

# 课程目标

- 1、了解 MQ 的本质和 RabbitMQ 的特性；
- 2、掌握 RabbitMQ 的 Java API 编程和 Spring 集成 RabbitMQ

# 内容定位

适合没有使用过 RabbitMQ，或者不理解 RabbitMQ 工作原理及高级特性的同学。

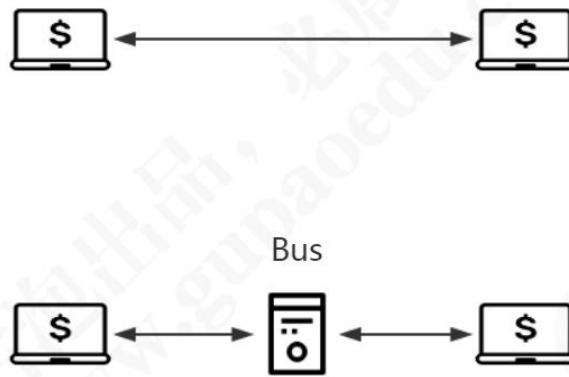
## 1. MQ 入门

### 1.1. 消息队列简介

#### 1.1.1. MQ 的诞生历程

我们要去用 MQ，先来了解一下 MQ 是怎么诞生的，这样对于它解决了什么问题理解会更加深刻。大家知不知道世界上第一个 MQ 叫什么名字，是什么时候诞生的？

1983 年的时候，有个在 MIT 工作的印度小伙突发奇想，以前我们的软件相互通信，都是点对点的，而且要实现相同的协议，能不能有一种专门用来通信的中间件，就像主板（BUS）一样，把不同的软件集成起来呢？于是他搞了一家公司（Teknekron），开发了世界上第一个消息队列软件 The Information Bus(TIB)。最开始的时候，它被高盛这些公司用在金融交易里面。因为 TIB 实现了发布订阅(Publish/Subscribe)模型，信息的生产者和消费者可以完全解耦，这个特性引起了电信行业特别是新闻机构的注意。1994 年路透社收购了 Teknekron。



TIB 的成功马上引起了业界大佬 IBM 的注意，他们研发了自己的 IBM MQ (IBM Wesphe)。后面微软也加入了这场战斗，研发了 MSMQ。这个时候，每个厂商的产品是孤立的，大家都有自己的技术壁垒。比如一个应用订阅了 IBM MQ 的消息，如果有要订阅 MSMQ 的消息，因为协议、API 不同，又要重复去实现。为什么大家都不愿意去创建标准接口，来实现不同的 MQ 产品的互通呢？跟现在微信里面不能打开淘宝页面是一个道理（商业竞争）。

JDBC 协议大家非常熟悉吧？J2EE 制定了 JDBC 的规范，那么那么各个数据库厂商自己去实现协议，提供 jar 包，在 Java 里面就可以使用相同的 API 做操作不同的数据库了。MQ 产品的问题也是一样的，2001 年的时候，SUN 公司发布了 JMS 规范，它想要在各厂商的 MQ 上面统一包装一层 Java 的规范，大家都只需要针对 API 编程就可以了，不需要关注使用了什么样的消息中间件，只要选择合适的 MQ 驱动。但是 JMS 只适用于 Java 语言，它是跟语言绑定的，没有从根本上解决这个问题（只是一个 API）。

所以在 06 年的时候，AMQP 规范发布了。它是跨语言和跨平台的，真正地促进了消息队列的繁荣发展。

07 年的时候，Rabbit 技术公司基于 AMQP 开发了 RabbitMQ 1.0。为什么要用

Erlang 语言呢？因为 Erlang 是作者 Matthias 擅长的开发语言。第二个就是 Erlang 是为电话交换机编写的语言，天生适合分布式和高并发。



为什么要取 Rabbit Technologies 这个名字呢？因为兔子跑得很快，而且繁殖起来很疯狂。

从最开始用在金融行业里面，现在 RabbitMQ 已经在世界各地的公司中遍地开花。国内的绝大部分大厂都在用 RabbitMQ，包括头条，美团，滴滴（TMD），去哪儿，艺龙，淘宝也有用。

### 1.1.2. 什么是 MQ（Message Queue）？

MQ 的本质是什么呢？

消息队列，又叫做消息中间件。是指用高效可靠的消息传递机制进行与平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息队列模型，可以在分布式环境下扩展进程的通信（维基百科）。

基于以上的描述（MQ 是用来解决通信的问题），我们知道，MQ 的几个主要特点：

- 1、 是一个独立运行的服务。生产者发送消息，消费者接收消费，需要先跟服务器建立连接。
- 2、 采用队列作为数据结构，有先进先出的特点。

### 3、 具有发布订阅的模型，消费者可以获取自己需要的消息。



我们可以把 RabbitMQ 类比成邮局和邮差，它是用来帮我们存储和转发消息的。

问题：如果仅仅是解决消息消费的问题，Java 里面有这么多的队列的实现，为什么不用他们呢？这个问题的答案，就跟有了 HashMap 之后，为什么还要 Redis 做缓存是一样的。

```
AbstractQueue (java.util)
ArrayBlockingQueue (java.util.concurrent)
ArrayDeque (java.util)
AsLIFOQueue in Collections (java.util)
BlockingDeque (java.util.concurrent)
BlockingQueue (java.util.concurrent)
CheckedQueue in Collections (java.util)
CompilationSequence in Compiler (jdk.nashorn)
ConcurrentLinkedDeque (java.util.concurrent)
ConcurrentLinkedQueue (java.util.concurrent)
```

Queue 不能跨进程，不能在分布式系统中使用，并且没有持久化机制等等。

#### 1.1.3. 为什么要使用 MQ?

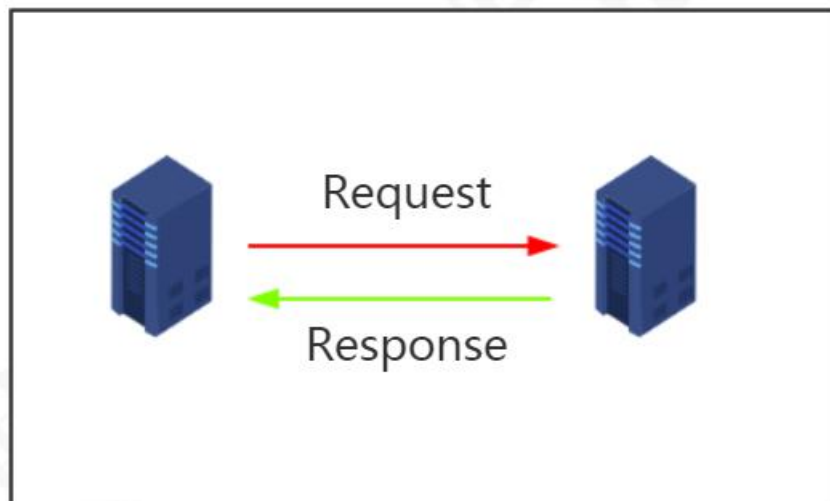
我们已经知道 MQ 是什么了，那在什么地方可以用 MQ，或者说，为什么要使用 MQ 呢？这是一个很常见的面试题，如果你在项目里面用了 MQ，还不知道这个问题的答案，说明你自己从来没有思考总结过，因为这个项目是别人架构设计的，你可能只是做了些维护的工作。有一天让你自己去做项目架构的时候，你搞一个 MQ 进去，理由就是以前的项目也是这么干的，这是很危险的。

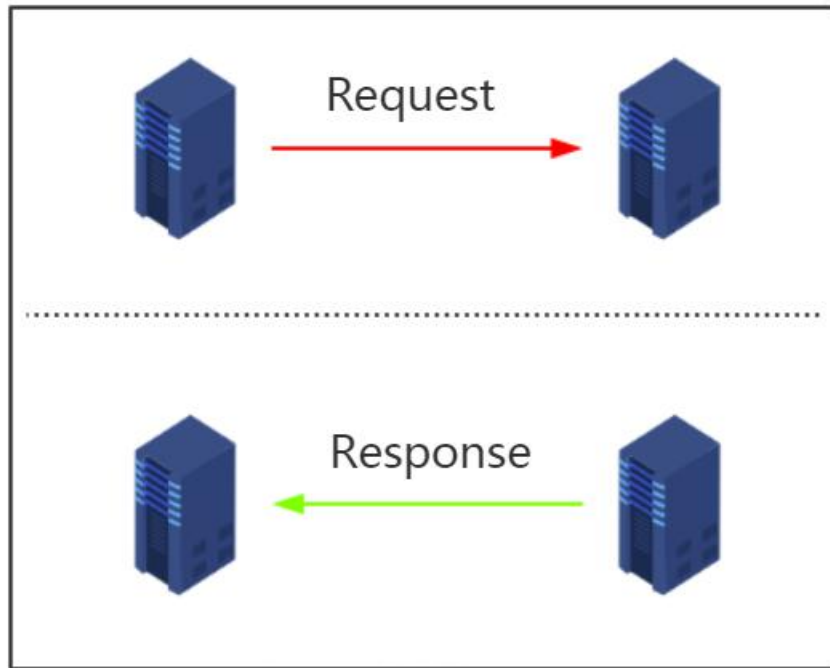
#### 1.1.1.1. 实现异步通信

同步的通信是什么样的？

发出一个调用请求之后，在没有得到结果之前，就不返回。由调用者主动等待这个调用的结果。

而异步是相反的，调用在发出之后，这个调用就直接返回了，所以没有返回结果。也就是说，当一个异步过程调用发出后，调用者不会马上得到结果。而是在调用发出后，被调用者通过状态、通知来通知调用者，或通过回调函数处理这个调用。





举个例子：

大家都用过手机银行的跨行转账功能。大家用 APP 的转账功能的时候，有一个实时模式，有一个非实时模式。

实时转账实际上是异步通信，因为这个里面涉及的机构比较多，调用链路比较长，本行做了一些列的处理之后，转发给银联或者人民银行的支付系统，再转发给接收行，接收行处理以后再原路返回。

所以转账以后会有一行小字提示：具体到账时间以对方行处理为准，也就是说转出行只保证了这个转账的消息发出。那为什么到账时间又这么快呢？很多时候我们转账之后，不用几秒钟对方就收到了。是因为大部分的 MQ 都有一个低延迟的特性，能在短时间内处理非常多的消息。

