Red Hat AMQ 2021.Q1

AMQ Core Protocol JMS クライアントの使用

AMQ Clients 2.9 向け

Last Updated: 2023-02-27

Red Hat AMQ 2021.Q1 AMQ Core Protocol JMS クライアントの使用

AMQ Clients 2.9 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

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社 の CTO、Chris Wright のメッセージを参照してください。

第1章 概要

AMQ Core Protocol JMS は、Artemis Core Protocol 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 Core Protocol JMS は AMQ Clients (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) に含まれています。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 2.9 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
- 自動再接続およびフェイルオーバー
- 分散トランザクション (XA)
- Pure-Java 実装

1.2. サポートされる標準およびプロトコル

AMQ Core Protocol 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
- 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
- IPv6 での最新の TCP

1.3. サポートされる構成

AMQ Core Protocol JMS は、以下に挙げる OS および言語のバージョンをサポートします。詳細は、「Red Hat AMQ 7 Supported Configurations」 を参照してください。

- 以下の JDK を備えた Red Hat Enterprise Linux 7 および 8:
  - OpenJDK 8 および 11
  - Oracle JDK 8
  - IBM JDK 8
- IBM JDK 8 と IBM AIX 7.1
- Microsoft Windows 10 Pro と Oracle JDK 8
- Microsoft Windows Server 2012 R2 および 2016 with Oracle JDK 8
- Oracle JDK 8 での Oracle Solaris 10 および 11

AMQ Core Protocol JMS は、最新バージョンの AMQ Broker と組み合わせてサポートされます。

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。
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 Core Protocol 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 インストールシステム

本章では、環境に AMQ Core Protocol 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 Core Protocol 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 Core Protocol 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.activemq</groupId>
  <artifactId>artemis-jms-client</artifactId>
  <version>2.16.0.redhat-00005</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 Core Protocol 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 Broker 7.9.0 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-broker-7.9.0-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 Broker 7.9.0**.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-broker-7.9.0.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-broker-7.9.0** 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 はじめに

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。
- **exampleQueue** 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. 最初のサンプルの実行

この例では、**exampleQueue** という名前のキューにコンシューマーおよびプロデューサーを作成します。テキストメッセージを送信してから受信し、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/features/standard/queue** ディレクトリーで以下のコマンドを実行し、Maven を使用してサンプルを構築します。

```
$ 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.activemq.artemis.jms.example.QueueExample
```

Windows の場合:

```
> java -cp "target\classes;target\dependency\*"
org.apache.activemq.artemis.jms.example.QueueExample
```

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.activemq.artemis.jms.example.QueueExample
Sent message: This is a text message
Received message: This is a text message
```

この例のソースコードは **<install-dir>/examples/features/standard/queue/src** ディレクトリーにあります。追加の例は、**<install-dir>/examples/features/standard** ディレクトリーにあります。

第4章 設定

本章では、AMQ Core Protocol 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。

4.1. JNDI 初期コンテキストの設定

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 Core Protocol JMS は、**org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory** 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。これには、**jndi.properties** ファイルの使用、システムプロパティーの使用、または初期コンテキスト API の使用の 3 つの方法があります。

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.activemq.artemis.jndi.ActiveMQInitialContextFactory
```

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.activemq.artemis.jndi.ActiveMQInitialContextFactory
...
```

初期コンテキスト API の使用

JNDI 初期コンテキスト API を使用してプロパティーをプログラム的に設定します。

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.activemq.artemis.jndi.ActiveMQInitialContextFactory");

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

同じ API を使用して、接続ファクトリー、キュー、およびトピックの JNDI プロパティーを設定できます。

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.<lookup-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 = tcp://example.net:61616?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

コネクションは接続 URI を使用して設定されます。接続 URI は、クエリーパラメーターとして設定されるリモートホスト、ポート、および設定オプションのセットを指定します。利用可能なオプションの詳細は、5章*設定オプション* を参照してください。

接続 URI 形式

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

たとえば、以下はポート **61616** 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

```
tcp://example.net:61616?clientID=backend
```

AMQ Core Protocol JMS は、**tcp** の他に、**vm**、**udp**、および **jgroups** スキームもサポートします。これらは代替トランスポートを表し、対応するアクセプター設定がブローカーにあります。

フェイルオーバー URI

URI には複数のターゲット接続 URI を含めることができます。あるターゲットへの最初の接続に失敗すると、別のターゲットへの接続は試行されます。URI の形式は、以下のとおりです。

フェイルオーバー URI 形式

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

括弧外のオプションは、すべての接続 URI に適用されます。

4.4. 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.<lookup-name> = <queue-name>
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");
```

第5章 設定オプション

本章では、AMQ Core Protocol JMS で利用可能な設定オプションについて説明します。

JMS 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設定されます。詳細は、「「接続 URI」」を参照してください。

