# 概述

## 1.1消息队列的几种模式

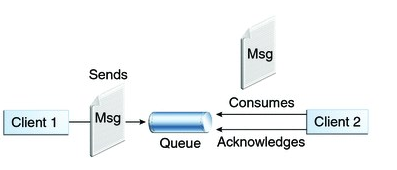
消息队列是分布式系统开发中的一种重要的中间件，在消息队列中，存在两种场景：

·生产者-消费者模式：一个消息只能有一个消费者

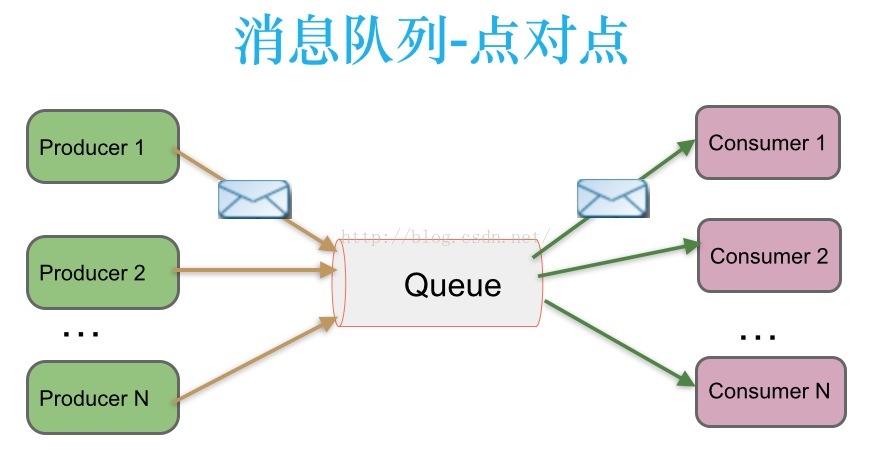
·发布者-订阅者模式：一个消息可以被多个消费者收到

### 1.1.1点对点模式

点对点模式是最原始的生产者-消费者模型，消息的生产者向一个特定的消息队列(queue)发送消息，消息的消费者从该队列中接收消息并消费(consuumes)，每一个成功处理的消息(msg)都由消息消费者签收确认（Acknowledge）



点对点的消息队列有一个特点：一个消息只能被消费一次，也就是说，当存在多个消费者时，对于一个消息，有且只有一个消费者会接受到。即一旦消费者读取队列中的消息，消息就从该队列中消失，这和发布-订阅模式有着极大的不同



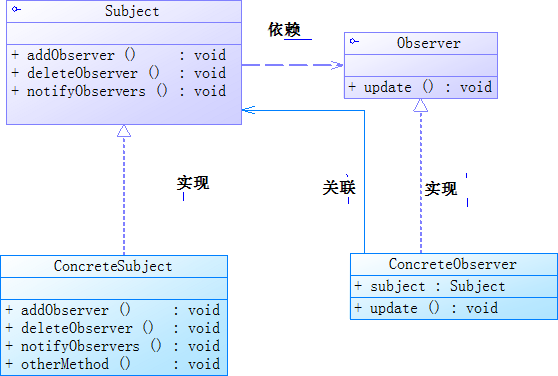
### 1.1.2发布-订阅模式

在讨论发布-订阅模式之前，我们需要先了解一种设计模式——观察者模式

#### 1.1.2.1观察者模式

什么是观察者模式？观察者模式不同于上面所说的模式，他是23中软件设计模式中的最常用的一种。它在对象之间定义了一对多的依赖，这样一来，当父对象改变状态，它的依赖对象会收到通知并自动更新。

比如一个公众号(被观察者)，公众号持有一个用户列表，存储所有关注该公众号的用户信息(观察者)，当公众号有文章推送时，就会去遍历所有持有用户，并通知他们，用户接到通知后，自行进行相应操作



从上图中可以看出，该模式包含四种角色：

* 抽象被观察者角色(Subject)：也就是一个抽象Subject，它把所有观察者对象的引用保存在一个集合中，每个Subject都可以有任意数量的观察者。抽象Subject提供可以增加和删除观察者角色的接口 为，还有一个通知观察者对象的接口。一般用一个抽象类和接口来实现。

public interface Subject {

public void registerObserver(Observer o);

public void removeObserver(Observer o);

public void notifyObserver();

}

* 抽象观察者角色(Observer)：为所有的具体观察者定义一个接口，在得到主题通知时更新自己，执行相关操作。

public interface Observer {

public void update(String message);

}

* 具体被观察者角色(ConcreateSubject)：也就是一个具体的主题，在集体主题的内部状态改变时，所有登记过的观察者发出通知。

public class ConcreateSubject implements Subject {

//注意到这个List集合的泛型参数为Observer接口，设计原则：面向接口编程而不是面向实现编程

private List<Observer> list;

private String message;

public ConcreateSubject () {

list = new ArrayList<Observer>();

}

@Override

public void registerObserver(Observer o) {

list.add(o);

}

@Override

public void removeObserver(Observer o) {

if(!list.isEmpty())

list.remove(o);

}

//遍历

@Override

public void notifyObserver() {

for(int i = 0; i < list.size(); i++) {

Observer observer = list.get(i);

observer.update(message);

}

}

public void setInfomation(String s) {

this.message = s;

System.out.println("微信服务更新消息： " + s);

//消息更新，通知所有观察者

notifyObserver();

}

}

* 具体观察者角色(ConcreateSubject)：实现抽象观察者角色所需要的更新接口，一边使本身的状态与制图的状态相协调。

public class ConcreateSubject implements Observer {

private String name;

private String message;

public User(String name) {

this.name = name;

}

@Override

public void update(String message) {

this.message = message;

read();

}

public void read() {

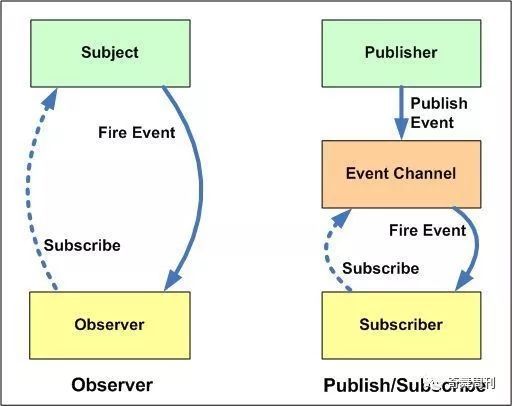
System.out.println(name + " 收到推送消息： " + message);

