**每特学院 Java互联网架构培训-www.itmayiedu.com 余胜军老师QQ644064779**

**ActiveMQ**

# 消息中间件概述

## 1.1消息中间件产生的背景

在客户端与服务器进行通讯时.客户端调用后，必须等待服务对象完成处理返回结果才能继续执行。

**客户与服务器对象的生命周期紧密耦合,客户进程和服务对象进程都都必须正常运行;如果由于服务对象崩溃或者网络故障导致用户的请求不可达,客户会受到异常**

**点对点通信: 客户的一次调用只发送给某个单独的目标对象。**

(画图演示)

## 1.2 什么是消息中间件

面向消息的中间件(MessageOrlented MiddlewareMOM)较好的解决了以上问  
题。发送者将消息发送给消息服务器，消息服务器将消感存放在若千队列中，在合适  
的时候再将消息转发给接收者。

这种模式下，发送和接收是异步的，发送者无需等  
待; 二者的生命周期未必相同: 发送消息的时候接收者不一定运行，接收消息的时候  
发送者也不一定运行;一对多通信: 对于一个消息可以有多个接收者。

# 二 、JMS介绍

## 2.1 什么是JMS？

JMS是java的消息服务，JMS的客户端之间可以通过JMS服务进行异步的消息传输。

## 2.2 什么是消息模型

|  |
| --- |
| ○ Point-to-Point(P2P) --- 点对点  ○ Publish/Subscribe(Pub/Sub)--- 发布订阅 |

即点对点和发布订阅模型

### 2.2.1 P2P (点对点)

P2P

1. P2P模式图   
   
2. 涉及到的概念   
   1. 消息队列（Queue）
   2. 发送者(Sender)
   3. 接收者(Receiver)
   4. 每个消息都被发送到一个特定的队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。
3. P2P的特点
   1. 每个消息只有一个消费者（Consumer）(即一旦被消费，消息就不再在消息队列中)
   2. 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列
   3. 接收者在成功接收消息之后需向队列应答成功

如果你希望发送的每个消息都应该被成功处理的话，那么你需要P2P模式。

#### 应用场景

A用户与B用户发送消息

### 2.2.2Pub/Sub (发布与订阅)

Pub/Sub模式图   


涉及到的概念 

主题（Topic）

发布者（Publisher）

订阅者（Subscriber）   
客户端将消息发送到主题。多个发布者将消息发送到Topic,系统将这些消息传递给多个订阅者。

Pub/Sub的特点

每个消息可以有多个消费者

发布者和订阅者之间有时间上的依赖性。针对某个主题（Topic）的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息，而且为了消费消息，订阅者必须保持运行的状态。

为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息。

如果你希望发送的消息可以不被做任何处理、或者被一个消息者处理、或者可以被多个消费者处理的话，那么可以采用Pub/Sub模型

消息的消费   
在JMS中，消息的产生和消息是异步的。对于消费来说，JMS的消息者可以通过两种方式来消费消息。   
○ 同步   
订阅者或接收者调用receive方法来接收消息，receive方法在能够接收到消息之前（或超时之前）将一直阻塞   
○ 异步   
订阅者或接收者可以注册为一个消息监听器。当消息到达之后，系统自动调用监听器的onMessage方法。

#### 应用场景:

用户注册、订单修改库存、日志存储

画图演示







# 三、 MQ产品的分类

## RabbitMQ

是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正是如此，使的它变的非常重量级，更适合于企业级的开发。同时实现了一个经纪人(Broker)构架，这意味着消息在发送给客户端时先在中心队列排队。对路由(Routing)，负载均衡(Load balance)或者数据持久化都有很好的支持。

## Redis

是一个Key-Value的NoSQL数据库，开发维护很活跃，虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 入队 | | | | 出队 | | | |
|  | 128B | 512B | 1K | 10K | 128B | 512B | 1K | 10K |
| Redis | 16088 | 15961 | 17094 | 25 | 15955 | 20449 | 18098 | 9355 |
| RabbitMQ | 10627 | 9916 | 9370 | 2366 | 3219 | 3174 | 2982 | 1588 |

