Red Hat AMQ 2021.Q3

使用 AMQ JMS 客户端

用于 AMQ 客户端 2.10

Last Updated: 2023-02-20

Red Hat AMQ 2021.Q3 使用 AMQ JMS 客户端

用于 AMQ 客户端 2.10

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律通告

摘要

本指南描述了如何安装和配置客户端，运行实践示例，并将您的客户端与其他 AMQ 组件搭配使用。

目录

使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 CTO Chris Wright 信息。

第 1 章 概述

AMQ JMS 是一个 Java 消息服务(JMS)2.0 客户端，用于发送和接收 AMQP 消息的消息传递应用。

AMQ JMS 是 AMQ 客户端的一部分，这是一套支持多种语言和平台的消息传递库。有关客户端的概述，请参阅 AMQ 客户端概述。有关此发行版本的详情，请参考 AMQ Clients 2.10 发行注记。

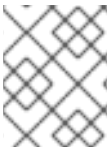
AMQ JMS 基于 Apache Qpid 的 JMS 实施。如需有关 JMS API 的更多信息，请参阅 JMS API 参考 和 JMS 教程。

1.1. 主要特性

- JMS 1.1 和 2.0 兼容
- 用于安全通信的 SSL/TLS
- 灵活的 SASL 身份验证
- 自动重新连接和故障转移
- 准备好与 OSGi 容器搭配使用
- 纯 Java 实施
- 基于 OpenTracing 标准的分布式追踪

重要

AMQ 客户端中的分布式追踪只是一个技术预览功能。技术预览功能不被红帽产品服务等级协议 (SLA) 支持，且可能在功能方面有缺陷。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。有关红帽技术预览功能支持范围的详情，请参阅 https://access.redhat.com/support/offerings/techpreview/。



注意

AMQ JMS 目前不支持分布式事务(XA)。如果您的应用需要分布式事务，建议您使用 AMQ 核心协议 JMS 客户端。

1.2. 支持的标准和协议

AMQ JMS 支持以下业界认可的标准和网络协议：

- Java Message Service API 版本 2.0
- 高级消息队列协议(AMQP)版本 1.0
- AMQP JMS 映射的版本 1.0
- 传输层安全(TLS)协议的版本 1.0、1.1、1.2 和 1.3，后跟 SSL
- 简单的身份验证和安全层(SASL)机制，包括 ANONYMOUS、PLAIN、SCRAM、EXTERNAL 和 GSSAPI(Kerberos)
- 使用 IPv6的现代 TCP

1.3. 支持的配置

有关 AMQ JMS 支持的配置，请参阅红帽客户门户网站中的 Red Hat AMQ 7 支持的配置。

1.4. 术语和概念

本节介绍核心 API 实体，并描述它们如何协同运作。

表 1.1. API 术语

实体	描述
ConnectionFactory	用于创建连接的入口点。
Connection	个网络上两个同级之间的通信通道。它包含会话。
Session	用于生成和使用消息的环境。它包含消息制作者和消费者。
MessageProducer	用于将消息发送到目的地的频道。它具有目标目的地。
MessageConsumer	从目的地接收消息的频道。它具有源目的地。
Destination	消息的指定位置，可以是队列，也可以是主题。
Queue	存储的消息序列.
Topic	存储的用于多播分发的消息序列。
Message	特定于应用的信息。

AMQ JMS 发送和接收 *消息*。*消息使用消息生产者和消费者* 在连接的对等点之间传输 。生产者和消费者通过 *会话* 建立。通过 *连接* 建立会话。*连接由连接工厂* 创建。

发送对等点会创建一个制作者来发送消息。制作者具有在远程同级上标识目标队列或主题 *的目的地*。接收方创建接收消息的消费者。与制作者一样，消费者也有在远程同级上标识源队列或主题的目的地。

目标是 *队列* 或 *主题*。在 JMS 中，队列和主题是包含消息的指定代理实体的客户端显示。

队列实施点对点语义。每条消息仅能被一个使用者看到，消息会在读取后从队列中删除。主题实施发布与订阅语义。每条消息都由多个使用者看到，该消息在读取后可供其他消费者使用。

如需更多信息，请参阅 JMS 指南。

1.5. 文档惯例

sudo 命令
在本文档中，**sudo** 用于任何需要 root 权限的命令。使用 **sudo** 时要小心，因为任何更改都可能会影响整个系统。有关 **sudo** 的详情请参考 使用 sudo 命令。

文件路径

在这个文档中，所有文件路径都对 Linux、UNIX 和类似操作系统有效（例如 **/home/andrea**）。在 Microsoft Windows 中，您必须使用等效的 Windows 路径（例如 **C:\Users\andrea**）。

变量文本

本文档包含代码块，它们需要使用特定于环境的值替换。变量文本括在箭头大括号内，样式为圆形单空间。例如，在以下命令中，将 ***<project-dir>*** 替换为您的环境的值：

```
$ cd <project-dir>
```

第 2 章 安装

本章指导您完成在您的环境中安装 AMQ JMS 的步骤。

2.1. 先决条件

- 您必须有 订阅 才能访问 AMQ 发行文件和存储库。
- 要使用 AMQ JMS 构建程序，您必须安装 Apache Maven。
- 要使用 AMQ JMS，您必须安装 Java。

2.2. 使用 RED HAT MAVEN 存储库

配置您的 Maven 环境，以从红帽 Maven 存储库下载客户端库。

流程

1. 将红帽存储库添加到您的 Maven 设置或 POM 文件。如需示例配置文件，请参阅 第 B.1 节 “使用在线存储库”。

```
<repository>
  <id>red-hat-ga</id>
  <url>https://maven.repository.redhat.com/ga</url>
</repository>
```

2. 将库依赖关系添加到您的 POM 文件。

```
<dependency>
  <groupId>org.apache.qpid</groupId>
  <artifactId>qpid-jms-client</artifactId>
  <version>1.0.0.redhat-00001</version>
</dependency>
```

该客户端现在在 Maven 项目中可用。

2.3. 安装本地 MAVEN 存储库

作为在线存储库的替代选择，可以将 AMQ JMS 作为基于文件的 Maven 存储库安装到本地文件系统中。

流程

1. 使用您的订阅 下载 **AMQ Clients 2.10.0 JMS Maven 存储库**.zip 文件。
2. 将文件内容提取到您选择的目录中。
   在 Linux 或 UNIX 中，使用 **unzip** 命令提取文件内容。

```
$ unzip amq-clients-2.10.0-jms-maven-repository.zip
```

   在 Windows 上，右键单击 .zip 文件并选择" **提取所有"。**

3. 配置 Maven，以使用提取的安装目录中 **maven-repository** 目录中的存储库。如需更多信息，请参阅 第 B.2 节 “使用本地存储库”。

2.4. 安装示例

流程

1. 使用 **git clone** 命令将源存储库克隆到名为 **qpid-jms** 的本地目录中：

```
$ git clone https://github.com/apache/qpid-jms.git qpid-jms
```

2. 进入 **qpid-jms** 目录，并使用 **git checkout** 命令切换到 **1.0.0** 分支：

```
$ cd qpid-jms
$ git checkout 1.0.0
```

在本文档中生成的本地目录被称为 ***<source-dir>***。

第 3 章 入门

本章将引导您完成设置环境并运行简单消息传递程序的步骤。

3.1. 先决条件

- 若要构建示例，必须将 Maven 配置为 使用红帽存储库 或 本地存储库。
- 您必须 安装示例。
- 您必须有一个在 **localhost** 中侦听连接的消息代理。它必须启用匿名访问。如需更多信息，请参阅 启动代理。
- 您必须有一个名为 **queue** 的队列。如需更多信息，请参阅 创建队列。

3.2. 运行 HELLO WORLD

Hello World 示例创建与代理的连接，发送一条包含问候语的消息到 **queue** 队列，然后重新接收它。成功后，它会将收到的消息打印到控制台。

流程

1. 在 ***<source-dir>*/qpid-jms-examples** 目录中运行以下命令来使用 Maven 构建示例：

```
$ mvn clean package dependency:copy-dependencies -DincludeScope=runtime -DskipTests
```

