# 第七章 消息中间件

李 会 格

E-mail: 1034434100@qq.com

# 大纲

- 消息中间件概述
  - ✓ 消息中间件的概念
  - ✓ 消息中间件的发展历史
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- JAVA消息中间件JMS
- 消息驱动的Bean
- 小结

# 消息中间件的概念

- 消息中间件（Message Oriented Middleware）是在分布式系统中完成消息的发送和接收的基础软件。

- 分布式应用程序之间的通信接口由消息中间件提供，消息中间件支持与保障分布式应用程序之间同步或异步的收发消息。

# 消息中间件的概念

- 消息中间件的通信方式
  - ✓ 消息中间件采用异步的通信方式，发送方在发送消息时不必知道接收方的状态，更无需等待接收方的回复。
  - ✓ 接收方在收到消息时也不必知道发送方的目前状态，更无需进行同步的消息处理。
  - ✓ 消息的收发双方完全是松耦合的，通信是非阻塞的；收发双方彼此不知道对方的存在，也不受对方影响。
  - ✓ 消息发送者可以将消息间接传给多个接收者，大大提高了程序的性能、可扩展性及健壮性。

# 消息中间件的概念

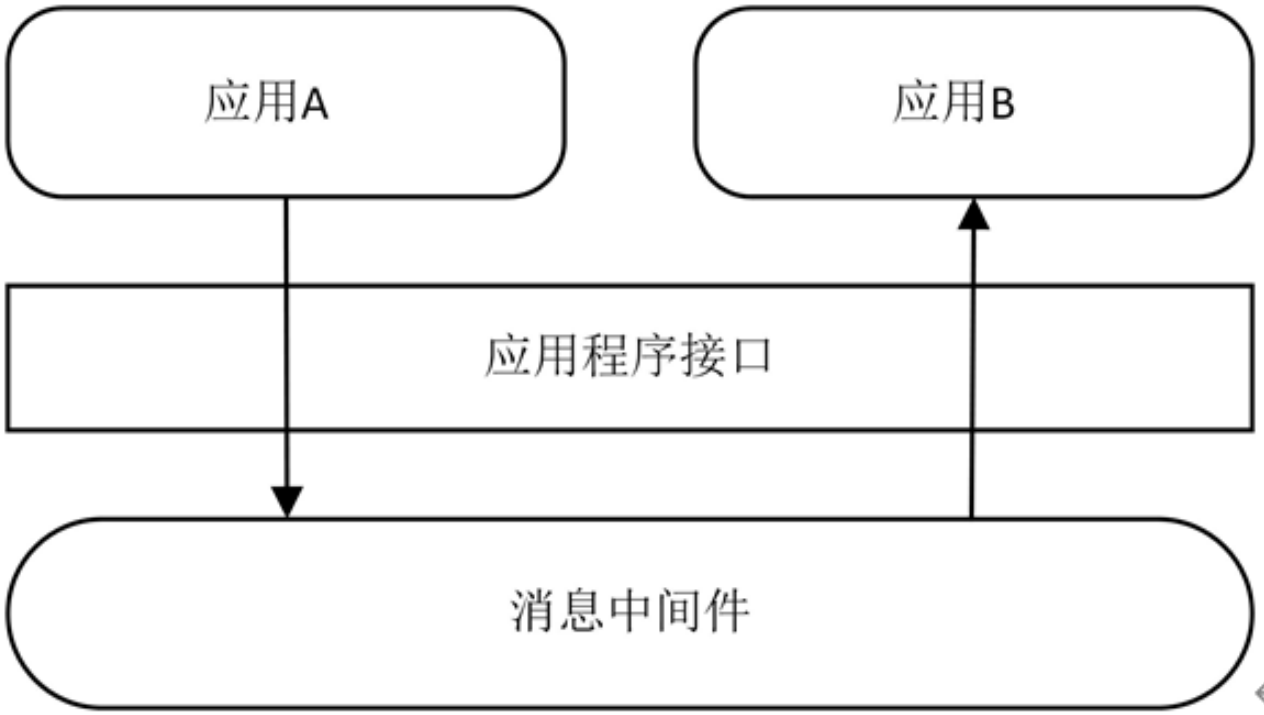


图 6-1 应用与消息中间件交互示意图

# 大纲

- 消息中间件概述
  - ✓ 消息中间件的概念
  - ✓ 消息中间件的发展历史
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- JAVA消息中间件JMS
- 消息驱动的Bean
- 小结

# 消息中间件的发展历史

- 80年代后期，IBM推出了消息中间件产品MQSeries
- 90年代，OMG制定了公共对象服务标准（COSS），其中对消息服务进行了规范。
- 90年代末期，消息中间件开始向发布/订阅架构转变，并成为企业应用集成中间件的一种核心机制
- 进入21世纪，由于J2EE技术的广泛应用，J2EE的消息服务规范JMS（Java Message Service）得到消息中间件厂商的广泛采纳，并逐渐成为消息中间件的事实标准
- W3C组织定义了Web服务的可靠消息传送规范（WS-Reliable Messaging）

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- JAVA消息中间件JMS
- 消息驱动的Bean
- 小结

# 消息中间件的产品

- IBM MQSeries
- WebLogic
- SonicMQ
- Active MQ
- OpenJMS
- RocketMQ
- RabbitMQ
- Kafka
- ZeroMQ
- Redis

# 消息中间件的使用场景

- 代理化、服务化、流程化、平台化是目前消息中间件**发展的主要趋势**。
  - ✓ 代理化是指消息中间件体系架构逐渐向消息代理架构靠拢；
  - ✓ 服务化是指消息中间件在应用高端支持面向服务的体系架构；
  - ✓ 流程化是指消息中间件在应用形态上逐渐与业务流程管理机制相融合，成为企业应用集成中间件的一个核心组成部件；
  - ✓ 平台化是指围绕消息处理，各种应用开发和管理工具与消息中间件有机结合在一起，为分布式应用的消息处理提供一个有机的统一平台。

# 消息中间件的使用场景

## 消息中间件的使用场景：

- 1）异步通信
  - ✓ 消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。
- 2）解耦合
  - ✓ 通过消息系统在处理过程中间插入了一个隐含的、基于数据的接口层
  - ✓ 当应用发生变化时，可以独立的扩展或修改两边的处理过程，以此来降低工程间的强依赖程度
- 3）增加冗余
  - ✓ 消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。

