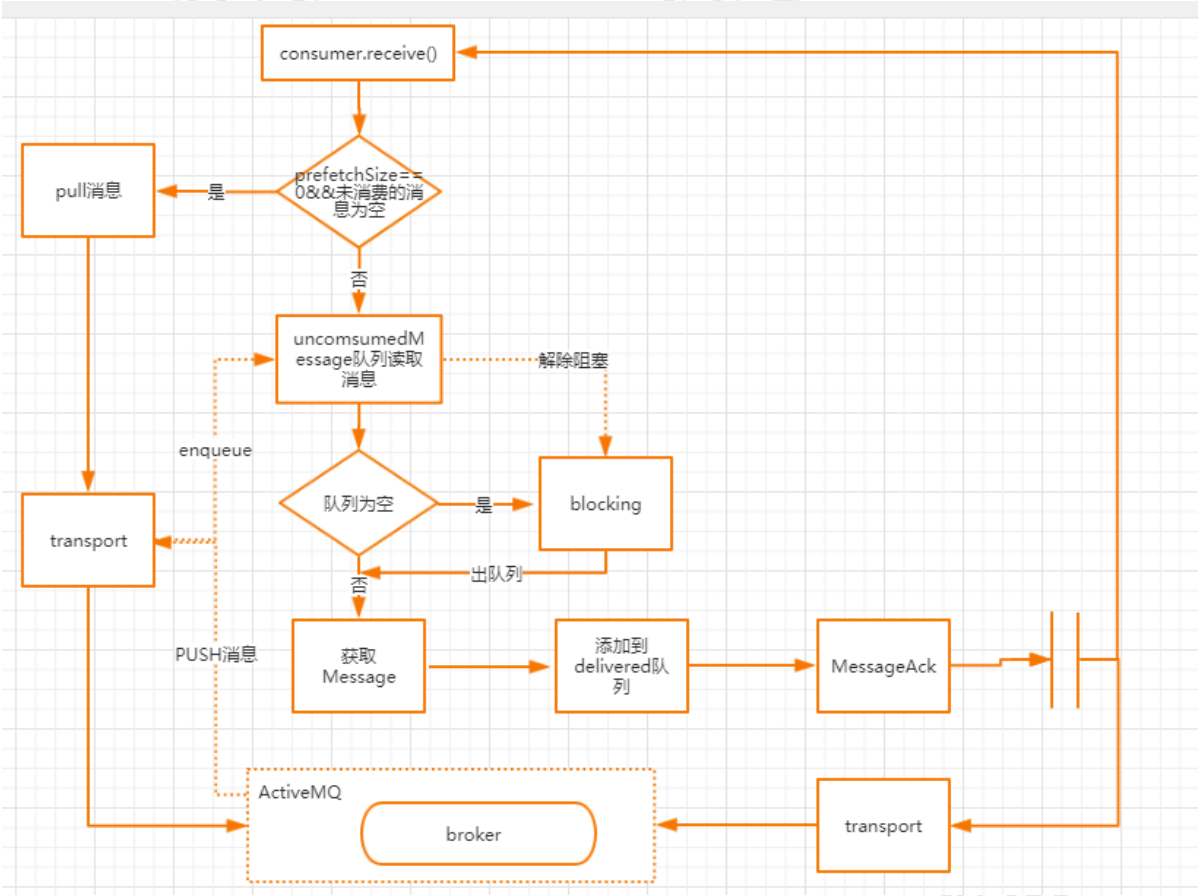
咕泡学院 VIP 课：分布式消息通信 ActiveMQ 原理分析

课程目标

1. unconsumedMessage 源码分析
2. 消费端的 PrefetchSize
3. 消息的确认过程
4. 消息重发机制
5. ActiveMQ 多节点高性能方案

消息消费流程图



unconsumedMessages 数据的获取过程

那我们来看看 ActiveMQConnectionFactory. createConnection 里面做了什么事情。

1. 动态创建一个传输协议
2. 创建一个连接

3. 通过 transport.start()

```
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);

       configureConnection(connection);

       transport.start();

       if (clientID != null) {

           connection.setDefaultClientID(clientID);

      }

       return connection;

  }
```

transport.start()