}

}

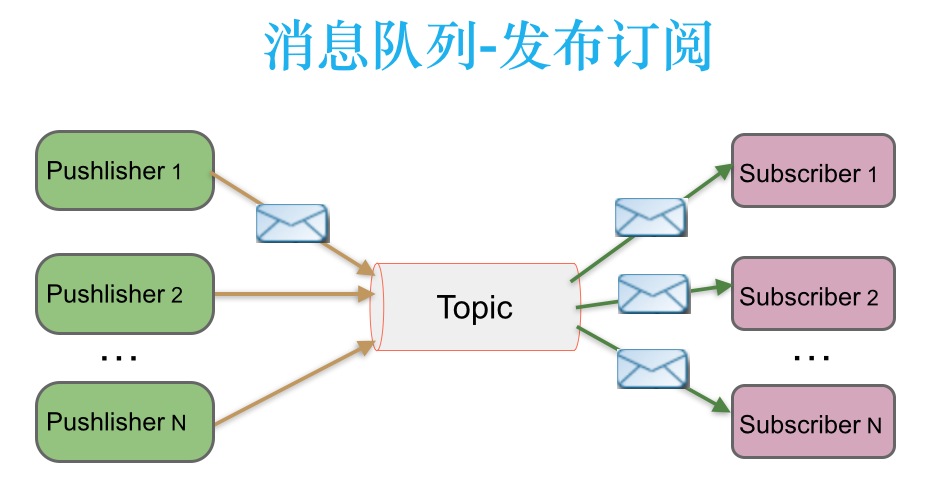
#### 1.1.2.2发布-订阅模式

那么发布-订阅模式和观察者模式有什么关联吗？发布-订阅模式是消息队列的一种消息模式，不属于23种设计模式，其类似于观察者模式，但也有不同之处



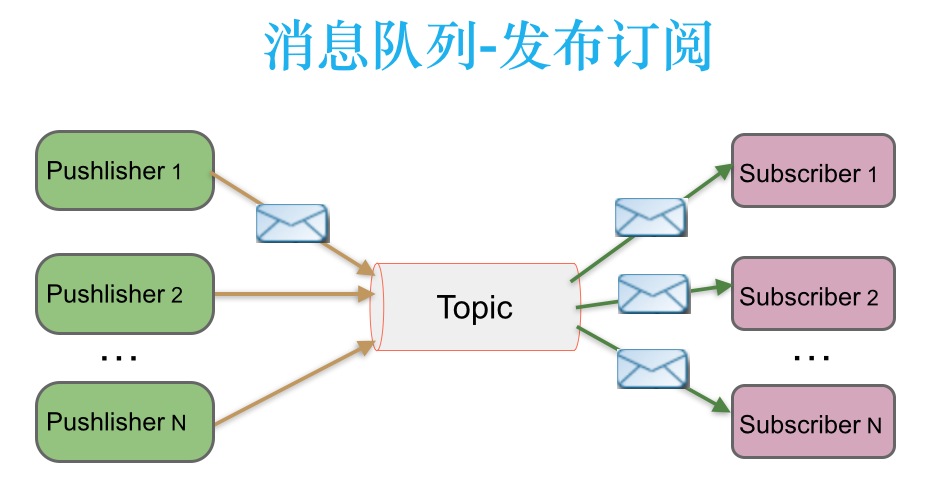
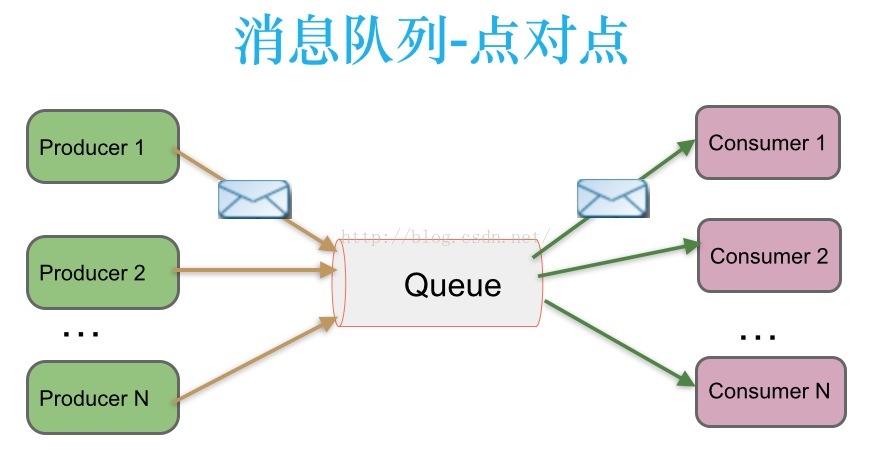
如上图所示，左侧是观察者模式，右侧是发布-订阅模式，那么他们最大的不同在哪呢？

1. 观察者模式中，发布者是知道订阅者有哪些的，然而，在发布-订阅模式中，发布者并不知道订阅者有哪些，它们只有通过消息代理进行通信。
2. 观察者模式大多数时候是同步执行的，比如当事件触发，Subject就会去遍历调用观察者的update方法(通知观察者)。而发布-订阅模式大多数时候是异步的（使用消息队列）。



传统的观察者模式并没有加入消息队列的机制，当有发布消息的需要时，被观察者会立即遍历通知所有观察者，并依次调用观察者的update方法，当这一切都完成后，才算发布完毕，这是极为耗时的操作，试想一下，如果观察者达到数万甚至数十万的量级，或者观察者的update方法耗时极长，那么每次的消息发布，我们都需要付出极大的代价。而发布-订阅模式加入了消息队列，发布者发送到topic的消息，只有订阅了topic的订阅者才会收到消息，并且所有订阅了topic的订阅者都会收到，这样一来，发布者只需要把要发布的消息发送到对应的topic队列中，发布就算完成了，订阅者会自行去topic中取出消息进行” 消费”。

### 1.1.3两种模式的比较



生产者发送一条消息到queue，一个queue可以有很多消费者，但是一个消息只能被一个消费者接受

当发布者发布一个消息，所有订阅这个topic的服务都能得到这个消息，所以从1到N个订阅者都能得到这个消息的拷贝。

### 1.1.4消费模式之推模式和拉模式

推模式(Push)： 即当生产者(Producer)发出的消息到达后，服务端马上将这条消息投递(push)给消费者(Consumer)；

拉模式(Pull)： 即服务端收到这条消息后什么也不做，只是等着消费者(Consumer)主动到自己这里来读，即消费者(Consumer)这里有一个“拉取”(Pull)的动作。

这两种模式都是消费端获取消息的方式，那么这两种方式有什么区别呢？这里根据几种场景进行分析：

**场景1：Producer 的速率大于 Consumer 的速率(生产>消费)**

对于 Producer 速率大于 Consumer 速率的情况，有两种可能性需要讨论，第一种是Producer 本身的效率就要比 Consumer 高(比如说：Consumer 端处理消息的业务逻辑可能很复杂，或者涉及到磁盘、网络等I/O操作)；另一种则是Consumer出现故障，导致短时间内无法消费或消费不畅。

Push模式：

在这种情况下，Push 方式由于无法得知当前 Consumer 的状态，所以只要有数据产生，便会不断地进行推送，在以上两种情况下时，可能会导致 Consumer 的负载进一步加重，甚至是崩溃。需要Consumer 有合适的反馈机制能够让服务端知道自己的状况。

Pull模式：

而采取 Pull 的方式问题就简单了许多，由于 Consumer 是主动到服务端拉取数据，当消费者跟不上生产者的效率时，只需要降低消费者的访问(pull)频率就好了。

