厦門大學

软件学院

《中间件技术》大作业报告（一）

题　　目 为 ActiveMQ 添加一个自定义的 Message

姓　　名 陈澄

学　　号 32420212202930

班　　级 软工三班

实验时间 2024/04/25

2024 年 **04** 月 **25** 日

1 实验题目

为 ActiveMQ 添加一个自定义的 Message 并支持其收发等主要功能

2 本次实验目的

阅读并 ActiveMQ 的代码并理解其消息收发流程与原理

3 实验步骤

1. 消息发送调用流程：

首先会创建一个 ConnectionFactory 类，传入代理的 URL（BROKER_URL），该对象用于创建与代理的连接，并通过该类创建一个 Connection 对象。

```
//与ActiveMq建立连接
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(BROKER_URL);
Connection connection = connectionFactory.createConnection();
connection.setClientID(USER_NAME);
connection.start();
```

其后，通过创建的 Connection 对象创建一个 Session 对象，此处会设置消息的确认模式。

```
Session session = connection.createSession( b: false, Session.AUTO_ACKNOWLEDGE);
```

创建一个 Topic 对象，作为对话的主题。

```
Topic topic = session.createTopic(CHAT_TOPIC);
```

通过 Topic 创建一个 MessageProducer，用于消息发送。

```
MessageProducer producer = session.createProducer(topic);
```

通过 Session 创建一个文本消息 TextMessage。

```
TextMessage textMessage = session.createTextMessage( s: USER_NAME + ": " + messageText);
```

通过 MessageProducer 调用 send()发送该消息。

```
producer.send(textMessage);
```

2. 消息接收调用流程：

通过 Session 创建一个 MessageConsumer 对象，通过该对象创建一个消息监听器，根据收到的消息进行处理。

```
//创建消息监听事件，便于收到其他用户消息
MessageConsumer consumer = session.createDurableSubscriber(topic, USER_NAME);
consumer.setMessageListener(message -> {
    if (message instanceof TextMessage) {
        //如果是TextMessage即为普通聊天消息
```

3. 消息发送代码：

（1）ConnectionFactory 会将传入的 String 类型转化为 URL 类型并创建一个 Connection 对象返回。

```
public ActiveMQConnectionFactory(String brokerURL) { this(createURI(brokerURL)); }
```

createSession(int acknowledgeMode) 方法用于创建一个 JMS 会话对象。

acknowledgeMode 参数表示消息确认模式。

```
@Override
public Session createSession(int acknowledgeMode) throws JMSException {
    return createSession( transacted: acknowledgeMode == Session.SESSION_TRANSACTED, acknowledgeMode);
}
```

checkClosedOrFailed() 方法用于检查连接是否已关闭或失败，

ensureConnectionInfoSent() 方法用于确保连接信息已经被发送到代理，

isDispatchAsync() 方法用于判断是否异步分发消息，

isAlwaysSessionAsync() 方法用于判断是否总是使用异步会话。

```java
@Override
public Session createSession(boolean transacted, int acknowledgeMode) throws JMSException {
    checkClosedOrFailed();
    ensureConnectionInfoSent();
    if (!transacted) {
        if (acknowledgeMode == Session.SESSION_TRANSACTED) {
            throw new JMSException("acknowledgeMode SESSION_TRANSACTED cannot be used for an non-tra
        } else if (acknowledgeMode < Session.SESSION_TRANSACTED || acknowledgeMode > ActiveMQSession
            throw new JMSException("invalid acknowledgeMode: " + acknowledgeMode + ". Valid values a
                    "Session.CLIENT_ACKNOWLEDGE (2), Session.DUPS_OK_ACKNOWLEDGE (3), ActiveMQSessio
        }
    }
    return new ActiveMQSession( connection: this, getNextSessionId(), transacted ? Session.SESSION_TRAN
}
```