我们前面在分析消息发送的时候，已经知道 transport 是一个链式的调用，是一个多层包装的对象。

ResponseCorrelator(MutexTransport(WireFormatNegotiator(InactivityMonitor(TcpTransport()))))

最终调用 TcpTransport.start()方法，然而这个类中并没有 start，而是在父类 ServiceSupport.start()中。

```
public void start() throws Exception {

   if (started.compareAndSet(false, true)) {

       boolean success = false;

       stopped.set(false);

       try {

           preStart();

           doStart();

           success = true;

      } finally {

           started.set(success);

      }

       for(ServiceListener l:this.serviceListeners) {

           l.started(this);

      }
```

```
    }

}
```

这块代码看起来就比较熟悉了，我们之前看过的中间件的源码，通信层都是独立来实现及解耦的。而 ActiveMQ 也是一样，提供了 Transport 接口和 TransportSupport 类。这个接口的主要作用是为了让客户端有消息被异步发送、同步发送和被消费的能力。接下来沿着 doStart()往下看，又调用 TcpTransport.doStart() ,接着通过 super.doStart(),调用 TransportThreadSupport.doStart(). 创建了一个线程，传入的是 this，调用子类的 run 方法，也就是 TcpTransport.run().

TcpTransport.run

run 方法主要是从 socket 中读取数据包，只要 TcpTransport 没有停止，它就会不断去调用 doRun。

```
public void run() {
    LOG.trace("TCP consumer thread for " + this + " starting");
    this.runnerThread=Thread.currentThread();
    try {
        while (!isStopped()) {
            doRun();
        }
    } catch (IOException e) {
        stoppedLatch.get().countDown();
```

```
        onException(e);
    } catch (Throwable e){
        stoppedLatch.get().countDown();
        IOException ioe=new IOException("Unexpected error occurred: "
+ e);
        ioe.initCause(e);
        onException(ioe);
    }finally {
        stoppedLatch.get().countDown();
    }
}
```

TcpTransport.doRun

doRun 中，通过 readCommand 去读取数据

```
protected void doRun() throws IOException {
    try {
        Object command = readCommand();
         doConsume(command);
    } catch (SocketTimeoutException e) {
    } catch (InterruptedIOException e) {
    }
}
```

TcpTransport.readCommand

这里面，通过 wireFormat 对数据进行格式化，可以认为这是一个反序列化过程。wireFormat 默认实现是 OpenWireFormat，activeMQ 自定义的跨语言的 wire 协议

```
protected Object readCommand() throws IOException {

   return wireFormat.unmarshal(dataIn);

}
```

分析到这，我们差不多明白了传输层的主要工作是获得数据并且把数据转换为对象，再把对象对象传给 ActiveMQConnection

TransportSupport.doConsume

TransportSupport 类中最重要的方法是 doConsume，它的作用就是用来“消费消息”

```
public void doConsume(Object command) {

   if (command != null) {

       if (transportListener != null) {

           transportListener.onCommand(command);

       } else {

           LOG.error("No transportListener available to process
inbound command: " + command);

       }

   }
```

```
}
```

TransportSupport 类中唯一的成员变量是 TransportListener transportListener;，这也意味着一个 Transport 支持类绑定一个传送监听器类，传送监听器接口 TransportListener 最重要的方法就是 void onCommand(Object command);，它用来处理命令，

这个 transportListener 是在哪里赋值的呢？再回到 ActiveMQConnection 的构造方法中。->246 行

传递了 ActiveMQConnection 自己本身，(ActiveMQConnection 是 TransportListener 接口的实现类之一)

