## 第1章 RabbitMQ简介

一款消息中间件。高可靠、易扩展、高可用及丰富的功能

### 什么是消息中间件

Message是指在应用间传送的数据。消息可以非常可靠，比如只包含文本字符串、JSON等，也可以很复杂，比如内嵌对象。

消息队列中间件（Message Queue Middleware，MQ）是指利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。通过提高消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。

消息队列，它一般有两种传递模式：

1. 点对点（P2P，Point-to-Point）模式：

点对点模式是基于队列的，消息生产者发送消息到队列，消息消费者从队列中接收消息，队列的存在使得消息的异步传输成为可能。

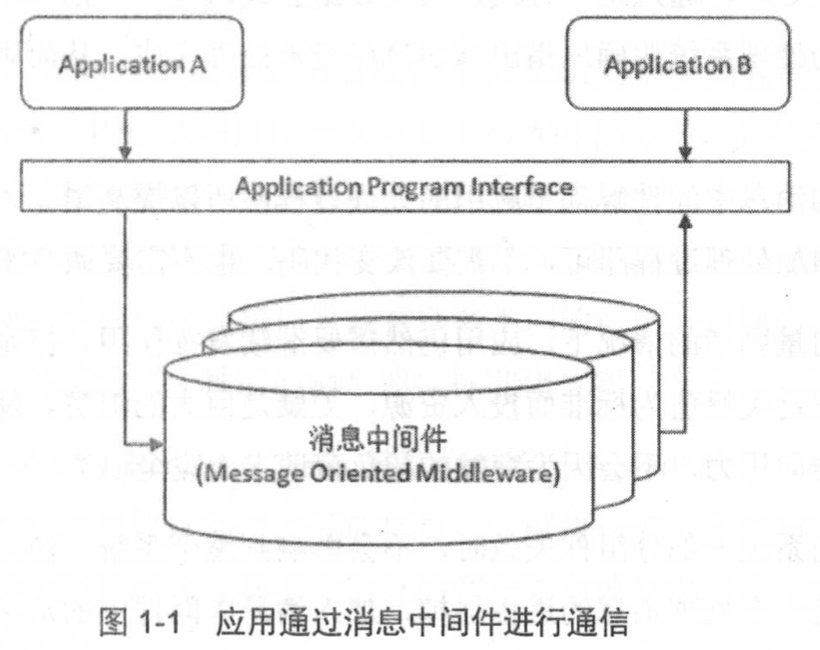
1. 发布/订阅（Pub/Sub）模式：

发布订阅模式定义了如何向一个内容节点发布和订阅消息，这个内容节点称为主题（topic，可以认为是消息传递的中介），消息发布者将消息发布到某个主题，而消息订阅者则从主题中订阅消息。主题使得消息的订阅者与消息的发布者互相保持独立，不需要进行接触即可保证消息的传递，发布/订阅模式在消息的一对多广播时采用。

目前比较主流的消息中间件有RabbitMQ、Kafka、ActiveMQ、RocketMQ等。面向消息的中间件（Message Oriented Middleware，MOM）提供了以松散耦合的灵活方式集成应用程序的一种机制。它们提供了基于存储和转发的应用程序之间的异步数据发送，即应用程序彼此不直接通信，而是与作为中介的消息中间件通信。消息中间件提供了有保证的消息发送，应用程序开发人员无需了解远程过程调用（RPC）和网络通信协议的细节。

消息中间件适用于需要可靠的数据传送的分布式环境。采用消息中间件的系统中，不同的对象之间通过传递消息来激活对方的事件，以完成相应的操作。发送者将消息发送给消息服务器，消息服务器将消息存放在若干队列中，在合适的时候再将消息转发给接收者。消息中间件能在不同平台之间通信，它常被用来屏蔽各种平台及协议之间的特性，实现应用程序之间的协同，其优点在于能够在客户和服务器之间提供同步和异步的连接，并且在任何时刻都可以将消息进行传送或者存储转发，这也是它比远程过程调用更进步的原因。

