auto-declare="true"/>
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_topic_queue_well" name="spring_topic_queue_well"
auto-declare="true"/>
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_topic_queue_well2" name="spring_topic_queue_well2"
auto-declare="true"/>

<rabbit:topic-exchange id="spring_topic_exchange"
name="spring_topic_exchange" auto-declare="true">
    <rabbit:bindings>
        <rabbit:binding pattern="lxs.*" queue="spring_topic_queue_star"/>
        <rabbit:binding pattern="lxs.#" queue="spring_topic_queue_well"/>
        <rabbit:binding pattern="mickey.#"
queue="spring_topic_queue_well2"/>
    </rabbit:bindings>
</rabbit:topic-exchange>

<!--定义rabbitTemplate对象操作可以在代码中方便发送消息-->
<rabbit:template id="rabbitTemplate" connection-
factory="connectionFactory"/>
</beans>

```

5.1.4. 发送消息

创建测试文件 `spring-rabbitmq-producer\src\test\java\com\lxs\rabbitmq\ProducerTest.java`

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring/spring-rabbitmq.xml")
public class ProducerTest {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    /**
     * 只发队列消息
     * 默认交换机类型为 direct
     * 交换机的名称为空，路由键为队列的名称
     */
    @Test
    public void queueTest(){
        //路由键与队列同名
        rabbitTemplate.convertAndSend("spring_queue", "只发队列spring_queue的消息。");
    }

    /**
     * 发送广播
     * 交换机类型为 fanout
     * 绑定到该交换机的所有队列都能够收到消息
     */
    @Test
    public void fanoutTest(){
        /**
         * 参数1: 交换机名称
         * 参数2: 路由键名（广播设置为空）
         * 参数3: 发送的消息内容
         */
        rabbitTemplate.convertAndSend("spring_fanout_exchange", "", "发送到spring_fanout_exchange交换机的广播消息");
    }

    /**
     * 通配符
     * 交换机类型为 topic
     * 匹配路由键的通配符，*表示一个单词，#表示多个单词
     * 绑定到该交换机的匹配队列能够收到对应消息
     */
    @Test
    public void topicTest(){
        /**
         * 参数1: 交换机名称
         * 参数2: 路由键名
         * 参数3: 发送的消息内容
         */
        rabbitTemplate.convertAndSend("spring_topic_exchange", "lxs.bj", "发送到spring_topic_exchange交换机lxs.bj的消息");
    }
}
```


```

        rabbitTemplate.convertAndSend("spring_topic_exchange", "lxs.bj.1", "发送到spring_topic_exchange交换机lxs.bj.1的消息");
        rabbitTemplate.convertAndSend("spring_topic_exchange", "lxs.bj.2", "发送到spring_topic_exchange交换机lxs.bj.2的消息");
        rabbitTemplate.convertAndSend("spring_topic_exchange", "mickey.cn", "发送到spring_topic_exchange交换机mickey.cn的消息");
    }
}

```

5.2. 搭建消费者工程

5.2.1. 创建工程

Parent:	 spring-rabbitmq-consumer
Name:	spring-rabbitmq-consumers
Location:	E:\work\lxs\spring-rabbitmq-consumer\spring-rabbitmq-consumers
▼ Artifact Coordinates	
GroupId:	com.lxs
The name of the artifact group, usually a company domain	
ArtifactId:	spring-rabbitmq-consumers
The name of the artifact within the group, usually a module name	
Version:	1.0-SNAPSHOT

5.2.2. 添加依赖

修改pom.xml文件内容为如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.lxs</groupId>
    <artifactId>spring-rabbitmq-consumer</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.amqp</groupId>
            <artifactId>spring-rabbit</artifactId>
            <version>2.1.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>

```



```

        <version>4.12</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.7.RELEASE</version>
    </dependency>

</dependencies>

</project>

```

5.2.3. 配置整合

1. 创建 `spring-rabbitmq-consumer\src\main\resources\properties\rabbitmq.properties` 连接参数等配置文件;

```

rabbitmq.host=192.168.192.168.220.12
rabbitmq.port=5672
rabbitmq.username=lx
rabbitmq.password=lx
rabbitmq.virtual-host=/lx

```

2. 创建 `spring-rabbitmq-consumer\src\main\resources\spring\spring-rabbitmq.xml` 整合配置文件;

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:rabbit="http://www.springframework.org/schema/rabbit"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/rabbit
        http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">
    <!--加载配置文件-->
    <context:property-placeholder
        location="classpath:properties/rabbitmq.properties"/>

    <!-- 定义rabbitmq connectionFactory -->
    <rabbit:connection-factory id="connectionFactory" host="${rabbitmq.host}"
        port="${rabbitmq.port}"
        username="${rabbitmq.username}"
        password="${rabbitmq.password}"
        virtual-host="${rabbitmq.virtual-host}"/>

    <bean id="springQueueListener"
        class="com.lxs.rabbitmq.listener.SpringQueueListener"/>

