咕泡学院VIP课：分布式消息通信ActiveMQ原理分析

课程目标

1. 持久化消息和非持久化消息的发送策略
2. 消息的持久化方案及实践
3. 消费端消费消息的原理
4. 关于PrefetchSize的优化

持久化消息和非持久化消息的发送策略

消息同步发送和异步发送

ActiveMQ支持同步、异步两种发送模式将消息发送到broker上。

同步发送过程中，发送者发送一条消息会阻塞直到broker反馈一个确认消息，表示消息已经被broker处理。这个机制提供了消息的安全性保障，但是由于是阻塞的操作，会影响到客户端消息发送的性能

异步发送的过程中，发送者不需要等待broker提供反馈，所以性能相对较高。但是可能会出现消息丢失的情况。所以使用异步发送的前提是在某些情况下允许出现数据丢失的情况。

默认情况下，非持久化消息是异步发送的，持久化消息并且是在非事务模式下是同步发送的。

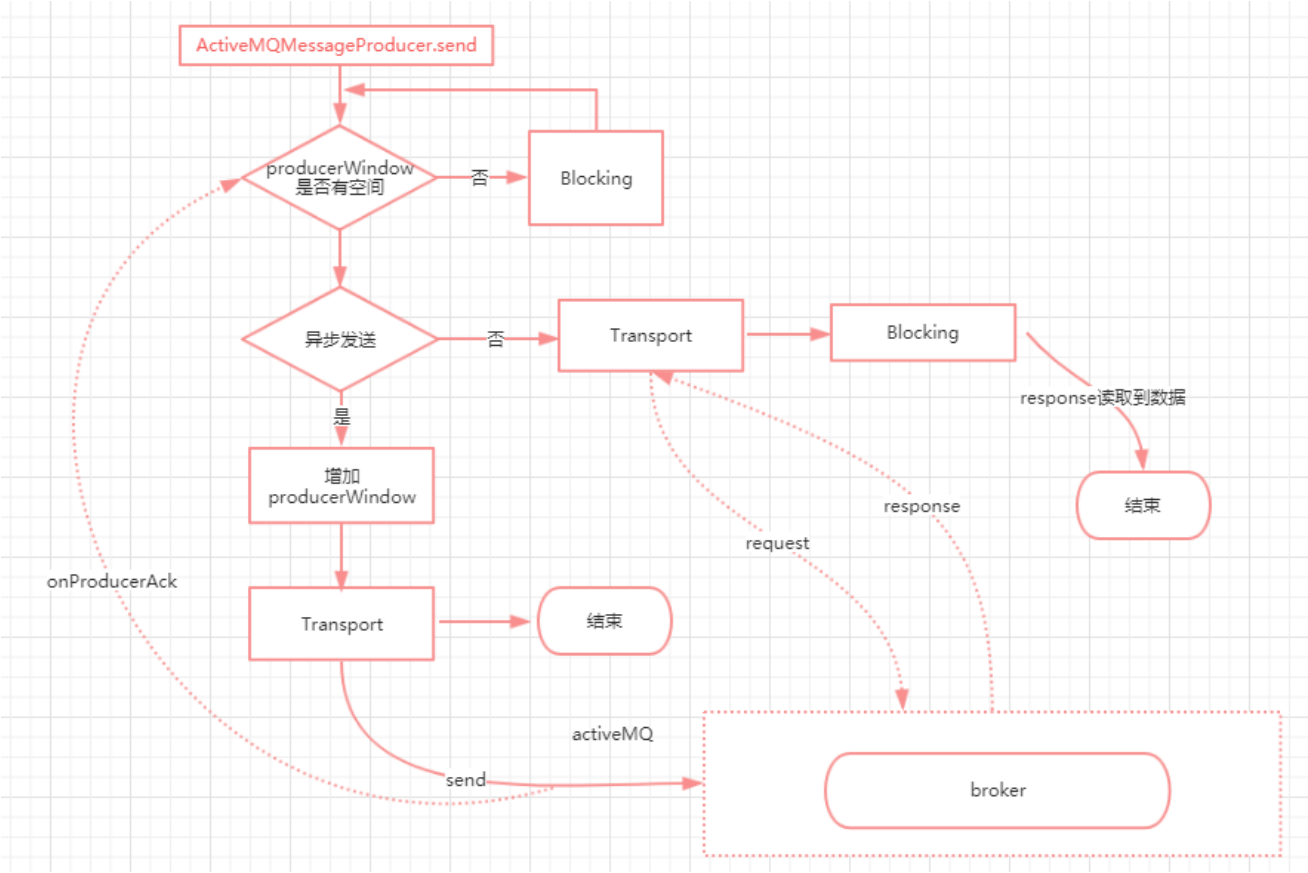
但是在开启事务的情况下，消息都是异步发送。由于异步发送的效率会比同步发送性能更高。所以在发送持久化消息的时候，尽量去开启事务会话。

