Red Hat AMQ 7.7

AMQ JMS クライアントの使用

AMQ Clients 2.7 向け

Last Updated: 2023-02-03

Red Hat AMQ 7.7 AMQ JMS クライアントの使用

AMQ Clients 2.7 向け

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.

法律上の通知

概要

本ガイドでは、クライアントをインストールして設定する方法、実例を実行し、他の AMQ コンポーネントでクライアントを使用する方法を説明します。

目次

第1章 概要

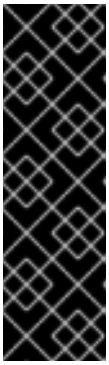
AMQ JMS は、AMQP メッセージを送受信するメッセージングアプリケーションで使用する Java Message Service (JMS) 2.0 クライアントです。

AMQ JMS は、複数の言語やプラットフォームをサポートするメッセージングライブラリースイートである AMQ Clients の一部です。クライアントの概要は、AMQ Clients の概要 を参照してください。本リリースに関する詳細は、AMQ Clients 2.7 リリースノート を参照してください。

AMQ JMS は、Apache Qpid からの JMS 実装に基づいています。JMS API の詳細は、 JMS API reference および JMS tutorial を参照してください。

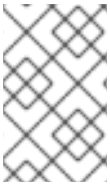
1.1. 主な特長

- JMS 1.1 および 2.0 との互換性
- セキュアな通信用の SSL/TLS
- 柔軟な SASL 認証
- 自動再接続およびフェイルオーバー
- OSGi コンテナーと使用する準備ができました。
- Pure-Java 実装
- OpenTracing 標準に基づく分散トレーシング



重要

AMQ Clients での分散トレーシングはテクノロジープレビュー機能です。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。



注記

AMQ JMS は現在分散トランザクション (XA) をサポートしていません。アプリケーションに分散トランザクションが必要な場合は、AMQ Core Protocol JMS クライアントを使用することが推奨されます。

1.2. サポートされる標準およびプロトコル

AMQ JMS は、以下の業界標準およびネットワークプロトコルをサポートします。

- Java Message Service API のバージョン 2.0
- Advanced Message Queueing Protocol (AMQP) のバージョン 1.0
- AMQP JMS Mapping のバージョン 1.0

- SSL の後継である TLS (Transport Layer Security) プロトコルのバージョン 1.0、1.1、1.2、および 1.3
- ANONYMOUS、PLAIN、SCRAM、EXTERNAL、および GSSAPI (Kerberos) を含む単純な認証およびセキュリティーレイヤー (SASL) メカニズム
- IPv6 での最新の TCP

1.3. サポートされる構成

AMQ JMS は、以下に示す OS と言語のバージョンをサポートしています。詳細は、Red Hat AMQ 7 Supported Configurations を参照してください。

- 以下の JDK を使用する Red Hat Enterprise Linux 7 および 8:
  - OpenJDK 8 および 11
  - Oracle JDK 8
  - IBM JDK 8
- 以下の JDK を使用する Red Hat Enterprise Linux 6:
  - OpenJDK 8
  - Oracle JDK 8
- IBM AIX 7.1 と IBM JDK 8
- Oracle JDK 8 を搭載した Microsoft Windows 10 Pro
- Oracle JDK 8 を搭載した Microsoft Windows Server 2012 R2 および 2016
- Oracle JDK 8 を使用する Oracle Solaris 10 および 11

AMQ JMS は、次の AMQ コンポーネントおよびバージョンとの組み合わせでサポートされています。

- AMQ ブローカーのすべてのバージョン
- AMQ Interconnect のすべてのバージョン
- AMQ Online のすべてのバージョン
- A-MQ 6 バージョン 6.2.1 以降

1.4. 用語および概念

本セクションでは、コア API エンティティーを紹介し、コア API が連携する方法を説明します。

表1.1 API の用語

エンティティー	説明
ConnectionFactory	接続を作成するエントリーポイント。

エンティティー	説明
接続	ネットワーク上の 2 つのピア間の通信チャネル。これにはセッションが含まれます。
Session	メッセージを生成および消費するためのコンテキスト。メッセージプロデューサーとコンシューマーが含まれます。
MessageProducer	メッセージを宛先に送信するためのチャネル。ターゲットの宛先があります。
MessageConsumer	宛先からメッセージを受信するためのチャネル。ソースの宛先があります。
宛先	メッセージの名前付きの場所 (キューまたはトピックのいずれか)。
Queue	メッセージの保存されたシーケンス。
トピック	マルチキャスト配布用のメッセージの保存されたシーケンス。
メッセージ	情報のアプリケーション固有の部分。

AMQ JMS は **メッセージ** を送受信します。メッセージは、**メッセージプロデューサー** と **コンシューマー** を使用して接続されたピア間で転送されます。プロデューサーとコンシューマーは **セッション** 上で確立されます。セッションは**接続**上で確立されます。接続は **接続ファクトリー** によって作成されます。

送信ピアは、メッセージ送信用のプロデューサーを作成します。プロデューサーには、リモートピアでターゲットキューまたはトピックを識別する **宛先** があります。受信ピアは、メッセージ受信用のコンシューマーを作成します。プロデューサーと同様に、コンシューマーにはリモートピアでソースキューまたはトピックを識別する宛先があります。

宛先は、**キュー** または **トピック** のいずれかです。JMS では、キューとトピックはメッセージを保持する名前付きブローカーエンティティーのクライアント側表現です。

キューは、ポイントツーポイントセマンティクスを実装します。各メッセージは 1 つのコンシューマーによってのみ認識され、メッセージは読み取り後にキューから削除されます。トピックはパブリッシュ/サブスクライブセマンティクスを実装します。各メッセージは複数のコンシューマーによって認識され、メッセージは読み取り後も他のコンシューマーで利用できるままになります。

詳細は、JMS tutorial を参照してください。

1.5. 本書の表記慣例

sudo コマンド
本書では、root 権限を必要とするすべてのコマンドに対して **sudo** が使用されています。すべての変更がシステム全体に影響する可能性があるため、**sudo** を使用する場合は注意が必要です。**sudo** の詳細は、sudo コマンドの使用を参照してください。

ファイルパス

本書では、すべてのファイルパスが Linux、UNIX、および同様のオペレーティングシステムで有効です (例: /**home**/**andrea**)。Microsoft Windows では、同等の Windows パスを使用する必要があります (例: **C:\Users\andrea**)。

変数テキスト

本書では、変数を含むコードブロックが紹介されていますが、これは、お客様の環境に固有の値に置き換える必要があります。可変テキストは矢印の中括弧で囲まれ、斜体の等幅フォントとしてスタイル設定されます。たとえば、以下のコマンドでは **<project-dir>** は実際の環境の値に置き換えます。

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

第2章 インストール

本章では、環境に AMQ JMS をインストールする手順を説明します。

2.1. 前提条件

- AMQ リリースファイルおよびリポジトリーにアクセスするには、サブスクリプション が必要です。
- AMQ JMS でプログラムを構築するには、Apache Maven をインストールする必要があります。
- AMQ JMS を使用するには、Java をインストールする必要があります。

2.2. RED HAT MAVEN リポジトリーの使用

Red Hat Maven リポジトリーからクライアントライブラリーをダウンロードするように Maven 環境を設定します。

手順

1. Red Hat リポジトリーを Maven 設定または POM ファイルに追加します。設定ファイルの例は、「オンラインリポジトリーの使用」 を参照してください。

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

2. ライブラリーの依存関係を POM ファイルに追加します。

```
<dependency>
  <groupId>org.apache.qpid</groupId>
  <artifactId>qpid-jms-client</artifactId>
  <version>0.51.0.redhat-00002</version>
</dependency>
```

これで、Maven プロジェクトでクライアントを使用できるようになります。

2.3. ローカル MAVEN リポジトリーのインストール

オンラインリポジトリーの代わりに、AMQ JMS をファイルベースの Maven リポジトリーとしてローカルファイルシステムにインストールできます。

手順

1. サブスクリプションを使用して、**AMQ Clients 2.7.0 JMS Maven リポジトリー**の .zip ファイルをダウンロードします。
2. 選択したディレクトリーにファイルの内容を抽出します。
   Linux または UNIX では、**unzip** コマンドを使用してファイルの内容を抽出します。

```
$ unzip amq-clients-2.7.0-jms-maven-repository.zip
```