举例：应用程序A与应用程序B通过使用消息中间件的应用程序编程接口发送消息来进行通信。



消息中间件将消息路由给应用程序B，这样消息就可存在于完全不同的计算机上。消息中间件负责处理网络通信，如果网络连接不可用，消息中间件会存储消息，直到连接变得可用，再将消息转发给应用程序B。灵活性的另一方面体现在，当应用程序A发送消息时，应用程序B甚至可以处于不运行状态，消息中间件将保留这份消息，直到应用程序B开始执行并消费消息，这样还防止了应用程序A因为等待应用程序B消费而出现阻塞。这种异步通信方式要求应用程序的设计与实现在大多数应用不同。不过对于时间无关或并行处理场景，它可能是一个及其有用的方法。

### 1.2 消息中间件的作用

消息中间件凭借其独到的特性，在不同的应用场景下可以展现不同的作用。总的来说，消息中间件的作用可以概括如下。

①解耦：在项目启动之初来预测将来会碰到什么需求是及其困难的，消息中间件在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口，这允许你独立地扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束即可。

②冗余（存储）：有些情况下，处理数据的过程会失败。消息中间件可以把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。在把一个消息从消息中间件中删除之前，需要你的处理系统明确地指出该消息已经被处理完成，从而确保你的数据被安全地保存直到你使用完毕。

1. 扩展性：因为消息中间件解耦了应用的处理过程，所以提高消息入队和处理的效率是很容易的，只要另外增加处理过程即可，不需要改变代码，也不需要调节参数。
2. 削峰：在访问剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果以能处理这类峰值为标准而投入资源，无疑是巨大的浪费。使用消息中间件能够使关键组件支撑突发访问压力，不会因为突发的超负荷请求而完全崩溃。
3. 可恢复性：当系统一部分组件失效时，不会影响到整个系统。消息中间件降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入消息中间件中的消息仍然可以在系统恢复后进行处理。
4. 顺序保证：在大多数使用场景下，数据处理的顺序很重要，大部分消息中间件支持一定程度上的顺序性。
5. 缓冲：在任何系统中，都会存在需要不同处理时间的元素。消息中间件通过一个缓冲层来帮助任务最高效率地执行，写入消息中间件的处理会尽可能快速。该缓冲层有助于控制和优化数据流经过系统的速度。
6. 异步通信：在很多时候应用不想也不需要立即处理消息。消息中间件提供了异步处理机制，允许应用把一些消息放入消息中间件中，但并不立即处理它，在之后需要的时候再慢慢处理。

### 1.3 RabbitMQ的起源

RabbitMQ是采用了Erlang语言实现AMQP（Advanced Message Queuing Protocol，高级消息队列协议）的消息中间件，它最初起源于金融系统，用于在分布式系统中存储转发消息。

在此之前，有一些消息中间件的商业实现，比如微软的MSMQ（MicroSoft Message Queue）、IBM的WebSphere等。由于高昂的价格，一般只应用于大型组织架构，它们需要可靠性、解耦及实时消息通信的功能。由于商业壁垒，商业MQ供应商想要解决应用互通的问题，而不是去创建标准来实现不同的MQ产品间的互通，或者允许应用程序更改MQ平台。

为了打破这个壁垒，同时为了能够让消息在各个消息队列平台间互融互通，JMS（Java Message Service）应运而生。JMS试图通过提供公共Java API的方式，隐藏单独MQ产品供应商提供的实际接口，从而跨越了壁垒，以及解决了互通问题。从技术上讲，Java应用程序只需针对JMS API编程，选择合适的MQ驱动即可，JMS会打理其他部分。ActiveMQ就是JMS的一种实现。不过尝试使用单独标准化接口来胶合众多不同的接口，最终会暴露问题，使得应用程序变得更加脆弱。所以急需一种新的消息通信标准化方案。

在2006年6月，有Cisco、Redhat、iMatix等联合制定了AMQP的公开标准，它是应用层协议的一个开放标准，以解决众多消息中间件的需求和拓扑结构问题。它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，并不受产品、开发语言等条件的限制。

RabbitMQ最初版本实现了AMQP的一个关键特性：使用协议本身就可以对队列和交换器（Exchange）这样的资源进行配置。对于商业MQ供应商来说，资源配置需要通过管理终端的特定工具才能完成。RabbitMQ的资源配置能力使其成为构建分布式应用的最完美的通信总线，特别有助于充分利用基于云的资源和进行快速开发。

