# MessageQueue

1.从社区活跃度

按照目前网络上的资料，RabbitMQ、activeMQ、ZeroMQ三者中，综合来看，RabbitMQ是首选。

2.持久化消息比较

ZeroMq不支持，ActiveMq和RabbitMq都支持。

3.综合技术实现

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统等等。

RabbitMq/Kafka最好，ActiveMq次之，ZeroMq最差。当然ZeroMq也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

4.高并发

RabbitMQ最高，原因是它的实现语言是天生具备高并发高可用的erlang语言。

5.比较关注的比较，RabbitMQ和Kafka

RabbitMq比Kafka成熟，在可用性上，稳定性上，可靠性上[RabbitMq](https://www.sojson.com/tag_rabbitmq.html)胜于[Kafka](https://www.sojson.com/tag_kafka.html)（理论上）。

Kafka的定位主要在日志等方面， 因为Kafka设计的初衷就是处理日志的，可以看做是一个日志（消息）系统一个重要组件，针对性很强，所以业务方面建议选择RabbitMq。还有就是，Kafka的性能（吞吐量、TPS）比RabbitMq要高出来很多。

# Kafka

# RabbitMQ

## 基础概念

AMQP

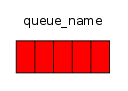
即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。 AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。 RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

ConnectionFactory、Connection、Channel

ConnectionFactory、Connection、Channel都是RabbitMQ对外提供的API中最基本的对象。Connection是RabbitMQ的socket链接，它封装了socket协议相关部分逻辑。ConnectionFactory为Connection的制造工厂。 Channel是我们与RabbitMQ打交道的最重要的一个接口，大部分的业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

Queue

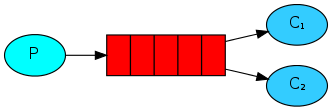
队列是RabbitMQ的内部对象，用于存储消息，用下图表示。



RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。

https://cdn.www.sojson.com/file/doc/1972176598

多个消费者可以订阅同一个Queue，这时Queue中的消息会被平均分摊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。



Message acknowledgment

在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。 这里会产生另外一个问题，如果开发时在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——Queue中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑…

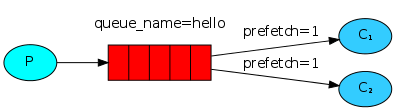
另外pub message是没有ack的。

Message durability

如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。由于这里仅为RabbitMQ的简单介绍，所以这里将不讲解RabbitMQ相关的事务。

Prefetch count

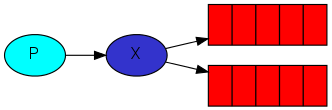
前面我们讲到如果有多个消费者同时订阅同一个Queue中的消息，Queue中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。我们可以通过设置prefetchCount来限制Queue每次发送给每个消费者的消息数，比如我们设置prefetchCount=1，则Queue每次给每个消费者发送一条消息；消费者处理完这条消息后Queue会再给该消费者发送一条消息。



Exchange

在上一节我们看到生产者将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。

在上一节我们看到生产者将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。



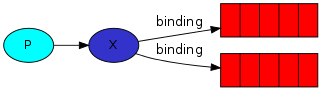
Exchange是按照什么逻辑将消息路由到Queue的？这个将在Binding一节介绍。 RabbitMQ中的Exchange有四种类型，不同的类型有着不同的路由策略，这将在Exchange Types一节介绍。

routing key

生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。 在Exchange Type与binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。 RabbitMQ为routing key设定的长度限制为255 bytes。

Binding

RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。



Binding key

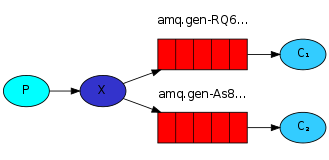
在绑定（Binding）Exchange与Queue的同时，一般会指定一个binding key；消费者将消息发送给Exchange时，一般会指定一个routing key；当binding key与routing key相匹配时，消息将会被路由到对应的Queue中。这个将在Exchange Types章节会列举实际的例子加以说明。 在绑定多个Queue到同一个Exchange的时候，这些Binding允许使用相同的binding key。 binding key 并不是在所有情况下都生效，它依赖于Exchange Type，比如fanout类型的Exchange就会无视binding key，而是将消息路由到所有绑定到该Exchange的Queue。