除了持久化消息和非持久化消息的同步和异步特性以外，我们还可以通过以下几种方式来设置异步发送

```
1.ConnectionFactory connectionFactory=new ActiveMQConnectionFactory("tcp://192.168.11.153:61616?
jms.useAsyncSend=true");
2.((ActiveMQConnectionFactory) connectionFactory).setUseAsyncSend(true);
3.((ActiveMQConnection)connection).setUseAsyncSend(true);
```

消息的发送原理分析图解

消息发送的流程图



ProducerWindowSize的含义

producer每发送一个消息，统计一下发送的字节数，当字节数达到ProducerWindowSize值时，需要等待broker的确认，才能继续发送。

代码在：ActiveMQSession的1957行

主要用来约束在异步发送时producer端允许积压的(尚未ACK)的消息的大小，且只对异步发送有意义。每次发送消息之后，都将会导致memoryUsage大小增加(+message.size)，当broker返回producerAck时，memoryUsage尺寸减少(producerAck.size，此size表示先前发送消息的大小)。

可以通过如下2种方式设置:

Ø 在brokerUrl中设置: "tcp://localhost:61616?jms.producerWindowSize=1048576",这种设置将会对所有的producer生效。

Ø 在destinationUri中设置: "test-queue?producer.windowSize=1048576",此参数只会对使用此Destination实例的producer失效，将会覆盖brokerUrl中的producerWindowSize值。

注意：此值越大，意味着消耗Client端的内存就越大。

消息发送的源码分析

以producer.send为入口

```
public void send(Destination destination, Message message, int deliveryMode, int priority, long
timeToLive, AsyncCallback onComplete) throws JMSException {
    checkClosed(); //检查session的状态，如果session以关闭则抛异常
    if (destination == null) {
        if (info.getDestination() == null) {
            throw new UnsupportedOperationException("A destination must be specified.");
        }
        throw new InvalidDestinationException("Don't understand null destinations");
    }
    ActiveMQDestination dest;
    if (destination.equals(info.getDestination())) {//检查destination的类型，如果符合要求，就转变为
ActiveMQDestination
        dest = (ActiveMQDestination)destination;
    } else if (info.getDestination() == null) {
        dest = ActiveMQDestination.transform(destination);
    } else {
        throw new UnsupportedOperationException("This producer can only send messages to: " +
this.info.getDestination().getPhysicalName());
    }
    if (dest == null) {
        throw new JMSException("No destination specified");
    }
    if (transformer != null) {
        Message transformedMessage = transformer.producerTransform(session, this, message);
        if (transformedMessage != null) {
            message = transformedMessage;
        }
    }
    if (producerWindow != null) {//如果发送窗口大小不为空，则判断发送窗口的大小决定是否阻塞
        try {
            producerWindow.waitForSpace();
        } catch (InterruptedException e) {
            throw new JMSException("Send aborted due to thread interrupt.");
        }
    }
   //发送消息到broker的topic
    this.session.send(this, dest, message, deliveryMode, priority, timeToLive, producerWindow,
sendTimeout, onComplete);
    stats.onMessage();
}
```

