BEA WebLogic Server™

WebLogic JMS プログラマーズ ガイド

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 :2002 年 8 月 23 日

著作権

限定的権利条項

商標または登録商標

WebLogic JMS プログラマーズ ガイド

パート番号	マニュアルの改訂	ソフトウェアのバージョン
なし	2002 年 8 月 23 日	BEA WebLogic Server バージョン 7.0

目次

3. WebLogic JMS の管理

4. WebLogic JMS アプリケーションの開発

5. WebLogic JMS によるトランザクションの使い方

6. WebLogic JMS アプリケーションの移植

A. コンフィグレーション チェックリスト

B. JDBC データベース ユーティリティ

このマニュアルの内容

このマニュアルでは、BEA WebLogic Server™ プラットフォームで Java™ Messaging Service（JMS）を実装してエンタープライズ メッセージング システムにアクセスする方法について説明します。

このマニュアルの構成は次のとおりです。

- 第 1 章「WebLogic JMS の概要」では、WebLogic Java Message Service（JMS）について概説します。
- 第 2 章「WebLogic JMS の基礎」では、WebLogic JMS のコンポーネントおよび機能について説明します。
- 第 3 章「WebLogic JMS の管理」では、WebLogic JMS のコンフィグレーションおよびモニタについて概説します。
- 第 4 章「WebLogic JMS アプリケーションの開発」では、WebLogic JMS アプリケーションを開発する方法について説明します。
- 第 5 章「WebLogic JMS によるトランザクションの使い方」では、WebLogic JMS でトランザクションを使用する方法について説明します。
- 第 6 章「WebLogic JMS アプリケーションの移植」では、WebLogic JMS アプリケーションを WebLogic Server の新リリースへ移植する方法について説明します。
- 付録 A「コンフィグレーション チェックリスト」では、各種 JMS 機能のモニタ チェックリストを提供します。
- 付録 B「JDBC データベース ユーティリティ」では、JDBC データベース ユーティリティを使用して、新しい JDBC ストアを生成したり、また削除したりする方法を説明します。

対象読者

このマニュアルは、Sun Microsystems の Java 2 Platform, Enterprise Edition（J2EE）を使用して JMS アプリケーションを設計、開発、コンフィグレーション、および管理するアプリケーション開発者を対象としています。JMS、JNDI（Java Naming and Directory Interface）、Java プログラミング言語、エンタープライズ JavaBeans™（EJB™）、および J2EE 仕様の Java Transaction API（JTA）に読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。または、WebLogic Server 製品ドキュメント ページ（http://edocs.beasys.co.jp/e-docs/index.html）を直接表示してください。

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルを一度に 1 章ずつ印刷できます。

このマニュアルの PDF 版は、Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体（または一部分）を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は Adobe の Web サイト（http://www.adobe.co.jp）で無料で入手できます。

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。JMS の詳細については、Sun Microsystems Web サイトの以下の場所にある JMS 仕様および Javadoc にアクセスしてください。

http://java.sun.com/products/jms/docs.html

サポート情報

BEA のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで docsupport-jp@beasys.com までお送りください。寄せられた意見については、WebLogic Server のドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェアの名前とバージョン、およびドキュメントのタイトルと日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSupport（www.bea.com）を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+ 〔Tab〕	複数のキーを同時に押すことを示す。
斜体	強調または書籍のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、データ構造体とそのメンバー、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

斜体の等幅テキスト	コード内の変数を示す。 例: `String CustomerName;`
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例: LPT1 BEA_HOME OR
{ }	構文の中で複数の選択肢を示す。

表記法	適用
[ ]	構文の中で任意指定の項目を示す。 例： `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	構文の中で相互に排他的な選択肢を区切る。 例： `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1 WebLogic JMS の概要

以下の節では、BEA WebLogic Server の Java Message Service（JMS）について概説します。

- JMS とは
- Java 仕様の実装
- WebLogic JMS の機能
- WebLogic JMS のアーキテクチャ
- WebLogic JMS の拡張機能
- WebLogic Server 7.0 における JMS の拡張機能

JMS とは

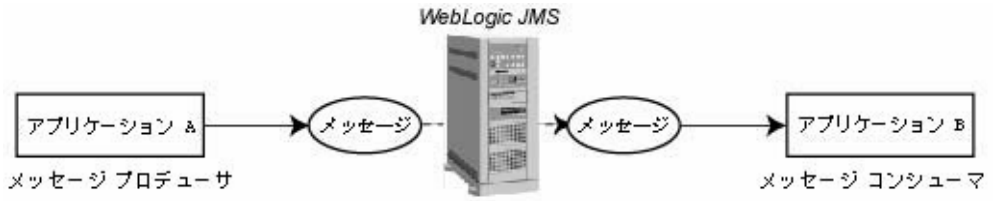
メッセージ指向ミドルウェア（Message-Oriented Middleware : MOM）とも呼ばれるエンタープライズ メッセージング システムを使用すると、複数のアプリケーションがメッセージの交換を通じて通信できます。メッセージとは、異なるアプリケーション間の通信を調整するために必要な情報が含まれている要求、レポート、またはイベントのことです。メッセージで提供される抽象化の階層により、送り先システムについての詳細情報をアプリケーション コードから切り離すことができます。

Java Message Service（JMS）は、エンタープライズ メッセージング システムにアクセスするための標準の API です。具体的な JMS の特長は以下のとおりです。

- メッセージング システムを共有する Java アプリケーション同士でメッセージを交換できます。
- メッセージを作成、送信、および受信するための標準インタフェースによりアプリケーションの開発が容易になります。

次の図は、WebLogic JMS によるメッセージングの仕組みを示しています。

図 1-1 WebLogic JMS のメッセージング



図で示されているように、WebLogic JMS はプロデューサ アプリケーションからメッセージを受信し、受け取ったメッセージをコンシューマ アプリケーションに配信します。

Java 仕様の実装

WebLogic Server は、以下の Java 仕様に準拠しています。

J2EE 仕様

WebLogic Server 7.0 は、Sun Microsystems の J2EE 1.3 仕様に準拠しています。

JMS 仕様

WebLogic Server 7.0 は、JMS 仕様バージョン 1.0.2b に完全に準拠しており、それをプロダクション段階で使用することもできます。

WebLogic JMS の機能

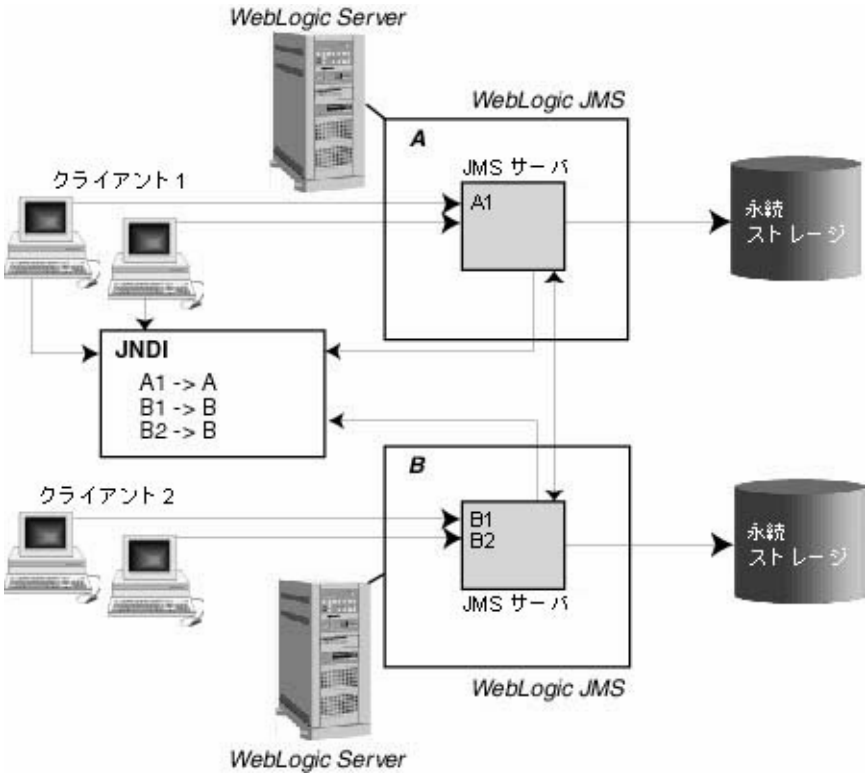
WebLogic JMS では、JMS API の完全な実装が提供されます。具体的な WebLogic JMS の機能は以下のとおりです。

- 1 つの統一的なメッセージング API を提供します。
- JMS 仕様バージョン 1.0.2b に厳密に従っています。
- クラスタ化をサポートします。
- さまざまなオペレーティング システムおよびマシン アーキテクチャにまたがるアプリケーションのメッセージングをサポートします。
- WebLogic Administration Console で属性を設定したり、JMS API を使用して値をオーバライドしたりすることでコンフィグレーションできます。
- Java Transaction API（JTA）トランザクションを使用して、JMS アプリケーションと他のリソース マネージャ（主にデータベース）間の相互運用性を可能にします。分散トランザクションや、2 フェーズ コミット プロトコルのサポートも含まれます。JMS アプリケーションは、WebLogic XA に準拠しないメッセージ ブローカなど、JTA を使用する他の Java API とのトランザクションに参加することもできます。
- XML（Extensible Markup Language）を含むメッセージをサポートします。
- マルチキャストをサポートします。IP マルチキャスト アドレスを使用して、選択したホストのグループにメッセージを配信できます。
- メッセージの永続ストレージとしてデータベースまたはファイルを使用できます。
- エンタープライズ JavaBean（EJB）、JDBC 接続プール、サーブレット、RMI など、他の WebLogic Server API や機能と共に使用できます。

WebLogic JMS のアーキテクチャ

次の図は、WebLogic JMS のアーキテクチャを示しています。

図 1-2 WebLogic JMS のアーキテクチャ



主要な構成要素

1-4 ページの「WebLogic JMS のアーキテクチャ」の図で示されているように、WebLogic JMS Server のアーキテクチャは主に以下の要素で構成されています。

- メッセージング機能を実装する WebLogic JMS サーバ
- クライアント アプリケーション
- サーバ ルックアップ機能を提供する JNDI（Java Naming and Directory Interface）
- 永続的なメッセージ データを格納するための永続ストレージ（ファイルまたはデータベース）

クラスタ化機能

WebLogic JMS のアーキテクチャでは、クラスタ内のあらゆるサーバから送り先へのクラスタワイドで透過的なアクセスをサポートすることで、複数の JMS サーバのクラスタ化が実装されます。WebLogic Server は、クラスタ全体への JMS の送り先と接続ファクトリの配布をサポートするようになりました。ただし、JMS トピックおよびキューが、クラスタ内の個々の WebLogic Server インスタンスによって管理される点は変わりません。

WebLogic JMS のクラスタ化のコンフィグレーションの詳細については、3-3 ページの「WebLogic JMS のクラスタ化のコンフィグレーション」を参照してください。WebLogic Server のクラスタ化の詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』を参照してください。

