JMS熟练应用

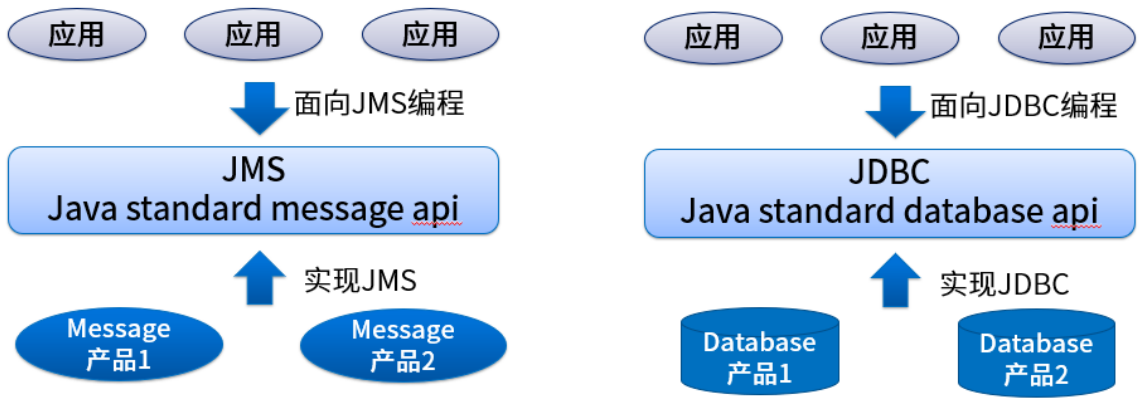
官方说明：https://www.oracle.com/technetwork/java/jms/index.html

1 JMS概述

Java Message Service (JMS).

JMS provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages。

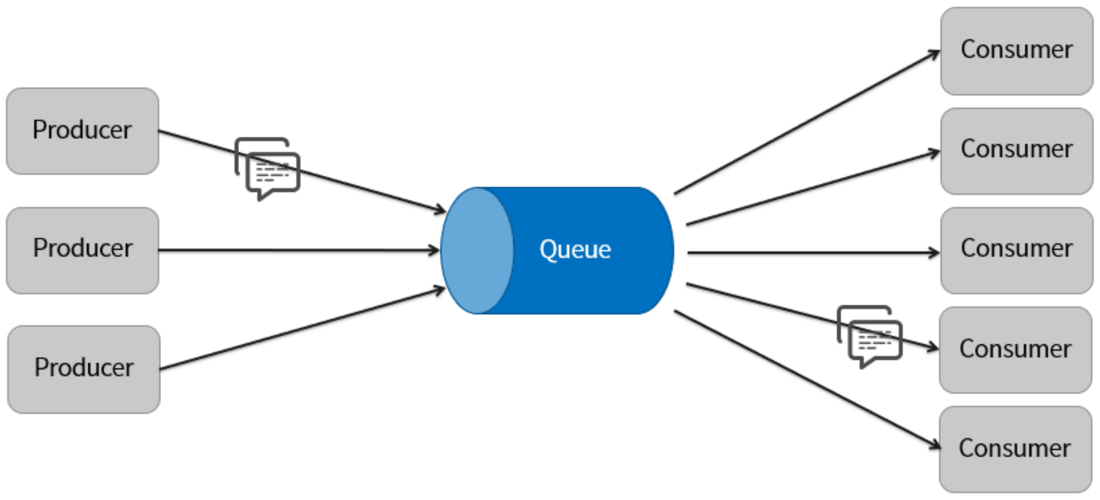
JMS was initially developed to provide a standard Java API for the established messaging products that already existed. Since then many more messaging products have been developed.
JMS provides a common way for both Java client applications and Java middle-tier services to use these messaging products. It defines some messaging semantics and a corresponding set of Java interfaces.



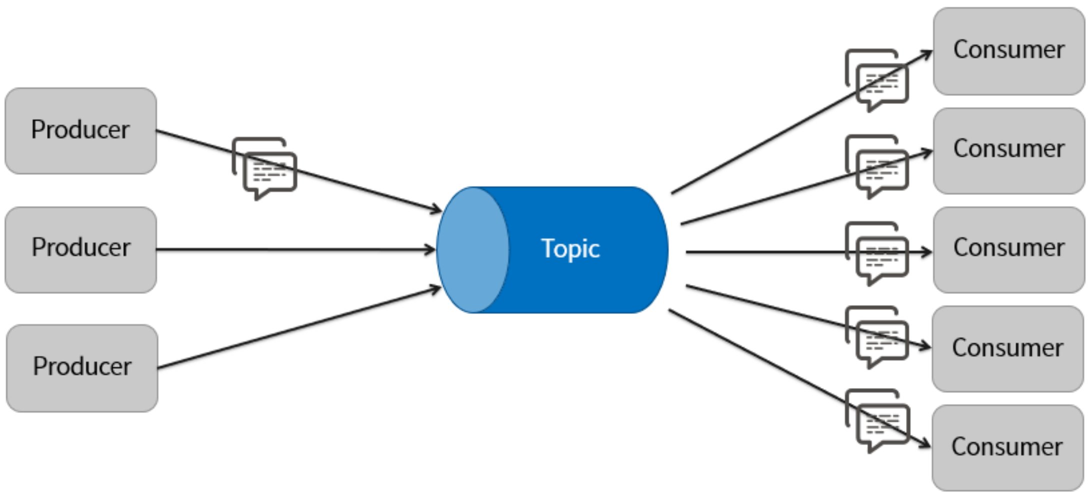
2 JMS支持的两种消息传递模式

JMS supports the two major styles of messaging provided by enterprise messaging products:

- **Point-to-point (PTP)** messaging allows a client to send a message to another client via an intermediate abstraction called a ***queue***. The client that sends the message sends it to a specific queue. The client that receives the message extracts it from that queue.



- **Publish and subscribe (pub/sub)** messaging allows a client to send a message to multiple clients via an intermediate abstraction called a ***topic***. The client that sends the message publishes it to a specific topic. The message is then delivered to all the clients that are subscribed to that topic.



3 JMS API

3.1 JMS API 发展历史

For historical reasons JMS offers four alternative sets of interfaces for sending and receiving messages.

- JMS 1.0 defined two **domain-specific APIs**, one for point-to-point messaging (queues) and one for pub/sub (topics). Although these remain part of JMS for reasons of backwards compatibility they should be considered to be completely superseded by the later APIs. 【不需要学习】
- JMS 1.1 introduced a new unified API which offered a single set of interfaces that could be used for both point-to-point and pub/sub messaging. This is referred to here as the **classic API**. 【学会】
- JMS 2.0 introduces a **simplified API** which offers all the features of the classic API but which requires fewer interfaces and is simpler to use. 【了解】
- Each API offers a different set of interfaces for connecting to a JMS provider and for sending and receiving messages. However they all share a **common set of interfaces** for representing messages and message destinations and to provide various utility features. 【了解】

All interfaces are in the javax.jms package.