RabbitMQ的特点概括：

* 可靠性：使用一些机制来确保可靠性，如持久化、传输确认以及发布确认等。
* 灵活的路由：在消息进入队列之前，通过交换器来路由消息。对于典型的路由功能，RabbitMQ已经提供了一些内置的交换器来实现。针对更复杂的路由功能，可以将多个交换器绑定在一起，也可以通过插件机制来实现自己的交换器。
* 扩展性：多个RabbitMQ节点可以组成一个集群，也可以根据实际业务情况动态地扩展集群中节点。
* 高可用性：队列可以在集群中的机器上设置镜像，使得在部分节点出现问题的情况下队列仍然可用。
* 多种协议：除了原生支持AMQP协议，还支持STOMP、MQTT等多种消息中间件协议。
* 多语言客户端：几乎支持所有常用语言。
* 管理界面：RabbitMQ提供了一个易用的用户界面，使得用户可以监控和管理消息、集群中的节点等。
* 插件机制：提供许多插件，以实现从多方面进行扩展，当然也可以编写自己的插件。

### 1.4 RabbitMQ的安装及简单使用

#### 1.4.1 Erlang的2中安装方法：

1）从Erlang Solution安装(推荐)

# 添加erlang solutions源

$ wget https://packages.erlang-solutions.com/erlang-solutions-1.0-1.noarch.rpm

$ sudo rpm -Uvh erlang-solutions-1.0-1.noarch.rpm

$ sudo yum install erlang

2）从EPEL源安装（这种方式安装的Erlang版本可能不是最新的，有时候不能满足RabbitMQ需要的最低版本）

# 启动EPEL源

$ sudo yum install epel-release

# 安装erlang

$ sudo yum install erlang

#### 1.4.2 安装RabbitMQ：

#先下载rpm：

wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.6/rabbitmq-server-3.6.6-1.el7.noarch.rpm

#下载完成后安装：

yum install rabbitmq-server-3.6.6-1.el7.noarch.rpm

#### 1.4.3 RabbitMQ的运行：

rabbitmq-server启动，自带web管理界面，通过浏览器访问[http://localhost:15672](https://link.jianshu.com/?t=http://localhost:15672)

默认用户名密码为guest

#### 1.4.4 生产和消费消息

演示如何使用RabbitMQ Java客户端生产和消费消息。

目前最新的RabbitMQ Java客户端版本为5.7.3

<!-- https://mvnrepository.com/artifact/com.rabbitmq/amqp-client -->

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>5.7.3</version>

</dependency>

默认情况下，访问RabbitMQ服务的用户名和密码都是“guest”，这个账户有限制，默认只能通过本地网络（如localhost）访问，远程网络访问受限所以在实现生产和消费之前，需要另外添加一个用户，并设置相应的访问权限。

添加新用户，用户名root，密码539976

$ rabbitmqctl add\_user 用户名 密码

设置root用户为管理员角色

$ rabbitmqctl set\_user\_tags 用户名 administrator

为root用户设置所有权限

$ rabbitmqctl set\_permissions -p / 用户名 ".\*" ".\*" ".\*"

如果在使用RabbitMQ的过程中遇到类似如下的报错，那么很可能就是账户管理的问题，需要根据上面的步骤进行设置，之后再运行。

Exception in thread ”main" com.rabbitmq.client.AuthenticationFailureException: ACCESS REFUSED - Login was refused using authentication mechanism PLAIN. For details see the broker logfile.