5.1. 一般的なオプション

user

クライアントが接続を認証するために使用するユーザー名。

password

クライアントが接続を認証するために使用するパスワード。

clientID

クライアントが接続に適用するクライアント ID。

groupID

クライアントが生成されたすべてのメッセージに適用するグループ ID。

autoGroup

有効な場合は、ランダムなグループ ID を生成し、生成されたすべてのメッセージに適用します。

preAcknowledge

有効にすると、メッセージが送信されると同時に、配信が完了する前にメッセージを承認します。最大 1 回の配信これはデフォルトでは無効にされます。

blockOnDurableSend

有効な場合、非トランザクションの永続メッセージを送信すると、リモートピアが受信を確認するまでブロックします。これは、デフォルトで有効になっています。

blockOnNonDurableSend

有効な場合、非トランザクションの非永続メッセージを送信すると、リモートピアが受信を確認するまでブロックします。これはデフォルトでは無効にされます。

blockOnAcknowledge

有効な場合、トランザクション以外の受信したメッセージを承認すると、リモートピアが確認を確認するまでブロックします。これはデフォルトでは無効にされます。

callTimeout

ブロック呼び出しが完了するまで待機する時間（ミリ秒単位）。デフォルトは 30000 (30 秒) です。

callFailoverTimeout

クライアントがフェイルオーバー中であるときに、ブロッキングコールを開始するまでの待機時間（ミリ秒単位）。デフォルトは 30000 (30 秒) です。

ackBatchSize

確認がブローカーに送信される前にクライアントが受信および確認できるバイト数。デフォルトは 1048576 (1 MiB) です。

dupsOKBatchSize

DUPS_OK_ACKNOWLEDGE 確認モードを使用する場合、確認バッチのバイト単位のサイズ。デフォルトは 1048576 (1 MiB) です。

transactionBatchSize

トランザクションでメッセージを受信するとき、確認バッチのバイト単位のサイズ。デフォルトは 1048576 (1 MiB) です。

cacheDestinations

有効にすると、宛先検索をキャッシュします。これはデフォルトでは無効にされます。

5.2. TCP オプション

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。

tcpSendBufferSize

送信バッファーサイズ (バイト単位)。デフォルトは 32768 (32 KiB) です。

tcpReceiveBufferSize

受信バッファーサイズ (バイト単位)。デフォルトは 32768 (32 KiB) です。

writeBufferLowWaterMark

この制限 （バイト単位） では、書き込みバッファーが書き込み可能になります。デフォルトは 32768 (32 KiB) です。

writeBufferHighWaterMark

この制限 （バイト単位） では、書き込みバッファーが書き込み不可になります。デフォルトは 131072 (128 KiB) です。

5.3. SSL/TLS オプション

sslEnabled

有効にすると、SSL/TLS を使用して接続を認証および暗号化します。これはデフォルトでは無効にされます。

keyStorePath

SSL/TLS キーストアへのパス。キーストアは相互 SSL/TLS 認証に必要です。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。

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。

trustStorePath

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。

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。

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。

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は無効にされます。

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。

enabledProtocols

有効にする SSL/TLS プロトコルのコンマ区切りリスト。未設定の場合は、JVM デフォルトプロトコルが使用されます。

5.4. フェイルオーバーオプション

initialConnectAttempts

最初に接続に成功する前に、クライアントがブローカートポロジーを検出するまでに許可される再接続試行回数。デフォルトは 0 で、試行は 1 回のみが許可されます。

failoverOnInitialConnection

有効な場合は、最初の接続に失敗した場合にバックアップサーバーへの接続を試みます。これはデフォルトでは無効にされます。

reconnnectAttempts

接続が失敗したと報告するまでに許可される再接続試行回数。デフォルトは -1 で、無制限を意味します。

retryInterval

再接続試行の間隔 （ミリ秒単位）。デフォルトは 2000 (2 秒) です。

retryIntervalMultiplier

再試行間隔を大きくするために使用される乗数。デフォルトは 1.0 で、等間隔を意味します。

maxRetryInterval

再接続試行の最大間隔 （ミリ秒単位）。デフォルトは 2000 (2 秒) です。

ha

有効な場合は、HA ブローカーのトポロジーの変更を追跡します。URI からのホストおよびポートは、初期接続にのみ使用されます。クライアントは、最初の接続後に現在のフェイルオーバーエンドポイントとトポロジーの変更から生じた更新を受け取ります。これはデフォルトでは無効にされます。

connectionTTL

サーバーが ping パケット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）。デフォルトは 60000 （1 分）です。-1 はタイムアウトを無効にします。

confirmationWindowSize

コマンド再生バッファーのサイズ（バイト単位）。これは、再接続時に自動セッションの再アタッチに使用されます。デフォルトは -1 で、自動的に再アタッチされません。

clientFailureCheckPeriod

期限切れの接続を定期的にチェックする間隔 (ミリ秒単位)。デフォルトは 30000 （30 秒）です。-1 はチェックを無効にします。

5.5. フロー制御オプション

詳細は、「8章 *フロー制御*」を参照してください。

consumerWindowSize

コンシューマーごとのメッセージの事前フェッチバッファーのサイズ （バイト単位）。デフォルトは 1048576 (1 MiB) で、1 は制限がないことを意味します。0 は事前フェッチを無効にします。

consumerMaxRate

1 秒あたりに消費するメッセージの最大数。デフォルトは -1 で、無制限を意味します。

producerWindowSize

より多くのメッセージを生成するためにクレジットに要求されたサイズ（バイト単位）。これにより、1度にインフライトデータの合計量が制限されます。デフォルトは 1048576 (1 MiB) で、1 は制限がないことを意味します。

producerMaxRate

1 秒あたりに生成するメッセージの最大数。デフォルトは -1 で、無制限を意味します。

5.6. 負荷分散オプション

useTopologyForLoadBalancing

有効な場合は、接続負荷分散にクラスタートポロジーを使用します。これは、デフォルトで有効になっています。

connectionLoadBalancingPolicyClassName

接続負荷分散ポリシーのクラス名。デフォルトは **org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy** です。

5.7. 大型メッセージのオプション

クライアントは、プロパティー **minLargeMessageSize** の値を設定することで、大きなメッセージサポートを有効にすることができます。**minLargeMessageSize** を超えるメッセージは、大きなメッセージとみなされます。

minLargeMessageSize

メッセージが大きなメッセージとして扱われる最小サイズ（バイト単位）。デフォルトは 102400 (100 KiB) 日です。

compressLargeMessages

有効な場合は、**minLargeMessageSize** で定義されているように大きなメッセージを圧縮します。これはデフォルトでは無効にされます。

注記

大きなメッセージの圧縮サイズが **minLargeMessageSize** の値より小さい場合、メッセージは通常のメッセージとして送信されます。そのため、ブローカーの大型メッセージデータディレクトリーには書き込まれません。