```

```

    <bean id="fanoutListener1"
class="com.lxs.rabbitmq.listener.FanoutListener1"/>
    <bean id="fanoutListener2"
class="com.lxs.rabbitmq.listener.FanoutListener2"/>
    <bean id="topicListenerStar"
class="com.lxs.rabbitmq.listener.TopicListenerStar"/>
    <bean id="topicListenerWell"
class="com.lxs.rabbitmq.listener.TopicListenerWell"/>
    <bean id="topicListenerWell2"
class="com.lxs.rabbitmq.listener.TopicListenerWell2"/>

    <rabbit:listener-container connection-factory="connectionFactory" auto-
declare="true">
        <rabbit:listener ref="springQueueListener" queue-names="spring_queue"/>
        <rabbit:listener ref="fanoutListener1" queue-
names="spring_fanout_queue_1"/>
        <rabbit:listener ref="fanoutListener2" queue-
names="spring_fanout_queue_2"/>
        <rabbit:listener ref="topicListenerStar" queue-
names="spring_topic_queue_star"/>
        <rabbit:listener ref="topicListenerWell" queue-
names="spring_topic_queue_well"/>
        <rabbit:listener ref="topicListenerWell2" queue-
names="spring_topic_queue_well2"/>
    </rabbit:listener-container>
</beans>

```

5.2.4. 消息监听器

1) 队列监听器

创建 `spring-rabbitmq-`

`consumer\src\main\java\com\lxs\rabbitmq\listener\SpringQueueListener.java`

```

public class SpringQueueListener implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("接收路由名称为: %s, 路由键为: %s, 队列名为: %s的消息: %s\n",
                                message.getMessageProperties().getReceivedExchange(),
                                message.getMessageProperties().getReceivedRoutingKey(),
                                message.getMessageProperties().getConsumerQueue(),
                                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

2) 广播监听器1

创建 `spring-rabbitmq-`

`consumer\src\main\java\com\lxs\rabbitmq\listener\FanoutListener1.java`

```
public class FanoutListener1 implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("广播监听器1: 接收路由名称为: %s, 路由键为: %s, 队列名为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3) 广播监听器2

创建 `spring-rabbitmq-`

`consumer\src\main\java\com\lxs\rabbitmq\listener\FanoutListener2.java`

```
public class FanoutListener2 implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("广播监听器2: 接收路由名称为: %s, 路由键为: %s, 队列名为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4) 星号通配符监听器

创建 `spring-rabbitmq-`

`consumer\src\main\java\com\lxs\rabbitmq\listener\TopicListenerStar.java`

```
public class TopicListenerStar implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");
```

```

        System.out.printf("通配符*监听器: 接收路由名称为: %s, 路由键为: %s, 队列为: %s的消息: %s \n",
            message.getMessageProperties().getReceivedExchange(),
            message.getMessageProperties().getReceivedRoutingKey(),
            message.getMessageProperties().getConsumerQueue(),
            msg);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

5) 井号通配符监听器

创建 spring-rabbitmq-

consumer\src\main\java\com\lxs\rabbitmq\listener\TopicListenerWell.java

```

public class TopicListenerWell implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("通配符#监听器: 接收路由名称为: %s, 路由键为: %s, 队列为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

6) 井号通配符监听器2

创建 spring-rabbitmq-

consumer\src\main\java\com\lxs\rabbitmq\listener\TopicListenerWell2.java

```

public class TopicListenerWell2 implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("通配符#监听器2: 接收路由名称为: %s, 路由键为: %s, 队列为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {

```

```
        e.printStackTrace();
    }
}
}
```

6. Spring Boot整合RabbitMQ

6.1. 简介

在Spring项目中，可以使用Spring-Rabbit去操作RabbitMQ

<https://github.com/spring-projects/spring-amqp>

尤其是在spring boot项目中只需要引入对应的amqp启动器依赖即可，方便的使用RabbitTemplate发送消息，使用注解接收消息。

一般在开发过程中:

生产者工程:

1. application.yml文件配置RabbitMQ相关信息;
2. 在生产者工程中编写配置类，用于创建交换机和队列，并进行绑定
3. 注入RabbitTemplate对象，通过RabbitTemplate对象发送消息到交换机

消费者工程:

1. application.yml文件配置RabbitMQ相关信息
2. 创建消息处理类，用于接收队列中的消息并进行处理

5.2. 搭建生产者工程

5.2.1. 创建工程

创建生产者工程springboot-rabbitmq-producer

Parent:	<None>
Name:	spring-boot-rabbitmq-producers
Location:	E:\work\1\springboot-rabbitmq-producer\spring-boot-rabbitmq-producers
▼ Artifact Coordinates	
GroupId:	org.example
The name of the artifact group, usually a company domain	
ArtifactId:	spring-boot-rabbitmq-producers
The name of the artifact within the group, usually a module name	
Version:	1.0-SNAPSHOT

5.2.2. 添加依赖

修改pom.xml文件内容为如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>
    <groupId>com.lxs</groupId>
    <artifactId>springboot-rabbitmq-producer</artifactId>
    <version>1.0-SNAPSHOT</version>

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

5.2.3. 启动类