于是，消息就这样从传送层到达了我们的连接层上。

```
protected ActiveMQConnection(final Transport transport, IdGenerator
clientIdGenerator, IdGenerator connectionIdGenerator, JMSStatsImpl
factoryStats) throws Exception {



   this.transport = transport; 绑定传输对象

   this.clientIdGenerator = clientIdGenerator;

   this.factoryStats = factoryStats;



   // Configure a single threaded executor who's core thread can
timeout if

   // idle

   executor = new ThreadPoolExecutor(1, 1, 5, TimeUnit.SECONDS, new
LinkedBlockingQueue<Runnable>(), new ThreadFactory() {
```

```
        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r, "ActiveMQ Connection
Executor: " + transport);
            //Don't make these daemon threads - see
https://issues.apache.org/jira/browse/AMQ-796
            //thread.setDaemon(true);
            return thread;
        }
    });
    // asyncConnectionThread.allowCoreThreadTimeOut(true);
    String uniqueId = connectionIdGenerator.generateId();
    this.info = new ConnectionInfo(new ConnectionId(uniqueId));
    this.info.setManageable(true);
    this.info.setFaultTolerant(transport.isFaultTolerant());
    this.connectionSessionId = new SessionId(info.getConnectionId(),
-1);

    this.transport.setTransportListener(this); //当 transport 绑定为自
己

    this.stats = new JMSConnectionStatsImpl(sessions, this instanceof
XAConnection);
    this.factoryStats.addConnection(this);
```

```
    this.timeCreated = System.currentTimeMillis();

    this.connectionAudit.setCheckForDuplicates(transport.isFaultTolerant());

}
```

从构造函数可以看出，创建 ActiveMQConnection 对象时，除了和 Transport 相互绑定，还对线程池执行器 executor 进行了初始化。下面我们看看该类的核心方法

onCommand

Ø 这里面会针对不同的消息做分发，比如传入的 command 是 MessageDispatch，那么这个 command 的 visit 方法就会调用 processMessageDispatch 方法

```
public void onCommand(final Object o) {

    final Command command = (Command)o;

    if (!closed.get() && command != null) {

        try {

            command.visit(new CommandVisitorAdapter() {

                @Override

                public Response processMessageDispatch(MessageDispatch md) throws Exception {

// 等待 Transport 中断处理完成

                    waitForTransportInterruptionProcessingToComplete();

// 这里通过消费者 ID 来获取消费者对象
```

```
//（ActiveMQMessageConsumer 实现了 ActiveMQDispatcher 接口），所以
MessageDispatch 包含了消息应该被分配到那个消费者的映射信息

//在创建 MessageConsumer 的时候，调用 ActiveMQMessageConsumer 的第 282
行，调用 ActiveMQSession 的 1798 行将当前的消费者绑定到 dispatchers 中
```

所以这里拿到的是 ActiveMQSession

```
                    ActiveMQDispatcher dispatcher =
dispatchers.get(md.getConsumerId());

                    if (dispatcher != null) {

                        // Copy in case a embedded broker is
dispatching via

                        // vm://

                        // md.getMessage() == null to signal end of
queue

                        // browse.

                        Message msg = md.getMessage();

                        if (msg != null) {

                            msg = msg.copy();

                            msg.setReadOnlyBody(true);

                            msg.setReadOnlyProperties(true);

                            msg.setRedeliveryCounter(md.getRedeliveryC
ounter());

                            msg.setConnection(ActiveMQConnection.this)
;

                            msg.setMemoryUsage(null);

                            md.setMessage(msg);

                        }
```

```
                        dispatcher.dispatch(md); // 调用会话
ActiveMQSession 自己的 dispatch 方法来处理这条消息

                    } else {

                        LOG.debug("{} no dispatcher for {} in {}",
this, md, dispatchers);

                    }

                     return null;

                }

                //如果传入的是 ProducerAck，则调用的是下面这个方法，这里我
们仅仅关注 MessageDispatch 就行了

                @Override

                public Response processProducerAck(ProducerAck pa)
throws Exception {

                    if (pa != null && pa.getProducerId() != null) {

                        ActiveMQMessageProducer producer =
producers.get(pa.getProducerId());

                        if (producer != null) {

                            producer.onProducerAck(pa);

                        }

                    }

                     return null;

                }

            //…后续的方法就不分析了
```

在现在这个场景中，我们只关注 processMessageDispatch 方法，在这个方法中，只是简单的去调用 ActiveMQSession 的 dispatch 方法来处理消息,

Ø tips: command.visit，这里使用了适配器模式，如果 command 是一个 MessageDispatch，那么它就会调用 processMessageDispatch 方法，其他方法他不会关心，代码如下：MessageDispatch.visit

```
@Override
public Response visit(CommandVisitor visitor) throws Exception {
   return visitor.processMessageDispatch(this);
}
```