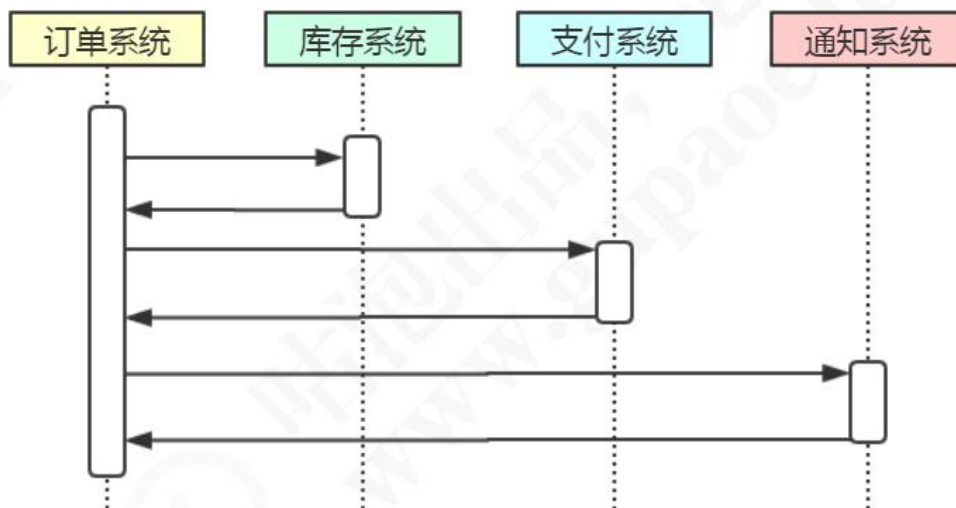
很多理财软件提现也是一样，先提交请求，到账时间不定。这个是用 MQ 实现系统间异步通信的一个场景。

#### 1.1.1.1. 实现系统解耦

第二个主要的功能，是用来实现系统解耦。既然说到解耦，那我们要先来了解一下耦合的概念。

耦合是系统内部或者系统之间存在相互作用，相互影响和相互依赖。

在我们的分布式系统中，一个业务流程涉及多个系统的时候，他们之间就会形成一个依赖关系。



比如我们以 12306 网站退票为例，在传统的通信方式中，订单系统发生了退货的动作，那么要依次调用所有下游系统的 API，比如调用库存系统的 API 恢复库存，因为这张火车票还要释放出去给其他乘客购买；调用支付系统的 API，不论是支付宝微信还是银行卡，要把手续费扣掉以后，原路退回给消费者；调用通知系统 API 通知用户退货成功。

// 伪代码

```
public void returnGoods(){
    stockService.updateInventory ();
    payService.refund();
    noticeService.notice();
}
```



```
}
```

这个过程是串行执行的，如果在恢复库存的时候发生了异常，那么后面的代码都不会执行。由于这一系列的动作，恢复库存，资金退还，发送通知，本质上没有一个严格的先后顺序，也没有直接的依赖关系，也就是说，只要用户提交了退货的请求，后面的这些动作都是要完成的。库存有没有恢复成功，不影响资金的退还和发送通知。

如果把串行改成并行，我们有什么思路？

(多线程)

多线程或者线程池是可以实现的，但是每一个需要并行执行的地方都引入线程，又会带来线程或者线程池的管理问题。

所以，这种情况下，我们可以引入 MQ 实现系统之间依赖关系的解耦合。

引入 MQ 以后：



订单系统只需要把退货的消息发送到消息队列上，由各个下游的业务系统自己创建队列，然后监听队列消费消息。

在这种情况下订单系统里面就不需要配置其他系统的 IP、端口、接口地址了，因为它不需要关心消费者在网络上的什么位置，所以下游系统改 IP 没有任何影响。甚至不需要关心消费者有没有消费成功，它只需要把消费发到消息队列的服务器上就可以了。



这样，我们就实现了系统之间依赖关系的解耦。

#### 1.1.1.2. 实现流量削峰

第三个主要功能，是实现流量削峰。

在很多的电商系统里面，有一个瞬间流量达到峰值的情况，比如京东的 618，淘宝的双 11，还有小米抢购。普通的硬件服务器肯定支撑不了这种百万或者千万级别的并发量，就像 2012 年的小米一样，动不动服务器就崩溃。

如果通过堆硬件的方式去解决，那么在流量峰值过去以后就会出现巨大的资源浪费。那要怎么办呢？如果说要保护我们的应用服务器和数据库，限流也是可以的，但是这样又会导致订单的丢失，没有达到我们的目的。

为了解决这个问题，我们就可以引入 MQ，MQ 既然是队列，一定有队列的特性，我们知道队列的特性是什么？

(先进先出 FIFO)

这样，我们就可以先把所有的流量承接下来，转换成 MQ 消息发送到消息队列服务器上，业务层就可以根据自己的消费速率去处理这些消息，处理之后再返回结果。就像我们在火车站排队一样，大家只能一个一个买票，不会因为人多就导致售票员忙不过来。如果要处理快一点，大不了多开几个窗口（增加几个消费者）。

这个是我们利用 MQ 实现流量削峰的一个案例。

如果大家的公司里面有用到 MQ 的话，也可以对号入座看看是起到了什么作用。

总结起来：

- 1) 对于数据量大或者处理耗时长的操作，我们可以引入 MQ 实现异步通信，减少

客户端的等待，提升响应速度。

- 2) 对于改动影响大的系统之间，可以引入 MQ 实现解耦，减少系统之间的直接依赖。
- 3) 对于会出现瞬间的流量峰值的系统，我们可以引入 MQ 实现流量削峰，达到保护应用和数据库的目的。

所以对于一些特定的业务场景，MQ 对于优化我们的系统还是有很大的帮助的，那么大家想一下，把传统的 RPC 通信改成 MQ 通信会不会带来一些问题呢？

#### 1.1.4. 使用消息队列带来的一些问题

系统可用性降低：原来是两个节点的通信，现在还需要独立运行一个服务，如果 MQ 服务器或者通信网络出现问题，就会导致请求失败。

系统复杂性提高：为什么说复杂？第一个就是你必须要理解相关的模型和概念，才能正确地配置和使用 MQ。第二个，使用 MQ 发送消息必须要考虑消息丢失和消息重复消费的问题。一旦消息没有被正确地消费，就会带来数据一致性的问题。

所以，我们在做系统架构的时候一定要根据实际情况来分析，不要因为我们说了这么多的 MQ 能解决的问题，就盲目地引入 MQ。

## 1.2. RabbitMQ 简介

### 1.2.1. 基本特性

官网 <https://www.rabbitmq.com/getstarted.html>

高可靠：RabbitMQ 提供了多种多样的特性让你在可靠性和性能之间做出权衡，包括持久化、发送应答、发布确认以及高可用性。

灵活的路由：通过交换机（Exchange）实现消息的灵活路由。

支持多客户端：对主流开发语言（Python、Java、Ruby、PHP、C#、JavaScript、Go、Elixir、Objective-C、Swift 等）都有客户端实现。

集群与扩展性：多个节点组成一个逻辑的服务器，支持负载。

高可用队列：通过镜像队列实现队列中数据的复制。

权限管理：通过用户与虚拟机实现权限管理。

插件系统：支持各种丰富的插件扩展，同时也支持自定义插件。

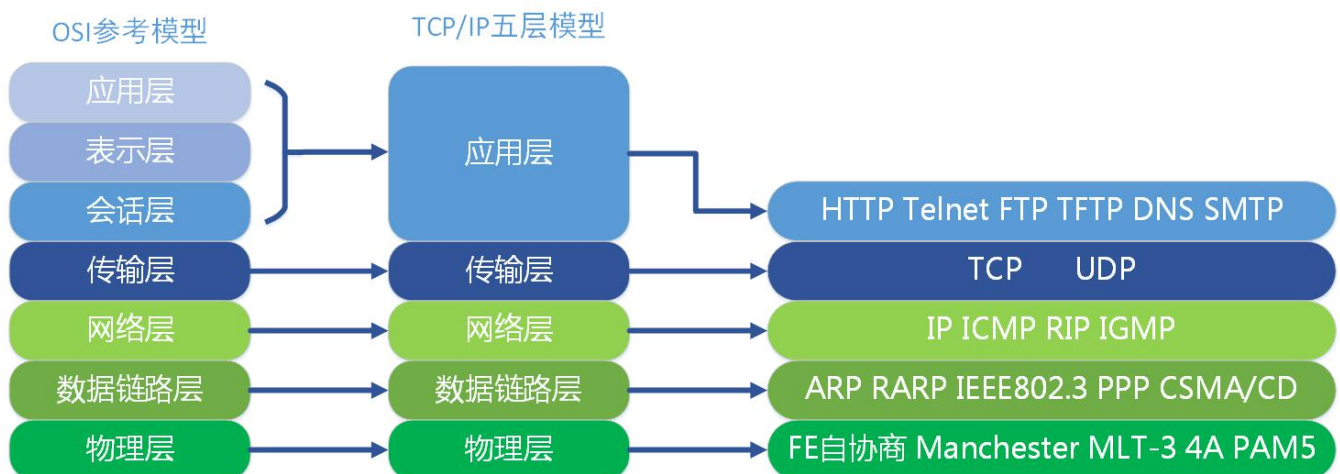
与 Spring 集成：Spring 对 AMQP 进行了封装。

## 1.2.2. AMQP 协议

### 1.2.2.1. 总体介绍

<http://www.amqp.org/sites/amqp.org/files/amqp.pdf>

AMQP：高级消息队列协议，是一个工作于应用层的协议，最新的版本是 1.0 版本。



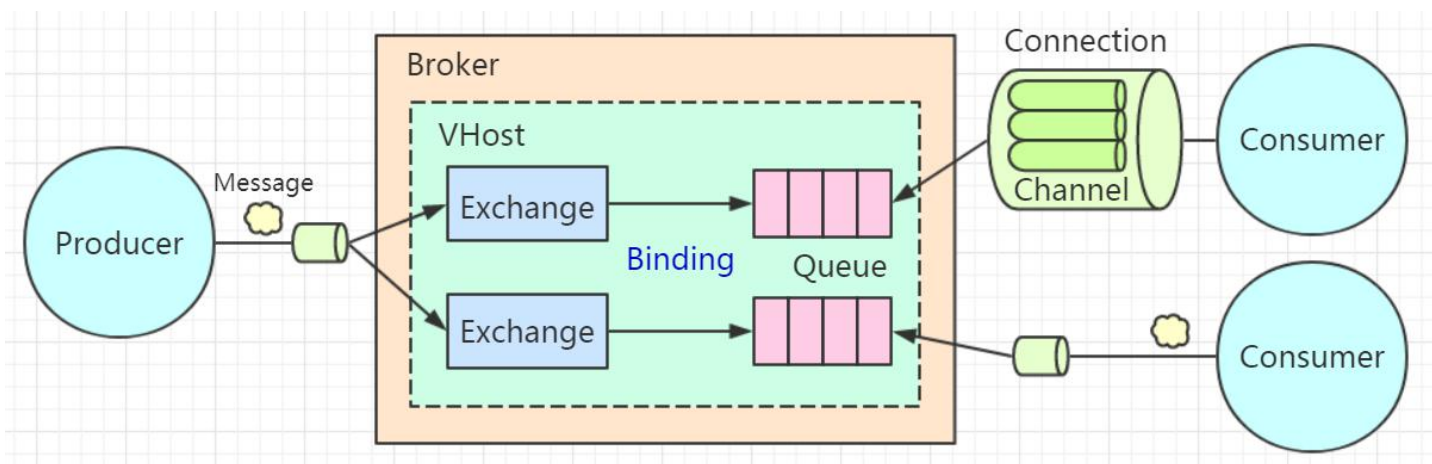
除了 RabbitMQ 之外，AMQP 的实现还有 OpenAMQ、Apache Qpid、Redhat Enterprise MRG、AMQP Infrastructure、ØMQ、Zyre。