首先生产者发送一条消息“HelloWorld！”至RabbitMQ中，之后再由消费者消费。

生产者客户端代码：

public class RabbitProducer {

private static final String *EXCHANGE\_NAME* = "exchange\_demo";  
 private static final String *ROUTING\_KEY* = "routingkey\_demo";  
 private static final String *QUEUE\_NAME* = "queue\_demo";  
 private static final String *IP\_ADDRESS* = "172.0.0.1";  
 private static final int *PORT* = 5672;  
  
 public static void main(String[] args) throws IOException, TimeoutException {  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setHost(*IP\_ADDRESS*);  
 factory.setPort(*PORT*);  
 factory.setUsername("root");  
 factory.setPassword("539976");  
  
 //创建连接  
 Connection connection = factory.newConnection();  
 //创建信道  
 Channel channel = connection.createChannel();  
 //创建一个type="direct"、持久化的、非自动删除的交换器  
 channel.exchangeDeclare(*EXCHANGE\_NAME*, "direct", true, false, null);  
 //创建一个持久化、非排他的、非自动删除的队列  
 channel.queueDeclare(*QUEUE\_NAME*, true, false, false, null);  
 //将交换器与队列通过路由键绑定  
 channel.queueBind(*QUEUE\_NAME*, *EXCHANGE\_NAME*, *ROUTING\_KEY*);  
  
 //发送一条持久化消息  
 String message = "Hello World!";  
 channel.basicPublish(*EXCHANGE\_NAME*, *ROUTING\_KEY*,   
 MessageProperties.*PERSISTENT\_TEXT\_PLAIN*,   
 message.getBytes());  
  
 //关闭资源  
 channel.close();  
 connection.close();  
 }  
}

上面的生产者客户端的代码

1. 首先和RabbitMQ服务器建立一个连接（Connection）；
2. 然后在这个连接之上创建一个信道（Channel）；
3. 之后创建一个交换器（Exchange）和一个队列（Queue），并通过路由键进行绑定；
4. 然后发送一条消息；
5. 最后关闭资源。

消费者客户端代码：

public class RabbitConsumer {

private static final String *QUEUE\_NAME* = "queue\_demo";  
 private static final String *IP\_ADDRESS* = "127.0.0.1";  
 private static final int *PORT* = 5672;  
  
 public static void main(String[] args) throws IOException, TimeoutException, InterruptedException {  
 Address[] addresses = new Address[]{new Address(*IP\_ADDRESS*, *PORT*)};  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setUsername("root");  
 factory.setPassword("539976");  
  
 //此处连接方式与生产者的demo略有不同，注意辨别区别  
 //创建连接  
 Connection connection = factory.newConnection(addresses);  
 //创建信道  
 final Channel channel = connection.createChannel();  
 //设置客户端最多接收未被ack的消息的个数  
 channel.basicQos(64);  
 Consumer consumer = new DefaultConsumer(channel) {  
 @Override  
 public void handleDelivery(String consumerTag,  
 Envelope envelope,  
 AMQP.BasicProperties properties,  
 byte[] body) throws IOException {  
 System.*out*.println("recv message:" + new String(body));  
 try {  
 TimeUnit.*SECONDS*.sleep(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 channel.basicAck(envelope.getDeliveryTag(), false);  
 }  
 };  
  
 channel.basicConsume(*QUEUE\_NAME*, consumer);  
 //等待回调函数执行完毕之后，关闭资源  
 TimeUnit.*SECONDS*.sleep(5);  
 channel.close();  
 connection.close();  
 }  
}

这里采用的是继承DefaultConsumer的方式来实现消费，有过RabbitMQ使用经验的读者也许会喜欢采用QueuingConsumer的方式来实现消费，但是我们并不推荐，使用QueuingConsumer会有一些隐患。同时，在RabbitMQ Java客户端4.0.0版本开始将QueuingConsumer标记为@Deprecated，在后面的大版本中会删除这个类。