```
package com.lxs.rabbitmq;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProducerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class);
    }
}
```

5.2.4. 配置RabbitMQ

1) 配置文件

创建application.yml，内容如下：

```
spring:
  rabbitmq:
    host: 192.168.220.12
    port: 5672
    virtual-host: /lxs
    username: lxs
    password: lxs
```

2) 绑定交换机和队列

创建RabbitMQ队列与交换机绑定的配置类com.lxs.rabbitmq.config.RabbitMQConfig

```
package com.lxs.rabbitmq.config;

import org.springframework.amqp.core.*;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {
    //交换机名称
    public static final String ITEM_TOPIC_EXCHANGE =
"springboot_item_topic_exchange";
    //队列名称
    public static final String ITEM_QUEUE = "springboot_item_queue";

    //声明交换机
    @Bean("itemTopicExchange")
    public Exchange topicExchange(){
        return
ExchangeBuilder.topicExchange(ITEM_TOPIC_EXCHANGE).durable(true).build();
    }

    //声明队列
    @Bean("itemQueue")
    public Queue itemQueue(){
        return QueueBuilder.durable(ITEM_QUEUE).build();
    }

    //绑定队列和交换机
    @Bean
    public Binding itemQueueExchange(@Qualifier("itemQueue") Queue queue,
                                     @Qualifier("itemTopicExchange") Exchange
exchange){
        return BindingBuilder.bind(queue).to(exchange).with("item.#").noargs();
    }
}
```

```
}
```

5.2.5. 测试

在生产者工程springboot-rabbitmq-producer中创建测试类，发送消息：

```
package com.lxs.rabbitmq;

import com.lxs.rabbitmq.config.RabbitMQConfig;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitMQTest {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Test
    public void test(){
        rabbitTemplate.convertAndSend(RabbitMQConfig.ITEM_TOPIC_EXCHANGE,
            "item.insert", "商品新增, routing key 为item.insert");
        rabbitTemplate.convertAndSend(RabbitMQConfig.ITEM_TOPIC_EXCHANGE,
            "item.update", "商品修改, routing key 为item.update");
        rabbitTemplate.convertAndSend(RabbitMQConfig.ITEM_TOPIC_EXCHANGE,
            "item.delete", "商品删除, routing key 为item.delete");
    }
}
```

先运行上述测试程序（交换机和队列才能先被声明和绑定），然后启动消费者；在消费者工程springboot-rabbitmq-consumer中控制台查看是否接收到对应消息。

另外；也可以在RabbitMQ的管理控制台中查看到交换机与队列的绑定：

OverviewConnectionsChannelsExchangesQueuesAdmin

Details

Type

topic

Features

durable: true

Policy

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
item_queue	item.#		Unbind

5.3. 搭建消费者工程

5.3.1. 创建工程

创建消费者工程springboot-rabbitmq-consumer

Parent: <None>

Name: springboot-rabbitmq-consumers

Location: E:\work\w1\springboot-rabbitmq-consumers

▼ Artifact Coordinates

GroupId: com.lxs

The name of the artifact group, usually a company domain

ArtifactId: springboot-rabbitmq-consumers

The name of the artifact within the group, usually a module name

Version: 1.0-SNAPSHOT

5.3.2. 添加依赖

修改pom.xml文件内容为如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
  </parent>
  <groupId>com.lxs</groupId>
  <artifactId>springboot-rabbitmq-consumer</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
```

```

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

</project>

```

5.3.3. 启动类

```

package com.lxs.rabbitmq;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class);
    }
}

```

5.3.4. 配置RabbitMQ

创建application.yml，内容如下：

```

spring:
  rabbitmq:
    host: 192.168.220.12
    port: 5672
    virtual-host: /lxs
    username: lxs
    password: lxs

```

5.3.5. 消息监听处理类

编写消息监听器com.lxs.rabbitmq.listener.MyListener

```

package com.lxs.rabbitmq.listener;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class MyListener {

```

```

/**
 * 监听某个队列的消息
 * @param message 接收到的消息
 */
@Component
public class MyListener implements RabbitListener<String> {
    @Override
    public void myListener1(String message) {
        System.out.println("消费者接收到的消息为: " + message);
    }
}