Exchange Types

RabbitMQ常用的Exchange Type有fanout、direct、topic、headers这四种（AMQP规范里还提到两种Exchange Type，分别为system与自定义，这里不予以描述），下面分别进行介绍。

fanout

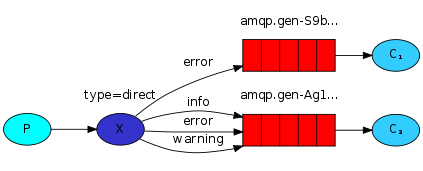
fanout类型的Exchange路由规则非常简单，它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。



上图中，生产者（P）发送到Exchange（X）的所有消息都会路由到图中的两个Queue，并最终被两个消费者（C1与C2）消费。

direct

direct类型的Exchange路由规则也很简单，它会把消息路由到那些binding key与routing key完全匹配的Queue中。

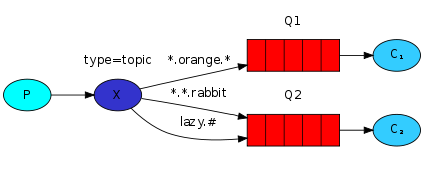


以上图的配置为例，我们以routingKey=”error”发送消息到Exchange，则消息会路由到Queue1（amqp.gen-S9b…，这是由RabbitMQ自动生成的Queue名称）和Queue2（amqp.gen-Agl…）；如果我们以routingKey=”info”或routingKey=”warning”来发送消息，则消息只会路由到Queue2。如果我们以其他routingKey发送消息，则消息不会路由到这两个Queue中。

topic

前面讲到direct类型的Exchange路由规则是完全匹配binding key与routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchage相似，也是将消息路由到binding key与routing key相匹配的Queue中，但这里的匹配规则有些不同，它约定：

* routing key为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”
* binding key与routing key一样也是句点号“. ”分隔的字符串
* binding key中可以存在两种特殊字符“\*”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）



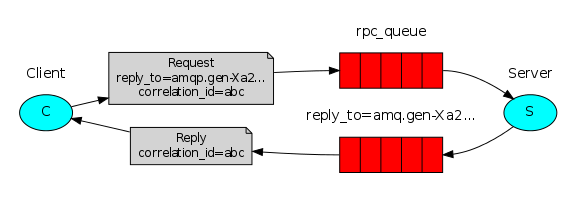
以上图中的配置为例，routingKey=”quick.orange.rabbit”的消息会同时路由到Q1与Q2，routingKey=”lazy.orange.fox”的消息会路由到Q1与Q2，routingKey=”lazy.brown.fox”的消息会路由到Q2，routingKey=”lazy.pink.rabbit”的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）；routingKey=”quick.brown.fox”、routingKey=”orange”、routingKey=”quick.orange.male.rabbit”的消息将会被丢弃，因为它们没有匹配任何bindingKey。

headers

headers类型的Exchange不依赖于routing key与binding key的匹配规则来路由消息，而是根据发送的消息内容中的headers属性进行匹配。 在绑定Queue与Exchange时指定一组键值对；当消息发送到Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。 该类型的Exchange没有用到过（不过也应该很有用武之地），所以不做介绍。

RPC

MQ本身是基于异步的消息处理，前面的示例中所有的生产者（P）将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。 但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。在RabbitMQ中也支持RPC。



[RabbitMQ](https://www.sojson.com/tag_rabbitmq.html)  中实现RPC 的机制是：

客户端发送请求（消息）时，在消息的属性（MessageProperties ，在AMQP 协议中定义了14中properties ，这些属性会随着消息一起发送）中设置两个值replyTo （一个Queue 名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue 中）和correlationId （此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）

服务器端收到消息并处理

服务器端处理完消息后，将生成一条应答消息到replyTo 指定的Queue ，同时带上correlationId 属性

客户端之前已订阅replyTo 指定的Queue ，从中收到服务器的应答消息后，根据其中的correlationId 属性分析哪条请求被执行了，根据执行结果进行后续业务处理

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl add\_user root root

'sudo' 不是内部或外部命令，也不是可运行的程序

或批处理文件。

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl add\_user root root

Adding user "root" ...

