# 1、rabbitmq的安装与集群搭建

## 1、1 rabbitmq安装

\*\*\*\*\*具体安装3.6.5的文档:https://www.cnblogs.com/sky-cheng/p/1 07091 04.html  
\*\*\*\*\*\*卸载文档:https://www.cnblogs.com/kingsonfu/p/1 1 023967.html  
插件地址:https://www.rabbitmq.com/community-plugins.html

延时插件地址

:https://dl.bintray.com/rabbitmq/communityplugins/3.6.x/rabbitmq\_delayed\_message\_exchange/  
下载延时插件:

Wget https://dl.bintray.com/rabbitmq/communityplugins/3.6.x/rabbitmq\_delayed\_message\_exchange/rabbitmq\_delayed\_message\_exchange-201 71 21 5-3.6.x.zip

### 1.1.1 安装

1、安装rabbitmq所需的依赖包

yum install build-essential openssl openssl-devel unixODBC unixODBC-devel make gcc gccc++ kernel-devel m4 ncurses-devel tk tc xz

2、下载安装包（路径参考usr/local/software）

wget www.rabbitmq.com/releases/erlang/erlang-1 8.3-1 .el7.centos.x86\_64.rpm  
wget http://repo.iotti.biz/CentOS/7/x86\_64/socat-1 .7.3.2-5.el7.lux.x86\_64.rpm  
wget [www.rabbitmq.com/releases/rabbitmq-server/v3.6.5/rabbitmq-server-3.6.5-1 .noarch.rpm](http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.5/rabbitmq-server-3.6.5-1%20.noarch.rpm)

3、安装服务命令

**第一步：安装erlang语言环境**

rpm -ivh erlang-1 8.3-1 .el7.centos.x86\_64.rpm

**第二步：安装socat加解密软件**

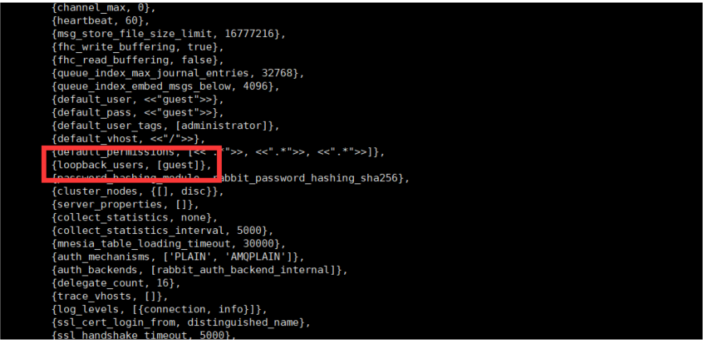
rpm -ivh socat-1 .7.3.2-5.el7.lux.x86\_64.rpm

**第三步：安装rabbitmq**

rpm -ivh rabbitmq-server-3.6.5-1 .noarch.rpm

**第四步：修改集群用户与连接心跳检测**

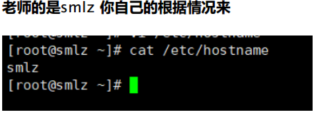
注意修改vim /usr/lib/rabbitmq/lib/rabbitmq\_server-3.6.5/ebin/rabbit.app文件  
修改： loopback\_users 中的 <<"guest">>,只保留guest（不修改只能通过localhost访问）



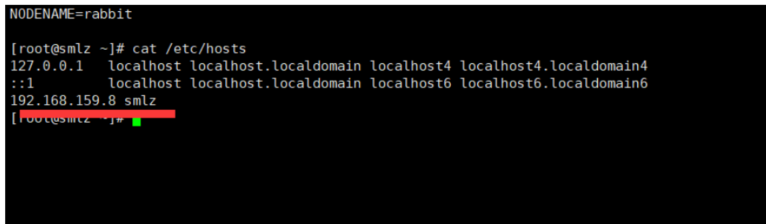
**第五步：修改本机系统文件**

a:修改 vim /etc/rabbitmq/rabbitmq-env.conf  
添加: NODENAME=rabbit

b:修改 vim /etc/hostname

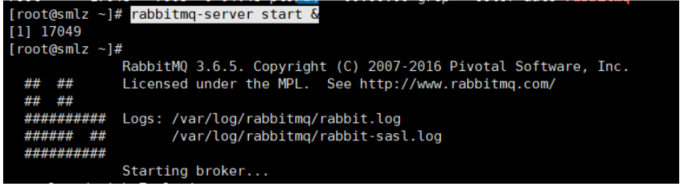


c:修改本地 vim /etc/hosts文件

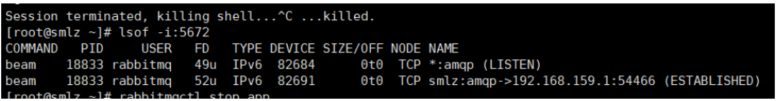


**第六步：验证服务器是否可用**

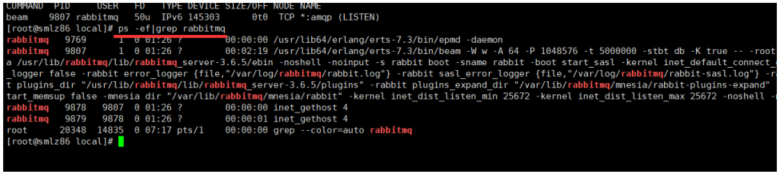
rabbitmq-server start &  
执行管控台插件:(不然不能在浏览器访问)  
rabbitmq-plugins enable rabbitmq\_management



**检查端口：lsof –i:5672**



通过 ps -ef| grep rabbitmq



### 1.1.2 rabbitmq的基本操作命令

测试连接： http://ip:1 5672(来访问) 用户名密码 guest/guest

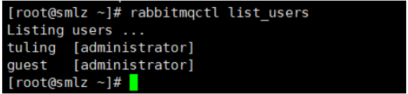
**1、起停服务命令**

启动服务rabbitmqctl start\_app(启动rabbitmq节点需要保证erlang虚拟机节点起来才能执行)

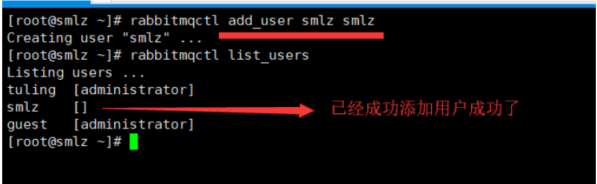
停止服务 rabbitmqctl stop\_app(停止rabbitmq节点，但是不会停止erlang节点)，rabbitmqctl stop（都会停止）

**2、用户操作命令**

查看所有用户列表：rabbitmq list\_users



添加用户：rabbitmqctl add\_user kevin kevin



设置用户角色（超级管理员）：rabbitmqctl set\_user\_tags kevin administrator

1、超级管理员(administrator)

可登陆管理控制台，可查看所有的信息，并且可以对用户，策略(policy)进行操作。

2、监控者(monitoring)

可登陆管理控制台，同时可以查看rabbitmq节点的相关信息(进程数，内存使用情况，磁盘使用情况等)

3、策略制定者(policymaker)

可登陆管理控制台, 同时可以对policy进行管理。但无法查看节点的相关信息(上图红框标识的部分)。

4、普通管理者(management)

仅可登陆管理控制台，无法看到节点信息，也无法对策略进行管理。

5、其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

**为用户设置权限（所有权限）**：rabbitmqctl set\_permissions -p / smlz ".\*" ".\*" ".\*"

rabbitmqctl set\_permissions -p <虚拟机> <用户名> ".\*" ".\*" ".\*"

列出用户权限：rabbitmqctl list\_user\_permissions kevin

清除用户权限：rabbitmqctl clear\_permissions -p <虚拟机> <用户名>

删除用户 rabbitmqctl delete\_user root  
修改密码 rabbitmqctl change\_password 用户名 新密码

3、虚拟主机命令

rabbitmqctl add\_vhost /cloudmall 增加一个虚拟主机  
rabbitmqctl list\_vhosts; 查看所有的虚拟主机  
rabbitmqctl list\_permissions -p /cloudmall 查看虚拟主机的权限  
rabbitmqctl delete\_vhost /cloudmall 删除虚拟主机

4、操作队列命令

rabbitmqctl list\_queues 查询所有队列  
rabbitmqctl -p vhostpath purge\_queue blue 清除队列消息

5、高级命令

rabbitmqctl reset 移除所有数据 该命令需要在 rabbitmqctl stop\_app命令之后才执行(也就是说 在服  
务停止后)  
rabbitmqctl join\_cluster <cluster\_node> [--ram] 组成集群命令  
rabbitmqctl cluster\_status 查看集群状态  
rabbitmqctl change\_cluster\_node\_type dist| ram 修改集群节点存储数据模式  
rabbitmqctl forget\_cluster\_node [--offline]忘记节点 （摘除节点）  
rabbitmqctc rename\_cluster\_node oldnode1 newnode1 oldnode2 newnode2 修改节点名称

## 1、2 rabbitmq集群搭建

### 1.2.1 集群搭建

1、按照单机安装的步骤在不同的机器上安装多台rabbitmq

2、从多个节点中选择一台机器作为主节点，然后把主节点的cookie文件同步到其他节点上去，进入var/lib/rabbitmq目录下，把var/lib/rabbitmq/erlang.cookie文件的权限修改为777（chmod 777 /var/lib/rabbitmq/.erlang.cookie，修改之后才能对文件进行复制操作）,原来是400；然后把.erlang.cookie文件copy到各个节点下；最后把所有cookie文件权限改为400即可

scp /var/lib/rabbitmq/.erlang.cookie 1 92.1 68.1 59.87:/var/lib/rabbitmq  
scp /var/lib/rabbitmq/.erlang.cookie 1 92.1 68.1 59.88:/var/lib/rabbitmq