クラスタ化のメリットは以下のとおりです。

- クラスタ内の複数のサーバにわたる送り先のロード バランシング
  - 管理者は、複数の JMS サーバをコンフィグレーションし、対象を使用してそれらを定義済みの WebLogic Server に割り当てることで、クラスタ内の複数のサーバにわたる送り先のロード バランシングを確立できます。各 JMS サーバは、厳密に 1 つの WebLogic Server にデプロイされ、複数の送り先に対する要求を処理します。

注意： ロード バランシングは動的ではありません。コンフィグレーションの段階で、システム管理者が JMS サーバの対象を指定してロード バランシングを定義します。

- 管理者は、クラスタ内の単一の分散送り先セットのメンバーとして、複数の物理的な送り先をコンフィグレーションすることもできます。プロデューサとコンシューマは、その分散送り先に対して送受信することができます。クラスタ内の 1 台のサーバに障害が発生した場合、WebLogic JMS は送り先セット内の使用可能な物理的送り先メンバーすべてに負荷を分散します。

  分散送り先の詳細については、『管理者ガイド』の「分散送り先のコンフィグレーション」を参照してください。

■ クラスタ内のあらゆるサーバからの、送り先へのクラスタワイドで透過的なアクセス

システム管理者は、クラスタ内のあらゆるサーバから送り先へのクラスタワイドで透過的なアクセスを確立できます。このようなアクセスを確立するには、クラスタ内の各サーバ インスタンスに対してデフォルトの接続ファクトリを有効化するか、1 つまたは複数の接続ファクトリをコンフィグレーションしてクラスタ内の 1 つまたは複数のサーバ インスタンスに割り当てます。これにより、各接続ファクトリを複数の WebLogic Server にデプロイすることが可能になります。

アプリケーションでは、Java Naming and Directory Interface（JNDI）を使用して接続ファクトリをルックアップし、JMS サーバとの通信を確立するための接続を作成します。各 JMS サーバでは、複数の送り先に対する要求が処理されます。JMS サーバで処理されない送り先への要求は、適切なサーバに転送されます。

接続ファクトリの詳細については、2-1 ページの「WebLogic JMS の基礎」を参照してください。

■ スケーラビリティ

スケーラビリティは以下の機能によって実現します。

- 前述した、クラスタ内の複数のサーバにわたる送り先のロード バランシング。
- 接続ファクトリを通じた、複数の JMS サーバでのアプリケーション負荷の分散。その結果として、個々の JMS サーバでの負荷が削減され、かつ接続先を特定のサーバに決めることでセッションの集中が可能となります。

- マルチキャストのサポート（オプション）。JMS サーバで配信しなければならないメッセージの数が削減されます。JMS サーバでは、サブスクライブしているアプリケーションの数に関係なく、マルチキャスト IP アドレスと関連付けられている各ホスト グループに対してメッセージが 1 コピーだけ転送されます。

■ 移行性

「1 回限りのサービス」として、WebLogic JMS はクラスタ化された環境に対して WebLogic Server に実装されている移行フレームワークを活用します。これによって、WebLogic JMS は移行の要求に適切に応答し、JMS サーバを適切にオンライン / オフラインに切り替えることができます。移行には、スケジューリング済み移行のほかに、WebLogic Server の障害に応答して発生する移行が含まれます。詳細については、3-7 ページの「JMS 移行可能対象のコンフィグレーション」を参照してください。

注意： 自動フェイルオーバは、このリリースの WebLogic JMS ではサポートされていません。手動フェイルオーバの実行の詳細については、3-12 ページの「WebLogic Server の障害からの回復」を参照してください。

WebLogic JMS の拡張機能

Sun Microsystems の JMS 仕様による API に加えて、WebLogic JMS には weblogic.jms.extensions というパブリック API が用意されています。この API には、以下の表で説明される拡張機能のクラスやメソッドが含まれています。

拡張機能	詳細情報の参照先
XML メッセージを作成する	4-15 ページの「手順 6a : メッセージ オブジェクトを作成する (メッセージ プロデューサ)」を参照
セッション例外リスナを定義する	4-50 ページの「セッション例外リスナの定義」を参照
事前に取得する非同期メッセージの、セッションで許可される最大数を設定または表示する	4-91 ページの「マルチキャストのコンフィグレーション属性の動的コンフィグレーション」を参照

拡張機能	詳細情報の参照先
メッセージが最大数に達したときに適用するマルチキャスト セッションの超過時のポリシーを設定または表示する	4-91 ページの「マルチキャストのコンフィグレーション属性の動的コンフィグレーション」を参照
永続的なキューまたはトピックを動的に作成する	4-53 ページの「JMSHelper クラス メソッドの使い方」を参照
WebLogic JMS 7.0 と 6.0 以前の `JMSMessageID` の形式をお互いの形式に変換する	4-63 ページの「メッセージ ヘッダ フィールドの設定」を参照
メッセージの再配信遅延を設定する	4-38 ページの「メッセージの再配信遅延の設定」を参照
プロデューサのメッセージ配信時間を設定する	4-41 ページの「プロデューサに対する配信時間の設定」を参照
メッセージの配信時間を設定する	4-42 ページの「メッセージに対する配信時間の設定」を参照
メッセージのスケジューリング済み配信時間を設定する	4-43 ページの「スケジューリング済み配信時間のオーバーライドの設定」を参照

この API では、`NO_ACKNOWLEDGE` と `MULTICAST_NO_ACKNOWLEDGE` の確認応答モード、および以下のような例外の送出を含む拡張例外もサポートされています。

- サーバ エラーまたは管理上の介入によってコンシューマの 1 つがサーバによって閉じられたときにセッション例外リスナ（設定されている場合）に例外を送出します。
- セッションで受信されたが、まだメッセージ リスナに配信されていないメッセージの数がそのセッションの最大メッセージ許容数を超えたときにマルチキャスト セッションから例外を送出します。
- データ ストリームでシーケンスの欠陥（シーケンスの異なる受信メッセージ）が検出されたときにマルチキャスト コンシューマから例外を送出します。

WebLogic Server 7.0 における JMS の拡張機能

このリリースの WebLogic Server では、JMS の以下の拡張機能が新しく導入されています。

高可用性拡張機能

WebLogic JMS は、クラスタ化された環境に対して WebLogic Server のコアに実装されている移行フレームワークを活用します。これによって、WebLogic JMS は移行の要求に適切に応答し、JMS サーバを適切にオンライン / オフラインに切り替えることができます。移行には、スケジューリング済み移行のほかに、WebLogic Server の障害に応答して発生する移行が含まれます。

詳細については、『WebLogic JMS プログラマーズ ガイド』の「WebLogic JMS の管理」を参照してください。

WebLogic クラスタ内の分散送り先

高い可用性を誇る WebLogic JMS の実装では、複数の物理的な送り先を単一の送り先セットのメンバーとしてコンフィグレーションできるようにすることで、1 台のサーバに障害が発生した場合でもサービスが継続されるようにしています。具体的に言うと、管理者は、クラスタ内にある特定の送り先の複数のインスタンスをコンフィグレーションできます。クラスタ内の 1 つのインスタンスに障害が発生した場合は、同じ送り先の他のインスタンスが JMS プロデューサとコンシューマにサービスを提供できます。

詳細については、『WebLogic JMS プログラマーズ ガイド』の「WebLogic JMS アプリケーションの開発」および『管理者ガイド』の「JMS の管理」を参照してください。

フロー制御

フロー制御機能を使用すると、JMS サーバまたは送り先が過負荷になったときに、メッセージ プロデューサの処理速度を遅くすることができます。具体的に言うと、JMS サーバまたは送り先が、指定のバイト数またはメッセージのしきい値を超過したとき、プロデューサにメッセージのフローを制限するよう指示します。

詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

WebLogic メッセージング ブリッジ

メッセージング ブリッジ（JMS ブリッジとも呼ばれる）は、2 つの JMS プロバイダ間でメッセージを転送します。WebLogic Server メッセージング ブリッジ機能により、2 つのメッセージング プロバイダ（WebLogic JMS の個別の実装を含む）間でのストアおよび転送メカニズムをコンフィグレーションできるようになります。

詳細については、『管理者ガイド』の「WebLogic メッセージング ブリッジの使い方」を参照してください。

メッセージ ページング

メッセージ ページング機能により、メッセージの負荷が指定のしきい値に達したときにメッセージを仮想メモリから永続ストレージにスワップすることで、メッセージ負荷のピーク期間中に仮想メモリが解放されます。パフォーマンスの点から見れば、この機能は、今日のエンタープライズ アプリケーションが必要とする大容量のメッセージ領域を持つ WebLogic Server の実装には大きなメリットがあります。

詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

2 WebLogic JMS の基礎

以下の節では、WebLogic JMS コンポーネントと機能について説明します。

- メッセージング モデル
- WebLogic JMS のクラス
- ConnectionFactory
- Connection
- Session
- Destination
- 送り先の分散
- MessageProducer と MessageConsumer
- Message
- ServerSessionPoolFactory
- ServerSessionPool
- ServerSession
- ConnectionConsumer

注意： この節で説明する JMS クラスの詳細については、Sun Microsystems の Java Web サイトにある以下の JMS 仕様と Javadoc を参照してください。
http://java.sun.com/products/jms/docs.html

メッセージング モデル

JMS では、ポイント ツー ポイント (PTP) とパブリッシュ / サブスクライブ (Pub/sub) の 2 つのメッセージング モデルがサポートされています。それらのメッセージング モデルは非常に似ていますが、以下の点で異なります。

- PTP メッセージング モデルでは、1 つの宛先に対してメッセージが配信される。
- Pub/sub メッセージング モデルでは、複数の宛先に対してメッセージが配信される。

各モデルは、共通の基本クラスを拡張したクラスで実装されます。たとえば、PTP クラスの `javax.jms.Queue` と Pub/sub クラスの `javax.jms.Topic` は両方とも `javax.jms.Destination` を拡張したクラスです。

各メッセージング モデルについては、以降の節で詳しく説明します。

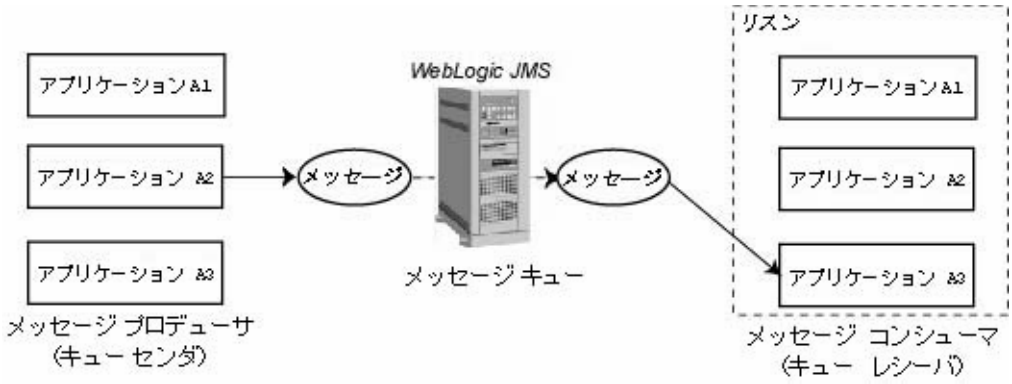
注意 : プロデューサおよびコンシューマという用語は、メッセージング モデルに関係なく、それぞれメッセージを送信および受信するアプリケーションを表すために汎用的に使用します。ただし、各メッセージング モデルでは、それぞれに固有のユニークな用語でプロデューサとコンシューマを表します。