**场景2：强调消息的实时性**

在一些强调实时性的场景中

Push模式：

采用 Push 的方式时，一旦消息到达，服务端即可马上将其推送给服务端，这种方式的实时性显然是非常好的

Pull模式：

采用 Pull 方式时，为了不给服务端造成压力（尤其是当数据量不足时，不停的轮询显得毫无意义），需要控制好自己轮询的间隔时间，但这必然会给实时性带来一定的影响。

**场景3：Pull 的长轮询**

Pull 模式有什么问题呢？由于主动权在消费方，消费方无法准确地决定何时去拉取最新的消息。如果一次 pull 取到消息了还可以继续去 pull，如果没有 pull 取到消息则需要等待一段时间再重新 pull。

但等待时间就很难判定了。你可能会说，我可以有xx动态拉取时间调整算法，但问题的本质在于，有没有消息到来这件事情决定权不在消费方。也许1分钟内连续来了1000条消息，然后半个小时没有新消息产生，可能你的算法算出下次最有可能到来的时间点是31分钟之后，或者60分钟之后，结果下条消息10分钟后到了，是不是很让人沮丧？

当然也不是说延迟就没有解决方案了，业界较成熟的做法是从短时间开始（不会对 CMQ broker 有太大负担），然后指数级增长等待。比如开始等5ms，然后10ms，然后20ms，然后40ms……直到有消息到来，然后再回到5ms。即使这样，依然存在延迟问题：假设40ms 到80ms 之间的50ms 消息到来，消息就延迟了30ms，而且对于半个小时来一次的消息，这些开销就是白白浪费的。

有一种优化的做法-长轮询，来平衡 pull/push 模型各自的缺点。基本方式是：消费者如果尝试拉取失败，不是直接 return，而是把连接挂在那里 wait，服务端如果有新的消息到来，把连接拉起，返回最新消息。

**场景4：部分或全部Consumer不在线**

在消息系统中，Producer 和 Consumer 是完全解耦的，Producer 发送消息时，并不要求Consumer 一定要在线，对于 Consumer 也是同样的道理，这也是消息通信区别于 RPC 通信的主要特点；但是对于 Consumer不在线的情况，却有很多值得讨论的场景。

首先，在 Consumer 偶然宕机或下线的情况下，Producer 的生产是可以不受影响的，Consumer 上线后，可以继续之前的消费，此时消息数据不会丢失；但是如果 Consumer 长期宕机或是由于机器故障无法再次启动，就会出现问题，即服务端需不需要为 Consumer 保留数据，以及保留多久的数据等等。

Push模式：

采用 Push 方式时，因为无法预知 Consumer 的宕机或下线是短暂的还是持久的，如果一直为该 Consumer 保留自宕机开始的所有历史消息，那么即便其他所有的 Consumer 都已经消费完成，数据也无法清理掉，随着时间的积累，队列的长度会越来越大，此时无论消息是暂存于内存还是持久化到磁盘上（采用 Push 模型的系统，一般都是将消息队列维护于内存中，以保证推送的性能和实时性），都将对服务端造成巨大压力，甚至可能影响到其他 Consumer 的正常消费，尤其当消息的生产速率非常快时更是如此；但是如果不保留数据，那么等该 Consumer 再次起来时，则要面对丢失数据的问题。

折中的方案是：给数据设定一个超时时间，当 Consumer 宕机时间超过这个阈值时，则清理数据；但这个时间阈值也并太容易确定。

Pull模式：

在采用 Pull 模型时，情况会有所改善；服务端不再关心 Consumer 的状态，而是采取“你来了我才服务”的方式，Consumer 是否能够及时消费数据，服务端不会做任何保证（也有超时清理时间）。