## 第2章 RabbitMQ入门

针对RabbitMQ本身及其所遵循的AMQP协议中的一些细节做进一步的探究。RabbitMQ的模型架构是什么？

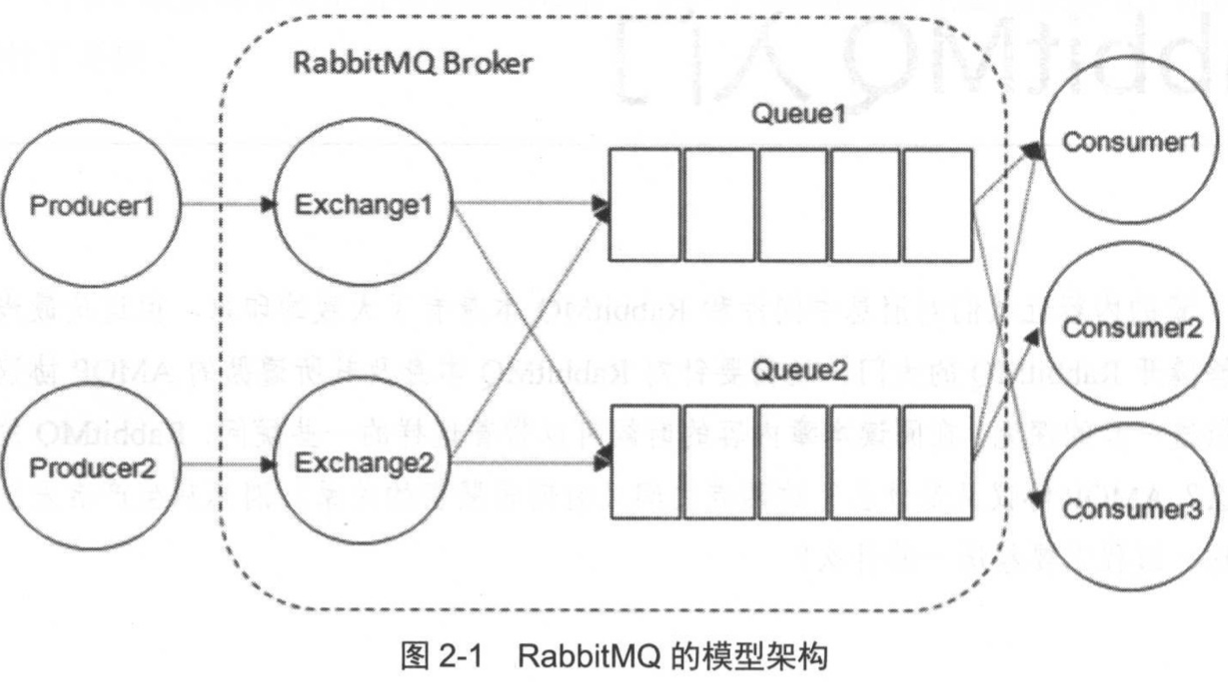
AMQP协议是什么？

两者之间有何种紧密的关系？

消息从生产者发出到消费者消费的这一过程中要经历些什么？

### 2.1 相关概念介绍

RabbitMQ整体上是一个生产者与消费者模型，主要负责接收、存储和转发消息。可以把消息传递的过程想象成：当你将一个包裹送到邮局，邮局会暂存并最终将邮件通过邮递员送到收件人的手上，RabbitMQ就好比由邮局、邮箱和邮递员组成的一个系统。从计算机术语层面来说，RabbitMQ模式更像是一种交换机模型。



#### 2.1.1 生产者和消费者

Producer：生产者，投递消息的一方。

生产者创建消息，然后发布到RabbitMQ中，消息一般包含2个部分：消息体和标签（Label）。消息体也可以称之为payload（有效载荷），在实际应用中，消息体一般是一个带有业务逻辑结构的数据，比如一个JSON字符串。当然可以进一步对这个消息体进行序列化操作。消息的标签用来表述这条消息，比如一个交换器的名称和一个路由键。生产者把消息交由RabbitMQ，RabbitMQ之后会根据标签把消息发送给感兴趣的消费者（Consumer）。