## ZeroMQ

号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果down机，数据将会丢失。其中，Twitter的Storm中使用ZeroMQ作为数据流的传输。

## ActiveMQ

是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。RabbitMQ、ZeroMQ、ActiveMQ均支持常用的多种语言客户端 C++、Java、.Net,、Python、 Php、 Ruby等。

## Jafka/Kafka

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式Publish/Subscribe消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现复杂均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制来统一了在线和离线的消息处理，这一点也是本课题所研究系统所看重的。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

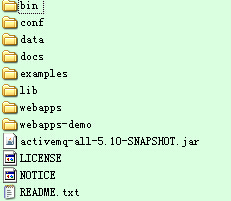
其他一些队列列表HornetQ、Apache Qpid、Sparrow、Starling、Kestrel、Beanstalkd、Amazon SQS就不再一一分析。

# 四、 ActiveMQ使用

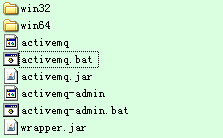
## 4.1 、window下 ActiveMQ安装

ActiveMQ部署其实很简单，和所有Java一样，要跑java程序就必须先安装JDK并配置好环境变量，这个很简单。

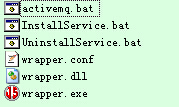
然后解压下载的apache-activemq-5.10-20140603.133406-78-bin.zip压缩包到一个目录，得到解压后的目录结构如下图：



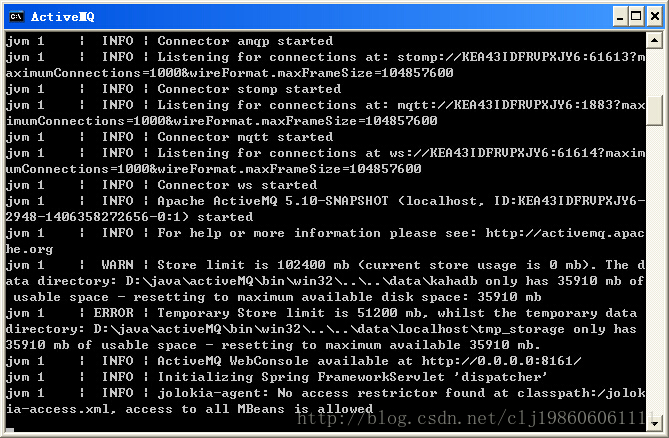
进入bin目录，发现有win32和win64两个文件夹，这2个文件夹分别对应windows32位和windows64位操作系统的启动脚本。



我的实验环境是windowsXP，就进入win32目录，会看到如下目录结构。



其中activemq.bat便是启动脚本，双击启动。



ActiveMQ默认启动到8161端口，启动完了后在浏览器地址栏输入：<http://localhost:8161/admin>要求输入用户名密码，默认用户名密码为admin、admin，这个用户名密码是在conf/users.properties中配置的。输入用户名密码后便可看到如下图的ActiveMQ控制台界面了。

### 4.1.1控制台介绍

Number Of Consumers  消费者 这个是消费者端的消费者数量   
Number Of Pending Messages 等待消费的消息 这个是当前未出队列的数量。可以理解为总接收数-总出队列数   
Messages Enqueued 进入队列的消息  进入队列的总数量,包括出队列的。 这个数量只增不减   
Messages Dequeued 出了队列的消息  可以理解为是消费这消费掉的数量   
这个要分两种情况理解   
在queues里它和进入队列的总数量相等(因为一个消息只会被成功消费一次),如果暂时不等是因为消费者还没来得及消费。   
在 topics里 它因为多消费者从而导致数量会比入队列数高。   
简单的理解上面的意思就是   
当有一个消息进入这个队列时，等待消费的消息是1，进入队列的消息是1。   
当消息消费后，等待消费的消息是0，进入队列的消息是1，出队列的消息是1.   
在来一条消息时，等待消费的消息是1，进入队列的消息就是2.   
  