ポイント ツー ポイント メッセージング

ポイント ツー ポイント (PTP) メッセージング モデルでは、アプリケーションが別の 1 つのアプリケーションにメッセージを送信できます。PTP メッセージング アプリケーションでは、名前付きのキューを使用してメッセージが送信および受信されます。キュー センダ (プロデューサ) では、特定のキューに対してメッセージが送信されます。キュー レシーバ (コンシューマ) では、特定のキューからメッセージが受信されます。

次の図は、PTP メッセージングの仕組みを示しています。

図 2-1 ポイント ツー ポイント (PTP) メッセージング



複数のキュー センダおよびキュー レシーバを 1 つのキューに関連付けることができますが、個々のメッセージは 1 つのキュー レシーバにしか配信できません。

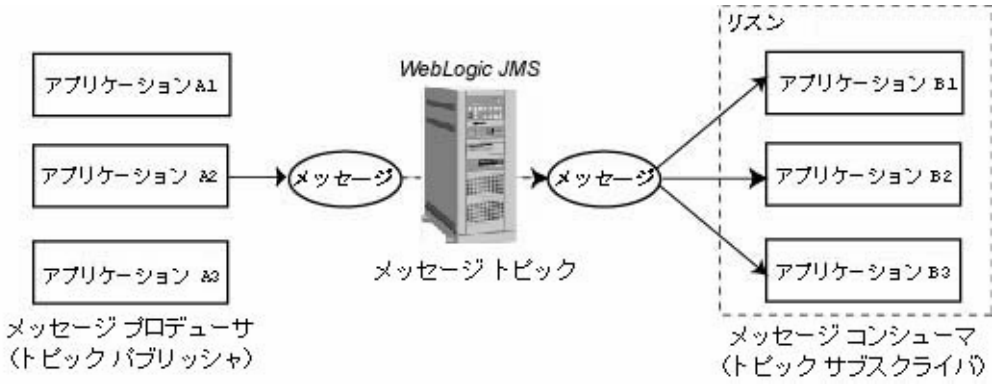
複数のキュー レシーバがキューのメッセージをリスンしている場合、次のメッセージを受信するキュー レシーバは先着順で決定されます。リスンしているキュー レシーバがない場合は、キュー レシーバがキューにアタッチされるまでメッセージはキューにとどまります。

パブリッシュ / サブスクライブ メッセージング

パブリッシュ / サブスクライブ (Pub/sub) メッセージング モデルでは、アプリケーションが複数のアプリケーションにメッセージを送信できます。Pub/sub メッセージング アプリケーションでは、トピックをサブスクライブすることでメッセージが送信および受信されます。トピック パブリッシャ (プロデューサ) では、特定のトピックに対してメッセージが送信されます。トピック サブスクライバ (コンシューマ) では、特定のトピックからメッセージが受信されます。

次の図は、Pub/sub メッセージングの仕組みを示しています。

図 2-2 パブリッシュ / サブスクライブ (Pub/sub) メッセージング



PTP メッセージング モデルの場合と違って、Pub/sub メッセージング モデルでは複数のトピック サブスクライバが同じメッセージを受信できます。メッセージは、すべてのトピック サブスクライバが受信するまで維持されます。

Pub/sub メッセージング モデルでは恒久サブスクライバがサポートされるので、トピック サブスクライバに名前を割り当て、ユーザまたはアプリケーションと関連付けることができます。恒久サブスクライバの詳細については、4-57 ページの「恒久サブスクリプションの設定」を参照してください。

メッセージの永続性

メッセージは、永続メッセージまたは非永続メッセージとして指定できます。詳細については、JMS 仕様の「Message Delivery Mode」の節を参照してください。

- 永続メッセージは必ず 1 回のみ配信されます。したがって、メッセージが失われたり、2 回配信されたりすることはありません。永続メッセージは、ファイルやデータベースに確実に書き込まれるまでは送信されたものとは見なされません。WebLogic JMS では、コンフィグレーション時に各 JMS サーバに割り当てられた永続バッキング ストア (ディスクベースのファイルまたは JDBC でアクセス可能なデータベース) に永続メッセージが書き込まれます。

- 非永続メッセージは格納されません。メッセージは最低 1 回は配信が保証されますが、システム障害が発生すると失われる場合があります。接続を閉じるか回復すると、確認応答されていないすべての非永続メッセージが再配信されます。非永続メッセージは確認応答されると再配信されません。

WebLogic JMS のクラス

JMS アプリケーションを作成するには、javax.jms API を使用します。この API では、JMS への接続やメッセージの送受信に必要なクラス オブジェクトを作成できます。JMS クラス インタフェースは、共通の親クラスのキュー バージョンとトピック バージョンを提供するサブクラスとして作成されます。

次の表は、以降の節で詳しく説明する JMS クラスを示しています。すべての JMS クラスの詳細については、javax.jms、weblogic.jms.ServerSessionPoolFactory、または weblogic.jms.extensions Javadoc を参照してください。

表 2-1 WebLogic JMS のクラス

JMS クラス	説明
ConnectionFactory	接続のコンフィグレーション情報をカプセル化する。接続ファクトリは接続を作成するために使用する。接続ファクトリは JNDI を使用してルックアップする。
Connection	メッセージング システムへの開いている通信チャネルを表す。接続はセッションを作成するために使用する。
Session	生成および消費されるメッセージの順序を定義する。
Destination	特定のプロバイダのアドレスをカプセル化して、キューまたはトピックを識別する。キューとトピックでは、それぞれ PTP メッセージング モデルおよび Pub/sub メッセージング モデルから配信されるメッセージが管理される。

表 2-1 WebLogic JMS のクラス

JMS クラス	説明
MessageProducer と MessageConsumer	メッセージを送信および受信するためのインタフェースを提供する。メッセージ プロデューサではキューまたはトピックにメッセージが送信される。メッセージ コンシューマではキューまたはトピックからメッセージが受信される。
Message	送信または受信される情報をカプセル化する。
ServerSessionPoolFactory[1]	メッセージ コンシューマのサーバ管理のプールに関するコンフィグレーション情報をカプセル化する。サーバ セッション プール ファクトリはサーバ セッション プールを作成するために使用する。
ServerSessionPool[1]	メッセージを平行処理するために使用できるサーバ セッションのプールを接続コンシューマに提供する。
ServerSession[1]	スレッドと JMS セッションを関連付ける。
ConnectionConsumer[1]	メッセージを平行処理するためにサーバ セッションを取り出すコンシューマを指定する。

1 複数のメッセージを平行して処理するためのオプションの JMS インタフェースがサポートされます。

JMS オブジェクトのコンフィグレーションについては、3-1 ページの「WebLogic JMS の管理」を参照してください。JMS アプリケーションを設定する手順については、4-4 ページの「JMS アプリケーションの設定」を参照してください。

ConnectionFactory

ConnectionFactory オブジェクトは、接続のコンフィグレーション情報をカプセル化し、JMS アプリケーションが Connection を作成できるようにします。接続ファクトリでは同時使用がサポートされており、複数のスレッドがオブジェク

トに同時にアクセスできます。アプリケーションに適合する事前定義の属性で接続を作成するには、あらかじめコンフィグレーションされたデフォルトの接続ファクトリを使用するか、システム管理者がコンフィグレーションした 1 つまたは複数の接続ファクトリを使用します。

デフォルトの接続ファクトリは、JNDI 名 `weblogic.jms.ConnectionFactory` でルックアップできます。接続ファクトリは、WebLogic JMS で用意されるデフォルトがアプリケーションに適合しない場合にのみ定義する必要があります。すべてのデフォルトの接続ファクトリ属性は、ユーザ定義の接続ファクトリと同じデフォルト値に設定されます。接続ファクトリ属性のデフォルト値の詳細については、Administration Console オンライン ヘルプの「[JMS 接続ファクトリ]」を参照してください。

デフォルトの接続ファクトリを使用する場合のもう 1 つの注意点は、接続ファクトリがデプロイされる WebLogic Server インスタンスを限定できない点です。ただし、デフォルトの接続ファクトリは、サーバごとに有効にしたり無効にしたりできます。デフォルトの接続ファクトリの有効化および無効化の詳細については、Administration Console オンライン ヘルプの「サーバ --> サービス --> JMS」を参照してください。

デフォルトの接続ファクトリがアプリケーションに適合しない場合、システム管理者は 1 つまたは複数の接続ファクトリを定義およびコンフィグレーションして、あらかじめ定義された属性で接続を作成します。ただしこれは、各接続ファクトリにユニークな名前が付けられている場合にかぎります。WebLogic Server では起動時にそれらの接続ファクトリが JNDI スペースに追加されます。アプリケーションでは、WebLogic JNDI を使用して接続ファクトリを取り出します。ユーザ定義の接続ファクトリのコンフィグレーションおよびデプロイメントについては、『管理者ガイド』の「JMS の管理」を参照してください。

システム管理者は、クラスタ内のあらゆるサーバから JMS 送り先へのクラスタワイドで透過的なアクセスを確立できます。このようなアクセスを確立するには、クラスタ内の各サーバ インスタンスにデフォルトの接続ファクトリを使用するか、1 つまたは複数の接続ファクトリをコンフィグレーションしてクラスタ内の 1 つまたは複数のサーバ インスタンスに割り当てます。これにより、各接続ファクトリを複数の WebLogic Server にデプロイすることが可能となります。JMS クラスタ化の詳細については、3-3 ページの「WebLogic JMS のクラスタ化のコンフィグレーション」を参照してください。

注意： 下位互換性を維持するため、WebLogic JMS では非推奨の 2 つのデフォルト接続ファクトリを現在もサポートしています。該当するファクトリの JNDI 名は、`javax.jms.QueueConnectionFactory` と `javax.jms.TopicConnectionFactory` です。

非推奨の接続ファクトリから新しいデフォルトまたはユーザ定義の接続ファクトリへの移行については、6-1 ページの「WebLogic JMS アプリケーションの移植」を参照してください。

`ConnectionFactory` クラスではメソッドは定義されませんが、そのサブクラスでは各メッセージング モデルのメソッドが定義されます。接続ファクトリでは同時使用がサポートされており、複数のスレッドがオブジェクトに同時にアクセスできます。

次の表は、`ConnectionFactory` のサブクラスを説明しています。

表 2-2 ConnectionFactory のサブクラス

サブクラス	メッセージング モデル	作成するもの
`QueueConnectionFactory`	PTP	JMS PTP プロバイダへの `QueueConnection`
`TopicConnectionFactory`	Pub/sub	JMS Pub/sub プロバイダへの `TopicConnection`

アプリケーションで `ConnectionFactory` クラスを使用する方法については、4-1 ページの「WebLogic JMS アプリケーションの開発」または `javax.jms.ConnectionFactory` Javadoc を参照してください。