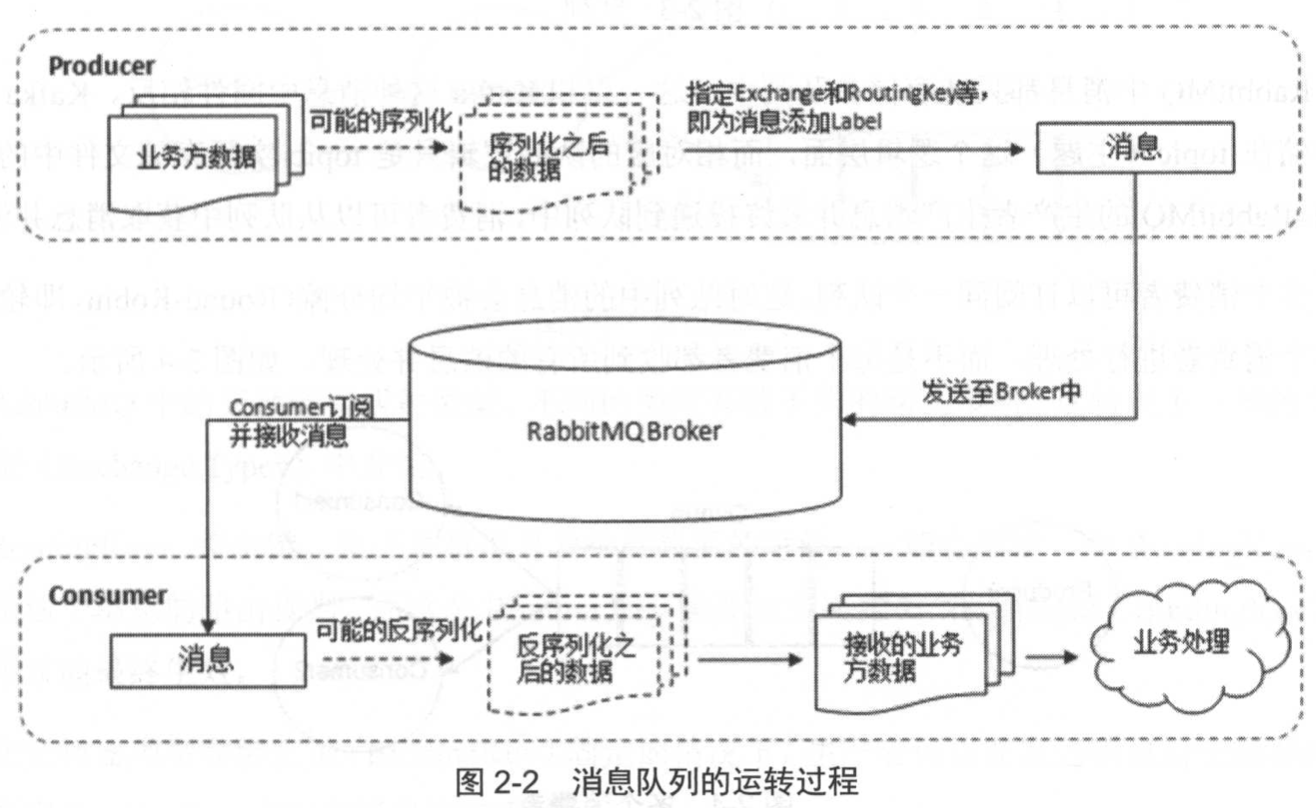
Consumer：消费者，接收消息的一方。

消费者连接到RabbitMQ服务器，并订阅到队列上。当消费者消费一条消息时，只是消费消息的消息体（payload）。在消息路由的过程中，消息的标签会丢弃，存入到队列的消息只有消息体，消费者也只会消费到消息体，也就不知道消息的生产者是谁，当然消费者也不需要知道。

Broker：消息中间件的服务节点。

对于RabbitMQ来说，一个Broker可以简单地看做一个RabbitMQ服务节点，或者RabbitMQ服务实例。大多数情况下也可以将一个RabbitMQ Broker看做一台RabbitMQ服务器。

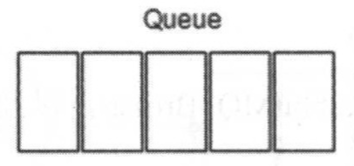
下图展示了生产者将消息存入RabbitMQ Broker，以及消费者从Broker中消费数据的完整流程。



1. 首先生产者将业务方数据进行可能的包装，之后封装成消息；
2. 发送（AMQP协议里这个动作对应的命令为Basic.Publish）到Broker中；
3. 消费者订阅并接收消息（AMQP协议里这个动作对应的命令为Basic.Consume或者Basic.Get）；
4. 经过可能的解包处理得到原始数据，之后再进行业务处理逻辑。（这个业务处理逻辑并不一定需要和接收消息的逻辑使用同一个线程。消费者进程可以使用一个线程去接收消息，存入到内存中，比如使用Java中的BlockingQueue。业务处理逻辑使用另一个线程从内存中读取数据，这样可以将应用进一步解耦，提高整个应用的处理效率。）