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl set\_user\_tags root administrator

Setting tags for user "root" to [administrator] ...

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl set\_permissions -p / root "." "." ".\*"

Setting permissions for user "root" in vhost "/" ...

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl list\_user\_permissions username

Listing permissions for user "username" ...

Error:

{:no\_such\_user, "username"}

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl list\_user\_permissions root

Listing permissions for user "root" ...

vhost configure write read

/ . . .\*

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl list\_users

Listing users ...

user tags

guest [administrator]

root [administrator]

D:\software\rabbitmq\rabbitmq\_server-3.8.2\sbin>rabbitmqctl delete\_user guest

Deleting user "guest" ...

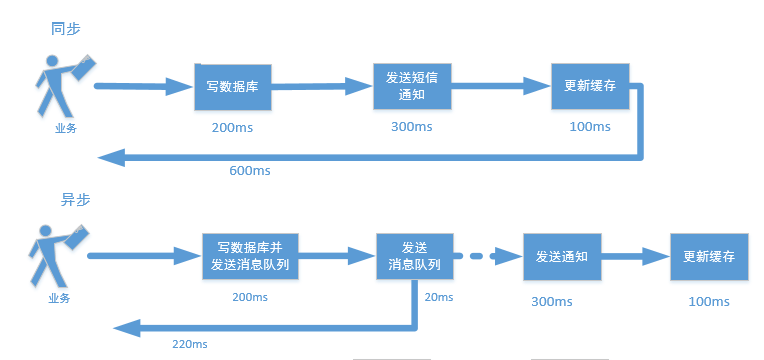
# ActiveMQ

## 消息队列技术

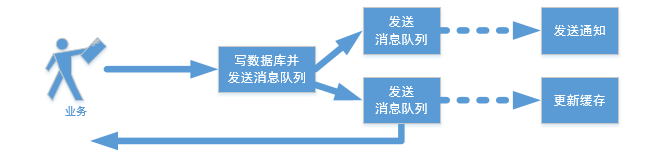
消息队列，也叫消息中间件。消息的传输过程中保存消息的容器。

### 消息队列技术解决的问题

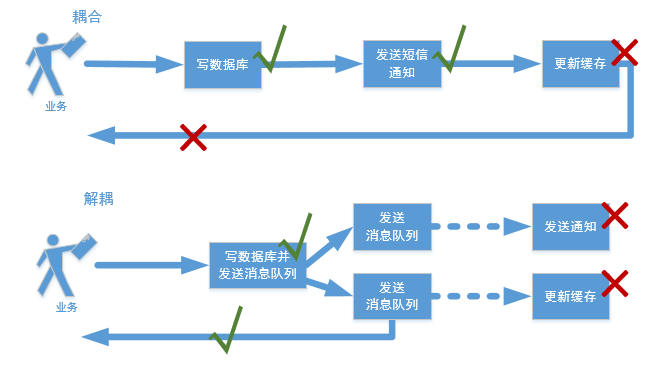
**异步**



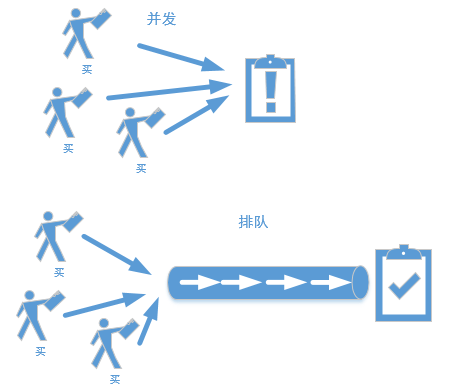