# 消息中间件的使用场景

- 4）增强扩展性
  - ✓ 因为消息队列解耦了处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。
- 5）过载保护
  - ✓ 使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。
- 6）增强可恢复性
  - ✓ 消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理
- 7）顺序保证
  - ✓ 大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。

# 消息中间件的使用场景

- 8）缓冲
  - ✓ 消息队列通过一个缓冲层来帮助任务最高效率的执行，该缓冲有助于控制和优化数据流经过系统的速度，以调节系统响应时间。
- 9）数据流处理
  - ✓ 针对数据流进行实时或批量采集汇总，然后进行大数据分析是当前互联网的必备技术，通过消息队列完成此类数据收集是最好的选择。

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
  - ✓ 点对点和消息代理结构
  - ✓ 消息中间件的要素
  - ✓ 消息中间件常用协议
- JAVA消息中间件JMS
- 消息驱动的Bean
- 小结

# 点对点和消息代理结构

- 传统的点对点消息中间件通常由消息队列服务、消息传递服务、消息队列和消息应用程序接口API组成

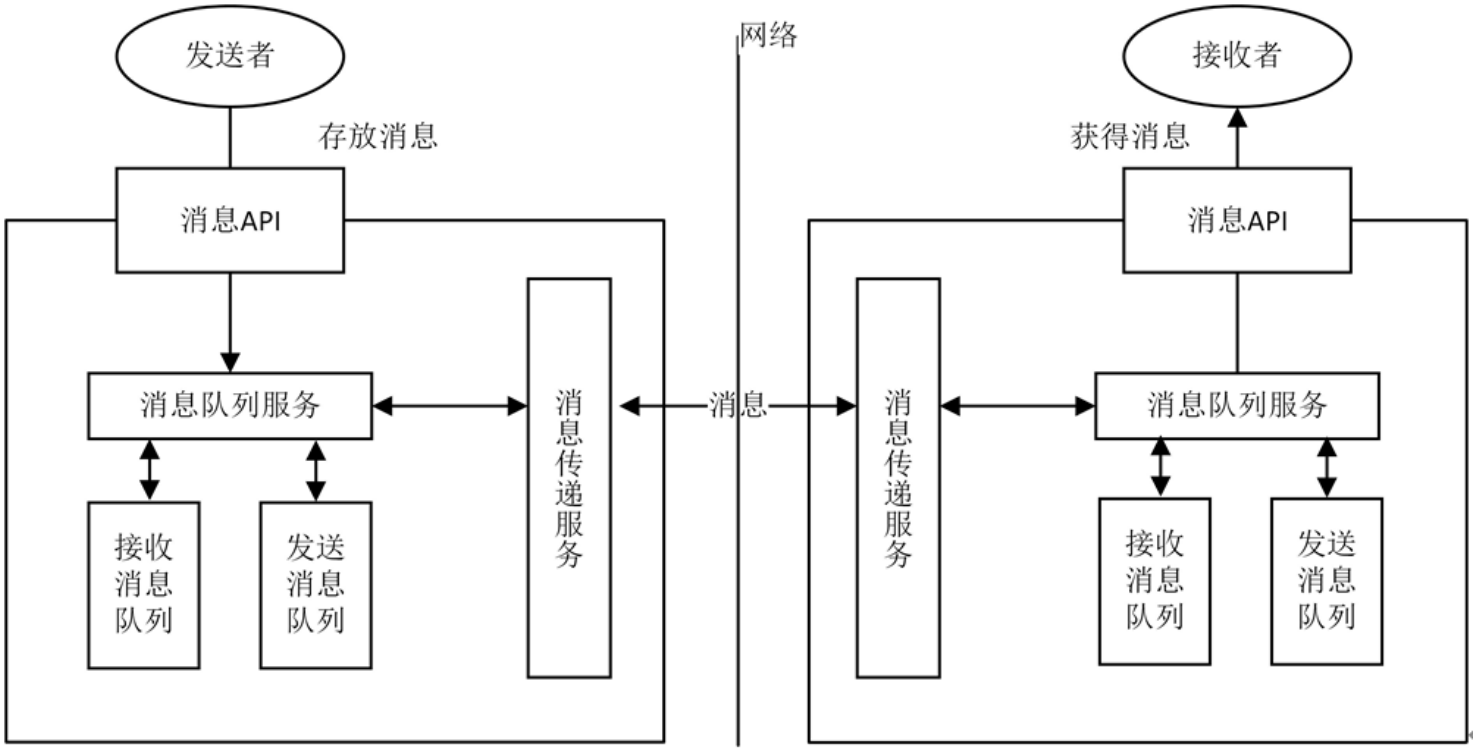


图 6-2 点对点消息中间件结构示意图

# 点对点和消息代理结构

- 点对点消息发送的基本流程：
- 1）消息发送者调用发送消息的API函数，将需要发送的消息经消息队列服务存储到发送消息队列中；
- 2）通过双方消息传递服务之间的交互，消息队列服务将需要发送的消息从发送队列取出，并送到接收方；
- 3）接收方的消息队列服务将接收到的消息存放到接收消息队列中；
- 4）消息接收者调用接收消息的API函数，同样经过消息队列服务，将需要的消息从接收队列中取出，并进行处理。
- 5）消息在发送或接收成功后，消息队列服务将对相应的消息队列进行管理。

# 点对点和消息代理结构

- 在基于消息代理的分布式应用系统中，消息的发送方称为发布者（publisher），消息的接收方称为订阅者（subscriber），不同的消息通过不同的主题进行区分。发布者和订阅者之间通过消息代理进行关联