（2）createProducer(Destination destination)用于创建一个 MessageProducer

同样先调用 checkClosed()检查连接是否关闭

getSendTimeout()获得连接的失效时间

随后创建一个新的 MessageProducer 对象

```java
@Override
public MessageProducer createProducer(Destination destination) throws JMSException {
    checkClosed();
    if (destination instanceof CustomDestination) {
        CustomDestination customDestination = (CustomDestination)destination;
        return customDestination.createProducer( session: this);
    }
    int timeSendOut = connection.getSendTimeout();
    return new ActiveMQMessageProducer( session: this, getNextProducerId(), ActiveMQMessageTransformati
}
```

（3）Send()方法用于消息的发送

首先，会检查连接是否已关闭。

接下来，会检查目的地是否为临时目的地，并且该目的地是否已被删除。

```java
checkClosed();
if (destination.isTemporary() && connection.isDeleted(destination)) {
    throw new InvalidDestinationException("Cannot publish to a deleted Destination: " + destination);
}
```

使用 sendMutex 对象进行同步，确保在发送消息时不会发生并发访问冲突。

调用 doStartTransaction()方法，告知 Broker 即将开始一个新的事务。

```
synchronized (sendMutex) {
    // tell the Broker we are about to start a new transaction
    doStartTransaction();
```

检查事务状态，事务不能被标记为回滚。

```
if (transactionContext.isRollbackOnly()) {
    throw new IllegalStateException("transaction marked rollback only");
}
```

设置消息的 JMS 头字段，包括交付模式（deliveryMode）、过期时间（expiration）、时间戳（timeStamp）、优先级（priority）和是否重新传递（redelivered）。

```
message.setJMSDeliveryMode(deliveryMode);
long expiration = 0L;
long timeStamp = System.currentTimeMillis();
if (timeToLive > 0) {
    expiration = timeToLive + timeStamp;
}
```

将原始消息转换为 ActiveMQ 消息格式。

```
if(!(message instanceof ActiveMQMessage)) {
    setForeignMessageDeliveryTime(message, timeStamp);
} else {
    message.setJMSDeliveryTime(timeStamp);
}
if (!disableMessageTimestamp && !producer.getDisableMessageTimestamp()) {
    message.setJMSTimestamp(timeStamp);
} else {
    message.setJMSTimestamp(0l);
}
```

根据禁用消息时间戳的设置，将消息的 JMS 时间戳（JMSTimestamp）设置为当前时间戳或 0。

设置消息的过期时间和优先级。

将消息的 JMS 重新传递标志设置为 false。

```
message.setJMSExpiration(expiration);
message.setJMSPriority(priority);
message.setJMSRedelivered(false);
```

对消息进行转换，将其转换为自定义的消息格式（ActiveMQMessage），并将连接信息传递给消息。

设置消息的生产者 ID（producerId），并进行日志记录。

```
// Set the message id.
if (msg != message) {
    message.setJMSMessageID(msg.getMessageId().toString());
    // Make sure the JMS destination is set on the foreign messages too.
    message.setJMSDestination(destination);
}
```

根据发送超时时间（sendTimeout）和完成回调函数（onComplete），选择是异步还是同步地发送消息。

```
if (onComplete==null && sendTimeout <= 0 && !msg.isResponseRequired() && !connection
    this.connection.asyncSendPacket(msg);
    if (producerWindow != null) {
        // Since we defer lots of the marshaling till we hit the
        // wire, this might not
        // provide and accurate size. We may change over to doing
        // more aggressive marshaling,
        // to get more accurate sizes.. this is more important once
        // users start using producer window
        // flow control.
        int size = msg.getSize();
        producerWindow.increaseUsage(size);
    }
} else {
    if (sendTimeout > 0 && onComplete==null) {
        this.connection.syncSendPacket(msg,sendTimeout);
    }else {
        this.connection.syncSendPacket(msg, onComplete);
    }
}
```