【注意】ActiveMQ 5 实现了 JMS1.1

http://activemq.apache.org/components/classic/

3.2 Interfaces common to multiple APIs 各套API共用接口【了解】

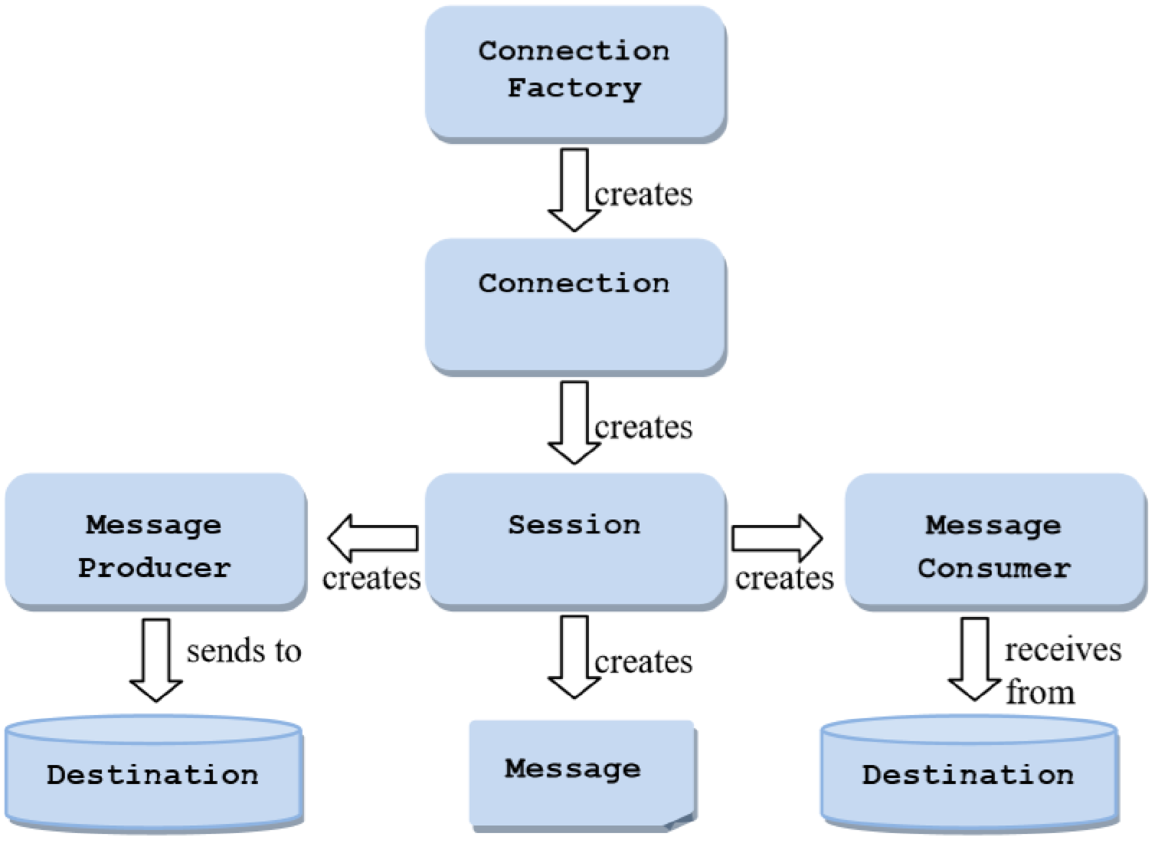
The main interfaces common to multiple APIs are as follows:

- **Message, BytesMessage, MapMessage, ObjectMessage, StreamMessage** and **TextMessage** – a message sent to or received from a JMS provider.
- **Queue** – an administered object that encapsulates the identity of a message destination for point-to-point messaging
- **Topic** – an administered object that encapsulates the identity of a message destination for pub/sub messaging.
- **Destination** - the common supertype of Queue and Topic

3.3 Classic API interfaces 经典API接口【学会】

The main interfaces provided by the classic API are as follows:

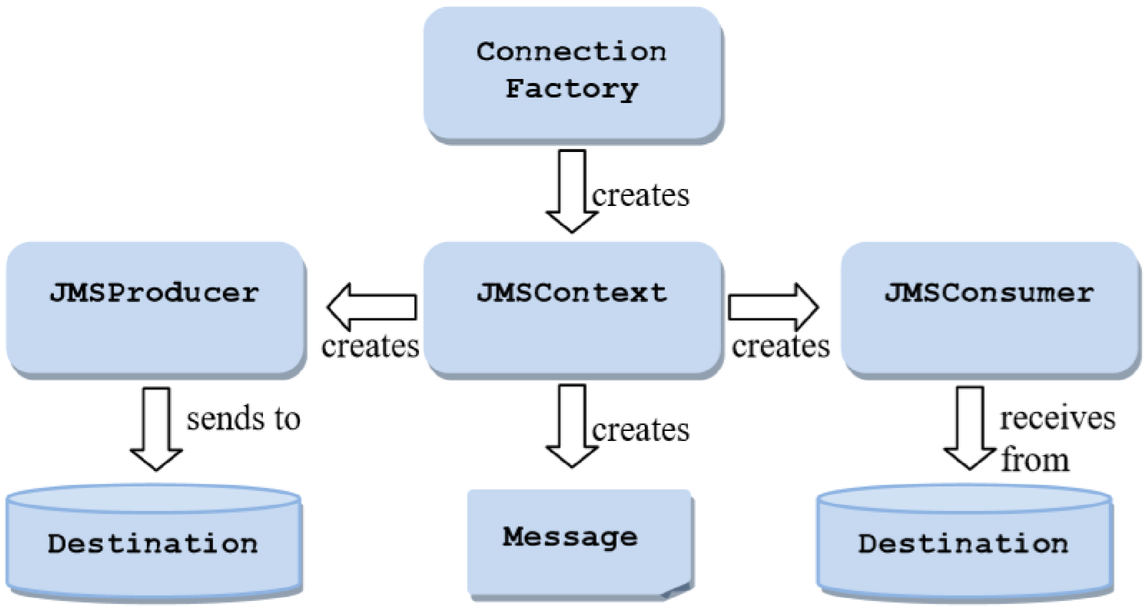
- **ConnectionFactory** - an administered object used by a client to create a Connection. This interface is also used by the simplified API.
- **Connection** - an active connection to a JMS provider
- **Session** - a single-threaded context for sending and receiving messages
- **MessageProducer** - an object created by a Session that is used for sending messages to a queue or topic
- **MessageConsumer** - an object created by a Session that is used for receiving messages sent to a queue or topic



3.4 Simplified API interfaces 简版API接口【了解】

The simplified API provides the same messaging functionality as the classic API but requires fewer interfaces and is simpler to use. The main interfaces provided by the simplified API are as follows:

- **ConnectionFactory** - an administered object used by a client to create a Connection. This interface is also used by the classic API.
- **JMSContext** - an active connection to a JMS provider and a singlethreaded context for sending and receiving messages
- **JMSProducer** - an object created by a JMSContext that is used for sending messages to a queue or topic
- **JMSConsumer** - an object created by a JMSContext that is used for receiving messages sent to a queue or topic



In the simplified API a single JMSContext object encompasses the behaviour which in the classic API is provided by two separate objects, a Connection and a Session. Although this specification refers to the JMSContext as having an underlying “connection” and “session”, the simplified API does not use the Connection and Session interfaces.

3.5 如何使用API 【掌握】

A typical JMS client using the classic API executes the following JMS setup procedure:

- Use JNDI to find a **ConnectionFactory** object
- Use JNDI to find one or more **Destination** objects
- Use the **ConnectionFactory** to create a JMS **Connection** object with message delivery inhibited
- Use the **Connection** to create one or more JMS **Session** objects
- Use a **Session** and the Destinations to create the **MessageProducer** and **MessageConsumer** objects needed
- Tell the **Connection** to **start** delivery of messages

In contrast, a typical JMS client using the simplified API does the following: 【了解】

- Use JNDI to find a **ConnectionFactory** object
- Use JNDI to find one or more **Destination** objects
- Use the **ConnectionFactory** to create a **JMSContext** object
- Use the **JMSContext** to create the **JMSProducer** and **JMSConsumer** objects needed.
- Delivery of messages is started automatically

At this point a client has the basic JMS setup needed to produce and consume messages.