5.8. スレッドオプション

useGlobalPools

有効にすると、すべての **ConnectionFactory** インスタンスにスレッドの 1 つのプールを使用します。それ以外の場合は、インスタンスごとに個別のプールを使用します。これは、デフォルトで有効になっています。

threadPoolMaxSize

概要スレッドプールの最大スレッド数。デフォルトは -1 で、無制限を意味します。

scheduledThreadPoolMaxSize

スケジュール済み操作のスレッドプールの最大スレッド数。デフォルトは 5 です。

第6章 ネットワーク接続

6.1. 自動フェイルオーバー

クライアントは、接続障害が発生した場合にスレーブブローカーに再接続できるように、すべてのマスターおよびスレーブブローカーに関する情報を受け取ることができます。その後、スレーブブローカーは、フェイルオーバー前に各接続に存在したセッションおよびコンシューマーを自動的に再作成します。この機能により、アプリケーションで手動で再接続ロジックをコーディングする必要がなくなります。

セッションがスレーブで再作成されると、送信または確認済みのメッセージを認識しません。フェイルオーバー時の in-flight の送信と確認応答も失われる可能性があります。ただし、透過的なフェイルオーバーがなくても、トランザクションの重複検出と再試行の組み合わせを使用することで、障害が発生した場合でも、**一度だけ** の配信を保証するのは簡単です。

クライアントは、設定可能な期間内にブローカーからパケットを受信しないと接続の障害を検出します。詳細は、「デッド接続の検出」 を参照してください。

マスターおよびスレーブについての情報を受信するようにクライアントを設定する方法は複数あります。1 つのオプションとして、特定のブローカーに接続し、クラスター内の他のブローカーに関する情報を受信するようにクライアントを設定します。詳細は、「静的検出の設定」 を参照してください。ただし、最も一般的な方法は、**ブローカー検出** を使用することです。ブローカー検出の設定方法の詳細は、「動的検出の設定」 を参照してください。

また、以下の例のように、ブローカーへの接続に使用される URI のクエリー文字列にパラメーターを追加して、クライアントを設定することもできます。

```
connectionFactory.ConnectionFactory=tcp://localhost:61616?ha=true&reconnectAttempts=3
```

手順

クエリー文字列を使用してフェイルオーバーを行うようにクライアントを設定するには、URI の以下のコンポーネントが適切に設定されていることを確認します。

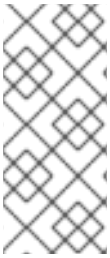
1. URI の **host:port** の部分は、バックアップで適切に設定されたマスターブローカーを参照する必要があります。このホストおよびポートは最初の接続にのみ使用されます。**host:port** の値は、ライブサーバーとバックアップサーバー間の実際の接続フェイルオーバーとは関係ありません。上記の例では、**localhost:61616** が **host:port** に使用されます。

2. （オプション） 複数のブローカーを可能な初期接続として使用するには、以下の例のように **host:port** エントリーをグループ化します。

   ```
   connectionFactory.ConnectionFactory=(tcp://host1:port,tcp://host2:port)?
   ha=true&reconnectAttempts=3
   ```

3. 名前/値のペア **ha=true** をクエリー文字列の一部として追加し、クライアントがクラスター内の各マスターおよびスレーブブローカーに関する情報を受け取るようにします。

4. 名前と値のペア **reconnectAttempts=n** を含めます。**n** は、0 より大きい整数です。このパラメーターは、クライアントがブローカーへの再接続を試行する回数を設定します。



注記

フェイルオーバーは **ha=true** と **reconnectAttempts** が 0 よりも大きい場合にのみ発生します。また、他のブローカーに関する情報を取得するために、クライアントはマスターブローカーへ最初の接続を確立する必要があります。最初の接続に失敗すると、クライアントは再試行して接続を確立できます。詳細は、「初期接続時のフェイルオーバー」 を参照してください。

6.1.1. 初期接続時のフェイルオーバー

クライアントは HA クラスターへの最初の接続が確立されるまですべてのブローカーに関する情報を受け取らないため、クライアントが接続 URI に含まれるブローカーにのみ接続できる期間があります。したがって、この最初の接続で障害が発生した場合、クライアントは他のマスターブローカーにフェイルオーバーできず、最初の接続の再確立のみを試すことができます。クライアントには、再接続試行回数を設定できます。試行がその回数行われると、例外がスローされます。

再接続試行回数の設定
以下の例は、AMQ Core Protocol JMS クライアントを使用して再接続試行回数を 3 に設定する方法を示しています。デフォルト値は 0 で、つまり、1 回のみ試行します。

手順

ServerLocator.setInitialConnectAttempts（） に値を渡して、再接続試行の数を設定します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setInitialConnectAttempts(3);
```

再接続試行回数のグローバル設定
または、ブローカーの設定内で再接続試行回数のグローバルな値を適用できます。最大値はすべてのクライアント接続に適用されます。

手順

以下の例のように、**<broker-instance-dir>/etc/broker.xml** を編集して **initial-connect-attempts** 設定要素を追加し、time-to-live の値を指定します。