3、启动集群

首先执行停止命令 ：rabbitmqctl stop

b:然后在三台服务器上86,87,88上执行 下面的命令(启动集群命令)

rabbitmq-server -detached  
c:切换到87的机器上执行下面三条命令（同样目录都是再/usr/local下执行的）

rabbitmqctl stop\_app

rabbitmqctl join\_cluster rabbit@smlz86

rabbitmqctl start\_app  
D:切换到88的机器上执行下面三条命令（同样目录都是再/usr/local下执行的）

rabbitmqctl stop\_app

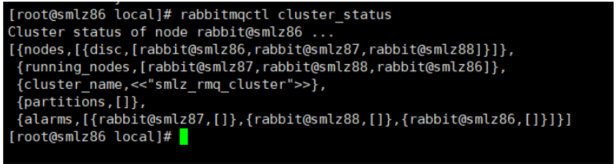
rabbitmqctl join\_cluster rabbit@smlz86

rabbitmqctl start\_app

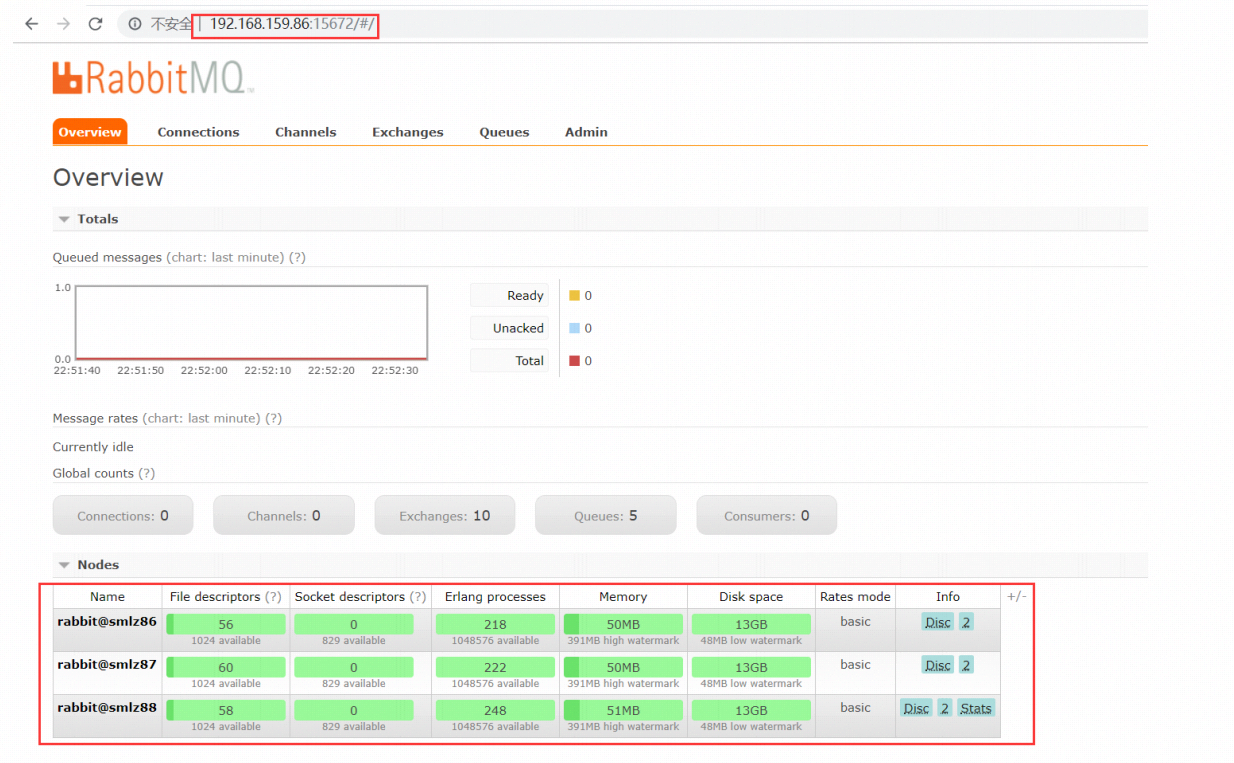
E:修改集群名称: 在86上执行给命令( /usr/local)

rabbitmqctl set\_cluster\_name rabbitmq\_cluster\_smlz

F:查看集群状态 rabbitmqctl cluster\_status



G:三台服务随意访问地址  
访问任意一个管控台节点： http://1 92.1 68.1 59.86:1 5672 这里我访问的是86节点



### 1.2.2 配置镜像队列

**在任意节点上执行 rabbitmqctl set\_policy ha-all "^" '{"ha-mode":"all"}'即可**

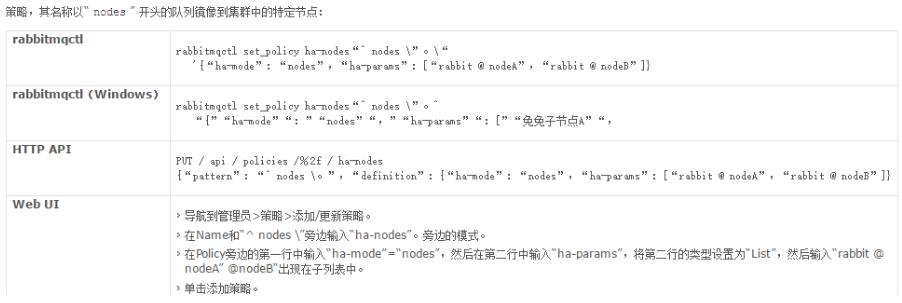
剔除节点命令  
PS:rabbitmqctl forget\_cluster\_node rabbit@节点名称(比如smlz87 smlz88) 剔除节点

ha-all:为策略名称

^:为匹配符，只有一个^代表匹配所有，^kevin为匹配名称为kevin的exchange或者queue

Ha-mode:为匹配类型，它分为三种模式：all所有的queue，exctly一部分（需要配置ha-params参数，此参数为int类型，例如3代表众多集群中的随机三台机器），node 指定（需要配置ha-params参数，此参数为数组类型例如[“3rabbit@F”,”rabbit@G”]这样指定为f与g这两台机器）





rabbitmqctl set\_policy [-p Vhost] Name Pattern Definition [Priority]

-p Vhost： 可选参数，针对指定vhost下的queue进行设置

Name: policy的名称

Pattern: queue的匹配模式(正则表达式)

Definition：镜像定义，包括三个部分ha-mode, ha-params, ha-sync-mode

        ha-mode:指明镜像队列的模式，有效值为 all/exactly/nodes

        all：表示在集群中所有的节点上进行镜像

        exactly：表示在指定个数的节点上进行镜像，节点的个数由ha-params指定

        nodes：表示在指定的节点上进行镜像，节点名称通过ha-params指定

        ha-params：ha-mode模式需要用到的参数

        ha-sync-mode：进行队列中消息的同步方式，有效值为automatic和manual（默认为manual，新节点加入后不会同步现存消息，新消息会同步，除非调用同步命令，队列开始阻塞，直到同步完成）

priority：可选参数，policy的优先级

当slave宕掉了，除了与slave相连的客户端连接全部断开之外，没有其他影响。

   当master宕掉时，会有以下连锁反应：

1. 与master相连的客户端连接全部断开；

2.选举最老的slave节点为master。若此时所有slave处于未同步状态，则未同步部分消息丢失；

3.新的master节点requeue所有unack消息，,因为这个新节点无法区分这些unack消息是否已经到达客户端，亦或是ack消息丢失在老的master的链路上，亦或者是丢在master组播ack消息到所有slave的链路上。所以处于消息可靠性的考虑，requeue所有unack的消息。此时客户端可能有重复消息；

4.如果客户端连着slave，并且Basic.Consume消费时指定了x-cancel-on-ha-failover参数，那么客户端会受到一个Consumer Cancellation Notification通知。如果未指定x-cancal-on-ha-failover参数，那么消费者就无法感知master宕机，会一直等待下去。

这就告诉我们，集群中存在镜像队列时，重新master节点有风险。

**镜像队列中节点启动顺序，非常有讲究：**

假设集群中包含两个节点，一般生产环境会部署三个节点，但为了方便说明，采用两个节点的形式进行说明。

**场景1：A先停，B后停**

该场景下B是master，只要先启动B，再启动A即可。或者先启动A，再在30s之内启动B即可恢复镜像队列。（**如果没有在30s内回复B，那么A自己就停掉自己**）

**场景2：A，B同时停**

该场景下可能是由掉电等原因造成，只需在30s内联系启动A和B即可恢复镜像队列。

**场景3：A先停，B后停，且A无法恢复。**

因为B是master，所以等B起来后，在B节点上调用rabbitmqctl forget\_cluster\_node A以解除A的cluster关系，再将新的slave节点加入B即可重新恢复镜像队列。

**场景4：A先停，B后停，且B无法恢复**

该场景比较难处理，旧版本的RabbitMQ没有有效的解决办法，在现在的版本中，因为B是master，所以直接启动A是不行的，当A无法启动时，也就没办法在A节点上调用rabbitmqctl forget\_cluster\_node B了，新版本中forget\_cluster\_node支持-offline参数，offline参数允许rabbitmqctl在离线节点上执行forget\_cluster\_node命令，迫使RabbitMQ在未启动的slave节点中选择一个作为master。当在A节点执行rabbitmqctl forget\_cluster\_node -offline B时，RabbitMQ会mock一个节点代表A，执行forget\_cluster\_node命令将B提出cluster，然后A就能正常启动了。最后将新的slave节点加入A即可重新恢复镜像队列

**场景5：A先停，B后停，且A和B均无法恢复，但是能得到A或B的磁盘文件**

这个场景更加难以处理。将A或B的数据库文件（$RabbitMQ\_HOME/var/lib目录中）copy至新节点C的目录下，再将C的hostname改成A或者B的hostname。如果copy过来的是A节点磁盘文件，按场景4处理，如果拷贝过来的是B节点的磁盘文件，按场景3处理。最后将新的slave节点加入C即可重新恢复镜像队列。

场景6：A先停，B后停，且A和B均无法恢复，且无法得到A和B的磁盘文件

无解。

启动顺序中有一个30s 的概念，这个是MQ 的时间间隔，用于检测master、slave是否可用，因此30s 非常关键。

 对于生产环境MQ集群的重启操作，需要分析具体的操作顺序，不可无序的重启，会有可能带来无法弥补的伤害(数据丢失、节点无法启动)。

简单总结下：镜像队列是用于节点之间同步消息的机制，避免某个节点宕机而导致的服务不可用或消息丢失，且针对排他性队列设置是无效的。另外很重要的一点，镜像队列机制不是负载均衡。

消息的发布（除了Basic.Publish之外）与消费都是通过master节点完成（未使用负载应用）。master节点对消息进行处理的同时将消息的处理动作通过GM广播给所有的slave节点，slave节点的GM收到消息后，通过回调交由mirror\_queue\_slave进行实际的处理。

对于Basic.Publish，消息同时发送到master和所有slave上，如果此时master宕掉了，消息还发送slave上，这样当slave提升为master的时候消息也不会丢失。

