# Red Hat AMQ 2021.Q3

## AMQ Core Protocol JMS クライアントの使用

AMQ Clients 2.10 向け

Last Updated: 2023-02-23

# Red Hat AMQ 2021.Q3 AMQ Core Protocol JMS クライアントの使用

AMQ Clients 2.10 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、クライアントのインストールや設定、実例の実行、他の AMQ コンポーネントでのクライアントの使用方法について説明します。

# 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。これは大規模な取り組みであるため、これらの変更は今後の複数のリリースで段階的に実施されます。詳細は、Red Hat CTO である Chris Wright のメッセージをご覧ください。

# 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される Java Message Service (JMS) 2.0 クライアントです。

AMQ Core Protocol 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 Overview」を参照してください。本リリースに関する詳細は、『AMQ Clients 2.10 Release Notes』を参照してください。

AMQ Core Protocol JMS は、Apache ActiveMQ Artemis 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
- SSL/TLS でのセキュアな通信
- 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. サポート対象の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 でサポートされる構成については、Red Hat カスタマーポータルの「Red Hat AMQ 7 Supported Configurations」を参照してください。

## 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、それらが一緒に操作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。 |
| **Connection** | ネットワーク上の 2 つのピア間の通信用のチャネル。これにはセッションが含まれます。 |

| エンティティー | 説明 |
|---|---|
| **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。 |
| **Destination** | 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。 |
| **Topic** | 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。 |
| **Message** | 情報のアプリケーション固有部分。 |

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。プロデューサーおよびコンシューマーは **セッション** 上で確立されます。セッションは**コネクション**上で確立されます。コネクションは**接続ファクトリー**によって作成されます。

送信ピアは、メッセージを送信するために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、メッセージを受信するために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が利用できます。

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 権限を必要とするコマンドには **sudo** が使用されています。何らか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**

本書には、実際の環境に固有の値に置き換える必要がある変数を含むコードブロックが含まれています。変数テキストは中括弧で囲まれ、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、以下の例では、**<project-dir>** を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、お使いの環境に 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。設定ファイルの例については、「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-00022</version>
</dependency>
```

これで、クライアントが Maven プロジェクトで利用できるようになりました。

## 2.3. 「INSTALLING A LOCAL MAVEN REPOSITORY」

オンラインリポジトリーの代わりに、ファイルベースの Maven リポジトリーとして AMQ Core Protocol JMS を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 7.8.2 Maven リポジトリー**の .zip ファイルをダウンロードします。
2. ファイルの内容を、選択したディレクトリーに展開します。
   Linux または UNIX の場合は、**unzip** コマンドを使用してファイルの内容を展開します。

```
$ unzip amq-broker-7.8.2-maven-repository.zip
```

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

3. 展開し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 7.8.2**の .zip ファイルをダウンロードします。
2. ファイルの内容を、選択したディレクトリーに展開します。
Linux または UNIX の場合は、**unzip** コマンドを使用してファイルの内容を展開します。

```
$ unzip amq-broker-7.8.2.zip
```

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

.zip ファイルの内容を展開すると、**amq-broker-7.8.2** 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、本書全体で **<install-dir>** と呼ばれます。

# 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

- サンプルをビルドするには、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、「Creating a queue」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た後、それを受信して、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サンプルを実行します。
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。その他の例は、**<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、特定の実装によって制限されることなく、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。

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

```
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。これを行う方法は、**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の設定

#### 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で設定します。

#### 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ることに注意してください。

## 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。この設定を **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、**app1** という名前のファクトリーを以下のように設定します。

**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** という名前を使い、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についての詳細は、5章*SSL 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、別のターゲットに対して接続を試みます。以下の形式を使用します。

**フェイルオーバー URI 形式**

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

括弧外部のオプションは、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に **jobs** および **notifications** という名前を定義します。

**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 SSL 設定オプション

この章では、AMQ Core Protocol JMS で利用可能な設定オプションを紹介します。

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、メッセージが確認されます。これは「at most once」(最大 1 度) の配信を提供します。これはデフォルトで無効になっています。

**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無効になっています。

**blockOnAcknowledge**

有効な場合は、受信された非トランザクションメッセージを確認すると、リモートピアがその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無効になっています。