```
<configuration>
 <core>
  ...
  <initial-connect-attempts>3</initial-connect-attempts> (1)
  ...
 </core>
</configuration>
```

(1) ブローカーに接続するすべてのクライアントは、最大 3 回の再接続の試行が許可されます。デフォルトは -1 で、クライアントは無制限の試行を許可します。

6.1.2. フェイルオーバー中のブロック呼び出しの処理

フェイルオーバーが発生し、クライアントが実行を継続するためにブローカーからの応答を待機している場合、新たに作成されたセッションには、進行中の呼び出しに関する情報がありません。初期呼び出しは、それ以外では、決して受け取ることのない応答を待機してハングする可能性があります。これを防ぐため、ブローカーは例外をスローすることで、フェイルオーバー時に進行中のブロッキング呼び出しのブロックを解除するように設計されています。クライアントコードはこれらの例外をキャッチし、必要に応じてすべての操作を再試行できます。

AMQ Core Protocol JMS クライアントを使用する場合、ブロックされていないメソッドが **commit（）** または **prepare（）** の呼び出しである場合、トランザクションは自動的にロールバックされ、ブローカーによって例外が発生します。

6.1.3. トランザクションによるフェイルオーバーの処理

AMQ Core Protocol JMS クライアントを使用すると、セッションがトランザクションであり、現在のトランザクションでメッセージがすでに送信または確認応答されている場合、ブローカーはフェイルオーバー中にメッセージまたはその確認応答が失われたかどうかがわかりません。そのため、トランザクションはロールバック用にのみマークされます。その後、コミットしようとすると **javax.jms.TransactionRolledBackException** がスローされます。

警告

このルールに関する注意点は、XA を使用する場合です。2 フェーズコミットが使用され、**prepare()** がすでに呼び出されている場合、ロールバックによって **HeuristicMixedException** が発生する可能性があります。このため、コミットによって **XAException.XA_RETRY** 例外が発生し、トランザクションマネージャーは後でコミットを再試行することを通知します。元のコミットが発生していない場合、それはまだ存在し、コミットされます。存在しない場合、コミットされたと見なされますが、トランザクションマネージャーが警告を記録する可能性があります。この例外の副次的な影響は、非永続的なメッセージが失われることです。このような損失を回避するために、XA を使用する際には常に永続メッセージを使用してください。**prepare()** が呼び出される前にサーバーにフラッシュされるため、これは確認応答の問題ではありません。

AMQ Core Protocol JMS クライアントコードは、例外をキャッチし、必要なクライアント側のロールバックを実行する必要があります。ただし、セッションはロールバックされているため、セッションをロールバックする必要はありません。ユーザーは、同じセッションに対してトランザクション操作を再試行できます。

コミット呼び出しの実行時にフェイルオーバーが発生すると、ブローカーは呼び出しをブロックしず、AMQ Core Protocol JMS クライアントが応答を無期限に待機しないようにします。この場合、障害が発生する前にトランザクションのコミットがライブサーバーで実際に処理されたかどうかをクライアントが判断するのは容易ではありません。

これを修正するには、AMQ Core Protocol JMS クライアントはトランザクションで重複検出を有効にし、呼び出しがブロック解除された後にトランザクション操作を再試行します。フェイルオーバー前にトランザクションが正常にマスターブローカーでコミットされた場合、重複検出により、再試行時にトランザクションに存在する永続メッセージがブローカー側で無視されるようにします。これにより、メッセージが複数回送信されなくなります。

セッションがトランザクションではない場合は、フェイルオーバー時にメッセージまたは確認応答が失われる可能性があります。トランザクション以外のセッションに **一度だけ** 配信する保証を提供する場合は、重複検出を有効にし、ブロック解除例外をキャッチします。

6.1.4. 接続失敗の通知

JMS は、接続障害の非同期に通知するための標準メカニズムである **java.jms.ExceptionListener** を提供します。

ExceptionListener または **SessionFailureListener** インスタンスは、接続が正常にフェイルオーバー、再接続、または再割り当てされたかどうかに関係なく、接続障害が発生した場合にブローカーによって常に呼び出されます。**SessionFailureListener** の **connectionFailed** で渡された **failedOver** フラグを調べると、再接続または再割り当てが発生したかどうかを確認できます。または、**javax.jms.JMSException** のエラーコードを確認できます。これは以下のいずれかになります。

表6.1 JMSException エラーコード

エラーコード	説明
FAILOVER	フェイルオーバーが発生し、ブローカーが正常に再割り当てまたは再接続しました。
DISCONNECT	フェイルオーバーは発生せず、切断されています。

6.2. アプリケーションレベルのフェイルオーバー

場合によっては、自動クライアントのフェイルオーバーは望ましくないかもしれませんが、障害ハンドラーで独自の再接続ロジックをコーディングしたい場合もあります。フェイルオーバーはユーザーアプリケーションレベルで処理されるため、これは、**アプリケーションレベル** のフェイルオーバーとして定義します。

JMS を使用する場合にアプリケーションレベルのフェイルオーバーを実装するには、JMS 接続で **ExceptionListener** クラスを設定します。**ExceptionListener** は、接続障害が検出された場合に JBoss EAP メッセージングによって呼び出されます。**ExceptionListener** では、古い JMS 接続を閉じる必要があります。JNDI から新しい接続ファクトリーインスタンスを検索し、新しい接続を作成することも可能です。

6.3. デッド接続の検出

ブローカーからデータを受信する限り、クライアントは接続がアライブ状態であると判断します。**client-failure-check-period** プロパティーの値を指定して、接続の失敗を確認するようにクライアントを設定します。ネットワーク接続のデフォルトのチェック期間は 30000 ミリ秒 (30 秒) で、仮想マシン間の接続のデフォルト値は -1 です。これは、データが受信されなかった場合、クライアントがその側から接続を失敗させないことを意味します。

通常、チェック期間はブローカーの接続の Time-to-live に使用される値よりもはるかに低い値に設定し、一時的な障害が発生した場合にクライアントが再接続できるようにします。

デッド接続を検出するためのチェック期間の設定
以下の例は、チェック期間を 10000 ミリ秒に設定する方法を示しています。

手順

- JNDI を使用している場合は、以下のように JNDI コンテキスト環境 (**jndi.properties**) 内のチェック期間を設定します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?
clientFailureCheckPeriod=10000
```