### 1.2.3 安装HaProxy

1、**HaProxy简介**

HAProxy是一款提供高可用性、 负载均衡以及基于TCP和HTTP应用的代理软  
件， HAProxy是完全免费的、 借助HAProxy可以快速并且可靠的提供基于TCP和 HTTP应用的代理解决方案。 HAProxy适用于那些负载较大的web站点， 这些站点通常又需要会话保持或七层  
处理。 HAProxy可以支持数以万计的并发连接,并且HAProxy的运行模式使得它可以很  
简单安全的整合进架构中， 同时可以保护web服务器不被暴露到网络上

**2、HAProxy安装**

⑴下载依赖包

yum install gcc vim wget

⑵下载hyproxy（目前此网站国内无法访问）

wget [http://www.haproxy.org/download/1 .6/src/haproxy-1 .6.5.tar.gz](http://www.haproxy.org/download/1%20.6/src/haproxy-1%20.6.5.tar.gz)

⑶cd /usr/local/software 进行解压  
tar -zxvf haproxy-1 .6.5.tar.gz -C /usr/local  
解压后会在 /usr/local 中生成一个文件 haproxy-1 .6.5文件夹

⑷进入目录、进行编译、安装

cd /usr/local/haproxy-1 .6.5 进入解压目 录  
编译 make TARGET=linux31 PREFIX=/usr/local/haproxy  
安装: make install PREFIX=/usr/local/haproxy

⑸创建一个haproxy目录，用于存放haproxy的配置文件：

创建目录：mkdir /etc/haproxy

赋权：groupadd –r –g 149 haproxy

Useradd –g haproxy –r –s /sbin/nologin –u 149 haproxy

创建配置文件

touch /etc/haproxy/haproxy.cfg

⑹haproxy.cfg配置文件详解

#lgging options  
global  
log 1 27.0.0.1 local0 info  
maxconn 51 20  
chroot /usr/local/haproxy  
uid 99  
gid 99  
daemon  
quiet  
nbproc 20  
pidfile /var/run/haproxy.pid  
defaults  
log global  
#使用4层代理模式（几乎可以对所有应用做负载，包括http、数据库、在线聊天室）， ”mode http”为7层代理模式（作用于应用层仅支持http协议）  
mode tcp  
#if you set mode to tcp,then you nust change tcplog into httplog  
option tcplog  
option dontlognull  
retries 3  
option redispatch  
maxconn 2000  
contimeout 5s  
##客户端空闲超时时间为 60秒 则HA 发起重连机制  
clitimeout 60s  
##服务器端链接超时时间为 15秒 则HA 发起重连机制  
srvtimeout 15s  
#front-end IP for consumers and producters  
listen rabbitmq\_cluster  
#监听的端口  
bind 0.0.0.0:5672  
#配置TCP模式  
mode tcp  
#简单的轮询  
balance roundrobin  
##===================================超级 超级 重要防止应用程序连接关闭===================================  
timeout client 3h  
timeout server 3h  
#rabbitmq集群节点配置 #inter 每隔五秒对mq集群做健康检查， 2次正确证明服务器可用， 2次失败证明服务器不可用， 并且配置主备机制  
server smlz86 1 92.1 68.1 59.86:5672 check inter 5000 rise 2 fall 2  
server smlz87 1 92.1 68.1 59.87:5672 check inter 5000 rise 2 fall 2  
server smlz88 1 92.1 68.1 59.88:5672 check inter 5000 rise 2 fall 2  
#配置haproxy web监控， 查看统计信息  
listen stats  
bind 192.168.159.89:8100  
mode http  
option httplog  
stats enable  
#设置haproxy监控地址为http://192.168.159.89:8100/rabbitmq-stats  
stats uri /rabbitmq-stats  
stats refresh 5s

⑺启动haproxy

/usr/local/haproxy/sbin/haproxy -f /etc/haproxy/haproxy.cfg

⑻查看haproxy是否启动

ps-ef| grep haproxy

haproxy的监控台访问  
http://1 92.1 68.1 59.89:81 00/rabbitmq-stats

关闭haproxy命令 killall haproxy

### 1.2.4HaProxy集群的搭建与Keepalived的整合

Haproxy本身可以在四层（TCP）进行负载，加入Keepalived对haproxy进行第一层负载，Keepalived本身支持lvs，主要的应用为Keepalived+lvs/DR 模式

# 2、rabbitmq基础知识

1、rabbitmq的优势

1.开源，性能好，稳定性保证；  
2.提供了消息的可靠性投递（confirm），返回模式；  
3.与sping amqp 整合和完美，提供丰富的api；  
4.集群模式十分丰富(普通集群模式，HA模式镜像队列模型)；  
5.保证数据不丢失的情况下，保证很好的性能；

**2、rabbitmq高性能的原因**

1.使用的语言是elang语言(通常使用到交互机上)，elang的语言的性能能与原生socket的延迟效果;  
2.消息入队的延时已经消息的消费的响应很快。

## 2.1 AMQP协议(Advanced message queue protocol) 高级消息队列协议

### 2.1.1 AMQP的概念

1、什么是AMQP协议

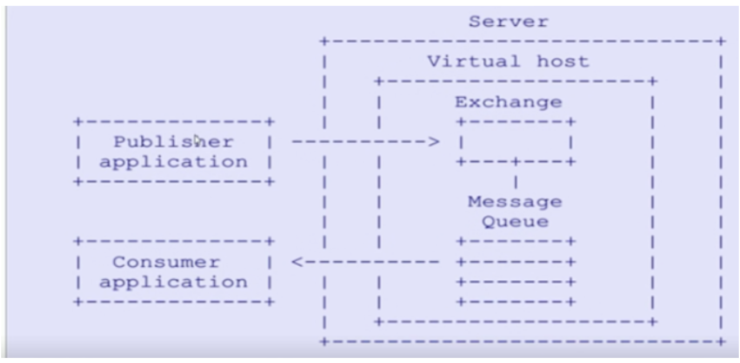
⑴是一个二进制协议；⑵amqp是一个应用层协议规范（定义了很多规范），可以有很多种不同的消息中间件产品（需要遵循该规范）

sever：消息队列节点

virtual host：虚拟主机

exchange：交换机（消息投递到交换机上）

message queue：消息队列（被消费者监听消息）



2、AMQP的核心概念

⑴server：又称broker，接受客户端连接，实现AMQP实体服务

⑵connection：连接，应用程序与broker建立网络连接

⑶channel：网络通道，几乎所有的 操作都是在channel中进行的，是进行消息传递的通道，客户端可以建立多个通道，每一个channel表示一个会话任务

⑷message：服务器与应用程序之间传递数据的载体，有properties（消息属性，用来修饰消息，比如消息的优先级，延时投递）和body（消息体）

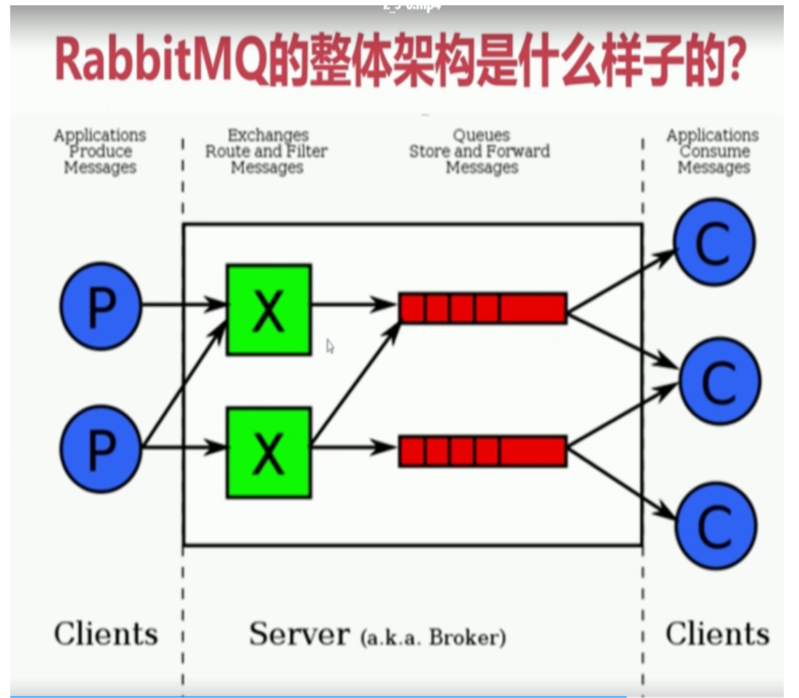
⑸virtual host（虚拟主机）：是一个逻辑概念，最上层的消息路由，一个虚拟主机可以包含多个exchange和queue，但是一个虚拟主机中不能有名称相同的exchange和queue

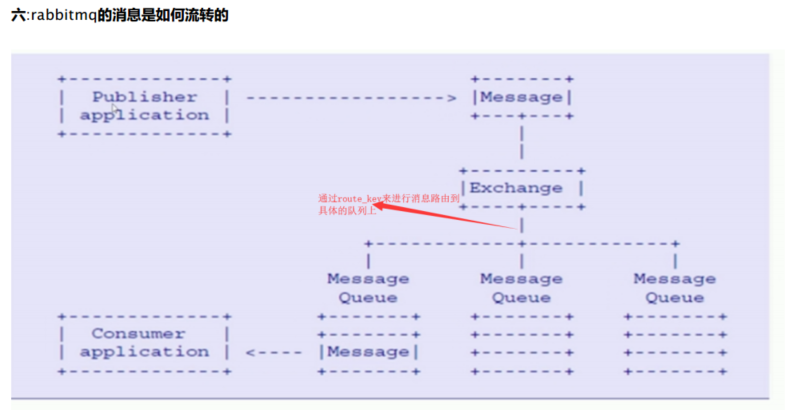
⑹exchange：交换机，消息直接投递到交换机上，然后交换机根据消息的路由key来路由到对应绑定的队列上

⑺binding：绑定exchange与queue的虚拟连接，binding中可以包含route\_key

⑻route\_key:路由key，他的作用是在交换机上通过route\_key来把消息路由到哪个队列上

⑼queue：队列，用于保存消息的载体，有消费者监听，然后消费消息





### 2.1.2 消息的具体属性

消息包括消息体body与属性properties

**常用属性：**

delivery mode:是否持久化（值为2为持久化）

headers：

content\_type:消息类型，当需要将消息传递过来的对象自动转换为map，list或者对象实例，参数要设置为”application/json”,当使用监听获取消息手动作对象转换时则不需要

content\_encoding:消息编码格式

correntlation\_id:判断消息队列和回调队列，方便RPC响应与请求关联，通过其确定回调队列返回的信息是哪条消息的回复，其必须唯一

reply\_to:消息失败是否重回队列

expiration：消息过期时间

message\_id:消息id

timestamp：

type：

user\_id:

app\_id:

cluster\_id:

**消费端代码：**

//7 获取消息  
Delivery delivery = queueingConsumer.nextDelivery();  
String msg = new String(delivery.getBody());  
System.out.println("消费端: " + msg);  
System.out.println(delivery.getProperties());  
System.out.println(delivery.getProperties().getHeaders());

**生产端代码：**

Map<String,Object> extraMap = new HashMap<>();  
extraMap.put("k1 ","v1 ");  
extraMap.put("k2","v2");  
/\*\*  
\* 附带额外信息的消息体  
\*/  
AMQP.BasicProperties basicProperties = new AMQP.BasicProperties.Builder()  
.deliveryMode(2)//2为持久化,1 不是持久化  
.appId("测试appid")  
.clusterId("测试集群id")  
.contentType("application/json")  
.contentEncoding("UTF-8")  
.headers(extraMap)

.build();  
for (int i = 0; i < 1 0; i++) {  
String tragetMsg = "这是我的第【"+(i+1 )+"】 条消息";  
channel.basicPublish("", "test001 ", basicProperties, tragetMsg.getBytes());

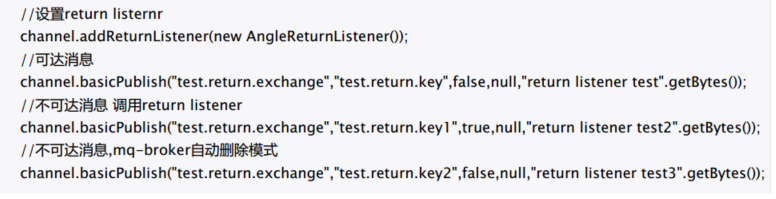
### 2.1.3 return listener 消息处理机制

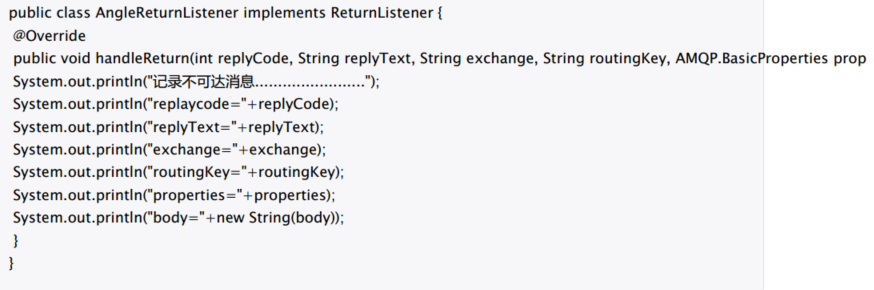
有两种情况需要return listener 处理

1、broker中根本没有对应的exchange交换机来接受消息

2、消息能够投递到broker的交换机上，但是交换机根据路由key路由不到某一个队列上

消息生产端的mandatory设置为true 那么就会调用生产端的return listener处理，否则mq-broker就会自动删除消息





### 2.1.4 rabbitmq 的配置

#### 1、CachingConnectionFactory

CachingConnectionFactory为我们提供了两种缓存的模式：

* CHANNEL模式：这也是CachingConnectionFactory的默认模式，在这种模式下，所有的createConnection（）方法实际上返回的都是同一个Connection，同样的Connection.close()方法是没用的，因为就一个，默认情况下，Connection中只缓存了一个Channel，在并发量不大的时候这种模式是完全够用的，当并发量较高的时候，我们可以setChannelCacheSize（）来增加Connection中缓存的Channel的数量。
* CONNECTION模式：在CONNECTION模式下，每一次调用createConnection（）方法都会新建一个或者从缓存中获取，根据你设置的ConnectionCacheSize的大小，当小于的时候会采用新建的策略，当大于等于的时候会采用从缓存中获取的策略，与CHANNEL模式不同的是，CONNECTION模式对Connection和Channel都进行了缓存，最新版本的client中已经将Channel的缓存数量从1增加到了25，但是在并发量不是特别大的情况下，作用并不是特别明显。  
  **使用CachingConnectionFactory需要注意的一点是：所有你获取的Channel对象必须要显式的关闭，所以finally中一定不要忘记释放资源，如果忘记释放，则可能造成连接池中没有资源可用，所以手动签收时不能使用，因为不会释放channel**。

#### 2、rabbitmqTemplate的convertAndSend与send方法的区别？

rabbitmqTemplate的convertAndSend与send方法底层都是调用rabbitmqTemplate的send方法，区别为参数列表不同，第一种可以加correlationData，用于可靠性投递，表明消息的唯一性标志，而send方法无此参数

#### 3. SimpleMessageListenerContainer与Jackson2JsonMessageConverter

Jackson2JsonMessageConverter

MessageProperties messageProperties = new MessageProperties();

messageProperties.setContentType("application/json");

设置contentType类型，消费端将消息转换为Map类型，而非字节数组，当传输的为实体的list集合时，接收端接收到的参数为List<T>

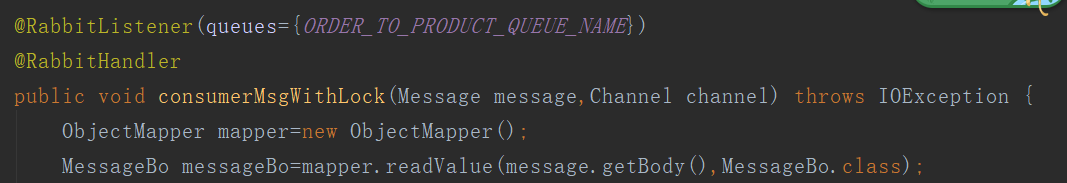




将发送的消息的对象转换为对象实体，对于跨系统的情况，发送消息的类和接受消息的类必须是一样的，不仅是要里面的字段一样，类名一样，连类的包路径都要一样。

所以当系统1使用 JsonMessageConverter 发送消息类A给系统2时，系统2可以有如下几种方式来接收：

* 1.依赖系统1的jar包,直接使用类A来接收
* 2.不依赖系统1的jar包，自己建一个和A一模一样的类，连名称，包路径都一样
* 3.负责监听 queue 的类实现 MessageListener 接口，直接接收 Message 类，再自己转换



项目中一般使用第三种情况，如果第二种情况（通过配置SimpleMessageListenerContainer实现消息的监听）也可以通过在消息的生产端设置消费端消息对应实体的全限定名，或者在消费端解析消息之前修改TypeId为消费端消息对应实体的全限定名

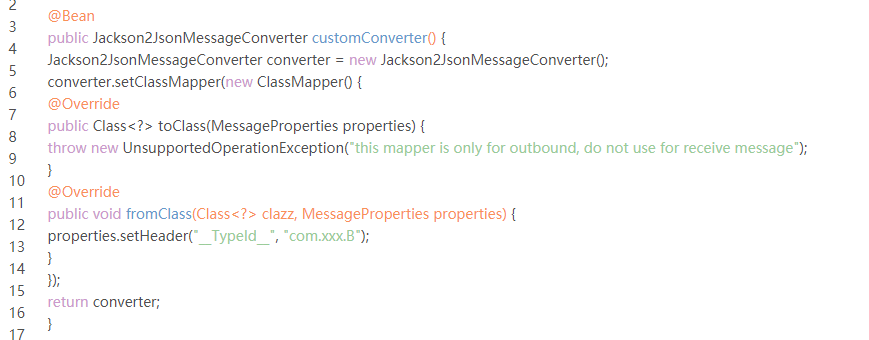
MessageProperties messageProperties = new MessageProperties();

messageProperties.setContentType("application/json");

//指定的\_\_TypeId\_\_属性值必须是消费端的Order的全类名，如果不匹配则会报错。

messageProperties.getHeaders().put("\_\_TypeId\_\_","com.zhihao.miao.test.day10.Sender.Order");

在消息的生产者端为 JsonMessageConverter， 设置一个自定义的 ClassMapper，重写 fromClass 方法，将 \_\_TypeId\_\_ 的值设为消费端用来接收的类的路径+名称。当然了，也可以在消费者端重写toClass方法，直接返回想要转换的目标类的类类型。两种选一种就可以

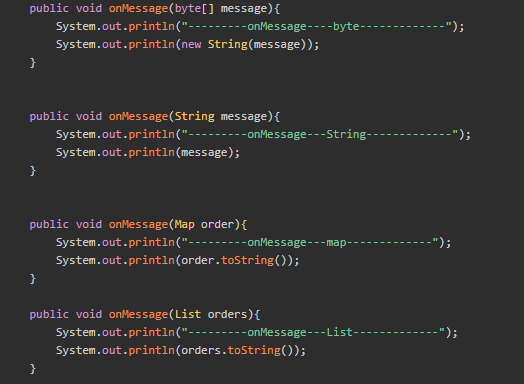


tulingvip-rabbitmq-springwithrabbitmq

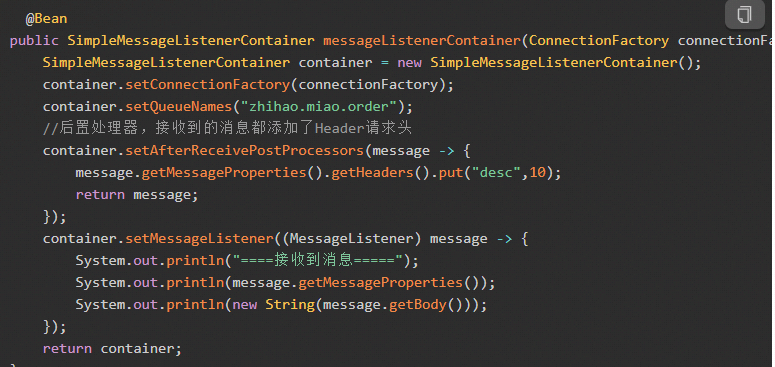
同一个queue上有多个消费者的时候，只会有一个消费者收到消息，一般是多个消费者轮流收到消息。