3.6 多线程并发支持 【了解】

classic API

JMS Object	Supports Concurrent Use
Destination	YES
ConnectionFactory	YES
Connection	YES
Session	NO
MessageProducer	NO
MessageConsumer	NO

simplified API

JMS Object	Supports Concurrent Use
Destination	YES
ConnectionFactory	YES
JMSContext	NO
JMSProducer	NO
JMSConsumer	NO

3.7 掌握Connection的API

1. 掌握Connection连接的start、stop 、close方法的使用
2. 了解 ExceptionListener

4 消息发送

4.1 同步发送

In the **classic API** the following methods on MessageProducer may be used to send a message synchronously:

- send(Message message)
- send(Message message, int deliveryMode, int priority, long timeToLive)
- send(Destination destination, Message message, int deliveryMode, int priority, long timeToLive)

- send(Destination destination, Message message)

These methods will **block** until the message has been sent. If necessary the call will block until a confirmation message has been received back from the JMS server.

4.2 异步发送

4.2.1 异步API

In the **classic API** the following methods on MessageProducer may be used to send a message asynchronously

- send(Message message, CompletionListener completionListener)
- send(Message message, int deliveryMode, int priority, long timeToLive, CompletionListener completionListener)
- send(Destination destination, Message message, CompletionListener completionListener)
- send(Destination destination, Message message, int deliveryMode, int priority, long timeToLive, CompletionListener completionListener)

掌握 CompletionListener 接口

```
public interface CompletionListener {

    /**
     * Notifies the application that the message has been successfully sent
     *
     * @param message
     *            the message that was sent.
     */
    void onCompletion(Message message);

    /**
     * Notifies user that the specified exception was thrown while attempting to
     * send the specified message. If an exception occurs it is undefined
     * whether or not the message was successfully sent.
     *
     * @param message
     *            the message that was sent.
     * @param exception
     *            the exception
     *
     */
    void onException(Message message, Exception exception);
}
```

4.2.2 异步时消息的顺序：不需担心

Message order If the same producer is used to send multiple messages then JMS message ordering requirements (see section 6.2.9 “Message order”) must be satisfied. This applies even if a combination of synchronous and asynchronous sends has been performed. The application is not required to wait for an asynchronous send to complete before sending the next message.

4.2.3 Close, commit or rollback

A CompletionListener callback method **must not call close** on its own producer, session (including JMSContext) or connection or **call commit or rollback** on its own session. Doing so will cause the close, commit or rollback to throw an IllegalStateException or IllegalStateRuntimeException (depending on the method signature).

4.2.4 ActiveMQ异步发送支持

https://activemq.apache.org/async-sends

ActiveMQ 支持同步、异步两种模式来发送消息。选用的发送模式将很大程度上影响发送的时延。由于时延通常是生产者可以达到的吞吐量的一个重要影响因素，因此使用异步发送可以显著提高系统的性能。

ActiveMQ在某些情况下默认以异步模式发送消息。只有在JMS规范要求使用同步发送的情况下，我们才默认同步发送。还有一种情况就是在非事务下发送持久消息，将以同步模式发送。

如果你不使用事务，且发送持久消息，那么每次发送都会同步并阻塞，直到broker向生产者发送确认消息，告知已安全持久存储到磁盘为止。此ack提供了消息不会丢失的保证，但由于客户机被阻塞，它还需要付出巨大的延迟代价。

许多高性能应用程序被设计为在故障场景中能够容忍少量消息丢失。如果您的应用程序是以这种方式设计的，那么即使在使用持久消息时，也可以启用异步发送来增加吞吐量。

Configuring Async Send using a Connection URI

You can use the Connection Configuration URI to configure async sends as follows

```
cf = new ActiveMQConnectionFactory("tcp://locahost:61616?
jms.useAsyncSend=true");
```

Configuring Async Send at the ConnectionFactory Level

You can enable this feature on the ActiveMQConnectionFactory object using the property.

```
((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);
```

Configuring Async Send at the Connection Level

Configuring the dispatchAsync setting at this level overrides the settings at the connection factory level.

You can enable this feature on the ActiveMQConnection object using the property.

```
((ActiveMQConnection)connection).setUseAsyncSend(true);
```

4.3 发送有过期时间消息

A client can specify a time-to-live value in milliseconds for each message it sends. This is used to determine the message’s expiration time which is calculated by adding the time-to-live value specified on the send method to the time the message was sent (for transacted sends, this is the time the client sends the message, not the time the transaction is committed).

4.4发送延时消息

A client can specify a delivery delay value in milliseconds for each message it sends. This is used to determine the message’s delivery time which is calculated by adding the delivery delay value specified on the send method to the time the message was sent (for transacted sends, this is the time the client sends the message, not the time the transaction is committed).

A message’s delivery time is the earliest time when a JMS provider may deliver the message to a consumer. The provider must not deliver messages before the delivery time has been reached.

If a message is published to a topic, it will only be added to a durable or non-durable subscription on that topic if the subscription exists at the time the message is sent.

An application may specify the required delivery delay using the method setDeliveryDelay on the producer object. This sets the delivery delay of all messages sent using that producer. Note however that the setDeliveryDelay method on Message cannot be used to set the delivery delay of a message.

ActiveMQ 延时、定时消息传递

schedulerSupport="true"

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

https://activemq.apache.org/delay-and-schedule-message-delivery

CRON表达式说明

```
.---------------------      minute (0 - 59)
|   .------------------     hour (0 - 23)
|   |   .---------------    day of month (1 - 31)
|   |   |   .------------   month (1 - 12) - 1 = January
|   |   |   |   .---------  day of week (0 - 7) (Sunday=0 or 7
|   |   |   |   |
*   *   *   *   *
```

4.5 发送更优先的消息

priority

4.6 设置消息的传递模式

JMS supports two modes of message delivery.

- The NON_PERSISTENT mode is the lowest overhead delivery mode because it does not require that the message be logged to stable storage. A JMS provider failure can cause a NON_PERSISTENT message to be lost.
- The PERSISTENT mode instructs the JMS provider to take extra care to ensure the message is not lost in transit due to a JMS provider failure.

A JMS provider must deliver a NON_PERSISTENT message at-most-once. This means it may lose the message, but it must not deliver it twice.

A JMS provider must deliver a PERSISTENT message once-and-only-once. This means a JMS provider failure must not cause it to be lost, and it must not deliver it twice.

PERSISTENT (once-and-only-once) and NON_PERSISTENT (at-most-once) message delivery are a way for a JMS client to select between delivery techniques that may lose a messages if a JMS provider dies and those which take extra effort to ensure that messages can survive such a failure. There is typically a performance/reliability trade-off implied by this choice. When a client selects the NON_PERSISTENT delivery mode, it is indicating that it values performance over reliability; a selection of PERSISTENT reverses the requested trade-off.