ActiveMQSession.dispatch(md)

executor 这个对象其实是一个成员对象 ActiveMQSessionExecutor，专门负责来处理消息分发

```
@Override
public void dispatch(MessageDispatch messageDispatch) {
    try {
        executor.execute(messageDispatch);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        connection.onClientInternalException(e);
    }
}
```

ActiveMQSessionExecutor.execute

Ø 这个方法的核心功能就是处理消息的分发。

```
void execute(MessageDispatch message) throws InterruptedException {



   if (!startedOrWarnedThatNotStarted) {

       ActiveMQConnection connection = session.connection;

       long aboutUnstartedConnectionTimeout =
connection.getWarnAboutUnstartedConnectionTimeout();

       if (connection.isStarted() || aboutUnstartedConnectionTimeout
< 0L) {

           startedOrWarnedThatNotStarted = true;

      } else {

           long elapsedTime = System.currentTimeMillis() -
connection.getTimeCreated();



           // lets only warn when a significant amount of time has
passed

           // just in case its normal operation

           if (elapsedTime > aboutUnstartedConnectionTimeout) {

               LOG.warn("Received a message on a connection which is
not yet started. Have you forgotten to call Connection.start()?
Connection: " + connection

                       + " Received: " + message);

               startedOrWarnedThatNotStarted = true;
```

```
            }

        }

    }
//如果会话不是异步分发并且没有使用 sessionpool 分发，则调用 dispatch 发送消息（这里不展开）

    if (!session.isSessionAsyncDispatch()
&& !dispatchedBySessionPool) {

        dispatch(message);

    } else {
//将消息直接放到队列里

        messageQueue.enqueue(message);

        wakeup();

    }

}
```

默认是采用异步消息分发。所以，直接调用 messageQueue.enqueue，把消息放到队列中，并且调用 wakeup 方法

异步分发的流程

```
public void wakeup() {

    if (!dispatchedBySessionPool) { //进一步验证

        if (session.isSessionAsyncDispatch()) { //判断 session 是否为异步分发

            try {

                TaskRunner taskRunner = this.taskRunner;
```

```
                if (taskRunner == null) {
                    synchronized (this) {
                        if (this.taskRunner == null) {
                            if (!isRunning()) {
                                // stop has been called
                                return;
                            }
//通过 TaskRunnerFactory 创建了一个任务运行类 taskRunner，这里把自己作为一个 task 传入到 createTaskRunner 中，说明当前
//的类一定是实现了 Task 接口的. 简单来说，就是通过线程池去执行一个任务，完成异步调度，简单吧
                            this.taskRunner = session.connection.getSessionTaskRunner().createTaskRunner(this,
                                    "ActiveMQ Session: " + session.getSessionId());
                        }
                        taskRunner = this.taskRunner;
                    }
                }
                taskRunner.wakeup();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        } else {
```

```
            while (iterate()) { //同步分发

            }

        }

    }

}
```

所以，对于异步分发的方式，会调用 ActiveMQSessionExecutor 中的 iterate 方法，我们来看看这个方法的代码

iterate

这个方法里面做两个事

Ø 把消费者监听的所有消息转存到待消费队列中

Ø 如果 messageQueue 还存在遗留消息，同样把消息分发出去

```
public boolean iterate() {

   // Deliver any messages queued on the consumer to their
listeners.

   for (ActiveMQMessageConsumer consumer : this.session.consumers) {

       if (consumer.iterate()) {

           return true;

       }

   }

   // No messages left queued on the listeners.. so now dispatch
messages

   // queued on the session
```

```
    MessageDispatch message = messageQueue.dequeueNoWait();
    if (message == null) {
        return false;
    } else {
        dispatch(message);
        return !messageQueue.isEmpty();
    }
}
```

ActiveMQMessageConsumer.iterate

```
public boolean iterate() {
    MessageListener listener = this.messageListener.get();
    if (listener != null) {
        MessageDispatch md = unconsumedMessages.dequeueNoWait();
        if (md != null) {
            dispatch(md);
            return true;
        }
    }
    return false;
}
```