**callTimeout**

ブロッキングコールが完了するまでの待ち時間をミリ秒単位で指定します。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ち時間をミリ秒単位で指定します。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までに、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** ack モードを使用する場合の、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時の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**

有効な場合、宛先検索をキャッシュします。これはデフォルトで無効になっています。

## 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) 未満であると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) を超えると書き込みバッファーが書き込み不可能になります。デフォルトは 131072 (128 KiB) です。

## 5.3. SSL/TLS オプション

**sslEnabled**

有効な場合、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無効になっています。

**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。設定しない場合、**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。設定しない場合、**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。設定しない場合、**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。未設定の場合、**javax.net.ssl.trustStorePassword** システムプロパティーの値が使用されます。

**trustAll**

有効な場合、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無効になっています。

**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無効になっています。

**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、JVM のデフォルトの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 のデフォルトの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無効になっています。

**reconnnectAttempts**

接続の失敗が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、初期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に伴う更新を受け取ります。これはデフォルトで無効になっています。

**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**

コマンドリプレイ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**

デッド接続をチェックする間隔 (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. フロー制御のオプション

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) です。-1 は無制限を意味します。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 度のインフライトデータの総量が制限されます。デフォルトは 1048576 (1 MiB) です。-1 は無制限を意味します。

producerMaxRate

1 秒あたりの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、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、**minLargeMessageSize** プロパティーの値を設定することで、大きなメッセージの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) です。

compressLargeMessages

有効な場合、**minLargeMessageSize** で定義され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無効になっています。

**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

有効な場合は、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

汎用スレッドプールのスレッドの最大数。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、すべてのマスターと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と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ない状態になります。フェイルオーバー時のインフライト送信または確認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 **once and only once** (1 回限り) 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 つのオプションは、特定のブローカーに接続し、クラスター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することです。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。

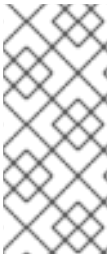
1. URI の **host:port** 部分は、バックアップで適切に設定されたマスターブローカーを示している必要があります。このホストおよびポートは最初の接続にのみ使用されます。**host:port** の値は、ライブサーバーとバックアップサーバー間の実際の接続フェイルオーバーとは関係ありません。上記の例では、**localhost:61616** は **host:port** に使用されます。

2. (任意手順) 複数のブローカーを最初の接続の候補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. 名前と値のペア **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 よりも大きな整数です。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** および **reconnectAttempts** が 0 よりも大きい場合にのみ発生します。また、他のブローカーについての情報を取得するためには、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。最初の接続に失敗した場合、クライアントは再試行して接続を確立することしかできません。詳細は、「最初の接続時のフェイルオーバー」 を参照してください。

### 6.1.1. 最初の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。そのため、この最初の接続中に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行できます。クライアントには、再接続の試行回数を設定することができます。試行回数が設定されると、例外が発生します。