Windows では、.zip ファイルを右クリックして、**Extract All** を選択します。

3. 抽出されたインストールディレクトリー内の **maven-repository** ディレクトリーにあるリポジトリーを使用するように Maven を設定します。詳細は、「ローカルリポジトリーの使用」 を参照してください。

2.4. サンプルのインストール

手順

1. サブスクリプションを使用して、**AMQ Clients 2.7.0 JMS**.zip ファイルをダウンロードします。

2. 選択したディレクトリーにファイルの内容を抽出します。
Linux または UNIX では、**unzip** コマンドを使用してファイルの内容を抽出します。

```
$ unzip amq-clients-2.7.0-jms.zip
```

Windows では、.zip ファイルを右クリックして、**Extract All** を選択します。

.zip ファイルの内容を抽出すると、**amq-clients-2.7.0-jms** という名前のディレクトリーが作成されます。これはインストールの最上位ディレクトリーであり、本書では **<install-dir>** と呼びます。

第3章 スタートガイド

本章では、環境を設定して簡単なメッセージングプログラムを実行する手順を説明します。

3.1. 前提条件

- 例を作成するには、Red Hat リポジトリー または ローカルリポジトリー を使用するように Maven を設定する必要があります。
- サンプルをインストール する必要があります。
- **localhost** での接続をリッスンするメッセージブローカーが必要です。匿名アクセスを有効にする必要があります。詳細は、ブローカーの開始を参照してください。
- **queue** という名前のキューが必要です。詳細は、キューの作成 を参照してください。

3.2. 実行中の HELLO WORLD

Hello World の例では、ブローカーへの接続を作成し、グリーティングを含むメッセージを **queue** キューに送信して、受信しなおします。成功すると、受信したメッセージをコンソールに出力します。

手順

1. **<install-dir>/examples** ディレクトリーで次のコマンドを実行し、Maven を使用してサンプルをビルドします。

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

dependency:copy-dependencies を追加すると、依存関係が **target/dependency** ディレクトリーにコピーされます。

2. **java** コマンドを使用して例を実行します。
Linux または UNIX の場合:

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

Windows の場合:

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

たとえば、Linux で実行すると、以下のような出力になります。

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

サンプルのソースコードは **<install-dir>/examples/src/main/java** ディレクトリーにあります。JNDI およびロギング設定は、**<install-dir>/examples/src/main/resources** ディレクトリーにあります。

第4章 設定

本章では、AMQ JMS 実装を JMS アプリケーションにバインドし、設定オプションを設定するプロセスについて説明します。

JMS は Java Naming Directory Interface (JNDI) を使用して、API 実装およびその他のリソースを登録し、検索します。これにより、特定の実装に固有のコードを作成せずに JMS API にコードを作成できます。

設定オプションは、接続 URI でクエリーパラメーターとして公開されます。一部のオプションは、**ConnectionFactory** 実装オブジェクトの対応する **set** および **get** メソッドとしても公開されます。

4.1. 初期コンテキストファクトリーの設定

JMS アプリケーションは **InitialContextFactory** から取得した JNDI **InitialContext** オブジェクトを使用して、接続ファクトリーなどの JMS オブジェクトを検索します。AMQ JMS は、**org.apache.qpid.jms.jndi.JmsInitialContextFactory** クラスで **InitialContextFactory** の実装を提供します。

InitialContextFactory の実装は、**InitialContext** オブジェクトがインスタンス化されると検出されます。

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

実装を見つけるには、お使いの環境で JNDI を設定する必要があります。これを実現するには、**jndi.properties** ファイルを使用する方法とシステムプロパティーを使用する方法の 2 つの主な方法があります。

jndi.properties ファイルの使用

jndi.properties という名前のファイルを作成し、Java クラスパスに配置します。**java.naming.factory.initial** キーでプロパティーを追加します。

例: jndi.properties ファイルを使用した JNDI 初期コンテキストファクトリーの設定

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

Maven ベースのプロジェクトでは、**jndi.properties** ファイルは **<project-dir>/src/main/resources** ディレクトリーに配置されます。

システムプロパティーの使用

java.naming.factory.initial システムプロパティーを設定します。

例: システムプロパティーを使用した JNDI 初期コンテキストファクトリーの設定

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

4.2. 接続ファクトリーの設定

JMS 接続ファクトリーは、接続を作成するためのエントリーポイントです。これは、アプリケーション固有の設定をエンコードする接続 URI を使用します。

ファクトリー名と接続 URI を設定するには、以下の形式でプロパティーを作成します。この設定は、**jndi.properties** ファイルに保存するか、対応するシステムプロパティーを設定できます。

接続ファクトリーの JNDI プロパティー形式

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

たとえば、以下のように **app1** という名前のファクトリーを設定します。

例: jndi.properties ファイルでの接続ファクトリーの設定

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

その後、JNDI コンテキストを使用して、**app1** の名前を使用して設定済みの接続ファクトリーを検索できます。

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

4.3. 接続 URI

接続ファクトリーは、次の形式の接続 URI を使用して設定されます。

接続 URI 形式

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

たとえば、以下はポート **5672** でホスト **example.net** に接続する接続 URI で、クライアント ID を **backend** に設定します。

例: 接続 URI

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

利用可能な接続オプションについては、この後のセクションで説明します。

フェイルオーバーが設定されると、現在のサーバーへの接続が失われた場合、クライアントは別のサーバーに自動的に再接続できます。フェイルオーバー URI には接頭辞 **failover:** があり、括弧内にサーバー URI のコンマ区切りリストが含まれます。追加のオプションは最後に指定されます。

フェイルオーバー URI 形式

```
failover:(amqp[s]://<host>:<port>[,amqp[s]://<host>:<port>...])[?<option>=<value>[&<option>=<value>...]]
```

接続 URI の例と同様に、クライアントはフェイルオーバー設定で URI を使用して多数の異なる設定で設定できます。これらの設定については、以下で詳しく説明します。「フェイルオーバーオプション」 セクションは特に興味深いものです。

amqps スキームを使用して SSL/TLS 接続を指定する場合、URI からのホスト名セグメントは JVM の TLS SNI (Server Name Indication) 拡張によって使用でき、TLS ハンドシェイク時に必要なサーバーのホスト名を通信できます。SNI 拡張は、完全修飾ドメイン名 (例: myhost.mydomain) が指定されている

場合に自動的に含まれますが、修飾されていない名前 (myhost など) やベア IP アドレスが使用される場合は定義されません。

4.4. JMS オプション

これらのオプションは、**Connection**、**Session**、**MessageConsumer**、**MessageProducer** などの JMS オブジェクトの動作を制御します。

jms.username

接続の認証に使用されるユーザー名。

jms.password

接続の認証に使用されるパスワード。

jms.clientID

接続に適用されるクライアント ID。

jms.forceAsyncSend

有効にすると、**MessageProducer** からのすべてのメッセージが非同期に送信されます。それ以外の場合、非永続メッセージやトランザクション内のメッセージなど、特定の種類のみが非同期で送信されます。これはデフォルトでは無効にされます。

jms.forceSyncSend

有効にすると、**MessageProducer** からのすべてのメッセージが同期的に送信されます。これはデフォルトでは無効にされます。

jms.forceAsyncAcks

有効にすると、すべてのメッセージ承認は非同期で送信されます。これはデフォルトでは無効にされます。

jms.localMessageExpiry

有効にすると、**MessageConsumer** が受信した期限切れのメッセージはフィルターされ、配信されません。これは、デフォルトで有効になっています。

jms.localMessagePriority

有効にすると、事前にフェッチされたメッセージはメッセージの優先度の値に基づいてローカルで並べ替えられます。これはデフォルトでは無効にされます。

jms.validatePropertyNames

有効にすると、メッセージプロパティー名が有効な Java 識別子である必要があります。これは、デフォルトで有効になっています。

jms.receiveLocalOnly

有効になっている場合、timeout 引数を指定して **receive** する呼び出しは、コンシューマーのローカルメッセージバッファーのみをチェックします。タイムアウトが期限切れになると、リモートピアをチェックして、メッセージがないことを確認します。これはデフォルトでは無効にされます。