同步分发的流程

同步分发的流程，直接调用 ActiveMQSessionExcutor 中的 dispatch 方法，代码如下

```
void dispatch(MessageDispatch message) {
   // TODO - we should use a Map for this indexed by consumerId
   for (ActiveMQMessageConsumer consumer : this.session.consumers) {
       ConsumerId consumerId = message.getConsumerId();
       if (consumerId.equals(consumer.getConsumerId())) {
           consumer.dispatch(message);
           break;
       }
   }
}
```

ActiveMQMessageConsumer.dispatch

调用 ActiveMQMessageConsumer.dispatch 方法，把消息转存到 unconsumedMessages 消息队列中。

```
public void dispatch(MessageDispatch md) {
   MessageListener listener = this.messageListener.get();
   try {
       clearMessagesInProgress();
```

```
        clearDeliveredList();

        synchronized (unconsumedMessages.getMutex()) {

            if (!unconsumedMessages.isClosed()) {

                if (this.info.isBrowser()
|| !session.connection.isDuplicate(this, md.getMessage())) {

                    if (listener != null &&
unconsumedMessages.isRunning()) {

                        if (redeliveryExceeded(md)) {

                            posionAck(md, "listener dispatch[" +
md.getRedeliveryCounter() + "] to " + getConsumerId() + " exceeds
redelivery policy limit:" + redeliveryPolicy);

                            return;

                        }

                        ActiveMQMessage message =
createActiveMQMessage(md);

                        beforeMessageIsConsumed(md);

                        try {

                            boolean expired =
isConsumerExpiryCheckEnabled() && message.isExpired();

                            if (!expired) {

                                listener.onMessage(message);

                            }

                            afterMessageIsConsumed(md, expired);

                        } catch (RuntimeException e) {

                            LOG.error("{} Exception while processing
message: {}", getConsumerId(), md.getMessage().getMessageId(), e);
```

```
                            md.setRollbackCause(e);

                            if (isAutoAcknowledgeBatch() ||
isAutoAcknowledgeEach() || session.isIndividualAcknowledge()) {

                                // schedual redelivery and possible dlq
processing

                                rollback();

                            } else {

                                // Transacted or Client ack: Deliver
the next message.

                                afterMessageIsConsumed(md, false);

                            }

                        }

                    } else {

                        if (!unconsumedMessages.isRunning()) {

                            // delayed redelivery, ensure it can be re
delivered

                            session.connection.rollbackDuplicate(this,
md.getMessage());

                        }


                        if (md.getMessage() == null) {

                            // End of browse or pull request timeout.

                            unconsumedMessages.enqueue(md);

                        } else {
```

```
                        if (!consumeExpiredMessage(md)) {
                            unconsumedMessages.enqueue(md);
                            if (availableListener != null) {
                                availableListener.onMessageAvailabl
e(this);
                            }
                        } else {
                            beforeMessageIsConsumed(md);
                            afterMessageIsConsumed(md, true);

                            // Pull consumer needs to check if pull
timed out and send
                            // a new pull command if not.
                            if (info.getCurrentPrefetchSize() == 0)
{
                                unconsumedMessages.enqueue(null);
                            }
                        }
                    }
                }
            } else {
                // deal with duplicate delivery
                ConsumerId consumerWithPendingTransaction;
                if (redeliveryExpectedInCurrentTransaction(md,
```

```
true)) {

                        LOG.debug("{} tracking transacted redelivery
{}", getConsumerId(), md.getMessage());

                        if (transactedIndividualAck) {

                            immediateIndividualTransactedAck(md);

                        } else {

                            session.sendAck(new MessageAck(md,
MessageAck.DELIVERED_ACK_TYPE, 1));

                        }

                    } else if ((consumerWithPendingTransaction =
redeliveryPendingInCompetingTransaction(md)) != null) {

                        LOG.warn("{} delivering duplicate {}, pending
transaction completion on {} will rollback", getConsumerId(),
md.getMessage(), consumerWithPendingTransaction);

                        session.getConnection().rollbackDuplicate(this
, md.getMessage());

                        dispatch(md);

                    } else {

                        LOG.warn("{} suppressing duplicate delivery on
connection, poison acking: {}", getConsumerId(), md);

                        posionAck(md, "Suppressing duplicate delivery
on connection, consumer " + getConsumerId());

                    }

                }

            }

        }
```

```
        if (++dispatchedCount % 1000 == 0) {
            dispatchedCount = 0;
            Thread.yield();
        }
    } catch (Exception e) {
        session.connection.onClientInternalException(e);
    }
}
```