ActiveMQSession的send方法

```
protected void send(ActiveMQMessageProducer producer, ActiveMQDestination destination, Message
message, int deliveryMode, int priority, long timeToLive,
                    MemoryUsage producerWindow, int sendTimeout, AsyncCallback onComplete)
throws JMSException {

    checkClosed();
    if (destination.isTemporary() && connection.isDeleted(destination)) {
```

```
        throw new InvalidDestinationException("Cannot publish to a deleted Destination: " +
destination);
    }
    synchronized (sendMutex) { //互斥锁，如果一个session的多个producer发送消息到这里，会保证消息发送
的有序性
        // tell the Broker we are about to start a new transaction
        doStartTransaction();//告诉broker开始一个新事务，只有事务型会话中才会开启
        TransactionId txid = transactionContext.getTransactionId();//从事务上下文中获取事务id
        long sequenceNumber = producer.getMessageSequence();

        //Set the "JMS" header fields on the original message, see 1.1 spec section 3.4.11
        message.setJMSDeliveryMode(deliveryMode); //在JMS协议头中设置是否持久化标识
        long expiration = 0L;//计算消息过期时间
        if (!producer.getDisableMessageTimestamp()) {
            long timeStamp = System.currentTimeMillis();
            message.setJMSTimestamp(timeStamp);
            if (timeToLive > 0) {
                expiration = timeToLive + timeStamp;
             }
        }
        message.setJMSExpiration(expiration);//设置消息过期时间
        message.setJMSPriority(priority);//设置消息的优先级
        message.setJMSRedelivered(false);//设置消息为非重发

        // transform to our own message format here
//将不通的消息格式统一转化为ActiveMQMessage
        ActiveMQMessage msg = ActiveMQMessageTransformation.transformMessage(message,
connection);
        msg.setDestination(destination);//设置目的地
//生成并设置消息id
        msg.setMessageId(new MessageId(producer.getProducerInfo().getProducerId(),
sequenceNumber));

        // Set the message id.
        if (msg != message) {//如果消息是经过转化的，则更新原来的消息id和目的地
            message.setJMSMessageID(msg.getMessageId().toString());
            // Make sure the JMS destination is set on the foreign messages too.
            message.setJMSDestination(destination);
        }
        //clear the brokerPath in case we are re-sending this message
        msg.setBrokerPath(null);

        msg.setTransactionId(txid);
        if (connection.isCopyMessageOnSend()) {
            msg = (ActiveMQMessage)msg.copy();
        }
        msg.setConnection(connection);
        msg.onSend();//把消息属性和消息体都设置为只读，防止被修改
        msg.setProducerId(msg.getMessageId().getProducerId());
        if (LOG.isTraceEnabled()) {
            LOG.trace(getSessionId() + " sending message: " + msg);
        }
```

```
//如果onComplete没有设置，且发送超时时间小于0，且消息不需要反馈，且连接器不是同步发送模式，且消息非持久化或者连接器是异步发送模式
//或者存在事务id的情况下，走异步发送，否则走同步发送
        if (onComplete==null && sendTimeout <= 0 && !msg.isResponseRequired() &&
!connection.isAlwaysSyncSend() && (!msg.isPersistent() || connection.isUseAsyncSend() || txid !=
null)) {
            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(); //异步发送的情况下，需要设置producerWindow的大小
                producerWindow.increaseUsage(size);
            }
        } else {
            if (sendTimeout > 0 && onComplete==null) {
                this.connection.syncSendPacket(msg,sendTimeout); //带超时时间的同步发送
            }else {
                this.connection.syncSendPacket(msg, onComplete); //带回调的同步发送
            }
        }
    }
}
```

ActiveMQConnection. doAsyncSendPacket

```
private void doAsyncSendPacket(Command command) throws JMSException {
    try {
        this.transport.oneway(command);
    } catch (IOException e) {
        throw JMSExceptionSupport.create(e);
    }
}
```

这个地方问题来了，this.transport是什么东西？在哪里实例化的？按照以前看源码的惯例来看，它肯定不是一个单纯的对象。

按照以往我们看源码的经验来看，一定是在创建连接的过程中初始化的。所以我们定位到代码

transport的实例化过程

从connection=connectionFactory.createConnection();这行代码作为入口，一直跟踪到ActiveMQConnectionFactory. createActiveMQConnection这个方法中。代码如下

```
protected ActiveMQConnection createActiveMQConnection(String userName, String password) throws
JMSException {
    if (brokerURL == null) {
        throw new ConfigurationException("brokerURL not set.");
    }
    ActiveMQConnection connection = null;
    try {
        Transport transport = createTransport();
        connection = createActiveMQConnection(transport, factoryStats);

        connection.setUserName(userName);
        connection.setPassword(password);
//省略后面的代码
}
```