Connection

`Connection` オブジェクトは、アプリケーションとメッセージング システムの間の開いている通信チャネルを表し、メッセージを生成および消費するための `Session` を作成するために使用します。接続では、アプリケーションと JMS の間のメッセージングを管理するサーバサイドとクライアントサイドのオブジェクトが作成されます。接続では、ユーザの認証も提供されます。

Connection は、JNDI ルックアップを通じて取得する ConnectionFactory によって作成されます。

ユーザの認証や通信の設定に関わるリソースのオーバーヘッドがあるため、ほとんどのアプリケーションではすべてのメッセージングで 1 つの接続を確立します。WebLogic Server では、JMS トラフィックはサーバとのクライアント接続で他の WebLogic サービスと多重化されます。JMS のために、新たな TCP/IP 接続が作成されることはありません。サーブレットや他のサーバサイド オブジェクトもまた、JMS Connection を使用する場合があります。

デフォルトでは、接続は停止モードで作成されます。停止状態の接続をいつどのように開始するのかについては、4-49 ページの「接続の開始、停止、クローズ」を参照してください。

接続では同時使用がサポートされており、複数のスレッドがオブジェクトに同時にアクセスできます。

次の表は、Connection のサブクラスを説明しています。

表 2-3 Connection のサブクラス

サブクラス .	メッセージング モデル .	作成するもの
QueueConnection	PTP	QueueSession。QueueConnectionFactory で作成された JMS PTP プロバイダへの接続で構成される。
TopicConnection	Pub/sub	TopicSession。TopicConnectionFactory で作成された JMS Pub/sub プロバイダへの接続で構成される。

アプリケーションで Connection クラスを使用する方法については、4-1 ページの「WebLogic JMS アプリケーションの開発」または javax.jms.Connection Javadoc を参照してください。

Session

Session オブジェクトでは、生成および消費されるメッセージの順序を定義し、複数のメッセージプロデューサとメッセージ コンシューマを作成できます。メッセージの生成と消費に同じスレッドを使用できます。アプリケーションでメッセージの生成と消費に別々のスレッドが必要な場合は、そのアプリケーションで機能ごとに個別のセッションを作成する必要があります。

Session は、Connection によって作成されます。

注意： セッションおよびそのメッセージのプロデューサとコンシューマには、一度に 1 つのスレッドしかアクセスできません。それらに複数のスレッドが同時にアクセスした場合、それらの動作は定義されません。

次の表は、Session のサブクラスを説明しています。

表 2-4 Session のサブクラス

サブクラス .	メッセージング モデル	提供するコンテキストの用途
`QueueSession`	PTP	JMS PTP プロバイダのメッセージを生成および消費する。QueueConnection で作成される。
`TopicSession`	Pub/sub	JMS Pub/sub プロバイダのメッセージを生成および消費する。TopicConnection によって作成される。

アプリケーションで Session クラスを使用する方法については、4-1 ページの「WebLogic JMS アプリケーションの開発」、または `javax.jms.Session` Javadoc および `weblogic.jms.extensions.WLSession` Javadoc を参照してください。

非トランザクション セッション

非トランザクション セッションでは、セッションを作成するアプリケーションで、次の表で定義されている 5 つの確認応答モードのいずれかが選択されます。

表 2-5 非トランザクション セッションで使用する確認応答モード

[確認応答モード]	説明
AUTO_ACKNOWLEDGE	受信側アプリケーションのメソッドが処理を終えたときに、Session オブジェクトでメッセージ受信の確認応答が行われる。
CLIENT_ACKNOWLEDGE	Session オブジェクトの動作は、アプリケーションによる確認応答メソッドの呼び出しに依存する。メソッドが呼び出されると、セッションでは、前回の確認応答以降に受信されたすべてのメッセージに対して確認応答が行われる。 このモードを使用すると、アプリケーションでは 1 回の呼び出しで複数メッセージの受信、処理、および確認応答を行うことができる。 **注意 :** Administration Console では、接続ファクトリの [確認応答ポリシー] 属性が Previous に設定されているのに対し、指定のセッションでのすべての受信メッセージを確認応答したい場合、最後のメッセージを使用して確認応答メソッドを呼び出す。[確認応答ポリシー] 属性の詳細については、『Administration Console オンライン ヘルプ』の「[JMS 接続ファクトリ]」を参照。
DUPS_OK_ACKNOWLEDGE	受信側アプリケーションのメソッドが処理を終えたときに、Session オブジェクトでメッセージ受信の確認応答が行われる。確認応答の重複が許可される。 このモードでは、最も効率的にリソースが利用される。 **注意 :** アプリケーションで重複メッセージを処理できない場合は、このモードは使用しない。重複メッセージは、メッセージを配信する最初の試行が失敗した場合に送信される。

表 2-5 非トランザクション セッションで使用する確認応答モード (続き)

[確認応答モード]	説明
NO_ACKNOWLEDGE	確認応答を必要としない。NO_ACKNOWLEDGE セッションに送信されたメッセージは、サーバから即座に削除される。このモードで受信されたメッセージは回復されないので、メッセージを配信する最初の試行が失敗した場合はメッセージが失われたり、重複メッセージが配信されたりする。 このモードは、セッションの確認応答で提供されるサービスの質を必要とせず、それに関連するオーバーヘッドを避ける必要があるアプリケーションで使用する。 **注意 :** アプリケーションで、失われたメッセージや重複メッセージを処理できない場合は、このモードは使用しない。重複メッセージは、メッセージを配信する最初の試行が失敗した場合に送信される。
MULTICAST_NO_ACKNOWLEDGE	確認応答を必要としないマルチキャスト モード。 MULTICAST_NO_ACKNOWLEDGE セッションに送信されたメッセージでは、前述の NO_ACKNOWLEDGE モードと同じ特性が共有される。 このモードは、マルチキャストをサポートし、セッションの確認応答で提供されるサービスの質を必要としないアプリケーションで使用する。マルチキャストの詳細については、4-87 ページの「マルチキャストの使い方」を参照。 **注意 :** アプリケーションで、失われたメッセージや重複メッセージを処理できない場合は、このモードは使用しない。重複メッセージは、メッセージを配信する最初の試行が失敗した場合に送信される。

トランザクション セッション

トランザクション セッションでは、一度に 1 つのトランザクションしかアクティブになりません。トランザクション時に送信または受信されたメッセージは、最小の単位として処理されます。

トランザクション セッションを作成すると、確認応答モードは無視されます。アプリケーションでトランザクションがコミットされると、そのトランザクションの間にアプリケーションで受信されたすべてのメッセージがメッセージング システムによって確認応答され、アプリケーションで送信されたメッセージは配信されるべく受け入れられます。アプリケーションでトランザクションがロールバックされると、そのトランザクションの間にアプリケーションで受信されたメッセージは確認応答されず、アプリケーションで送信されたメッセージは破棄されます。

JMS は、Java Transaction API (JTA) を使用する他の Java サービス (EJB など) との分散トランザクションに参加できます。トランザクションはそのトランザクションに関連付けられたメッセージへのアクセスが制限されているため、トランザクション セッションではこの機能をサポートしません。JTA の利用の詳細については、5-5 ページの「JTA ユーザ トランザクションの使い方」を参照してください。

Destination

`Destination` オブジェクトはキューまたはトピックになり、特定プロバイダのアドレス構文をカプセル化します。プロバイダによって構文はさまざまであるため、JMS 仕様では標準のアドレス構文は定義されていません。

接続ファクトリの場合と同じように、管理者が送り先を定義し、コンフィグレーションすると、WebLogic Server の起動時にそれが JNDI スペースに追加されます。また、アプリケーションでは、それが作成された JMS 接続の間だけ存在する一時的な送り先を作成することもできます。

注意： 管理者は、サーバのクラスタ内の単一の分散送り先セットのメンバーとして、複数の物理的な送り先をコンフィグレーションすることもできます。詳細については、2-15 ページの「送り先の分散」を参照してください。

クライアントサイドでは、`Queue` オブジェクトと `Topic` オブジェクトは、サーバ上のオブジェクトへのハンドルとして機能します。それらのメソッドでは、それらの名前が返されるだけです。メッセージングを目的としてアクセスするには、それらにアタッチするメッセージ プロデューサとメッセージ コンシューマを作成します。

送り先では同時使用がサポートされており、複数のスレッドがオブジェクトに同時にアクセスできます。JMS の Queue と Topic は、javax.jms.Destination を拡張します。次の表は、Destination のサブクラスを説明しています。

表 2-6 Destination のサブクラス

サブクラス	メッセージング モデル	メッセージング モデル
Queue	PTP	JMS PTP プロバイダのメッセージ。
TemporaryQueue	PTP	JMS PTP プロバイダのメッセージ。メッセージが作成された JMS 接続の間だけ存在する。一時的なキューはそれを作成したキュー接続によってのみ消費される。
Topic	Pub/sub	JMS Pub/sub プロバイダのメッセージ。
TemporaryTopic	Pub/sub	JMS Pub/sub プロバイダのメッセージ。メッセージが作成された JMS 接続の間だけ存在する。一時的なトピックはそれを作成したトピック接続によってのみ消費される。

注意： アプリケーションでは、キュー セッションで QueueBrowser オブジェクトを作成することによりキューを参照することができます。このオブジェクトでは、キュー ブラウザが作成された時点におけるキュー内のメッセージのスナップショットが生成されます。アプリケーションではキュー内のメッセージを表示できますが、メッセージは「読み込まれた」とは判断されず、したがってキューから削除されることはありません。キューの参照の詳細については、4-69 ページの「メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの参照」を参照してください。

アプリケーションで Destination クラスを使用する方法については、4-1 ページの「WebLogic JMS アプリケーションの開発」または javax.jms.Destination Javadoc を参照してください。

送り先の分散

管理者は、WebLogic Server のクラスタ内の単一の分散送り先セットのメンバーとして、複数の物理的な送り先をコンフィグレーションすることができます。適切に構成すると、プロデューサとコンシューマがその分散送り先に対して送受信できるようになります。WebLogic JMS は、分散送り先内で利用可能な全送り先メンバーにメッセージの負荷を分散します。

- ご使用のアプリケーションにおける送り先の分散の詳細については、4-94 ページの「分散送り先の使用」を参照してください。
- Administration Console を使用して分散送り先をコンフィグレーションする手順については、『管理者ガイド』の「分散送り先のコンフィグレーション」を参照してください。

MessageProducer と MessageConsumer

`MessageProducer` オブジェクトでは、メッセージがキューまたはトピックに送信されます。`MessageConsumer` オブジェクトでは、メッセージがキューまたはトピックから受信されます。メッセージプロデューサとメッセージ コンシューマは、互いに独立して機能します。メッセージ コンシューマが作成されてメッセージを待っているかどうかに関係なく、メッセージ プロデューサではメッセージが生成および送信されます (この逆も同じ)。

`Session` では、キューおよびトピックにアタッチされる `MessageProducer` と `MessageConsumer` が作成されます。

メッセージ センダ オブジェクトとメッセージ レシーバ オブジェクトは、MessageProducer クラスおよび MessageConsumer クラスのサブクラスとして作成されます。次の表は、MessageProducer と MessageConsumer のサブクラスを説明しています。