jms.receiveNoWaitLocalOnly

有効になっている場合、**receiveNoWait** の呼び出しは、コンシューマーのローカルメッセージバッファーのみをチェックします。それ以外の場合は、リモートピアをチェックして、使用可能なメッセージが実際にないことを確認します。これはデフォルトでは無効にされます。

jms.queuePrefix

Session から作成される **Queue** の名前に追加される任意の接頭辞値。

jms.topicPrefix

Session から作成される **Topic** の名前に追加される任意の接頭辞値。

jms.closeTimeout

返される前にクライアントが通常のリソースクリップを待つ時間 (ミリ秒単位)。デフォルトは 60000 (60 秒) です。

jms.connectTimeout

エラーを返す前にクライアントが接続確立を待つ時間 (ミリ秒単位)。デフォルトは 15000 (15 秒) です。

jms.sendTimeout

エラーを返す前に、クライアントが **同期メッセージの送信** の完了を待つ時間 (ミリ秒単位)。デフォルトでは、クライアントは送信が完了するまで無期限に待機します。

jms.requestTimeout

エラーを返す前に、リモートピアでプロデューサーまたはコンシューマー (送信を除く) を開くなど、**さまざまな同期インタラクション** が完了するまでクライアントが待機する時間 (ミリ秒単位)。デフォルトでは、クライアントはリクエストが完了するまで無期限に待機します。

jms.clientIDPrefix

ConnectionFactory によって新しい **Connection** が作成されると、クライアント ID 値を生成するために使用されるオプションの接頭辞値。デフォルトは **ID:** です。

jms.connectionIDPrefix

ConnectionFactory によって新しい **Connection** が作成されると、接続 ID 値を生成するために使用されるオプションの接頭辞値。この接続 ID は **Connection** オブジェクトから一部の情報をログに記録する際に使用されるため、設定可能な接頭辞によりログのブレッドをより容易にすることができます。デフォルトは **ID:** です。

jms.populateJMSXUserID

有効にすると、接続から認証済みユーザー名を使用して、送信された各メッセージの **JMSXUserID** プロパティーを設定します。これはデフォルトでは無効にされます。

jms.awaitClientID

有効にすると、URI でクライアント ID が設定されていない接続は、クライアント ID がプログラムによって設定されるのを待機するか、そうでなければ設定できないことを通知するために使用されている接続は、AMQP 接続オープンを送信する前に待機します。これは、デフォルトで有効になっています。

jms.useDaemonThread

有効にすると、コネクションはデーモン以外のスレッドではなく、エグゼキューターにデーモンスレッドを使用します。これはデフォルトでは無効にされます。

jms.tracing

トレースプロバイダーの名前。サポートされる値は **opentracing** および **noop** です。デフォルトは **noop** です。

Prefetch ポリシーオプション

Prefetch ポリシーは、各 **MessageConsumer** がリモートピアから取得し、ローカルの prefetch バッファーに保持するメッセージの数を決定します。

jms.prefetchPolicy.queuePrefetch

デフォルトは 1000 です。

jms.prefetchPolicy.topicPrefetch

デフォルトは 1000 です。

jms.prefetchPolicy.queueBrowserPrefetch

デフォルトは 1000 です。

jms.prefetchPolicy.durableTopicPrefetch

デフォルトは 1000 です。

jms.prefetchPolicy.all

これは、すべての事前にフェッチされた値を一度に設定するために使用できます。

prefetch の値は、キューまたは共有サブスクリプションの複数のコンシューマーへのメッセージの分散に影響します。値が大きいと、各コンシューマーに一度に送信されるバッチが大きくなる可能性があります。より均等にラウンドロビンの分散を実現するには、小さい値を使用します。

再配信ポリシーオプション

再配信ポリシーは、クライアント上で再配信されたメッセージの処理方法を制御します。

jms.redeliveryPolicy.maxRedeliveries

受信メッセージが再配信された回数に基づいて拒否されるタイミングを制御します。値が 0 の場合は、メッセージの再配信が許可されないことを示します。値が 5 の場合、メッセージを 5 回再送することができます。デフォルトは -1 で、無制限を意味します。

jms.redeliveryPolicy.outcome

設定された maxRedeliveries 値を超過した後にメッセージに適用される結果を制御します。サポートされる値は **ACCEPTED**、**REJECTED**、**RELEASED**、**MODIFIED_FAILED**、および **MODIFIED_FAILED_UNDELIVERABLE** です。デフォルト値は **MODIFIED_FAILED_UNDELIVERABLE** です。

メッセージ ID ポリシーオプション

メッセージ ID ポリシーは、クライアントから送信されたメッセージに割り当てられたメッセージ ID のデータタイプを制御します。

jms.messageIDPolicy.messageIDType

デフォルトでは、生成された **String** 値は送信メッセージのメッセージ ID に使用されます。その他の利用可能なタイプは、**UUID**、**UUID_STRING**、および **PREFIXED_UUID_STRING** です。

Presettle ポリシーオプション

Presettle ポリシーは、AMQP の事前設定されたメッセージングセマンティクスを使用するように設定されているプロデューサーまたはコンシューマーインスタンスが設定されるタイミングを制御します。

jms.presettlePolicy.presettleAll

有効にすると、作成されたプロデューサーおよび非トランザクションコンシューマーはすべて、事前設定モードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleProducers

有効にすると、すべてのプロデューサーが事前設定モードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleTopicProducers

有効にすると、**Topic** または **TemporaryTopic** 宛に送信されるプロデューサーは事前設定モードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleQueueProducers

有効にすると、**Queue** または **TemporaryQueue** 宛に送信されるプロデューサーは事前設定モードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleTransactedProducers

有効にすると、トランザクション **Session** で作成されたすべてのプロデューサーがプリセットモードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleConsumers

有効にすると、すべてのコンシューマーは事前に設定されたモードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleTopicConsumers

有効にすると、**Topic** または **TemporaryTopic** 宛から受信されたコンシューマーはすべて、事前設定モードで動作します。これはデフォルトでは無効にされます。

jms.presettlePolicy.presettleQueueConsumers

有効にすると、**Queue** または **TemporaryQueue** 宛から受信されるコンシューマーは事前設定モードで動作します。これはデフォルトでは無効にされます。

デシリアライズポリシーオプション

デシリアライズポリシーは、シリアライズされた Java **Object** コンテンツで設定される受信 **ObjectMessage** からボディーを取得しつつ、どの Java タイプをオブジェクトストリームからデシリアライズするかを制御する手段を提供します。デフォルトでは、ボディーのデシリアライズの試行時にすべてのタイプが信頼されます。デフォルトのデシリアライズポリシーは、ホワイトリストと Java クラスまたはパッケージ名のブラックリストを指定できるようにする URI オプションを提供します。

jms.deserializationPolicy.whiteList

ObjectMessage のコンテンツをデシリアライズする際に許可されるクラス名のコンマ区切りリスト (**blackList** によって上書きされない限り)。このリストの名前はパターンの値ではありません。**java.util.Map** または **java.util** にあるように、正確なクラスまたはパッケージ名を設定する必要があります。パッケージの一致には、サブパッケージが含まれます。デフォルトではすべてを許可します。

jms.deserializationPolicy.blackList

ObjectMessage の内容をデシリアライズする際に拒否されるべきであるクラス名とパッケージ名のコンマ区切りリスト。このリストの名前はパターンの値ではありません。**java.util.Map** または **java.util** にあるように、正確なクラスまたはパッケージ名を設定する必要があります。パッケージの一致には、サブパッケージが含まれます。デフォルトでは、none が回避されます。

4.5. TCP オプション

プレーン TCP を使用してリモートサーバーに接続する場合、以下のオプションは基礎となるソケットの動作を指定します。これらのオプションは、他の設定オプションと共に接続 URI に追加されます。