```

7.高级特性

7.1消息的可靠投递

在使用 RabbitMQ 的时候，作为消息发送方希望杜绝任何消息丢失或者投递失败场景。RabbitMQ 为我们提供了两种方式用来控制消息的投递可靠性模式。

- confirm 确认模式
- return 退回模式

rabbitmq 整个消息投递的路径为：

producer--->rabbitmq broker--->exchange--->queue--->consumer

- 消息从 producer 到 exchange 则会返回一个 confirmCallback 。
- 消息从 exchange-->queue 投递失败则会返回一个 returnCallback 。

我们将利用这两个 callback 控制消息的可靠性投递

7.1.1 确认模式

消息从 producer 到 exchange 则会返回一个 confirmCallback

```

/**
 * 确认模式：
 * 步骤：
 * 1. 确认模式开启：ConnectionFactory中开启publisher-confirms="true"
 * 2. 在rabbitTemplate定义ConfirmCallback回调函数
 */
@Test
public void testConfirm() {

    //2. 定义回调
    rabbitTemplate.setConfirmCallback(new RabbitTemplate.ConfirmCallback() {
        /**
         *
         * @param correlationData 相关配置信息
         * @param ack exchange交换机 是否成功收到了消息。true 成功，false代表失败
         * @param cause 失败原因
         */
        @Override
        public void confirm(CorrelationData correlationData, boolean ack, String cause) {

```

```

        System.out.println("confirm方法被执行了...");

        if (ack) {
            //接收成功
            System.out.println("接收成功消息" + cause);
        } else {
            //接收失败
            System.out.println("接收失败消息" + cause);
            //做一些处理，让消息再次发送。
        }
    }
}

//3. 发送消息
rabbitTemplate.convertAndSend("test_exchange_confirm", "confirm", "message
confirm...");
}

```

7.2.2 退回模式

消息从 exchange-->queue 投递失败则会返回一个 returnCallback

```

/**
 * 回退模式： 当消息发送给Exchange后，Exchange路由到Queue失败是 才会执行 ReturnCallback
 * 步骤：
 * 1. 开启回退模式:publisher-returns="true"
 * 2. 设置ReturnCallback
 * 3. 设置Exchange处理消息失败的模式： setMandatory

 * 1. 如果消息没有路由到Queue，则丢弃消息（默认）
 * 2. 如果消息没有路由到Queue，返回给消息发送方ReturnCallback
 */

@Test
public void testReturn() {

    //设置交换机处理失败消息的模式
    rabbitTemplate.setMandatory(true);

    //2.设置ReturnCallback
    rabbitTemplate.setReturnCallback(new RabbitTemplate.ReturnCallback() {
        /**
         *
         * @param message 消息对象
         * @param replyCode 错误码
         * @param replyText 错误信息
         * @param exchange 交换机
         * @param routingKey 路由键
         */
        @Override
        public void returnedMessage(Message message, int replyCode, String
replyText, String exchange, String routingKey) {
            System.out.println("return 执行了...");

            System.out.println(message);
        }
    });
}

```

```

        System.out.println(replyCode);
        System.out.println(replyText);
        System.out.println(exchange);
        System.out.println(routingKey);

        //处理
    }
});

//3. 发送消息
rabbitTemplate.convertAndSend("test_exchange_confirm", "confirm234", "message
confirm...");
}