- JNDI を使用していない場合は、値を **ActiveMQConnectionFactory.setClientFailureCheckPeriod()** に渡してチェック期間を直接設定します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setClientFailureCheckPeriod(10000);
```

6.4. TIME-TO-LIVE の設定

デフォルトでは、クライアントは独自の接続に TTL (time-to-live) を設定できます。以下の例は、TTL を設定する方法を示しています。

手順

- JNDI を使用して接続ファクトリーをインスタンス化する場合は、**connectionTtl** パラメーターを使用して xml 設定に指定できます。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?connectionTtl=30000
```

- JNDI を使用していない場合は、接続 TTL は **ActiveMQConnectionFactory** インスタンスの **ConnectionTTL** 属性によって定義されます。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConnectionTTL(30000);
```

6.5. 接続を閉じる

クライアントアプリケーションは、デッド接続が発生しないように、終了する前に制御された方法でリソースを閉じる必要があります。Java では、**finally** ブロック内で接続を閉じることが推奨されます。

```
Connection jmsConnection = null;
try {
   ConnectionFactory jmsConnectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(...);
   jmsConnection = jmsConnectionFactory.createConnection();
   ...use the connection...
}
finally {
   if (jmsConnection != null) {
      jmsConnection.close();
   }
}
```

6.6. 動的検出の設定

接続の確立を試みる際に、AMQ Core Protocol JMS を設定してブローカーのリストを検出できます。

クライアント上で JNDI を使用して JMS 接続ファクトリーインスタンスを検索する場合は、これらのパラメーターを JNDI コンテキスト環境で指定できます。通常、パラメーターは **jndi.properties** という名前のファイルで定義されます。接続ファクトリーの URI のホストおよび部分は、ブローカーの

broker.xml 設定ファイル内の対応する **broadcast-group** の **group-address** と **group-port** と一致する必要があります。以下は、ブローカーの検出グループに接続するように設定されている **jndi.properties** ファイルの例です。

```
java.naming.factory.initial = ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=udp://231.7.7.7:9876
```

接続ファクトリーがクライアントアプリケーションによって JNDI からダウンロードされ、JMS 接続が作成される場合、これらの接続は、ブローカーの検出グループ設定に指定されたマルチキャストアドレスでリッスンすることで、ディスカバリーグループが維持するサーバーのリスト全体で負荷分散されます。

JNDI を使用する代わりに、JMS 接続ファクトリーの作成時に、ディスカバリーグループパラメーターを直接 Java コードに指定できます。以下のコードは、その方法の例になります。

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;

DiscoveryGroupConfiguration discoveryGroupConfiguration = new DiscoveryGroupConfiguration();
UDPBroadcastEndpointFactory udpBroadcastEndpointFactory = new
UDPBroadcastEndpointFactory();
udpBroadcastEndpointFactory.setGroupAddress(groupAddress).setGroupPort(groupPort);
discoveryGroupConfiguration.setBroadcastEndpointFactory(udpBroadcastEndpointFactory);

ConnectionFactory jmsConnectionFactory = ActiveMQJMSClient.createConnectionFactoryWithHA
   (discoveryGroupConfiguration, JMSFactoryType.CF);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();
Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

setter メソッド **setRefreshTimeout（）** を使用して、更新タイムアウトを **DiscoveryGroupConfiguration** に直接設定できます。デフォルト値は 10000 ミリ秒です。

最初の使用時では、接続ファクトリーは、最初の接続を作成する前に、作成してからこの期間待機します。デフォルトの待機時間は 10000 ミリ秒ですが、新しい値を **DiscoveryGroupConfiguration.setDis.coveryInitialWaitTimeout（）** に渡すことで変更できます。

6.7. 静的検出の設定

使用しているネットワークで UDP を使用できないことがあります。この場合は、使用可能なサーバーの初期リストで接続を設定できます。リストは、常に利用できることがわかっている 1 つのブローカーだけになるか、少なくとも 1 つのブローカーが利用可能であるブローカーのリストにすることができます。

これは、すべてのサーバーがホストされる場所を、認識しなければならないという意味ではありません。信頼できるサーバーを使用して接続するように、これらのサーバーを設定できます。接続後、接続の詳細はサーバーからクライアントに伝播されます。

クライアント上で JNDI を使用して JMS 接続ファクトリーインスタンスを検索する場合は、これらのパラメーターを JNDI コンテキスト環境で指定できます。通常、パラメーターは **jndi.properties** という名前のファイルで定義されます。以下は、動的検出を使用する代わりに、ブローカーの静的リストを提供する **jndi.properties** ファイルの例です。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=(tcp://myhost:61616,tcp://myhost2:61616)
```

上記の接続ファクトリーがクライアントによって使用される場合、接続は括弧 () 内で定義されるブローカーのリスト全体で負荷分散されます。

JMS 接続ファクトリーを直接インスタンス化する場合は、以下の例のように JMS 接続ファクトリーの作成時にコネクターリストを明示的に指定できます。

```
HashMap<String, Object> map = new HashMap<String, Object>();
map.put("host", "myhost");
map.put("port", "61616");
TransportConfiguration broker1 = new TransportConfiguration
    (NettyConnectorFactory.class.getName(), map);

HashMap<String, Object> map2 = new HashMap<String, Object>();
map2.put("host", "myhost2");
map2.put("port", "61617");
TransportConfiguration broker2 = new TransportConfiguration
    (NettyConnectorFactory.class.getName(), map2);

ActiveMQConnectionFactory cf = ActiveMQJMSClient.createConnectionFactoryWithHA
    (JMSFactoryType.CF, broker1, broker2);
```