**并行**



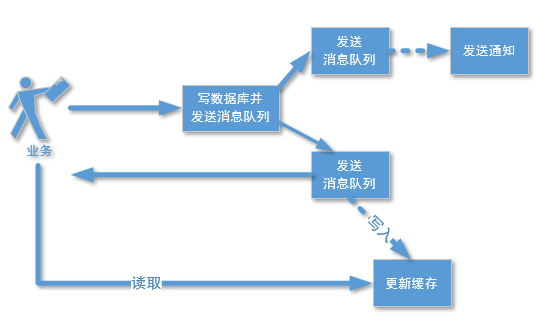
**解耦**



**排队**



**弊端**：不确定性和延迟，解决这个弊端的方式是保持最终一致性。



### 消息模式

**点对点**

Producer

Consumer

Queue

**订阅**

Producer

Topic

Consumer

Consumer

Consumer

### ActiveMQ



ActiveMQ对于处理10万条消息队列以上的数据时相对于其他消息队列工具效率较高。

同类产品： RabbitMQ 、 Kafka、Redis（List）

**对比RabbitMQ**

最接近的同类型产品，性能伯仲之间，基本上可以互相替代。最主要区别是二者的协议不同，RabbitMQ的协议是AMQP(Advanced Message Queueing Protoco)，而ActiveMQ使用的是JMS(Java Messaging Service )协议。顾名思义JMS是针对Java体系的传输协议，队列两端必须有JVM，所以如果开发环境都是java的话推荐使用ActiveMQ，可以用Java的一些对象进行传递比如Map、BLob、Stream等。而AMQP通用行较强，非java环境经常使用，传输内容就是标准字符串。

另外一点就是RabbitMQ用Erlang开发，安装前要装Erlang环境，比较麻烦。ActiveMQ解压即可用不用任何安装。

**对比Kafka**

Kafka性能超过ActiveMQ等传统MQ工具，集群扩展性好。

**弊端是：**

在传输过程中可能会出现消息重复的情况，

不保证发送顺序

一些传统MQ的功能没有，比如消息的事务功能。

所以通常用Kafka处理大数据日志。

**对比Redis**

其实Redis本身利用List可以实现消息队列的功能，但是功能很少，而且队列体积较大时性能会急剧下降。对于数据量不大、业务简单的场景可以使用。

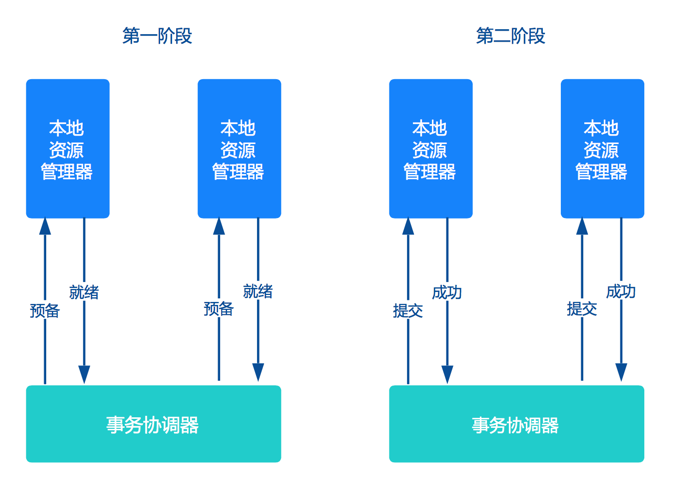
rabbitmq是使用C语言编写的MQ；kafka是大数据常用的MQ；zeroMQ是简单版的MQ；mateMQ是基于ActiveMQ的MQ，RocketMQ是阿里编写的MQ。

### 分布式事务

分布式事务指的是在分布式的环境下保证数据的一致性。现在一般采用基于消息队列的事务，即使用消息来统一管理事务的执行和回滚，从而追求最终一致性。（在某些服务回滚时，采取补偿型策略）。