例: トランスポートオプションを持つ接続 URI

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

TCP トランスポートオプションの完全なセットを以下に示します。

transport.sendBufferSize

送信バッファーサイズ (バイト単位)。デフォルトは、65536 (64 KiB) です。

transport.receiveBufferSize

受信バッファーサイズ (バイト単位)。デフォルトは、65536 (64 KiB) です。

transport.trafficClass

デフォルトは 0 です。

transport.connectTimeout

デフォルトは 60 秒です。

transport.soTimeout

デフォルトは 1 です。

transport.soLinger

デフォルトは 1 です。

transport.tcpKeepAlive

デフォルトは false です。

transport.tcpNoDelay

有効な場合、TCP 送信の遅延やバッファーを行いません。これは、デフォルトで有効になっています。

transport.useEpoll

利用可能な場合は、NIO レイヤーの代わりにネイティブの epoll IO レイヤーを使用します。これにより、パフォーマンスが向上します。これは、デフォルトで有効になっています。

4.6. SSL/TLS オプション

SSL/TLS トランスポートは、**amqps** URI スキームを使用して有効にします。SSL/TLS トランスポートは TCP ベースのトランスポートの機能を拡張するため、すべての TCP トランスポートオプションは SSL/TLS トランスポート URI で有効です。

例: 簡単な SSL/TLS 接続 URI

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

SSL/TLS トランスポートオプションの完全なセットを以下に示します。

transport.keyStoreLocation

SSL/TLS キーストアへのパス。設定しないと、**javax.net.ssl.keyStore** システムプロパティーの値が使用されます。

transport.keyStorePassword

SSL/TLS キーストアのパスワード。設定しないと、**javax.net.ssl.keyStorePassword** システムプロパティーの値が使用されます。

transport.trustStoreLocation

SSL/TLS トラストストアへのパス。設定しないと、**javax.net.ssl.trustStore** システムプロパティーの値が使用されます。

transport.trustStorePassword

SSL/TLS トラストストアのパスワード。設定しないと、**javax.net.ssl.trustStorePassword** システムプロパティーの値が使用されます。

transport.keyStoreType

設定しないと、**javax.net.ssl.keyStoreType** システムプロパティーの値が使用されます。システムプロパティーが設定されていない場合、デフォルトは **JKS** です。

transport.trustStoreType

設定しないと、**javax.net.ssl.trustStoreType** システムプロパティーの値が使用されます。システムプロパティーが設定されていない場合、デフォルトは **JKS** です。

transport.storeType

keyStoreType と **trustStoreType** の両方を同じ値に設定します。未設定の場合は、**keyStoreType** および **trustStoreType** が上記の値にデフォルト設定されます。

transport.contextProtocol

SSLContext の取得時に使用されるプロトコル引数。デフォルトは **TLS** です。OpenSSL を使用している場合は **TLSv1.2** です。

transport.enabledCipherSuites

有効にする暗号スイートのコンマ区切りリスト。未設定の場合は、context-default 暗号が使用されます。無効にした暗号は、この一覧から削除されます。

transport.disabledCipherSuites

無効にする暗号スイートのコンマ区切りリスト。ここに挙げられている暗号は、有効な暗号から削除されます。

transport.enabledProtocols

有効にするプロトコルのコンマ区切りリスト。未設定の場合は、context-default プロトコルが使用されます。無効にされたプロトコルはすべてこの一覧から削除されます。

transport.disabledProtocols

無効にするプロトコルのコンマ区切りリスト。ここに挙げられているプロトコルは、有効なプロトコルリストから削除されます。デフォルトは **SSLv2Hello,SSLv3** です。

transport.trustAll

有効にすると、設定されたトラストストアに関係なく、提供されたサーバー証明書を暗黙的に信頼します。これはデフォルトでは無効にされます。

transport.verifyHost

有効な場合は、接続ホスト名が、提供されたサーバー証明書と一致することを確認します。これは、デフォルトで有効になっています。

transport.keyAlias

クライアント証明書をサーバーに送信する必要がある場合には、キーストアからキーペアを選択する際に使用するエイリアス。

transport.useOpenSSL

有効にすると、利用可能な場合は SSL/TLS 接続にネイティブ OpenSSL ライブラリーを使用します。これはデフォルトでは無効にされます。
詳細は、「OpenSSL サポートの有効化」 を参照してください。

4.7. AMQP オプション

以下のオプションは、AMQP ワイヤプロトコルに関連する動作の要素に適用されます。

amqp.idleTimeout

ピアが AMQP フレームを送信しない場合に、接続に失敗するまでの時間 (ミリ秒単位)。デフォルトの期間は 60000 (1 分) です。

amqp.vhost

接続する仮想ホスト。これは、SASL および AMQP ホスト名フィールドを設定するために使用されます。デフォルトは、接続 URI からのメインホスト名です。

amqp.saslLayer

有効にすると、SASL は接続を確立するときに使用されます。これは、デフォルトで有効になっています。

amqp.saslMechanisms

サーバーによって提供される場合に、クライアントが選択でき、設定された認証情報で使用可能である場合に、クライアントが選択できる SASL メカニズムのコンマ区切りリスト。サポートされるメカニズムは、EXTERNAL、SCRAM-SHA-256、SCRAM-SHA-1、CRAM-MD5、PLAIN、ANONYMOUS、および Kerberos の GSSAPI です。デフォルトでは、ここで明示的に有効にするために明示的に組み込む必要のある GSSAPI 以外のすべてのメカニズムの選択を許可します。

amqp.maxFrameSize

クライアントによって許可される最大 AMQP フレームサイズ (バイト単位)。この値は、リモートピアにアドバタイズされます。デフォルトは 1048576 (1 MiB) です。

amqp.drainTimeout

コンシューマーのドレイン要求が作成されると、クライアントがリモートピアからの応答を待つ時間 (ミリ秒単位)。割り当てられたタイムアウト期間内に応答が見られない場合、リンクは失敗したと見なされ、関連するコンシューマーは閉じられます。デフォルトの期間は 60000 (1 分) です。

amqp.allowNonSecureRedirects

有効にすると、既存の接続がセキュアで、別の接続が利用できない場合に AMQP の代替ホストへのリダイレクトを許可します。たとえば、これを有効にすると、SSL/TLS 接続の raw TCP 接続へのリダイレクトが許可されます。これはデフォルトでは無効にされます。

4.8. フェイルオーバーオプション

フェイルオーバー URI は接頭辞 **failover:** で始まり、括弧内に接続 URI のコンマ区切りリストが含まれます。追加のオプションは最後に指定されます。**jms.** で始まるオプションは、括弧外にある全体的なフェイルオーバー URI に適用され、その有効期間の **Connection** オブジェクトに影響します。

例: フェイルオーバーオプションを含むフェイルオーバー URI

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

括弧内の個々のブローカー詳細は、前のステップで定義した **transport.** オプションまたは **amqp.** オプションを使用できます。各ホストが接続されていると、これらが適用されます。

例: 接続ごとのトランスポートと AMQP オプションを持つフェイルオーバー URI

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

フェイルオーバーのすべての設定オプションは以下のとおりです。

failover.initialReconnectDelay

クライアントが最初にリモートピアへの再接続を試みるまで待機する時間 (ミリ秒単位)。デフォルトは 0 で、最初の試行がすぐに実行されます。

failover.reconnectDelay

再接続試行の間隔 (ミリ秒単位)。バックオフオプションが有効になっていない場合、この値は定数のままになります。デフォルトは 10 です。

failover.maxReconnectDelay

クライアントが再接続を試みるまでに待機する最大時間。この値は、遅延が大きくなりすぎないようにバックオフ機能が有効な場合にのみ使用されます。デフォルトは 30 秒です。

failover.useReconnectBackOff

有効にすると、設定された乗数に基づいて再接続試行の間隔が長くなります。これは、デフォルトで有効になっています。

failover.reconnectBackOffMultiplier

再接続遅延値を拡張するために使用される乗数。デフォルトは 2.0 です。

failover.maxReconnectAttempts

接続をクライアントに失敗したと報告する前に許可される再接続試行の数。デフォルトは -1 で、無制限を意味します。

failover.startupMaxReconnectAttempts

以前にリモートピアに接続されていないクライアントの場合、このオプションは、接続を失敗と報告する前に接続を試行する回数を制御します。未設定の場合は、**maxReconnectAttempts** の値が使用されます。

failover.warnAfterReconnectAttempts

警告がログに記録されるまでの再接続試行の失敗回数。デフォルトは 10 です。

failover.randomize

有効にすると、いずれかの接続を試行する前にフェイルオーバー URI のセットが無作為にシャッフルされます。これにより、クライアント接続を複数のリモートピアに均等に分散させることができます。これはデフォルトでは無効にされます。

failover.amqpOpenServerListAction

サーバーから接続オープンフレームがクライアントへのフェイルオーバーホストの一覧を提供する場合に、フェイルオーバートランスポートがどのように動作するかを制御します。有効な値は **REPLACE**、**ADD**、または **IGNORE** です。**REPLACE** が設定されている場合、現在のサーバーのフェイルオーバー URI はすべてサーバーによって提供されるものに置き換えられます。**ADD** が設定されている場合、サーバーが提供する URI は重複排除を使用して既存のフェイルオーバー URI セットに追加されます。**IGNORE** を設定すると、サーバーからの更新は無視され、使用中のフェイルオーバー URI のセットに変更は行われません。デフォルトは **REPLACE** です。