## 1.2消息队列的应用场景

### 1.2.1异步处理

用户注册后，需要发注册邮件(也可以是其他操作比如给用户发注册后的优惠券等)和注册短信。传统的做法有两种 1.串行的方式；2.并行方式：

1.串行方式：将注册信息写入[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端



2.并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间



上述对比，串行处理需耗时150ms，并行需耗时100ms，但是这样的执行方式，效率仍然低下，下面引入消息队列



由图可知，响应时间降低到55ms，大幅提升了系统的吞吐量

### 1.2.2流量削峰

秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用加入消息队列。

用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面，秒杀业务根据消息队列中的请求信息，再做后续处理

### 1.2.3应用解耦

用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口



假如库存系统无法访问，则订单减库存将失败，从而导致订单失败，传统方式还会导致订单系统与库存系统耦合

加入消息队列后



 订单系统：用户下单后，订单系统完成持久化处理后，将消息写入消息队列，返回用户订单下单成功

 库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

这样一来即使库存系统宕机，用户仍然可以进行下单操作，同时也实现订单系统与库存系统的应用解耦

当然这里只是举个不恰当的例子，上述处理方式很容易造成商品的超买超卖

### 1.2.4日志处理

在一些应用场景中，需要对日志进行大量采集和处理操作，比如能源管理系统，底层设备将本设备能源使用情况记录成日志，并将日志传输给上层处理，kafka更适用当前场景



日志采集客户端，负责日志数据采集，定时写入Kafka队列

Kafka消息队列，负责日志数据的接收，存储和转发

日志处理应用：订阅并消费kafka队列中的日志数据

上述所列只是消息队列的部分应用场景，仅供参考

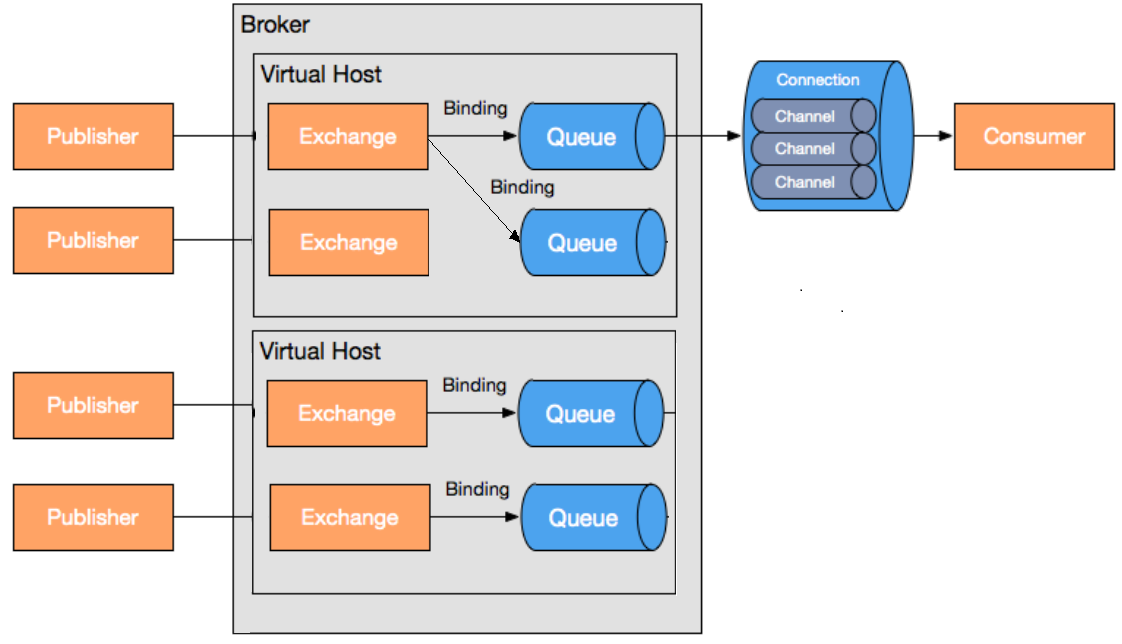
# RabbitMQ

Rabbit是一款开源的消息处理中间件，Rabbit是通过Erlang语言编写的，其基于AMQP协议

## 2.1 基础概念和使用

### 2.1.1 AMQP协议

AMQP（Advanced Message Queuing Protocol，高级消息队列协议）是一个进程间传递**异步消息**的**网络协议**



如上图：

1. **Publisher：**消息的生产者
2. **Broker：**翻译过来是代理。即接收和分发消息的应用，RabbitMQ Server就是Message Broker
3. **Virtual Host**：一个broker中可以有多个virtual host，每个VirtualHost相当于一个相对独立的RabbitMQ服务器，且host之间是相互隔离的，其中的exchange、queue、message不能互通。如果把broker比作mysql，那么virtual host就是mysql中的库。这样一来多个用户就可以各自使用各自的virtual host，做到互不干扰
4. **Exchange：**消息到达的第一站，一个exchange可以关联(binding)多个queue，消息到达后，匹配查询表中的（routing key）路由键（Fanout Exchange除外），分发消息到队列（queue）中去。Exchange有以下三种类型
5. **Dirct Exchange:** Direct类型的Exchange会将消息中的routing key与该Exchange关联的所有Binding中的binding key进行比较，如果相等，则发送到该Binding对应的Queue中。这是单播路由
6. **FanOut Exchange:** FanOut类型的Exchange会将消息发送给所有该 Exchange定义过Binding的Queues中去，这是一种广播行为。
7. **Topic Exchange：**Topic类型的Exchange会按照正则表达式，对routing key与binding key进行匹配，如果匹配成功，则发送到对应的Queue中。这是多播路由
8. **Binding：**exchange和queue之间的虚拟连接，binding中包含binding key。Binding信息被保存到exchange中的查询表中，用于消息的分发依据。
9. **Queue:** 消息最终到达队列中，等待消费者消费。Queue和Exchange是多对多的关系
10. **Connection:** 这是一条TCP连接，无论是生产者和消费者都是通过connection和broker进行通信的
11. **Channel：**channel是对TCP的多路复用。那么为什么有了Connection还需要Channel呢？那是因为Connection是TCP连接，而建立和销毁TCP连接是非常耗时的，通常来说，一个应用程序中常常有多个线程都需要从RabbitMQ中获取消息，如果仅通过Connection来通信，频繁地建立和销毁TCP连接，那么性能可想而知。而引入Channel之后，做到了TCP连接的复用。每个Channel都有一个ID，broker和client使用channel ID来确定方法对应的channel。因此实现channel之间的数据隔离。

数据的读写都是在channel中进行的。channel不能单独存在，仅存在connection上下文中。当connection关闭时，channel也会关闭

1. **Consumer：**消息的消费者
2. **Message：**message是消息的实体，分为properties和body两部分组成。Properties可以对消息进行修饰，比如优先级和延迟等，而body则是消息的内容

### 2.1.2安装和使用rabbitmq

#### 2.1.2.1 安装RabbitMQ

由于RabbitMQ基于Erlang语言，因此在安装rabbitMQ之前通常需要安装Erlang环境，这里就会涉及到Erlang的版本和RabbitMQ版本是否匹配的问题，可以在<https://www.rabbitmq.com/which-erlang.html>页面查看

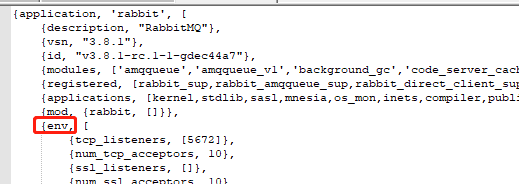
这里我们通过rpm进行安装

1. 首先下载Erlang，可以在这里下载erlang rpm包<https://pkgs.org/download/erlang>或<https://www.erlang-solutions.com/resources/download.html>
2. 通过**yum install 下载下来的erlang的rpm包名** 命令进行安装
3. 同理下载rabbitMQ <https://www.rabbitmq.com/install-rpm.html> 并安装

#### 2.1.2.2 配置RabbitMQ

安装完成后，我们需要配置rabbitMQ

1. 进入/usr/lib/rabbitmq/lib/rabbitmq\_server-3.8.1/ebin文件夹，打开rabbit.app文件
2. 关键的配置文件在env属性中，现在，我们需要修改loopback\_user属性为guest来允许远程连接：



这里介绍一些关键的配置：

1. **tcp\_listeners：**RabbitMQ使用的端口号
2. **loopback\_users**： 默认情况下，访客是禁止通过远程连接来访问rabbitMQ的，因此我们需要修改这一配置，这里需要将注释去掉



改为： {loopback\_users, [guest]},

#### 2.1.2.3 启动RabbitMQ

**rabbitmq-server start &** 启动RabbitMQ服务器

**rabbitmqctl start\_app** 启动RabbitMQ服务

**rabbitmqctl stop\_app** 关闭RabbitMQ服务

**rabbitmqctl status** 查看RabbitMQ服务状态

**rabbitmq-plugins enable rabbitmq\_management** 开启管理控制台插件

如何确认管理控制台开启了呢？

打开浏览器访问ip:15672，即可进入控制台，15672是控制台默认端口号，默认的账号密码都是guest

#### 2.1.2.4 命令行基础操作

这一节主要讲述如何操作RabbitMQ命令行和控制台

命令行主要有三大命令：rabbitmqctl(服务管理操作)、rabbitmq-plugins(插件管理操作)、rabbitmq-server(针对服务器本身的操作)

|  |  |
| --- | --- |
| **命令** | **涵义** |
| **rabbitmqctl start\_app** | 启动RabbitMQ服务 |
| **rabbitmqctl stop\_app** | 关闭RabbitMQ服务 |
| **rabbitmqctl status** | 查看RabbitMQ服务状态 |
| **rabbitmqctl add\_user 用户名 密码** | 新增账号，创建的用户需要设置权限 |
| **rabbitmqctl set\_user\_tags 用户名 权限** | 授权，如：administrator(超级管理员) |
| **rabbitmqctl list\_users** | 列出所有用户 |
| **rabbitmqctl delete\_user 用户名** | 删除用户 |
| **rabbitmqctl add\_vhost 主机名** | 创建虚拟主机 |
| **rabbitmqctl list\_vhosts** | 列出虚拟主机 |
| 队列和交换机同理 |  |
| **rabbitmqctl reset** | 移除rabbitmq上的所有数据，需要在**rabbitmqctl stop\_app**命令后执行 |
| **rabbitmqctl join\_cluster 集群 [--disc|--ram]** | 将本节点加入指定集群，在加入之前，该节点需要reset。  可选参数disc意味着本节点加入后数据将存储在磁盘，ram意味着本节点加入后数据存储在内存 |