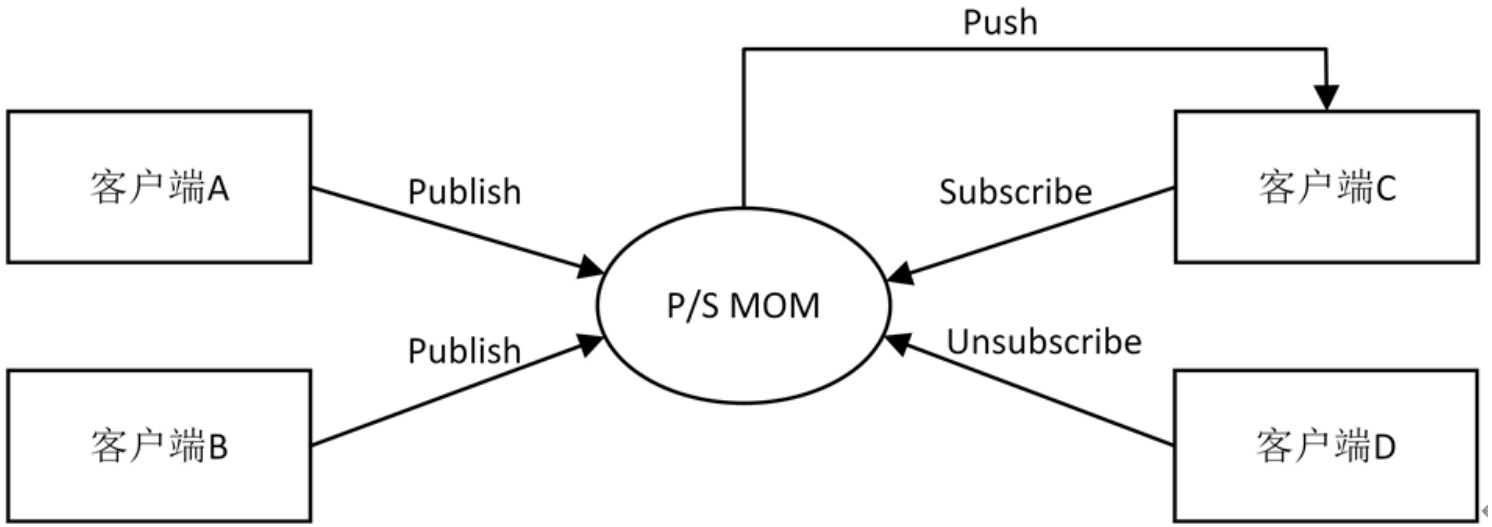


图 6-3 消息代理的基本结构示意图

# 点对点和消息代理结构

- 消息代理的工作流程

✓ 1）消息发布者和订阅者分别同消息代理进行通信。消息发布者将包含主题的消息发布到消息代理；消息订阅者向消息代理订阅自己感兴趣的主题。

✓ 2）消息代理对双方的主题进行匹配后，不断将订阅者感兴趣的消息推(Push)给订阅者，直到订阅者向消息代理发出取消订阅的消息。

# 点对点和消息代理结构

- 消息代理实现了发布者和订阅者的解耦：
  - ✓ 时间解耦：发布方和订阅方无需同时在线就能够进行消息传输，消息中间件通过存储转发提供了这种异步传输的能力；
  - ✓ 空间解耦：发布方和订阅方都无需知道对方的物理地址、端口，甚至无需知道对方的逻辑名字和个数；
  - ✓ 流程解耦：发布方和订阅方在发送和接收数据时并不阻塞各自的控制流程。

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
  - ✓ 点对点消息代理结构
  - ✓ 消息中间件的要素
  - ✓ 消息中间件常用协议
- JAVA消息中间件JMS
- 消息驱动的Bean
- 小结

# 消息中间件的要素

- 无论是点对点消息中间件还是消息代理，消息中间件的体系结构都是非常清晰简单的。
- 但由于分布式应用及其环境的多样性和复杂性，导致了消息中间件的复杂性:
  - ✓ 跨平台、跨语言
  - ✓ 存储转发或消息路由
  - ✓ 消息传输的安全性、事务性、时限等质量要求
  - ✓ 持久存储能力

消息中间件要素
- 消息的表示
- 消息队列
- 消息路由
- 消息QoS机制

# 消息中间件的要素

- 消息的表示
  - ✓ 消息头：用于描述消息发送者和接收者的地址或消息主题，以及消息的服务质量要求，例如，消息传输的时限、优先级、安全属性等；
  - ✓ 消息体：用于描述消息中具体携带的信息内容。
- 目前多采用XML作为消息表示的格式。

# 消息中间件的要素

- 消息队列
  - ✓ 为了有效控制消息收发过程而在消息中间件中内置的存储消息的数据结构。
  - ✓ 由于消息多采用先进先出的控制方式，通常采用队列作为消息的存储结构。

- 队列的分类：
  - ✓ 消息的内容：发送队列、接收队列、死信队列
  - ✓ 消息发送的质量：优先队列、普通队列。
  - ✓ 队列存储介质：持久消息队列、内存队列和高速缓存队列。

# 消息中间件的要素

- 按队列存储介质的队列分类
  - ✓ 持久消息队列：基于数据库或文件系统，提供消息持久存储功能，同时又具有最小的内存开销，适合于消息需要可靠传输的应用环境。
  - ✓ 内存队列：基于内存的消息队列。不提供消息持久功能，完全基于内存来进行消息的缓存和分发。适合对性能要求非常苛刻，但是消息无需可靠持久的应用环境。
  - ✓ 高速缓存队列：基于数据库和内存Cache。提供可靠持久功能，同时又使用内存作为Cache，具有最大的资源开销，同时又具有很高的性能。适合于大部分应用场合。

# 消息中间件的要素

- 消息路由
  - ✓ 消息路由借用了IP层的路由和路由器中的路由的概念
  - ✓ 不同之处在于：消息路由属于应用层的概念，它是为了保证应用之间的消息交换处于可控的状态而设计的软件功能模块，其机制是按照消息路由规则将消息从发送者传送到目标应用，并提供消息流量控制功能。
  - ✓ 消息路由有时也称为“流量控制”、“基于内容的路由”、“智能路由”。

# 消息中间件的要素

- 消息QoS机制
  - ✓服务质量(Quality of Service，简称QoS)是指与用户对服务满意程度相关的各种性能效果。
  - ✓消息QoS机制是指消息中间件提供的消息传送过程中在性能、安全、可靠性等方面的各种非功能型需求约束。
  - ✓特性：可靠性、事务性、安全性、优先级、时间约束、队列管理

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
  - ✓ 点对点消息代理结构
  - ✓ 消息中间件的要素
  - ✓ 消息中间件常用协议
- JAVA消息中间件JMS
- 消息驱动的Bean
- 小结

# 中间件常用协议

- AMQP协议
  - ✓ AMQP（Advanced Message Queuing Protocol，高级消息队列协议）是一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计的。
  - ✓ 基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同开发语言等条件的限制。
  - ✓ 该协议具有可靠、通用的优点。