添加 **dependency:copy-dependencies** 会导致依赖项复制到 **target/dependency** 目录中。

2. 使用 **java** 命令来运行示例。
在 Linux 或 UNIX 中：

```
$ java -cp "target/classes:target/dependency/*" org.apache.qpid.jms.example.HelloWorld
```

在 Windows 中：

```
> java -cp "target\classes;target\dependency\*" org.apache.qpid.jms.example.HelloWorld
```

例如，在 Linux 上运行它会产生以下输出：

```
$ java -cp "target/classes/:target/dependency/*" org.apache.qpid.jms.example.HelloWorld
Hello world!
```

这个示例的源代码位于 ***<source-dir>*/qpid-jms-examples/src/main/java** 目录中。JNDI 和日志记录配置位于 ***<source-dir>*/qpid-jms-examples/src/main/resources** 目录中。

第 4 章 CONFIGURATION

本章介绍了将 AMQ JMS 实施绑定到您的 JMS 应用并设置配置选项的流程。

JMS 使用 Java 命名目录接口(JNDI)来注册和查找 API 实施和其他资源。这可让您编写代码到 JMS API，而不将它与特定的实施关联。

配置选项作为连接 URI 上的查询参数公开。

4.1. 配置 JNDI 初始上下文

JMS 应用使用从 **InitialContextFactory** 获取的 JNDI **InitialContext** 对象来查找 JMS 对象，如连接工厂。AMQ JMS 在 **org.apache.qpid.jms.jndi.JmsInitialContextFactory** 类中提供 **InitialContextFactory** 实施。

当 **InitialContext** 对象被实例化时，**InitialContextFactory** 实现会被发现：

```
javax.naming.Context context = new javax.naming.InitialContext();
```

要查找实施，必须在您的环境中配置 JNDI。实现这一目标的方法有三种：使用 **jndi.properties** 文件、使用系统属性或使用初始上下文 API。

使用 jndi.properties 文件

创建名为 **jndi.properties** 的文件，并将其放置在 Java 类路径中。使用键 **java.naming.factory.initial** 添加属性。

示例：使用 jndi.properties 文件设置 JNDI 初始上下文工厂

```
java.naming.factory.initial = org.apache.qpid.jms.jndi.JmsInitialContextFactory
```

在基于 Maven 的项目中，**jndi.properties** 文件放置在 ***<project-dir>*/src/main/resources** 目录中。

使用系统属性

设置 **java.naming.factory.initial** 系统属性。

示例：使用系统属性设置 JNDI 初始上下文工厂

```
$ java -Djava.naming.factory.initial=org.apache.qpid.jms.jndi.JmsInitialContextFactory ...
```

使用初始上下文 API

使用 JNDI 初始上下文 API 以编程方式设置属性。

示例：以编程方式设置 JNDI 属性

```
Hashtable<Object, Object> env = new Hashtable<>();

env.put("java.naming.factory.initial", "org.apache.qpid.jms.jndi.JmsInitialContextFactory");

InitialContext context = new InitialContext(env);
```

请注意，您可以使用同一 API 为连接工厂、队列和主题设置 JNDI 属性。

4.2. 配置连接工厂

JMS 连接工厂是创建连接的入口点。它使用对应用特定配置设置进行编码的连接 URI。

要设置工厂名称和连接 URI，请按照以下格式创建一个属性：您可以将该配置存储在 **jndi.properties** 文件中，或者设置对应的系统属性。

连接工厂的 JNDI 属性格式

```
connectionFactory.<lookup-name> = <connection-uri>
```

例如，这是您可以如何配置一个名为 **app1** 的工厂：

示例：在 jndi.properties 文件中设置连接工厂

```
connectionFactory.app1 = amqp://example.net:5672?jms.clientID=backend
```

然后，您可以使用 JNDI 上下文使用名称 **app1** 来查找配置的连接工厂：

```
ConnectionFactory factory = (ConnectionFactory) context.lookup("app1");
```

4.3. 连接 URI

连接配置使用连接 URI。连接 URI 指定远程主机、端口和一组配置选项，这些选项设置为查询参数。有关可用选项的详情请参考 第 5 章 *配置选项*。

连接 URI 格式

```
<scheme>://<host>:<port>[?<option>=<value>[&<option>=<value>...]]
```

这个方案是 **amqp** 用于未加密的连接，**amqps** 用于 SSL/TLS 连接。

例如，以下连接 URI 通过端口 **5672** 连接到主机 **example.net**，并将客户端 ID 设置为 **backend**：

示例：连接 URI

```
amqp://example.net:5672?jms.clientID=backend
```

故障切换 URI
配置故障转移后，客户端可以自动重新连接到另一服务器（如果与当前服务器的连接丢失）。故障切换 URI 具有前缀 **failover:**，它包含在括号内以逗号分隔的连接 URI 列表。末尾指定了附加选项。

故障转移 URI 格式

```
failover:(<connection-uri>[,<connection-uri>...])[?<option>=<value>[&<option>=<value>...]]
```

例如，以下是可以连接到两个主机 **host1** 或 **host2** 的故障切换 URI：

示例：故障切换 URI

```
failover:(amqp://host1:5672,amqp://host2:5672)?jms.clientID=backend
```

与连接 URI 示例一样，客户端可以使用故障转移配置中的 URI 配置多个不同的设置。这些设置在 第 5 章 *配置选项* 中详细介绍，第 5.5 节 “故障切换选项” 部分特别值得关注。

SSL/TLS Server Name Indication

当 **amqps** 方案用于指定 SSL/TLS 连接时，JVM 的 TLS Server Name Indication(SNI)扩展可以使用 URI 的主机片段在 TLS 握手期间通信所需的服务器主机名。如果指定了完全限定域名（如"myhost.mydomain"），但没有指定非限定名称（例如："myhost"）或裸机 IP 地址，则会自动包含 SNI 扩展。

4.4. 配置队列和主题名称

JMS 提供使用 JNDI 来查找特定于部署的队列和主题资源的选项。

要在 JNDI 中设置队列和主题名称，请使用以下格式创建属性：将此配置放在 **jndi.properties** 文件中，或者设置对应的系统属性。

队列和主题的 **JNDI** 属性格式

```
queue.<lookup-name> = <queue-name>
topic.<lookup-name> = <topic-name>
```

例如，以下属性为两个特定于部署的资源定义名称 **jobs** 和 **notifications** ：

示例：在 **jndi.properties** 文件中设置队列和主题名称

```
queue.jobs = app1/work-items
topic.notifications = app1/updates
```

然后，您可以根据 JNDI 名称查找资源：

```
Queue queue = (Queue) context.lookup("jobs");
Topic topic = (Topic) context.lookup("notifications");
```

4.5. JNDI 属性中的变量扩展

JNDI 属性值可以包含 **${*<variable-name>*}** 格式的变量。该程序库通过按以下位置搜索顺序解析变量值：

- Java 系统属性
- OS 环境变量
- JNDI 属性文件或环境 Hashtable

例如：在 Linux **${HOME}** 中，解析到 **HOME** 环境变量，即当前用户的主目录。

可以使用语法 **${*<variable-name>*:-*<default-value>*}** 提供默认值。如果没有找到 ***<variable-name>*** 的值，则会改为使用默认值。