接着调用 connection 中的包发送方法进行发包。

4. 消息接收代码：

createConsumer()用于创建一个消息消费者

destination 参数表示消息消费的目标地址。

messageSelector 参数表示消息选择器，它是一个字符串，用于过滤要被消费的消息。

noLocal 参数表示是否禁止消费者接收自己发送的消息。

该方法会将 jms 中非 custom 的对象转化为 custom 的对象。

```
public MessageConsumer createConsumer(Destination destination, String messageSelector, boolean noLocal, MessageListener messageListener) throws JMSException {
    checkClosed();

    if (destination instanceof CustomDestination) {
        CustomDestination customDestination = (CustomDestination)destination;
        return customDestination.createConsumer( session: this, messageSelector, noLocal);
    }

    ActiveMQPrefetchPolicy prefetchPolicy = connection.getPrefetchPolicy();
    int prefetch = 0;
    if (destination instanceof Topic) {
        prefetch = prefetchPolicy.getTopicPrefetch();
    } else {
        prefetch = prefetchPolicy.getQueuePrefetch();
    }
    ActiveMQDestination activemqDestination = ActiveMQMessageTransformation.transformDestination(destination);
    return new ActiveMQMessageConsumer( session: this, getNextConsumerId(), activemqDestination, name: null, messageSelector,
            prefetch, prefetchPolicy.getMaximumPendingMessageLimit(), noLocal, browser: false, isAsyncDispatch(), messageListener);
}
```

设置消息监听器则是直接调用 jms 中的消息监听器设置方法。

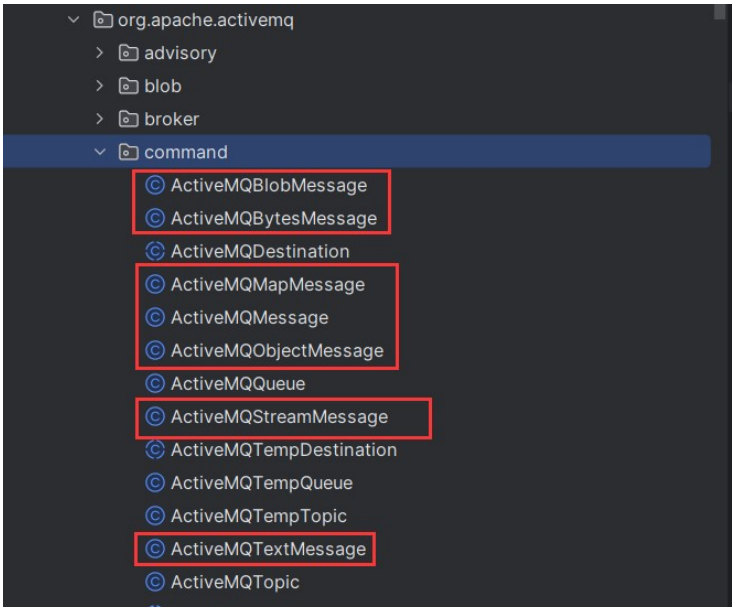
```
0 个用法
@Override
public void setMessageListener(MessageListener listener) throws JMSRuntimeException {
    try {
        activemqMessageConsumer.setMessageListener(listener);
    } catch (JMSException e) {
        throw JMSExceptionSupport.convertToJMSRuntimeException(e);
    }
}
```

5. ActiveMQ 中的 Message

ActiveMQ 中的 ActiveMQMessage 都继承自 Message 抽象类，并且实现 org.apache.activemq.Message, ScheduledMessage 两个接口。

在 ActiveMQMessage 实现了 Message 需要的主要方法。

其余各种 Message 都是继承 ActiveMQMessage




```
public class ActiveMQMessage extends Message implements org.apache.activemq.Message, ScheduledMessage {
```

例如：ActiveMQTextMessage 则是继承 ActiveMQMessage 并实现 jms 中的 TextMessage 接口。

```
public class ActiveMQTextMessage extends ActiveMQMessage implements TextMessage {
```