createTransport

调用ActiveMQConnectionFactory.createTransport方法，去创建一个transport对象。

1. 构建一个URI
2. 根据URL去创建一个连接TransportFactory.connect

Ø 默认使用的是tcp的协议

```
protected Transport createTransport() throws JMSException {
    try {
        URI connectBrokerUL = brokerURL;
        String scheme = brokerURL.getScheme();
        if (scheme == null) {
            throw new IOException("Transport not scheme specified: [" + brokerURL + "]");
        }
        if (scheme.equals("auto")) {
            connectBrokerUL = new URI(brokerURL.toString().replace("auto", "tcp"));
        } else if (scheme.equals("auto+ssl")) {
            connectBrokerUL = new URI(brokerURL.toString().replace("auto+ssl", "ssl"));
        } else if (scheme.equals("auto+nio")) {
            connectBrokerUL = new URI(brokerURL.toString().replace("auto+nio", "nio"));
        } else if (scheme.equals("auto+nio+ssl")) {
            connectBrokerUL = new URI(brokerURL.toString().replace("auto+nio+ssl", "nio+ssl"));
        }
        return TransportFactory.connect(connectBrokerUL);
    } catch (Exception e) {
        throw JMSExceptionSupport.create("Could not create Transport. Reason: " + e, e);
    }
}
```

TransportFactory. findTransportFactory

1. 从TRANSPORT_FACTORYS这个Map集合中，根据scheme去获得一个TransportFactory指定的实例对象
2. 如果Map集合中不存在，则通过TRANSPORT_FACTORY_FINDER去找一个并且构建实例

Ø 这个地方又有点类似于我们之前所学过的SPI的思想吧？他会从META-INF/services/org/apache/activemq/transport/ 这个路径下，根据URI组装的scheme去找到匹配的class对象并且实例化，所以根据tcp为key去对应的路径下可以找到TcpTransportFactory

```
public static TransportFactory findTransportFactory(URI location) throws IOException {
    String scheme = location.getScheme();
    if (scheme == null) {
        throw new IOException("Transport not scheme specified: [" + location + "]");
    }
    TransportFactory tf = TRANSPORT_FACTORYS.get(scheme);
    if (tf == null) {
        // Try to load if from a META-INF property.
        try {
            tf = (TransportFactory)TRANSPORT_FACTORY_FINDER.newInstance(scheme);
            TRANSPORT_FACTORYS.put(scheme, tf);
        } catch (Throwable e) {
            throw IOExceptionSupport.create("Transport scheme NOT recognized: [" + scheme + "]",
e);
        }
    }
    return tf;
}
```

调用TransportFactory.doConnect去构建一个连接

```
public Transport doConnect(URI location) throws Exception {
    try {
        Map<String, String> options = new HashMap<String, String>
(URISupport.parseParameters(location));
        if( !options.containsKey("wireFormat.host") ) {
            options.put("wireFormat.host", location.getHost());
        }
        WireFormat wf = createWireFormat(options);
        Transport transport = createTransport(location, wf); //创建一个Transport，创建一个socket连
接 -> 终于找到真相了
        Transport rc = configure(transport, wf, options);//配置configure，这个里面是对Transport做
链路包装
        //remove auto
        IntrospectionSupport.extractProperties(options, "auto.");

        if (!options.isEmpty()) {
            throw new IllegalArgumentException("Invalid connect parameters: " + options);
        }
```

```
            return rc;
        } catch (URISyntaxException e) {
            throw IOExceptionSupport.create(e);
        }
    }
```

configure

```
public Transport configure(Transport transport, WireFormat wf, Map options) throws Exception {
//组装一个复合的transport，这里会包装两层，一个是IactivityMonitor.另一个是WireFormatNegotiator
transport = compositeConfigure(transport, wf, options);
    transport = new MutexTransport(transport); //再做一层包装,MutexTransport
    transport = new ResponseCorrelator(transport); //包装ResponseCorrelator

    return transport;
}
```

到目前为止，这个transport实际上就是一个调用链了，他的链结构为

ResponseCorrelator(MutexTransport(WireFormatNegotiator(IactivityMonitor(TcpTransport()))

每一层包装表示什么意思呢?