### 2.1.3快速入门（整合spring）

首先创建一个spring项目，然后引入maven依赖

<dependency>

<groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit</artifactId>  
 <version>2.2.0.RELEASE</version>  
</dependency>

创建配置文件，配置rabbitmq的相关参数

//rabbitmq-config.properties

mq.host=10.1.170.152  
mq.username=DJ  
mq.password=2571838183  
mq.port=5672  
mq.vhost=DJHost

xml配置如下：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:rabbit="http://www.springframework.org/schema/rabbit"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  
 http://www.springframework.org/schema/rabbit  
 http://www.springframework.org/schema/rabbit/spring-rabbit-1.7.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context-4.2.xsd">

<!--引入properties文件-->

<context:property-placeholder location="classpath:configlocal/rabbit.properties"/>  
  
<!-- 连接配置 -->  
<rabbit:connection-factory id="connectionFactory" host="${mq.host}" username="${mq.username}" password="${mq.password}" port="${mq.port}" virtual-host="${mq.vhost}"/>  
<rabbit:admin connection-factory="connectionFactory"/>  
<!-- queue声明-->  
<rabbit:queue id="djTestQueue" name="djTestQueue" durable="true" auto-delete="false" exclusive="false" />  
<!--声明一个direct类型的exchange-->  
<rabbit:direct-exchange id="djExchange" name="djExchange" durable="true" auto-delete="false">  
 <rabbit:bindings>  
 <rabbit:binding queue="djTestQueue" key="toDjQueue"/>  
 </rabbit:bindings>  
</rabbit:direct-exchange>  
<!-- spring template声明-->  
<rabbit:template exchange="djExchange" queue="djTestQueue" id="amqpTemplate" connection-factory="connectionFactory" message-converter="jsonMessageConverter" />  
  
<!-- 配置监听器-->  
<rabbit:listener-container connection-factory="connectionFactory" acknowledge="auto">  
 <rabbit:listener queues="djTestQueue" ref="djConsumer"/>  
</rabbit:listener-container>  
  
<!-- 消息对象json转换类 -->  
<bean id="jsonMessageConverter" class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter" />

编写消费者和生产者

@Service("djConsumer")

public class Consumer implements MessageListener {  
 @Override  
 public void onMessage(Message message) {  
 try{  
 System.*out*.print("收到消息！"+message.toString());  
 }catch(Exception e){  
 e.printStackTrace();  
 }  
 }  
}

@Service  
public class Producer {  
  
 @Resource  
 private AmqpTemplate amqpTemplate;  
 private final static Logger *LOGGER* = LoggerFactory.*getLogger*(Producer.class);  
  
 */\*\*  
 \* 发送消息  
 \** ***@param*** *queueKey  
 \** ***@param*** *object  
 \*/* public void sendDataToQueue(String queueKey, Object object) {  
 try {  
 amqpTemplate.convertAndSend(queueKey, object);  
 } catch (Exception e) {  
 *LOGGER*.error("sendDataToQueue has an error：",e);  
 }  
  
 }  
}

### 2.1.4Exchange交换机

在AMQP模型中，交换机是一个很重要的概念，它负责接收消息，并根据Exchange的类型和routing key来转发消息到对应的队列中

#### 2.1.4.1 常用属性

交换机常用属性如下：

|  |  |
| --- | --- |
| **属性名** | **描述** |
| **name** | 交换机的标识名称 |
| **type** | 交换机的类型，常用的主要有direct、fanout、topic |
| **durability** | 是否需要持久化，true为持久化。如果要真正做到持久化，需要Exchange、Queue和Message同时设置持久化 |
| **auto delete** | 当最后一个绑定到Exchange的队列被删除后，自动删除Exchange。即没有队列绑定后自动删除 |
| **interval** | 当前Exchange是否用于rabbitMQ内部使用，多数情况下用不到，默认false |

#### 2.1.4.2 交换机的类型

##### 2.1.4.2.1 direct直连交换机

direct是rabbitMQ默认的交换机类型，如果消息的routing key和交换机绑定的某个binding key字符完全匹配的话，就将消息投递到对应的queue中去。是一对一模式

在spring配置文件中声明一个direct类型的交换机的配置如下：

<rabbit:direct-exchange id="directExchange" name="helloExchange" durable="true" auto-delete="false">

<rabbit:bindings>  
 <rabbit:binding queue="djTestQueue" key="toDjQueue"/>  
 </rabbit:bindings>  
</rabbit:direct-exchange>

在spring文件中添加上述配置之后，将自动创建一个name为**helloExchange**的direct类型交换机。并在交换机上添加一个binding key为**toDjQueue**的绑定，将交换机与名为**djTestQueue**的queue绑定起来

需要注意的是，rabbitmq在配置启动后会有一个direct类型的default Exchange，default Exchange隐式地绑定到每个队列，其binding key等于queue name。无法显式绑定到默认交换器或从默认交换器取消绑定。default Exchange也不能被删除。该default Exchange的name是空字符串

##### 2.1.4.2.2 topic匹配/订阅模式

Topic类型的交换机会按照正则表达式尝试让routing key匹配binding。也就是说，topicExchange每次收到一条消息，都会去尝试以通配符的形式匹配所有的queue，一旦有一个或多个queue匹配上，就将消息分别投递给这些queue，也就是说每个匹配的queue都会收到一条消息。是一对多模式

|  |  |
| --- | --- |
| **通配符** | **涵义** |
| **#** | 匹配一个或者多个单词。单词是以.分隔的 |
| **\*** | 匹配且只能匹配一个单词。单词是以.分隔的 |

<!-- 配置topic交换机，这里配置了三个路由规则，分别连接三个队列-->

<rabbit:topic-exchange id="topicExchange" name="DJtopicExchange" durable="true" auto-delete="false">

<rabbit:bindings>  
 <rabbit:binding pattern="topic.#" queue="topicQueue"/>  
 <rabbit:binding pattern="#.message" queue="messageQueue"/>  
 <rabbit:binding pattern="#.cn" queue="cnQueue"/>  
 </rabbit:bindings>  
</rabbit:topic-exchange>

<rabbit:template exchange="topicExchange" id="topicTemplate" connection-factory="connectionFactory" message-converter="jsonMessageConverter" />  
  
<!-- 配置监听器，这里一共配置了4个消费者分别监听4个队列-->  
<rabbit:listener-container connection-factory="connectionFactory" acknowledge="auto">  
 <rabbit:listener queues="djTestQueue" ref="djConsumer"/>  
 <rabbit:listener queues="messageQueue" ref="messageConsumer"/>  
 <rabbit:listener queues="topicQueue" ref="topicConsumer"/>  
 <rabbit:listener queues="cnQueue" ref="cnConsumer"/>  
</rabbit:listener-container>