**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 に設定します。デフォルト値は 0 で、1 回のみ試行します。

**手順**

値を **ServerLocator.setInitialConnectAttempts()** に渡して、再接続試行数を設定します。

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

**再接続試行回数のグローバル設定**
また、ブローカーの設定内で再接続試行回数のグローバルな最大値を適用できます。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、**initial-connect-attempts** 設定要素を追加して、time-to-live の値を提供することで **<broker-instance-dir>/etc/broker.xml** を編集します。

```
<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では、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、クライアントの再試行は無制限です。

### 6.1.2. フェイルオーバー時のブロッキングコールの対処

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っているとき、新たに作成されたセッションは、進行中のコールについて認識しません。そうでないと、最初の呼び出しは永久にハングし、来るはずのない応答を待つことになります。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コール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()** への呼び出しであれば、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スローされます。

### 6.1.3. トランザクションによるフェイルオーバーの対処

AMQ Core Protocol JMS クライアントを使用する場合、セッションがトランザクションであり、メッセージが現在のトランザクションですでに送信または確認されていると、ブローカーはフェイルオーバー中にこれらのメッセージまたはその確認が失われたことを確認できません。そのため、トランザクションはロールバックのみにマークされます。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は、ロールバックすると **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ように通知します。元のコミットが発生しなかった場合、それはまだ存在し、コミットすることができます。コミットが存在しない場合は、トランザクションマネージャーが警告を記録することがありますが、コミットされたものとみなされます。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。このような損失を避けるために、XA を使用する際には必ず永続メッセージを使用してください。確認応答は、**prepare()** が呼び出される前にブローカーにフラッシュされるため、問題はありません。

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。その後、ユーザーは同じセッションで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のブロックを解除し、AMQ Core Protocol JMS クライアントが応答を永遠に待たないようにします。そのため、クライアントは、障害が発生する前に、トランザクションのコミットがマスターブローカーで実際に処理されたかどうかを判断できません。

これに対処するため、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できます。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ない場合、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。非トランザクションセッションに **once and only once** (1 回限り) 配信を保証する場合は、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. 接続の障害の通知

JMS は、接続障害が非同期に通知される標準メカニズムを提供します。これは **java.jms.ExceptionListener** です。

接続の障害が発生すると、接続のフェイルオーバー、再接続、または再アタッチが正常に行われたかに関わらず、**ExceptionListener** または **SessionFailureListener** インスタンスは常にブローカーによって呼び出されます。**SessionFailureListener** の **connectionFailed** で渡された **failedOver** フラグを調べることで、再接続または再アタッチ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アタッチまたは再接続しました。 |
| DISCONNECT | フェイルオーバーは発生せず、ブローカーが切断されています |

## 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、自動的なクライアントのフェイルオーバーを望まず、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こともあります。フェイルオーバーは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呼ばれます。

JMS の使用時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るとブローカーによって呼び出されます。**ExceptionListener** で、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場合もあります。

## 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 30,000 ミリ秒 (30 秒) です。VM 内接続のデフォルト値は -1 で、これは、データが受信されなかった場合にクライアント側から接続を失敗させません。

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ます。これにより、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なります。

**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 10,000 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 (例: **jndi.properties**) 内で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 time-to-live (TTL) を設定できます。以下の例は、TTL の設定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 を使用しない場合、接続 TTL は **ActiveMQConnectionFactory** インスタンスの **ConnectionTTL** 属性で定義されます。

```
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するため、終了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 のホストおよびパートは、ブローカーの **broker.xml** 設定ファイル内の対応する **broadcast-group** からの **group-address** および **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
```

この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で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、検出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、作成からこの期間待つようにします。デフォルトの待機時間は 10000 ミリ秒ですが、新しい値を **DiscoveryGroupConfiguration.setDiscoveryInitialWaitTimeout()** 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。常に利用できることが判明している 1 つのブローカーだけのリストであっても、少なくとも 1 つは利用できるブローカーのリストであってもかまいません。

これは、すべてのサーバーがホストされる場所を認識する必要があるわけではありません。これらのサーバーが、信頼できるサーバーを使用して接続するように設定できます。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、それを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。以下の例は、メッセージからバイトを読み取り、それをファイルに書き込みます。

**例: ストリーミングし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、以下の特徴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** になります。
- 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使用されます。元のコンシューマーが閉じられている場合、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に処理したい

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るようになります。

### 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、プロパティー **JMSXGroupID** 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);
```

### その他のリソース

メッセージグループの設定方法と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. 複製メッセージ検出の使用

AMQ Broker には、受信する重複メッセージを除外する自動複製メッセージ検出が含まれています。そのため、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。

複製メッセージの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。ある場合、ブローカーはメモリーキャッシュを確認し、その値のあるメッセージをすでに受信したかどうかを確認します。同じ値がある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、トランザクションのすべてのメッセージに **_AMQ_DUPL_ID** を設定する必要はなく、1 つのメッセージのみに設定します。ブローカーがトランザクションにあるメッセージの重複メッセージを検出した場合、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 **_AMQ_DUPL_ID** の名前に定数 **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 を使用すると、クライアントが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処理を行うことができます。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。返された値が **true** の場合、メッセージパケットは処理を継続します。返された値が **false** の場合、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そ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行われます。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した場合、呼び出し元に ActiveMQException がスローされます。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** インターフェー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しするには、クライアントの Java クラスパ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の間のデータの流れを制限することで、プロデューサーとコンシューマーの負荷が大きくなるのを防ぎます。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の制御**
コンシューマーフローの制御は、クライアントがブローカーからのメッセージを消費する際に、ブローカーとクライアントの間のデータフローを制御します。AMQ Core Protocol JMS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は、クライアントはメッセージを消費する前に、最初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」の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ぐに処理できず、バッファーが受信メッセージでオーバーフローすることがあるため、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 のウイ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、デフォルトのままで問題ありません。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ために、システムのベンチマークが必要な場合があります。AMQ Core Protocol JMS を使用すると、デフォルトを変更する必要がある場合に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、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、接続文字列 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 を格納します。以下の例では、**jndi.properties** ファイルを使用して URL を格納します。

```
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、クレジットを基に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対して 1 クレジットです。メッセージが送信されおよびクレジットが使用されると、プロデューサーは、さらにメッセージを送信する前に、ブローカーにクレジットを要求し、それが付与される必要があります。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制限されます。

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 を格納します。以下の例では、**jndi.properties** ファイルを使用して URL を格納します。