4.7 消息顺序

了解消息顺序。默认是时间先后顺序，明白哪些情况将对顺序产生影响。

- Messages of higher priority may jump ahead of previous lower-priority messages.
- Messages with a later delivery time may be delivered after messages with an earlier delivery time.
- A client may not receive a NON_PERSISTENT message due to a JMS provider failure.
- If both PERSISTENT and NON_PERSISTENT messages are sent to a destination, order is only guaranteed within delivery mode. That is, a later NON_PERSISTENT message may arrive ahead of an earlier PERSISTENT message; however, it will never arrive ahead of an earlier NON_PERSISTENT message with the same priority.
- A client may use a transacted session to group its sent messages into atomic units (the producer component of a JMS transaction). A transaction’s order of messages to a particular destination is significant. The order of sent messages across destinations is not significant. See Section 6.2.7 “Transactions” for more information.

5 消息接收

5.1 Queue消息接收

掌握Queue 的Consumer创建

5.2 Topic消息接收

掌握各种不同订阅方式

5.2.1 非共享非持久化订阅

An unshared non-durable subscription is the simplest way to consume messages from a topic.

An unshared non-durable subscription is created, and a consumer object created on that subscription, using one of the following methods:

- In the **classic API**, one of several **createConsumer methods on Session**. These return a MessageConsumer object.
- In the simplified API, one of several createConsumer methods on JMSContext. These return a JMSConsumer object.

Each unshared non-durable subscription has a single consumer. If the application needs to create multiple consumers on the same subscription then a shared non-durable subscription should be used instead.

The **noLocal** parameter may be used to specify that messages published to the topic by its own connection must not be added to the subscription.

5.2.2 非共享持久化订阅

A durable subscription is used by an application that needs to receive all the messages published on a topic, including the ones published when there is no consumer associated with it. The JMS provider retains a record of this durable subscription and ensures that all messages from the topic’s publishers are retained until they are delivered to, and acknowledged by, a consumer on the durable subscription or until they have expired.

An unshared durable subscription may have only one active (i.e. not closed) consumer at the same time.

An unshared durable subscription is created, and a consumer created on that subscription, using one of the following methods:

- In the **classic API**, one of several **createDurableConsumer methods on Session**. These return a MessageConsumer object.
- In the **simplified API**, one of several **createDurableConsumer methods on JMSContext**. These return a JMSConsumer object.

必须指定订阅名

```
MessageConsumer createDurableConsumer(Topic topic, String name) throws
JMSException;
```

An unshared durable subscription is identified by a name specified by the client and by the client identifier, which must be set. A client which subsequently wishes to create a consumer on that unshared durable subscription must use the same client identifier.

取消持久订阅的方式：

An unshared durable subscription is persisted and will continue to accumulate messages until it is deleted using the **unsubscribe method on the Session, JMSContext or TopicSession**. It is erroneous for a client to delete a durable subscription while it has an active consumer or while a message received from it is part of a current transaction or has not been acknowledged in the session.

```
void unsubscribe(String name) throws JMSException;
```

5.2.3 共享非持久化订阅

A non-durable shared subscription is used by a client that needs to be able to share the work of receiving messages from a non-durable topic subscription amongst multiple consumers. A non-durable shared subscription may therefore have more than one consumer. Each message from the subscription will be delivered to only one of the consumers on that subscription.

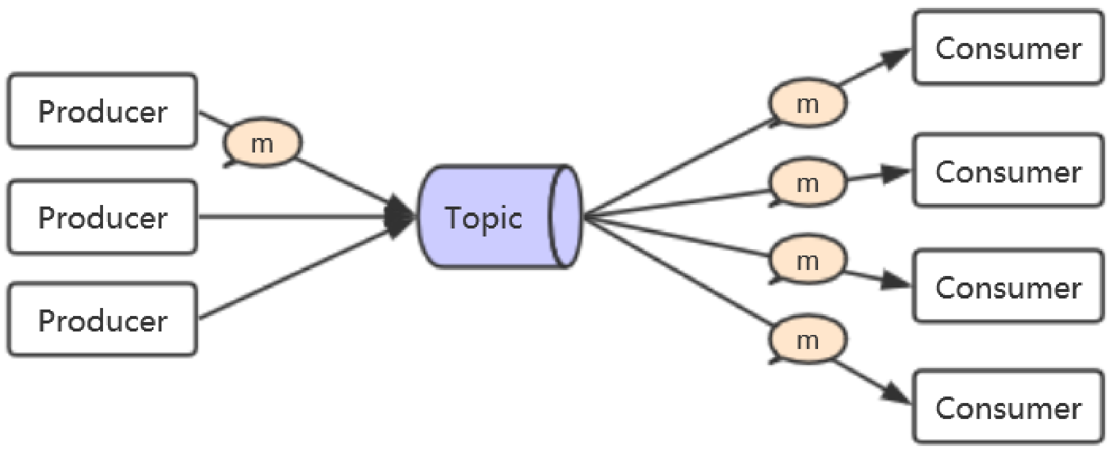
A shared non-durable subscription is created, and a consumer created on that subscription, using one of the following methods:

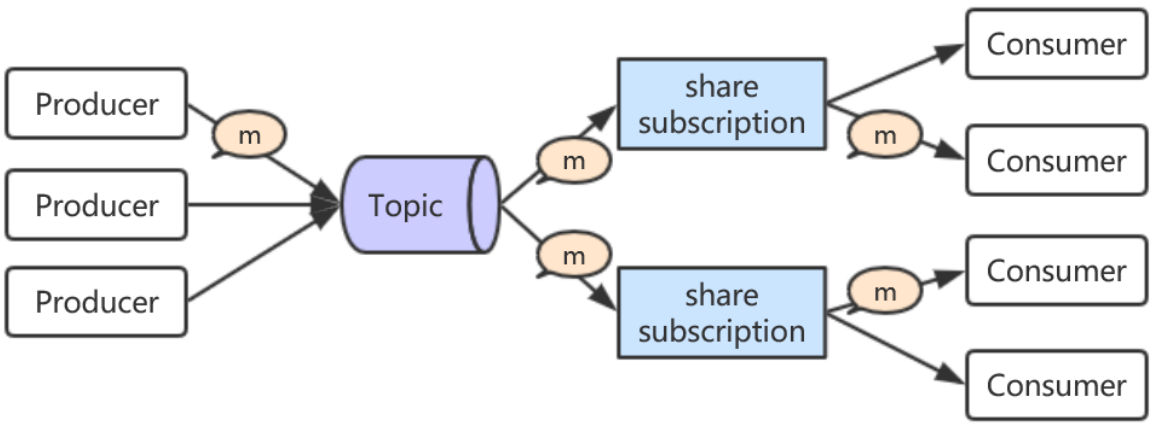
- In the **classic API**, one of several **createSharedConsumer methods on Session**. These return a MessageConsumer object.
- In the **simplified API**, one of several **createSharedConsumer methods on JMSContext**. These return a JMSConsumer object.

必须指定订阅名（自定义）

```
MessageConsumer createSharedConsumer(Topic topic, String sharedSubscriptionName)
throws JMSException;
```

share 图示说明





目的:

负载均衡

容错 高可用

5.2.4 共享持久化订阅

A durable subscription is used by an application that needs to receive all the messages published on a topic, including the ones published when there is no consumer associated with it. The JMS provider retains a record of this durable subscription and ensures that all messages from the topic's publishers are retained until they are delivered to, and acknowledged by, a consumer on the durable subscription or until they have expired.

A shared non-durable subscription is used by a client that needs to be able to share the work of receiving messages from a durable subscription amongst multiple consumers. A shared durable subscription may therefore have more than one consumer. Each message from the subscription will be delivered to only one of the consumers on that subscription.