```

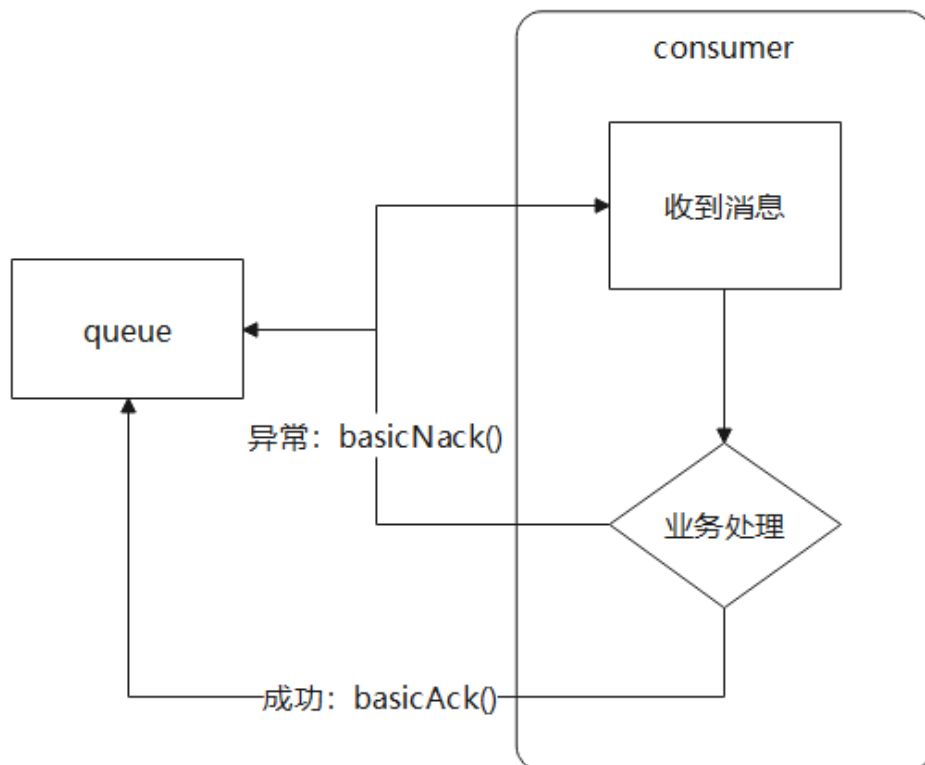
7.2 Consumer Ack

ack指Acknowledge，确认。表示消费端收到消息后的确认方式。

有两种确认方式：

- 自动确认：acknowledge="none"
- 手动确认：acknowledge="manual"

其中自动确认是指，当消息一旦被Consumer接收到，则自动确认收到，并将相应 message 从 RabbitMQ 的消息缓存中移除。但是在实际业务处理中，很可能消息接收到，业务处理出现异常，那么该消息就会丢失。如果设置了手动确认方式，则需要在业务处理成功后，调用channel.basicAck()，手动签收，如果出现异常，则调用channel.basicNack()方法，让其自动重新发送消息。



```
package com.lxs.listener;
```

```

import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageListener;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Component;

import java.io.IOException;

/**
 * Consumer ACK机制:
 * 1. 设置手动签收。acknowledge="manual"
 * 2. 让监听器类实现ChannelAwareMessageListener接口
 * 3. 如果消息成功处理，则调用channel的 basicAck() 签收
 * 4. 如果消息处理失败，则调用channel的basicNack()拒绝签收，broker重新发送给consumer
 */

@Component
public class AckListener implements ChannelAwareMessageListener {

    @Override
    public void onMessage(Message message, Channel channel) throws Exception {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();

        try {
            //1.接收转换消息
            System.out.println(new String(message.getBody()));

            //2. 处理业务逻辑
            System.out.println("处理业务逻辑...");
            int i = 3/0;//出现错误
            //3. 手动签收
            channel.basicAck(deliveryTag,true);
        } catch (Exception e) {
            //e.printStackTrace();

            //4.拒绝签收
            /*
            第三个参数: requeue: 重回队列。如果设置为true，则消息重新回到queue，broker会重新发送该消息给消费端
            */
            channel.basicNack(deliveryTag,true,true);
            //channel.basicReject(deliveryTag,true);
        }
    }
}

```

```

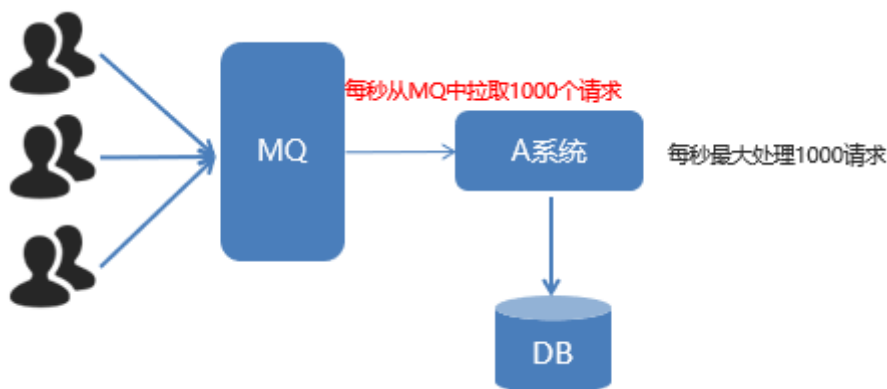
<rabbit:listener-container connection-factory="connectionFactory"
acknowledge="manual">
.....

```

7.3 消费端限流

假设一个场景，首先，我们 Rabbitmq 服务器积压了有上万条未处理的消息，我们随便打开一个消费者客户端，会出现这样情况: 巨量的消息瞬间全部推送过来，但是我们单个客户端无法同时处理这么多数数据!

当数据量特别大的时候，我们对生产端限流肯定是不科学的，因为有时候并发量就是特别大，有时候并发量又特别少，我们无法约束生产端，这是用户的行为。所以我们应该对消费端限流，用于保持消费端的稳定，当消息数量激增的时候很有可能造成资源耗尽，以及影响服务的性能，导致系统的卡顿甚至直接崩溃。



```
package com.lxs.listener;

import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Component;

/**
 * Consumer 限流机制
 * 1. 确保ack机制为手动确认。
 * 2. listener-container配置属性
 *    prefetch = 1,表示消费端每次从mq拉去一条消息来消费，直到手动确认消费完毕后，才会继续
    拉去下一条消息。
 */

@Component
public class QosListener implements ChannelAwareMessageListener {

    @Override
    public void onMessage(Message message, Channel channel) throws Exception {

        //      Thread.sleep(1000);
        //1.获取消息
        System.out.println(new String(message.getBody()));

        //2. 处理业务逻辑

        //3. 签收

        //
        channel.basicAck(message.getMessageProperties().getDeliveryTag(), true);
    }
}
```

```
}  
}
```

7.4 TTL

Time To Live, 消息过期时间设置

管控台中设置队列TTL

▼ Add a new queue

Virtual host:

Name: *

Durability:

Auto delete: (?)