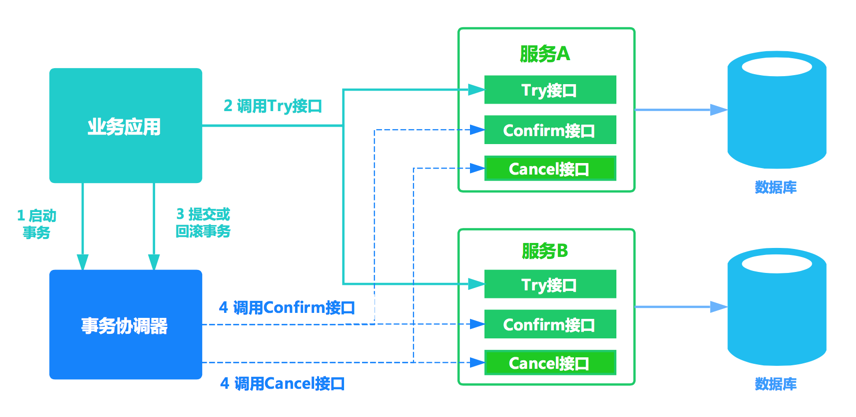
**基于xa的两段式提交**

事务协调器，统一管理事务的提交或者回滚。



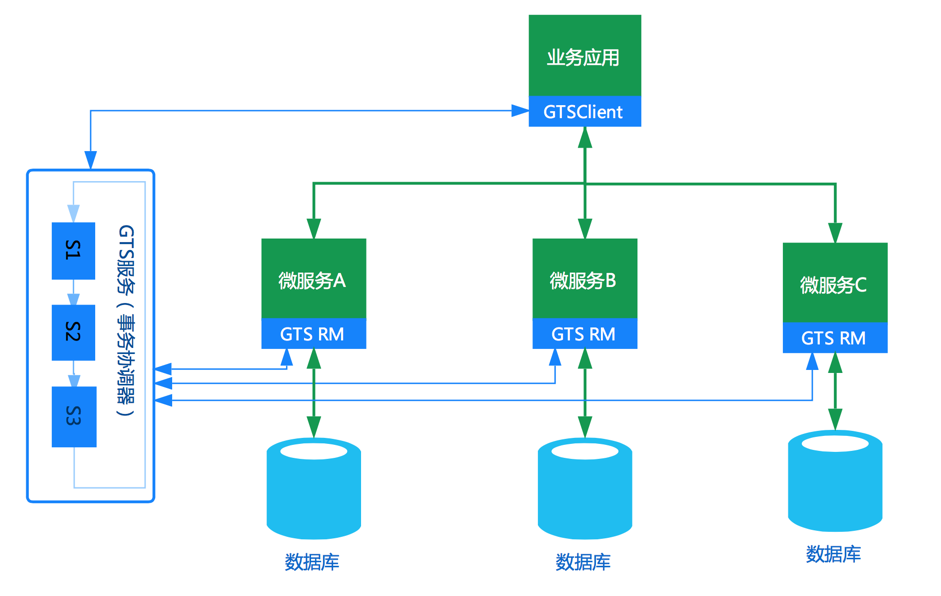
**两段式提交升级版tcc**

tcc是两段式提交的进化版，在业务中实现了提交的接口，但是缺点在于入侵性强。



**阿里的gts**

gts是阿里封装的基于tcc的分布式事务框架。



## 在Linux上的安装

### 安装ActiveMQ

1、上传ActiveMQ的tar包到Linux系统，并解压到/opt目录下

2、修改/conf目录下的activemq.xml，添加开启消息延迟的功能

|  |
| --- |
| <broker  xmlns="http://activemq.apache.org/schema/core"  schedulerSupport="true"  brokerName="localhost"  dataDirectory="${activemq.data}"> |

3、启动ActiveMQ并查看相关信息

|  |
| --- |
| [root@xmm bin]# ./activemq start  INFO: Loading '/opt/apache-activemq-5.14.4//bin/env'  INFO: Using java '/opt/jdk1.8.0\_121/bin/java'  INFO: Starting - inspect logfiles specified in logging.properties and log4j.properties to get details  INFO: pidfile created : '/opt/apache-activemq-5.14.4//data/activemq.pid' (pid '3790')  [root@xmm bin]# netstat -anp | grep 3790  tcp6 0 0 :::61613 :::\* LISTEN 3790/java  tcp6 0 0 :::61614 :::\* LISTEN 3790/java  tcp6 0 0 :::61616 :::\* LISTEN 3790/java  tcp6 0 0 :::1883 :::\* LISTEN 3790/java  tcp6 0 0 :::8161 :::\* LISTEN 3790/java  tcp6 0 0 :::5672 :::\* LISTEN 3790/java  tcp6 0 0 :::38697 :::\* LISTEN 3790/java  unix 2 [ ] STREAM CONNECTED 39534 3790/java  unix 2 [ ] STREAM CONNECTED 40353 3790/java |

activemq两个重要的端口，一个是提供消息队列的默认端口：61616 ，另一个是控制台端口8161。启动后在浏览器中输入IP:8161进行访问ActiveMQ。