フェイルオーバー URI は、AMQP およびトランスポートオプションをすべてネスト化されたブローカー URI に指定する手段として、ネストされたオプションの定義もサポートします。これは、同じ **transport.** と **amqp.** を使用して実行できます。非フェイルオーバーブローカー URI について前述されている URI オプションが **failover.nested.** で接頭辞として付けられます。たとえば、**amqp.vhost** オプションに同じ値を接続されたすべてのブローカーに適用するには、以下のような URI が含まれる場合があります。

例: 共有トランスポートと AMQP オプションを持つフェイルオーバー URI

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

4.9. 検出オプション

クライアントには任意の検出モジュールがあります。これは、接続するブローカー URI が初期 URI で指定されず、検出エージェントと対話して検出されるカスタムフェイルオーバー層を提供します。現在、2 つの検出エージェントの実装があります。これは、ファイルから URI をロードするファイル監視と、クライアントをリッスンするブローカーアドレスをブロードキャストするように設定された ActiveMQ 5.x ブローカーと連携するマルチキャストリスナーです。

検出を使用する際のフェイルオーバー関連のオプションの一般的なセットは前述のものと同じで、主な接頭辞は **failover.** から **discovery.** に変更されています。また、検出されたすべてのブローカー URI に共通する URI オプションを指定するために使用される **nested** 接頭辞があります。たとえば、エージェントの URI の詳細がないと、一般的な検出 URI は以下のようになります。

例: 検出 URI

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

ファイルウォッチャー検出エージェントを使用するには、以下のようなエージェント URI を作成します。

例: ファイルウォッチャーエージェントを使用した検出 URI

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

ファイルウォッチャーの検出エージェントの URI オプションを以下に示します。

updateInterval
ファイル変更を確認する間隔 (ミリ秒単位)。デフォルトは 30000 (30 秒) です。

ActiveMQ 5.x ブローカーでマルチキャスト検出エージェントを使用するには、以下のようなエージェント URI を作成します。

例: マルチキャストリスナーエージェントを使用した検出 URI

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

上記のマルチキャストエージェント URI のホストとして **default** をホストとして使用することは、エージェントがデフォルトの **239.255.2.3:6155** に置き換える特別な値であることに注意してください。これを変更して、マルチキャスト設定で使用している実際の IP アドレスとポートを指定できます。

マルチキャスト検出エージェントの URI オプションを以下に示します。

group
更新をリッスンするために使用されるマルチキャストグループ。デフォルトは **default** です。

4.10. JNDI リソースの設定

4.10.1. キューおよびトピック名の設定

JMS は、JNDI を使用してデプロイメント固有のキューとトピックリソースを検索するオプションを提供します。

JNDI でキューおよびトピック名を設定するには、以下の形式でプロパティーを作成します。この設定を **jndi.properties** ファイルに置くか、対応するシステムプロパティーを設定します。

キューおよびトピックの JNDI プロパティー形式

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

たとえば、以下のプロパティーは、2 つのデプロイメント固有のリソースの名前、**jobs** および **notifications** を定義します。

例: jndi.properties ファイルでのキューおよびトピック名の設定

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

その後、JNDI 名でリソースを検索できます。

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

4.10.2. プログラムによる JNDI プロパティーの設定

jndi.properties ファイルまたはシステムプロパティーを使用して JNDI を設定する代わりに、JNDI 初期コンテキスト API を使用してプログラムでプロパティーを定義できます。

例: プログラムでの JNDI プロパティーの設定

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

env.put("java.naming.factory.initial", "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
env.put("connectionFactory.app1", "amqp://example.net:5672?jms.clientID=backend");
env.put("queue.jobs", "app1/work-items");
env.put("topic.notifications", "app1/updates");

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

4.10.3. JNDI プロパティーの変数拡張

JNDI プロパティー値には、**${<variable-name>}** 形式の変数を含めることができます。ライブラリーは、以下の場所の順序を検索して、変数の値を解決します。

- Java システムプロパティー
- OS 環境変数
- JNDI プロパティーファイルまたは環境ハッシュテーブル

たとえば、Linux **${HOME}** では、現在のユーザーのホームディレクトリーである **HOME** 環境変数に解決されます。

構文 **${<variable-name>:-<default-value>}** を使用してデフォルト値を指定できます。**<variable-name>** の値が見つからない場合は、代わりにデフォルト値が使用されます。

第5章 例

本章では、サンプルプログラムで AMQ JMS を使用する方法について説明します。

その他の例については、JMS サンプルスイート を参照してください。

5.1. JNDI コンテキストの設定

通常、JMS を使用するアプリケーションは JNDI を使用してアプリケーションが使用する **ConnectionFactory** および **Destination** オブジェクトを取得します。これにより、設定がプログラムから分離され、特定のクライアント実装から分離されます。

これらの例を使用する場合は、前述のように JNDI コンテキストを設定するために、**jndi.properties** という名前のファイルをクラスパスに配置する必要があります。

jndi.properties ファイルの内容は、以下に示す内容と一致している必要があります。これにより、クライアントの **InitialContextFactory** 実装を使用し、ローカルサーバーに接続する **ConnectionFactory** を設定し、**queue** という名前の宛先キューを定義します。

```
# 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
```

5.2. メッセージの送信

この例では、最初に JNDI **Context** を作成し、これを使用して **ConnectionFactory** および **Destination** を検索し、ファクトリーを使用して **Connection** を作成および起動し、**Session** を作成します。次に、**MessageProducer** が **Destination** に作成され、それを使用してメッセージが送信されます。その後、**Connection** が閉じられ、プログラムは終了します。

この **Sender** の例の実行可能なバリアントは、**<install-dir>/examples**ディレクトリー にあります。3 章*スタートガイド*.

例: メッセージの送信

```
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) JNDI **Context** を作成し、**ConnectionFactory** および **Destination** オブジェクトを検索します。設定は、前述 の **jndi.properties** ファイルから選択されます。

(2) ConnectionFactory および Destination オブジェクトは、ルックアップ名を使用して JNDI コンテキストから取得されます。

(3) ファクトリーは **Connection** の作成に使用され、次に **ExceptionListener** が登録されて開始します。接続の作成時に指定される認証情報は、通常適切な外部設定ソースから取得され、アプリケーション自体とは別のままとなり、個別に更新できます。

(4) トランザクション以外の自動承認 **Session** が **Connection** に作成されます。

(5) **MessageProducer** は、メッセージを **Destination** に送信するために作成されます。

(6) **TextMessage** は、指定の内容で作成されます。

7 **TextMessage** が送信されます。非永続的な送信は、デフォルトの優先度で、有効期限はありません。

8 **Connection** は閉じられます。**Session** および **MessageProducer** は暗黙的に閉じられます。

これは単なる例であることに注意してください。実際のアプリケーションは、通常有効期限の長い MessageProducer を使用し、時間の経過とともに多数のメッセージを送信します。通常、メッセージごとに **Connection**、**Session**、および **MessageProducer** を開くことは効率的ではありません。

5.3. メッセージの受信

この例では、JNDI コンテキストを作成し、そのコンテキストを使用して **ConnectionFactory** および **Destination** を検索し、ファクトリーを使用して **Connection** を作成および起動し、**Session** を作成します。次に、**Destination** に **MessageConsumer** が作成され、それを使用してメッセージを受信し、その内容がコンソールに出力されます。その後、Connection が閉じられ、プログラムは終了します。送信例 と同じ JNDI 設定が使用されます。

この **Receiver** の例の実行可能なバリアントは、以前に *3章スタートガイド* に記載されている Hello World の例とともに、クライアントディストリビューションのサンプルディレクトリーに含まれます。

例: メッセージの受信