# 中间件常用协议

- MQTT协议
  - ✓ MQTT（Message Queuing Telemetry Transport，消息队列遥测传输）是IBM开发的一个即时通讯协议，有可能成为物联网的重要组成部分。
  - ✓ 该协议支持所有平台，几乎可以把所有联网物品和外部连接起来，被用来当做传感器和致动器的通信协议。
  - ✓ 该协议具有格式简洁、占用带宽小等特点，适用于移动端通信及嵌入式系统。

# 中间件常用协议

- STOMP协议
  - ✓ STOMP（Streaming Text Orientated Message Protocol，流文本定向消息协议）是一种为MOM（Message Oriented Middleware，面向消息的中间件）设计的简单文本协议。
  - ✓ STOMP提供一个可互操作的连接格式，允许客户端与任意STOMP消息代理（Broker）进行交互。
  - ✓ 由于协议简单且易于实现，几乎所有的编程语言都有 STOMP 的客户端实现。但是它在消息大小和处理速度方面并无优势。

# 中间件常用协议

- XMPP协议
  - ✓ XMPP（Extensible Messaging and Presence Protocol，可扩展消息处理现场协议）是基于可扩展标记语言（XML）的协议，多用于即时消息（IM）以及在线现场探测。
  - ✓ 适用于服务器之间的准即时操作。
  - ✓ 核心是基于XML流传输，这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息，即使其操作系统和浏览器不同。
  - ✓ 该协议具有通用公开、兼容性强、可扩展、安全性高等特点，但XML编码格式占用带宽大。

# 中间件常用协议

- 其他自定义协议
  - ✓ 有些特殊框架（如：redis、kafka、zeroMq等）根据自身需要未严格遵循MQ规范，而是基于TCP/IP自行封装了一套协议，通过网络socket接口进行传输，实现了MQ的功能。

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- **JAVA消息中间件JMS**
  - ✓ **JMS简介**
  - ✓ **JMS架构**
  - ✓ **JMS编程示例**
- 消息驱动的Bean
- 小结

# JMS简介

- JMS即Java消息服务（Java Message Service），是Java平台上有关面向消息中间件(MOM)的技术规范
- 便于消息系统中的Java应用程序进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发。
- JMS 通过 MOM 产品为 Java 程序提供了一个发送和接收消息的标准的、便利的方法。用 JMS 编写的程序可以在任何实现 JMS 标准的 MOM 上运行。

# JMS简介

- JMS 1.0版本于1998年推出，支持消息中间件的两种传递模式：点到点模式和代理（发布-订阅）模式。
- JMS 1.1版本提供了单一的一组接口，允许客户机可以在两个模式中发送和接收消息。

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- **JAVA消息中间件JMS**
  - ✓ JMS简介
  - ✓ **JMS架构**
  - ✓ JMS编程示例
- 消息驱动的Bean
- 小结

# JMS架构

- 为了发送或接收消息，JMS 客户端必须首先连接到 JMS 消息服务器（通常称为代理）。该连接打开了客户端和代理之间的通信通道。
- 接下来，客户端必须建立一个会话来创建、生成和使用消息。可以将会话视为定义客户端和代理之间特定对话的消息流。
- 客户端本身是消息生产者和/或消息消费者，消息生产者将消息发送到代理管理的目的地。

# JMS架构

- 消息的分类
- 一条JMS Message由三个部分组成：头（header），属性（property）和主体（body）。
- 消息有下面几种类型，他们都是派生自 Message 接口
  - ✓ StreamMessage：一种主体中包含 Java 基元值流的消息。其填充和读取均按顺序进行。
  - ✓ MapMessage：一种主体中包含一组名-值对的消息。没有定义条目顺序。
  - ✓ TextMessage：一种主体中包含 Java 字符串的消息（例如，XML 消息）。
  - ✓ ObjectMessage：一种主体中包含序列化 Java 对象的消息。
  - ✓ BytesMessage：一种主体中包含连续字节流的消息

# JMS架构

- JMS模型
- JMS 支持两种消息传递模型：
  - 点对点（point-to-point，简称 PTP）
  - 发布/订阅（publish/subscribe，简称 pub/sub）
- 两种模型相似但有如下区别
  - PTP消息传递模型规定了一条消息只能传递给一个接收方。用javax.jms.Queue表示。
  - Pub/sub消息传递模型允许一条消息传递给多个接收方。用javax.jms.Topic表示。
- 这两种模型都通过扩展公用基类来实现
  - javax.jms.Queue 和javax.jms.Topic 都扩展自javax.jms.Destination 类。