4、创建软连接并注册服务

|  |
| --- |
| [root@xmm bin]# ln -s /opt/apache-activemq-5.14.4/bin/activemq /etc/init.d/activemq  [root@xmm init.d]# chkconfig --add activemq  [root@xmm init.d]# chkconfig activemq on |

5、配置开机启动

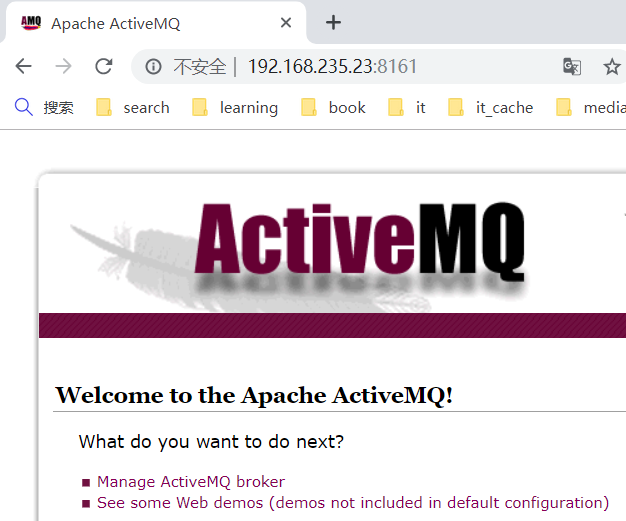
|  |
| --- |
| [root@xmm init.d]# chkconfig activemq on  [root@xmm init.d]# reboot |

### 通过控制台测试

1、启动consumer消费端

|  |
| --- |
| [root@xmm init.d]# service activemq consumer  INFO: Loading '/opt/apache-activemq-5.14.4//bin/env'  INFO: Using java '/bin/java'  Java Runtime: Oracle Corporation 1.8.0\_131 /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.131-11.b12.el7.x86\_64/jre  Heap sizes: current=62976k free=61992k max=932352k  JVM args: -Xms64M -Xmx1G -Djava.util.logging.config.file=logging.properties -Djava.security.auth.login.config=/opt/apache-activemq-5.14.4//conf/login.config -Dactivemq.classpath=/opt/apache-activemq-5.14.4//conf:/opt/apache-activemq-5.14.4//../lib/: -Dactivemq.home=/opt/apache-activemq-5.14.4/ -Dactivemq.base=/opt/apache-activemq-5.14.4/ -Dactivemq.conf=/opt/apache-activemq-5.14.4//conf -Dactivemq.data=/opt/apache-activemq-5.14.4//data  Extensions classpath:  [/opt/apache-activemq-5.14.4/lib,/opt/apache-activemq-5.14.4/lib/camel,/opt/apache-activemq-5.14.4/lib/optional,/opt/apache-activemq-5.14.4/lib/web,/opt/apache-activemq-5.14.4/lib/extra]  ACTIVEMQ\_HOME: /opt/apache-activemq-5.14.4  ACTIVEMQ\_BASE: /opt/apache-activemq-5.14.4  ACTIVEMQ\_CONF: /opt/apache-activemq-5.14.4/conf  ACTIVEMQ\_DATA: /opt/apache-activemq-5.14.4/data  INFO | Connecting to URL: failover://tcp://localhost:61616 (null:null)  INFO | Consuming queue://TEST  INFO | Sleeping between receives 0 ms  INFO | Running 1 parallel threads  INFO | Successfully connected to tcp://localhost:61616  INFO | consumer-1 wait until 1000 messages are consumed |

2、在浏览器地址栏中输入虚拟机IP和ActiveMQ的端口号8161进入控制台，账号和密码默认均为admin



Manage ActiveMQ broker→Queues→Operations→Send To→Message body中输入队列消息→send

3、在consumer中观察接收到的消息

|  |
| --- |
| INFO | consumer-1 Received Hello world  INFO | consumer-1 Received Enter some text here for the message body...xmm  INFO | consumer-1 Received 1  INFO | consumer-1 Received 2  INFO | consumer-1 Received 3 |