A shared durable subscription is created, and a consumer created on that subscription, using one of the following methods:

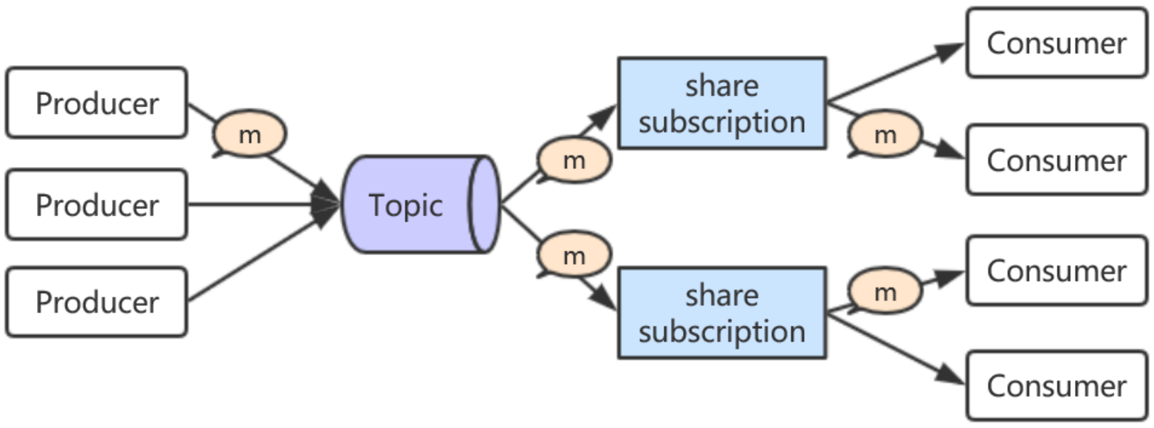
- In the classic API, one of several **createSharedDurableConsumer methods on Session**. These return a MessageConsumer object.
- In the simplified API, one of several **createSharedDurableConsumer methods on JMSContext**. These return a JMSConsumer object.

```
MessageConsumer createSharedDurableConsumer(Topic topic, String name) throws
JMSException;
```

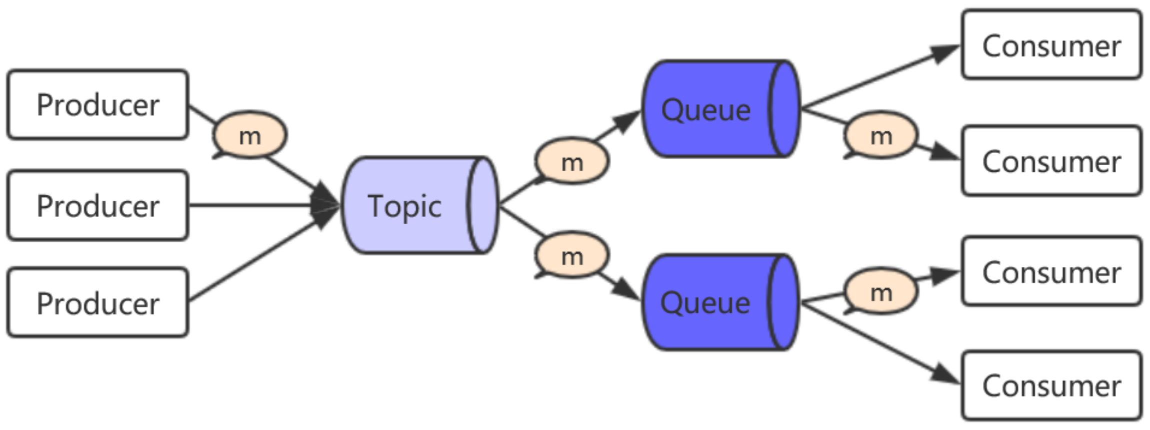
5.2.5 ActiveMQ 中的共享订阅实现

原理

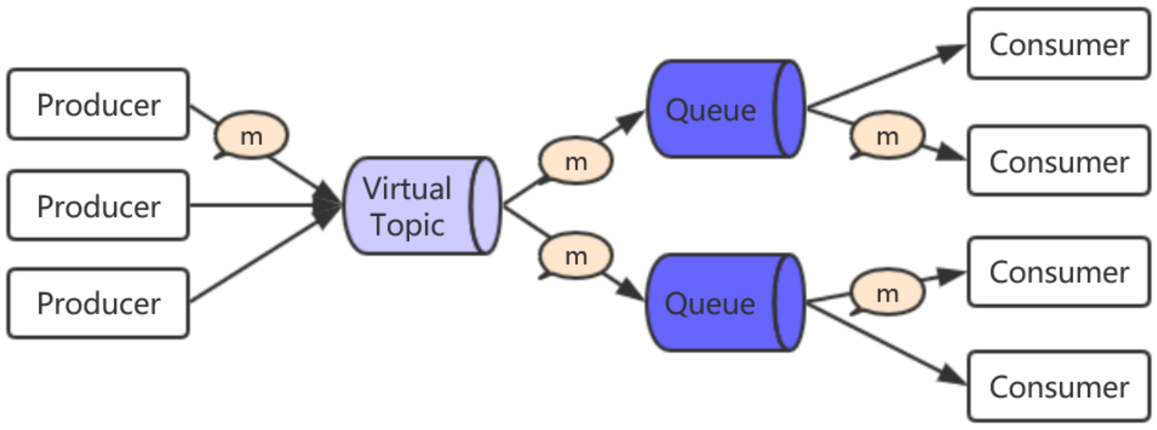
JMS中共享是2.0规范定义的。ActiveMQ只实现了1.1规范，它早就提出它的共享解决办法。



上图和下图是不是一个道理



ActiveMQ Virtual Topic



官网说明文档：https://activemq.apache.org/virtual-destinations

配置

在activemq.xml的broker节点下增加如下配置

```
<destinationInterceptors>
  <virtualDestinationInterceptor>
    <virtualDestinations>
        <!--
            name：主题名，可以是通配符
            prefix：队列的前缀
            selectorAware：表示从Topic中将消息转发给Queue时，是否关注Consumer的
selector情况。如果为false，那么Topic中的消息全部转发给Queue，否则只会转发匹配Queue
Consumer的selector的消息
            -->
      <virtualTopic name="VirtualTopic.>" prefix="VirtualTopicConsumers.*."
selectorAware="false"/>
      <virtualTopic name="aa" prefix="VirtualTopicConsumers.*."
selectorAware="false"/>
    </virtualDestinations>
  </virtualDestinationInterceptor>
</destinationInterceptors>
```

通配符说明：https://activemq.apache.org/wildcards

ActiveMQ 的queue和topic名称支持通配符，通过通配符可以增强queue和topic的功能

. 用于分隔path名称

* 用于匹配任何path名称

> 用于递归匹配以xx名称开头的任何目标

示例：

- com.study.*.mq 匹配com.study.a.mq,不匹配com.study.a.b.mq
- com.> com.study.a.mq和com.study.a.b.mq都匹配

通配符可用于生产和消费者，在生产者中使用通配符时，消息将发送到所有匹配的目标上；在消费者中使用通配符时，将接收所有匹配的目标的消息。

代码

生成者向**虚拟Topic**发送消息（注意：目标是Topic）

```
new ProducerThread("tcp://mq.study.com:61616", "VirtualTopic.Order").start();
```

```
Destination destination = session.createTopic(destinationUrl);
```