##### 2.1.4.2.3fanout发布/订阅模式

Fanout不会管生产者传递过来的routing key，而是直接将消息向自己绑定的每一个queue上进行广播

<!—fanout交换机将会向绑定的这三个队列进行广播-->

<rabbit:fanout-exchange id="fanoutExchange" name="DJFanoutExchange" durable="true" auto-delete="false">

<rabbit:bindings>  
 <rabbit:binding queue="topicQueue"/>  
 <rabbit:binding queue="messageQueue"/>  
 <rabbit:binding queue="cnQueue"/>  
 </rabbit:bindings>  
</rabbit:fanout-exchange>

<rabbit:template exchange="fanoutExchange" id="fanoutTemplate" connection-factory="connectionFactory" message-converter="jsonMessageConverter" />

<!-- 配置监听器，监听4个队列-->  
<rabbit:listener-container connection-factory="connectionFactory" acknowledge="auto">  
 <rabbit:listener queues="djTestQueue" ref="djConsumer"/>  
 <rabbit:listener queues="messageQueue" ref="messageConsumer"/>  
 <rabbit:listener queues="topicQueue" ref="topicConsumer"/>  
 <rabbit:listener queues="cnQueue" ref="cnConsumer"/>  
</rabbit:listener-container>

### 2.1.5Queue队列

#### 2.1.5.1 常用属性

消息队列是实际存储消息的组件，其常用属性有：

|  |  |
| --- | --- |
| **属性** | **描述** |
| **durable** | 是否持久化到。如果不持久化，那么在服务器宕机或重启之后Queue就会丢失。如果要真正做到持久化，需要Exchange、Queue和Message同时设置持久化 |
| **auto-delete** | 是否自动删除。当最后一个消费者不再监听queue后，自动删除queue。即队列没有消费者后自动删除。一般选择false |
| **exclusive** | 是否设置为排他队列。对于排他队列，只对首次声明它的连接（Connection）有权访问，连接断开后，排他队列将自动删除。很少用，一般设为false |

Xml配置如下：

<rabbit:queue id="topicQueue" name="topicQueue" durable="true" auto-delete="false" exclusive="false" />

#### 2.1.5.2 Binding绑定

Binding是queue队列和Exchange之间的绑定关系。binding还有一个作用，它还可以在Exchange之间进行绑定。

Direct Exchange，Topic Exchange进行Binding的时候，需要指定binding key。而Fanout Exchange则不需要

### 2.1.6Message消息

Message是服务器和应用程序之间传递的数据，主要由Properties和Playload(即body)组成，其常用属性如下。Message中的所有属性都需要生产者自己设置才会有，否则都是null：

|  |  |
| --- | --- |
| **属性** | **描述** |
| **delivery mode** | 是否持久化。果未设置持久化，转发到queue中并未消费则重启服务或者服务宕机则消息丢失。如果要真正做到持久化，需要Exchange、Queue和Message同时设置持久化 |
| **headers** | 头信息，是由一个或多个键值对组成的，当固定的Properties不满足我们需要的时候，可以自己扩展。 |
| **Expiration** | 过期时间，单位毫秒，超过消息的过期时间后，消息将从队列中移除。如果队列也设置了过期时间，则二者取最小的那个 |

在**org.springframework.amqp.core**包中提供了**Message**类供我们使用。

Eg：

new Message("Hello,topic!".getBytes(),MessagePropertiesBuilder.*newInstance*().setExpiration("10000").build())

## 2.2 高级特性

### 2.2.1 生产者的可靠性投递

简单地来说，就是保障消息100%能够投递成功。那么要做到这一点，需要完成哪些步骤呢？

1. 保证消息的成功发出
2. 保证MQ节点的成功接收
3. 生产者收到MQ节点的ack应答
4. 对投递失败的消息进行补偿

业界对于消息的可靠性投递主要有如下两种解决方案：

1. **消息落库，**并记录消息投递过程中各阶段的状态
2. **延迟投递，**做二次确认，回调检查

#### 2.2.1.1 消息落库方案



1. 定时任务将抽取指定时间内未收到应答的状态为0的消息，判断其重试次数是否为3次，如果已重试3次，则将其状态改为2发送失败，状态为2的消息就需要人工干预了
2. 该方案的消息不会丢失，只会重复，因此**消费端需要做幂等性处理**
3. 生产者在发送消息后需要通过confirmListener监听应答，并通过回调函数将消息的状态修改为1已发送
4. 由于消息需要落库，因此这种方案在并发量过大时受数据库性能制约

代码示例如下：

public class CustomConfirmAndReturnCallback implements RabbitTemplate.ConfirmCallback {

private static final Logger *logger* = LoggerFactory.getLogger(CustomConfirmAndReturnCallback.class);  
  
 @Autowired  
 private RabbitTemplate rabbitTemplate;  
 @Autowired  
 private BrokerMessageLogMapper brokerMessageLogMapper; @PostConstruct  
 public void init() {  
 //配置rabbitTemplate的bean对象。指定 ConfirmCallback回调函数  
 rabbitTemplate.setConfirmCallback(this);  
 }  
  
 */\*\*  
 \* 实现RabbitTemplate.ConfirmCallback接口中的方法*

*\* 如果消息没有到达交换机,则该方法中isSendSuccess = false,error为错误信息;  
 \* 如果消息正确到达交换机,则该方法中isSendSuccess = true;  
 \*/* @Override  
 public void confirm(CorrelationData correlationData, boolean isSendSuccess, String error) {  
 String messageId = correlationData.getId();  
 if (isSendSuccess) {  
 //如果消息到达MQ Broker，更新消息  
 brokerMessageLogMapper.changeBrokerMessageLogStatus(messageId, Constants.ORDER\_SEND\_SUCCESS, new Date());  
 } else {

//由定时任务决定是否重发  
 *logger*.error("消息发送异常...");  
 }  
 }  
  
}

#### 2.2.1.2 延迟投递方案



相比第一种消息落库方案，该方案减少了一次消息落库，两个DB实现解耦，提高了并发性能，因此这种方案对于高并发量更加友好，但是相比于第一种方案，这种方案丢失消息的可能性更大，比如因网络异常丢失了第10步，那么消息就真的丢失了。

若消费端对队列1中数据消费成功，则回调服务监听到消费端的确认消费消息，会将这条消息持久化到消息DB中。当回调服务监听到队列2的延迟消息时，检查消息 DB是否存在记录。有记录则不处理，无记录则回调服务发送一条ReSend命令给生产端，执行第一步重复操作

**消费者的消费操作需要实现幂等性**

#### 2.2.1.3 幂等性处理

上述两种方案各有优缺点，但都离不开消费端的幂等性处理

下面介绍几种最常见的幂等性解决方案：

##### 2.2.1.3.1 唯一ID+指纹码机制

这个方案比较简单，就是取一个全局唯一的业务id，如订单ID，消费之后将其入库。每次消费之前，都尝试从数据库中取出该记录，若取不出，则说明消息未消费，那就消费后再入库，若取出了则说明该消息已经消费过了，直接返回即可。