除了 AMQP 之外，RabbitMQ 支持多种协议，STOMP、MQTT、HTTP and WebSockets。

可以使用 WireShark 等工具对 RabbitMQ 通信的 AMQP 协议进行抓包。

#### 1.2.2.2.工作模型

由于 RabbitMQ 实现了 AMQP 协议，所以 RabbitMQ 的工作模型也是基于 AMQP 的。理解这张图片至关重要。



### 1、Broker

我们要使用 RabbitMQ 来收发消息，必须要安装一个 RabbitMQ 的服务，可以安装在 Windows 上面也可以安装在 Linux 上面，默认是 5672 的端口。这台 RabbitMQ 的服务器我们把它叫做 Broker，中文翻译是代理/中介，因为 MQ 服务器帮助我们做的事情就是存储、转发消息。

### 2、Connection

无论是生产者发送消息，还是消费者接收消息，都必须要跟 Broker 之间建立一个连

接，这个连接是一个 TCP 的长连接。

### 3、Channel

如果所有的生产者发送消息和消费者接收消息，都直接创建和释放 TCP 长连接的话，对于 Broker 来说肯定会造成很大的性能损耗，因为 TCP 连接是非常宝贵的资源，创建和释放也要消耗时间。

所以在 AMQP 里面引入了 Channel 的概念，它是一个虚拟的连接。我们把它翻译成通道，或者消息信道。这样我们就可以在保持的 TCP 长连接里面去创建和释放 Channel，大大地减少了资源消耗。另外一个需要注意的是，Channel 是 RabbitMQ 原生 API 里面的最重要的编程接口，也就是说我们定义交换机、队列、绑定关系，发送消息消费消息，调用的都是 Channel 接口上的方法。

<https://stackoverflow.com/questions/18418936/rabbitmq-and-relationship-between-channel-and-connection>

### 4、Queue

现在我们已经连到 Broker 了，可以收发消息了。在其他一些 MQ 里面，比如 ActiveMQ 和 Kafka，我们的消息都是发送到队列上的。

队列是真正用来存储消息的，是一个独立运行的进程，有自己的数据库 (Mnesia)。

消费者获取消息有两种模式，一种是 Push 模式，只要生产者发到服务器，就马上推送给消费者。另一种是 Pull 模式，消息存放在服务端，只有消费者主动获取才能拿到消息。消费者需要写一个 while 循环不断地从队列获取消息吗？不需要，我们可以基于事件机制，实现消费者对队列的监听。

由于队列有 FIFO 的特性，只有确定前一条消息被消费者接收之后，才会把这条消息从数据库删除，继续投递下一条消息。

## 5、Exchange

在 RabbitMQ 里面永远不会出现消息直接发送到队列的情况。因为在 AMQP 里面引入了交换机（Exchange）的概念，用来实现消息的灵活路由。

交换机是一个绑定列表，用来查找匹配的绑定关系。

队列使用绑定键（Binding Key）跟交换机建立绑定关系。

生产者发送的消息需要携带路由键（Routing Key），交换机收到消息时会根据它保存的绑定列表，决定将消息路由到哪些与它绑定的队列上。

注意：交换机与队列、队列与消费者都是多对多的关系。

## 5、Vhost

我们每个需要实现基于 RabbitMQ 的异步通信的系统，都需要在服务器上创建自己要用的交换机、队列和它们的绑定关系。如果某个业务系统不想跟别人混用一个系统，怎么办？再采购一台硬件服务器单独安装一个 RabbitMQ 服务？这种方式成本太高了。在同一个硬件服务器上安装多个 RabbitMQ 的服务呢？比如再运行一个 5673 的端口？没有必要，因为 RabbitMQ 提供了虚拟主机 VHOST。

VHOST 除了可以提高硬件资源的利用率之外，还可以实现资源的隔离和权限的控制。它的作用类似于编程语言中的 namespace 和 package，不同的 VHOST 中可以有同名的 Exchange 和 Queue，它们是完全透明的。

这个时候，我们可以为不同的业务系统创建不同的用户（User），然后给这些用户分配 VHOST 的权限。比如给风控系统的用户分配风控系统的 VHOST 的权限，这个用户可以访问里面的交换机和队列。给超级管理员分配所有 VHOST 的权限。

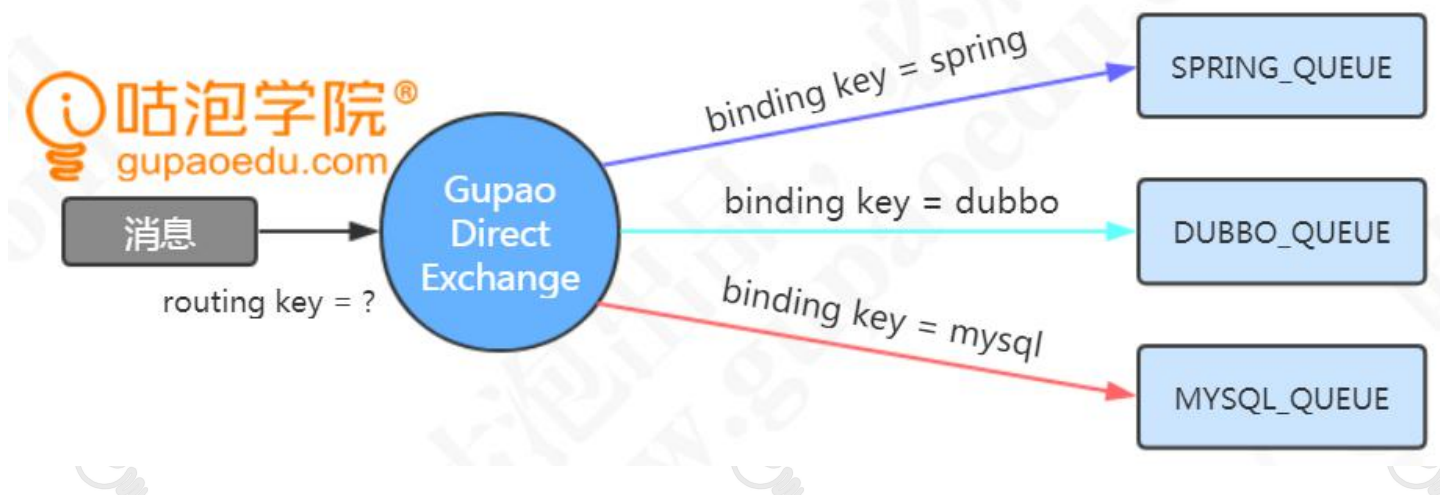
我们说到 RabbitMQ 引入 Exchange 是为了实现消息的灵活路由，到底有哪些路由方式？