# JMS架构

- JMS客户端应用程序可以使用两种消息传递模式（或域）来发送和接收消息

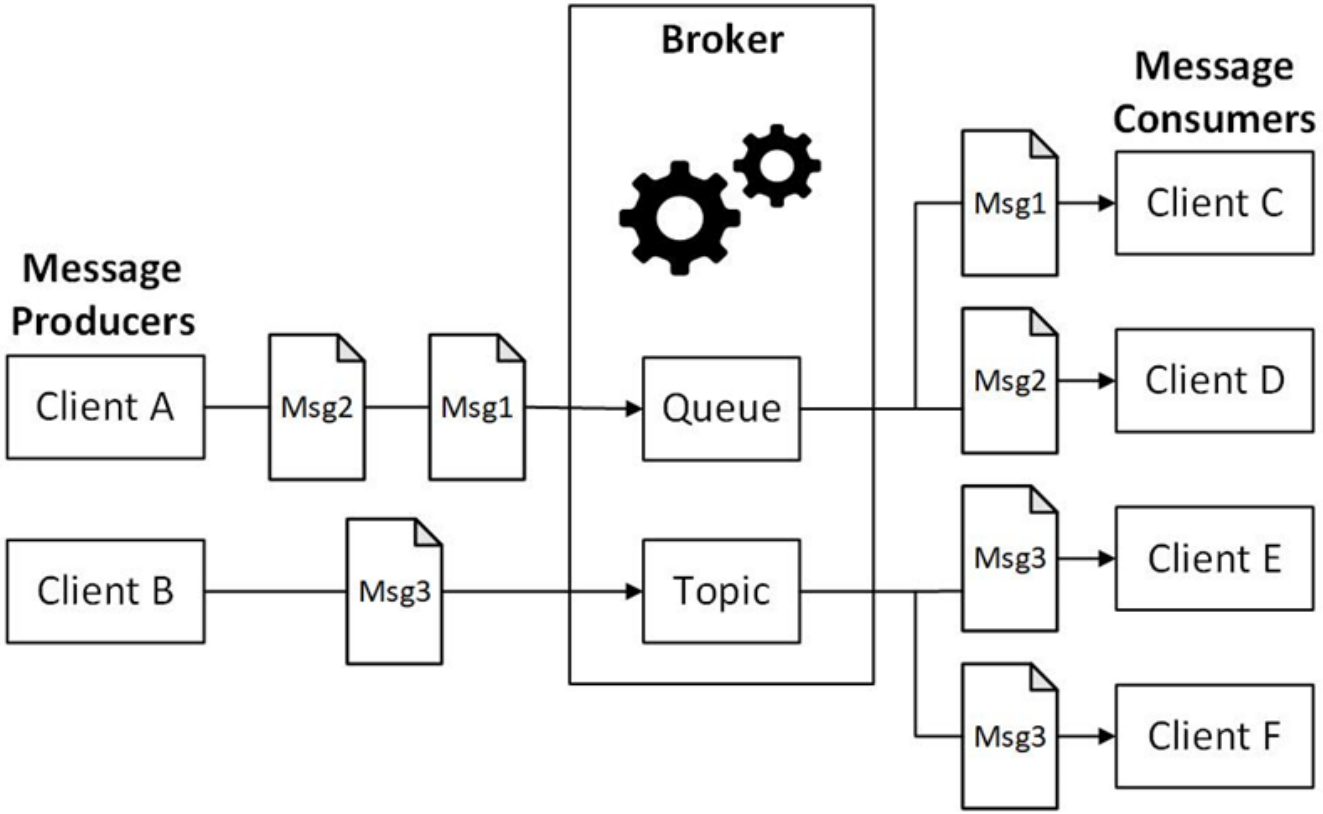


图 6-4 JMS 消息传递域

# JMS架构

- 传递消息方式
- JMS现在有两种传递消息的方式.
  - ✓ 标记为NON_PERSISTENT的消息最多投递一次
  - ✓ 标记为PERSISTENT的消息将使用暂存后再转送的机理投递
- 如果一个JMS服务离线，那么持久性消息不会丢失但是得等到这个服务恢复联机时才会被传递。
- JMS定义了从0到9的优先级路线级别，0是最低的优先级而9则是最高的。
- 更特殊的是0到4是正常优先级的变化幅度，而5到9是加快的优先级的变化幅度。

# JMS架构

- 例子
- topicPublisher.publish(message, DeliveryMode.PERSISTENT, 8, 10000);或queueSender.send(message,DeliveryMode.PERSISTENT, 8, 10000);这个代码片断，有两种消息模型。
- 映射递送方式是持久的，优先级为加快型，生存周期是10000 （以毫秒度量）。
- 如果生存周期设置为零，则消息将永远不会过期。当消息需要时间限制，超过限制将使其无效时，设置生存周期是有用的。

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- **JAVA消息中间件JMS**
  - ✓ JMS简介
  - ✓ JMS架构
  - ✓ JMS编程示例
- 消息驱动的Bean
- 小结