6.8. ブローカーコネクターの設定

コネクターは、クライアントがブローカーに接続する方法を定義します。JMS 接続ファクトリーを使用してクライアントから設定できます。

```
Map<String, Object> connectionParams = new HashMap<String, Object>();

connectionParams.put(org.apache.activemq.artemis.core.remoting.impl.netty.TransportConstants.POR
T_PROP_NAME, 61617);

TransportConfiguration transportConfiguration =
    new TransportConfiguration(
    "org.apache.activemq.artemis.core.remoting.impl.netty.NettyConnectorFactory",
connectionParams);

ConnectionFactory connectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(JMSFactoryType.CF,
transportConfiguration);

Connection jmsConnection = connectionFactory.createConnection();
```

第7章 メッセージ配信

7.1. ストリーミングされた大きなメッセージへの書き込み

大きなメッセージに書き込むには、**BytesMessage.writeBytes（）** メソッドを使用します。以下の例では、ファイルからバイトを読み取り、メッセージに書き込みます。

例: ストリーミングされた大きなメッセージへの書き込み

```
BytesMessage message = session.createBytesMessage();
File inputFile = new File(inputFilePath);
InputStream inputStream = new FileInputStream(inputFile);

int numRead;
byte[] buffer = new byte[1024];

while ((numRead = inputStream.read(buffer, 0, buffer.length)) != -1) {
    message.writeBytes(buffer, 0, numRead);
}
```

7.2. ストリームされた大きなメッセージからの読み取り

大きなメッセージから読み取るには、**BytesMessage.readBytes（）** メソッドを使用します。以下の例では、メッセージからバイトを読み取り、ファイルに書き込みます。

例: ストリームされた大きなメッセージからの読み取り

```
BytesMessage message = (BytesMessage) consumer.receive();
File outputFile = new File(outputFilePath);
OutputStream outputStream = new FileOutputStream(outputFile);

int numRead;
byte buffer[] = new byte[1024];

for (int pos = 0; pos < message.getBodyLength(); pos += buffer.length) {
    numRead = message.readBytes(buffer);
    outputStream.write(buffer, 0, numRead);
}
```

7.3. メッセージグループの使用

メッセージグループは、以下の特性を持つメッセージセットです。

- メッセージグループのメッセージは、同じグループ ID を共有します。つまり、グループ識別子のプロパティーは同じです。JMS メッセージの場合、プロパティーは **JMSXGroupID** です。
- メッセージグループのメッセージは、キューに多くのコンシューマーがある場合でも、常に同じコンシューマーによって消費されます。元のコンシューマーが閉じられている場合は、別のコンシューマーがメッセージグループを受信するように選択されます。

メッセージグループは、プロパティーの特定値のすべてのメッセージを同じコンシューマーで順次に処理したい場合に便利です。たとえば、特定の株式購入の注文を同じコンシューマーで順次処理したい場合があります。これを行うには、コンシューマーのプールを作成し、株式名をメッセージプロパティー

の値として設定します。これにより、特定の株式のすべてのメッセージが常に同じコンシューマーによって処理されます。

グループ ID の設定

以下の例は、AMQ Core Protocol JMS でメッセージグループを使用する方法を示しています。

手順

- JNDI を使用して JMS クライアントの JMS 接続ファクトリーを確立する場合は、**groupID** パラメーターを追加して値を指定します。この接続ファクトリーを使用して送信されたすべてのメッセージでは、**JMSXGroupID** プロパティーが指定の値に設定されます。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?groupID=MyGroup
```

- JNDI を使用していない場合は、**setStringProperty()** メソッドを使用して **JMSXGroupID** プロパティーを設定 します。

```
Message message = new TextMessage();
message.setStringProperty("JMSXGroupID", "MyGroup");
producer.send(message);
```

関連情報

メッセージグループの設定および使用方法の作業例は、**<install-dir>/examples/features/standard** の **message-group** および **message-group2** を参照してください。

7.4. 複製メッセージの検出の使用

AMQ Broker には、受信する重複メッセージをフィルターする自動複製メッセージ検出が含まれるため、独自の複製検出ロジックをコーディングする必要はありません。

複製メッセージ検出を有効にするには、メッセージプロパティー **_AMQ_DUPL_ID** に一意の値を指定します。ブローカーがメッセージを受信すると、**_AMQ_DUPL_ID** の値があるかどうかを確認します。存在する場合、ブローカーはメモリーキャッシュを確認し、その値を持つメッセージをすでに受信したかどうかを確認します。同じ値を持つメッセージが見つかると、受信メッセージは無視されます。

トランザクションでメッセージを送信する場合は、トランザクション内のすべてのメッセージに **_AMQ_DUPL_ID** を設定する必要はなく、そのうちの 1 つにのみ設定する必要はありません。ブローカーがトランザクションのメッセージに対して重複メッセージを検出すると、トランザクション全体を無視します。

重複 ID メッセージプロパティーの設定

以下の例は、AMQ Core Protocol JMS を使用して重複検出プロパティーを設定する方法を示しています。便宜上、クライアントは重複 ID プロパティーの名前**_AMQ_DUPL_ID** の一貫した **org.apache.activemq.artemis.api.core.Message.HDR_DUPLICATE_DETECTION_ID** の値を使用することに注意してください。

手順

_AMQ_DUPL_ID の値を一意の文字列値に設定します。