这种方法最大的优势是简单，其缺点也很明显，主要有以下两点：

1. **消费前尝试读取数据，消费后将消息入库。这加大了数据库的压力，且高并发时吞吐量下降**
2. **非严格幂等。在并发较大时，假设同时来了两个业务涵义相同的消息，同时尝试读取消息记录，发现都没读到，会出现两个消息同时消费的情况。**

那么是不是可以通过java语法中的Lock加锁来解决这种并发问题呢？答案是否定的，因为并发量较大时，通常都是有多台服务器负载均衡的，而加锁只能控制一台服务器上的并发。如果一定要加锁的话，可以通过加分布式锁来处理，但这就意味着无论部署多少台消费者，同一时刻都只会有一个消费者进行消费，会进一步降低消费者的性能

也可以通过数据库加悲观锁来处理，在读取的时候加排它锁，在事务提交时释放锁，这样就可以保证数据一致了，同样的，还是会有并发问题

还可以通过乐观锁，类似cas的机制，如修改订单时，update t\_order set version = 查询到的version+1 where order\_id = trade\_no and version =查询到的version;这种方案适合并发不大时处理，如果并发较大，update操作还是会一直空转

**在并发量较大时还是尽量避免这种方案**

##### 2.2.1.3.2 redis原子性实现幂等

Redis具有单线程的特点，因此，无论是多少并发操作，redis都会串行执行，我们可以利用这个特点实现分布式锁，通过分布式锁来解决幂等性问题，并且由于redis的高性能，因此其并发能力也很强

需要注意以下几点：

1. redis操作时要设置过期时间，且加锁和设置时间都必须是一步完成(原子操作)，推荐setEx和set nx px操作。如果不设置过期时间，假设加锁之后程序崩溃，那么就永远无法解锁了
2. 必须保证只有锁的持有者才可以释放分布式锁

eg：

public boolean timeTaskLock(String keyString, int time, TimeUnit unit) {  
 if (time < 1 || StringUtils.*isEmpty*(keyString) || null == unit || null == jedisCluster) {  
 throw new IllegalArgumentException("定时任务加锁时参数异常");  
 }  
 long millis = unit.toMillis(time);  
 //原子操作，并且redis单线程，不会有并发问题  
 String setResult = jedisCluster.set(keyString, ipUtil.getThisServerHostIp(), "NX", "PX", millis);  
 return "ok".equalsIgnoreCase(setResult);  
}  
public boolean timeTaskUnLock(String keyString) {  
 if (StringUtils.*isEmpty*(keyString) || null == jedisCluster) {  
 throw new IllegalArgumentException("定时任务释放锁时参数异常");  
 }  
 String lockOwerIp = jedisCluster.get(keyString);  
 if (ipUtil.getThisServerHostIp().equals(lockOwerIp)) {  
 return "1".equals(String.*valueOf*(jedisCluster.del(keyString)));  
 }  
 return false;  
}

### 2.2.2 生产端confirm确认机制

在生产者投递消息之后，若broker收到消息，则会给生产者返回一个ack应答，生产者通过ack应答来判断消息是否正常发送

### 2.2.3 生产端return消息机制

Return Listener用于处理一些不可路由的消息。简单来说，就是在某些情况下，我们发送的消息无法路由(如exchange不存在、routing key不存在等)，这时候，为了监听这些无法路由的消息，我们就需要Return Listener。Return Listener是配置在生产者端的

Return Listener中有一个Mandatory的关键配置(在**<rabbit:template>**中配置)，如果该配置设置为true，则监听器会接受无法路由的消息并进行后续处理，如果配置为false，则broker会自动删除不可路由的消息！该配置默认为false

### 2.2.4 消费端限流

假设一个场景下，broker中有大量未处理的消息，而我们只有少量的消费者，我们的消费者根本无法在短时间内处理如此多的消息，为了防止消费端崩溃，我们就需要对消费端限流。

Rabbitmq提供了qos(服务质量保证)功能，这种功能在关闭autoAck的前提下，如果一定数量的消息未被确认前，消费者不会进行新消息的消费

以下为常用属性，是在消费者中配置的：

**prefetchSize**：设置消息的大小限制，一般设置为0，即不限制

**prefetchCount**：告诉Broker不要一次给一个消费者推送太多消息，一旦有这么多个消息还没有ack，则停止推送

**global**：true/false将指定上述限流是否应用于channel，true是应用于channel级别，false是应用于整个consumer级别

请注意！如果设置了自动应答，则限流不会生效

### 2.2.5 消费端ack机制

消费端的消息签收方式分为两种：手动签收和自动签收，而手动签收又分为手动Ack和手动Nack。一般来说，我们不会开启自动签收，必须要进行手动签收。

1. 自动签收：一旦消息被consumer接收到，消费者立刻发送ack
2. 手动签收：消息接收后不会自动发送ack，需要消费者自己调用
3. ack：表示消息已消费成功
4. Nack：表示消息消费失败，broker将重新发送消息

对于手动Nack的消息，消息将重新回到broker中，这被称作消息的重回队列，**一般来说重回队列会被关闭**，这是因为重回队列的消息依然有很大概率会处理失败

### 2.2.6 TTL过期时间

TTL(Time To Live)，即生存时间

Rabbitmq支持在消息级别、队列级别设置消息的过期时间，消息级别设置的过期时间在消息发送时可以指定，队列的过期时间是从消息入队时开始计算，只要超过了队列超时时间的配置，那么消息将自动清除

### 2.2.7 死信队列

DLX(Dead-Letter-Exchange)死信队列。当消息在一个队列中变成死信之后，它会被重新publish到另一个Exchange中，这个Exchange就是DLX死信队列。下面几种情况，消息将变成死信：

1. **消息被拒绝(basic.reject/basic.nack)并且requeue=false(即不需要重回队列)**
2. **TTL消息过期**
3. **消息队列达到最大长度后再到来的消息**

DLX也是一个正常的Exchange，可以对它进行任何Exchange的操作，当有消息变为死信之后，死信就会被发布到DLX中去。死信队列需要我们自己去设置

1. 配置一个exchange和一个queue，将二者的binding key设置为#，即不管任何routingkey都会被路由到队列中
2. 在正常使用的所有queue中，配置参数x-dead-letter-exchange，指定为我们刚刚设置的exchange

死信队列常用于补偿操作，在日常开发中非常重要

## 2.3 spring AMQP

SpringAMPQ是spring对于rabbitmq的一些整合封装

需要引入依赖

<dependency>

<groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit</artifactId>  
 <version>2.2.0.RELEASE</version>  
</dependency>

### 2.3.1 RabbitAdmin

RabbitAdmin类可以很方便的操作RabbitMQ，它可以帮助我们用于创建、绑定、管理队列与交换机。我们可以在xml中配置该类的bean

<rabbit:connection-factory id="connectionFactory" host="${mq.host}" username="${mq.username}" password="${mq.password}" port="${mq.port}" virtual-host="${mq.vhost}"/>  
<rabbit:admin connection-factory="connectionFactory" auto-startup="true"/>