第 5 章 配置选项

本章列出了 AMQ JMS 的可用配置选项。

JMS 配置选项设置为连接 URI 上的查询参数。如需更多信息，请参阅 第 4.3 节 “连接 URI”。

5.1. JMS 选项

这些选项控制 JMS 对象的产生，如 **Connection**、**Session**、**MessageConsumer** 和 **MessageProducer**。

jms.username

客户端用于验证连接的用户名。

jms.password

客户端用于验证连接的密码。

jms.clientID

客户端应用到连接的客户端 ID。

jms.forceAsyncSend

如果启用，则异步发送来自 **MessageProducer** 的所有信息。否则，只有某些类型（如非持久性消息或事务内的消息）才会异步发送。它默认是禁用的。

jms.forceSyncSend

如果启用，则会同步发送来自 **MessageProducer** 的所有信息。它默认是禁用的。

jms.forceAsyncAcks

如果启用，则异步发送所有消息确认。它默认是禁用的。

jms.localMessageExpiry

如果启用，**MessageConsumer** 接收的所有过期信息都会被过滤掉且未发送。它会被默认启用。

jms.localMessagePriority

如果启用，则预先获取的消息会根据消息优先级值在本地重新排序。它默认是禁用的。

jms.validatePropertyNames

如果启用，则消息属性名称必须是有效的 Java 标识符。它会被默认启用。

jms.receiveLocalOnly

如果启用，调用 **receive** 并带有超时参数会只检查消费者的本地信息缓冲。否则，如果超时过期，将检查远程对等点以确保真正没有消息。它默认是禁用的。

jms.receiveNoWaitLocalOnly

如果启用，调用 **receiveNoWait** 只会检查消费者的本地信息缓冲。否则，将检查远程对等点以确保真正没有可用的消息。它默认是禁用的。

jms.queuePrefix

一个可选的前缀值，添加到从 **Session** 创建的任何 **Queue** 的名称中。

jms.topicPrefix

一个可选的前缀值，添加到从 **Session** 创建的任何 **Topic** 的名称中。

jms.closeTimeout

客户端在返回之前等待正常资源的时间以毫秒为单位。默认值为 60000（60 秒）。

jms.connectTimeout

客户端在返回错误前等待连接建立的时间（毫秒）。默认值为 15000（15 秒）。

jms.sendTimeout

客户端在返回错误前等待 *异步消息发送* 的时间（毫秒为单位）。默认情况下，客户端会无限期等待发送完成。

jms.requestTimeout

客户端等待完成 *各种同步交互* 的时间（以毫秒为单位），如打开制作者或消费者（除外发送），然后返回错误。默认情况下，客户端会无限期等待请求完成。

jms.clientIDPrefix

当 **ConnectionFactory** 创建新 **Connection** 时，用于生成客户端 ID 值的可选前缀值。默认值为 **ID:**。

jms.connectionIDPrefix

当 **ConnectionFactory** 创建新 **Connection** 时，用于生成连接 ID 值的可选前缀值。此连接 ID 在记录 **Connection** 对象中的一些信息时使用，因此可配置的前缀可简化日志的面包。默认值为 **ID:**。

jms.populateJMSXUserID

如果启用，使用连接中经过身份验证的用户名填充每个发送的消息的 **JMSXUserID** 属性。它默认是禁用的。

jms.awaitClientID

如果启用，在 URI 中未配置客户端 ID 的连接会等待以编程方式设置客户端 ID，或确认在发送 AMQP 连接"open" 之前无法设置任何客户端 ID。它会被默认启用。

jms.useDaemonThread

如果启用，连接将守护进程线程用于其 executor，而不是非后台程序线程。它默认是禁用的。

jms.tracing

追踪提供程序的名称。支持的值有 **opentracing** 和 **noop**。默认值为 **noop**。

预定义策略选项

prefetch 策略决定每个 **MessageConsumer** 从远程 peer 获取的信息数，并保存在本地"prefetch"缓冲区中。

jms.prefetchPolicy.queuePrefetch

默认值为 1000。

jms.prefetchPolicy.topicPrefetch

默认值为 1000。

jms.prefetchPolicy.queueBrowserPrefetch

默认值为 1000。

jms.prefetchPolicy.durableTopicPrefetch

默认值为 1000。

jms.prefetchPolicy.all

这可以用于一次性设置所有 prefetch 值。

prefetch 的值可能会影响在队列或共享订阅上向多个消费者发送消息。更高的值可能会导致向每位消费者同时发送的批处理。要实现更加均匀的循环分布，请使用更低的值。

重新传送策略选项

重新传送策略控制如何在客户端上处理重新传送消息。

jms.redeliveryPolicy.maxRedeliveries

根据重新传送的次数控制拒绝传入消息的时间。值 0 表示不接受任何消息重新传送。值 5 允许将消息重新传送五次，以此类推。默认值为 -1，即没有限制。

jms.redeliveryPolicy.outcome

在消息超过配置的 maxRedeliveries 值后，控制应用到消息的结果。支持的值有：**ACCEPTED**、**REJECTED**、**RELEASED MODIFIED_FAILED** 和 **MODIFIED_FAILED_UNDELIVERABLE**。默认值为 **MODIFIED_FAILED_UNDELIVERABLE**。

消息 ID 策略选项

消息 ID 策略控制分配给从客户端发送的消息的数据类型。

jms.messageIDPolicy.messageIDType

默认情况下，生成的 **String** 值用于传出信息中的信息 ID。其他可用的类型有 **UUID**、**UUID_STRING** 和 **PREFIXED_UUID_STRING**。

Presettle 策略选项

Presettle 策略控制生产者或消费者实例何时配置为使用 AMQP 预设置的消息传递语义。

jms.presettlePolicy.presettleAll

如果启用，则所有创建的生产者和非翻译消费者都在预先设置的模式下运作。它默认是禁用的。

jms.presettlePolicy.presettleProducers

如果启用，则所有制作者都以预设模式运作。它默认是禁用的。

jms.presettlePolicy.presettleTopicProducers

如果启用，发送到 **Topic** 或 **TemporaryTopic** 目标的所有制作者都以预设置模式运行。它默认是禁用的。

jms.presettlePolicy.presettleQueueProducers

如果启用，发送到 **Queue** 或 **TemporaryQueue** 目标的所有制作者都以预设置模式运行。它默认是禁用的。

jms.presettlePolicy.presettleTransactedProducers

如果启用，在转换的 **Session** 中创建的任何制作者都以预设置模式运行。它默认是禁用的。

jms.presettlePolicy.presettleConsumers

如果启用，则所有消费者都以预设模式运作。它默认是禁用的。

jms.presettlePolicy.presettleTopicConsumers

如果启用，从 **Topic** 或 **TemporaryTopic** 目标接收的消费者都以预设置模式运行。它默认是禁用的。

jms.presettlePolicy.presettleQueueConsumers

如果启用，从 **Queue** 或 **TemporaryQueue** 目标接收的消费者都以预设置模式运行。它默认是禁用的。

序列化策略选项

反序列化策略提供了控制 Java 类型被信任从对象流中取消序列化的方法，同时从传入的 **ObjectMessage** 检索由序列化 Java **Object** 内容组成的 正文。默认情况下，在尝试对正文进行反序列的过程中，所有类型都是受信任的。默认反序列策略提供 URI 选项，允许指定 Java 类或软件包名称的白名单和黑名单。

jms.deserializationPolicy.whiteList

以逗号分隔的类和软件包名称列表，在对 **ObjectMessage** 的内容进行反序列时允许，除非被 **blackList** 覆盖。这个列表中的名称不是模式值。必须配置具体的类或软件包名称，如 **java.util.Map** 或 **java.util** 所示。软件包匹配包括子软件包.默认值为允许所有.

jms.deserializationPolicy.blackList

以逗号分隔的类和软件包名称列表，在对 **ObjectMessage** 的内容进行降序时应拒绝这些列表。这个列表中的名称不是模式值。必须配置具体的类或软件包名称，如 **java.util.Map** 或 **java.util** 所示。软件包匹配包括子软件包.默认设置是防止 none。

5.2. TCP 选项

使用普通 TCP 连接到远程服务器时，以下选项指定底层套接字的行为：这些选项与任何其他配置选项一起附加到连接 URI 中。

示例：带有传输选项的连接 URI

```
amqp://localhost:5672?jms.clientID=foo&transport.connectTimeout=30000
```

下方列出了一整套 TCP 传输选项：

transport.sendBufferSize

发送缓冲区大小，以字节为单位。默认值为 65536(64 KiB)。

transport.receiveBufferSize

接收缓冲区大小（以字节为单位）。默认值为 65536(64 KiB)。

transport.trafficClass

默认值为 0。

transport.connectTimeout

默认为 60 秒。

transport.soTimeout

默认值为 -1。

transport.soLinger

默认值为 -1。

transport.tcpKeepAlive

默认值为 false。

transport.tcpNoDelay

如果启用，则不要延迟，也不会缓冲 TCP。它会被默认启用。

transport.useEpoll

当可用时，使用原生 epoll IO 层而不是 NIO 层。这可以提高性能。它会被默认启用。

5.3. SSL/TLS 选项

使用 **amqps** URI 方案启用 SSL/TLS 传输。由于 SSL/TLS 传输扩展基于 TCP 的传输功能，因此所有 TCP 传输选项都在 SSL/TLS 传输 URI 上有效。