Ø 到这里为止，消息如何接受以及他的处理方式的流程，我们已经搞清楚了，希望对大家理解 activeMQ 的核心机制有一定的帮助

消费端的 PrefetchSize

还记得我们在分析消费端的源码的时候，所讲到的 prefetchsize 吗？这个 prefetchsize 是做什么的？我们接下来去研究一下

代码演示

参考课堂的案例

原理剖析

activemq 的 consumer 端也有窗口机制，通过 prefetchSize 就可以设置窗口大小。不同的类型的队列，prefetchSize 的默认值也是不一样的

Ø 持久化队列和非持久化队列的默认值为 1000

Ø 持久化 topic 默认值为 100

Ø 非持久化队列的默认值为 Short.MAX_VALUE-1

通过上面的例子，我们基本上应该知道 prefetchSize 的作用了，消费端会根据 prefetchSize 的大小批量获取数据，比如默认值是 1000，那么消费端会预先加载 1000 条数据到本地的内存中。

prefetchSize 的设置方法

在 createQueue 中添加 consumer.prefetchSize，就可以看到效果

```
Destination
destination=session.createQueue("myQueue?consumer.prefetchSize=10");
```

既然有批量加载，那么一定有批量确认，这样才算是彻底的优化

optimizeAcknowledge

ActiveMQ 提供了 optimizeAcknowledge 来优化确认，它表示是否开启“优化 ACK”，只有在为 true 的情况下，prefetchSize 以及 optimizeAcknowledgeTimeout 参数才会有意义

优化确认一方面可以减轻 client 负担（不需要频繁的确认消息）、减少通信开销，另一方面由于延迟了确认（默认 ack 了 0.65*prefetchSize 个消息才确认），broker 再次发送消息时又可以批量发送

如果只是开启了 prefetchSize，每条消息都去确认的话，broker 在收到确认后也只是发送一条消息，并不是批量发布，当然也可以通过设置 DUPS_OK_ACK 来手动延迟确认， 我们需要在 brokerUrl 指定 optimizeACK 选项

ConnectionFactory connectionFactory= new ActiveMQConnectionFactory

("tcp://192.168.11.153:61616?jms.optimizeAcknowledge=true&jms.optimiz

eAcknowledgeTimeOut=10000");

Ø 注意，如果 optimizeAcknowledge 为 true，那么 prefetchSize 必须大于 0.

当 prefetchSize=0 的时候，表示 consumer 通过 PULL 方式从 broker 获取消息

总结

到目前为止，我们知道了 optimizeAcknowledge 和 prefetchSize 的作用，两者协同工作，通过批量获取消息、并延迟批量确认，来达到一个高效的消息消费模型。它比仅减少了客户端在获取消息时的阻塞次数，还能减少每次获取消息时的网络通信开销

Ø 需要注意的是，如果消费端的消费速度比较高，通过这两者组合是能大大提升 consumer 的性能。如果 consumer 的消费性能本身就比较慢，设置比较大的 prefetchSize 反而不能有效的达到提升消费性能的目的。因为过大的 prefetchSize 不利于 consumer 端消息的负载均衡。因为通常情况下，我们都会部署多个 consumer 节点来提升消费端的消费性能。

这个优化方案还会存在另外一个潜在风险，当消息被消费之后还没有来得及确认时，client 端发生故障，那么这些消息就有可能会被重新发送给其他 consumer，那么这种风险就需要 client 端能够容忍“重复”消息。