Arguments: =

Add Message TTL (?) | Auto expire (?) | Max length (?) | Max length bytes (?)
Dead letter exchange (?) | Dead letter routing key (?) | Maximum priority (?)

代码实现

配置文件

```
<!--ttl-->  
<rabbit:queue name="test_queue_ttl" id="test_queue_ttl">  
  <!--设置queue的参数-->  
  <rabbit:queue-arguments>  
    <!--x-message-ttl指队列的过期时间-->  
    <entry key="x-message-ttl" value="10000" value-type="java.lang.Integer">  
</entry>  
  </rabbit:queue-arguments>  
</rabbit:queue>  
  
<rabbit:topic-exchange name="test_exchange_ttl" >  
  <rabbit:bindings>  
    <rabbit:binding pattern="ttl.#" queue="test_queue_ttl"></rabbit:binding>  
  </rabbit:bindings>  
</rabbit:topic-exchange>
```

代码

```
/**  
 * TTL: 过期时间  
 * 1. 队列统一过期  
 *  
 * 2. 消息单独过期  
 *  
 * 如果设置了消息的过期时间, 也设置了队列的过期时间, 它以时间短的为准。  
 * 队列过期后, 会将队列所有消息全部移除。  
 * 消息过期后, 只有消息在队列顶端, 才会判断其是否过期(移除掉)  
 */  
@Test
```



```

public void testTtl() {

    /* for (int i = 0; i < 10; i++) {
        // 发送消息
        rabbitTemplate.convertAndSend("test_exchange_ttl", "ttl.hehe",
"message ttl....");
    }*/

    // 消息后处理对象，设置一些消息的参数信息
    MessagePostProcessor messagePostProcessor = new MessagePostProcessor() {

        @Override
        public Message postProcessMessage(Message message) throws
AmqpException {
            //1.设置message的信息
            message.getMessageProperties().setExpiration("5000");//消息的过期时
间

            //2.返回该消息
            return message;
        }
    };

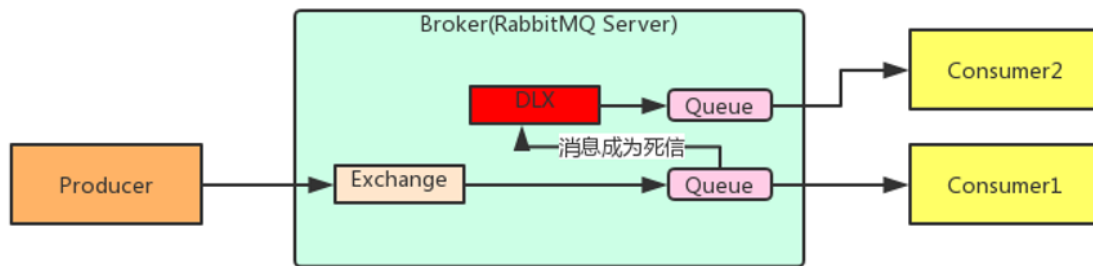
    //消息单独过期
    rabbitTemplate.convertAndSend("test_exchange_ttl", "ttl.hehe", "message
ttl....", messagePostProcessor);

    //
    for (int i = 0; i < 10; i++) {
        //
        if(i == 5){
            //消息单独过期
            rabbitTemplate.convertAndSend("test_exchange_ttl", "ttl.hehe",
"message ttl....",messagePostProcessor);
        }else{
            //
            //不过期的消息
            rabbitTemplate.convertAndSend("test_exchange_ttl", "ttl.hehe",
"message ttl....");
        }
    }
}

```

7.5 死信队列

死信队列，英文缩写：DLX。Dead Letter Exchange（死信交换机），当消息成为Dead message后，可以被重新发送到另一个交换机，这个交换机就是DLX。