表 2-7 MessageProducer と MessageConsumer のサブクラス

サブクラス	メッセージング モデル	機能
QueueSender	PTP	JMS PTP プロバイダのメッセージを送信する。
QueueReceiver	PTP	JMS PTP プロバイダのメッセージを受信し、メッセージの作成された JMS 接続が閉じるまで存続する。
TopicPublisher	Pub/sub	JMS Pub/sub プロバイダのメッセージを送信する。
TopicSubscriber	Pub/sub	JMS Pub/sub プロバイダのメッセージを受信し、メッセージの作成された JMS 接続が閉じるまで存続する。メッセージの送り先は、適切な JNDI インタフェースを使用して明示的にバインドしなければならない。

2-3 ページの「ポイント ツー ポイント (PTP) メッセージング」の図で示されているように、PTP モデルでは複数のセッションが同じキューからメッセージを受信できます。ただし、メッセージは、1 つのキュー レシーバにしか配信できません。複数のキュー レシーバがある場合、次にメッセージを受信するキュー レシーバは先着順で決まります。

2-4 ページの「パブリッシュ / サブスクライブ (Pub/sub) メッセージング」の図で示されているように、Pub/sub モデルではメッセージを複数のトピック サブスクライバに配信できます。トピック サブスクライバは、4-57 ページの「恒久サブスクリプションの設定」で説明されているように恒久にも非恒久にもなります。

アプリケーションでは、1 つのトピックのパブリッシュとサブスクライブに同じ JMS 接続を使用できます。トピック メッセージはすべてのサブスクライバに配信されるので、アプリケーションは自身がパブリッシュしたメッセージを受信する可能性があります。メッセージがパブリッシュ元のクライアントで受信される

のを防ぐために、JMS アプリケーションではトピック サブスクライバに対して noLocal 属性を設定できます。詳細については、4-12 ページの「手順 5 : セッションと送り先を使用してメッセージ プロデューサとメッセージ コンシューマを作成する」を参照してください。

アプリケーションで MessageProducer クラスと MessageConsumer クラスを使用する方法については、4-4 ページの「JMS アプリケーションの設定」、または javax.jms.MessageProducer Javadoc および javax.jms.MessageConsumer Javadoc を参照してください。

Message

Message オブジェクトでは、アプリケーション間で交換される情報がカプセル化されます。この情報は、標準ヘッダ フィールド、アプリケーション固有のプロパティ、およびメッセージ本文という 3 つの要素で構成されます。以降の節では、これらの構成要素について説明します。

メッセージ ヘッダ フィールド

すべての JMS メッセージには、デフォルトで挿入され、メッセージ コンシューマで利用できる標準のヘッダ フィールドがあります。一部のフィールドは、メッセージ プロデューサで設定できます。

メッセージ ヘッダ フィールドの設定については、4-62 ページの「メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの設定と参照」または javax.jms.Message Javadoc を参照してください。

次の表は、メッセージ ヘッダのフィールドを説明し、各フィールドで値がどのように定義されるのかを示しています。

表 2-8 メッセージ ヘッダ フィールド

フィールド	説明	どこで定義されるか
JMSCorrelationID	WebLogic JMSMessageID (この表内で後述)、アプリケーション固有の文字列、または byte[] 配列のいずれかを指定する。JMSCorrelationID はメッセージを相互に関連させるために使用する。 このフィールドには 2 つの一般的な用途がある。 最初の用途は、次のような要求と応答の方式を設定してメッセージをリンクすること。 1. メッセージを送信するときに、アプリケーションではそのメッセージに割り当てられた JMSMessageID 値を格納する。 2. そのメッセージを受信したアプリケーションでは、送信側アプリケーションに送り返す応答メッセージの JMSCorrelationID フィールドに JMSMessageID をコピーする。 2 番目の用途は、選択した文字列を JMSCorrelationID フィールドに格納し、一連のメッセージをアプリケーション指定の値でリンクできるようにすること。 すべての JMSMessageID は ID: というプレフィックスで始まる。他のアプリケーション固有の文字列で JMSCorrelationID を使用する場合は、文字列の先頭にプレフィクス ID: が付いてはならない。	アプリケーション

表 2-8 メッセージ ヘッダ フィールド (続き)

フィールド	説明	どこで定義されるか
JMSDeliveryMode	PERSISTENT または NON_PERSISTENT を指定する。 永続メッセージが送信された場合、そのメッセージは JMS ファイルまたは JDBC データベースに格納される。send() 処理は、メッセージの配信を保証できるまで成功とは判断されない。永続メッセージは少なくとも 1 回は確実に配信される。 非永続メッセージは JMS データベースに格納されない。このモードでは、処理のオーバーヘッドが最小限に抑えられる。メッセージは最低 1 回は配信が保証されるが、システム障害が発生すると失われる場合がある。接続を閉じるか回復すると、確認応答されていないすべての非永続メッセージが再配信される。非永続メッセージは確認応答されると再配信されない。 メッセージが送信されるときには、この値は無視される。メッセージが受信されるときには、送信側のメソッドで指定された配信モードが格納されている。	send() メソッド
JMSDeliveryTime	メッセージをコンシューマに配信できる最も早い絶対時間を定義する。このフィールドは、送り先でのメッセージのソート、またはメッセージの選択に使用できる。データ型変換の目的で、JMSDeliveryTime は長整数として保存される。	send() メソッド
JMSDestination	メッセージが配信される送り先 (キューまたはトピック) を指定する。このフィールドの値は、メッセージの送信時にアプリケーションのメッセージ プロデューサで設定される。 メッセージが送信されるときには、この値は無視される。メッセージが受信されるとき、その送り先の値は送信時に割り当てられた値と同じでなければならない。	send() メソッド

表 2-8 メッセージ ヘッダ フィールド (続き)

フィールド	説明	どこで定義されるか
JMSExpiration	メッセージの有効期限 (存続時間値) を指定する。 JMSExpiration の値は、アプリケーションの存続時間とその時点の GMT の合計として算出される。アプリケーションで存続時間が 0 として指定されると、JMSExpiration は 0 に設定され、メッセージは無期限になる。 期限の切れたメッセージは、誤って配信されないようにシステムから削除される。	send() メソッド
JMSMessageID	JMS プロバイダによって送信される各メッセージをユニークに識別する文字列値を格納する。 すべての JMSMessageID は ID: というプレフィックスで始まる。 メッセージが送信されるときには、この値は無視される。メッセージが受信されるときには、プロバイダの割り当てた値が格納されている。	send() メソッド
JMSPriority	優先順位のレベルを指定する。このフィールドはメッセージが送信される前に設定される。 JMS では、0 ～ 9 の 10 段階で優先順位が定義されている (0 が最低の優先順位)。レベル 0 ～ 4 は通常の範囲に属し、レベル 5 ～ 9 は至急の範囲に属する。 メッセージが受信されるときには、メッセージを送信するメソッドで指定された値が格納されている。 送り先キーをコンフィグレーションすれば、優先順位を基準に送り先をソートすることができる。詳細については、『管理者ガイド』の「JMS の管理」を参照。	send() メソッド

表 2-8 メッセージ ヘッダ フィールド (続き)

フィールド	説明	どこで定義されるか
JMSRedelivered	確認応答がないためメッセージが再配信されるときに設定されるフラグを指定する。このフラグは受信側アプリケーションのみに関係がある。 フラグが設定されている場合は、以下のいずれかの理由のために、同じメッセージが以前に配信されている可能性がある。 ■ アプリケーションでは既にメッセージが受信されているが、確認応答が行われていない。 ■ セッションの recover() メソッドが呼び出されて、最後に確認応答されたメッセージの後からセッションが再開された。recover() メソッドの詳細については、4-35 ページの「受信メッセージの回復」を参照。	WebLogic JMS
JMSReplyTo	応答メッセージが送信されるキューまたはトピックを指定する。このフィールドはメッセージの送信前に送信側アプリケーションで設定される。 この機能は JMSCorrelationID ヘッダ フィールドと共に使用して要求と応答のメッセージを連係させることができる。 JMSReplyTo フィールドをただ設定するだけでは応答は保証されない。受信側アプリケーションに応答させるには、そのように選択する必要がある。 JMSReplyTo は NULL に設定することもできる。それは、受信側アプリケーションにとって通知イベントなどの意味を持つ場合がある。	アプリケーション
JMSTimestamp	メッセージが送信されたときの時刻を格納する。タイムスタンプは、アプリケーションでメッセージが送信されたときではなく、WebLogic JMS で配信用にメッセージが受け付けられたときにメッセージに書き込まれる。 メッセージが受信されるときには、タイムスタンプが格納されている。 このフィールドには、Java のミリ時間の値が格納される。	WebLogic JMS

表 2-8 メッセージ ヘッダ フィールド (続き)

フィールド	説明	どこで定義されるか
JMSType	送信側アプリケーションで設定されたメッセージ タイプ識別子 (String) を示す。 JMS 仕様では、多様な JMS プロバイダに適応するため、このフィールドに若干の柔軟性を持たせている。一部のメッセージング システムでは、アプリケーション固有のメッセージ タイプを使用できる。そのようなシステムの場合、JMSType フィールドは、格納されている型定義にアクセスするためのメッセージ タイプ ID を保持するために使用できる。 WebLogic JMS では、このフィールドの使用に制限を設けていない。	アプリケーション

メッセージ プロパティ フィールド

メッセージのプロパティ フィールドには、送信側アプリケーションによって追加されたヘッダ フィールドが格納されます。プロパティは、標準的な Java の名前と値の組み合わせです。プロパティ名は、javax.jms.Message Javadoc で定義されているメッセージ セレクタの構文仕様に準拠していなければなりません。有効な値は、boolean、byte、double、float、int、long、および String です。

メッセージ プロパティ フィールドは、アプリケーション固有の目的に使用できますが、それらは基本的にはメッセージ セレクタで使用するために用意されています。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

メッセージ プロパティ フィールドの設定については、4-62 ページの「メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの設定と参照」または javax.jms.Message Javadoc を参照してください。

メッセージ本文

メッセージ本文は、プロデューサからコンシューマに配信される内容です。

次の表は、JMS で定義されているメッセージ タイプを説明しています。すべてのメッセージ タイプは、メッセージ ヘッダとメッセージ プロパティ (メッセージ本文はなし) で構成される `javax.jms.Message` を拡張します。

表 2-9 JMS メッセージ タイプ

タイプ	説明
javax.jms.BytesMessage	未解釈バイトのストリーム。センダとレシーバによって理解されなければならない。このメッセージ タイプのアクセス メソッドは、`java.io.DataInputStream` および `java.io.DataOutputStream` に基づくストリーム対応のリーダとライタ。
javax.jms.MapMessage	名前が文字列であり、値が Java プリミティブ型である、複数の名前と値の組み合わせ。名前と値の組み合わせは、名前を指定することによって順次的にもランダムにも読み込むことができる。
javax.jms.ObjectMessage	1 つのシリアライズ可能な Java オブジェクト。
javax.jms.StreamMessage	Java プリミティブ型だけがストリームで読み書きされること以外は、BytesMessage と同じ。
javax.jms.TextMessage	1 つの String。TextMessage では XML コンテンツも格納できる。
weblogic.jms.extensions.XMLMessage	XML コンテンツ。XMLMessage タイプを使用すると、TextMessage で送信される XML コンテンツでは処理しにくいメッセージのフィルタ処理を容易に行うことができる。