  
没有消费者时  Pending Messages   和 入队列数量一样   
有消费者消费的时候 Pedding会减少 出队列会增加   
到最后 就是 入队列和出队列的数量一样多   
以此类推，进入队列的消息和出队列的消息是池子，等待消费的消息是水流。

## 4.2 、实现点对点通讯模式

使用ActiveMQ完成点对点（p2p）通讯模式

引入pom文件依赖

|  |
| --- |
| **<dependencies>**  **<dependency>**  **<groupId>org.apache.activemq</groupId>**  **<artifactId>activemq-core</artifactId>**  **<version>5.7.0</version>**  **</dependency>**  **</dependencies>** |

#### 生产者

|  |
| --- |
| /\*\*  \* mq通讯地址  \*/  **private** **final** **static** String ***URL*** = "tcp://localhost:61616";  /\*\*  \* 队列名称  \*/  **private** **final** **static** String ***QUEUENAME*** = "my\_queue";  **public** **static** **void** main(String[] args) **throws** JMSException {  // 1.创建ActiveMQFactory  ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory(***URL***);  // 2.创建连接  Connection cnnection = factory.createConnection();  // 3.启动连接  cnnection.start();  // 4.创建Session 不开启事务,自动签收模式  Session session = cnnection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 5.创建一个目标  Queue queue = session.createQueue(***QUEUENAME***);  // 6.创建生产者  MessageProducer producer = session.createProducer(queue);  **for** (**int** i = 1; i <= 10; i++) {  // 7.创建消息  TextMessage textMessage = session.createTextMessage("消息" + i);  // 8.发送消息  producer.send(textMessage);  System.***out***.println(textMessage.toString());  }  // 9.关闭连接  cnnection.close();  } |

#### 消费者

|  |
| --- |
| **public** **class** Consumer {  /\*\*  \* mq通讯地址  \*/  **private** **final** **static** String ***URL*** = "tcp://localhost:61616";  /\*\*  \* 队列名称  \*/  **private** **final** **static** String ***QUEUENAME*** = "my\_queue";  **public** **static** **void** main(String[] args) **throws** JMSException {  // 1.创建ActiveMQFactory  ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory(***URL***);  // 2.创建连接  Connection cnnection = factory.createConnection();  // 3.启动连接  cnnection.start();  // 4.创建Session 不开启事务,自动签收模式  Session session = cnnection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 5.创建一个目标  Queue queue = session.createQueue(***QUEUENAME***);  // 6.创建生产者  MessageConsumer createConsumer = session.createConsumer(queue);  createConsumer.setMessageListener(**new** MessageListener() {  **public** **void** onMessage(Message message) {  **try** {  TextMessage textMessage = (TextMessage) message;  System.***out***.println("消费者消费消息:" + textMessage.getText());  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  });  // 先不要关闭连接  }  } |