使用SimpleMessageListenerContainer的一个优点是可以根据消息传输的具体类型走具体的逻辑

运用方法的重载



setAfterReceivePostProcessors方法可以对消息进行后置处理



## 2.2 rabbitmq 队列queue

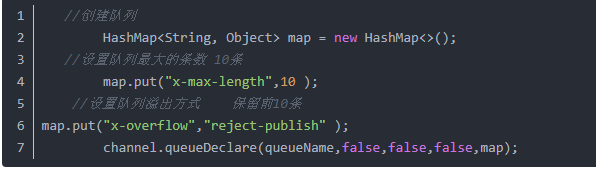
### 2.2.1队列参数配置

1、队列消息承载量设置

通过为队列声明参数 x-max-length 提供一个非负整数值来设置最大消息数。

通过为队列声明参数 x-max-length-bytes 提供一个非负整数值，设置最大字节长度。

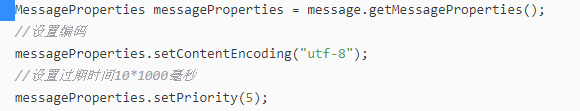
溢出行为通过参数x-overflow设置,默认为丢掉最早的消息，如果 overflow 设置为 reject-publish，则最近发布的消息将被丢弃。此外，如果 [发布者确认](https://www.rabbitmq.com/confirms.html#publisher-confirms) 已启用，将通过 basic.nack 消息对发布者进行拒绝通知。如果一条消息被路由到多个队列并被其中至少一个队列拒绝，该信道将通过 basic.nack 通知发布者。该消息仍将被发布到可以将其排队的所有其他队列。



2、消息的优先级设置

配置队列  主要填充x-max-priority属性 并设置最高优先级值为10 默认最低为0





3、消息的过期时间设置

队列声明时map集合中添加info.put(“x-message-ttl”,10000),表示队列中的消息10s没有被消费就会过期

是否持久化、是否独占、是否自动删除、具体的配置参数（map）

channel.queueDeclare(queueName,true,false,true,null);

### 2.2.2 消息的签收、确认、顺序消费

第一步:在channel上开启确认模式 channel.confirmSelect();  
第二部;在channel上增加confirm监听， 来监听成功和异常的confirm结果

channel.confirmSelect();  
channel.addConfirmListener(new ConfirmListener(){}

消费消息，消费端限流，需要关闭消息自动签收，false为手动签收，true为自动签收  
channel.basicConsume(queueName,false,newAckConsumer(channel));

//手动签收，服务端会将该条消息删除，false为multiple，删除一条，true为删除多条  
channel.basicAck(envelope.getDeliveryTag(),false);

//重回队列,在手动签收模式下重回队列会死循环，卡在这条消息上，false为multiple，删除一条，true为删除多条，true为重回队列  
channel.basicNack(envelope.getDeliveryTag(),false,true);

longdeliveryTag=message.getMessageProperties().getDeliveryTa();

消息签收或者拒签都需要发送deliveryTag，deliveryTag（唯一标识 ID，与correlationId不同）：当一个消费者向 RabbitMQ 注册后，会建立起一个 Channel ，RabbitMQ 会用 basic.deliver 方法向消费者推送消息，这个方法携带了一个 delivery tag， 它代表了 RabbitMQ 向该 Channel 投递的这条消息的唯一标识 ID，是一个单调递增的正整数，delivery tag 的范围仅限于 Channel，delivery\_tag是消息投递序号，每个channel对应一个(long类型)，从1开始到9223372036854775807范围，在手动消息确认时可以对指定delivery\_tag的消息进行ack、nack、reject等操作。

每次消费或者重新投递requeue后，delivery\_tag都会增加，理论上该正常业务范围内，该值永远不会达到最大范围上限。可以根据每个消费者对应channel的delivery\_tag消费速率计算到达最大值需要的时间。

假设：每秒钟一个消费者可以消费1000w个消息(假设每个消费者一个channel)，则 9223372036854775807 / (60 \* 60 \* 24 \* 365 \* 1000w) = 29247年后能达到上限数值。

**消息的限流**

prefetchSize：每条消息的小的设置，目前不可用

prefetchCount：标识每次推送多少条消息，一般为一条

global：false标识channel级别，true标识消费者级别，即上面的设置应用于哪个级别

以上设置只有在手动签收的情况下生效，不论手动签收或者自动签收，或者签收与否，rabbitmq都会把队列中的所有消息一次性全部发送给消费者，当使用QoS与手动签收后，只有消息被手动签收后，队列才会将新的消息推送给消费者

channel.basicQos(0,1,false);

**消息的顺序消费**：

如果消息有业务上的逻辑先后性，需要顺序执行，则需要为每一个队列只限制有一个消费者，并使用手动签收，每次拉取一条消息

### 2.2.3 消息的事物

消息发出后是否到达broker服务器，未解决这个问题rabbitmq提供了两种方案

1、通过AMQP事务机制实现，这是AMQP协议层的解决方案

2、通过将channel设置成confirm模式实现

**事务机制：**

RabbitMQ中与事务机制有关的方法有三个：txSelect(), txCommit()以及txRollback(), txSelect用于将当前channel设置成transaction模式，txCommit用于提交事务，txRollback用于回滚事务，在通过txSelect开启事务之后，我们便可以发布消息给broker代理服务器了，如果txCommit提交成功了，则消息一定到达了broker了，如果在txCommit执行之前broker异常崩溃或者由于其他原因抛出异常，这个时候我们便可以捕获异常通过txRollback回滚事务了。

关键代码：

带事务的多了四个步骤：

client发送Tx.Select

broker发送Tx.Select-Ok(之后publish)

client发送Tx.Commit

broker发送Tx.Commit-Ok

采用事务机制会严重降低RabbitMQ的消息吞吐量，所以通常都是使用确认模式。

Confirm：详见2.2.2消息的签收

### 2.3.4 死信队列

1、什么是死信？

队列中的消息如果没有消费者消费，那么该消息就成为一个死信，那这个消息被重新发送到另外一个exchange上的话，那这个exchange就是死信队列

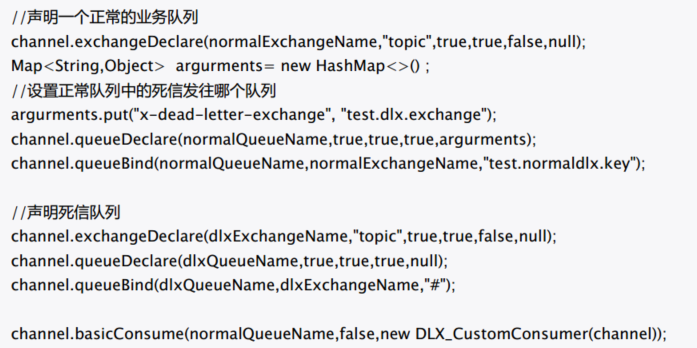
2、消息变成死信的几种情况

⑴消息被拒绝：（basic.reject/basic.nack）并且requeue重回队列的属性设置为false表示不需要重回队列，那么该消息就成为死信

⑵消息ttl过期：消息本身设置了过期时间，或者队列设置了消息的过期时间x-message-ttl

⑶队列达到最大长度：比如队列最大长度是3000，那么超出长度的消息会被发送到死信队列上

死信队列也是一个正常的exchange，也会通过routingkey绑定到具体的队列上



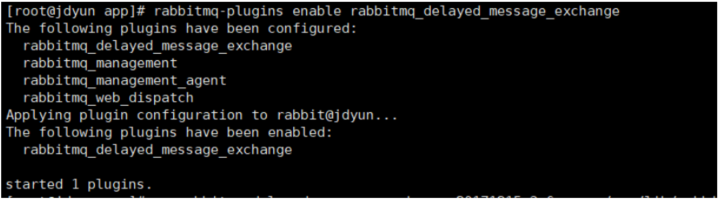
### 2.3.5 延迟队列

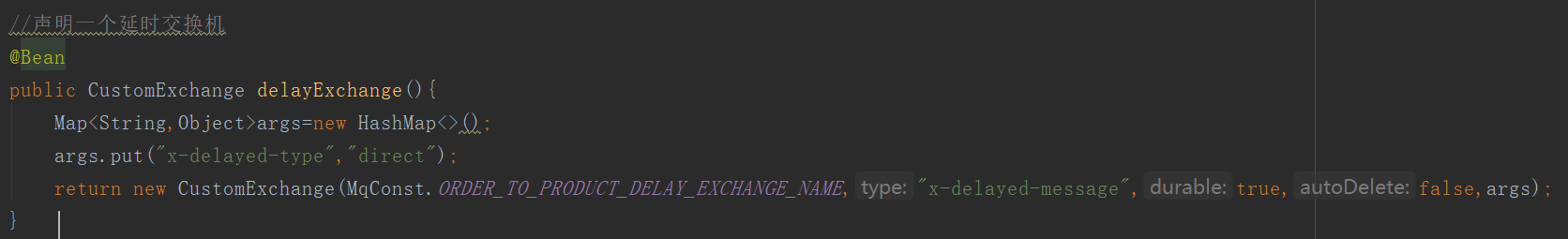
Rabbitmq实现延迟队列的两种方式

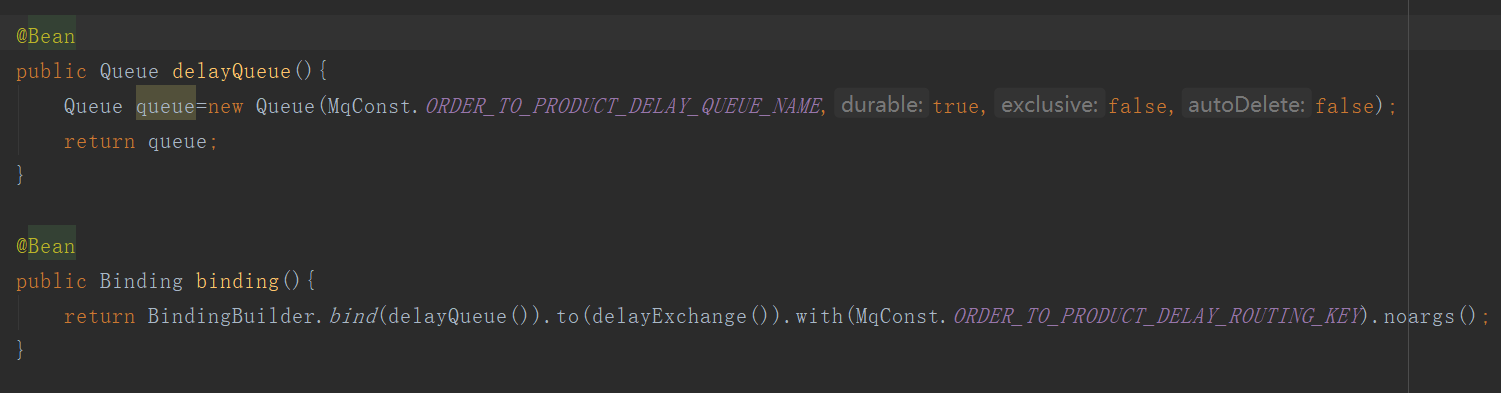
1、延迟队列可以由死信队列改造，通过设置消息过期时间，将消息转发到死信队列然后，进行消费