示例：简单的 SSL/TLS 连接 URI

```
amqps://myhost.mydomain:5671
```

下方列出了完整的 SSL/TLS 传输选项集合。

transport.keyStoreLocation

到 SSL/TLS 密钥存储的路径。如果未设置，则使用 **javax.net.ssl.keyStore** 系统属性的值。

transport.keyStorePassword

SSL/TLS 密钥存储的密码。如果未设置，则使用 **javax.net.ssl.keyStorePassword** 系统属性的值。

transport.trustStoreLocation

SSL/TLS 信任存储的路径。如果未设置，则使用 **javax.net.ssl.trustStore** 系统属性的值。

transport.trustStorePassword

SSL/TLS 信任存储的密码。如果未设置，则使用 **javax.net.ssl.trustStorePassword** 系统属性的值。

transport.keyStoreType

如果未设置，则使用 **javax.net.ssl.keyStoreType** 系统属性的值。如果没有设置系统属性，则默认为 **JKS**。

transport.trustStoreType

如果未设置，则使用 **javax.net.ssl.trustStoreType** 系统属性的值。如果没有设置系统属性，则默认为 **JKS**。

transport.storeType

将 **keyStoreType** 和 **trustStoreType** 设置为相同的值。如果未设置，**keyStoreType** 和 **trustStoreType** 默认为以上指定的值。

transport.contextProtocol

获取 SSLContext 时所用的协议参数。默认值为 **TLS**，如果使用 OpenSSL，则默认为 **TLSv1.2**。

transport.enabledCipherSuites

要启用的密码套件的逗号分隔列表。如果未设置，则使用 context-default 密码。任何禁用的密码都会从此列表中移除。

transport.disabledCipherSuites

要禁用的密码套件的逗号分隔列表。此处列出的密码将从已启用的密码中删除。

transport.enabledProtocols

要启用的协议的逗号分隔列表。如果未设置，则使用上下文默认协议。所有禁用的协议都会从此列表中移除。

transport.disabledProtocols

要禁用的协议的逗号分隔列表。此处列出的协议将从启用的协议列表中删除。默认值为 **SSLv2Hello,SSLv3**。

transport.trustAll

如果启用，请隐式信任提供的服务器证书，无论配置的信任存储是什么。它默认是禁用的。

transport.verifyHost

如果启用，验证连接主机名是否与提供的服务器证书匹配。它会被默认启用。

transport.keyAlias

如果需要向服务器发送客户端证书，请从密钥存储中选择密钥对时使用的别名。

transport.useOpenSSL

如果启用，请将原生 OpenSSL 库用于 SSL/TLS 连接（如果可用）。它默认是禁用的。
如需更多信息，请参阅 第 7.1 节 “启用 OpenSSL 支持”。

5.4. AMQP 选项

以下选项适用于与 AMQP 线路协议相关的行为方面：

amqp.idleTimeout

如果 peer 没有发送 AMQP 帧，则连接失败的时间以毫秒为单位。默认值为 60000（1 分钟）。

amqp.vhost

要连接的虚拟主机。这用于填充 SASL 和 AMQP 主机名字段。默认为来自连接 URI 的主主机名。

amqp.saslLayer

如果启用，则在建立连接时使用 SASL。它会被默认启用。

amqp.saslMechanisms

以逗号分隔的 SASL 机制列表，客户端应允许选择（由服务器提供并可使用配置的凭据）。支持的机制有 EXTERNAL、SCRAM-SHA-256、SCRAM-SHA-1、CRAM-MD5、PLAIN、ANONYMOUS 和适用于 Kerberos 的 GSSAPI。默认设置是允许从除 GSSAPI 以外的所有机制中选择，必须在此明确包含在此处才能启用。

amqp.maxFrameSize

客户端允许的最大 AMQP 帧大小，以字节为单位。此值公告给远程同级。默认值为 1048576(1 MiB)。

amqp.drainTimeout

客户端在发出消费者排空请求时等待来自远程对等点的响应的时间（毫秒）。如果在分配的超时时间内没有看到响应，则链接将被视为失败，相关的使用者将关闭。默认值为 60000（1 分钟）。

amqp.allowNonSecureRedirects

如果启用，则允许 AMQP 在现有连接安全且备用连接没有时重定向到其他主机。例如，如果启用此项，则允许将 SSL/TLS 连接重定向到原始 TCP 连接。它默认是禁用的。

5.5. 故障切换选项

故障转移 URI 以前缀 **failover:** 开头，并在括号内包含一个用逗号分开的连接 URI 列表。末尾指定了附加选项。前缀为 **jms.** 的选项应用于括号之外的整体故障切换 URI，并影响其生命周期中的 **Connection** 对象。

示例：带有故障切换选项的故障切换 URI

```
failover:(amqp://host1:5672,amqp://host2:5672)?
jms.clientID=foo&failover.maxReconnectAttempts=20
```

括号中的单个代理详情可以使用之前定义的 **transport.** 或者 **amqp.** 选项。它们会在每个主机连接时应用。

示例：带有每个连接传输和 AMQP 选项的故障切换 URI

```
failover:(amqp://host1:5672?amqp.option=value,amqp://host2:5672?transport.option=value)?
jms.clientID=foo
```

下方列出了故障转移的所有配置选项。

failover.initialReconnectDelay

客户端在第一次尝试重新连接到远程对等点前等待的时间（毫秒）。默认值为 0，表示第一次尝试会立即进行。

failover.reconnectDelay

重新连接尝试之间的时间（毫秒）。如果没有启用 backoff 选项，这个值会保持不变。默认值为 10。

failover.maxReconnectDelay

客户端在尝试重新连接前等待的最长时间。这个值只有在启用了 backoff 功能时才使用，以确保延迟不会增长过大。默认值为 30 秒。

failover.useReconnectBackOff

如果启用，则重新连接尝试之间的时间会根据配置的倍数而增加。它会被默认启用。

failover.reconnectBackOffMultiplier

用于增大重新连接延迟值的倍数。默认值为 2.0。

failover.maxReconnectAttempts

在向客户端报告连接失败前允许的重新连接尝试数。默认值为 -1，即没有限制。

failover.startupMaxReconnectAttempts

对于之前从未连接到远程对等点的客户端，此选项控制在将连接报告失败前要进行连接的尝试次数。如果未设置，则使用 **maxReconnectAttempts** 的值。

failover.warnAfterReconnectAttempts

在记录警告前尝试失败的次数。默认值为 10。

failover.randomize

如果启用，则在尝试连接到其中之一前，故障转移 URI 集合会被随机清除。这有助于在多个远程同级之间更均匀地分发客户端连接。它默认是禁用的。

failover.amqpOpenServerListAction

控制服务器中的连接"open"帧提供故障转移主机到客户端列表时故障转移传输的行为。有效值为 **REPLACE**、**ADD** 或 **IGNORE**。如果配置了 **REPLACE**，当前服务器以外的所有故障切换 URI 都将替换为服务器提供的 URI。如果配置了 **ADD**，服务器提供的 URI 会被添加到现有的故障切换 URI 集合中，并带有 deduplication。如果配置了 **IGNORE**，服务器的所有更新都会被忽略，且不会更改正在使用的故障切换 URI。默认值为 **REPLACE**。

故障转移 URI 还支持将嵌套选项定义为指定 AMQP 和传输选项值的方法，适用于所有嵌套代理 URI。这可以通过之前为非故障切换代理 URI 提供的相同 **transport.** 和 **amqp.** URI 选项实现，但前缀为 **failover.nested.**。例如，要将 **amqp.vhost** 选项相同的值应用到连接到的每个代理中，则可能具有类似如下的 URI：