ResponseCorrelator 用于实现异步请求。

MutexTransport 实现写锁，表示同一时间只允许发送一个请求

WireFormatNegotiator 实现了客户端连接broker的时候先发送数据解析相关的协议信息，比如解析版本号，是否使用缓存等

InactivityMonitor 用于实现连接成功成功后的心跳检查机制，客户端每10s发送一次心跳信息。服务端每30s读取一次心跳信息。

同步发送和异步发送的区别

```
public Object request(Object command, int timeout) throws IOException {
    FutureResponse response = asyncRequest(command, null);
    return response.getResult(timeout); // 从future方法阻塞等待返回
}
```

在ResponseCorrelator的request方法中，需要通过response.getResult去获得broker的反馈，否则会阻塞

持久化消息和非持久化消息的存储原理

正常情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中的。能够存储的最大消息数据在${ActiveMQ_HOME}/conf/activemq.xml文件中的systemUsage节点

SystemUsage配置设置了一些系统内存和硬盘容量

```
<systemUsage>
       <systemUsage>
           <memoryUsage>
//该子标记设置整个ActiveMQ节点的“可用内存限制”。这个值不能超过ActiveMQ本身设置的最大内存大小。其中的percentOfJvmHeap属性表示百分比。占用70%的堆内存
               <memoryUsage percentOfJvmHeap="70" />
           </memoryUsage>
           <storeUsage>
//该标记设置整个ActiveMQ节点，用于存储“持久化消息”的“可用磁盘空间”。该子标记的limit属性必须要进行设置
               <storeUsage limit="100 gb"/>
           </storeUsage>
           <tempUsage>
//一旦ActiveMQ服务节点存储的消息达到了memoryUsage的限制，非持久化消息就会被转储到 temp store区域，虽然我们说过非持久化消息不进行持久化存储，但是ActiveMQ为了防止“数据洪峰”出现时非持久化消息大量堆积致使内存耗尽的情况出现，还是会将非持久化消息写入到磁盘的临时区域—temp store。这个子标记就是为了设置这个temp store区域的“可用磁盘空间限制”
               <tempUsage limit="50 gb"/>
           </tempUsage>
       </systemUsage>
</systemUsage>
```

Ø 从上面的配置我们需要get到一个结论，当非持久化消息堆积到一定程度的时候，也就是内存超过指定的设置阀值时，ActiveMQ会将内存中的非持久化消息写入到临时文件，以便腾出内存。但是它和持久化消息的区别是，重启之后，持久化消息会从文件中恢复，非持久化的临时文件会直接删除

消息的持久化策略分析

消息持久性对于可靠消息传递来说是一种比较好的方法，即时发送者和接受者不是同时在线或者消息中心在发送者发送消息后宕机了，在消息中心重启后仍然可以将消息发送出去。消息持久性的原理很简单，就是在发送消息出去后，消息中心首先将消息存储在本地文件、内存或者远程数据库，然后把消息发送给接受者，发送成功后再把消息从存储中删除，失败则继续尝试。接下来我们来了解一下消息在broker上的持久化存储实现方式

持久化存储支持类型

ActiveMQ支持多种不同的持久化方式，主要有以下几种，不过，无论使用哪种持久化方式，消息的存储逻辑都是一致的。

Ø KahaDB存储（默认存储方式）

Ø JDBC存储

Ø Memory存储

Ø LevelDB存储

Ø JDBC With ActiveMQ Journal

KahaDB存储

KahaDB是目前默认的存储方式,可用于任何场景,提高了性能和恢复能力。消息存储使用一个事务日志和仅仅用一个索引文件来存储它所有的地址。

KahaDB是一个专门针对消息持久化的解决方案,它对典型的消息使用模式进行了优化。在Kaha中,数据被追加到data logs中。当不再需要log文件中的数据的时候,log文件会被丢弃。

KahaDB的配置方式

```
<persistenceAdapter>
          <kahaDB directory="${activemq.data}/kahadb"/>
      </persistenceAdapter>
```