### 1.2.2.3.路由方式

#### 直连 Direct

队列与直连类型的交换机绑定，需指定一个精确的绑定键。

生产者发送消息时会携带一个路由键。只有当路由键与其中的某个绑定键完全匹配时，这条消息才会从交换机路由到满足路由关系的此队列上。



例如：`channel.basicPublish("MY_DIRECT_EXCHANGE", "spring", msg 1)`；只有第一个队列能收到消息。



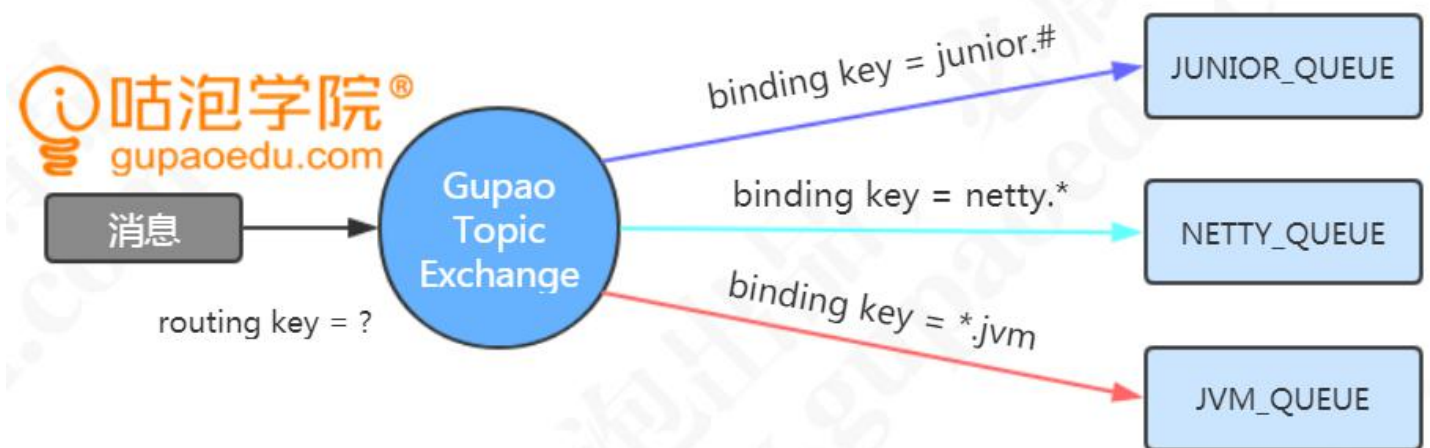
## 主题 Topic

队列与主题类型的交换机绑定时，可以在绑定键中使用通配符。两个通配符：

# 0 个或者多个单词

\* 不多不少一个单词

单词 (word) 指的是用英文的点 "." 隔开的字符。例如 abc.def 是两个单词。



解读：第一个队列支持路由键以 junior 开头的消息路由，后面可以有单词，也可以没有。

第二个队列支持路由键以 netty 开头，并且后面是一个单词的消息路由。

第三个队列支持路由键以 jvm 结尾，并且前面是一个单词的消息路由。

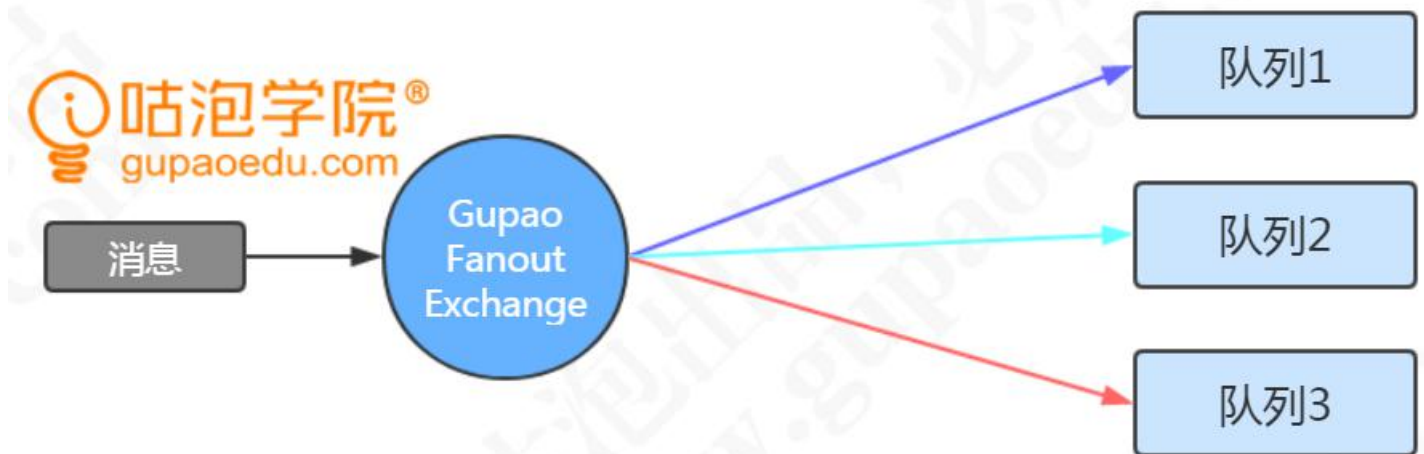
例如：

`channel.basicPublish("MY_TOPIC_EXCHANGE", "junior.fjd.klj", "msg 2");` 只有第一个队列能收到消息。

`channel.basicPublish("MY_TOPIC_EXCHANGE", "junior.jvm", "msg 3");` 第一个队列和第三个队列能收到消息。

## 广播 Fanout

主题类型的交换机与队列绑定时，不需要指定绑定键。因此生产者发送消息到广播类型的交换机上，也不需要携带路由键。消息达到交换机时，所有与之绑定了的队列，都会收到相同的消息的副本。



例如：

`channel.basicPublish("MY_FANOUT_EXCHANGE", "", "msg 4");` 三个队列都会收到 msg 4。

## 1.3. 基本使用

### 1.3.1. 安装

由于 RabbitMQ 是用 Erlang 语言编写的，必须要先安装 Erlang。

Windows 安装步骤见预习资料。

安装成功以后，会提供默认的 VHost、Exchange。

### 1.3.2. Java API 编程

### 1.3.2.1.引入依赖

#### 创建 Maven 工程，pom.xml 引入依赖

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

### 1.3.2.2.生产者

```
package com.gupaoedu.simple;

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

public class MyProducer {
    private final static String EXCHANGE_NAME = "SIMPLE_EXCHANGE";

    public static void main(String[] args) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        // 连接 IP
        factory.setHost("127.0.0.1");
        // 连接端口
        factory.setPort(5672);
        // 虚拟机
        factory.setVirtualHost("/");
        // 用户
        factory.setUsername("guest");
        factory.setPassword("guest");

        // 建立连接
        Connection conn = factory.newConnection();
        // 创建消息通道
        Channel channel = conn.createChannel();
```

```

// 发送消息
String msg = "Hello world, Rabbit MQ";

// String exchange, String routingKey, BasicProperties props, byte[] body
channel.basicPublish(EXCHANGE_NAME, "gupao.best", null, msg.getBytes());

channel.close();
conn.close();
}
}

```

### 1.3.2.3.消费者

```

package com.gupaoedu.simple;

import com.rabbitmq.client.*;
import java.io.IOException;

public class MyConsumer {
    private final static String EXCHANGE_NAME = "SIMPLE_EXCHANGE";
    private final static String QUEUE_NAME = "SIMPLE_QUEUE";

    public static void main(String[] args) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        // 连接 IP
        factory.setHost("127.0.0.1");
        // 默认监听端口
        factory.setPort(5672);
        // 虚拟机
        factory.setVirtualHost("/");

        // 设置访问的用户
        factory.setUsername("guest");
        factory.setPassword("guest");
        // 建立连接
        Connection conn = factory.newConnection();
        // 创建消息通道
        Channel channel = conn.createChannel();

        // 声明交换机
        // String exchange, String type, boolean durable, boolean autoDelete, Map<String, Object> arguments
    }
}

```

```

channel.exchangeDeclare(EXCHANGE_NAME,"direct",false, false, null);

// 声明队列
// String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
System.out.println(" Waiting for message....");

// 绑定队列和交换机
channel.queueBind(QUEUE_NAME,EXCHANGE_NAME,"gupao.best");

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

// 开始获取消息
// String queue, boolean autoAck, Consumer callback
channel.basicConsume(QUEUE_NAME, true, consumer);
}
}

```

当你明白了 RabbitMQ 的 Java 原生 API 编程，可以用它来干什么？

(不要离开 Spring 就不知道怎么实现功能了)

#### 1.3.2.4. 参数详解

##### 1) 声明交换机的参数

String type: 交换机的类型，direct, topic, fanout 中的一种。

boolean durable: 是否持久化，代表交换机在服务器重启后是否还存在。

##### 2) 声明队列的参数

boolean durable: 是否持久化，代表队列在服务器重启后是否还存在。

boolean exclusive: 是否排他性队列。排他性队列只能在声明它的 Connection 中使用（可以在同一个 Connection 的不同的 channel 中使用），连接断开时自动删除。

boolean autoDelete: 是否自动删除。如果为 true，至少有一个消费者连接到这个队列，之后所有与这个队列连接的消费者都断开时，队列会自动删除。

Map<String, Object> arguments: 队列的其他属性，例如：

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

Add queue

属性	含义
x-message-ttl	队列中消息的存活时间，单位毫秒
x-expires	队列在多久没有消费者访问以后会被删除
x-max-length	队列的最大消息数
x-max-length-bytes	队列的最大容量，单位 Byte
x-dead-letter-exchange	队列的死信交换机
x-dead-letter-routing-key	死信交换机的路由键
x-max-priority	队列中消息的最大优先级，消息的优先级不能超过它

### 3) 消息属性 BasicProperties

以下列举了一些主要的参数：

```

m ➤ setDeliveryMode(MessageDeliveryMode deliver
m ➤ setHeader(String key, Object value)
m ➤ setExpiration(String expiration)
m ➤ setContentType(String contentType)
m ➤ setAppId(String appId)
m ➤ setClusterId(String clusterId)
m ➤ setConsumerQueue(String consumerQueue)
m ➤ setConsumerTag(String consumerTag)
m ➤ setContentEncoding(String contentEncoding)
m ➤ setContentLength(long contentLength)
m ➤ setCorrelationId(String correlationId)
m ➤ setDelay(Integer delay)
m ➤ setDeliveryTag(long deliveryTag)

```

参数	释义
Map<String,Object> headers	消息的其他自定义参数
Integer deliveryMode	2 持久化，其他：瞬态
Integer priority	消息的优先级
String correlationId	关联 ID，方便 RPC 相应与请求关联
String replyTo	回调队列
String expiration	TTL，消息过期时间，单位毫秒

### 1.3.3. UI 管理界面的使用

RabbitMQ 可以通过命令（RabbitMQ CLI）、HTTP API 管理，也可以通过可视化的界面去管理，这个网页就是 management 插件。

#### 1.3.3.1. 启用管理插件

##### Windows 启用管理插件

```
cd C:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.6\sbin
```



```
rabbitmq-plugins.bat enable rabbitmq_management
```

## Linux 启用管理插件

```
cd /usr/lib/rabbitmq/bin
./rabbitmq-plugins enable rabbitmq_management
```

### 1.3.3.2. 管理界面访问端口

默认端口是 15672，默认用户 guest，密码 guest。

guest 用户默认只能在本机访问，远程用户需要创建其他的用户。

### 1.3.3.3. 虚拟机

在 Admin 选项卡中：



默认的虚拟机是 /，可以创建自定义的虚拟机。

### 1.3.3.4. Linux 创建 RabbitMQ 用户，权限

例如创建用户 admin，密码 admin，授权访问所有的 Vhost

```
firewall-cmd --permanent --add-port=15672/tcp
firewall-cmd --reload
rabbitmqctl add_user admin admin
rabbitmqctl set_user_tags admin administrator
rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

管理界面详细介绍略，视频中有讲解。

## 2. RabbitMQ 进阶知识

### 2.1. TTL(Time To Live)

#### 2.1.1. 消息的过期时间

有两种设置方式：

##### 1) 通过队列属性设置消息过期时间

所有队列中的消息超过时间未被消费时，都会过期。

代码位置：com.gupaoedu.ttl.TtlConfig.java

```
@Bean("ttlQueue")
public Queue queue() {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("x-message-ttl", 11000); // 队列中的消息未被消费 11 秒后过期
    return new Queue("GP_TTL_QUEUE", true, false, false, map);
}
```

##### 2) 设置单条消息的过期时间

在发送消息的时候指定消息属性。

代码位置：com.gupaoedu.ttl.TtlSender.java

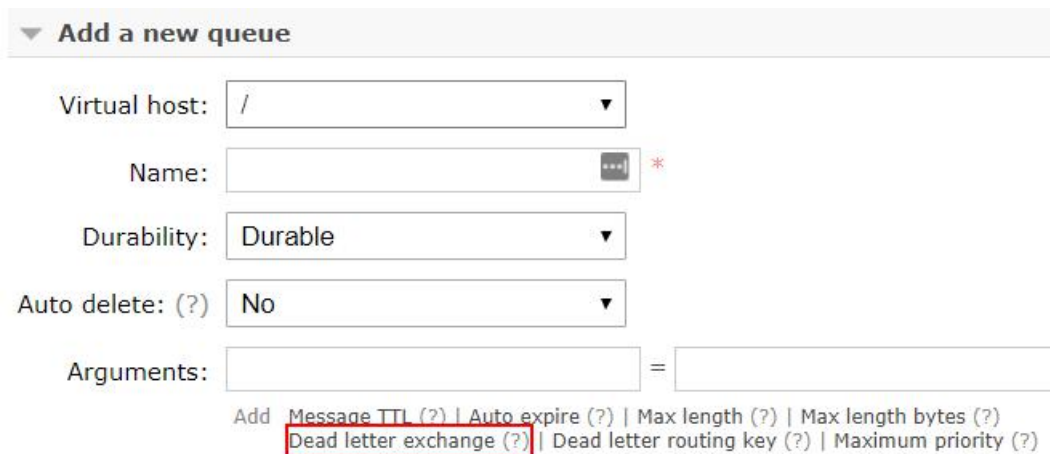
```
MessageProperties messageProperties = new MessageProperties();
messageProperties.setExpiration("4000"); // 消息的过期属性，单位 ms
Message message = new Message("这条消息 4 秒后过期".getBytes(), messageProperties);
rabbitTemplate.send("GP_TTL_EXCHANGE", "gupao.ttl", message);
```

如果同时指定了 Message TTL 和 Queue TTL，则小的那个时间生效。

## 2.2. 死信队列

消息在某些情况下会变成死信（Dead Letter）。

队列在创建的时候可以指定一个死信交换机 DLX（Dead Letter Exchange）。死信交换机绑定的队列被称为死信队列 DLQ（Dead Letter Queue），DLX 实际上也是普通的交换机，DLQ 也是普通的队列（例如替补球员也是普通球员）。



▼ Add a new queue

Virtual host: /

Name: \*

Durability: Durable

Auto delete: (?) No

Arguments: =

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

什么情况下消息会变成死信？

- 1) 消息被消费者拒绝并且未设置重回队列：(NACK || Reject) && requeue == false
- 2) 消息过期
- 3) 队列达到最大长度, 超过了 Max length(消息数)或者 Max length bytes (字节数), 最先入队的消息会被发送到 DLX。

死信队列如何使用？

- 1、声明原交换机 (GP\_ORI\_USE\_EXCHANGE)、原队列 (GP\_ORI\_USE\_QUEUE),

相互绑定。

队列中的消息 10 秒钟过期，因为没有消费者，会变成死信。指定原队列的死信交换机 (GP\_DEAD\_LETTER\_EXCHANGE) 。

代码位置: com.gupaoedu.dlx.ttl.DlxConfig.java