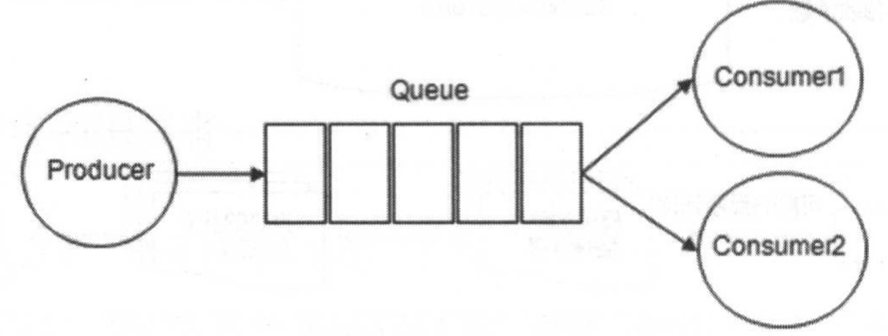
#### 2.1.2 队列

Queue：队列，是RabbitMQ的内部对象，用于存储消息。



RabbitMQ中消息都只能存储在队列中，这一点和Kafka这种消息中间件相反。Kafka将消息存储在topic（主题）这个逻辑层面，而相对应的队列逻辑只是topic实际存储文件中的位移标识。RabbitMQ的生产者生产消息并最终投递到队列中，消费者可以从队列中获取消息并消费。

多个消费者可以订阅同一个队列，这时队列中的消息会被平均分摊（Round-Robin，轮询）给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。

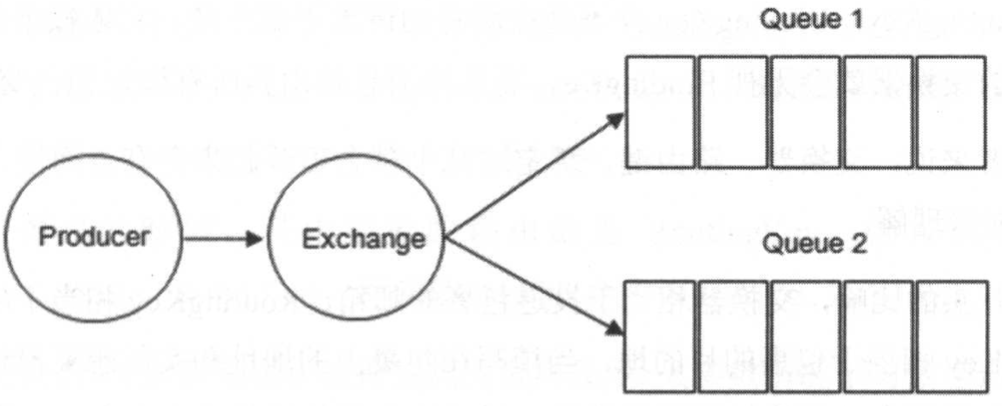


RabbitMQ不支持队列层面的广播消费，如果需要广播消费，需要在其上进行二次开发，处理逻辑会变得异常复杂，同时也不建议这么做。

#### 2.1.3 交换器、路由键、绑定

Exchange：交换器。

生产者将消息发送到Exchange，由交换器将消息路由到一个或多个队列中。如果路由不到，或许返回给生产者，或许直接丢弃。这里可以将RabbitMQ中的交换器看做一个简单的实体。