```
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対処

高速コンシューマーは、メッセージを消費するのと同じ速さでメッセージを処理することができます。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、この設定には注意が必要です。コンシューマーがメッセージの受信と同時に処理できない場合、クライアントのメモリーがオーバーフローする可能性があります。

### 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、接続文字列 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 を格納します。以下の例では、**jndi.properties** ファイルを使用して URL を格納します。

```
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対処

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の時間を要します。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決定性のある分配が可能になることです。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 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、接続文字列 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 を格納します。以下の例では、**jndi.properties** ファイルを使用して URL を格納します。

```
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);
```

**その他のリソース**
低速なコンシューマーの対処時にコンシューマーのバッファーを防ぐため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比率を規制できます。これは**スロットリング** とも呼ばれ、コンシューマーが設定の許容範囲を超えてメッセージを消費しないように、消費率を制限します。

**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せて使用できます。レート制限のあるフロー制御は、クライアントが 1 秒ごと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いっぱいになってしまいます。

この機能を有効にするには、レートには正の整数を指定する必要ががあります。これは 1 秒あたりのメッセージ数を単位として指定された希望する最大のメッセージ消費率になります。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。

以下の例は、メッセージの消費レートを毎秒 10 個のメッセージに制限するクライアントを示しています。

**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 **consumerMaxRate** パラメーターを含めます。JNDI コンテキスト環境内に URL を格納します。以下の例では、**jndi.properties** ファイルを使用して URL を格納します。

```
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);
```

**その他のリソース**
コンシューマーレートの制限方法の例は、**<install-dir>/examples/standard** の **consumer-rate-limit** の例を参照してください。

## 8.6. メッセージ生成率の設定

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数を単位として指定されます。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 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。各例では、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 を格納します。以下の例では、**jndi.properties** ファイルを使用して URL を格納します。

```
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);
```

**その他のリソース**
メッセージ送信レートの制限方法の例は、**<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、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. **INTEGRATION AND AUTOMATION** カテゴリーで **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. パッケージを受信するためのシステムの登録

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、お使いのシステムを登録する必要があります。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 リポジトリーの使用

本セクションでは、ソフトウェアで Red Hat が提供する Maven リポジトリーを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 Maven リポジトリーを維持します。詳細は、リポジトリーの welcome 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
- リポジトリーを POM ファイルに追加する

### Maven 設定へのリポジトリーの追加

この設定の手法は、POM ファイルがリポジトリー設定を上書きせず、含まれ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 **settings.xml** ファイルを見つけます。通常、これはユーザーのホームディレクトリー内の **.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. 以下の例のように、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 リポジトリーが含まれる 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 設定に関する詳細は、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**
プロジェクトに直接リポジトリーを設定するには、以下の例のように、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 リポジトリーが含まれる 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対してファイルベースの 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。

ローカルに抽出し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 ファイルに以下の XML を適用します。

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

**${repository-url}** 展開し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の URL でなければなりません。

**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 サンプルでの AMQ BROKER の使用

AMQ Core Protocol JMS のサンプルでは、**exampleQueue** という名前のキューが含まれる稼働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

『**AMQ Broker の使用**』の説明に従い ブロッ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、**<broker-instance-dir>** としてブローカーインスタンスの場所を参照します。

## 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. コンソールの出力で、起動時にログに記録される重要なエラーの有無を確認します。ブローカーは、準備が整う際に **Server is now live** をログに記録します。

```
$ 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
```

yes または no の質問への回答を求めるプロンプトが表示されます。そのすべてに no (**N**) 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と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:** 2021-08-29 15:56:06 +1000