```
@Bean("oriUseExchange")
public DirectExchange exchange() {
    return new DirectExchange("GP_ORI_USE_EXCHANGE", true, false, new HashMap<>());
}

@Bean("oriUseQueue")
public Queue queue() {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("x-message-ttl", 10000); // 10 秒钟后成为死信
    map.put("x-dead-letter-exchange", "GP_DEAD_LETTER_EXCHANGE"); // 队列中的消息变成死信后，进入死信交换机
    return new Queue("GP_ORI_USE_QUEUE", true, false, false, map);
}

@Bean
public Binding binding(@Qualifier("oriUseQueue") Queue queue, @Qualifier("oriUseExchange") DirectExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("gupao.ori.use");
}
```

## 2、声明死信交换机 ( GP\_DEAD\_LETTER\_EXCHANGE ) 、死信队列 (GP\_DEAD\_LETTER\_QUEUE) ，相互绑定

```
@Bean("deatLetterExchange")
public TopicExchange deadLetterExchange() {
    return new TopicExchange("GP_DEAD_LETTER_EXCHANGE", true, false, new HashMap<>());
}

@Bean("deatLetterQueue")
public Queue deadLetterQueue() {
    return new Queue("GP_DEAD_LETTER_QUEUE", true, false, false, new HashMap<>());
}
```

```

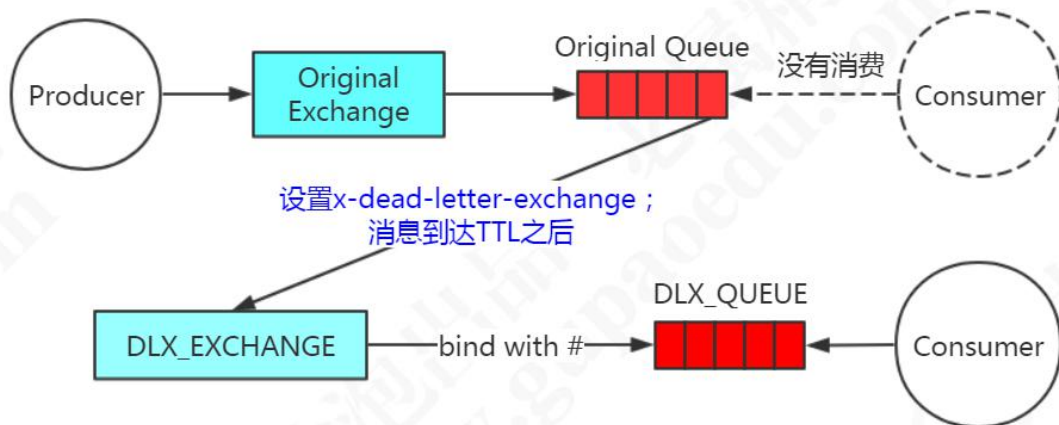
@Bean
public Binding bindingDead(@Qualifier("deatLetterQueue") Queue queue, @Qualifier("deatLetterExchange")
TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("#"); // 无条件路由
}

```

3、最终消费者监听死信队列。

4、生产者发送消息。

消息流转图



## 2.3. 延迟队列

我们在实际业务中有一些需要延时发送消息的场景，例如：

- 1、家里有一台智能热水器，需要在 30 分钟后启动
- 2、未付款的订单，15 分钟后关闭

RabbitMQ 本身不支持延迟队列，总的来说有三种实现方案：

- 1、先存储到数据库，用定时任务扫描

2、 利用 RabbitMQ 的死信队列 (Dead Letter Queue) 实现

3、 利用 rabbitmq-delayed-message-exchange 插件

### 2.3.1. TTL+DLX 的实现

基于消息 TTL，我们来看一下如何利用死信队列 (DLQ) 实现延迟队列：

总体步骤：

- 1) 创建一个交换机
- 2) 创建一个队列，与上述交换机绑定，并且通过属性指定队列的死信交换机。
- 3) 创建一个死信交换机
- 4) 创建一个死信队列
- 4) 将死信交换机绑定到死信队列
- 5) 消费者监听死信队列

消息的流转流程：

生产者——原交换机——原队列 (超过 TTL 之后) ——死信交换机——死信队列——最终消费者

使用死信队列实现延时消息的缺点：

- 1) 如果统一用队列来设置消息的 TTL，当梯度非常多的情况下，比如 1 分钟，2 分钟，5 分钟，10 分钟，20 分钟，30 分钟.....需要创建很多交换机和队列来路由消息。
- 2) 如果单独设置消息的 TTL，则可能会造成队列中的消息阻塞——前一条消息没有出队 (没有被消费)，后面的消息无法投递 (比如第一条消息过期 TTL 是 30min，第

二条消息 TTL 是 10min。10 分钟后，即使第二条消息应该投递了，但是由于第一条消息还未出队，所以无法投递）。

3) 可能存在一定的时间误差。

### 2.3.2. 基于延迟队列插件的实现（Linux）

在 RabbitMQ 3.5.7 及以后的版本提供了一个插件（rabbitmq-delayed-message-exchange）来实现延时队列功能。同时插件依赖 Erlang/OTP 18.0 及以上。

插件源码地址：

<https://github.com/rabbitmq/rabbitmq-delayed-message-exchange>

插件下载地址：

[https://bintray.com/rabbitmq/community-plugins/rabbitmq\\_delayed\\_message\\_exchange](https://bintray.com/rabbitmq/community-plugins/rabbitmq_delayed_message_exchange)

#### 1、进入插件目录

```
whereis rabbitmq  
cd /usr/lib/rabbitmq/lib/rabbitmq_server-3.6.12/plugins
```

#### 2、下载插件

```
wget  
https://bintray.com/rabbitmq/community-plugins/download_file?file_path=rabbitmq_delayed_message_exchange-0.0.1.e  
z
```



如果下载的文件名带问号则需要改名，如图：

```
2019-01-14 21:44:54 (58.6 KB/s) - 已保存 "download_file?file_path=rabbitmq_delayed_message_exchange-0.0.1.ez" [32019/32019])
```

```
mv download_file?file_path=rabbitmq_delayed_message_exchange-0.0.1.ez  
rabbitmq_delayed_message_exchange-0.0.1.ez
```

```
rabbit_common-3.6.12.ez  
rabbitmq_amqp1_0-3.6.12.ez  
rabbitmq_auth_backend_ldap-3.6.12.ez  
rabbitmq_auth_mechanism_ssl-3.6.12.ez  
rabbitmq_consistent_hash_exchange-3.6.12.ez  
rabbitmq_delayed_message_exchange-0.0.1.ez  
rabbitmq_event_exchange-3.6.12.ez  
rabbitmq_federation-3.6.12.ez  
rabbitmq_federation_management-3.6.12.ez
```

### 3、启用插件

```
rabbitmq-plugins enable rabbitmq_delayed_message_exchange
```

### 4、停用插件

```
rabbitmq-plugins disable rabbitmq_delayed_message_exchange
```

### 5、插件使用

通过声明一个 x-delayed-message 类型的 Exchange 来使用 delayed-messaging 特性。x-delayed-message 是插件提供的类型，并不是 rabbitmq 本身的（区别于 direct、topic、fanout、headers）。

Virtual host	Name	Type
/	(AMQP default)	direct
/	DELAY_EXCHANGE	x-delayed-message
/	amq.direct	direct
/	amq.fanout	fanout

代码位置: com.gupaoedu.dlx.delayplugin.DelayPluginConfig.java

```
@Bean("delayExchange")
public TopicExchange exchange() {
    Map<String, Object> argss = new HashMap<String, Object>();
    argss.put("x-delayed-type", "direct");
    return new TopicExchange("GP_DELAY_EXCHANGE", true, false, argss);
}
```

生产者:

消息属性中指定 x-delay 参数。

代码位置: com.gupaoedu.dlx.delayplugin.DelayPluginProducer.java

```
MessageProperties messageProperties = new MessageProperties();
// 延迟的间隔时间，目标时刻减去当前时刻
messageProperties.setHeader("x-delay", delayTime.getTime() - now.getTime());
Message message = new Message(msg.getBytes(), messageProperties);

// 不能在本地测试，必须发送消息到安装了插件的 Linux 服务端
rabbitTemplate.send("GP_DELAY_EXCHANGE", "#", message);
```

## 2.4. 服务端流控 (Flow Control)

<https://www.rabbitmq.com/configure.html>  
<https://www.rabbitmq.com/flow-control.html>  
<https://www.rabbitmq.com/memory.html>  
<https://www.rabbitmq.com/disk-alarms.html>

当 RabbitMQ 生产 MQ 消息的速度远大于消费消息的速度时，会产生大量的消息堆

积，占用系统资源，导致机器的性能下降。我们想要控制服务端接收的消息的数量，应该怎么做呢？

队列有两个控制长度的属性：

**x-max-length**：队列中最大存储最大消息数，超过这个数量，队头的消息会被丢弃。

**x-max-length-bytes**：队列中存储的最大消息容量（单位 bytes），超过这个容量，队头的消息会被丢弃。

需要注意的是，设置队列长度只在消息堆积的情况下有意义，而且会删除先入队的消息，不能真正地实现服务端限流。

有没有其他办法实现服务端限流呢？

#### 2.4.1. 内存控制

RabbitMQ 会在启动时检测机器的物理内存数值。默认当 MQ 占用 40% 以上内存时，MQ 会主动抛出一个内存警告并阻塞所有连接（Connections）。可以通过修改 rabbitmq.config 文件来调整内存阈值，默认值是 0.4，如下所示：