多个消费者从同一**队列**消费消息

```
    public static void main(String[] args) {
        new ConsumerThread("tcp://mq.study.com:61616",
"VirtualTopicConsumers.A.VirtualTopic.Order").start();
        new ConsumerThread("tcp://mq.study.com:61616",
"VirtualTopicConsumers.A.VirtualTopic.Order").start();
        new ConsumerThread("tcp://mq.study.com:61616",
"VirtualTopicConsumers.B.VirtualTopic.Order").start();
        new ConsumerThread("tcp://mq.study.com:61616",
"VirtualTopicConsumers.B.VirtualTopic.Order").start();
    }
```

```
Destination destination = session.createQueue(destinationUrl);
```

5.3 同步方式接收消息

方法	说明
Message receive ();	Returns the next message produced for this JMSConsumer
Message receive (long timeout);	Returns the next message produced for this JMSConsumer that arrives within the specified timeout period
Message receiveNoWait();	Returns the next message produced for this JMSConsumer if one is immediately available

5.4 异步方式接收消息

A client can register an object that implements the JMS MessageListener interface with a consumer. As messages arrive for the consumer, the provider delivers them by calling the listener's onMessage method. It is possible for a listener to throw a RuntimeException; however, this is considered a client programming error. Well behaved listeners should catch such exceptions and attempt to divert messages causing them to some form of application-specific 'unprocessable message' destination.

```
            // 6、异步接收消息
            consumer.setMessageListener(new MessageListener() {

                @Override
                public void onMessage(Message message) {
                    if (message instanceof TextMessage) {
                        try {
                            System.out.println(
                                    "收到文本消息: " + ((TextMessage)
message).getText());
                        } catch (JMSException e) {
                            e.printStackTrace();
                        }
                    } else {
                        System.out.println(message);
                    }
                }
            });
```

异步接收时如果listener抛出RuntimeException，此时的处理结果与 session's acknowledgment mode有关：

- AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE - 消息将立马重发，重发几次由provider定。JMSRedelivered 头和 JMSXDeliveryCount 属性都会被设置。
- CLIENT_ACKNOWLEDGE - 传递下一条消息。如果客户端向要provider重复上一条未确认的消息，则需手动调用session.recover()来恢复session（recover的说明见下节消息确认）。
- Transacted Session - 传递下一条消息。客户端可以选择commit or rollback 会话 (in other words, a RuntimeException does not automatically rollback the session)。

【注意】：一个Session是使用单线程来执行它所有的message listeners。

The result of a listener throwing a RuntimeException depends on the session's acknowledgment mode.

- AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE - the message will be immediately redelivered. The number of times a JMS provider will redeliver the same message before giving up is provider-dependent. The JMSRedelivered message header field will be set, and the JMSXDeliveryCount message property incremented, for a message redelivered under these circumstances.
- CLIENT_ACKNOWLEDGE - the next message for the listener is delivered. If a client wishes to have the previous unacknowledged message redelivered, it must manually recover the session.
- Transacted Session - the next message for the listener is delivered. The client can either commit or roll back the session (in other words, a RuntimeException does not automatically rollback the session).

JMS providers should flag clients with message listeners that are throwing RuntimeException as possibly malfunctioning.

A session uses a single thread to run all its message listeners. While the thread is busy executing one listener, all other messages to be asynchronously delivered to the session must wait.

5.5 消息确认

消息确认有四种方式：

- **事务控制方式**：如果session是开启事务的，消息的确认是由commit来自动处理的，而恢复则是由rollback自动处理。If a session is transacted, message acknowledgment is handled automatically by commit, and recovery is handled automatically by rollback.

如果session是未开启事务的，则有下面三种可选确认方式，而恢复则需手动来控制。If a session is not transacted, there are three acknowledgment options and recovery is handled manually:

- **DUPS_OK_ACKNOWLEDGE** - This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails. It it should therefore only be used by consumers that are tolerant of duplicate messages. Its benefit is the reduction of session overhead achieved by minimizing the work the session does to prevent duplicates. **懒惰(延缓）确认收到消息**，适用于能容忍重复消息的情况，它带来的好处是减轻session判断重复消息的工作。一般实现可以是异步、批量方式（ActiveMQ）
- **AUTO_ACKNOWLEDGE** - With this option, the session automatically acknowledges a client's receipt of a message when it has either successfully returned from a call to receive or the message listener it has called to process the message successfully returns. **Session自动确认**，在成功收到消息后或在成功从message listener的消息方法处理返回后，自动确认消息。
- **CLIENT_ACKNOWLEDGE** - With this option, a client acknowledges a message by calling the **message's acknowledge** method. Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been delivered by its session. **客户端自己来回复确认**（我们自己来控制回复确认）

```
// 3、创建会话（可以创建一个或者多个session）
// 自动回复消息确认
//session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
// 懒惰回复消息确认（在可以容忍重复消息的情况下使用，可提高效率）
// session = conn.createSession(false, Session.DUPS_OK_ACKNOWLEDGE);
// 用户手动回复消息确认
session = conn.createSession(false, Session.CLIENT_ACKNOWLEDGE);
// 开启事务，由事务控制来回复消息确认
// session = conn.createSession(false, Session.SESSION_TRANSACTED);
```

客户端自己来回复确认：

```
message.acknowledge();
```

session’s recover method

stop session 并以第一个未确认消息重新开始。

A **session’s recover method** is used to stop a session and restart it with its first unacknowledged message. In effect, the session’s series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered due to message expiration, the arrival of higher-priority messages, or the delivery of messages which could not previously be delivered as they had not reached their specified delivery time.

5.6 Selector

见 7.3 章节

6 事务处理

掌握事务管理（概念、操作方法），了解分布式事务

Transactions organize a session’s input message stream and output message stream into a series of atomic units. When a transaction commits, its atomic unit of input is acknowledged and its associated atomic unit of output is sent. If a transaction rollback is done, its produced messages are destroyed and its consumed messages are automatically recovered.

A transaction is completed using either its **session’s commit() or rollback()** method. The completion of a session’s current transaction automatically begins the next. The result is that a transacted session always has a current transaction within which its work is done.

```
// 3、创建会话（可以创建一个或者多个session）// 开启事务
session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

```
// 6、创建文本消息
for (int i = 0; i < 10; i++) {
    String text = i + "message! From: " + Thread.currentThread().getName() ;
    TextMessage message = session.createTextMessage(text);
```

```
        // 7、通过producer 发送消息
        producer.send(message);

        System.out.println("Sent message: " + text);
        Thread.sleep(1000L);
    }

    // 提交事务
    session.commit();

    // 回滚事务
    // session.rollback();
```

Spring JTA

spring 中使用时，一般都是数据库+MQ的场景，需要用JTA来进行分布式事务管理.

1. 引入分布式事务管理实现

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jta-atomikos</artifactId>
        </dependency>
```

2. 配置数据源

```
spring:
  jta:
    log-dir: d:/tmp    # 可不配，使用默认目录
  activemq:
    broker-url: tcp://mq.study.com:61616
    pool:
      enabled: true
      max-connections: 50
    #user: admin
    #password: secret

#数据源参数配置
spring.datasource:
    url: jdbc:mysql://127.0.0.1:3306/test?
useUnicode=true&characterEncoding=utf-8&serverTimezone=UTC
    username: root
    password: root
```