**auto-startup**的值一定要设置为true，这样spring容器加载的时候才会加载这个bean，其默认值就是true，这里写出来是为了强调

使用时将rabbitAdmin对象注入即可，eg：

//删除交换机  
rabbitAdmin.deleteExchange("djExchange");  
// 声明一个交换机  
rabbitAdmin.declareExchange(new DirectExchange("admin\_exchange", false, false));  
// 声明一个队列  
rabbitAdmin.declareQueue(new Queue("admin\_queue", false, false, false));  
// 声明一个绑定，**由于binding可以将Exchange绑到Exchange上，因此这里还要提供一个参数DestinationType**  
rabbitAdmin.declareBinding(new Binding("admin\_queue", Binding.DestinationType.*QUEUE*, "admin\_exchange", "key", null));

### 2.3.2 springAMQP的声明

我们可以在xml中声明Exchange、queue、binding，eg：

<!-- queue声明-->  
<rabbit:queue id="djTestQueue" name="djTestQueue" durable="true" auto-delete="false" exclusive="false" />  
<!--声明一个direct类型的exchange-->  
<rabbit:direct-exchange id="djExchange" name="djExchange" durable="true" auto-delete="false">  
 <rabbit:bindings>  
 <rabbit:binding queue="djTestQueue" key="toDjQueue"/>  
 </rabbit:bindings>  
</rabbit:direct-exchange>

### 2.3.3 rabbitTemplate消息模板

rabbitTemplate是我们进行消息发送时的关键类，该类提供了丰富的消息发送方法，包括回调监听消息接口ConfirmCallback、返回值确认接口ReturnCallback等。同样我们需要将其注入spring容器后使用

<rabbit:template id="rabbitTemplate" connection-factory="connectionFactory"/>

@Resource  
private RabbitTemplate rabbitTemplate;

我们也可以给rabbitTempla指定默认的Exchange、queue和routingkey，当发送消息时未指定这三者中的任意一个，就使用默认值

常用方法：

***//设置confirm确认机制的回调函数***

void setConfirmCallback(ConfirmCallback confirmCallback)

***//设置return机制的回调函数***

void setReturnCallback(ReturnCallback returnCallback)

***//发送消息，拥有多个重载方法***

void send(final String exchange, final String routingKey, final Message message)

***//转换并发送消息，拥有多个重载方法，MessagePostProcessor在消息转换完成后、发送之前添加/修改消息的属性。它还可以用于修收监听器收到的消息***

void convertAndSend(String exchange, String routingKey, final Object message,final MessagePostProcessor messagePostProcessor)

### 2.3.4 MessageListenerContainer消息监听容器

消息监听容器是为消费端准备的一个类，通过这个类我们可以做到：

1. 监听队列（或者多个队列）、自动启动、自动声明
2. 设置事务特性、事务回滚、事务属性、事务容量（并发）、是否开启事务、回滚事务等！
3. 设置消费者数量、最小最大数量、批量消费
4. 设置消费端消息ack模式、是否重回队列、异常捕获handler函数。
5. 设置消费者标签生成策略、是否独占模式、消费者属性等

Xml配置如下：

<rabbit:listener-container connection-factory="connectionFactory" acknowledge="manual" requeue-rejected="false" max-concurrency="3">   
 <rabbit:listener queues="djTestQueue" ref="djConsumer"/>  
</rabbit:listener-container>

**acknowledge**属性是设置其ack的确认模式，**manual**意味着将手动ack确认。**requeue-rejected**属性确定是否允许重回队列

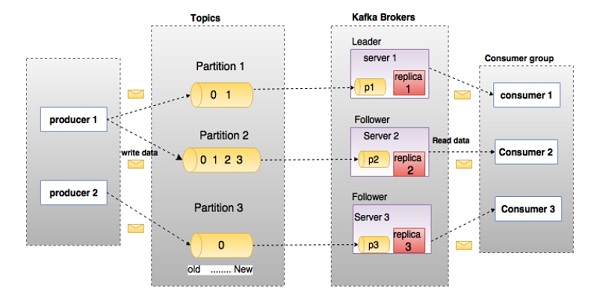
### 2.3.5 MessageConverter消息转换器

一般来说，消息体是通过二进制的格式进行传输的，如果在

# Kafka

## 1.3kafka相关概念

Kafka专为分布式高吞吐量的系统设计，相比于其他消息队列，Kafka非常快，并保证零停机和零数据丢失。



**主题(Topics)：**同一类消息被归纳进一个topic中。生产者发送消息到指定的Topic下，消费者从这个Topic下消费消息。

**消息分区(Partition)：**一个Topic下面会分为很多分区，它是topic在物理上的分组。 每每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id(offset)。

**复制因子(replication-factor)：**为了提高消息的可靠性，Kafka每个topic的partition有N个副本（replicas），其中N(大于等于1)是topic的复制因子的个数，实际上就是为partition中的消息指定副本个数。

Partition（分区）

主题可能有许多分区，因此它可以处理任意数量的数据。

3

Partition offset（分区偏移）

每个分区消息具有称为" offset "的唯一序列标识。

4

Replicas of partition（分区备份）

副本只是一个分区的"备份"。 副本从不读取或写入数据。 它们用于防止数据丢失。

5

Brokers（经纪人）

代理是负责维护发布数据的简单系统。 每个代理中的每个主题可以具有零个或多个分区。 假设，如果在一个主题和N个代理中有N个分区，每个代理将有一个分区。

假设在一个主题中有N个分区并且多于N个代理(n + m)，则第一个N代理将具有一个分区，并且下一个M代理将不具有用于该特定主题的任何分区。

假设在一个主题中有N个分区并且小于N个代理(n-m)，每个代理将在它们之间具有一个或多个分区共享。 由于代理之间的负载分布不相等，不推荐使用此方案。

6

Kafka Cluster（Kafka集群）

Kafka有多个代理被称为Kafka集群。 可以扩展Kafka集群，无需停机。 这些集群用于管理消息数据的持久性和复制。

7

Producers（生产者）

生产者是发送给一个或多个Kafka主题的消息的发布者。 生产者向Kafka经纪人发送数据。 每当生产者将消息发布给代理时，代理只需将消息附加到最后一个段文件。 实际上，该消息将被附加到分区。 生产者还可以向他们选择的分区发送消息。

8

Consumers（消费者）

Consumers从经纪人处读取数据。 消费者订阅一个或多个主题，并通过从代理中提取数据来使用已发布的消息。

Leader（领导者）

" Leader "是负责给定分区的所有读取和写入的节点。 每个分区都有一个服务器充当Leader

。

10

Follower（追随者）

跟随领导者指令的节点被称为Follower。 如果领导失败，一个追随者将自动成为新的领导者。 跟随者作为正常消费者，拉取消息并更新其自己的数据存储。