## 4.3 、JMS消息可靠机制

ActiveMQ消息签收机制：

客戶端成功接收一条消息的标志是一条消息被签收，成功应答。

消息的签收情形分两种：

1、带事务的session

   如果session带有事务，并且事务成功提交，则消息被自动签收。如果事务回滚，则消息会被再次传送。

2、不带事务的session

   不带事务的session的签收方式，取决于session的配置。

   Activemq支持一下三種模式：

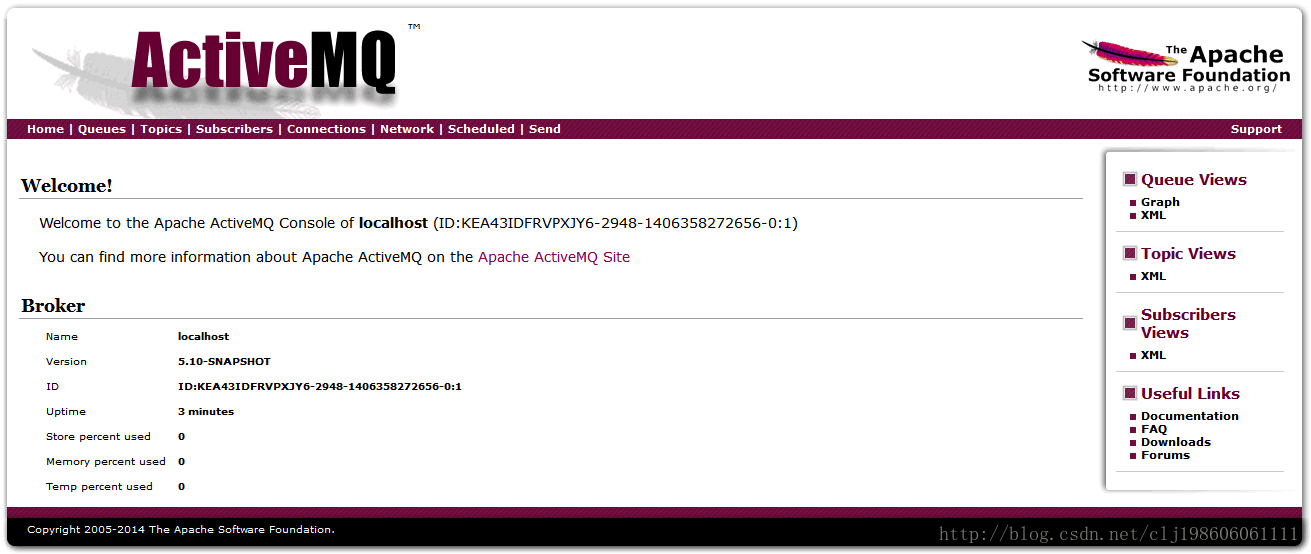
   Session.AUTO\_ACKNOWLEDGE  消息自动签收

   Session.CLIENT\_ACKNOWLEDGE  客戶端调用acknowledge方法手动签收

textMessage.acknowledge();//手动签收

   Session.DUPS\_OK\_ACKNOWLEDGE 不是必须签收，消息可能会重复发送。在第二次重新传送消息的时候，消息

只有在被确认之后，才认为已经被成功地消费了。消息的成功消费通常包含三个阶段：客户接收消息、客户处理消息和消息被确认。 在事务性会话中，当一个事务被提交的时候，确认自动发生。在非事务性会话中，消息何时被确认取决于创建会话时的应答模式（acknowledgement mode）。该参数有以下三个可选值：



Number Of Consumers  消费者 这个是消费者端的消费者数量   
  
Number Of Pending Messages 等待消费的消息 这个是当前未出队列的数量。可以理解为总接收数-总出队列数   
Messages Enqueued 进入队列的消息  进入队列的总数量,包括出队列的。 这个数量只增不减   
Messages Dequeued 出了队列的消息  可以理解为是消费这消费掉的数量

## 4.4 、发布订阅

### 生产者:

|  |
| --- |
| **public** **class** Producter2 {  /\*\*  \* mq通讯地址  \*/  **private** **final** **static** String ***URL*** = "tcp://localhost:61616";  /\*\*  \* 主题名称  \*/  **private** **final** **static** String ***TOPICNAME*** = "my\_topic";  **public** **static** **void** main(String[] args) **throws** JMSException {  // 1.创建ActiveMQFactory  ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory(***URL***);  // 2.创建连接  Connection cnnection = factory.createConnection();  // 3.启动连接  cnnection.start();  // 4.创建Session 不开启事务,自动签收模式  Session session = cnnection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 5.创建一个目标  Topic createTopic = session.createTopic(***TOPICNAME***);  // 6.创建生产者  MessageProducer producer = session.createProducer(createTopic);  // 设置消息持久化  producer.setDeliveryMode(DeliveryMode.***PERSISTENT***);  **for** (**int** i = 1; i <= 10; i++) {  // 7.创建消息  TextMessage textMessage = session.createTextMessage("消息" + i);  // 8.发送消息  producer.send(textMessage);  System.***out***.println(textMessage.toString());  }  // 9.关闭连接  cnnection.close();  }  } |