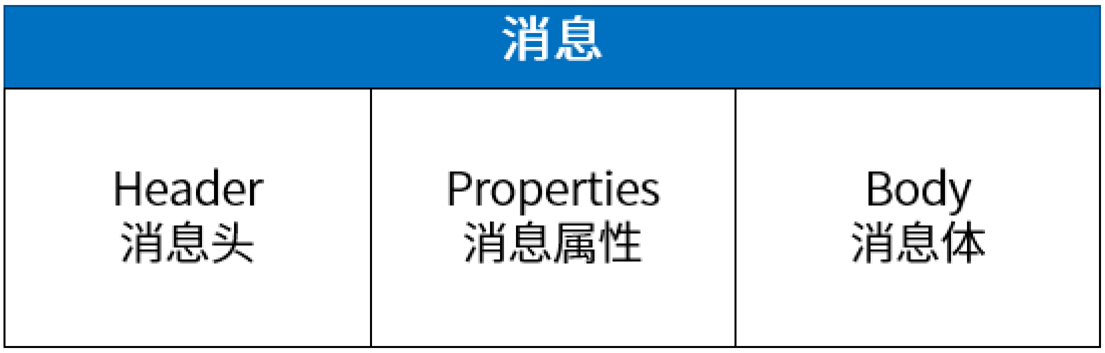
3. 像平常一样使用@Transactional 注解

```
package com.study.activemq.le2_example.spring.transaction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.transaction.annotation.Transactional;

@SpringBootApplication
public class TransactionSample {
    @Autowired
    private JmsTemplate jmsTemplate;
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional
    public void saveDbAndSendMessage(String data) {
        this.jmsTemplate.convertAndSend("transaction-test", data);
        System.out.println("发送消息完成： " + data);
        this.jdbcTemplate.update("insert t_log(id,log) values(?,?)",
System.currentTimeMillis(), data);
    }

    @Transactional
    public void reciveMessageAndSaveDb() {
        String data = (String)
this.jmsTemplate.receiveAndConvert("transaction-test");
        this.jdbcTemplate.update("insert t_log(id,log) values(?,?)",
System.currentTimeMillis(), data);
    }

    // @Transactional 异步方式下，加 @Transactional 无效，而是走异步异常根据消息确认
机制来处理 ，
    // 默认的消息确认模式 为
    // AUTO_ACKNOWLEDGE,抛出异常，则会立马重发，重发provider指定的次数（这里可看出默
认配置的重发次数为6），
    // 重发指定次数后还是不成功，则消息将被转移到死信队列 ActiveMQ.DLQ
    @JmsListener(destination = "transaction-test")
    public void reciveMessageAndSaveDb2(String data) {
        System.out.println("收到消息： " + data);
        this.jdbcTemplate.update("insert t_log(id,log) values(?,?)",
System.currentTimeMillis(), data);
    }

    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(TransactionSample.class, args);
        TransactionSample ts = context.getBean(TransactionSample.class);
        // 发送的示例
        ts.saveDbAndSendMessage("aaaaaaaaaa aaaaaaaaaa");
        // 接收的示例，请把异步接收注释掉先
        // ts.reciveMessageAndSaveDb();
    }
}
```

7 JMS消息模型【掌握】

JMS Message Mode

消息由三部分构成，Header、Body 必需，Properties 非必需



- **Header** - All messages support the same set of header fields. Header fields contain values used by both clients and providers to identify and route messages.
- **Properties** - In addition to the standard header fields, messages provide a built-in facility for adding optional header fields to a message.
  - Application-specific properties - In effect, this provides a mechanism for adding application specific header fields to a message.
  - Standard properties - JMS defines some standard properties that are, in effect, optional header fields.
  - Provider-specific properties – Some JMS providers may require the use of provider-specific properties. JMS defines a naming convention for these.
- **Body** - JMS defines several types of message body which cover the majority of messaging styles currently in use.

API接口 Message

7.1 Message header field

掌握字段含义，值是由谁来设置

头字段名	说明	Set By	Setter method
JMSDestination	消息的目的地，Topic或者是Queue	JMS provider send method	setJMSDestination (not for client use)
JMSDeliveryMode	消息的发送模式	JMS provider send method	setJMSDeliveryMode(not for client use)
JMSTimestamp	消息传递给Broker的时间戳，它不是实际发送的时间	JMS provider send method	setJMSTimestamp (not for client use)
JMSExpiration	消息的有效期，在有效期内，消息消费者才可以消费这个消息	JMS provider send method	setJMSExpiration (not for client use)
JMSPriority	消息的优先级。0-4为正常的优先级，5-9为高优先级	JMS provider send method	setJMSPriority (not for client use)
JMSMessageID	一个字符串用来唯一标示一个消息	JMS provider send method	setJMSMessageID (not for client use)
JMSReplyTo	有时消息生产者希望消费者回复一个消息，JMSReplyTo为一个Destination，表示需要回复的目的地	Client application	setJMSReplyTo
JMSCorrelationID	通常用来关联多个Message	Client application	setJMSCorrelationID, setJMSCorrelationIDAsByte
JMSType	表示消息体的结构，和JMS提供者有关	Client application	setJMSType
JMSRedelivered	如果这个值为true，表示消息是被重新发送了	JMS provider prior to delivery	setJMSRedelivered (not for client use)

设置消息头

Prior to sending a message, the application may use methods on the Message object to set the JMSCorrelationID, JMSReplyTo and JMSType message headers.

7.2 Message properties

JSM 规范文档： 3.5. Message properties

掌握命名规范、值的数据类型范围

Property names

Property names must obey the rules for a message selector identifier. See Section 3.8 “Message selection” for more information.

Property values

Property values can be boolean, byte, short, int, long, float, double, and String.

设置消息属性

Prior to sending a message, the client application may use methods on the Message object to set message properties.

7.3 Message selection 【掌握】

JSM 规范文档：3.8. Message selection

Selection 是消费端选择性消费消息的功能，通过使用消息头、消息属性中的字段来编写一个SQL条件表达式来说明要消息提供者如何选择性递送消息给消费者。

A message selector is a String whose syntax is based on a subset of the SQL92 conditional expression syntax.

看Session的创建Consumer方法：

```
MessageConsumer
    createConsumer(Destination destination, String messageSelector)
```

JMS 示例：

消费者:

```
// 5、创建消息消费者
MessageConsumer consumer = session.createConsumer(destination, "subtype='aaa'");
```

生产者：

```
String text = "message! From: " + Thread.currentThread().getName() + " : "
                            + System.currentTimeMillis();
TextMessage message = session.createTextMessage(text);

// 设置消息属性
message.setStringProperty("subtype", “aaa");
// 7、通过producer 发送消息
producer.send(message);
```

Spring 示例

消费者:

```
    @JmsListener(destination = "queue1", selector = "subtype='bbb'")
    public void receive(String text, @Header("subtype") String subtype) {
        System.out.println(Thread.currentThread().getName() + " Received <" +
text + "> subtype=" + subtype);
    }
```

生产者：

```
    @Transactional // 在开启事务的情况下需要加事务注解
    public void sendMessage(String subtype) {

        // 发送延时消息
        jmsTemplate.convertAndSend("queue1", "message with property", message ->
{
            // 设置消息属性
            message.setStringProperty("subtype", subtype);
            return message;
        });

        System.out.println("Sending an message with subtype=" + subtype);
    }
```

7.4 Message Body

消息体实现	说明
BytesMessage	用来传递字节消息
MapMessage	用来传递键值对消息
ObjectMessage	用来传递序列化对象
StreamMessage	用来传递文件等
TextMessage	用来传递字符串