# JMS编程示例

- 该示例使用ActiveMQ来作为消息中间件，ActiveMQ基本运行原理如图6-5所示

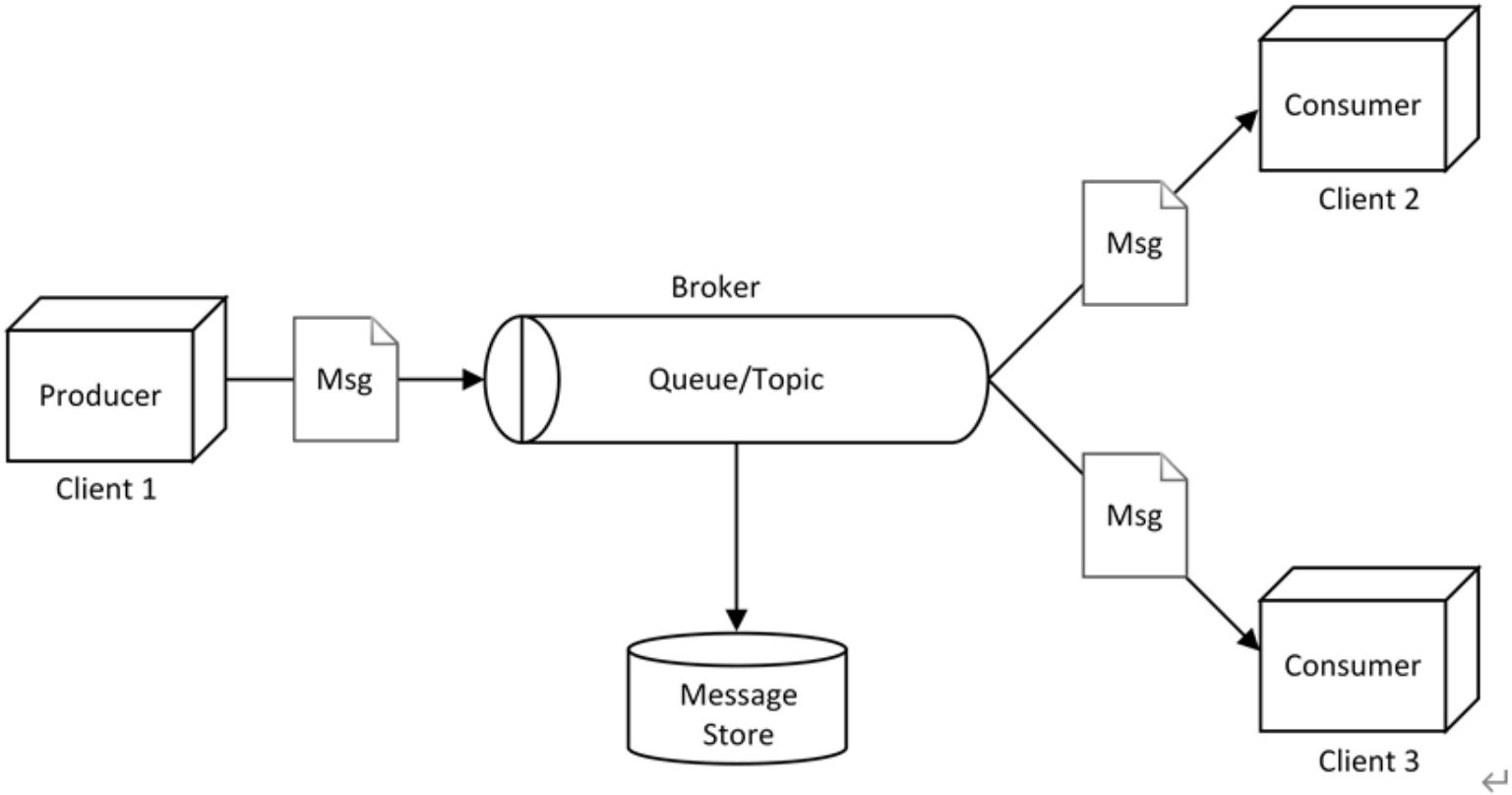


图 6-5 ActiveMQ 基本运行原理示意图

# JMS编程示例

- 主要代码
- 生产者代码

```
ConnectionFactory  connectionFactory =
      new  ActiveMQConnectionFactory("tcp://localhost:61616");//服务地址，须查看配置
Connection connection = connectionFactory.createConnection();
//通过连接服务工厂提供连接对象 connection
connection.start();//开启连接，直接调用 connection 对象的 start 方法即可。

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//通过 connection 对象创建 session 会话
```

```
Queue queue = session.createQueue("Myqueue");
//参数表示队列的名称
MessageProducer producer = session.createProducer(queue);
//为生产者指定消息传播“目的地”
TextMessage textMessage = session.createTextMessage("hello world");//构建消息对象
producer.send(textMessage);
producer.close();
session.close();
connection.close();//使用完毕后关闭资源
```

# JMS编程示例

- 主要代码
- 消费者代码

```
ConnectionFactory connectionFactory =
      new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("Myqueue ");
```

```
MessageConsumer consumer = session.createConsumer(queue);
```

```
consumer.setMessageListener(new MessageListener() {
   public void onMessage(Message message) {
      try {
         TextMessage textMessage = (TextMessage) message;
         String text = null;
         //取消息的内容
         text = textMessage.getText();
         //第八步：打印消息。
         System.out.println(text);
      } catch (JMSException e) {
         e.printStackTrace();
      }
   }
});
```

# JMS编程示例

- JMS接口介绍
- JMS消息由以下几部分组成：消息头，属性，消息体
  - ✓ 消息头（header）：JMS消息头包含了许多字段，它们是消息发送后由JMS提供者或消息发送者产生，用来表示消息、设置优先权和失效时间等等，并且为消息确定路由。
  - ✓ 属性（property）：由消息发送者产生，用来添加删除消息头以外的附加信息。
  - ✓ 消息体（body）：由消息发送者产生，JMS中定义了5种消息体：ByteMessage、MapMessage、ObjectMessage、StreamMessage和TextMessage。

# JMS编程示例

- JMS接口介绍
- JMS接口：
  - ConnectionFactory：连接工厂，JMS用它创建连接，是创建Connection对象的工厂。
  - Connection：JMS客户端到JMS Provider的连接；表示在客户端和JMS系统之间建立的连接（对TCP/IP socket的包装）。
  - Destination：消息生产者的消息发送目标或者说消息消费者的消息来源。
  - Session：操作消息的接口，可以通过session创建生产者、消费者、消息。
  - JMSProducer：用来发送消息的对象

# JMS编程示例

- JMS接口介绍
- JMS接口：
  - ✓ JMSConsumer：用于接收被发送到Destination的消息。
  - ✓ MessageListener：消息监听器。如果注册了消息监听器，一旦消息到达，将自动调用监听器的onMessage方法。

# JMS编程示例

- JMS接口介绍
- 一个JMS应用是通常是几个JMS 客户端交换消息，开发JMS客户端应用由以下几步构成：
  - ✓ 用JNDI得到ConnectionFactory对象；
  - ✓ 用JNDI得到目标队列或主题对象，即Destination对象；
  - ✓ 用ConnectionFactory创建Connection 对象；
  - ✓ 用Connection对象创建一个或多个JMS Session；
  - ✓ 用Session和Destination创建MessageProducer和MessageConsumer；
  - ✓ 通知Connection 开始传递消息。

# JMS编程示例

- 消息生产和消费
- 消息生产者服务实现的代码

```
@Service
public class ProducerServiceImpl implements ProducerService {
//生产者必须实现 ProducerService 接口
   @Autowired
   JmsTemplate jmsTemplate;
   @Resource(name = "topicDestination")
   //这里的参数指定 destination 名称必须和配置文件中保持一致
   private Destination destination;
   public void sendMessage(final String message) {
      jmsTemplate.send(destination, new MessageCreator() {
      //jmsTemplate 调用 send 方法，第一个参数表示发布的 destionation，
      //第二个参数是 MessageCreator 的匿名实现对象
            public Message createMessage(Session session) throws JMSException {
                TextMessage textMessage = session.createTextMessage(message);
                System.out.println("发送消息 = [" + textMessage.getText() + "]");
                return textMessage;
                //MessageCreator 匿名实现，
                //方法中调用 session 的 createTextMessage()返回 textMessage
            }
      });
   }
}
```

# JMS编程示例

- 消息生产和消费
- 消息消费者的代码

```
@Component
public class ConsumerMessageListener implements MessageListener {
//消费者必须实现 MessageListenser 接口
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;
        try {
            System.out.println("接收 message: " + textMessage.getText());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

# JMS编程示例

- 消息生产和消费
- 启动消息消费者的代码

```
public class AppListener {
   private static int N1 = 1;
   
   public static void main(String[] args) {
      ApplicationContext[] contexts = new ApplicationContext[N1];
      for (int i = 0; i < N1; i++) {
         contexts[i] = new ClassPathXmlApplicationContext("spring-listener.xml");
      }
   }
}
```

# JMS编程示例

- 消息生产和消费
- 启动消息生产者的代码

```
public class AppProducer {
   private static int N2 = 9999;
   public static void main(String[] args) {
      ClassPathXmlApplicationContext context;
      ProducerService service;
      for (int i = 0; i < N2; i++) {
         context = new ClassPathXmlApplicationContext("spring-producer.xml");
         service = context.getBean(ProducerService.class);
         service.sendMessage("消息" + (i+1));
         context.close();
      }
   }
}
```

# JMS编程示例

- 异步消息监听
- 在JMS中，消息的产生和消费是异步的。对于消费来说，JMS的消费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方法来接受消息，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。

# JMS编程示例

- 异步消息监听
- 对客户端来说，消息驱动Bean（MDB）就是异步消息的消费者。当消息到达之后，由容器负责调用MDB，客户端发送消息到destination，MDB作为一个MessageListener接收消息。