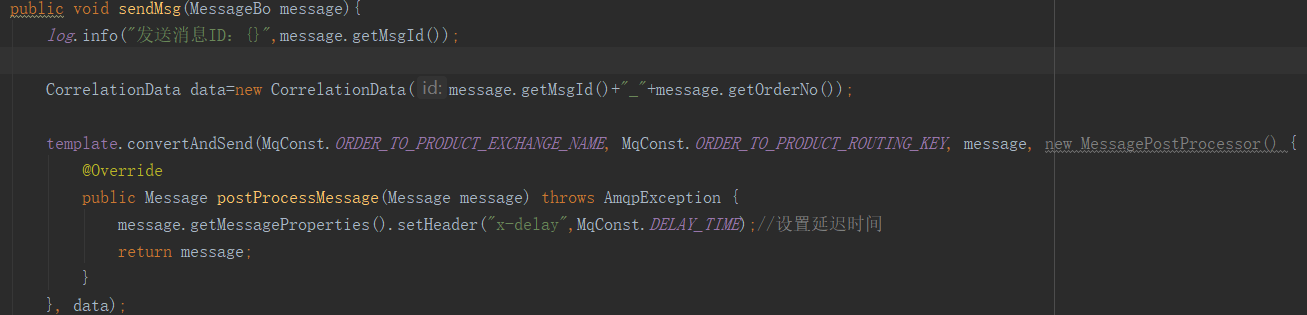
2、通过使用延时队列插件来实现延时队列的功能rabbitmq\_delayed\_message\_exchange，目前实际业务推荐用法

解压延时插件: unzip rabbitmq\_delayed\_message\_exchange-201 71 21 5-3.6.x.zip  
把延时插件拷贝到指定目 录下:cp rabbitmq\_delayed\_message\_exchange-201 71 21 5-3.6.x.ez  
/usr/lib/rabbitmq/lib/rabbitmq\_server-3.7.5/plugins  
启动延时插件(安装后需要先重启rabbitmq): rabbitmq-plugins enable rabbitmq\_delayed\_message\_exchange









## 2.3 rabbitmq 交换机

### 2.3.1 交换机的参数

Name: 交换机的名称

Type: 交换机的类型,direct,topic,fanout,headers

Durability :是否需要持久化

autodelete:假如 没有队列绑定到该交换机，那么该交换机会自动删除

Internal: 当前交换机是否用户rabbitmq内部使用，不常用,默认为false

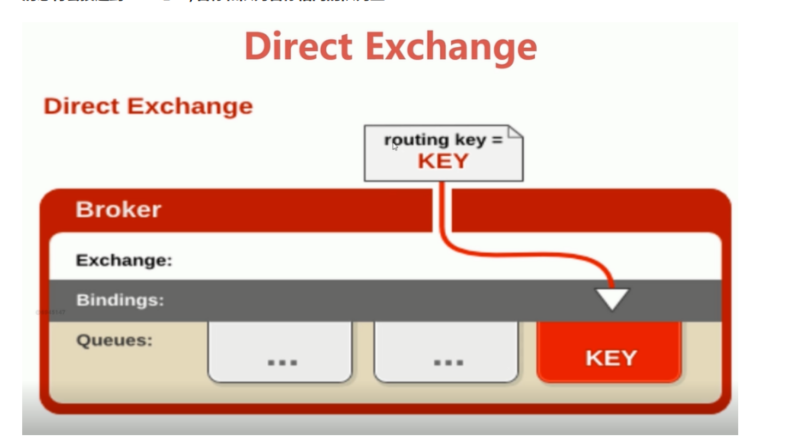
Argurements: 扩展参数,用户扩展AMQP 定制化协议

### 2.3.2 直接交换机

1、rabbitmq的默认交换机

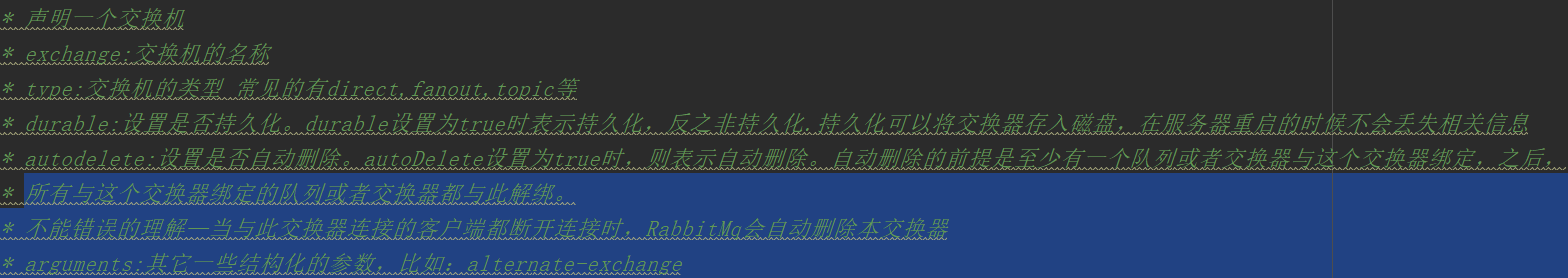
默认的exchange:如果用空字符串去声明一个exchange，那么系统就会使用””AMQP default”这个exchange，我们创建一个queue时,默认的都会有一个和新建queue同名的routingKey绑定到这个默认的exchange上去，发送消息时使用的routingkey必须与队列名称相同

直接交换机与queue绑定时，routingkey不支持通配符，发送消息是使用的routingkey必须与绑定时的routingkey完全相同才可以

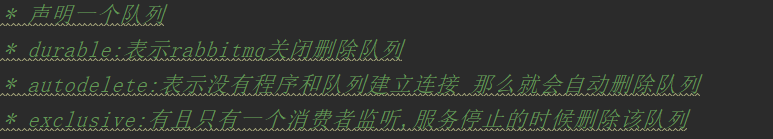


2、直接交换机消费者代码

声明交换机的参数



声明队列的参数



// 创建连接工厂  
 ConnectionFactory factory=new ConnectionFactory();  
 factory.setHost("192.168.159.8");  
 factory.setPort(5672);  
 factory.setUsername("kevin");  
 factory.setPassword("kevin");  
  
 //创建连接  
 Connection connection=factory.newConnection();  
 //创建一个channel  
 Channel channel=connection.createChannel();  
  
 String exchangeName="kevin.directChange";  
 String exchangrType="kevin.directQueue";  
 String queueName="kevin.directQueue";  
 String routingKey="kevin.directChange.key";  
 /\*\*  
 durable：是否持久化，队列的声明默认是存放在内存中的，如果rabbit重启会丢失，如果想要重启之后依然存在就要使用队列持久化  
\* 保存到Erlang自带的Mnesia数据库中，当rabbitmq重启之后会读取该数据库  
\* exclusive：该队列是否有私有的private，如果不是排外的，可以使用两个消费者都访问同一个队列  
\* 如果是排外的，会对当前队列加锁，其他通道channel是不能访问的，如果强制访问回报异常  
\* com.rabbitmq.client.ShutdownSignalException: channel error; protocol method: #method<channel.close>(reply-code=405,  
\* reply-text=RESOURCE\_LOCKED - cannot obtain exclusive access to locked queue 'queue\_name' in vhost '/', class-id=50, method-id=20)  
\*一般等于true的话用于一个队列只能有一个消费者来消费的场景  
\* autodelete：是否自动删除，当最后一个消费者断开连接之后队列是否自动删除，可以通过RabbitMQ Management  
\* 查看某个队列的消费者数量，当consumers=0时队列就会自动删除

\* arguments：其他一些结构化的参数，比如：alternate-exchange  
 \*  
 \* \*/  
 channel.exchangeDeclare(exchangeName,exchangrType,true,false,null);  
  
 /\*\*  
 \* 声明一个队列  
 \* durable：表示rabbitmq关闭则删除队列  
 \* autodelete：表示表示没有程序和队列建立连接，那么就会自动删除队列  
 \*  
 \* \*/  
 channel.queueDeclare(queueName,true,false,false,null);  
  
 //队列与交换机绑定  
 channel.queueBind(queueName,exchangeName,routingKey);  
  
 //创建一个消费者  
 QueueingConsumer consumer=new QueueingConsumer(channel);  
  
 channel.basicConsume(queueName,true,consumer);  
  
 while (true){  
 QueueingConsumer.Delivery delivery=consumer.nextDelivery();  
 String message=new String(delivery.getBody());  
 System.out.println("消费信息--"+message);  
 }  
}

3、生产者代码

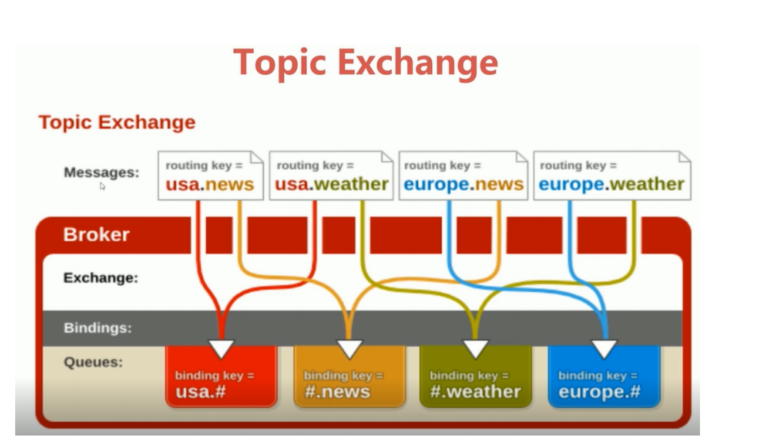
//创建连接工厂  
 ConnectionFactory factory=new ConnectionFactory();  
 factory.setHost("192.168.159.8");  
 factory.setPort(5672);  
 factory.setVirtualHost("kevin");  
 factory.setUsername("kevin");  
 factory.setPassword("kevin");  
  
 //创建连接  
 Connection connection=factory.newConnection();  
 //创建一个channel  
 Channel channel=connection.createChannel();  
  
 //定义交换机名称  
 String exchangeName="kevin.directchange";  
  