詳細については、`javax.jms.Message` Javadoc を参照してください。特定のメッセージ タイプのアクセス メソッドや変換表については、そのメッセージ タイプの Javadoc を参照してください。

ServerSessionPoolFactory

サーバ セッション プールは、メッセージの並行処理を実現する WebLogic 固有の JMS 機能です。サーバ セッション プール ファクトリは、サーバサイドの `ServerSessionPool` を作成するために使用します。

WebLogic JMS では、デフォルトで次のような `ServerSessionPoolFactory` オブジェクトが定義されています。`weblogic.jms.ServerSessionPoolFactory:<name>`。ここで <*name*> には、セッション プールの作成先になる JMS サーバの名前を指定します。WebLogic Server ではデフォルトのサーバ セッション プール ファクトリが起動時に JNDI スペースに追加され、アプリケーションでは WebLogic JNDI を使用してサーバ セッション プール ファクトリが取り出されます。

アプリケーションでサーバ セッション プール ファクトリを使用する方法については、4-76 ページの「サーバ セッション プールの定義」または `weblogic.jms.ServerSessionPoolFactory` Javadoc を参照してください。

ServerSessionPool

`ServerSessionPool` アプリケーション サーバ オブジェクトでは、メッセージを並行処理するために接続コンシューマで取り出すことができるサーバ セッションのプールが提供されます。

`ServerSessionPool` は、JNDI ルックアップで取得される `ServerSessionPoolFactory` オブジェクトによって作成されます。

アプリケーションでサーバ セッション プールを使用する方法については、4-76 ページの「サーバ セッション プールの定義」または `javax.jms.ServerSessionPool` Javadoc を参照してください。

ServerSession

ServerSession アプリケーション サーバ オブジェクトでは、メッセージを作成、送信、および受信するためのコンテキストが提供され、スレッドと JMS セッションを関連付けることができます。

ServerSession は、ServerSessionPool オブジェクトによって作成されます。

アプリケーションでサーバ セッションを使用する方法については、4-76 ページの「サーバ セッション プールの定義」または javax.jms.ServerSession Javadoc を参照してください。

ConnectionConsumer

ConnectionConsumer オブジェクトでは、サーバ セッションを使用して受信メッセージを処理します。メッセージ トラフィックが大きい場合は、スレッド コンテキストの切り替えを最小限に抑えるために、接続コンシューマでは複数のメッセージで各サーバ セッションをロードすることができます。

ConnectionConsumer は、Connection オブジェクトによって作成されます。

アプリケーションで接続コンシューマを使用する方法については、4-76 ページの「サーバ セッション プールの定義」または javax.jms.ConnectionConsumer Javadoc を参照してください。

注意： 接続コンシューマ リスナは、サーバと同じ JVM で動作します。

3 WebLogic JMS の管理

Administration Console は、JMS を含む WebLogic Server の機能を有効化、コンフィグレーション、およびモニタするためのインタフェースを備えています。Administration Console を起動する手順については、『管理者ガイド』の「Administration Console の起動と使い方」を参照してください。

以下の節では、WebLogic JMS のコンフィグレーションおよびモニタについて概説します。

- WebLogic JMS のコンフィグレーション
- WebLogic JMS のクラスタ化のコンフィグレーション
- JMS 移行可能対象のコンフィグレーション
- WebLogic JMS のチューニング
- WebLogic JMS のモニタ
- WebLogic Server の障害からの回復

WebLogic JMS のコンフィグレーション

Administration Console を使用して、以下のコンフィグレーション属性を定義します。

- JMS を有効にします。
- JMS サーバを作成します。
- JMS サーバの値、接続ファクトリ、送り先 (キューとトピック)、分散送り先 (クラスタ内の物理的キューとトピック メンバーのセット)、送り先テンプレート、(送り先キーを使用した) 送り先のソート順指定、永続ストレージ、

ページング ストア、セッション プール、および接続コンシューマを作成またはカスタマイズします。

- カスタム JMS アプリケーションを設定します。
- しきい値と割当を定義します。
- 必要な JMS 機能を有効にします。
  - サーバのクラスタ化
  - メッセージの同時処理
  - 永続的なメッセージング
  - メッセージ ページング
  - フロー制御

WebLogic JMS では、一部のコンフィグレーション属性に対して、デフォルト値が用意されていますが、それ以外のすべての属性に対しては値を指定する必要があります。コンフィグレーション属性に対して無効な値を指定した場合や、デフォルト値が存在しない属性に対して値を指定しなかった場合は、再起動時に JMS が起動されません。製品には、JMS のサンプル コンフィグレーションが用意されています。

以前のリリースから移行する場合、コンフィグレーション情報は自動的に変換されます (6-9 ページの「既存のアプリケーションの移植」を参照)。

注意 : 付録 A「コンフィグレーション チェックリスト」にあるチェックリストでは、各種 JMS 機能をサポートするための必須属性やオプション属性を確認できます。

WebLogic JMS のクラスタ化のコンフィグレーション

WebLogic Server のクラスタはより強力で、より信頼性のあるアプリケーションプラットフォームを提供するためのサーバ群です。クラスタはそのクライアントにとって単一のサーバに見えますが、実際には、一体で機能するサーバ群です。クラスタは、単一のサーバを超える下記の 3 つの重要な機能を提供します。

- スケーラビリティ - サーバをクラスタに動的に追加して能力を増大させることができます。
- 高可用性 - 複数サーバの冗長性によってアプリケーションが障害から保護されます。クラスタ内の複数の物理的送り先 (キューとトピック) を単一の分散送り先セットのメンバーとする冗長性によって、1 メンバーが使用できなくなった場合でも他の使用可能なメンバーにメッセージの負荷が確実に再分配されます。
- 移行性 - 移行リクエストに適切に応答し、JMS サーバを正しい順序でオンラインまたはオフラインにすることができます。これには、スケジュールされた移行と WebLogic Server の障害への応答としての移行の両方が含まれます。

注意： JMS クライアントは、WebLogic Server が異なるドメインに存在する場合でも、ユニークな WebLogic Server 名に依存してクラスタにアクセスします。そのため、JMS クライアントがアクセスするすべての WebLogic Server にはユニークなサーバ名を付ける必要があります。

WebLogic クラスタおよびその機能と利点の詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「クラスタのコンフィグレーションとアプリケーションのデプロイメント」を参照してください。

JMS クラスタ化の仕組み

クラスタ内のあらゆるサーバから JMS 送り先へのクラスタワイドで透過的なアクセスを確立するには、クラスタ内の各サーバ インスタンスにデフォルトの接続ファクトリを使用するか、1 つまたは複数の接続ファクトリをコンフィグレー

ションしてクラスタ内の 1 つまたは複数のサーバ インスタンスに割り当てます。これにより、各接続ファクトリを複数の WebLogic Server にデプロイすることが可能となります。ただし、複数の WebLogic Server に正常にデプロイするには、ユーザ定義の各接続ファクトリの名前がユニークでなければなりません。

管理者は、クラスタ内のさまざまなノード上の複数の JMS サーバにユニークな名前が付けられているかぎり、それらの JMS サーバをコンフィグレーションして JMS 送り先を割り当てることができます。

アプリケーションでは、Java Naming and Directory Interface (JNDI) を使用して接続ファクトリをルックアップし、JMS サーバとの通信を確立するための接続を作成します。各 JMS サーバでは、複数の送り先に対するリクエストが処理されます。JMS サーバで処理されない送り先へのリクエストは、適切な WebLogic Server に転送されます。

JMS クラスタ化のネーミング要件

JMS クライアントは、WebLogic Server が異なるドメインに存在する場合でも、ユニークな WebLogic Server 名に依存してクラスタにアクセスします。そのため、JMS クライアントがアクセスするすべての WebLogic Server にはユニークなサーバ名を付ける必要があります。

以下のユニークなネーミング要件は、単一の WebLogic ドメイン環境またはマルチドメイン環境におけるクラスタ環境で機能するように WebLogic JMS をコンフィグレーションする場合に適用されます。

- ドメイン内のノードの対象になるすべての JMS 接続ファクトリにはユニークな名前を付ける必要がある。
- すべてのドメイン内のノードの対象になるすべての JMS サーバにはユニークな名前を付ける必要がある。
- 永続的なメッセージングが必要な場合は、すべてのドメイン内のすべての JMS ストアにユニークな名前を付ける必要がある。

単一のドメイン環境およびマルチドメイン環境における JMS リソースの命名規則の詳細については、『管理者ガイド』の「ドメインの相互運用性を実現するための JMS リソースの命名規則」を参照してください。

クラスタ内での JMS 分散送り先

管理者は、単一の分散送り先の一部としてクラスタ内に複数の JMS 送り先をコンフィグレーションすることもできます。分散送り先は、単一の JNDI 名でアクセスされ、JMS クライアントからは単一の論理的な送り先に見える一連のキューまたはトピックのセットです。分散送り先のメンバーは、実際にはクラスタ内の複数のサーバに分散されており、各送り先メンバーは個々の JMS サーバに属しています。クラスタ内のサーバ 1 台に障害が発生した場合は、分散送り先の他の使用可能なメンバー全体にメッセージ負荷が分散されます。分散送り先のコンフィグレーションの詳細については、『管理者ガイド』の「分散送り先のコンフィグレーション」を参照してください。

クラスタ内での移行可能なサービスとしての JMS

WebLogic JMS は、クラスタ環境用の WebLogic Server コアに実装されている移行フレームワークを利用します。これにより、WebLogic JMS は、移行リクエストに適切に応答し、JMS サーバを正しい順序でオンラインまたはオフラインにすることができます。これには、スケジューリングされた移行だけでなく、WebLogic Server の障害への対応としての移行も含まれます。詳細については、3-7 ページの「JMS 移行可能対象のコンフィグレーション」を参照してください。

JMS クラスタ化のコンフィグレーションのガイドライン

クラスタ環境で WebLogic JMS を使用するには、次のガイドラインに従います。

1. 『WebLogic Server クラスタ ユーザーズ ガイド』の「WebLogic クラスタの設定」にある説明に従って、クラスタ環境をコンフィグレーションします。
2. Administration Console を使用して JMS 接続ファクトリの対象サーバを識別します。接続ファクトリでは、単一サーバの対象とクラスタの対象を識別できます。クラスタの対象とは、クラスタ化をサポートするために接続ファクトリに関連付けられたサーバ インスタンスです。

これらの接続ファクトリのコンフィグレーション属性の詳細については、『管理者ガイド』の「接続ファクトリのコンフィグレーション」を参照してください。

3. 必要に応じて、Administration Console を使用して JMS サーバの移行可能な対象サーバを識別します。JMS サーバでは、単一サーバの対象と移行可能な対象を識別できます。移行可能な対象とは、クラスタ内にあるサーバ インスタンスのセットで、1 台のサーバに障害が発生した場合に、JMS のような「1 回限りの」サービスをホストできます。