```
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. JNDI **Context** を作成し、**ConnectionFactory** および **Destination** オブジェクトを検索します。設定は、前述 の **jndi.properties** ファイルから選択されます。

2. **ConnectionFactory** および **Destination** オブジェクトは、ルックアップ名を使用して JNDI **Context** から取得されます。

3. ファクトリーは **Connection** の作成に使用され、次に **ExceptionListener** が登録されて開始します。接続の作成時に指定される認証情報は、通常適切な外部設定ソースから取得され、アプリケーション自体とは別のままとなり、個別に更新できます。

4. トランザクション以外の自動承認 **Session** が **Connection** に作成されます。

5. **MessageConsumer** は、**Destination** からメッセージを受信するために作成されます。

6. メッセージを受信する呼び出しは、5 秒のタイムアウトで行われます。

7. 結果は確認され、メッセージが受信された場合はその内容が出力されます。または、メッセージが受信されなかったことが通知されます。結果は、**Sender** が送信された内容であるため、**TextMessage** に明示的にキャストされます。

8. **Connection** は閉じられます。**Session** および **MessageConsumer** は暗黙的に閉じられます。

これは単なる例であることに注意してください。実際のアプリケーションは、通常有効期限の長い **MessageConsumer** を使用し、時間の経過とともに多くのメッセージを受信します。通常、各メッセージの **Connection**、**Session**、および **MessageConsumer** を開くと効率的ではありません。

第6章 セキュリティー

AMQ JMS には、アプリケーションのニーズに応じて活用できるセキュリティー関連の設定オプションが複数あります。

アプリケーション内で **Connection** を作成する際に、ユーザー名とパスワードなどの基本的なユーザーの認証情報を **ConnectionFactory** に直接渡す必要があります。ただし、no-argument ファクトリーメソッドを使用している場合は、接続 URI でユーザーの認証情報を指定することもできます。詳細は、「JMS オプション」 セクションを参照してください。

もう 1 つの一般的なセキュリティー対策として、SSL/TLS が使用されます。クライアントは、**amqps** URI スキームが接続 URI で指定され、動作を設定するさまざまなオプションとともに、SSL/TLS トランスポートを介してサーバーに 接続 します。詳細は、「SSL/TLS オプション」 セクションを参照してください。

以前の項目と並べると、クライアントが、対応するすべてのものから選択するのではなく、サーバーで提供できる特定の SASL メカニズムのみを使用できるようにクライアントを制限することが望ましい場合があります。詳細は、「AMQP オプション」 セクションを参照してください。

受信した **ObjectMessage** で **getObject()** を呼び出すアプリケーションはデシリアライズ中に作成された型を制限する必要がある場合があります。AMQP タイプシステムを使用して設定されたメッセージ本文は **ObjectInputStream** メカニズムを使用しないため、この予防措置は必要ありません。詳細は、「デシリアライズポリシーオプション」 セクションを参照してください。

6.1. KERBEROS を使用した認証

クライアントは、適切に設定されたサーバーで使用される場合に Kerberos を使用して認証するように設定できます。Kerberos を有効にするには、以下の手順に従います。

1. **amqp.saslMechanisms** URI オプションを使用して、SASL 認証に **GSSAPI** メカニズムを使用するようにクライアントを設定します。

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

2. **java.security.auth.login.config** システムプロパティーを、Kerberos **LoginModule** の適切な設定が含まれる JAAS ログイン設定ファイルのパスに設定します。

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

ログイン設定ファイルは以下の例のようになります。

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

使用される正確な設定は、接続に対してクレデンシャルを確立する方法と、使用中の特定の **LoginModule** によって異なります。Oracle **Krb5LoginModule** の詳細は、Oracle **Krb5LoginModule** class reference を参照してください。IBM Java 8 **Krb5LoginModule** の詳細は、IBM **Krb5LoginModule** class reference を参照してください。

LoginModule を設定して、プリンシパルの指定や既存のチケットキャッシュまたはキータブを使用するかどうかなど、Kerberos プロセスに使用する認証情報を確立できます。しかし、**LoginModule** 設定

が必要なすべてのクレデンシャルを確立する手段を提供しない場合、**ConnectionFactory** を使用して **Connection** を作成する場合、クライアント **Connection** オブジェクトからユーザー名とパスワードの値が渡されます。

Kerberos は認証の目的でのみサポートされることに注意してください。暗号化には SSL/TLS 接続を使用します。

以下の接続 URI オプションを使用して、Kerberos 認証プロセスに影響を与えることができます。

sasl.options.configScope

認証に使用されるログイン設定エントリーの名前。デフォルトは **amqp-jms-client** です。

sasl.options.protocol

GSSAPI SASL プロセス中に使用されるプロトコル値。デフォルトは **amqp** です。

sasl.options.serverName

GSSAPI SASL プロセス中に使用される **serverName** 値。デフォルトは、接続 URI からのサーバーのホスト名です。

これまでで説明した **amqp.** オプションおよび **transport.** オプションと同様に、これらのオプションはホストごとに指定するか、フェイルオーバー URI のすべてのホストネストされたオプションとして指定する必要があります。

6.2. OPENSSL サポートの有効化

SSL/TLS 接続は、パフォーマンスを向上させるためにネイティブの OpenSSL 実装を使用するように設定できます。OpenSSL を使用するには、**transport.useOpenSSL** オプションを有効にし、OpenSSL サポートライブラリーをクラスパスで利用できるようにする必要があります。

Red Hat Enterprise Linux でインストールされた OpenSSL ライブラリーを使用するには、**openssl** パッケージおよび **apr** RPM パッケージをインストールし、以下の依存関係を POM ファイルに追加します。