### 消费者:

|  |
| --- |
| **public** **class** Consumer2 {  /\*\*  \* mq通讯地址  \*/  **private** **final** **static** String ***URL*** = "tcp://localhost:61616";  /\*\*  \* 主题名称  \*/  **private** **final** **static** String ***TOPICNAME*** = "my\_topic";  **public** **static** **void** main(String[] args) **throws** JMSException {  // 1.创建ActiveMQFactory  ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory(***URL***);  // 2.创建连接  Connection cnnection = factory.createConnection();  // 3.启动连接  cnnection.start();  // 4.创建Session 不开启事务,自动签收模式  Session session = cnnection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 5.创建一个目标  Topic topic = session.createTopic(***TOPICNAME***);  // 6.创建生产者  MessageConsumer createConsumer = session.createConsumer(topic);  createConsumer.setMessageListener(**new** MessageListener() {  **public** **void** onMessage(Message message) {  **try** {  TextMessage textMessage = (TextMessage) message;  System.***out***.println("消费者消费消息:" + textMessage.getText());  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  });  // 先不要关闭连接  }  } |

# 4.5 、SpringBoot整合ActiveMQ

## 生产者:

### 4.5.1 引入 maven依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.1.RELEASE</version>  </parent>  <!-- 管理依赖 -->  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Finchley.M7</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <!-- SpringBoot整合Web组件 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- SpringBoot Activemq -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-activemq</artifactId>  </dependency>  </dependencies>  <!-- 注意： 这里必须要添加， 否者各种依赖有问题 -->  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/libs-milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

### 4.5.2 引入 application.yml配置

|  |
| --- |
| spring:  activemq:  broker-url: tcp://127.0.0.1:61616  user: admin  password: admin  queue: springboot-queue  server:  port: 8080 |

### 4.5.3 创建QueueConfig

|  |
| --- |
| **@Configuration**  **public class QueueConfig {**  **@Value("${queue}")**  **private String queue;**  **@Bean**  **public Queue logQueue() {**  **return new ActiveMQQueue(queue);**  **}**  **}** |

### 4.5.4 创建Producer

|  |
| --- |
| **@Component**  **@EnableScheduling**  **public class Producer {**  **@Autowired**  **private JmsMessagingTemplate jmsMessagingTemplate;**  **@Autowired**  **private Queue queue;**  **@Scheduled(fixedDelay = 5000)**  **public void send() {**  **jmsMessagingTemplate.convertAndSend(queue, "测试消息队列" + System.*currentTimeMillis*());**  **}**  **}** |

### 4.5.5 启动

|  |
| --- |
| **@SpringBootApplication**  **@EnableScheduling**  **public class App {**  **public static void main(String[] args) {**  **SpringApplication.*run*(App.class, args);**  **}**  **}** |

## 消费者:

### 4.5.1 引入 maven依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.1.RELEASE</version>  </parent>  <!-- 管理依赖 -->  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Finchley.M7</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <!-- SpringBoot整合Web组件 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- SpringBoot Activemq -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-activemq</artifactId>  </dependency>  </dependencies>  <!-- 注意： 这里必须要添加， 否者各种依赖有问题 -->  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/libs-milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

### 4.5.2 引入 YML配置

application.yml

|  |
| --- |
| spring:  activemq:  broker-url: tcp://127.0.0.1:61616  user: admin  password: admin  queue: springboot-queue  server:  port: 8081 |

### 4.5.3 创建Consumer

|  |
| --- |
| **@Component**  **public class Consumer {**  **@JmsListener(destination = "${queue}")**  **public void receive(String msg) {**  **System.*out*.println("监听器收到msg:" + msg);**  **}**  **}** |

### 4.5.4 启动

|  |
| --- |
| **@SpringBootApplication**  **public class App {**  **public static void main(String[] args) {**  **SpringApplication.*run*(App.class, args);**  **}**  **}** |