```
[{rabbit, [{vm_memory_high_watermark, 0.4}]}].
```

也可以用命令动态设置，如果设置成 0，则所有的消息都不能发布。

```
rabbitmqctl set_vm_memory_high_watermark 0.3
```

## 2.4.2. 磁盘控制

另一种方式是通过磁盘来控制消息的发布。当磁盘空间低于指定的值时（默认 50MB），触发流控措施。

例如：指定为磁盘的 30%或者 2GB：

<https://www.rabbitmq.com/configure.html>

```
disk_free_limit.relative = 3.0  
disk_free_limit.absolute = 2GB
```

## 2.5. 消费端限流

<https://www.rabbitmq.com/consumer-prefetch.html>

默认情况下，如果不进行配置，RabbitMQ 会尽可能快速地把队列中的消息发送到消费者。因为消费者会在本地缓存消息，如果消息数量过多，可能会导致 OOM 或者影响其他进程的正常运行。

在消费者处理消息的能力有限，例如消费者数量太少，或者单条消息的处理时间过长的情况下，如果我们希望在一定数量的消息消费完之前，不再推送消息过来，就要用到消费端的流量限制措施。

可以基于 Consumer 或者 channel 设置 prefetch count 的值，含义为 Consumer

端的最大的 unacked messages 数目。当超过这个数值的消息未被确认，RabbitMQ 会停止投递新的消息给该消费者。

```
channel.basicQos(2); // 如果超过 2 条消息没有发送 ACK，当前消费者不再接受队列消息  
channel.basicConsume(QUEUE_NAME, false, consumer);
```

## SimpleMessageListenerContainer

```
container.setPrefetchCount(2);
```

Spring Boot 配置：

```
spring.rabbitmq.listener.simple.prefetch=2
```

举例：

channel 的 prefetch count 设置为 5。当消费者有 5 条消息没有给 Broker 发送 ACK 后，RabbitMQ 不再给这个消费者投递消息。

参考：com.gupaoedu.limit

com.gupaoedu.amqp.container.ContainerConfig

## 3. Spring AMQP

[https://www.docs4dev.com/docs/zh/spring-amqp/2.1.2.RELEASE/reference/\\_reference.html](https://www.docs4dev.com/docs/zh/spring-amqp/2.1.2.RELEASE/reference/_reference.html)

### 3.1. Spring AMQP 介绍

思考：

Java API 方式编程，有什么问题？

Spring 封装 RabbitMQ 的时候，它做了什么事情？

- 1、管理对象（队列、交换机、绑定）
- 2、封装方法（发送消息、接收消息）

Spring AMQP 是对 Spring 基于 AMQP 的消息收发解决方案，它是一个抽象层，不依赖于特定的 AMQP Broker 实现和客户端的抽象，所以可以很方便地替换。比如我们可以使用 spring-rabbit 来实现。

```
<dependency>
  <groupId>org.springframework.amqp</groupId>
  <artifactId>spring-rabbit</artifactId>
  <version>1.3.5.RELEASE</version>
</dependency>
```

包括 3 个 jar 包：

Amqp-client-3.3.4.jar

Spring-amqp.jar

Spring.rabbit.jar

## 3.2. Spring AMQP 核心组件

### 3.2.1. ConnectionFactory

Spring AMQP 的连接工厂接口，用于创建连接。CachingConnectionFactory 是

ConnectionFactory 的一个实现类。

### 3.2.2. RabbitAdmin

RabbitAdmin 是 AmqpAdmin 的实现，封装了对 RabbitMQ 的基础管理操作，比如对交换机、队列、绑定的声明和删除等。

代码位置：

com.gupaoedu.amqp.admin.AdminConfig.java

com.gupaoedu.amqp.admin.AdminTest.java

```
// 声明一个交换机
rabbitAdmin.declareExchange(new DirectExchange("GP_ADMIN_EXCHANGE", false, false));

// 声明一个队列
rabbitAdmin.declareQueue(new Queue("GP_ADMIN_QUEUE", false, false, false));

// 声明一个绑定
rabbitAdmin.declareBinding( new Binding("GP_ADMIN_QUEUE", Binding.DestinationType.QUEUE,
    "GP_ADMIN_EXCHANGE", "admin", null));
```

为什么我们在配置文件（Spring）或者配置类（SpringBoot）里面定义了交换机、队列、绑定关系，并没有直接调用 Channel 的 declare 的方法，Spring 在启动的时候可以帮我们创建这些元数据？这些事情就是由 RabbitAdmin 完成的。

RabbitAdmin 实现了 InitializingBean 接口，里面有唯一的一个方法 afterPropertiesSet()，这个方法会在 RabbitAdmin 的属性值设置完的时候被调用。

在 afterPropertiesSet ()方法中，调用了一个 initialize()方法。这里面创建了三个



Collection，用来盛放交换机、队列、绑定关系。

最后依次声明返回类型为 Exchange、Queue 和 Binding 这些 Bean，底层还是调用了 Channel 的 declare 的方法。

```
declareExchanges(channel, exchanges.toArray(new Exchange[exchanges.size()]));
declareQueues(channel, queues.toArray(new Queue[queues.size()]));
declareBindings(channel, bindings.toArray(new Binding[bindings.size()]));
```

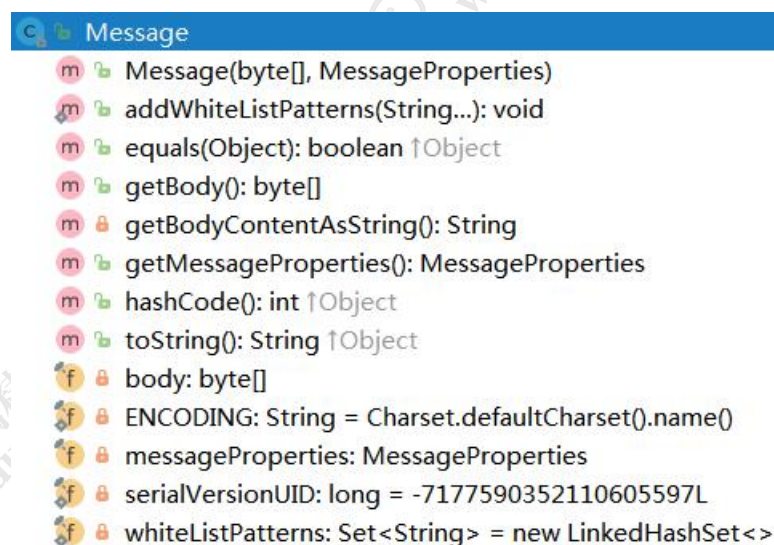
### 3. 2. 3. Message

Message 是 Spring AMQP 对消息的封装。

两个重要的属性：

body：消息内容。

messageProperties：消息属性。



### 3. 2. 4. RabbitTemplate 消息模板

RabbitTemplate 是 AmqpTemplate 的一个实现（目前为止也是唯一的实现），用

来简化消息的收发，支持消息的确认（Confirm）与返回（Return）。跟 JdbcTemplate 一样，它封装了创建连接、创建消息信道、收发消息、消息格式转换（ConvertAndSend→Message）、关闭信道、关闭连接等等操作。

针对于多个服务器连接，可以定义多个 Template。可以注入到任何需要收发消息的地方使用。

代码位置：

com.gupaoedu.amqp.template.TemplateConfig.java

确认与回发：

ConfirmCallBack

com.gupaoedu.amqp.template.TemplateConfig.java

@Bean

```
public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
    RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
    rabbitTemplate.setMandatory(true);
    rabbitTemplate.setReturnCallback(new RabbitTemplate.ReturnCallback() {
        public void returnedMessage(Message message,
                                    int replyCode,
                                    String replyText,
                                    String exchange,
                                    String routingKey){
        }
    });
    return rabbitTemplate;
}
```

ReturnCallBack

## com.gupaoedu.amqp.template.TemplateSender.java

```
rabbitTemplate.setConfirmCallback(new RabbitTemplate.ConfirmCallback(){  
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {  
        if (ack) {  
            System.out.println("消息确认成功");  
        } else {  
            // nack  
            System.out.println("消息确认失败");  
        }  
    }  
});
```

### 3.2.5. MessageListener 消息侦听

#### MessageListener

MessageListener 是 Spring AMQP 异步消息投递的监听器接口，它只有一个方法 onMessage，用于处理消息队列推送来的消息，作用类似于 Java API 中的 Consumer。

#### MessageListenerContainer

MessageListenerContainer 可以理解为 MessageListener 的容器，一个 Container 只有一个 Listener，但是可以生成多个线程使用相同的 MessageListener 同时消费消息。

Container 可以管理 Listener 的生命周期，可以用于对于消费者进行配置。

例如：动态添加移除队列、对消费者进行设置，例如 ConsumerTag、Arguments、并发、消费者数量、消息确认模式等等。

代码位置：com.gupaoedu.amqp.admin.AmqpConfig.java

@Bean

```
public SimpleMessageListenerContainer messageContainer(ConnectionFactory connectionFactory) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);
    container.setQueues(getSecondQueue(), getThirdQueue()); // 监听的队列
    container.setConcurrentConsumers(1); // 最小消费者数
    container.setMaxConcurrentConsumers(5); // 最大的消费者数量
    container.setDefaultRequeueRejected(false); // 是否重回队列
    container.setAcknowledgeMode(AcknowledgeMode.AUTO); // 签收模式
    container.setExposeListenerChannel(true);
    container.setConsumerTagStrategy(new ConsumerTagStrategy() { // 消费端的标签策略
        @Override
        public String createConsumerTag(String queue) {
            return queue + "_" + UUID.randomUUID().toString();
        }
    });
    return container;
}
```

在 SpringBoot2.0 中新增了一个 DirectMessageListenerContainer。

MessageListenerContainerFactory

Spring 去整合 IBM MQ、JMS、Kafka 也是这么做的。

代码位置：com.gupaoedu.amqp.admin.AmqpConfig.java

@Bean