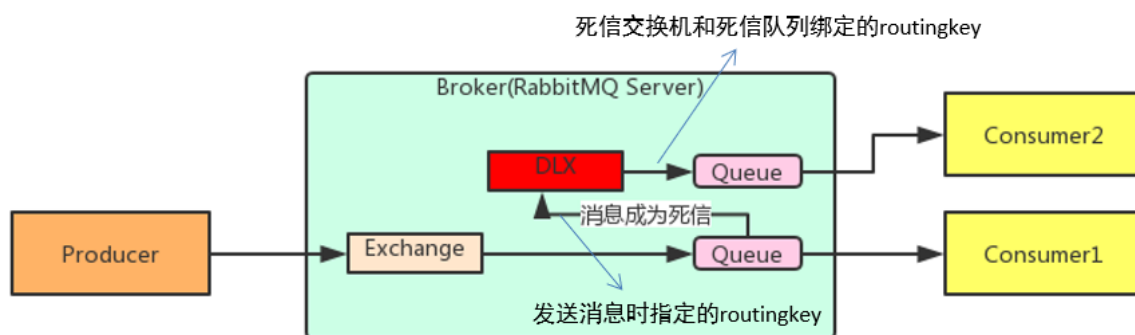


消息成为死信的三种情况：

1. 队列消息长度到达限制；
2. 消费者拒接消费消息，basicAck/basicReject,并且不把消息重新放入原目标队列, requeue=false；
3. 原队列存在消息过期设置，消息到达超时时间未被消费；

队列绑定死信交换机：

给队列设置参数：x-dead-letter-exchange 和 x-dead-letter-routing-key



代码实现

配置

```

<!--
    1. 声明正常的队列(test_queue_dlx)和交换机(test_exchange_dlx)
-->
<rabbit:queue name="test_queue_dlx" id="test_queue_dlx">
    <!--3. 正常队列绑定死信交换机-->
    <rabbit:queue-arguments>
        <!--3.1 x-dead-letter-exchange: 死信交换机名称-->
        <entry key="x-dead-letter-exchange" value="exchange_dlx" />

        <!--3.2 x-dead-letter-routing-key: 发送给死信交换机的routingkey-->
        <entry key="x-dead-letter-routing-key" value="dlx.hehe" />

        <!--4.1 设置队列的过期时间 ttl-->
        <entry key="x-message-ttl" value="10000" value-type="java.lang.Integer"
    />

        <!--4.2 设置队列的长度限制 max-length -->
        <entry key="x-max-length" value="10" value-type="java.lang.Integer" />
    </rabbit:queue-arguments>
</rabbit:queue>
<rabbit:topic-exchange name="test_exchange_dlx">

```

```

    <rabbit:bindings>
        <rabbit:binding pattern="test.dlx.#" queue="test_queue_dlx">
    </rabbit:binding>
    </rabbit:bindings>
</rabbit:topic-exchange>

<!--
    2. 声明死信队列(queue_dlx)和死信交换机(exchange_dlx)
-->
<rabbit:queue name="queue_dlx" id="queue_dlx"></rabbit:queue>

<rabbit:topic-exchange name="exchange_dlx">
    <rabbit:bindings>
        <rabbit:binding pattern="dlx.#" queue="queue_dlx"></rabbit:binding>
    </rabbit:bindings>
</rabbit:topic-exchange>

```

测试代码

生产端测试

```

/**
 * 发送测试死信消息：
 * 1. 过期时间
 * 2. 长度限制
 * 3. 消息拒收
 */
@Test
public void testDlx(){
    //1. 测试过期时间，死信消息
    //rabbitTemplate.convertAndSend("test_exchange_dlx","test.dlx.haha","我是一条消息，我会死吗? ");

    //2. 测试长度限制后，消息死信
    /* for (int i = 0; i < 20; i++) {
        rabbitTemplate.convertAndSend("test_exchange_dlx","test.dlx.haha","我是一条消息，我会死吗? ");
    }*/

    //3. 测试消息拒收
    rabbitTemplate.convertAndSend("test_exchange_dlx","test.dlx.haha","我是一条消息，我会死吗? ");
}

```

消费端监听

```

package com.lxs.listener;

import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Component;

```

```

@Component
public class DlxListener implements ChannelAwareMessageListener {

    @Override
    public void onMessage(Message message, Channel channel) throws Exception {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();

        try {
            //1.接收转换消息
            System.out.println(new String(message.getBody()));

            //2. 处理业务逻辑
            System.out.println("处理业务逻辑...");
            int i = 3/0;//出现错误
            //3. 手动签收
            channel.basicAck(deliveryTag,true);
        } catch (Exception e) {
            //e.printStackTrace();
            System.out.println("出现异常，拒绝接受");
            //4.拒绝签收，不重回队列 requeue=false
            channel.basicNack(deliveryTag,true,false);
        }
    }
}

```

```

<rabbit:listener ref="dlxListener" queue-names="test_queue_dlx">
</rabbit:listener>