示例：带有共享传输和 **AMQP** 选项的故障切换 **URI**

```
failover:(amqp://host1:5672,amqp://host2:5672)?
jms.clientID=foo&failover.nested.amqp.vhost=myhost
```

5.6. 发现选项

客户端具有一个可选的发现模块，它提供了一个自定义故障转移层，用于连接的代理 URI 不在初始 URI 中指定，而是通过与发现代理交互来发现。目前有两种发现代理实施：文件观察程序从文件加载 URI 和多播侦听器，它可以与 ActiveMQ 5.x 代理配合使用，以广播其代理地址以供侦听客户端。

使用发现时的常规故障转移相关选项集与之前详述的相同，主前缀从 **failover.** 改为 **discovery.**，以及用于提供所有发现的代理 URI 的 URI 选项的 **nested** 前缀。例如，如果没有代理 URI 详情，一般发现 URI 可能类似如下：

示例：发现 **URI**

```
discovery:(<agent-uri>)?
discovery.maxReconnectAttempts=20&discovery.discovered.jms.clientID=foo
```

要使用文件监视器发现代理，请创建一个代理 URI，如下所示：

示例：使用文件监视器代理的一个发现 **URI**

```
discovery:(file:///path/to/monitored-file?updateInterval=60000)
```

文件监视器发现代理的 URI 选项如下所列。

updateInterval

文件更改检查之间的时间（毫秒）。默认值为 30000（30 秒）。

要将多播发现代理与 ActiveMQ 5.x 代理搭配使用，请创建如下代理 URI：

示例：使用多播监听器代理的一个发现 URI

```
discovery:(multicast://default?group=default)
```

请注意，在上面的多播代理 URI 中使用 **default** 作为主机是一个特殊值，由代理使用默认的 **239.255.2.3:6155** 替换。您可以对此进行更改，以指定用于多播配置的实际 IP 地址和端口。

下方列出了多播发现代理的 URI 选项。

group

用于侦听更新的多播组。默认值为 **default**。

第 6 章 示例

本章介绍如何通过示例程序使用 AMQ JMS。

有关更多示例，请参阅 AMQ JMS 示例套件 和 Qpid JMS 示例。

6.1. 配置 JNDI 上下文

使用 JMS 的应用通常使用 JNDI 获取应用使用的 **ConnectionFactory** 和 **Destination** 对象。这会将配置与程序分开，并将其与特定的客户端实施隔离。

为了使用这些示例，必须将名为 **jndi.properties** 的文件放在类路径中，以配置 JNDI 上下文，如前文所述。

jndi.properties 文件的内容应与下面显示的内容匹配，它会建立应使用客户端的 **InitialContextFactory** 实施，配置 **ConnectionFactory** 以连接到本地服务器，并定义一个名为 **queue** 的目标队列。

```
# Configure the InitialContextFactory class to use
java.naming.factory.initial = org.apache.qpid.jms.jndi.JmsInitialContextFactory

# Configure the ConnectionFactory
connectionfactory.myFactoryLookup = amqp://localhost:5672

# Configure the destination
queue.myDestinationLookup = queue
```

6.2. 发送消息

这个示例首先创建一个 JNDI **Context**，使用它查找 **ConnectionFactory** 和 **Destination**，使用工厂创建并启动 **Connection**，然后创建一个 **Session**。然后，**MessageProducer** 为 **Destination** 创建，并使用它发送信息。然后 **Connection** 被关闭，程序会退出。

这个 **Sender** 示例的可运行变体位于 **<source-dir>/qpid-jms-examples** 目录中，以及之前在 第 3 章 入门 中的 Hello World 示例。

示例：发送消息

```
package org.jboss.amq.example;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.DeliveryMode;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;

public class Sender {
  public static void main(String[] args) throws Exception {
```

```
    try {
      Context context = new InitialContext(); (1)

      ConnectionFactory factory = (ConnectionFactory) context.lookup("myFactoryLookup");
      Destination destination = (Destination) context.lookup("myDestinationLookup"); (2)

      Connection connection = factory.createConnection("<username>", "<password>");
      connection.setExceptionListener(new MyExceptionListener());
      connection.start(); (3)

      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE); (4)

      MessageProducer messageProducer = session.createProducer(destination); (5)

      TextMessage message = session.createTextMessage("Message Text!"); (6)
      messageProducer.send(message, DeliveryMode.NON_PERSISTENT,
                           Message.DEFAULT_PRIORITY, Message.DEFAULT_TIME_TO_LIVE); (7)

      connection.close(); (8)
    } catch (Exception exp) {
      System.out.println("Caught exception, exiting.");
      exp.printStackTrace(System.out);
      System.exit(1);
    }
  }

  private static class MyExceptionListener implements ExceptionListener {
    @Override
    public void onException(JMSException exception) {
      System.out.println("Connection ExceptionListener fired, exiting.");
      exception.printStackTrace(System.out);
      System.exit(1);
    }
  }
}
```

(1) 创建 JNDI **Context** 来查找 **ConnectionFactory** 和 **Destination** 对象。之前 从 **jndi.properties** 文件中获取配置。

(2) ConnectionFactory 和 Destination 对象使用它们的查找名称从 JNDI 上下文检索。

(3) 该工厂用于创建 **Connection**，然后注册 **ExceptionListener**，然后启动。创建连接时提供的凭据通常从适当的外部配置源获取，确保它们与应用本身保持独立，并可独立更新。

(4) 在 **Connection** 上创建一个未翻译的、自动确认的 **Session**。

(5) **MessageProducer** 创建用于发送信息到 **Destination**。

(6) 使用给定内容创建一个 **TextMessage**。

(7) 发送 **TextMessage**。它将以非持久性方式发送，具有默认优先级且没有过期。

(8) **Connection** 已关闭。**Session** 和 **MessageProducer** 隐式关闭。

请注意，这只是一个示例。现实应用通常会使用长期存在的 MessageProducer，并随着时间推移发送多个消息。打开并关闭每个信息 **Connection**、**Session** 和 **MessageProducer** 通常不高效。

6.3. 接收消息

这个示例首先创建一个 JNDI 上下文，使用它查找 **ConnectionFactory** 和 **Destination**，使用工厂创建并启动 **Connection**，然后创建一个 **Session**。然后为 **Destination** 创建 **MessageConsumer**，会收到一条信息，其内容将输出到控制台。然后，连接关闭，程序退出。在发送示例中 使用相同的 JNDI 配置。

这个 **Receiver** 示例的可执行变体包含在客户端分发的示例目录中，以及前面在 第 3 章 入门中使用的 Hello World 示例。

示例：接收消息

```
package org.jboss.amq.example;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;

public class Receiver {
  public static void main(String[] args) throws Exception {
    try {
      Context context = new InitialContext(); (1)

      ConnectionFactory factory = (ConnectionFactory) context.lookup("myFactoryLookup");
      Destination destination = (Destination) context.lookup("myDestinationLookup"); (2)

      Connection connection = factory.createConnection("<username>", "<password>");
      connection.setExceptionListener(new MyExceptionListener());
      connection.start(); (3)

      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE); (4)

      MessageConsumer messageConsumer = session.createConsumer(destination); (5)

      Message message = messageConsumer.receive(5000); (6)

      if (message == null) { (7)
        System.out.println("A message was not received within given time.");
      } else {
        System.out.println("Received message: " + ((TextMessage) message).getText());
      }

      connection.close(); (8)
    } catch (Exception exp) {
```

```
            System.out.println("Caught exception, exiting.");
            exp.printStackTrace(System.out);
            System.exit(1);
        }
    }

    private static class MyExceptionListener implements ExceptionListener {
        @Override
        public void onException(JMSException exception) {
            System.out.println("Connection ExceptionListener fired, exiting.");
            exception.printStackTrace(System.out);
            System.exit(1);
        }
    }
}
```