```
public class ConsumeMessageListener implements MessageListener {
   private final static Logger log =Logger.getLogger("Log");
   public void onMessage(Message rcvMessage) {
      TextMessage msg = null;
      try {
         if (rcvMessage instanceof TextMessage) {
            msg = (TextMessage) rcvMessage;
            log.info("Received Message from queue: " + msg.getText());
         } else {
            log.warning("Message of wrong type: " + rcvMessage.getClass().getName());
         }
      } catch (JMSException e) {
         throw new RuntimeException(e);
      }
   }
}
```

# JMS编程示例

- 异步消息监听
- 一般而言，异步消息消费者的执行和伸缩性都优于同步消息接收者
  - ✓ 异步消息接收者创建的网络流量比较小。
  - ✓ 异步消息接收者使用的线程比较少。
  - ✓ 对于服务器上运行的应用程序代码，使用异步消息接收者几乎总是最佳选择，尤其是通过消息驱动Bean。

# 大纲

- 消息中间件概述
- 消息中间件产品和使用场景
- 消息中间件的架构和协议
- JAVA消息中间件JMS
  - ✓ JMS简介
  - ✓ JMS架构
  - ✓ JMS编程示例
- 消息驱动的Bean
- 小结

# 消息驱动的Bean

- 消息驱动的Bean组件 (Message Driven Bean) 是用来转换处理基于消息请求的组件。
- MDB负责处理消息，而EJB容器则负责处理服务(事务、安全、资源、并发、消息确认,等等),使bean开发者把精力集中在消息处理的业务逻辑上。
- MDB它和无状态Session Bean一样也使用了实例池机制,容器可以为它创建大量的实例,用来并发处理成百上千个JMS消息。

# 消息驱动的Bean

- MDB 通常要实现 MessageListener 接口，该接口定义了 onMessage()方法。
- 当容器检测到 bean 守候的管道有消息到达时，容器调用 onMessage()方法，将消息作为参数传入 MDB。
- onMessage()中决定如何处理该消息：
  - ✓ 可以使用注解指定 MDB 监听哪一个目标地址(Destination)。当 MDB 部署时，容器将读取其中的配置信息。

# 消息驱动的Bean

- 通过注解可以描述消息驱动的属性相关信息
- PTP消息传递模型下的配置

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(propertyName="destinationType",
                             propertyValue="javax.jms.Queue"),
   @ActivationConfigProperty(propertyName="destination",
                             propertyValue="queue/MQshop"),
})
```

- Pub/sub消息传递模型下的配置

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(propertyName="destinationType",
                             propertyValue="javax.jms.Topic"),
   @ActivationConfigProperty(propertyName="destination",
                             propertyValue="top/MQshop"),
})
```

# 消息驱动的Bean

- 使用MDB接收消息
- 以下代码为一个完整的MDB接收消息的例子，使用了两个MDB分别对Queue消息和Topic消息进行监听。

```
//接受 queue 信息的 MDB
package org.jboss.as.quickstarts.mdb;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
   @ActivationConfigProperty(propertyName = "destinationLookup",
                             propertyValue = "queue/HELLOWORLDMDBQueue"),
   @ActivationConfigProperty(propertyName = "destinationType",
                             propertyValue = "javax.jms.Queue"),
                             //指定 destination 的类型为 Queue 类型
   @ActivationConfigProperty(propertyName = "acknowledgeMode",
                             propertyValue = "Auto-acknowledge")
                             //自动进行应答
})
```

# 消息驱动的Bean

- 使用MDB接收消息

```
public class HelloWorldQueueMDB implements MessageListener {
//消息的监听者继承 MessageListienser 接口，类中的 onMessage 方法会在消息到达的时候
自动被调用，实现异步监听
  
  private final static Logger LOGGER = Logger.getLogger(HelloWorldQueueMDB.class.toString());
  //获取日志
  
  /**
   * @see MessageListener#onMessage(Message)
   */
  public void onMessage(Message rcvMessage) {
    TextMessage msg = null;
    try {
      if (rcvMessage instanceof TextMessage) {
        msg = (TextMessage) rcvMessage;
        LOGGER.info("Received Message from queue: " +
                    msg.getText());
        //TextMessage 类型的消息通过 getText()方法可以获取消息具体内容
      } else {
        LOGGER.warning("Message of wrong type: " + rcvMessage.getClass().getName());
      }
    } catch (JMSException e) {
      throw new RuntimeException(e);
    }
  }
}
```

# 消息驱动的Bean

- 使用MDB接收消息

```
//接受 topic 信息的 MDB
package org.jboss.as.quickstarts.mdb;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(name = "HelloWorldQTopicMDB", activationConfig = {
   @ActivationConfigProperty(propertyName = "destinationLookup",
                             propertyValue = "topic/HELLOWORLDMDBTopic"),
   @ActivationConfigProperty(propertyName = "destinationType",
                             propertyValue = "javax.jms.Topic"),
                             //这里 destination 指定为 topic 类型
   @ActivationConfigProperty(propertyName = "acknowledgeMode",
                             propertyValue = "Auto-acknowledge") //自动应答
})
```

# 消息驱动的Bean

- 使用MDB接收消息

```
public class HelloWorldTopicMDB implements MessageListener {
//同理 onMessage 方法会在
  
  private final static Logger LOGGER = Logger.getLogger(HelloWorldTopicMDB.class.toString());
  //获取日志
  
  /**
   * @see MessageListener#onMessage(Message)
   */
  public void onMessage(Message rcvMessage) {
    TextMessage msg = null;
    try {
      if (rcvMessage instanceof TextMessage) {
        msg = (TextMessage) rcvMessage;
        LOGGER.info("Received Message from topic: " + msg.getText());
      } else {
        LOGGER.warning("Message of wrong type: " + rcvMessage.getClass().getName());
      }
    } catch (JMSException e) {
      throw new RuntimeException(e);
    }
  }
}
```

# 消息驱动的Bean

- 使用MDB接收消息