```
public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(ConnectionFactory connectionFactory)
{
    SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory);
    factory.setMessageConverter(new Jackson2JsonMessageConverter());
    factory.setAcknowledgeMode(AcknowledgeMode.NONE);
    factory.setAutoStartup(true);
    return factory;
}
```

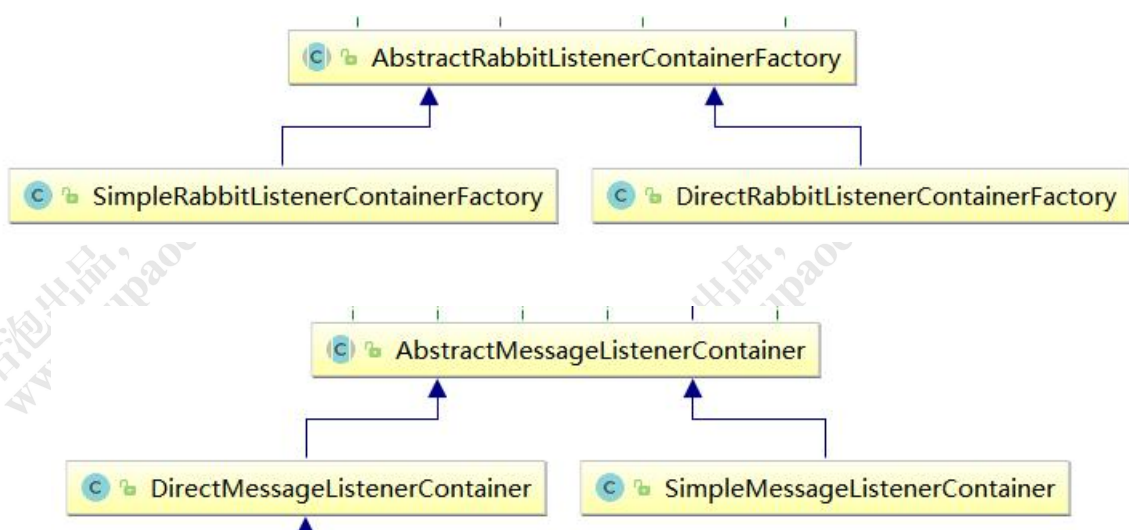
可以在消费者上指定，当我们需要监听多个 RabbitMQ 的服务器的时候，指定不同的 MessageListenerContainerFactory。

代码位置：gupaoedu-vip-springboot-project，消费者的监听类。

```
@Component
@PropertySource("classpath:gupaomq.properties")
@RabbitListener(queues = "${com.gupaoedu.firstqueue}", containerFactory="rabbitListenerContainerFactory")
public class FirstConsumer {
    @RabbitHandler
    public void process(@Payload Merchant merchant){
        System.out.println("First Queue received msg : " + merchant.getName());
    }
}
```

产生关系与继承关系：

MessageListenerContainerFactory——MessageListenerContainer——MessageListener



## 整合演示：

```
public class ContainerSender {
    public static void main(String[] args) throws Exception {
        ConnectionFactory connectionFactory = new CachingConnectionFactory(new
        URI("amqp://guest:guest@localhost:5672"));
        SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        SimpleMessageListenerContainer container = factory.createListenerContainer();
        // 不用工厂模式也可以创建
        // SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);
        container.setConcurrentConsumers(1);
        container.setQueueNames("GP_BASIC_SECOND_QUEUE");
        container.setMessageListener(new MessageListener() {
            @Override
            public void onMessage(Message message) {
                System.out.println("收到消息: "+message);
            }
        });
        container.start();

        AmqpTemplate template = new RabbitTemplate(connectionFactory);
        template.convertAndSend("GP_BASIC_SECOND_QUEUE", "msg 1");
    }
}
```

### 3.2.6. 转换器 MessageConvertor

MessageConvertor 的作用？

RabbitMQ 的消息在网络传输中需要转换成 byte[]（字节数组）进行发送，消费者需要对字节数组进行解析。

在 Spring AMQP 中，消息会被封装为 org.springframework.amqp.core.Message 对象。消息的序列化和反序列化，就是处理 Message 的消息体 body 对象。

如果消息已经是 byte[] 格式，就不需要转换。

如果是 String，会转换成 byte[]。

如果是 Java 对象，会使用 JDK 序列化将对象转换为 byte[]（体积大，效率差）。

在调用 RabbitTemplate 的 convertAndSend() 方法发送消息时，会使用 MessageConverter 进行消息的序列化，默认使用 SimpleMessageConverter。

在某些情况下，我们需要选择其他的高效的序列化工具。如果我们不想在每次发送消息时自己处理消息，就可以直接定义一个 MessageConverter。

```
@Bean
public RabbitTemplate rabbitTemplate(final ConnectionFactory connectionFactory) {
    final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
    rabbitTemplate.setMessageConverter(new Jackson2JsonMessageConverter());
    return rabbitTemplate;
}
```

MessageConverter 如何工作？

调用了 RabbitTemplate 的 convertAndSend() 方法时会使用对应的 MessageConverter 进行消息的序列化和反序列化。

序列化：Object —— Json —— Message(body) —— byte[]

反序列化：byte[] —— Message —— Json —— Object

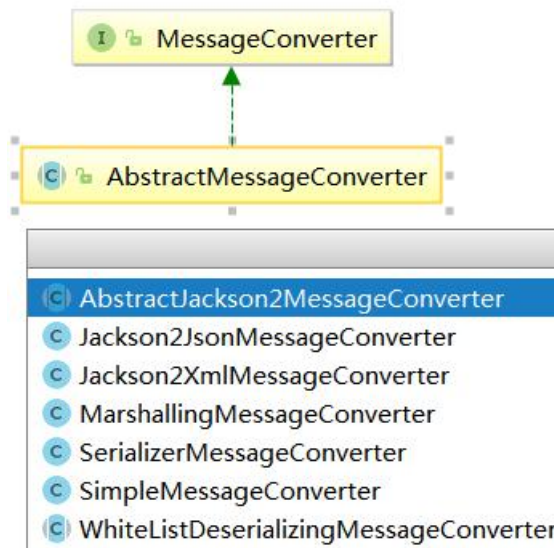
有哪些 MessageConverter？

在 Spring 中提供了一个默认转换器：SimpleMessageConverter。

Jackson2JsonMessageConverter（RabbitMQ 自带）：将对象转换为 json，然



后再转换成字节数组进行传递。



如何自定义 MessageConverter?

例如：我们要使用 Gson 格式化消息：

创建一个类，实现 MessageConverter 接口，重写 toMessage()和 fromMessage()方法。

toMessage(): Java 对象转换为 Message

fromMessage(): Message 对象转换为 Java 对象

### 3.3. Spring 集成 RabbitMQ 配置解读

```
<rabbit:connection-factory id="connectionFactory" virtual-host="/" username="guest" password="guest"
host="127.0.0.1" port="5672" />
```

```
<rabbit:admin id="connectAdmin" connection-factory="connectionFactory" />
```

```
<rabbit:queue name="MY_FIRST_QUEUE" durable="true" auto-delete="false" exclusive="false"
declared-by="connectAdmin" />
```

```
<rabbit:direct-exchange name="MY_DIRECT_EXCHANGE" durable="true" auto-delete="false"
```

```

declared-by="connectAdmin">
    <rabbit:bindings>
        <rabbit:binding queue="MY_FIRST_QUEUE" key="FirstKey">
        </rabbit:binding>
    </rabbit:bindings>
</rabbit:direct-exchange>

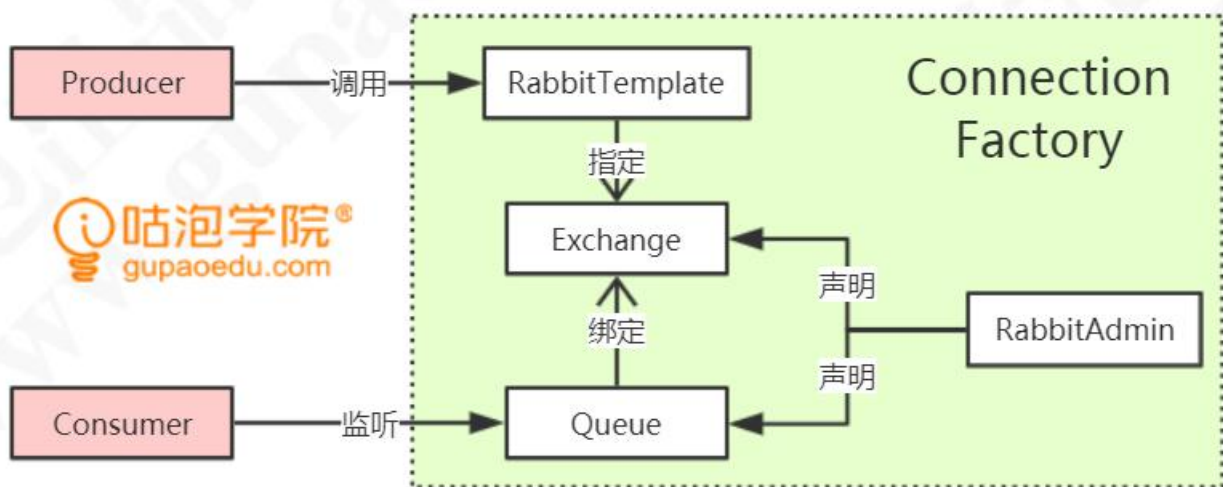
<bean id="jsonMessageConverter"
class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter" />

<rabbit:template id="amqpTemplate" exchange="${gupao.exchange}" connection-factory="connectionFactory"
message-converter="jsonMessageConverter" />

<bean id="messageReceiver" class="com.gupaoedu.consumer.FirstConsumer"></bean>

<rabbit:listener-container connection-factory="connectionFactory">
    <rabbit:listener queues="MY_FIRST_QUEUE" ref="messageReceiver" />
</rabbit:listener-container>

```

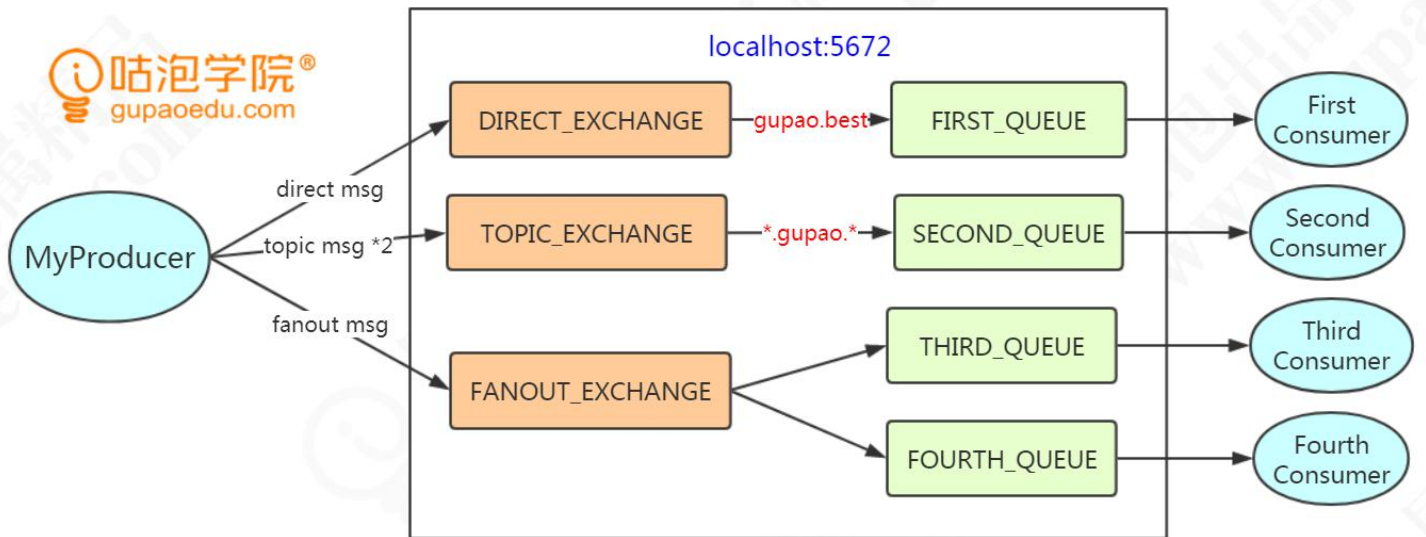


### 3.4. Spring Boot 集成 RabbitMQ

gupaoedu-vip-springboot-demo 工程中, 为什么没有定义 Spring AMQP 的任何一个对象, 也能实现消息的收发? Spring Boot 做了什么?

RabbitAutoConfiguration.java

参考工程：gupaoedu-vip-springboot-project



3 个交换机与 4 个队列绑定。4 个消费者分别监听 4 个队列。

生产者发送 4 条消息，4 个队列收到 5 条消息。消费者打印出 5 条消息。

### 3.4.1. 配置文件

RabbitConfig.java

定义交换机