例: ネイティブ OpenSSL サポートの追加

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-tcnative</artifactId>
  <version>2.0.29.Final-redhat-00001</version>
</dependency>
```

OpenSSL ライブラリーの実装の一覧 は、Netty プロジェクトから入手できます。

第7章 メッセージ配信

7.1. 承認されていない配信の処理

メッセージングシステムは、メッセージ確認を使用して、メッセージの送信ゴールが完全に行われるかどうかを追跡します。

メッセージが送信されると、メッセージが送信されてから確認応答するまでの期間が発生します (メッセージは in flight (インフライト) です。その間ネットワーク接続が失われた場合、メッセージ配信のステータスは不明となり、配信が完了するまでアプリケーションコードで特別な処理が必要になる場合があります。

以下のセクションでは、接続に失敗した場合にメッセージ配信の条件を説明します。

承認されていない配信とトランザクション以外のプロデューサー
メッセージが進行中の場合は、送信タイムアウトが設定されておらず、経過していないと、再接続後に再度送信されます。

ユーザーアクションは不要です。

トランザクションがコミットされていないトランザクションとのトランザクションプロデューサー
メッセージが進行中の場合は、再接続後に再度送信されます。新しいトランザクションで送信が最初に送信された場合は、再接続後に通常通りに送信が続行されます。トランザクションに以前の送信がある場合、トランザクションは失敗とみなされ、後続のコミット操作によって **TransactionRolledBackException** が出力されます。

配信を図るには、失敗したトランザクションに属するメッセージを再送信する必要があります。

保留中のコミットとトランザクションプロデューサー
コミットがフライトの場合、トランザクションは失敗とみなされ、後続のコミット操作によって **TransactionRolledBackException** が出力されます。

配信を図るには、失敗したトランザクションに属するメッセージを再送信する必要があります。

承認されていない配信のある非トランザクションコンシューマー
メッセージが受信してもまだ確認応答されない場合、メッセージを承認するとエラーは生成されませんが、クライアントによるアクションはありません。

受信したメッセージは確認されていないため、プロデューサーは再送信する可能性があります。重複を回避するために、ユーザーはメッセージ ID で重複メッセージを除外する必要があります。

コミットされていないトランザクションを使用したトランザクションコンシューマー
アクティブなトランザクションがまだコミットされていない場合は、失敗とみなされ、保留中の承認はドロップされます。後続のコミット操作によって **TransactionRolledBackException** が出力されます。

プロデューサーは、トランザクションに属するメッセージを再送信する可能性があります。重複を回避するために、ユーザーはメッセージ ID で重複メッセージを除外する必要があります。

保留中のコミットのあるトランザクションコンシューマー
コミットがフライトの場合、トランザクションは失敗とみなされます。後続のコミット操作によって **TransactionRolledBackException** が出力されます。

プロデューサーは、トランザクションに属するメッセージを再送信する可能性があります。重複を回避するために、ユーザーはメッセージ ID で重複メッセージを除外する必要があります。

7.2. 拡張セッション承認モード

クライアントは、JMS 仕様で定義されたもの以外の追加のセッション確認モードをサポートします。

個別確認応答
このモードでは、セッションが **CLIENT_ACKNOWLEDGE** モードの場合に使用される **Message.acknowledge()** メソッドを使用して、メッセージを個別に承認する必要があります。**CLIENT_ACKNOWLEDGE** モードとは異なり、ターゲットメッセージのみが確認されます。それ以外の配信されたメッセージはすべて承認されていないままになります。このモードを有効にするために使用される整数値は 101 です。

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

確認なし
このモードでは、クライアントにディスパッチされる前にサーバーでメッセージを受け入れ、承認はクライアントによって実行されません。クライアントは、このモードをアクティブにするために 2 つの整数値をサポートします (100 および 257)。

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

第8章 ロギングとトレース

クライアントは SLF4J API を使用し、ユーザーがニーズに基づいて特定のロギング実装を選択できるようにします。たとえば、ユーザーは **slf4j-log4j** バインディングを提供して Log4J 実装を選択できます。SLF4J の詳細は、その Web サイト を参照してください。

クライアントは **org.apache.qpid.jms** 階層内で存在する **Logger** 名を使用します。これを使用して、ニーズに合わせてロギング実装を設定することができます。

8.1. プロトコルロギングの有効化

デバッグ時には、Qpid Proton AMQP 1.0 ライブラリーから追加のプロトコルトレースロギングを有効にすると便利です。これを行うには、以下の 2 つの方法があります。

- 環境変数 (Java システムプロパティーではない) **PN_TRACE_FRM** を **1** に設定します。これにより、Proton はコンソールにフレームログを出力します。
- **amqp.traceFrames=true** オプションを 接続 URI に追加し、**org.apache.qpid.jms.provider.amqp.FRAMES** ロガーをログレベル **TRACE** に設定します。これにより、Proton にプロトコルトレーサーが追加され、ログに出力が含まれます。

入力および出力バイトの低レベルのトレースを出力するようにクライアントを設定することもできます。これを有効にするには、接続 URI に **transport.traceBytes=true** オプションを追加し、**org.apache.qpid.jms.transports.netty.NettyTcpTransport** ロガーをログレベル **DEBUG** に設定します。

8.2. 分散トレースの有効化

クライアントは、OpenTracing 標準の Jaeger 実装に基づいて分散トレーシングを提供します。アプリケーションでトレースを有効にするには、以下の手順に従います。

手順

1. Jaeger クライアントの依存関係を POM ファイルに追加します。

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

   ${jaeger-version} は 1.0.0 以降である必要があります。

2. 接続 URI に **jms.tracing** オプションを追加します。この値は **opentracing** に設定します。

   例: トレースが有効になっている接続 URI

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

3. グローバルトレーサーを登録します。

   例: グローバルトレーサーの登録

```
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. トレーシングのための環境を設定します。

   例: トレーシング設定

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

   ここで示された設定はデモ目的で使用されます。Jaeger 設定についての詳細は、環境経由の設定 および Jaeger サンプリング を参照してください。

アプリケーションをキャプチャーするトレースを表示するには、Jaeger Getting Started を使用して Jaeger インフラストラクチャーおよびコンソールを実行します。

第9章 相互運用性

本章では、AMQ JMS を他の AMQ コンポーネントと組み合わせて使用する方法を説明します。AMQ コンポーネントの互換性の概要は、製品の概要 を参照してください。

9.1. 他の AMQP クライアントとの相互運用

AMQP メッセージ は AMQP タイプシステム を使用して設定されます。この共通の形式を持つことは、異なる言語の AMQP クライアントが相互運用できる理由の 1 つです。本セクションでは、クライアントと他の AMQP クライアントの使用を支援するために、使用されるさまざまな JMS メッセージタイプに関連して、クライアントによって送受信された AMQP ペイロードに関する動作を文書化します。

9.1.1. メッセージの送信

このセクションでは、さまざまな JMS メッセージタイプの使用時にクライアントが送信する各種ペイロードを文書化するため、他のクライアントを使用してそれらを受信するのを支援する方法を説明します。

9.1.1.1. メッセージタイプ

JMS メッセージタイプ	提出された AMQP メッセージの説明
TextMessage	TextMessage は、本文テキストの utf8 でエンコードされた 文字列 を含む amqp-value ボディーセクションを使用して送信されます。本文テキストが設定されていない場合は null になります。"x-opt-jms-msg-type" の 記号 キーを持つメッセージアノテーションは 5 の バイト 値に設定されます。
BytesMessage	BytesMessage は、BytesMessage ボディーからの raw バイトを含む data ボディーセクションを使用して送信され、properties セクションの **content-type** フィールドは 記号 値 **"application/octet-stream"** に設定されます。"x-opt-jms-msg-type" の 記号 キーを持つメッセージアノテーションは、3 の バイト 値に設定されます。
MapMessage	MapMessage ボディーは、単一の map 値が含まれる amqp-value ボディーセクションを使用して送信されます。MapMessage ボディーの byte[] 値はマップの バイナリー エントリーとしてエンコードされます。"x-opt-jms-msg-type" の 記号 キーを持つメッセージアノテーションは、2 の バイト 値に設定されます。
StreamMessage	StreamMessage は、StreamMessage ボディーのエントリーが含まれる amqp-sequence ボディーセクションを使用して送信されます。StreamMessage ボディーの byte[] エントリーは、シーケンスの バイナリー エントリーとしてエンコードされます。"x-opt-jms-msg-type" の 記号 キーを持つメッセージアノテーションは、4 の バイト 値に設定されます。
ObjectMessage	ObjectMessage は、ObjectOutputStream を使用して ObjectMessage ボディーをシリアライズするバイトを含む data ボディーセクションを使用して送信されます。properties セクションの **content-type** フィールドは、シンボル の値 **"application/x-java-serialized-object"** に設定されます。"x-opt-jms-msg-type" の 記号 キーを持つメッセージアノテーションは、1 の バイト 値に設定されます。

JMS メッセージタイプ	提出された AMQP メッセージの説明
メッセージ	プレーンな JMS メッセージにはボディーがなく、null を含む amqp-value ボディーセクションとして送信されます。"x-opt-jms-msg-type" の 記号 キーを持つメッセージアノテーションは、0 の バイト 値に設定されます。

9.1.1.2. メッセージのプロパティー

JMS メッセージでは、さまざまな Java タイプのアプリケーションプロパティーの設定がサポートされます。このセクションでは、これらのプロパティータイプと、送信されたメッセージの application-properties セクションの AMQP 型の値とのマッピングを説明します。JMS と AMQP はいずれもプロパティー名に文字列キーを使用します。

JMS プロパティータイプ	AMQP アプリケーションプロパティーのタイプ
boolean	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
String	文字列 または null

9.1.2. メッセージの受信

このセクションでは、クライアントが受信した各種ペイロードをさまざまな JMS メッセージタイプにマッピングし、他のクライアントを使用してメッセージを受信して JMS クライアントによる受信用にメッセージを送信する方法を説明します。