## ActiveMQ概念

### 事务控制

|  |  |  |
| --- | --- | --- |
| producer提交时  的事务 | 事务开启 | 只执行send并不会提交到队列中，只有当执行session.commit()时，消息才被真正的提交到队列中。 |
| 事务不开启 | 只要执行send，就进入到队列中。 |
| consumer 接收时  的事务 | 事务开启，签收必须写  Session.SESSION\_TRANSACTED | 收到消息后，消息并没有真正的被消费。消息只是被锁住。一旦出现该线程死掉、抛异常，或者程序执行了session.rollback()那么消息会释放，重新回到队列中被别的消费端再次消费。 |
| 事务不开启，签收方式选择  Session.AUTO\_ACKNOWLEDGE | 只要调用comsumer.receive方法 ，自动确认。 |
| 事务不开启，签收方式选择  Session.CLIENT\_ACKNOWLEDGE | 需要客户端执行 message.acknowledge(),否则视为未提交状态，线程结束后，其他线程还可以接收到。这种方式跟事务模式很像，区别是不能手动回滚,而且可以单独确认某个消息。 |
| 事务不开启，签收方式选择  Session.DUPS\_OK\_ACKNOWLEDGE | 在Topic模式下做批量签收时用的，可以提高性能。但是某些情况消息可能会被重复提交，使用这种模式的consumer要可以处理重复提交的问题。 |

### 持久化与非持久化

通过producer.setDeliveryMode(DeliveryMode.PERSISTENT) 进行设置

持久化的好处就是当activemq宕机的话，消息队列中的消息不会丢失，但是会消耗一定的性能。非持久化会丢失。

## 整合到Java项目

### ActiveMQConfig

|  |
| --- |
| @Configuration public class ActiveMQConfig {  @Value("${spring.activemq.broker-url:disabled}")  String brokerURL;  @Value("${activemq.listener.enable:disabled}")  String listenerEnable;  @Bean  public ActiveMQUtil getActiveMQUtil(){  if("disabled".equals(brokerURL)){  return null;  }  ActiveMQUtil activeMQUtil = new ActiveMQUtil();  activeMQUtil.init(brokerURL);  return activeMQUtil;  }  //定义一个消息监听器连接工厂，这里定义的是点对点模式的监听器连接工厂  @Bean(name = "jmsQueueListener")  public DefaultJmsListenerContainerFactory jmsQueueListenerContainerFactory(  ActiveMQConnectionFactory activeMQConnectionFactory){  DefaultJmsListenerContainerFactory factory =  new DefaultJmsListenerContainerFactory();  if(!"true".equals(listenerEnable)){  return null;  }  factory.setConnectionFactory(activeMQConnectionFactory);  factory.setConcurrency("5");  factory.setRecoveryInterval(5000L);  factory.setSessionTransacted(false);  factory.setSessionAcknowledgeMode(Session.*CLIENT\_ACKNOWLEDGE*);  return factory;  }  @Bean  public ActiveMQConnectionFactory activeMQConnectionFactory ( ){  ActiveMQConnectionFactory activeMQConnectionFactory =  new ActiveMQConnectionFactory(brokerURL);  return activeMQConnectionFactory;  } } |

### ActiveMQUtil

|  |
| --- |
| public class ActiveMQUtil {  PooledConnectionFactory pooledConnectionFactory = null;  public ConnectionFactory init(String brokerUrl){  ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(brokerUrl);  pooledConnectionFactory = new PooledConnectionFactory(factory);  pooledConnectionFactory.setReconnectOnException(true);  pooledConnectionFactory.setMaxConnections(5);  pooledConnectionFactory.setExpiryTimeout(10800);  return pooledConnectionFactory;  }  public ConnectionFactory getConnectionFactory(){  return pooledConnectionFactory;  } } |

### SpringBoot配置

|  |
| --- |
| spring.activemq.broker-url=tcp://192.168.235.23:61616 activemq.listener.enable=true |

### ActiveMQ相关依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-activemq</artifactId>  <exclusions>  <exclusion>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  </exclusion>  </exclusions> </dependency> <dependency>  <groupId>org.apache.activemq</groupId>  <artifactId>activemq-pool</artifactId>  <version>5.15.2</version>  <exclusions>  <exclusion>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  </exclusion>  </exclusions> </dependency> |

## 消息队列的不确定性

使用分布式事务异步通信会有很大的不确定性，Producer发送消息后不会原地等待Consumer返回回执消息，而且Consumer也可能发送消息失败，这时就需要对这种不确定性进行补充。