KahaDB的存储原理

在data/kahadb这个目录下，会生成四个文件

Ø db.data 它是消息的索引文件，本质上是B-Tree（B树），使用B-Tree作为索引指向db-*.log里面存储的消息

Ø db.redo 用来进行消息恢复

Ø db-*.log 存储消息内容。新的数据以APPEND的方式追加到日志文件末尾。属于顺序写入，因此消息存储是比较快的。默认是32M，达到阀值会自动递增

Ø lock文件 锁，表示当前获得kahadb读写权限的broker

JDBC存储

使用JDBC持久化方式，数据库会创建3个表：activemq_msgs，activemq_acks和activemq_lock。

ACTIVEMQ_MSGS 消息表，queue和topic都存在这个表中

ACTIVEMQ_ACKS 存储持久订阅的信息和最后一个持久订阅接收的消息ID

ACTIVEMQ_LOCKS 锁表，用来确保某一时刻，只能有一个ActiveMQ broker实例来访问数据库

JDBC存储实践

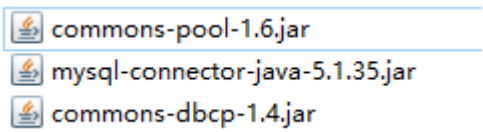
```
<persistenceAdapter>
    <jdbcPersistenceAdapter dataSource="# MySQL-DS " createTablesOnStartup="true" />
</persistenceAdapter>
```

dataSource指定持久化数据库的bean，createTablesOnStartup是否在启动的时候创建数据表，默认值是true，这样每次启动都会去创建数据表了，一般是第一次启动的时候设置为true，之后改成false

Mysql持久化Bean配置

```
<bean id="Mysql-DS" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://192.168.11.156:3306/activemq?
relaxAutoCommit=true"/>
      <property name="username" value="root"/>
      <property name="password" value="root"/>
   </bean>
```

添加Jar包依赖



LevelDB存储

LevelDB持久化性能高于KahaDB，虽然目前默认的持久化方式仍然是KahaDB。并且，在ActiveMQ 5.9版本提供了基于LevelDB和Zookeeper的数据复制方式，用于Master-slave方式的首选数据复制方案。

不过，据ActiveMQ官网对LevelDB的表述：LevelDB官方建议使用以及不再支持，推荐使用的是KahaDB

```
<persistenceAdapter>
      <levelDBdirectory="activemq-data"/>
</persistenceAdapter>
```

Memory 消息存储

基于内存的消息存储，内存消息存储主要是存储所有的持久化的消息在内存中。persistent=”false”,表示不设置持久化存储，直接存储到内存中

```
<beans>
<broker brokerName="test-broker" persistent="false"
xmlns="http://activemq.apache.org/schema/core">
<transportConnectors>
<transportConnector uri="tcp://localhost:61635"/>
</transportConnectors> </broker>
</beans>
```

JDBC Message store with ActiveMQ Journal

这种方式克服了JDBC Store的不足，JDBC每次消息过来，都需要去写库和读库。

ActiveMQ Journal，使用高速缓存写入技术，大大提高了性能。

当消费者的消费速度能够及时跟上生产者消息的生产速度时，journal文件能够大大减少需要写入到DB中的消息。举个例子，生产者生产了1000条消息，这1000条消息会保存到journal文件，如果消费者的消费速度很快的情况下，在journal文件还没有同步到DB之前，消费者已经消费了90%的以上的消息，那么这个时候只需要同步剩余的10%的消息到DB。

如果消费者的消费速度很慢，这个时候journal文件可以使消息以批量方式写到DB。

Ø 将原来的标签注释掉

Ø 添加如下标签

```
<persistenceFactory>
          <journalPersistenceAdapterFactory  dataSource="#Mysql-DS" dataDirectory="activemq-
data"/>
</persistenceFactory>
```

Ø 在服务端循环发送消息。可以看到数据是延迟同步到数据库的

消费端消费消息的原理

我们通过上一节课的讲解，知道有两种方法可以接收消息，一种是使用同步阻塞的MessageConsumer#receive方法。另一种是使用消息监听器MessageListener。这里需要注意的是，在同一个session下，这两者不能同时工作，也就是说不能针对不同消息采用不同的接收方式。否则会抛出异常。