 //定义routingKey  
 String routingKey="kevin.directchange.key";  
  
 //消息体内容  
 String messageBody="hello kevin";  
 channel.basicPublish(exchangeName,routingKey,null,messageBody.getBytes());  
}

### 2.3.3 主题交换机TopicExchange

本质是绑到top交换机上的路由key可以通过通配符来匹配，匹配规则如下

log.#:可以匹配一个单词，也可以匹配多个单词，比如log.a,log.a.b,log.a.c.b

log.\*:只可以匹配一个单词，例如log.s，不能匹配log.s.c



**生产者代码**

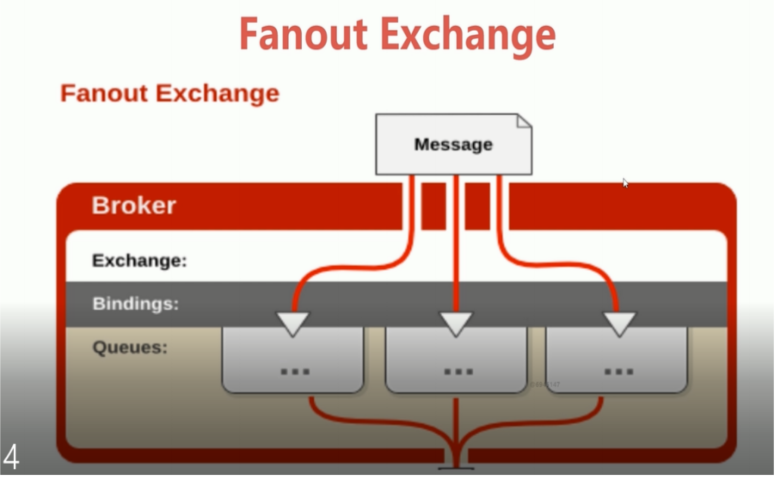
String topExchangeName = "top.exchange";  
String routingKey1 = "top.key.1 ";  
String routingKey2 = "top.key.2";  
channel.basicPublish(topExchangeName,routingKey1 ,null,"测试交换机".getBytes());  
channel.basicPublish(topExchangeName,routingKey2,null,"测试交换机".getBytes());  
channel.close();  
connection.close();

**消费者代码：**

//声明一个交换机  
channel.exchangeDeclare(topicExchangeName,"topic",true,true,false,null);  
//声明一个队列  
channel.queueDeclare("top.queue",true,false,true,null);  
//队列绑定到交换机  
channel.queueBind("top.queue",topicExchangeName,"top.#");  
QueueingConsumer queueingConsumer = new QueueingConsumer(channel);  
//参数： 队列名称、 是否自动ACK、 Consumer，自动ack指消息拉取过来之后直接确认接收成功  
channel.basicConsume("top.queue", true, queueingConsumer);  
while (true) {  
QueueingConsumer.Delivery delivery = queueingConsumer.nextDelivery();  
System.out.println(new String(delivery.getBody()));  
}

### 2.3.3 扇形交换机fanout exchange

交换机与队列绑定时无需routingkey，所以该模式是最快的，没有字符串匹配的环节，消息会发送到所有与交换机绑定的队列上



**生产端代码：**

String fanoutExchangeName = "test.fanout.exchange";  
String routingKey = "test.fanout.key";  
channel.basicPublish(fanoutExchangeName,routingKey,null,"測試扇形交换机。 。 。 。 。 。 ".getBytes());

**消费端代码：**

//创建交换机  
channel.exchangeDeclare("test.fanout.exchange","fanout",true,true,false,null);  
//创建队列  
channel.queueDeclare("test.fanout.queue",true,false,true,null);  
//绑定队列  
channel.queueBind("test.fanout.queue","test.fanout.exchange","");  
QueueingConsumer queueingConsumer = new QueueingConsumer(channel);  
channel.basicConsume("test.fanout.queue",true,queueingConsumer);  
while (true) {  
QueueingConsumer.Delivery delivery = queueingConsumer.nextDelivery();  
System.out.println(new String(delivery.getBody()));

## 2.4 rabbitMQ的七种工作模式

### 4.1 简单队列模式

此种方式没有引入交换机，工作队列也没有

最简单的工作队列，其中一个消息生产者，一个消息消费者，一个队列。也称为点对点模式

Rabbitmq队列中的单条消息只能被一个消费者消费

### 4.2 工作队列

用来将耗时的任务分发给多个消费者（工作者）

主要解决问题：处理资源密集型任务，并且还要等他完成。有了工作队列，我们就可以将具体的工作放到后面去做，将工作封装为一个消息，发送到队列中，一个工作进程就可以取出消息并完成工作。如果启动了多个工作进程，那么工作就可以在多个进程间共享。

工作队列有循环分发与公平性分发两种模式

循环分发，假如我们拥有两个消费者，默认情况下，RabbitMQ 将按顺序将每条消息发送给下一个消费者，平均而言，每个消费者将获得相同数量的消息，这种分发消息的方式称为轮询。

生产者



**消费者：**



/\*\* 保证一次只分发一次 限制发送给同一个消费者 不得超过一条消息 \*/  
channel.basicQos(1);

通过此设置来决定分发模式，如果没有此设置，mq会轮询给生产者发送消息

场景一：如果生产者发送6条消息，消费者一接收到1、3、5条消息，消费者二接收到2、4、6条消息，如果此时消息为自动签收且消费者二处理速度慢，此时killed消费者二进程则消息会丢失，解决方式将消息签收改为手动签收，相同的场景消费者一会处理掉消费者二没处理完的消息

公平模式下消费者处理完一条消息后再去拉取一条消息，做到能者多劳

### 4.3 发布订阅模式

RabbitMQ的发布订阅模式不同于activeMQ，其实现方式是通过交换机将消息发送给多个队列，每个队列一个消费者，从而实现发布订阅，交换机可以使用扇形交换机较为方便，主题交换机也可以实现相应的功能

### 4.4 主题模式

使用主题交换机根据routingkey进行模糊匹配队列，具体参考主题交换机

### 4.5 路由模式

使用直接交换机，routingkey只能匹配到一个队列，具体参考直接交换机

### 4.6 RPC模式

RPC模式为客户端发送消息，接收端为其他系统，具体逻辑为客户端发送消息包含消息的correlationId与回调队列，消费端接收到消息后会将响应发送到回调队列，客户端接收回调队列的消息，以correlationId确认回调消息为哪条消息的回复

客户端代码

import com.rabbitmq.client.AMQP;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;

import java.util.UUID;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.TimeoutException;

public class RPCClient implements AutoCloseable {

private Connection connection;

private Channel channel;

private String requestQueueName = "rpc\_queue";

public RPCClient() throws IOException, TimeoutException {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

connection = factory.newConnection();

channel = connection.createChannel();

}

public static void main(String[] argv) {

try (RPCClient fibonacciRpc = new RPCClient()) {

for (int i = 0; i < 32; i++) {

String i\_str = Integer.toString(i);

System.out.println(" [x] Requesting fib(" + i\_str + ")");

String response = fibonacciRpc.call(i\_str);

System.out.println(" [.] Got '" + response + "'");

}

} catch (IOException | TimeoutException | InterruptedException e) {

e.printStackTrace();

}

}

public String call(String message) throws IOException, InterruptedException {

final String corrId = UUID.randomUUID().toString();

String replyQueueName = channel.queueDeclare().getQueue();

AMQP.BasicProperties props = new AMQP.BasicProperties

.Builder()

.correlationId(corrId)

.replyTo(replyQueueName)

.build();

channel.basicPublish("", requestQueueName, props, message.getBytes("UTF-8"));

final BlockingQueue<String> response = new ArrayBlockingQueue<>(1);

String ctag = channel.basicConsume(replyQueueName, true, (consumerTag, delivery) -> {

if (delivery.getProperties().getCorrelationId().equals(corrId)) {

response.offer(new String(delivery.getBody(), "UTF-8"));

}

}, consumerTag -> {

});

String result = response.take();

channel.basicCancel(ctag);

return result;

}

public void close() throws IOException {

connection.close();

}

}

服务端代码

import com.rabbitmq.client.\*;

public class RPCServer {

private static final String RPC\_QUEUE\_NAME = "rpc\_queue";

private static int fib(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

return fib(n - 1) + fib(n - 2);

}

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

try (Connection connection = factory.newConnection();

Channel channel = connection.createChannel()) {

channel.queueDeclare(RPC\_QUEUE\_NAME, false, false, false, null);

channel.queuePurge(RPC\_QUEUE\_NAME);//用于清空队列中的消息

channel.basicQos(1);

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

Object monitor = new Object();

DeliverCallback deliverCallback = (consumerTag, delivery) -> {

AMQP.BasicProperties replyProps = new AMQP.BasicProperties

.Builder()

.correlationId(delivery.getProperties().getCorrelationId())

.build();

String response = "";

try {

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

int n = Integer.parseInt(message);

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

response += fib(n);

} catch (RuntimeException e) {

System.out.println(" [.] " + e.toString());

} finally {

channel.basicPublish("", delivery.getProperties().getReplyTo(), replyProps, response.getBytes("UTF-8"));

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

// RabbitMq consumer worker thread notifies the RPC server owner thread

synchronized (monitor) {

monitor.notify();

}

}

};

channel.basicConsume(RPC\_QUEUE\_NAME, false, deliverCallback, (consumerTag -> { }));

// Wait and be prepared to consume the message from RPC client.

while (true) {

synchronized (monitor) {

try {

monitor.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

}

### 4.7 发布者确认模式

策略一：单独发布消息，同步等待确认：简单，但吞吐量非常有限。

策略二：批量发布消息，同步等待批量确认：简单，合理的吞吐量，但是很难推断出什么时候出了问题。

策略三：异步处理：最佳性能和资源使用，在发生错误的情况下可以很好地控制，但是可以正确实施。

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.ConfirmCallback;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import java.time.Duration;

import java.util.UUID;

import java.util.concurrent.ConcurrentNavigableMap;

import java.util.concurrent.ConcurrentSkipListMap;

import java.util.function.BooleanSupplier;

public class PublisherConfirms {

static final int MESSAGE\_COUNT = 50\_000;

static Connection createConnection() throws Exception {

ConnectionFactory cf = new ConnectionFactory();

cf.setHost("localhost");

cf.setUsername("guest");

cf.setPassword("guest");

return cf.newConnection();

}

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

publishMessagesIndividually();

publishMessagesInBatch();

handlePublishConfirmsAsynchronously();