1. 创建 JNDI **Context** 来查找 **ConnectionFactory** 和 **Destination** 对象。之前 从 **jndi.properties** 文件中获取配置。
2. **ConnectionFactory** 和 **Destination** 对象使用它们的查找名称从 JNDI **Context** 检索。
3. 该工厂用于创建 **Connection**，然后注册 **ExceptionListener**，然后启动。创建连接时提供的凭据通常从适当的外部配置源获取，确保它们与应用本身保持独立，并可独立更新。
4. 在 **Connection** 上创建一个未翻译的、自动确认的 **Session**。
5. **MessageConsumer** 创建用于接收来自 **Destination** 的信息。
6. 调用接收消息时会发出五秒超时。
7. 将检查结果，如果收到消息，则打印其内容，或注意未收到任何消息。结果会显式转换为 **TextMessage**，这就是我们知道 **Sender** 发送的内容。
8. **Connection** 已关闭。**Session** 和 **MessageConsumer** 隐式关闭。

请注意，这只是一个示例。现实应用程序通常会使用长期存在的 **MessageConsumer**，并随着时间推移接收许多使用它的信息。打开并关闭每个消息的 **Connection**、**Session** 和 **MessageConsumer** 通常并不高效。

第 7 章 安全性

AMQ JMS 具有一系列与安全相关的配置选项，可以根据应用的需求来利用这些配置选项。

在应用程序中创建 **Connection** 时，用户名和密码等基本用户凭证应该直接传递给 **ConnectionFactory**。但是，如果您使用的是 no-gument 工厂方法，也可以在连接 URI 中提供用户凭据。如需更多信息，请参阅 第 5.1 节 “JMS 选项” 部分。

另一个常见的安全考虑是使用 SSL/TLS。当 连接 URI 中指定了 **amqps** URI 方案，且提供各种选项来配置行为时，客户端通过 SSL/TLS 传输连接到服务器。如需更多信息，请参阅 第 5.3 节 “SSL/TLS 选项” 部分。

与前面的项目一致，可能最好限制客户端，仅允许使用服务器可能提供的特定 SASL 机制，而不是从服务器支持的所有程序中选择。如需更多信息，请参阅 第 5.4 节 “AMQP 选项” 部分。

在接收的 **ObjectMessage** 上调用 **getObject()** 的应用程序可能希望限制在反序列过程中创建的类型。请注意，使用 AMQP 类型系统组成的消息正文不使用 **ObjectInputStream** 机制，因此不需要采取此措施。如需更多信息，请参阅 “序列化策略选项”一节 部分。

7.1. 启用 OPENSSL 支持

SSL/TLS 连接可以配置为使用原生 OpenSSL 实现来提高性能。要使用 OpenSSL，必须启用 **transport.useOpenSSL** 选项，且必须在类路径中提供 OpenSSL 支持库。

要在 Red Hat Enterprise Linux 中使用系统安装的 OpenSSL 库，在您的 POM 文件中安装 **openssl** 和 **apr** RPM 软件包并在您的 POM 文件中添加以下依赖关系：

示例：添加原生 OpenSSL 支持

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-tcnative</artifactId>
  <version>2.0.39.Final-redhat-00001</version>
  <classifier>linux-x86_64-fedora</classifier>
</dependency>
```

在 Netty 项目中 提供了 OpenSSL 库实施的列表。

7.2. 使用 KERBEROS 进行身份验证

与正确配置的服务器一起使用时，可以将客户端配置为使用 Kerberos 进行身份验证。要启用 Kerberos，请使用以下步骤：

1. 使用 **amqp.saslMechanisms** URI 选项，将客户端配置为使用 SASL 身份验证的 **GSSAPI** 机制。

   ```
   amqp://myhost:5672?amqp.saslMechanisms=GSSAPI
   failover:(amqp://myhost:5672?amqp.saslMechanisms=GSSAPI)
   ```

2. 将 **java.security.auth.login.config** 系统属性设置为 JAAS 登录配置文件的路径，其中包含 Kerberos **LoginModule** 的适当配置。

   ```
   -Djava.security.auth.login.config=<login-config-file>
   ```

   登录配置文件可能类似以下示例：

```
amqp-jms-client {
   com.sun.security.auth.module.Krb5LoginModule required
   useTicketCache=true;
};
```

使用的确切配置取决于您希望如何为连接创建凭证，以及所使用的 **LoginModule**。有关 Oracle **Krb5LoginModule** 的详情，请查看 Oracle **Krb5LoginModule** 类参考。有关 IBM Java 8 **Krb5LoginModule** 的详情，请查看 IBM **Krb5LoginModule** 类参考。

可以配置 **LoginModule** 来建立用于 Kerberos 流程的凭证，如指定主体以及是否使用现有票据缓存或 keytab。但是，如果 **LoginModule** 配置没有提供建立所有所需凭证的方法，那么如果使用 **ConnectionFactory** 创建 **Connection** 或者之前通过其 URI 选项配置，则可以从客户端 **Connection** 对象请求和传递用户名和密码值。

请注意，Kerberos 仅支持用于身份验证目的。使用 SSL/TLS 连接进行加密.

以下连接 URI 选项可用于影响 Kerberos 身份验证流程。

sasl.options.configScope

用于身份验证的登录配置条目的名称。默认值为 **amqp-jms-client**。

sasl.options.protocol

GSSAPI SASL 流程中使用的协议值。默认值为 **amqp**。

sasl.options.serverName

GSSAPI SASL 流程中使用的 **serverName** 值。默认为来自连接 URI 的服务器主机名。

与前面详述的 **amqp.** 和 **transport.** 选项类似，这些选项必须在每个主机基础上指定，或者作为故障切换 URI 中的 all-host 嵌套选项指定。

第 8 章 消息发送

8.1. 处理未确认的交付

消息传递系统使用消息确认来跟踪发送消息的目标是否真正实现。

发送邮件时，发送邮件之后和确认之前有一段时间（消息为"在机中"）。如果在这段时间内丢失网络连接，消息发送的状态未知，并且传输可能需要在应用代码中特殊处理以确保完成。

以下小节描述了连接失败时消息发送的条件。

不经过翻译且交付未经确认的生产者

如果邮件处于运行状态，会在重新连接后再次发送，只要未设置发送超时且未通过。

不需要用户操作。

有未提交的事务的转译制作者

如果邮件处于转机状态，则在重新连接后再次发送。如果发送是新事务中的第一个，则在重新连接后发送会正常进行。如果之前发送了事务，则事务被视为失败，且任何后续的提交操作都会抛出 **TransactionRolledBackException**。

为确保发送，用户必须重新发送属于失败事务的任何消息。

带有待处理的提交进行转换的制作者

如果提交处于机状态，则事务被视为失败，且任何后续的提交操作都会抛出 **TransactionRolledBackException**。

为确保发送，用户必须重新发送属于失败事务的任何消息。

带有未经确认的交付的非已传输消费者

如果收到消息但尚未确认，则确认消息不会生成任何错误，但不会导致客户端不执行任何操作。

由于收到的消息未被确认，生产者可能会重新发送消息。为避免重复，用户必须通过消息 ID 过滤掉重复的消息。

带有未提交交易的转换消费者

如果一个活跃的事务尚未提交，它将被视为失败，任何待处理的确认都会被丢弃。任何后续提交操作都会抛出 **TransactionRolledBackException**。

生产者可能会重新发送属于该事务的消息。为避免重复，用户必须通过消息 ID 过滤掉重复的消息。

带有待处理的提交进行转换的消费者

如果提交正在进行，则该交易将被视为失败。任何后续提交操作都会抛出 **TransactionRolledBackException**。

生产者可能会重新发送属于该事务的消息。为避免重复，用户必须通过消息 ID 过滤掉重复的消息。

8.2. 扩展会话确认模式

客户端支持除 JMS 规范中定义的两种其他会话确认模式：

个人确认

在这个模式中，应用程序必须使用会话处于 **CLIENT_ACKNOWLEDGE** 模式时使用的 **Message.acknowledge()** 方法单独确认信息。与 **CLIENT_ACKNOWLEDGE** 模式不同，只有目标信息会被确认。所有其他消息仍未被确认。用于激活此模式的整数值为 101。

```
connection.createSession(false, 101);
```

Nofirm

在这种模式下，在将消息分配到客户端之前，服务器将接受消息，并且客户端不会执行任何确认。客户端支持两个整数值来激活此模式，即 100 和 257。