```
Message jmsMessage = session.createMessage();
String myUniqueID = "This is my unique id";
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
```

7.5. メッセージインターセプターの使用

AMQ Core Protocol JMS では、クライアントに出入りするパケットを傍受して、パケットの監査やメッセージのフィルターを実行できます。インターセプターは、傍受するパケットを変更できます。これによりインターセプターは強力になりますが、注意して使用する必要があります。

インターセプターは、**boolean** 値を返す **intercept()** メソッドを実装する必要があります。戻り値が **true** の場合は、メッセージパケットが続行します。戻り値が **false** の場合は、プロセスは中止され、他のインターセプターは呼び出されず、メッセージパケットはこれ以上処理されません。

メッセージインターセプションは、送信パケットがブロッキング送信モードで送信される場合を除き、メインのクライアントコードに対して透過的に実行されます。ブロックが有効な状態で送信パケットが送信され、そのパケットが **false** を返すインターセプターに遭遇すると、 ActiveMQException が呼び出し元にスローされます。スローされた例外にはインターセプターの名前が含まれます。

インターセプターは、**org.apache.artemis.activemq.api.core.Interceptor** インターフェイスを実装する必要があります。クライアントインターセプタークラスとその依存関係を適切にインスタンス化および呼び出すには、クライアントの Java クラスに追加する必要があります。

```
package com.example;

import org.apache.artemis.activemq.api.core.Interceptor;
import org.apache.activemq.artemis.core.protocol.core.Packet;
import org.apache.activemq.artemis.spi.core.protocol.RemotingConnection;

public class MyInterceptor implements Interceptor {
   private final int ACCEPTABLE_SIZE = 1024;

   @Override
   boolean intercept(Packet packet, RemotingConnection connection) throws ActiveMQException {
      int size = packet.getPacketSize();
      if (size <= ACCEPTABLE_SIZE) {
         System.out.println("This Packet has an acceptable size.");
         return true;
      }
      return false;
   }
}
```

第8章 フロー制御

フロー制御は、プロデューサーとコンシューマー間のデータのフローを制限することで、プロデューサーとコンシューマーの超過を防ぎます。AMQ Core Protocol JMS を使用すると、コンシューマーとプロデューサーの両方のフロー制御を設定できます。

コンシューマーフロー制御
コンシューマーフロー制御は、クライアントがブローカーからメッセージを消費する際に、ブローカーとクライアント間のデータフローを制御します。AMQ Broker クライアントは、デフォルトでメッセージをバッファーしてからコンシューマーに配信します。バッファーがない場合、クライアントはまず、消費する前にブローカーから各メッセージを要求する必要があります。このタイプの「ラウンドトリップ」通信はコストがかかります。メモリー不足の問題によりコンシューマーがメッセージをすばやく処理できず、バッファーが受信メッセージでオーバーフローを開始するため、クライアント側のデータのフローを制限することが重要になります。

プロデューサーフロー制御
コンシューマーウィンドウベースのフロー制御と同様に、クライアントはプロデューサーからブローカーに送信されるデータ量をブローカーに制限し、ブローカーが大量のデータで過負荷にならないようにすることができます。プロデューサーの場合、ウィンドウサイズは 1 度にインフライトにできるバイト数を決定します。

8.1. コンシューマーウィンドウサイズの設定

クライアント側のバッファーに保持されるメッセージの最大サイズは、その **ウィンドウサイズ** によって決定されます。AMQ Core Protocol JMS のウィンドウのデフォルトサイズは 1 MiB または 1024 * 1024 バイトです。ほとんどのユースケースでは、デフォルトでは問題ありません。その他のケースでは、ウィンドウサイズの最適な値を見つけるには、システムのベンチマークが必要になる場合があります。AMQ Core Protocol JMS では、デフォルトを変更する必要がある場合はバッファーウインドウサイズを設定できます。

以下の例は、AMQ Core Protocol JMSを使用する場合にコンシューマーウインドウサイズパラメーターを設定する方法を示しています。それぞれの例では、コンシューマーウィンドウサイズを 300,000 バイトに設定します。

手順

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化する場合は、connection string URL の一部として **consumerWindowSize** パラメーターを含めます。JNDI コンテキスト環境内に URL を保存します。以下の例では、**jndi.properties** ファイルを使用して URL を保存します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?
consumerWindowSize=300000
```

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化しない場合は、値を **ActiveMQConnectionFactory.setConsumerWindowSize()** に渡します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(300000);
```

8.2. プロデューサーウィンドウサイズの設定

ウィンドウサイズは、クレジットに基づいてブローカーとプロデューサーの間でネゴシエートされます。これは、ウィンドウの各バイトに1つのクレジットです。メッセージが送信されてクレジットが使用されると、プロデューサーは追加のメッセージを送信する前に、ブローカーからクレジットを要求し、付与する必要があります。プロデューサーとブローカー間のクレジットの交換により、プロデューサーとブローカー間のデータフローが分離されます。

以下の例は、AMQ Core Protocol JMSを使用する場合にプロデューサーウインドウサイズを 1024 バイトに設定する方法を示しています。

手順

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化する場合は、接続文字列 URL の一部として **producerWindowSize** パラメーターを含めます。JNDI コンテキスト環境内に URL を保存します。以下の例では、**jndi.properties** ファイルを使用して URL を保存します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616?producerWindowSize=1024
```