8 Spring-JMS API 详解

JmsTemplate

是简化JMS同步操作的帮助类。学习要点：

- 掌握各类简便方法
- 掌握execute(..)方法来自己操作、设置jms api
- 掌握可配置属性及参数方式配置属性 spring.jms.template.*

注意点：

1. 默认的目标类型为Point-to-Point (Queues)， 如是Topic，通过pubSubDomain 属性设置（true)

2. JMS Session 默认设置为"not transacted" and "auto-acknowledge",如果 JMS Session是在事务中创建的，则这两个参数将被忽略，是通过事务来控制。而如果是使用原生JMS，可通过 "sessionTransacted" and "sessionAcknowledgeMode" bean properties来设置。
3. JmsTemplate 使用 DynamicDestinationResolver 和 SimpleMessageConverter 作为目标名解析和消息转换的默认策略。可通过 "destinationResolver" and "messageConverter" bean properties 来覆盖默认值。
4. JmsTemplate使用的ConnectionFactory需返回池化的Connections(or a single shared Connection)

正确的用法:

1. 至少要分两个JmsTempate queue topic
2. 如果有多种业务队列/topic 他们的方式各不相同，则需要按业务队列/topic 配置topic

```
@Configuration
public class JmsConfiguration {
    @Primary
    @Bean
    public JmsTemplate queueJmsTemplate(ConnectionFactory connectionFactory) {
        PropertyMapper map = PropertyMapper.get();
        JmsTemplate template = new JmsTemplate(connectionFactory);
        // template.setDestinationResolver(destinationResolver);
        template.setSessionAcknowledgeMode(Session.AUTO_ACKNOWLEDGE);
        return template;
    }

    @Bean
    public JmsTemplate topicJmsTemplate(ConnectionFactory connectionFactory) {
        PropertyMapper map = PropertyMapper.get();
        JmsTemplate template = new JmsTemplate(connectionFactory);
        // template.setDestinationResolver(destinationResolver);

        return template;
    }
}
```

Helper class that simplifies synchronous JMS access code.

If you want to use dynamic destination creation, you must specify the type of JMS destination to create, using the "pubSubDomain" property. For other operations, this is not necessary. Point-to-Point (Queues) is the default domain. 默认的目标类型为Point-to-Point (Queues)，如是Topic，通过 pubSubDomain 属性设置（true)

Default settings for JMS Sessions are "not transacted" and "auto-acknowledge". As defined by the Java EE specification, the transaction and acknowledgement parameters are ignored when a JMS Session is created inside an active transaction, no matter if a JTA transaction or a Spring-managed transaction. To configure them for native JMS usage, specify appropriate values for the "sessionTransacted" and "sessionAcknowledgeMode" bean properties.

This template uses a `org.springframework.jms.support.destination.DynamicDestinationResolver` and a `org.springframework.jms.support.converter.SimpleMessageConverter` as default strategies for resolving a destination name or converting a message, respectively. These defaults can be overridden through the "destinationResolver" and "messageConverter" bean properties.

NOTE: The ConnectionFactory used with this template should return pooled Connections (or a single shared Connection) as well as pooled Sessions and MessageProducers. Otherwise, performance of ad-hoc JMS operations is going to suffer. The simplest option is to use the Spring-provided `org.springframework.jms.connection.SingleConnectionFactory` as a decorator for your target `ConnectionFactory`, reusing a single JMS Connection in a thread-safe fashion; this is often good enough for the purpose of sending messages via this template. In a Java EE environment, make sure that the `ConnectionFactory` is obtained from the application's environment naming context via JNDI; application servers typically expose pooled, transaction-aware factories there.

JmsMessagingTemplate

在JmsTemplate上增加了一些方法

covertAndsent

MessageConverter

掌握它的用途，已有实现

The default `MessageConverter` is able to convert only basic types (such as `String`, `Map`, `Serializable`) and our `Email` is not `Serializable` on purpose. We want to use Jackson and serialize the content to json in text format (i.e. as a `TextMessage`). Spring Boot will detect the presence of a `MessageConverter` and will associate it to both the default `JmsTemplate` and any `JmsListenerContainerFactory` created by `DefaultJmsListenerContainerFactoryConfigurer`.

```
    @Bean // Serialize message content to json using TextMessage
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new
MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_type");
        return converter;
    }
```

@JmsListener

掌握@JmsListener各注解项的意义用途

掌握开启@JmsListener注解支持的方式

Processing of `@JmsListener` annotations is performed by registering a `JmsListenerAnnotationBeanPostProcessor`. This can be done manually or, more conveniently, through the `<jms:annotation-driven/>` element or `@EnableJms` annotation.

掌握@JmsListener注释方法的方法签名可定义参数，及返回值的用途

Annotated JMS listener methods are allowed to have flexible signatures similar to what `MessageMapping` provides:

- `javax.jms.Session` to get access to the JMS session
- `javax.jms.Message` or one of its subclasses to get access to the raw JMS message
- `org.springframework.messaging.Message` to use Spring's messaging abstraction counterpart
- `@Payload`-annotated method arguments, including support for validation
- `@Header`-annotated method arguments to extract specific header values, including standard JMS headers defined by `org.springframework.jms.support.JmsHeaders`
- `@Headers`-annotated method argument that must also be assignable to `java.util.Map` for obtaining access to all headers
- `org.springframework.messaging.MessageHeaders` arguments for obtaining access to all headers
- `org.springframework.messaging.support.MessageHeaderAccessor` or `org.springframework.jms.support.JmsMessageHeaderAccessor` for convenient access to all method arguments

Annotated methods may have a non-`void` return type. When they do, the result of the method invocation is sent as a JMS reply to the destination defined by the `JMSReplyTO` header of the incoming message. If this header is not set, a default destination can be provided by adding `@SendTo` to the method declaration.

掌握DefaultJmsListenerContainerFactory 的配置

Annotation that marks a method to be the target of a JMS message listener on the specified `destination`. The `containerFactory` identifies the `org.springframework.jms.config.JmsListenerContainerFactory` to use to build the JMS listener container. If not set, a *default* container factory is assumed to be available with a bean name of `jmsListenerContainerFactory` unless an explicit default has been provided through configuration.

Consider setting up a custom org.springframework.jms.config.DefaultJmsListenerContainerFactory bean. For production purposes, you'll typically fine-tune timeouts and recovery settings. Most importantly, the default 'AUTO_ACKNOWLEDGE' mode does not provide reliability guarantees, so make sure to use transacted sessions in case of reliability needs.

```
@Configuration
static class JmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory myFactory(ConnectionFactory
connectionFactory,
            DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        // This provides all boot's default to this factory, including the
        // message converter
        configurer.configure(factory, connectionFactory);
        // You could still override some of Boot's default if necessary.
        // factory.setSessionAcknowledgeMode(Session.CLIENT_ACKNOWLEDGE);
        // factory.setSessionTransacted(true);
        return factory;
    }

}
```

```
@Component
public class Receiver {

    @JmsListener(destination = "mailbox", containerFactory = "myFactory")
    public void receiveMessage(Email email) {
        System.out.println("Received <" + email + ">");
    }

}
```

This annotation may be used as a *meta-annotation* to create custom *composed annotations* with attribute overrides.

CachingConnectionFactory

CachingConnectionFactory extends SingleConnectionFactory

JmsPoolConnectionFactory

```
<dependency>
    <groupId>org.messaginghub</groupId>
    <artifactId>pooled-jms</artifactId>
</dependency>
```

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```