消息的确认过程

ACK_MODE

通过前面的源码分析，基本上已经知道了消息的消费过程，以及消息的批量获取和批量确认，那么接下来再了解下消息的确认过程

从第一节课的学习过程中，我们知道，消息确认有四种 ACK_MODE，分别是

AUTO_ACKNOWLEDGE = 1 自动确认

CLIENT_ACKNOWLEDGE = 2 客户端手动确认

DUPS_OK_ACKNOWLEDGE = 3 自动批量确认

SESSION_TRANSACTED = 0 事务提交并确认

虽然 Client 端指定了 ACK 模式,但是在 Client 与 broker 在交换 ACK 指令的时候,还需要告知 ACK_TYPE,ACK_TYPE 表示此确认指令的类型，不同的 ACK_TYPE 将传递着消息的状态，broker 可以根据不同的 ACK_TYPE 对消息进行不同的操作。

ACK_TYPE

DELIVERED_ACK_TYPE = 0 消息"已接收"，但尚未处理结束

STANDARD_ACK_TYPE = 2 "标准"类型,通常表示为消息"处理成功"，broker 端可以删除消息了

POSION_ACK_TYPE = 1 消息"错误",通常表示"抛弃"此消息，比如消息重发多次后，都无法正确处理时，消息将会被删除或者 DLQ(死信队列)

REDELIVERED_ACK_TYPE = 3 消息需"重发"，比如 consumer 处理消息时抛出了异常，broker 稍后会重新发送此消息

INDIVIDUAL_ACK_TYPE = 4 表示只确认"单条消息",无论在任何 ACK_MODE 下

UNMATCHED_ACK_TYPE = 5 在 Topic 中，如果一条消息在转发给“订阅者”时，发现此消息不符合 Selector 过滤条件，那么此消息将 不会转发给订阅者，消息将会被存储引擎删除(相当于在 Broker 上确认了消息)。

Client 端在不同的 ACK 模式时,将意味着在不同的时机发送 ACK 指令,每个 ACK Command 中会包含 ACK_TYPE,那么 broker 端就可以根据 ACK_TYPE 来决定此消息的后续操作

消息的重发机制原理

消息重发的情况

在正常情况下，有几中情况会导致消息重新发送

Ø 在事务性会话中，没有调用 session.commit 确认消息或者调用 session.rollback 方法回滚消息

Ø 在非事务性会话中，ACK 模式为 CLIENT_ACKNOWLEDGE 的情况下，没有调用 acknowledge 或者调用了 recover 方法；

一个消息被 redelivedred 超过默认的最大重发次数（默认 6 次）时，消费端会给 broker 发送一个”poison ack”(ActiveMQMessageConsumer#dispatch：1460 行)，表示这个消息有毒，告诉 broker 不要再发了。这个时候 broker 会把这个消息放到 DLQ（死信队列）。

死信队列

ActiveMQ 中默认的死信队列是 ActiveMQ.DLQ，如果没有特别的配置，有毒的消息都会被发送到这个队列。默认情况下，如果持久消息过期以后，也会被送到 DLQ 中。

演示效果

参考课程的演示过程自己实践

死信队列配置策略

缺省所有队列的死信消息都被发送到同一个缺省死信队列，不便于管理，可以通过 individualDeadLetterStrategy 或 sharedDeadLetterStrategy 策略来进行修改

```
<destinationPolicy>

          <policyMap>

            <policyEntries>

              <policyEntry topic=">" >

                <pendingMessageLimitStrategy>

                  <constantPendingMessageLimitStrategy
limit="1000"/>

                </pendingMessageLimitStrategy>

              </policyEntry>

               // “>”表示对所有队列生效，如果需要设置指定队列，则直接写队列名称

<policyEntry queue=">">
```

```
                <deadLetterStrategy>

//queuePrefix:设置死信队列前缀

                //useQueueForQueueMessage 设置队列保存到死信。

                  <individualDeadLetterStrategy queuePrefix="DLQ."
useQueueForQueueMessages="true"/>

              </deadLetterStrategy>

              </policyEntry>

            </policyEntries>

          </policyMap>

       </destinationPolicy>
```

自动丢弃过期消息

```
<deadLetterStrategy>

   <sharedDeadLetterStrategy processExpired="false" />

</deadLetterStrategy>
```