   JMS サーバの移行可能な対象の詳細については、3-7 ページの「JMS 移行可能対象のコンフィグレーション」を参照してください。JMS サーバのコンフィグレーション属性の詳細については、『管理者ガイド』の「JMS サーバのコンフィグレーション」を参照してください。

   注意： 同じ送り先を複数の JMS サーバにデプロイすることはできません。また、1 つの JMS サーバを複数の WebLogic Server にデプロイすることもできません。

4. 必要に応じて、単一の分散送り先セットの一部としてクラスタ内に物理的な JMS 送り先をコンフィグレーションすることができます。3-5 ページの「クラスタ内での JMS 分散送り先」を参照してください。

フェイルオーバの場合

WebLogic 7.0 のクラスタ環境の一部である WebLogic JMS の実装では、JMS は、単一の WebLogic Server に障害が発生した場合、単一の分散送り先セットの一部として複数の物理的送り先 (キューとトピック) をコンフィグレーションすることで、サービスを継続します。さらに、移行可能サービスの機能を実装すると、JMS のような固定された 「1 回限りの」サービスによって、クラスタ内にある依存するアプリケーションに対してシングル ポイント障害が発生しなくなります。

ただし、自動フェイルオーバは現在 WebLogic JMS でサポートされていません。手動フェイルオーバの実行の詳細については、3-12 ページの「WebLogic Server の障害からの回復」を参照してください。

JMS 移行可能対象のコンフィグレーション

WebLogic JMS は「1 回限りの」サービスなので、クラスタ内のすべての WebLogic Server のインスタンスに対してアクティブというわけではありません。代わりに、データの一貫性を保持するためにクラスタ内の単一サーバに「固定」されます。固定されたサービスにより、クラスタ内にある依存するアプリケーションに対してシングル ポイント障害が発生しないように、移行できる対象リスト内のあらゆるサーバをコンフィグレーションして、1 回だけサービスを移行するようにできます。

WebLogic JMS では、Administration Console の説明にあるとおり、移行フレームワークを活用して、管理者が JMS サーバに対する移行できる対象を指定できます。いったん適切にコンフィグレーションすると、JMS サーバとその送り先すべてを別の WebLogic Server へ移行することができます。

これにより、WebLogic JMS は、移行リクエストに適切に応答し、JMS サーバを正しい順序でオンラインまたはオフラインにすることができます。移行には、スケジューリング済み移行のほかに、クラスタ内の WebLogic Server の障害に応答して発生する移行が含まれます。

移行できる対象の定義の詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「固定サービスの移行」を参照してください。

JMS 移行の仕組み

WebLogic のクラスタ環境の一部として実装する場合、WebLogic JMS ではサーバ移行フレームワークが実装され、これによって JMS サーバがアクティブ化リクエストおよび非アクティブ化リクエストに応答できるようになります。

表 3-1 WebLogic JMS の移行手順

移行状況	起こること
初期化	JMS サーバの初期化には、あらゆるコンフィグレーション情報またはデプロイメント情報の処理、および適切なオブジェクトの作成などの作業が含まれる。この時点では、送り先および JMS リソースは利用できない。さらに、永続ストレージの完全性を低下させる可能性があるので、永続ストレージはオープンされていない。JMS サーバでは、初期化とアクティベーションの間で発生するコンフィグレーションの変更を処理する準備ができている。
アクティベーション	JMS サーバがアクティブになると、永続ストレージがオープンし、必要なあらゆる回復が実行され、ストレージの内容と現在のコンフィグレーションとの整合性がとられ、アプリケーションから送り先にアクセスできるようになる。さらに、アクティベーションが完了すると、コンフィグレーションされたあらゆるサーバ セッション プールの処理が開始される。
非アクティベーション	JMS サーバが非アクティブ化されると、サーバ セッション プールの処理がすべて停止され、すべての送り先が利用不可となり、永続ストレージがフラッシュおよびクローズされ、送り先がパージされ、JMS サーバのすべてのオブジェクトが削除される。

JMS サーバ移行のコンフィグレーション手順

クラスタ環境で JMS サーバを移行可能なサービスとするには、以下の作業が必要です。

1. 必要に応じて、『WebLogic Server クラスタ ユーザーズ ガイド』の「固定サービスの移行」を読み、固定サービスのサーバの移行の仕組みを理解します。
2. 永続的なメッセージングを使用する JMS 実装では、移行可能な対象内にあるすべての候補サーバがストアへのアクセスを共有するように、JMS ストアをコンフィグレーションするようにしてください。JMS ストアの移行の詳細については、3-9 ページの「永続ストレージの移行」を参照してください。
3. 『WebLogic Server クラスタ ユーザーズ ガイド』の「固定サービスの移行可能対象をコンフィグレーションする」にある説明に従って、JMS サーバのホストとなることができる、クラスタの移行可能な対象サーバをコンフィグレーションします。

   注意： 移行された JMS サーバをホストする移行可能な対象サーバ インスタンス用にユニークなリスン アドレス値を設定する必要があります。そうしないと、移行は失敗します。
4. 『WebLogic Server クラスタ ユーザーズ ガイド』の「移行可能対象のサーバ インスタンスに JMS をデプロイする」にある説明に従って、JMS サーバをデプロイする移行可能な対象サーバ インスタンスを識別します。

   注意： 移行可能な対象サーバが起動すると、クラスタ内のユーザが選んだサーバ上で、JMS サーバも自動的に起動します。
5. サーバ メンテナンスを実行する前やホスト サーバに障害が発生した場合に、問題のないサーバに JMS サーバとそのすべての送り先を手動で移行できます。詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「対象サーバ インスタンスに固定サービスを移行する」を参照してください。

   注意： 対象のサーバ インスタンスがすでに JMS サーバとその分散送り先メンバーをホストしているときでも、JMS サーバの分散送り先メンバーは、クラスタ内の別のサーバ インスタンスへ移行できます。分散送り先のフェイルオーバの詳細については、4-110 ページの「分散送り先のフェイルオーバ」を参照してください。

永続ストレージの移行

Weblogic JMS 永続ストレージは、JMS サーバとともに移行できません。したがって、JMS サーバを移行した後にほかの物理マシンから永続ストレージへのアクセスを必要とするアプリケーションでは、次のように代替のソリューションを実装する必要があります。

- デュアル ポート SCSI ディスクまたはストレージ エリア ネットワーク (SAN) などのハードウェア ソリューションを実装して、他のマシンから JMS 永続ストレージが利用できるようにする。
- JDBC を使用して JMS JDBC ストアにアクセスする。JMS JDBC ストアは別のサーバ上にある可能性もあります。その後、アプリケーションでは、データベース ベンダによって提供されるあらゆる高可用性ソリューションまたはフェイルオーバ ソリューションを利用することができます。

  JMS JDBC ストアのコンフィグレーションの詳細については、『管理者ガイド』の「ストアのコンフィグレーション」を参照してください。

移行のフェイルオーバ

WebLogic Server の障害の回復手順については、『管理者ガイド』の「JMS の管理」を参照してください。

WebLogic JMS のチューニング

以下の節では、WebLogic JMS で利用できる管理パフォーマンス チューニング機能を実装することによって、アプリケーションを最大限に活用する方法について説明します。

- JMS ファイル ストアへの同期書き込みポリシー - 通常、同期書き込みを無効にすると、ファイル ストアのパフォーマンスは大幅に向上します。その代わり、オペレーティング システムがクラッシュしたり、ハードウェアの障害が発生したりした場合には、送信したメッセージが失われたり、同じメッセージを重複して受信したりする可能性があります。

  詳細については、『管理者ガイド』の「JMS ファイル ストアの同期書き込みポリシーのコンフィグレーション」を参照してください。
- メッセージ ページング - メッセージの負荷が指定のしきい値に達したときにメッセージをメモリから永続ストレージにスワップすることで、メッセージ負荷のピーク期間中に貴重な仮想メモリを解放できます。パフォーマンスの点から見れば、この機能は、今日のエンタープライズ アプリケーションが必要とする大容量のメッセージ領域を持つ WebLogic Server の実装には大きなメリットがあります。

詳細については、『管理者ガイド』の「メッセージ ページングのコンフィグレーション」を参照してください。

- メッセージ フロー制御の確立 - メッセージ フローのオーバーフローと判断された場合にメッセージ フローを制限する指示をメッセージ プロデューサに与えるように、JMS サーバまたは JMS 送り先 (キューまたはトピック) をコンフィグレーションすることができます。

  詳細については、『管理者ガイド』の「メッセージのフロー制御の確立」を参照してください。

- 分散送り先のチューニング - 次の JMS 接続ファクトリ上の属性をコンフィグレーションして、分散送り先をチューニングできます。
  - ロード バランシング - WebLogic JMS が分散送り先間でメッセージの負荷を分散するか、バランスを取るかが定義されます。
  - サーバ アフィニティ - WebLogic Server が分散送り先セット内の物理的送り先の間でコンシューマまたはプロデューサのロード バランシングを試みる場合、同じ WebLogic Server で実行されている物理的送り先の間でロード バランシングを最初に行うかどうか定義します。

  詳細については、『管理者ガイド』の「分散送り先のチューニング」を参照してください。

WebLogic JMS のモニタ

JMS サーバ、接続、セッション、送り先、恒久サブスクライバ、メッセージ プロデューサ、メッセージ コンシューマ、サーバ セッション プールなどの JMS オブジェクトに関する統計が提供されます。Administration Console を使用して JMS 統計をモニタできます。

サーバの実行中は、JMS 統計は増え続けます。サーバを再起動するときにのみ、統計をリセットできます。

WebLogic JMS のコンフィグレーションとモニタの詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

WebLogic JMS をコンフィグレーションしたら、アプリケーションで JMS API を使用してメッセージの送受信ができるようになります。詳細については、4-1 ページの「WebLogic JMS アプリケーションの開発」を参照してください。

WebLogic Server の障害からの回復

WebLogic Server の障害からの回復手順、手動フェイルオーバの実行手順、およびプログラミングの考慮事項については、『管理者ガイド』の「JMS の管理」を参照してください。

4 WebLogic JMS アプリケーションの開発

以下の節では、WebLogic Server JMS アプリケーションを開発する方法について説明します。

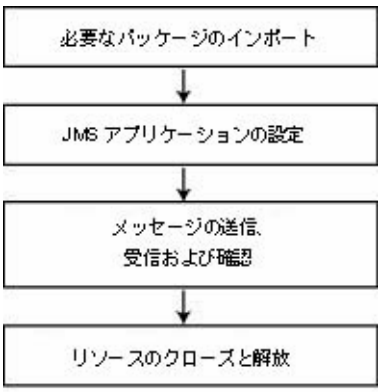
- アプリケーション開発フロー
- 必要なパッケージのインポート
- JMS アプリケーションの設定
- メッセージの送信
- メッセージの受信
- 受信メッセージの確認応答
- オブジェクト リソースの解放
- ロールバック、回復、または期限切れとなったメッセージの管理
- メッセージ配信時間の設定
- 接続の管理
- セッションの管理
- 一時的な送り先の使い方
- 恒久サブスクリプションの設定
- メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの設定と参照
- メッセージのフィルタ処理
- サーバ セッション プールの定義
- マルチキャストの使い方
- 分散送り先の使用