```

7.6 延迟队列

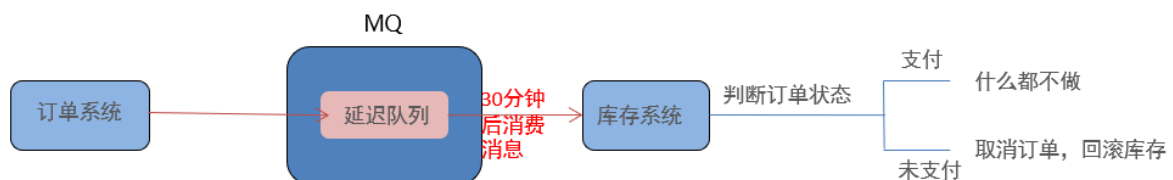
延迟队列，即消息进入队列后不会立即被消费，只有到达指定时间后，才会被消费。

需求：

1. 下单后，30分钟未支付，取消订单，回滚库存。
2. 新用户注册成功7天后，发送短信问候。

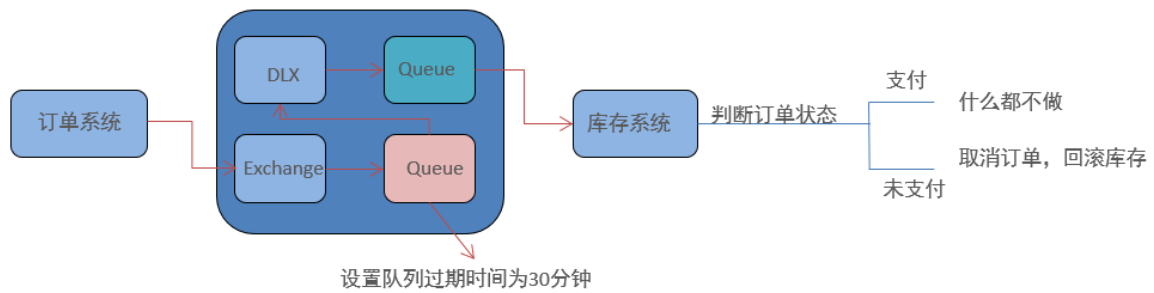
实现方式：

1. 定时器
2. 延迟队列



很可惜，在RabbitMQ中并未提供延迟队列功能。

但是可以使用：TTL+死信队列 组合实现延迟队列的效果。



代码实现

配置

```

<!--
    延迟队列：
    1. 定义正常交换机（order_exchange）和队列（order_queue）
    2. 定义死信交换机（order_exchange_dlx）和队列（order_queue_dlx）
    3. 绑定，设置正常队列过期时间为30分钟
-->
-->
<!-- 1. 定义正常交换机（order_exchange）和队列（order_queue）-->
<rabbit:queue id="order_queue" name="order_queue">
    <!-- 3. 绑定，设置正常队列过期时间为30分钟-->
    <rabbit:queue-arguments>
        <entry key="x-dead-letter-exchange" value="order_exchange_dlx" />
        <entry key="x-dead-letter-routing-key" value="dlx.order.cancel" />
        <entry key="x-message-ttl" value="10000" value-type="java.lang.Integer"
    />
    />
    </rabbit:queue-arguments>
</rabbit:queue>
<rabbit:topic-exchange name="order_exchange">
    <rabbit:bindings>
        <rabbit:binding pattern="order.#" queue="order_queue"></rabbit:binding>
    </rabbit:bindings>
</rabbit:topic-exchange>

<!-- 2. 定义死信交换机（order_exchange_dlx）和队列（order_queue_dlx）-->
<rabbit:queue id="order_queue_dlx" name="order_queue_dlx"></rabbit:queue>

<rabbit:topic-exchange name="order_exchange_dlx">
    <rabbit:bindings>
        <rabbit:binding pattern="dlx.order.#" queue="order_queue_dlx">
    </rabbit:binding>
    </rabbit:bindings>
</rabbit:topic-exchange>

```

生产端测试

```

@Test
public void testDelay() throws InterruptedException {
    //1.发送订单消息。 将来是在订单系统中，下单成功后，发送消息
    rabbitTemplate.convertAndSend("order_exchange","order.msg","订单信息:
id=1,time=2019年8月17日16:41:47");

    /*//2.打印倒计时10秒
    for (int i = 10; i > 0 ; i--) {
        System.out.println(i+"...");
        Thread.sleep(1000);
    }*/
}

```

消费端监听

```

package com.lxs.listener;

import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Component;

@Component
public class OrderListener implements ChannelAwareMessageListener {

    @Override
    public void onMessage(Message message, Channel channel) throws Exception {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();

        try {
            //1.接收转换消息
            System.out.println(new String(message.getBody()));

            //2. 处理业务逻辑
            System.out.println("处理业务逻辑...");
            System.out.println("根据订单id查询其状态...");
            System.out.println("判断状态是否为支付成功");
            System.out.println("取消订单，回滚库存...");
            //3. 手动签收
            channel.basicAck(deliveryTag,true);
        } catch (Exception e) {
            //e.printStackTrace();
            System.out.println("出现异常，拒绝接受");
            //4.拒绝签收，不重回队列 requeue=false
            channel.basicNack(deliveryTag,true,false);
        }
    }
}

```

```

<rabbit:listener ref="orderListener" queue-names="order_queue_dlx">
</rabbit:listener>

```