死信队列的再次消费

当定位到消息不能消费的原因后，就可以在解决掉这个问题之后，再次消费死信队列中的消息。因为死信队列仍然是一个队列

ActiveMQ 静态网络配置

配置说明

修改 activeMQ 服务器的 activeMQ.xml，增加如下配置

```
<networkConnectors>

    <networkConnector
uri="static://(tcp://192.168.11.153:61616,tcp://192.168.11.154:61616
)"/>

</networkConnectors>
```

两个 Brokers 通过一个 static 的协议来进行网络连接。一个 Consumer 连接到 BrokerB 的一个地址上，当 Producer 在 BrokerA 上以相同的地址发送消息是，此时消息会被转移到 BrokerB 上，也就是说 BrokerA 会转发消息到 BrokerB 上

消息回流

从 5.6 版本开始，在 destinationPolicy 上新增了一个选项 replayWhenNoConsumers 属性，这个属性可以用来解决当 broker1 上有需要转发的消息但是没有消费者时，把消息回流到它原始的 broker。同时把 enableAudit 设置为 false，为了防止消息回流后被当作重复消息而不被分发

通过如下配置，在 activeMQ.xml 中。 分别在两台服务器都配置。即可完成消息回流处理

```
<policyEntry queue=">" enableAudit="false">

 <networkBridgeFilterFactory>

    <conditionalNetworkBridgeFilterFactory
replayWhenNoConsumers="true"/>

 </networkBridgeFilterFactory>

</policyEntry>
```

动态网络连接

ActiveMQ 使用 Multicast 协议将一个 Service 和其他的 Broker 的 Service 连接起来。Multicast 能够自动的发现其他 broker，从而替代了使用 static 功能列表 brokers。用 multicast 协议可以在网络中频繁

multicast://ipadaddress:port?transportOptions

基于 zookeeper+levelDB 的 HA 集群搭建

activeMQ5.9 以后推出的基于 zookeeper 的 master/slave 主从实现。虽然 ActiveMQ 不建议使用 LevelDB 作为存储，主要原因是，社区的主要精力都几种在 kahadb 的维护上，包括 bug 修复等。所以并没有对 LevelDB 做太多的关注，所以他在是不做为推荐商用。但实际上在很多公司，仍然采用了 LevelDB+zookeeper 的高可用集群方案。而实际推荐的方案，仍然是基于 KahaDB 的文件共享以及 Jdbc 的方式来实现。

配置

在三台机器上安装 activemq，通过三个实例组成集群。

修改配置

directory：表示 LevelDB 所在的主工作目录

replicas:表示总的节点数。比如我们的及群众有 3 个节点，且最多允许一个节点出现故障，那么这个值可以设置为 2，也可以设置为 3. 因为计算公式为(replicas/2)+1. 如果我们设置为 4， 就表示不允许 3 个节点的任何一个节点出错。

bind：当当前的节点为 master 时，它会根据绑定好的地址和端口来进行主从复制协议

zkAddress：zk 的地址

hostname：本机 IP

sync：在认为消息被消费完成前，同步信息所存储的策略。

local_mem/local_disk

ActiveMQ 的优缺点

ActiveMQ 采用消息推送方式，所以最适合的场景是默认消息都可在短时间内被消费。数据量越大，查找和消费消息就越慢，消息积压程度与消息速度成反比。

缺点

1.吞吐量低。由于 ActiveMQ 需要建立索引，导致吞吐量下降。这是无法克服的缺点，只要使用完全符合 JMS 规范的消息中间件，就要接受这个级别的 TPS。

2.无分片功能。这是一个功能缺失，JMS 并没有规定消息中间件的集群、分片机制。而由于 ActiveMQ 是伟企业级开发设计的消息中间件，初衷并不是为了处理海量消息和高并发请求。如果一台服务器不能承受更多消息，则需要横向拆分。ActiveMQ 官方不提供分片机制，需要自己实现。

适用场景

1. 对 TPS 要求比较低的系统，可以使用 ActiveMQ 来实现，一方面比较简单，能够快速上手开发，另一方面可控性也比较好，还有比较好的监控机制和界面

不适用的场景

1.消息量巨大的场景。ActiveMQ 不支持消息自动分片机制，如果消息量巨大，导致一台服务器不能处理全部消息，就需要自己开发消息分片功能。