}

//策略一：同步发送确认单条消息

static void publishMessagesIndividually() throws Exception {

try (Connection connection = createConnection()) {

Channel ch = connection.createChannel();

String queue = UUID.randomUUID().toString();

ch.queueDeclare(queue, false, false, true, null);

ch.confirmSelect();

long start = System.nanoTime();

for (int i = 0; i < MESSAGE\_COUNT; i++) {

String body = String.valueOf(i);

ch.basicPublish("", queue, null, body.getBytes());

ch.waitForConfirmsOrDie(5\_000);

}

long end = System.nanoTime();

System.out.format("Published %,d messages individually in %,d ms%n", MESSAGE\_COUNT, Duration.ofNanos(end - start).toMillis());

}

}

//策略二：批量的同步确认消息

static void publishMessagesInBatch() throws Exception {

try (Connection connection = createConnection()) {

Channel ch = connection.createChannel();

String queue = UUID.randomUUID().toString();

ch.queueDeclare(queue, false, false, true, null);

ch.confirmSelect();

int batchSize = 100;

int outstandingMessageCount = 0;

long start = System.nanoTime();

for (int i = 0; i < MESSAGE\_COUNT; i++) {

String body = String.valueOf(i);

ch.basicPublish("", queue, null, body.getBytes());

outstandingMessageCount++;

if (outstandingMessageCount == batchSize) {

ch.waitForConfirmsOrDie(5\_000);

outstandingMessageCount = 0;

}

}

if (outstandingMessageCount > 0) {

ch.waitForConfirmsOrDie(5\_000);

}

long end = System.nanoTime();

System.out.format("Published %,d messages in batch in %,d ms%n", MESSAGE\_COUNT, Duration.ofNanos(end - start).toMillis());

}

}

//策略三：一步确认消息

序列号：标识已确认或未确认消息的数字，确认时判断是否为相应的消息。

ch.getNextPublishSeqNo()

static void handlePublishConfirmsAsynchronously() throws Exception {

try (Connection connection = createConnection()) {

Channel ch = connection.createChannel();

String queue = UUID.randomUUID().toString();

ch.queueDeclare(queue, false, false, true, null);

ch.confirmSelect();

ConcurrentNavigableMap<Long, String> outstandingConfirms = new ConcurrentSkipListMap<>();

ConfirmCallback cleanOutstandingConfirms = (sequenceNumber, multiple) -> {

if (multiple) {

ConcurrentNavigableMap<Long, String> confirmed = outstandingConfirms.headMap(

sequenceNumber, true

);

confirmed.clear();

} else {

outstandingConfirms.remove(sequenceNumber);

}

};

ch.addConfirmListener(cleanOutstandingConfirms, (sequenceNumber, multiple) -> {

String body = outstandingConfirms.get(sequenceNumber);

System.err.format(

"Message with body %s has been nack-ed. Sequence number: %d, multiple: %b%n",

body, sequenceNumber, multiple

);

cleanOutstandingConfirms.handle(sequenceNumber, multiple);

});

long start = System.nanoTime();

for (int i = 0; i < MESSAGE\_COUNT; i++) {

String body = String.valueOf(i);

outstandingConfirms.put(ch.getNextPublishSeqNo(), body);

ch.basicPublish("", queue, null, body.getBytes());

}

if (!waitUntil(Duration.ofSeconds(60), () -> outstandingConfirms.isEmpty())) {

throw new IllegalStateException("All messages could not be confirmed in 60 seconds");

}

long end = System.nanoTime();

System.out.format("Published %,d messages and handled confirms asynchronously in %,d ms%n", MESSAGE\_COUNT, Duration.ofNanos(end - start).toMillis());

}

}

static boolean waitUntil(Duration timeout, BooleanSupplier condition) throws InterruptedException {

int waited = 0;

while (!condition.getAsBoolean() && waited < timeout.toMillis()) {

Thread.sleep(100L);

waited = +100;

}

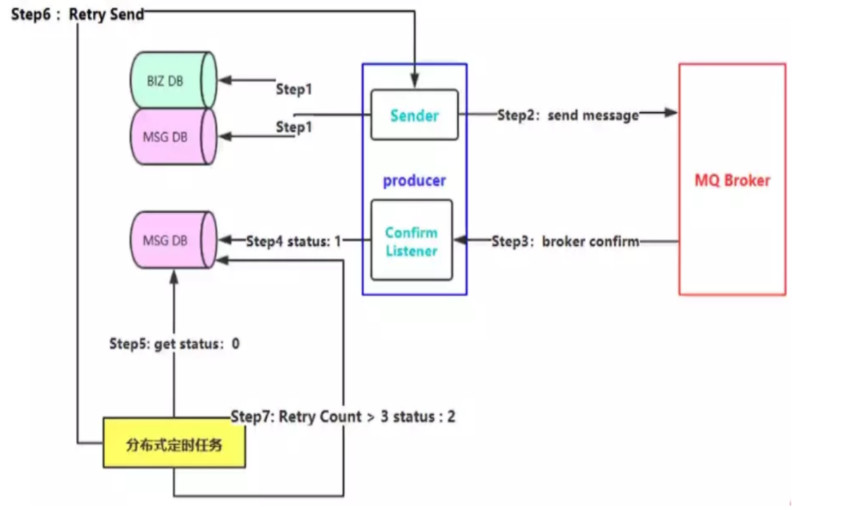
return condition.getAsBoolean();

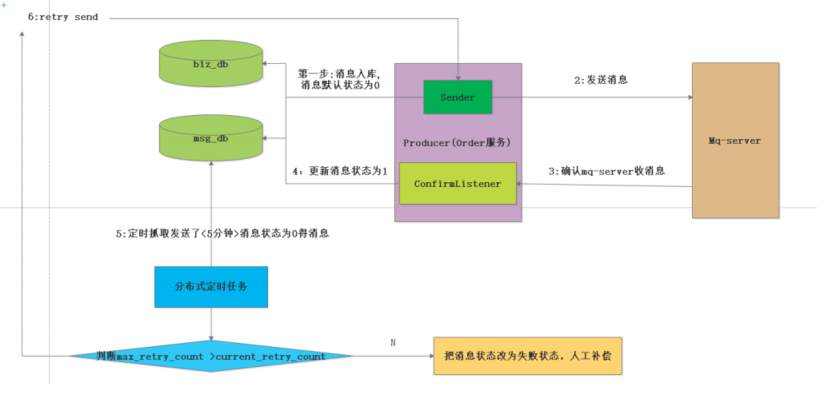
}

}

## 2.5 rabbitmq实现消息可靠性投递的两种方案

### 2.5.1 消息落库

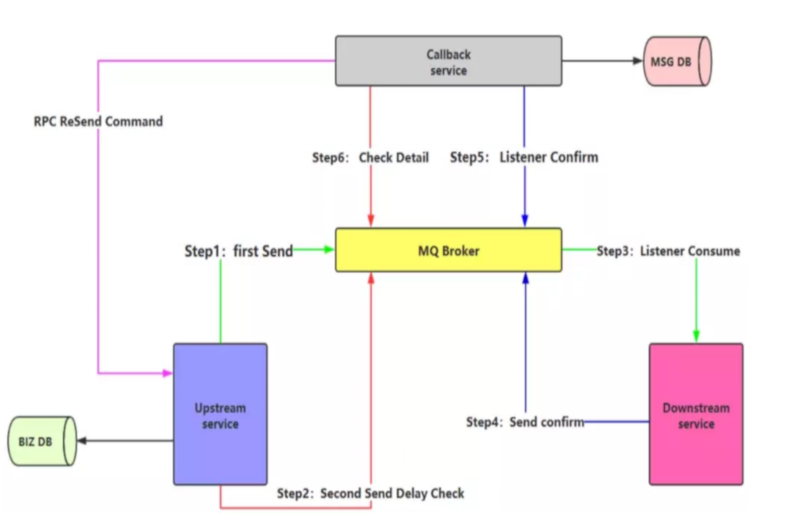




正常链路流程：  
第一步(该环节调用了操作了二次数据库):在创建订单的操作的时候， 把数据插入到订单相关的表中,并且构造调用物流模块的数据消息，把消息插入到消息表中,初始状态为0  
第二步: 把物流消息投递到消息队列中,  
第三步； 消息队列访问一个确认消息,并且由，订单服务来监控mq server的确认消息  
第四步:根据收到的确认消息来更新数据库中的消息记录的状态  
异常链路流程  
第一步(该环节调用了操作了二次数据库):在创建订单的操作的时候， 把数据插入到订单相关的表中,并且构造调用物流模块的数据消息， 把消息插入到消息表中,初始状态为0  
第二步: 把物流消息投递到消息队列中,  
第三步:由于网络闪断，导致消费端监控mq服务访问的确认消息没有收到，那么在msg\_db中的那条消息的状态永远就是0状态。 这个时候我们需要对这种情况下做出补偿  
补偿机制:  
启动一个分布式的定时任务,不定时的去扫描msg\_db的这个表， 状态为0的消息记录， 在这里我们可以根据业务来设置扫描重发规则  
规则1 :插入msg\_db 表中5Min后状态还是为0的记录，进行消息重试，使用redis分布式锁保证数据的幂等性  
规则2:若重试的次数超过五次状态还是为0的话， 我们就把消息状态改为2,此时我们需要人工的去确认状态为2的消息是什么原因导致没有成功的  
消息入库打标的缺点:  
在第一步的过程中，既插入了业务数据表，也同时插入了消息记录表，进行了二次db

操作，在高并发的环境下，这个环境就会造成性能瓶颈

### 2.5.2 延时投递，做二次确认检测



**正常的链路：**

第一步：在订单服务下单，保存订单数据到数据库，并发送订单消息与延迟消息

第二步：商品服务消费消息，进行手动签收，并发送确认消息到callback服务

第三步：订单服务接到商品服务对订单消息的确认，消息消费成功，如果ack为false则删除订单（正常业务将订单置为无效）

第四步：callback服务接收到商品服务的消息，将订单与消息的组合实体存库

第五步：callback服务接收到订单服务的延迟消息，根据订单号查询订单与消息的组合实体是否存在，存在则订单业务完成，不存在则调用RPC接口重新发送消息

**异常链路：**

如果因为网络抖动订单服务未接到确认消息，会将订单置为无效，当延迟队列消息被消费后如果订单未完结，即订单与消息的组合实体未入库，则重新调用重试接口发送消息，如果延迟队列消息未ack，则直接重发

问题：如果商品服务成功扣除库存后，发生网络抖动，则后续怎么办