至于为什么这么做，最大的原因还是在事务性会话中，两种消费模式的事务不好管控

消费端消费消息源码分析

ActiveMQMessageConsumer.receive

消费端同步接收消息的源码入口

```
public Message receive() throws JMSException {
    checkClosed();
    checkMessageListener();  //检查receive和MessageListener是否同时配置在当前的会话中
    sendPullCommand(0); //如果PrefetchSizeSize为0并且unconsumerMessage为空，则发起pull命令
    MessageDispatch md = dequeue(-1); //从unconsumerMessage出队列获取消息
    if (md == null) {
        return null;
    }
    beforeMessageIsConsumed(md);
    afterMessageIsConsumed(md, false); //发送ack给到broker
    return createActiveMQMessage(md);//获取消息并返回
}
```

sendPullCommand

发送pull命令从broker上获取消息，前提是prefetchSize=0并且unconsumedMessages为空。
unconsumedMessage表示未消费的消息，这里面预读取的消息大小为prefetchSize的值

```
protected void sendPullCommand(long timeout) throws JMSException {
    clearDeliveredList();
    if (info.getCurrentPrefetchSize() == 0 && unconsumedMessages.isEmpty()) {
        MessagePull messagePull = new MessagePull();
        messagePull.configure(info);
        messagePull.setTimeout(timeout);
        session.asyncSendPacket(messagePull); //向服务端异步发送messagePull指令
    }
}
```

clearDeliveredList

在上面的sendPullCommand方法中，会先调用clearDeliveredList方法，主要用来清理已经分发的消息链表deliveredMessages

deliveredMessages，存储分发给消费者但还为应答的消息链表

Ø 如果session是事务的，则会遍历deliveredMessage中的消息放入到previouslyDeliveredMessage中来做重发

Ø 如果session是非事务的，根据ACK的模式来选择不同的应答操作

```
private void clearDeliveredList() {
    if (clearDeliveredList) {
        synchronized (deliveredMessages) {
            if (clearDeliveredList) {
                if (!deliveredMessages.isEmpty()) {
                    if (session.isTransacted()) {
```

```
                        if (previouslyDeliveredMessages == null) {
                            previouslyDeliveredMessages = new PreviouslyDeliveredMap<MessageId,
Boolean>(session.getTransactionContext().getTransactionId());
                        }
                        for (MessageDispatch delivered : deliveredMessages) {

 previouslyDeliveredMessages.put(delivered.getMessage().getMessageId(), false);
                        }
                        LOG.debug("{} tracking existing transacted {} delivered list ({}) on
transport interrupt",
                                  getConsumerId(), previouslyDeliveredMessages.transactionId,
deliveredMessages.size());
                    } else {
                        if (session.isClientAcknowledge()) {
                            LOG.debug("{} rolling back delivered list ({}) on transport
interrupt", getConsumerId(), deliveredMessages.size());
                            // allow redelivery
                            if (!this.info.isBrowser()) {
                                for (MessageDispatch md: deliveredMessages) {
                                    this.session.connection.rollbackDuplicate(this,
md.getMessage());
                                }
                            }
                        }
                        LOG.debug("{} clearing delivered list ({}) on transport interrupt",
getConsumerId(), deliveredMessages.size());
                        deliveredMessages.clear();
                        pendingAck = null;
                    }
                }
                clearDeliveredList = false;
            }
        }
    }
}
```

dequeue

从unconsumedMessage中取出一个消息，在创建一个消费者时，就会未这个消费者创建一个未消费的消息通道，这个通道分为两种，一种是简单优先级队列分发通道SimplePriorityMessageDispatchChannel；另一种是先进先出的分发通道FifoMessageDispatchChannel.

至于为什么要存在这样一个消息分发通道，大家可以想象一下，如果消费者每次去消费完一个消息以后再去broker拿一个消息，效率是比较低的。所以通过这样的设计可以允许session能够一次性将多条消息分发给一个消费者。默认情况下对于queue来说，prefetchSize的值是1000

beforeMessageIsConsumed

这里面主要是做消息消费之前的一些准备工作，如果ACK类型不是DUPS_OK_ACKNOWLEDGE或者队列模式（简单来说就是除了Topic和DupAck这两种情况），所有的消息先放到deliveredMessages链表的开头。并且如果当前是事务类型的会话，则判断transactedIndividualAck，如果为true，表示单条消息直接返回ack。