9.1.2.1. メッセージタイプ

受信した AMQP メッセージに "x-opt-jms-msg-type" message-annotation が存在する場合は、その値を使用して、以下の表で説明するマッピングに従って、そのメッセージタイプを表すのに使用する JMS メッセージタイプを判断します。これは、JMS クライアントによって送信される メッセージについて説明したマッピングのリバースプロセスを反映します。

AMQP "x-opt-jms-msg-type" message-annotation 値 (タイプ)	JMS メッセージタイプ
0	メッセージ

AMQP "x-opt-jms-msg-type" message-annotation 値 (タイプ)	JMS メッセージタイプ
1 (バイト)	ObjectMessage
2 (バイト)	MapMessage
3 (バイト)	BytesMessage
4 (バイト)	StreamMessage
5 (バイト)	TextMessage

"x-opt-jms-msg-type" message-annotation が存在しない場合、以下の表でメッセージが JMS メッセージタイプにマップされます。StreamMessage タイプおよび MapMessage タイプは、アノテーション付きメッセージのみに割り当てられることに注意してください。

"x-opt-jms-msg-type" アノテーションなしの Received AMQP メッセージの説明	JMS メッセージタイプ
• 文字列 または null を含む amqp-value ボディーセクション。 • プロパティー セクションの **content-type** フィールドが **"text/plain"**、**"application/xml"**、または **"application/json"** などの一般的なテキストメディアタイプを表す 記号 の値に設定される data ボディーセクション。	TextMessage
• バイナリー を含む amqp-value ボディーセクション。 • プロパティー セクションの **content-type** フィールドが設定されていないか、記号 値 **"application/octet-stream"** を設定するか、別のメッセージタイプと関連しない値に設定されている data ボディーセクション。	BytesMessage
• プロパティー セクションの **content-type** フィールドが 記号 値 **"application/x-java-serialized-object"** に設定された data ボディーセクション。 • 上記の説明のない値が含まれる amqp-value ボディーセクション。 • amqp-sequence ボディーセクション。これは ObjectMessage 内で List として表されます。	ObjectMessage

9.1.2.2. メッセージのプロパティー

このセクションでは、受信した AMQP メッセージの application-properties セクションの値と、JMS メッセージで使用される Java 型とのマッピングを説明します。

AMQP アプリケーションプロパティーのタイプ	JMS プロパティータイプ
boolean	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
string	String
null	String

9.2. AMQ BROKER への接続

AMQ Broker は AMQP 1.0 クライアントと相互運用するために設計されています。以下を確認して、ブローカーが AMQP メッセージング用に設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- AMQ Broker AMQP アクセプターが有効になっている。デフォルトのアクセプター設定 を参照してください。
- 必要なアドレスがブローカーに設定されている。アドレス、キュー、およびトピック を参照してください。
- ブローカーはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。Broker Security を参照してください。

9.3. AMQ INTERCONNECT への接続

AMQ Interconnect は AMQP 1.0 クライアントであれば機能します。以下をチェックして、コンポーネントが正しく設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- ルーターはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。ネットワーク接続のセキュリティー保護 を参照してください。

付録A サブスクリプションの使用

AMQ は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

A.1. アカウントへのアクセス

手順

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

A.2. サブスクリプションのアクティベート

手順

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

A.3. リリースファイルのダウンロード

.zip、.tar.gz およびその他のリリースファイルにアクセスするには、カスタマーポータルを使用してダウンロードする関連ファイルを検索します。RPM パッケージまたは Red Hat Maven リポジトリーを使用している場合は、この手順は必要ありません。

手順

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **JBOSS INTEGRATION AND AUTOMATION** カテゴリーの **Red Hat AMQ** エントリーを見つけます。
3. 必要な AMQ 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

A.4. パッケージ用のシステムの登録

RPM パッケージを Red Hat Enterprise Linux にインストールするには、システムが登録されている必要があります。ダウンロードしたリリースファイルを使用している場合は、この手順は必要ありません。

手順

1. access.redhat.com に移動します。
2. **Registration Assistant** に移動します。

3. ご使用の OS バージョンを選択し、次のページに進みます。
4. システムの端末に一覧表示されたコマンドを使用して、登録を完了します。

詳細は、How to Register and Subscribe a System to the Red Hat Customer Portal を参照してください。

付録B RED HAT MAVEN リポジトリーの追加

このセクションでは、Red Hat が提供する Maven リポジトリーをソフトウェアで使用する方法を説明します。

B.1. オンラインリポジトリーの使用

Red Hat は、Maven ベースのプロジェクトで使用する中央の Maven リポジトリーを維持しています。詳細は、リポジトリーのウェルカムページ を参照してください。

Red Hat リポジトリーを使用するように Maven を設定する方法は 2 つあります。

- Maven 設定にリポジトリーを追加する
- POM ファイルにリポジトリーを追加する

Maven 設定へのリポジトリーの追加

この設定方法は、POM ファイルがリポジトリー設定をオーバーライドせず、含まれているプロファイルが有効になっている限り、ユーザーが所有するすべての Maven プロジェクトに適用されます。

手順

1. Maven の **settings.xml** ファイルを見つけます。これは通常、ユーザーのホームディレクトリーの **.m2** ディレクトリー内にあります。ファイルが存在しない場合は、テキストエディターを使用して作成します。
   Linux または UNIX の場合:

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

   Windows の場合:

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

2. 次の例のように、Red Hat リポジトリーを含む新しいプロファイルを **settings.xml** ファイルの **profiles** 要素に追加します。

   例: Red Hat リポジトリーを含む Maven settings.xml ファイル

   ```
   <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>
```

Maven 設定の詳細は、Maven 設定リファレンス を参照してください。

POM ファイルへのリポジトリーの追加
プロジェクトで直接リポジトリーを設定するには、次の例のように、POM ファイルの **repositories** 要素に新しいエントリーを追加します。

例: Red Hat リポジトリーを含む Maven pom.xml ファイル

```
<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>
```

POM ファイル設定の詳細は、Maven POM リファレンス を参照してください。

B.2. ローカルリポジトリーの使用

Red Hat は、そのコンポーネントの一部にファイルベースの Maven リポジトリーを提供します。これらは、ローカルファイルシステムに抽出できるダウンロード可能なアーカイブとして提供されます。

ローカルに抽出されたリポジトリーを使用するように Maven を設定するには、Maven 設定または POM ファイルに次の XML を適用します。

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

${repository-url} は、抽出されたリポジトリーのローカルファイルシステムパスを含むファイル URL である必要があります。

表B.1 ローカル Maven リポジトリーの URL の例

オペレーティングシステム	ファイルシステムパス	URL
Linux または UNIX	**/home/alice/maven-repository**	**file:/home/alice/maven-repository**
Windows	**C:\repos\red-hat**	**file:C:\repos\red-hat**

付録C 例で AMQ ブローカーの使用

AMQ JMS の例では、名前が **queue** というキューが含まれる実行中のメッセージブローカーが必要です。以下の手順に従って、ブローカーをインストールして起動し、キューを定義します。

C.1. ブローカーのインストール

Getting Started with AMQ Brokerの手順に従って、ブローカーをインストール して、ブローカーインスタンスを作成 します。匿名アクセスを有効にします。

以下の手順では、ブローカーインスタンスの場所を **<broker-instance-dir>** と呼びます。

C.2. ブローカーの起動

手順

1. **artemis run** コマンドを使用してブローカーを起動します。

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

2. 起動時にログに記録された重大なエラーがないか、コンソールの出力を確認してください。ブローカーでは、準備が整うと **Server is now live** とログが記録されます。

```
$ 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. キューの作成

新しいターミナルで、**artemis queue** コマンドを使用して **queue** という名前のキューを作成します。

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

プロンプトで質問に Yes または No で回答するように求められます。すべての質問に **N** (いいえ) と回答します。

キューが作成されると、ブローカーはサンプルプログラムで使用できるようになります。

C.4. ブローカーの停止

サンプルの実行が終了したら、**artemis stop** コマンドを使用してブローカーを停止します。

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

改訂日時：2023-01-28 12:23:27 +1000