```
connection.createSession(false, 100);
```

第 9 章 日志记录

9.1. 配置日志记录

客户端使用 SLF4J API，允许用户根据自己的需求选择特定的日志实施。例如，用户可以提供 *slf4j-log4j* 绑定来选择 Log4J 实施。有关 SLF4J 的更多详细信息，请访问 其网站。

客户端使用位于 **org.apache.qpid.jms** 层次结构中的 **Logger** 名称，您可以使用它根据您的需要配置日志实现。

9.2. 启用协议日志记录

调试时，从 Qpid Proton AMQP 1.0 库启用额外的协议追踪日志记录有时有用。实现这一目标的方式有两种。

- 将环境变量（而不是 Java 系统属性） **PN_TRACE_FRM** 设置为 **1**。当变量设置为 **1** 时，Proton 放出框架日志记录到控制台。
- 将选项 **amqp.traceFrames=true** 添加到 连接 URI 中，并将 **org.apache.qpid.jms.provider.amqp.FRAMES** 日志记录器配置为日志级别 **TRACE**。这会在 Proton 中添加协议跟踪器，并在日志中包含输出。

您还可以将客户端配置为发送输入和输出字节的低级别追踪。要启用此功能，将选项 **transport.traceBytes=true** 添加到 连接 URI 中，并将 **org.apache.qpid.jms.transports.netty.NettyTcpTransport** 日志记录器配置为日志级别 **DEBUG**。

第 10 章 分布式追踪

客户端根据 OpenTracing 标准的 Jaeger 提供分布式追踪。

10.1. 启用分布式追踪

使用以下步骤在应用程序中启用追踪：

流程

1. 将 Jaeger 客户端依赖项添加到您的 POM 文件中。

```
<dependency>
  <groupId>io.jaegertracing</groupId>
  <artifactId>jaeger-client</artifactId>
  <version>${jaeger-version}</version>
</dependency>
```

${jaeger-version} 必须是 1.0.0 或更高版本。

2. 在连接 URI 中添加 **jms.tracing** 选项。将值设为 **opentracing**

示例：启用了追踪的连接 **URI**

```
amqps://example.net?jms.tracing=opentracing
```

3. 注册全球跟踪器。

示例：全局追踪器注册

```
import io.jaegertracing.Configuration;
import io.opentracing.Tracer;
import io.opentracing.util.GlobalTracer;

public class Example {
    public static void main(String[] args) {
        Tracer tracer = Configuration.fromEnv("<service-name>").getTracer();
        GlobalTracer.registerIfAbsent(tracer);

        // ...
    }
}
```

4. 配置您的环境以进行跟踪。

示例：跟踪配置

```
$ export JAEGER_SAMPLER_TYPE=const
$ export JAEGER_SAMPLER_PARAM=1
$ java -jar example.jar net.example.Example
```

此处显示的配置用于演示目的。如需有关 Jaeger 配置的更多信息，请参阅 通过环境和 Jaeger Sampling 配置。

要查看应用程序捕获的 trace，请使用 Jaeger Getting Started 运行 Jaeger 基础架构和控制台。

第 11 章 互操作性

本章讨论如何与其他 AMQ 组件结合使用 AMQ JMS。有关 AMQ 组件兼容性的概述，请参阅 产品简介。

11.1. 与其他 AMQP 客户端互操作

AMQP 消息 通过 AMQP 类型系统来 构成。采用这种通用格式是不同语言的 AMQP 客户端能够相互互操作的原因之一。本节旨在记录与所使用的各种 JMS 消息类型相关的客户端发送和接收的 AMQP 载荷，以帮助与其他 AMQP 客户端一起使用。

11.1.1. 发送消息

本节旨在记录客户端在使用各种 JMS 消息类型时发送的不同有效负载，以帮助使用其他客户端接收它们。

11.1.1.1. 消息类型

JMS 消息类型	传输的 AMQP 消息的描述
TextMessage	文本将以 amqp-value body 部分发送，其中包含正文文本的 utf8 编码字符串，如果没有设置正文文本，则为 null。带有 符号 键 "x-opt-jms-msg-type" 的消息注释将设置为 字节 值 5。
BytesMessage	BytesMessage 将使用包含 BytesMessage 正文原始字节 的数据 正文部分发送 BytesMessage，并将 properties 部分 *content-type* 字段设置为 符号 值 *"application/octet-stream"*。带有 符号 键 "x-opt-jms-msg-type" 的消息注释将设置为 字节 值 3。
MapMessage	MapMessage 正文将使用包含单个 映射 值的 amqp-value body 部分发送。MapMessage 正文中的任何字节[] 值将编码为映射中 的二进制 条目。带有 符号 键 "x-opt-jms-msg-type" 的消息注释将设置为 字节 值 2。
StreamMessage	StreamMessage 将使用包含 StreamMessage 正文中的条目的 amqp-sequence 正文部分发送 StreamMessage。StreamMessage 正文中的任何字节[] 条目将编码为序列中 的二进制 条目。带有 符号 键 "x-opt-jms-msg-type" 的消息注释将设置为 字节 值 4。
ObjectMessage	ObjectMessage 将使用 数据 正文部分发送，其中包含使用 ObjectOutputStream 序列化 ObjectMessage 正文的字节，并将 properties 部分 *content-type* 字段设置为 符号 值 *"application/x-java-serialized-object"*。带有 符号 键 "x-opt-jms-msg-type" 的消息注释将设置为 字节 值 1。
消息	普通 JMS Message 没有正文，并将作为包含 null 的 amqp-value body 部分发送。带有 符号 键 "x-opt-jms-msg-type" 的消息注释将设置为 字节 值 0。

11.1.1.2. 消息属性

JMS 消息支持设置各种 Java 类型的应用属性。本节介绍这些属性类型到发送消息的 application-properties 部分中的 AMQP 类型值的映射。JMS 和 AMQP 将字符串键用于属性名称。

JMS 属性类型	AMQP 应用程序属性类型
布尔值	布尔值
字节	byte
短	short
int	int
long	long
浮点值	浮点值
双	double
字符串	字符串 或 null

11.1.2. 接收消息

此部分用于记录客户端收到的不同载荷将映射到各种 JMS Message 类型，以帮助使用其他客户端发送消息，供 JMS 客户端接收。

11.1.2.1. 消息类型

如果收到的 AMQP 消息中存在 "x-opt-jms-msg-type" message-annotation，则其值将用于确定用于表示它的 JMS 消息类型，具体取决于下表中详述的映射。这反映了 关于 JMS 客户端发送的消息所讨论消息的映射的 反向流程。

AMQP "x-opt-jms-msg-type" message-annotation 值（类型）	JMS 消息类型
0（字节）	消息
1（字节）	ObjectMessage
2（字节）	MapMessage
3（字节）	BytesMessage
4（字节）	StreamMessage
5（字节）	TextMessage

如果 "x-opt-jms-msg-type" message-annotation 不存在，则下表详细说明消息将如何映射到 JMS Message 类型。请注意，St reamMessage 和 MapMessage 类型仅分配给注解的消息。

接收 AMQP 消息的描述，但没有 "x-opt-jms-msg-type" 注解	JMS 消息类型
• 包含 字符串 或 null 的 amqp-value body 部分。 • 数据 正文部分，属性 *部分内容类型* 字段设置为表示常见文本介质类型的 符号 值，如 *"text/plain"*、*" application/xml"* 或 *"application/json"*。	TextMessage
• 包含 二进制 的 amqp-value body 部分。 • 个 数据 正文部分，其 properties 部分 *content-type* 字段未设置，设置为 符号 值 *"application/octet-stream"*，或设置为不理解与另一消息类型关联的任何值。	BytesMessage
• 个 数据 正文部分，其 properties 部分 *content-type* 字段设置为 符号 值 *"application/x-java-serialized-object"*。 • 包含以上未涵盖的值的 amqp-value body 部分。 • amqp-sequence 正文部分.这将作为对象邮件内的列表来表示。	ObjectMessage

11.1.2.2. 消息属性

本节介绍收到的 AMQP 消息的 application-properties 部分中的值映射到 JMS 消息中使用的 Java 类型。

AMQP 应用程序属性类型	JMS 属性类型
布尔值	布尔值
byte	字节
short	短
int	int
long	long
浮点值	浮点值
double	双
字符串	字符串
null	字符串

11.2. 连接到 AMQ BROKER

AMQ Broker 旨在与 AMQP 1.0 客户端互操作。检查以下内容以确保为 AMQP 消息传递配置了代理：

- 网络防火墙中的端口 5672 已打开。
- 启用了 AMQ Broker AMQP 接收器。请参阅 默认接收器设置。
- 代理上配置了必要的地址。请参阅地址、队列和主题。
- 代理配置为允许来自您的客户端的访问，客户端被配置为发送所需的凭证。请参阅 Broker 安全。

11.3. 连接到 AMQ INTERCONNECT

AMQ 互连可与任何 AMQP 1.0 客户端配合工作。检查以下内容以确保正确配置了组件：

- 网络防火墙中的端口 5672 已打开。
- 路由器配置为允许从您的客户端进行访问，并且客户端配置为发送所需的凭据。请参阅保护网络连接。

附录 A. 使用您的订阅

AMQ 通过软件订阅提供。要管理您的订阅，请访问红帽客户门户中的帐户。

A.1. 访问您的帐户

流程

1. 转至 access.redhat.com。
2. 如果您还没有帐户，请创建一个帐户。
3. 登录到您的帐户。

A.2. 激活订阅

流程

1. 转至 access.redhat.com。
2. 导航到 **My Subscriptions**。
3. 导航到 **激活订阅** 并输入您的 16 位激活号。

A.3. 下载发行文件

要访问 .zip、.tar.gz 和其他发布文件，请使用客户门户查找要下载的相关文件。如果您使用 RPM 软件包或 Red Hat Maven 存储库，则不需要这一步。

流程

1. 打开浏览器并登录红帽客户门户网站 **产品下载页面**，网址为 access.redhat.com/downloads。
2. 查找 **INTEGRATION 类别中 的红帽 AMQ** 条目。
3. 选择所需的 AMQ 产品。此时会打开 **Software Downloads** 页面。
4. 单击组件的 **Download** 链接。

A.4. 为系统注册软件包

要在 Red Hat Enterprise Linux 上安装此产品的 RPM 软件包，必须注册您的系统。如果您使用下载的发行文件，则不需要这一步。

流程

1. 转至 access.redhat.com。
2. 进入 **Registration Assistant**。
3. 选择您的操作系统版本，再继续到下一页。
4. 使用您的系统终端中列出的命令完成注册。

有关注册您的系统的更多信息，请参阅以下资源之一：

- Red Hat Enterprise Linux 7 - 注册系统并管理订阅
- Red Hat Enterprise Linux 8 - 注册系统并管理订阅

附录 B. 使用红帽 MAVEN 存储库

本节论述了如何在您的软件中使用红帽提供的 Maven 存储库。

B.1. 使用在线存储库

红帽维护一个中央 Maven 存储库，用于基于 Maven 的项目。如需更多信息，请参阅 存储库欢迎页面。

可以通过两种方式将 Maven 配置为使用 Red Hat 存储库：

- 将存储库添加到您的 Maven 设置中
- 将软件仓库添加到您的 POM 文件

将存储库添加到 Maven 设置中

这种配置方法适用于您的用户拥有的所有 Maven 项目，只要您的 POM 文件不覆盖存储库配置并启用包含的配置集。

流程

1. 找到 Maven **settings.xml** 文件。它通常位于用户主目录的 **.m2** 目录中。如果文件不存在，请使用文本编辑器创建该文件。
   在 Linux 或 UNIX 中：

   ```
   /home/<username>/.m2/settings.xml
   ```

   在 Windows 中：

   ```
   C:\Users\<username>\.m2\settings.xml
   ```

2. 在 **settings.xml** 文件的 **profiles** 元素中添加包含红帽存储库的新配置集，如下例所示：

   示例：包含 **Red Hat** 软件仓库的 **Maven settings.xml** 文件

   ```
   <settings>
     <profiles>
       <profile>
         <id>red-hat</id>
         <repositories>
           <repository>
             <id>red-hat-ga</id>
             <url>https://maven.repository.redhat.com/ga</url>
           </repository>
         </repositories>
         <pluginRepositories>
           <pluginRepository>
             <id>red-hat-ga</id>
             <url>https://maven.repository.redhat.com/ga</url>
             <releases>
               <enabled>true</enabled>
             </releases>
             <snapshots>
               <enabled>false</enabled>
             </snapshots>
   ```