1、Producer在发送消息时设置一个时间值，超过这个时间时就询问Consumer是否收到消息，这种方式称之为延迟队列。

2、Producer发送消息后设置一个时间点，每到这个时间点时就查看所有以发送的且没有获取到回执消息的消息，并一一查询Consumer，这种方式称之为定时扫描。

延迟队列更加精准，但是如果周期过长，就会导致队列十分冗长，因此延迟队列可以用于核查支付宝支付功能是否完成；定时扫描更适合于周期较长且不需要精确到分秒级别的事件，比如30分支未付款订单就关闭。

如果要开启延迟消息队列功能首先需要在解压目录下的conf目录下修改配置文件activemq.xml，从而开启消息延迟功能。

### 延迟队列

#### 应用场景

当用户选择支付后，通常来说用户都会在支付宝正常支付，支付宝转账成功后，通过后台异步发送成功的请求到电商支付模块。

但是如果用户点击支付后，支付模块可能会长时间没有收到支付宝的支付成功通知。这种情况会‘有两种可能性，一种是用户在弹出支付宝付款界面时没有继续支付，另一种就是用户支付成功了，但是因为网络等各种问题，支付模块没有收到通知。

如果是上述第二种可能性，对于用户来说体验是非常糟糕的，甚至会怀疑平台的诚信。

所以为了尽可能避免第二种情况，在用户点击支付后一段时间后，不管用户是否付款，都要去主动询问支付宝，该笔单据是否付款。

#### 实现思路

首先，需要知道如何主动查询支付宝中某笔交易的状态。

支付宝查询接口文档：https://docs.open.alipay.com/api\_1/alipay.trade.query

其次，利用延迟队列反复调用。

#### 实现支付宝订单状态查询

支付宝文档中的样例



Out\_trade\_no ：第三方交易编号

Trade\_no ：支付宝回执的编号，由于没有外网我们得不到支付宝回执编号

所以在测试的时候只用out\_trade\_no.

1、首先通过基本参数初始化AlipayClient,此处和支付模块部分相同，不再详述。

2、业务参数



业务参数就两个，选哪个都可以，其中out\_trade\_no是电商系统生成的，trade\_no是支付宝回调后产生的。因为有可能一直就没收到支付宝的回调，也就没有trade\_no，所以咱们这里使用out\_trade\_no。

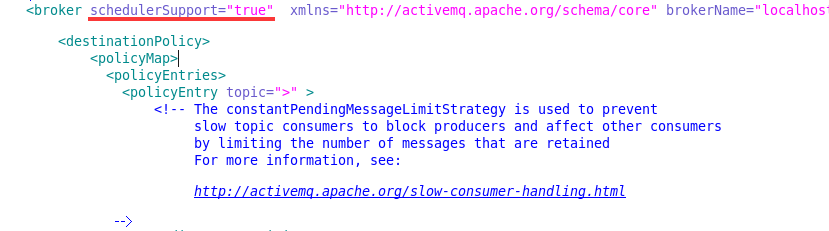
#### 利用延迟队列反复调用查询接口。

执行策略：

选择支付渠道后，点击支付后提交到延迟队列，每隔一分钟执行一次查询操作，查询三次。

首先在消息队列中打开延迟队列配置：在activemq的conf目录下activemq.xml中

# vim /opt/activemq/conf/activemq.xml



开启 schedulerSupport="true"

配置完成，要重启

#service activemq restart

### 定时扫描

#### 应用场景

长期没有付款的订单，要定期关闭掉。

如果时限比较小，比如30分钟未付款的订单就关闭（一般是锁了库存的订单），也可以用延时队列解决。

如果时限比较长比如1-2天，可以选择用轮询扫描。

#### 实现方式 spring task

轮询扫描有很多工具，比较经典的就是 quartz。

但是springboot整合了自家的spring task ，功能上基本和quartz差不多，但是配置更简单，全程只用注解就可以，不用额外的xml。

#### 利用多线程实现异步并发操作

默认扫描是单线程的即一次任务执行完，第二次的任务才能执行。如果第一次的任务被一些其他情况阻塞住了，那么第二次的扫描就没法开始了。