RabbitMQ中的交换器有四种类型，不同类型有着不同的路由策略。

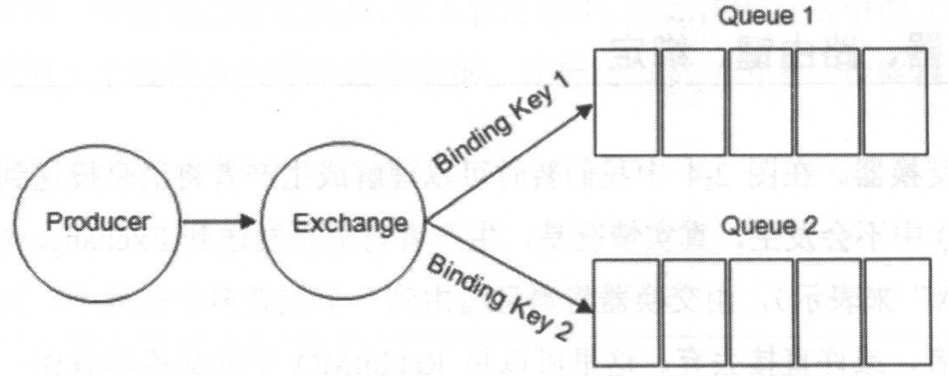
RoutingKey：路由键。

生产者将消息发给交换器的时候，一般会指定一个RoutingKey，用来指定这个消息的路由规则，而这个RoutingKey需要与交换器类型和绑定键（BindingKey）联合使用才能生效。

在交换器类型和绑定键（BindingKey）固定的情况下，生产者可以在发送消息给交换器时，通过指定RoutingKey来决定消息流向哪里。

Binding：绑定。

RabbitMQ中通过绑定将交换器与队列关联起来，在绑定的时候一般会指定一个绑定键（BindingKey），这样RabbitMQ就知道如何正确地将消息路由到队列了。



生产者将消息发送给交换器时，需要一个RoutingKey，当BindingKey和RoutingKey相匹配时，消息会被路由到对应的队列中。在绑定多个队列到同一个交换器的时候，这些绑定允许使用相同的BindingKey。BindingKey并不是在所有情况下都生效，它依赖于交换器的队列类型，比如fanout类型的交换器就会无视BindingKey，而是将消息路由到所有绑定到该交换器的队列中。

交换器相当于投递包裹的邮箱，RoutingKey相当于填写在包裹上的地址，BindingKey相当于包裹的目的地，当填写在包裹上的地址和实际想要投递的地址相匹配时，那么这个包裹就会被正确投递到目的地，最后这个目的地的“主人”——队列可以保留这个包裹。如果填写的地址出错，邮递员不能正确投递到目的地，包裹可能会回退给寄件人，也可能被丢弃。

在某些情形下，RoutingKey和BindingKey可以看做同一个东西。

channel.exchangeDeclare(EXCHANGE\_NAME, “direct”, true, false, null);

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

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, ROUTING\_KEY);

String message = “Hello World!”;

channel.basicPublish(EXCHANGE\_NAME, ROUTING\_KEY,

MessageProperties.PERSISTENT\_TEXT\_PLAIN,

message.getByte());

声明了一个direct类型的交换器，然后将交换器和队列绑定起来。

在本该使用BindingKey的queueBind方法中却和basicPublish方法同样使用了RoutingKey，这样做的潜台词是：这里的RoutingKey和BindingKey是同一个东西。在direct交换器类型下，RoutingKey和BindingKey需要完全匹配才能使用，所以上面代码中采用了此种写法会显得方便许多。

但是在topic交换器类型下，RoutingKey和BindingKey之间需要做模糊匹配，两者并不是相同的。

BindingKey其实也属于路由键中的一种，官方解释为：the routing key to use for the binding。大多数时候都把BindingKey和RoutingKey看作RoutingKey，为避免混淆可以这么理解：

* 在使用绑定的时候，其中需要的路由键是BindingKey。涉及的客户端方法如：channel.exchangeBind、channel.queueBind，对应的AMQP命令为Exchange.Bind、Queue.Bind。
* 在发送消息的时候，其中需要的路由键是RoutingKey。涉及的客户端方法如channel.basicPublish，对应的AMQP命令为Basic.Publish。

#### 2.1.4 交换器类型

RabbitMQ常用的交换器类型有fanout、direct、topic、headers。AMQP协议里还提供到了另外两种类型：System和自定义。

fanout

它会把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中。

direct

它会把消息路由到那些BindingKey和RoutingKey完全匹配的队列中。

如下，交换器的类型为direct，如果我们发送一条消息，并在发送消息的时候设置路由键为“warning”，则消息会路由到Queue1和Queue2，对应的示例代码如下：

channel.basicPublsh(EXCHANGE\_NAME, “warning”,

MessageProperties.PERSISTENT\_TEXT\_PLAIN,

message.getBytes());