```
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
    <activeProfile>red-hat</activeProfile>
  </activeProfiles>
</settings>
```

有关 Maven 配置的更多信息，请参阅 Maven 设置参考。

在您的 **POM** 文件中添加软件仓库
要在项目中直接配置存储库，请在 POM 文件的 **repositories** 元素中添加一个新的条目，如下例所示：

示例：包含 **Red Hat** 软件仓库的 **Maven pom.xml** 文件

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>example-app</artifactId>
  <version>1.0.0</version>

  <repositories>
    <repository>
      <id>red-hat-ga</id>
      <url>https://maven.repository.redhat.com/ga</url>
    </repository>
  </repositories>
</project>
```

有关 POM 文件配置的更多信息，请参阅 Maven POM 参考。

B.2. 使用本地存储库

红帽为其部分组件提供基于文件的 Maven 存储库。这些内容作为可下载存档提供，您可以提取到本地文件系统。

要将 Maven 配置为使用本地提取的存储库，请在 Maven 设置或 POM 文件中应用以下 XML：

```
<repository>
  <id>red-hat-local</id>
  <url>${repository-url}</url>
</repository>
```

${repository-url} 必须是包含提取仓库本地文件系统路径的文件 URL。

表 B.1. 本地 Maven 存储库的 URL 示例

操作系统	文件系统路径	URL
Linux 或 UNIX	**/home/alice/maven-repository**	**file:/home/alice/maven-repository**

操作系统	文件系统路径	URL
Windows	**C:\repos\red-hat**	**file:C:\repos\red-hat**

附录 C. 将 AMQ BROKER 与示例搭配使用

AMQ JMS 示例需要一个正在运行的消息代理，它带有名为 **queue** 的队列。使用以下步骤安装和启动代理并定义队列。

C.1. 安装代理

按照 AMQ Broker 入门 以 安装代理 并创建代理 实例 中的内容进行操作。启用匿名访问。

以下步骤将代理实例的位置称为 ***<broker-instance-dir>***。

C.2. 启动代理

流程

1. 使用 **artemis run** 命令启动代理。

```
$ <broker-instance-dir>/bin/artemis run
```

2. 检查控制台输出中是否有启动过程中记录的严重错误。代理日志记录 **Server is now live** （当它就绪时）。

```
$ example-broker/bin/artemis run

        __  __  ____    ____           _
       /  \  |  \/  |/ __ \  |  _ \         | |
      / /\ \ | \  / | |  | | | |_) |_ __ ___ | | _____ _ __
     / /  \ \| |\/| | |  | | |  _ <| '__/ _ \| |/ / _ \ '__|
    / ____ \ | |  | | |__| | | |_) | | | (_) |   <  __/ |
   /_/    \_\_|  |_|\____\_\ |____/|_|  \___/|_|\_\___|_|

    Red Hat AMQ <version>

2020-06-03 12:12:11,807 INFO  [org.apache.activemq.artemis.integration.bootstrap]
AMQ101000: Starting ActiveMQ Artemis Server
...
2020-06-03 12:12:12,336 INFO  [org.apache.activemq.artemis.core.server] AMQ221007:
Server is now live
...
```

C.3. 创建队列

在新终端中，使用 **artemis queue** 命令创建名为 **queue** 的队列。

```
$ <broker-instance-dir>/bin/artemis queue create --name queue --address queue --auto-create-address --anycast
```

系统将提示您回答一系列"是"或无问题。对所有设备回答 **N**。

队列创建后，代理就可与示例程序配合使用。

C.4. 停止代理

运行完示例后，使用 **artemis stop** 命令停止代理。

```
$ <broker-instance-dir>/bin/artemis stop
```

2021-08-31 15:46:02 +1000 修订