```
@Bean("vipDirectExchange")
public DirectExchange getDirectExchange(){
    return new DirectExchange(directExchange);
}

@Bean("vipTopicExchange")
public TopicExchange getTopicExchange(){
    return new TopicExchange(topicExchange);
}

@Bean("vipFanoutExchange")
```

```
public FanoutExchange getFanoutExchange(){
    return new FanoutExchange(fanoutExchange);
}
```

## 定义队列

```
@Bean("vipFirstQueue")
public Queue getFirstQueue(){
    return new Queue(firstQueue);
}

@Bean("vipSecondQueue")
public Queue getSecondQueue(){
    return new Queue(secondQueue);
}

@Bean("vipThirdQueue")
public Queue getThirdQueue(){
    return new Queue(thirdQueue);
}

@Bean("vipFourthQueue")
public Queue getFourthQueue(){
    return new Queue(fourthQueue);
}
```

## 定义绑定

```
@Bean
public Binding bindFirst(@Qualifier("vipFirstQueue") Queue queue, @Qualifier("vipDirectExchange")
DirectExchange exchange){
    return BindingBuilder.bind(queue).to(exchange).with("gupao.best");
}

@Bean
public Binding bindSecond(@Qualifier("vipSecondQueue") Queue queue, @Qualifier("vipTopicExchange")
TopicExchange exchange){
    return BindingBuilder.bind(queue).to(exchange).with("*.gupao.*");
}
```

```

@Bean
public Binding bindThird(@Qualifier("vipThirdQueue") Queue queue, @Qualifier("vipFanoutExchange")
FanoutExchange exchange){
    return BindingBuilder.bind(queue).to(exchange);
}

@Bean
public Binding bindFourth(@Qualifier("vipFourthQueue") Queue queue, @Qualifier("vipFanoutExchange")
FanoutExchange exchange){
    return BindingBuilder.bind(queue).to(exchange);
}

```

### 3. 4. 2. 消费者

#### FirstConsumer.java

定义监听（后面三个消费者省略）；

在消费者类中可以有多处理（不同类型的消息）的方法。

```

@Component
@PropertySource("classpath:gupaomq.properties")
@RabbitListener(queues = "${com.gupaoedu.firstqueue}")
public class FirstConsumer {

    @RabbitHandler
    public void process(@Payload Merchant merchant){
        System.out.println("First Queue received msg : " + merchant.getName());
    }
}

```

String <b>id()</b> default ""
String <b>containerFactory()</b> default ""
String[] <b>queues()</b> default {}
Queue[] <b>queuesToDeclare()</b> default {}
boolean <b>exclusive()</b> default false
String <b>priority()</b> default ""
String <b>admin()</b> default ""
QueueBinding[] <b>bindings()</b> default {}
String <b>group()</b> default ""
String <b>returnExceptions()</b> default ""
String <b>errorHandler()</b> default ""
String <b>concurrency()</b> default ""
String <b>autoStartup()</b> default ""

注解属性

### 3.4.3. 生产者

RabbitSender.java

注入 RabbitTemplate 发送消息

```
@Autowired
```

```
AmqpTemplate gupaoTemplate;
```

```
public void send() throws JsonProcessingException {
```

```
    Merchant merchant = new Merchant(1001,"a direct msg : 中原镖局","汉中市解放路 266 号");
```

```
    gupaoTemplate.convertAndSend(directExchange,directRoutingKey, merchant);
```

```
    gupaoTemplate.convertAndSend(topicExchange,topicRoutingKey1,"a topic msg : shanghai.gupao.teacher");
```

```
    gupaoTemplate.convertAndSend(topicExchange,topicRoutingKey2,"a topic msg : changsha.gupao.student");
```

```
// 发送 JSON 字符串
```

```
    ObjectMapper mapper = new ObjectMapper();
```

```
    String json = mapper.writeValueAsString(merchant);
```

```
    System.out.println(json);
```

```
    gupaoTemplate.convertAndSend(fanoutExchange,"", json);
```

```
}
```

### 3.5. Spring Boot 参数解析



<https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/common-application-properties.html>

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

注：前缀 spring.rabbitmq.全部省略

全部配置总体上分成三大类：连接类、消息消费类、消息发送类

基于 Spring Boot 2.1.5

属性值	说明	默认值
address	客户端连接的地址，有多个的时候使用逗号分隔，该地址可以是 IP 与 Port 的结合	
host	RabbitMQ 的主机地址	localhost
port	RabbitMQ 的端口号	
virtual-host	连接到 RabbitMQ 的虚拟主机	
username	登录到 RabbitMQ 的用户名	
password	登录到 RabbitMQ 的密码	
ssl.enabled	启用 SSL 支持	false
ssl.key-store	保存 SSL 证书的地址	
ssl.key-store-password	访问 SSL 证书的地址使用的密码	
ssl.trust-store	SSL 的可信地址	
ssl.trust-store-password	访问 SSL 的可信地址的密码	
ssl.algorithm	SSL 算法，默认使用 Rabbit 的客户端算法库	
cache.channel.checkout-timeout	当缓存已满时，获取 Channel 的等待时间，单位为毫秒	
cache.channel.size	缓存中保持的 Channel 数量	
cache.connection.mode	连接缓存的模式	CHANNEL
cache.connection.size	缓存的连接数	
connection-timeout	连接超时参数单位为毫秒：设置为“0”代表无穷大	
dynamic	默认创建一个 AmqpAdmin 的 Bean	true
listener.simple.acknowledge-mode	容器的 acknowledge 模式	
listener.simple.auto-startup	启动时自动启动容器	true
listener.simple.concurrency	消费者的最小数量	
listener.simple.default-requeue-rejected	投递失败时是否重新排队	true
listener.simple.max-concurrency	消费者的最大数量	
listener.simple.missing-queues-fatal	容器上声明的队列不可用时是否失败	
listener.simple.prefetch	在单个请求中处理的消息个数，他应该大于等于事务数量	
listener.simple.retry.enabled	不论是不是重试的发布	false
listener.simple.retry.initial-interval	第一次与第二次投递尝试的时间间隔	1000ms
listener.simple.retry.max-attempts	尝试投递消息的最大数量	3
listener.simple.retry.max-interval	两次尝试的最大时间间隔	10000ms
listener.simple.retry.multiplier	上一次尝试时间间隔的乘数	1.0
listener.simple.retry.stateless	重试是有状态的还是无状态的	true
listener.simple.transaction-size	在一个事务中处理的消息数量。为了获得最佳效果，该值应设置为小于等于每个请求中处理的消息个数，即	



	listener.prefetch 的值	
publisher-confirms	开启 Publisher Confirm 机制	
publisher-returns	开启 Publisher Return 机制	
template.mandatory	启用强制信息	false
template.receive-timeout	receive()方法的超时时间	0
template.reply-timeout	sendAndReceive()方法的超时时间	5000
template.retry.enabled	设置为 true 的时候 RabbitTemplate 能够实现重试	false
template.retry.initial-interval	第一次与第二次发布消息的时间间隔	1000
template.retry.max-attempts	尝试发布消息的最大数量	3
template.retry.max-interval	尝试发布消息的最大时间间隔	10000
template.retry.multiplier	上一次尝试时间间隔的乘数	1.0

### 3.6. Spring Boot 项目实战分析

#### 项目介绍

商户管理系统，用于管理与公司合作的商户信息，包括商户准入和审核的全流程。有很多下游业务系统要用到商户信息，每一个系统都会在自己的数据库里面存放商户的关键信息。比如提单系统提单，要商户的名称。放款系统要放款，需要商户的账号户名。风控系统也要关注商户信息的变动。这种同步数据的场景，我们原来用的是 ETL，定时同步，也就是依赖于一个核心的数据库，我们只修改核心数据的商户信息，其他系统自己定时去核心系统拉取数据。

这种方式有两个缺点，一个是实时性不高，因为 ETL 的定时任务也不可能每时每分都运行。还有一个，是依赖于核心的数据库，一旦核心的数据库出现问题，其他所有的系统都无法同步，耦合性过高。

所以这种情况就可以改成用 MQ 同步，因为考虑到还有其他系统也要用到商户信息，我们采用了广播的方式。

#### SSM 增删改查

这里我们做了一个界面，模拟商户信息增删改查的情况。一旦商户信息发生变化，除了修改我们自己系统的数据之外，我们还要发送 MQ 广播出去，让所有的下游系统拿到消息，修改自己数据库的商户信息。

商户数据增删改查这一块我们就不用说了，SSM 的框架，非常简单。关键就是 MQ 的消息什么时候发，在哪一层发，是先更新数据库后发送 MQ 消息还是后先发送 MQ 消息发送数据库。

## 消息发送

我们看一下 Service 的实现类。以更新商户信息为例。我们先更新数据库，然后调用注入的 template 发送消息。

gupaoedu-vip-springboot-project, 生产者工程:

```
com.gupaoedu.service.impl.MerchantServiceImpl
```

这里的顺序千万要注意。一定是先更新数据库后发送消息。否则，数据库回滚的话就会导致数据一致性问题。

但是，如果先更新数据库成功，后发送消息失败了呢？比如，服务器没有成功接收，或者路由出现问题，或者在 Queue 存储出现问题，或者消费者消费时出现问题，怎么解决？

作者：咕泡学院-青山

最后更新时间：2019 年 8 月 25 日 00:12:12