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化しない場合は、値を **ActiveMQConnectionFactory.setProducerWindowSize()** に渡します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setProducerWindowSize(1024);
```

8.3. 高速コンシューマーの処理

高速コンシューマーは、メッセージをコンシュームすると同時に処理できます。メッセージングシステムのコンシューマーが高速であると確信できる場合は、ウィンドウサイズを -1 に設定することを検討してください。ウィンドウサイズをこの値に設定すると、クライアントでバインドされていないメッセージのバッファリングが可能になります。ただし、この設定は注意して使用してください。コンシューマーが受信と同時にメッセージを処理できない場合、クライアントのメモリーをオーバーフローさせることができます。

高速コンシューマーのウィンドウサイズの設定

以下の例は、メッセージの高速コンシューマーである AMQ Core Protocol JMS クライアントを使用する場合に、ウィンドウサイズを -1 に設定する方法を示しています。

手順

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化する場合は、connection string URL の一部として **consumerWindowSize** パラメーターを含めます。JNDI コンテキスト環境内に URL を保存します。以下の例では、**jndi.properties** ファイルを使用して URL を保存します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?consumerWindowSize=-1
```

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化しない場合は、値を **ActiveMQConnectionFactory.setConsumerWindowSize()** に渡します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(-1);
```

8.4. 低速なコンシューマーの処理

低速なコンシューマーは、各メッセージを処理するのにかなり時間がかかります。このような場合、クライアントでメッセージをバッファーすることは推奨されません。メッセージはブローカーに残り、他のコンシューマーによって消費されます。バッファーをオフにする利点の 1 つは、キュー上の複数のコンシューマー間で確定的な分散を提供することです。クライアント側のバッファーを無効にして低速なコンシューマーを処理するには、ウィンドウサイズを 0 に設定します。

低速なコンシューマーのウィンドウサイズの設定
以下の例は、メッセージの低速なコンシューマーである Core JMS クライアントを使用する場合に、ウィンドウサイズを 0 に設定する方法を示しています。

手順

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化する場合は、connection string URL の一部として **consumerWindowSize** パラメーターを含めます。JNDI コンテキスト環境内に URL を保存します。以下の例では、**jndi.properties** ファイルを使用して URL を保存します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?consumerWindowSize=0
```

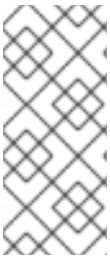
- クライアントが JNDI を使用して接続ファクトリーをインスタンス化しない場合は、値を **ActiveMQConnectionFactory.setConsumerWindowSize()** に渡します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(0);
```

関連情報
低速なコンシューマーを処理する場合にコンシューマーをバッファーしないようにブローカーを設定する方法を示す例は、**<install-dir>/examples/standard** の **no-consumer-buffering** の例を参照してください。

8.5. メッセージ消費率の設定

コンシューマーがメッセージを消費できるレートを調整できます。**スロットリング** としても知られており、消費率は、コンシューマーが設定を許可するよりも高速にメッセージを消費しないようにします。



注記

レート制限のあるフロー制御は、ウィンドウベースのフロー制御と併用できます。レート制限のあるフロー制御は、クライアントが 1 秒に消費できるメッセージ数のみに影響し、バッファー内のメッセージ数には影響しません。レート制限が遅く、ウィンドウベースの制限が高いと、クライアントの内部バッファーがメッセージですぐに一杯になります。

この機能を有効にするには、レートは正の整数である必要があります。1 秒あたりのメッセージ単位で指定される必要なメッセージ消費率の最大値です。レートを -1 に設定すると、レート制限のあるフロー制御が無効になります。デフォルト値は -1 です。

以下の例では、メッセージの消費速度を毎秒 10 メッセージに制限するクライアントを使用しています。

手順

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化する場合は、connection string URL の一部として **consumerMaxRate** パラメーターを含めます。JNDI コンテキスト環境内に URL を保存します。以下の例では、**jndi.properties** ファイルを使用して URL を保存します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616?consumerMaxRate=10
```

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化しない場合は、値を **ActiveMQConnectionFactory.setConsumerMaxRate()** に渡します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerMaxRate(10);
```

関連情報
コンシューマーレートの制限方法の作業例は、**<install-dir>/examples/standard** の **consumer-rate-limit** の例を参照してください。

8.6. メッセージ実稼働環境率の設定

AMQ Core Protocol JMS は、プロデューサーがメッセージを送信するレートを制限することもできます。プロデューサーレートは、1 秒あたりのメッセージ単位で指定します。デフォルトの -1 に設定すると、レート制限のあるフロー制御が無効になります。

以下の例は、プロデューサーが AMQ Core Protocol JMS クライアントを使用している場合にメッセージを送信するレートを設定する方法を示しています。それぞれの例では、最大レートを 1 秒あたり 10 個のメッセージに設定します。

手順

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化する場合は、接続文字列 URL の一部として **producerMaxRate** パラメーターを含めます。JNDI コンテキスト環境内に URL を保存します。以下の例では、**jndi.properties** ファイルを使用して URL を保存します。

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616?producerMaxRate=10
```

- クライアントが JNDI を使用して接続ファクトリーをインスタンス化しない場合は、値を **ActiveMQConnectionFactory.setProducerMaxRate()** に渡します。

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setProducerMaxRate(10);
```

関連情報
メッセージの送信速度を制限する方法の例は、**<install-dir>/examples/standard** の **producer-rate-limit** の例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. パッケージ用システムの登録

この製品の 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。

システムを登録する方法は、以下のリソースを参照してください。

- Red Hat Enterprise Linux 7 - システム登録およびサブスクリプション管理
- Red Hat Enterprise Linux 8 - システム登録およびサブスクリプション管理

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 Core Protocol JMS の例では、**exampleQueue** という名前のキューを持つ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、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 **exampleQueue** という名前のキューを作成します。

```
$ <broker-instance-dir>/bin/artemis queue create --name exampleQueue --address exampleQueue
--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： 2022-09-08 16:28:42 +1000