```
/**
 * Definition of the two JMS destinations used by the quickstart
 * (one queue and one topic).
 */
@JMSDestinationDefinitions(
  value = {
    @JMSDestinationDefinition(
      name = "java:/queue/HELLOWORLDMDBQueue",
      interfaceName = "javax.jms.Queue",
      destinationName = "HelloWorldMDBQueue".//
    ),
    @JMSDestinationDefinition(
      name = "java:/topic/HELLOWORLDMDBTopic",
      interfaceName = "javax.jms.Topic",
      destinationName = "HelloWorldMDBTopic"
    )
  })//为 client 发送的消息指定 destination，第一个为 queue 类型，第二个为 topic 类型
@WebServlet("/HelloWorldMDBServletClient")
```

# 消息驱动的Bean

- 使用MDB接收消息

```
public class HelloWorldMDBServletClient extends HttpServlet {
//这个 servlet 充当发送消息的客户端
   private static final long serialVersionUID = -8314035702649252239L;

   private static final int MSG_COUNT = 5;

   @Inject
   private JMSContext context;

   @Resource(lookup = "java:/queue/HELLOWORLDMDBQueue")
   private Queue queue;

   @Resource(lookup = "java:/topic/HELLOWORLDMDBTopic")
   private Topic topic;

   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
               ServletException, IOException {
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();
      out.write("<h1>Quickstart: Example demonstrates the use of <strong>JMS 2.0</strong>
                  and <strong>EJB 3.2 Message-Driven Bean</strong> in WildFly.</h1>");
      try {
         boolean useTopic = req.getParameterMap().keySet().contains("topic");
        //获取请求参数中要求的传递给的 destination 类型，为 true 则使用 topic 为消息
        //发送的 destination,为 false 则使用 queue 为消息的 destination
```

# 消息驱动的Bean

- 使用MDB接收消息

```
        final Destination destination = useTopic ? topic : queue;
        out.write("<p>Sending messages to <em>" + destination + "</em></p>");
        out.write("<h2>Following messages will be send to the destination:</h2>");
        for (int i = 0; i < MSG_COUNT; i++) {
          String text = "This is message " + (i + 1);
          context.createProducer().send(destination, text);//发送消息
          out.write("Message (" + i + "): " + text + "</br>");
        }
        out.write("<p><i>Go to your WildFly Server console or Server log to see the result of
                messages processing</i></p>");
      } finally {
        if (out != null) {
          out.close();
        }
      }
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
            ServletException, IOException {
      doGet(req, resp);
    }
}
```

# 消息驱动的Bean

- 消息选择器
- 消息选择器（Message Selector）允许MDB选择性地接收来自队列或主题特定的消息。
- 消息选择器是基于消息属性进行选择的。
- 消息属性是一种可以被附加于消息之上的头信息，开发人员可以通过它为消息附加一些信息，而这些信息不属于消息正文。
- 当应用需要增加一种新业务，这种新业务需要在旧的消息格式上增加若干个参数。为了避免影响到其它业务模块，可以增加新的MDB来处理新的业务消息。新的业务消息不会被旧的业务模块所接收。

# 消息驱动的Bean

- 增加新的MDB来处理新的业务消息示例
- 1、在消息生产端，自定义一个消息版本属性MessageVersion，把它附加到消息属性里。

```
InitialContext ctx = new InitialContext();
//获取 ConnectionFactory 对象
TopicConnectionFactory factory = (TopicConnectionFactory)ctx.lookup("ConnectionFactory");
//创建 TopicConnection 对象
TopicConnection connection = factory.createTopicConnection();
//创建 TopicSession 对象，第一个参数表示事务自动提交，第二个参数标识一旦消息被正确送达，将自动发回响应
TopicSession session =connection.createTopicSession(false,
TopicSession.AUTO_ACKNOWLEDGE);
//获得 Destination 对象
Topic topic = (Topic)ctx.lookup("topic/mytopic");
//创建文本消息
TextMessage msg = session.createTextMessage("世界，你好");
//消息版本属性 MessageVersion
msg.setStringProperty("MessageVersion", "2.0");
//创建发布者
TopicPublisher publisher = session.createPublisher(topic);
//发送消息
publisher.publish(msg);
//关闭会话
session.close();
```

# 消息驱动的Bean

- 2、让处理新业务的MDB只接收2.0版本的消息，在@ActivationConfigProperty注释中，使用标准属性messageSelector来声明消息选择器。

```
@MessageDriven(
   activationConfig = {
      @ActivationConfigProperty(propertyName="destinationType",
                                propertyValue="javax.jms.Topic"),
      @ActivationConfigProperty(propertyName="destination",
                                propertyValue="topic/mytopic"),
      @ActivationConfigProperty(propertyName="messageSelector",
                                propertyValue="MessageVersion='2.0'")
   }
)
public class MyTopicMDBSelectorBean implements MessageListener {
   public void onMessage(Message msg) {
      try {
         TextMessage textMessage = (TextMessage)msg;
         System.out.println("SelectorMDBBean 被调用了！ ["+textMessage.getText()+"]");
      } catch (JMSException e) {
         e.printStackTrace();
      }
   }
}
```

# 消息驱动的Bean

- 3、让处理旧业务的MDB只接收1.0版本的消息

```
@MessageDriven(
   activationConfig = {
      @ActivationConfigProperty(propertyName="destinationType",
                                propertyValue="javax.jms.Topic"),
      @ActivationConfigProperty(propertyName="destination",
                                propertyValue="topic/mytopic"),
      @ActivationConfigProperty(propertyName="messageSelector",
                                propertyValue="MessageVersion='1.0'")
   }
)
public class MyTopicMDBBean2 implements MessageListener {
   public void onMessage(Message msg) {
      try {
         TextMessage textMessage = (TextMessage)msg;
         System.out.println("OldMDBBean2 被调用了! ["+textMessage.getText()+"]");
      } catch (JMSException e) {
         e.printStackTrace();
      }
   }
}
```

# 本章小结

- 消息中间件（Message Oriented Middleware）是在分布式系统中完成消息的发送和接收的基础软件。

- 本章首先介绍了消息中间件的概念和发展历史，然后介绍了中间件的产品和使用场景，详细说明了消息中间件的架构、要素、以及常用的协议。接下来介绍了在Java平台上的消息中间件规范JMS，包括JMS架构和程序接口的介绍，并通过实际编程的例子来帮助读者学习消息中间件的使用。再接着介绍了消息驱动的Bean组件，包括消息的异步处理和消息选择器。