否则，调用ackLater，批量应答, client端在消费消息后暂且不发送ACK，而是把它缓存下来(pendingACK)，等到这些消息的条数达到一定阀值时，只需要通过一个ACK指令把它们全部确认；这比对每条消息都逐个确认，在性能上要提高很多

```
private void beforeMessageIsConsumed(MessageDispatch md) throws JMSException {
    md.setDeliverySequenceId(session.getNextDeliveryId());
    lastDeliveredSequenceId = md.getMessage().getMessageId().getBrokerSequenceId();
    if (!isAutoAcknowledgeBatch()) {
        synchronized(deliveredMessages) {
            deliveredMessages.addFirst(md);
        }
        if (session.getTransacted()) {
            if (transactedIndividualAck) {
                immediateIndividualTransactedAck(md);
            } else {
                ackLater(md, MessageAck.DELIVERED_ACK_TYPE);
            }
        }
    }
}
```

afterMessageIsConsumed

这个方法的主要作用是执行应答操作，这里面做以下几个操作

Ø 如果消息过期，则返回消息过期的ack

Ø 如果是事务类型的会话，则不做任何处理

Ø 如果是AUTOACK或者（DUPS_OK_ACK且是队列），并且是优化ack操作，则走批量确认ack

Ø 如果是DUPS_OK_ACK，则走ackLater逻辑

Ø 如果是CLIENT_ACK，则执行ackLater

```
private void afterMessageIsConsumed(MessageDispatch md, boolean messageExpired) throws
JMSException {
    if (unconsumedMessages.isClosed()) {
        return;
    }
    if (messageExpired) {
        acknowledge(md, MessageAck.EXPIRED_ACK_TYPE);
        stats.getExpiredMessageCount().increment();
    } else {
        stats.onMessage();
```

```
        if (session.getTransacted()) {
            // Do nothing.
        } else if (isAutoAcknowledgeEach()) {
            if (deliveryingAcknowledgements.compareAndSet(false, true)) {
                synchronized (deliveredMessages) {
                    if (!deliveredMessages.isEmpty()) {
                        if (optimizeAcknowledge) {
                            ackCounter++;

                            // AMQ-3956 evaluate both expired and normal msgs as
                            // otherwise consumer may get stalled
                            if (ackCounter + deliveredCounter >= (info.getPrefetchSize() * .65)
|| (optimizeAcknowledgeTimeOut > 0 && System.currentTimeMillis() >= (optimizeAckTimestamp +
optimizeAcknowledgeTimeOut))) {
                                MessageAck ack =
makeAckForAllDeliveredMessages(MessageAck.STANDARD_ACK_TYPE);
                                if (ack != null) {
                                    deliveredMessages.clear();
                                    ackCounter = 0;
                                    session.sendAck(ack);
                                    optimizeAckTimestamp = System.currentTimeMillis();
                                }
                                // AMQ-3956 - as further optimization send
                                // ack for expired msgs when there are any.
                                // This resets the deliveredCounter to 0 so that
                                // we won't sent standard acks with every msg just
                                // because the deliveredCounter just below
                                // 0.5 * prefetch as used in ackLater()
                                if (pendingAck != null && deliveredCounter > 0) {
                                    session.sendAck(pendingAck);
                                    pendingAck = null;
                                    deliveredCounter = 0;
                                }
                            }
                        } else {
                            MessageAck ack =
makeAckForAllDeliveredMessages(MessageAck.STANDARD_ACK_TYPE);
                            if (ack!=null) {
                                deliveredMessages.clear();
                                session.sendAck(ack);
                            }
                        }
                    }
                }
                deliveryingAcknowledgements.set(false);
            }
        } else if (isAutoAcknowledgeBatch()) {
            ackLater(md, MessageAck.STANDARD_ACK_TYPE);
        } else if (session.isClientAcknowledge()||session.isIndividualAcknowledge()) {
            boolean messageUnackedByConsumer = false;
            synchronized (deliveredMessages) {
                messageUnackedByConsumer = deliveredMessages.contains(md);
            }
```

```
            if (messageUnackedByConsumer) {
                ackLater(md, MessageAck.DELIVERED_ACK_TYPE);
            }
        }
        else {
            throw new IllegalStateException("Invalid session state.");
        }
    }
}
```