注意： この節で説明する JMS クラスの詳細については、Sun Microsystems の Java Web サイト (http://java.sun.com/products/jms/docs.html) にある JMS Javadoc を参照してください。

アプリケーション開発フロー

WebLogic JMS アプリケーションを開発するには、次の図に示す手順を行う必要があります。

図 4-1 WebLogic JMS アプリケーション開発フロー - 必要な手順



上の図に示したアプリケーション開発手順の他にも、設計開発時に以下の手順を任意に行うことができます。

- 接続およびセッション処理の管理
- 送り先の動的作成
- 恒久サブスクリプションの作成
- メッセージ ヘッダおよびメッセージ プロパティ フィールドの設定と参照、メッセージのフィルタ処理、およびメッセージの並行処理によるメッセージ処理の管理
- マルチキャストの使用

- トランザクション内での JMS の使用 (5-1 ページの「WebLogic JMS によるトランザクションの使い方」を参照)

以降の節では、すべてのアプリケーション開発手順について説明します (最後の手順を除く)。

必要なパッケージのインポート

次の表に、WebLogic JMS アプリケーションで一般に使用されるパッケージを示します。

表 4-1 WebLogic JMS パッケージ

パッケージ名	説明
javax.jms	Sun Microsystems の JMS API。このパッケージは常に WebLogic JMS アプリケーションで使用される。
java.util	日付や時刻機能などのユーティリティ API。
java.io	システム入力および出力 API。
javax.naming weblogic.jndi	サーバおよび送り先ルックアップに必要な JNDI パッケージ。
javax.transaction.UserTransaction	JTA ユーザ トランザクション サポートに必要な JTA API。
weblogic.jms.ServerSessionPoolFactory	JMS 仕様で定義されているオプションのアプリケーション サーバ機能、サーバ セッション プールを使用するための WebLogic JMS パブリック API。
weblogic.jms.extensions	追加のクラスとメソッドを提供する WebLogic 固有の JMS パブリック API (1-7 ページの「WebLogic JMS の拡張機能」を参照)。

プログラムの最初に、以下のパッケージ import 文を挿入します。

```
import javax.jms.*;
import java.util.*;
import java.io.*;
import javax.naming.*;
import javax.transaction.*;
```

サーバ セッション プール アプリケーションを実装する場合は、次のクラスをインポート リストに追加します。

```
import weblogic.jms.ServerSessionPoolFactory;
```

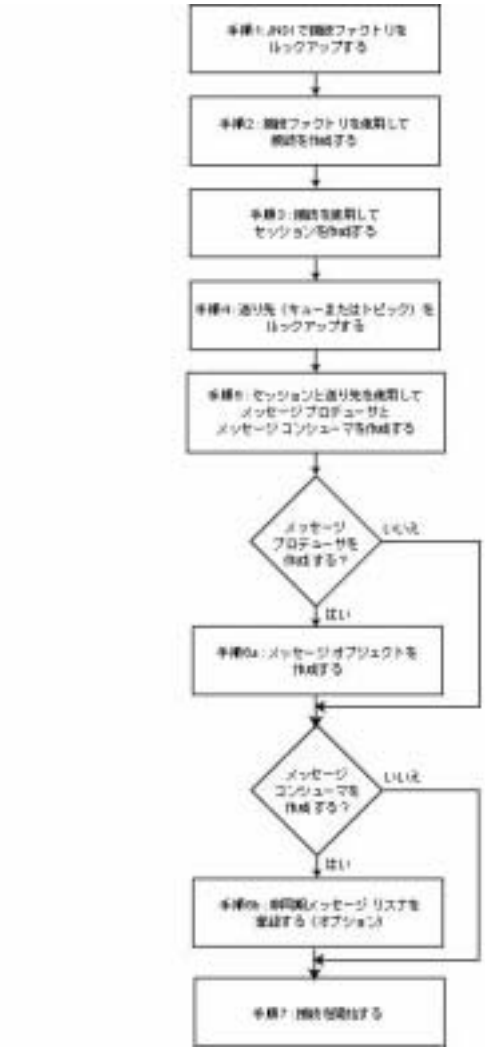
上の表に示した WebLogic JMS 拡張クラスを使用する場合は、次の文もインポート リストに追加します。

```
import weblogic.jms.extensions.*;
```

JMS アプリケーションの設定

メッセージを送受信するには、あらかじめ JMS アプリケーションを設定しておく必要があります。次の図に、JMS アプリケーションの設定に必要な手順を示します。

図 4-2 JMS アプリケーションの設定



以下の節では、この設定手順について説明します。また、ポイント ツー ポイント (PTP) およびパブリッシュ / サブスクライブ (Pub/Sub) アプリケーションの詳しい例も示します。これらの例は、

WL_HOME\samples\server\src\examples\jms ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms パッケージからの抜粋です。

設定手順に進む前に、WebLogic Server のコンフィグレーションを担当するシステム管理者が必要な JMS 機能 (接続ファクトリ、JMS サーバ、送り先など) をコンフィグレーションしたことを確認してください。詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

これらの節で説明する JMS クラスおよびメソッドの詳細については、2-5 ページの「WebLogic JMS のクラス」、または javax.jms、weblogic.jms.ServerSessionPoolFactory、weblogic.jms.extensions の Javadoc を参照してください。

トランザクション アプリケーションと JTA ユーザ トランザクションの設定については、5-1 ページの「WebLogic JMS によるトランザクションの使い方」を参照してください。

手順 1 :JNDI で接続ファクトリをルックアップする

接続ファクトリをルックアップするには、あらかじめ接続ファクトリをコンフィグレーション情報の一部として定義しておく必要があります。WebLogic JMS には、コンフィグレーションの一部として組み込まれているデフォルト接続ファクトリが 1 つ用意されています。WebLogic JMS システム管理者は、コンフィグレーション時に接続ファクトリを追加または更新できます。接続ファクトリのコンフィグレーションと使用可能なデフォルトについては、『管理者ガイド』の「JMS の管理」を参照してください。

接続ファクトリを定義したら、その接続ファクトリをルックアップするために、まず NamingManager.InitialContext() メソッドを使用して JNDI コンテキスト (context) を定義します。サーブレット アプリケーション以外のアプリケーションの場合は、初期コンテキストの作成に使用する環境を渡す必要があります。詳細については、NamingManager.InitialContext() Javadoc を参照してください。

コンテキストを定義したら、JNDI で接続ファクトリをルックアップするために、以下のコマンド (PTP または Pub/Sub メッセージング用) のいずれかを実行します。

```
QueueConnectionFactory queueConnectionFactory =
  (QueueConnectionFactory) context.lookup(CF_name);

TopicConnectionFactory topicConnectionFactory =
  (TopicConnectionFactory) context.lookup(CF_name);
```

CF_name 引数には、コンフィグレーション時に定義した接続ファクトリ名を指定します。

`ConnectionFactory` クラスの詳細については、2-6 ページの「ConnectionFactory」、または `javax.jms.ConnectionFactory` Javadoc を参照してください。

手順 2 : 接続ファクトリを使用して接続を作成する

キューまたはトピックにアクセスするための接続を作成するには、以降の節で説明する `ConnectionFactory` メソッドを使用します。

`Connection` クラスの詳細については、2-8 ページの「Connection」、または `javax.jms.Connection` Javadoc を参照してください。

キュー接続の作成

`QueueConnectionFactory` は、キュー接続を作成するための、以下の 2 つのメソッドを提供します。

```
public QueueConnection createQueueConnection(
) throws JMSException

public QueueConnection createQueueConnection(
  String userName,
  String password
) throws JMSException
```

最初のメソッドは `QueueConnection` を作成し、2 番目のメソッドは指定されたユーザ ID を使用して `QueueConnection` を作成します。どちらのケースでも、接続は停止モードで作成されます。メッセージを受け付けるには、4-17 ページの「手順 7 : 接続を開始する」で説明するとおりに接続を開始しなければなりません。

QueueConnectionFactory クラス メソッドの詳細については、javax.jms.QueueConnectionFactory Javadoc を参照してください。QueueConnection クラスの詳細については、javax.jms.QueueConnection Javadoc を参照してください。

トピック接続の作成

TopicConnectionFactory は、トピック接続を作成するための、以下の 2 つのメソッドを提供します。

```
public TopicConnection createTopicConnection(
) throws JMSException
```

```
public TopicConnection createTopicConnection(
  String userName,
  String password
) throws JMSException
```

最初のメソッドは TopicConnection を作成し、2 番目のメソッドは指定されたユーザ ID を使用して TopicConnection を作成します。どちらのケースでも、接続は停止モードで作成されます。メッセージを受け付けるには、4-17 ページの「手順 7 : 接続を開始する」で説明するとおりに接続を開始しなければなりません。

TopicConnectionFactory クラス メソッドの詳細については、javax.jms.TopicConnectionFactory Javadoc を参照してください。TopicConnection クラスの詳細については、javax.jms.TopicConnection Javadoc を参照してください。

手順 3 : 接続を使用してセッションを作成する

キューまたはトピックにアクセスするために 1 つまたは複数のセッションを作成するには、以降の節で説明する Connection メソッドを使用します。

注意 : セッションおよびそのメッセージのプロデューサとコンシューマには、一度に 1 つのスレッドしかアクセスできません。それらに複数のスレッドが同時にアクセスした場合、それらの動作は定義されません。

Session クラスの詳細については、2-10 ページの「Session」、または javax.jms.Session Javadoc を参照してください。

キュー セッションの作成

QueueConnection クラスは、キュー セッション作成用の次のメソッドを定義します。

```
public QueueSession createQueueSession(
  boolean transacted,
  int acknowledgeMode
) throws JMSException
```

このメソッドでは、セッションをトランザクション処理するか (true)、またはトランザクション処理しないか (false) を示す boolean 引数と、2-11 ページの表 2-5 「非トランザクション セッションで使用する確認応答モード」で説明した非トランザクション セッションの確認応答モードを示す整数値を指定する必要があります。トランザクション セッションの場合、acknowledgeMode 属性は無視されます。この場合、メッセージは commit() メソッドでトランザクションがコミットされたときに確認応答されます。

QueueConnection クラス メソッドの詳細については、javax.jms.QueueConnection Javadoc を参照してください。QueueSession クラスの詳細については、javax.jms.QueueSession Javadoc を参照してください。

トピック セッションの作成

TopicConnection クラスは、トピック セッション作成用の次のメソッドを定義します。

```
public TopicSession createTopicSession(
  boolean transacted,
  int acknowledgeMode
) throws JMSException
```

このメソッドでは、セッションをトランザクション処理するか (true)、またはトランザクション処理しないか (false) を示す boolean 引数と、2-11 ページの「非トランザクション セッションで使用する確認応答モード」で説明した非トランザクション セッションの確認応答モードを示す整数値を指定する必要があります。トランザクション セッションの場合、acknowledgeMode 属性は無視されます。この場合、メッセージは commit() メソッドでトランザクションがコミットされたときに確認応答されます。

TopicConnection クラス メソッドの詳細については、javax.jms.TopicConnection Javadoc を参照してください。TopicSession クラスの詳細については、javax.jms.TopicSession Javadoc を参照してください。

手順 4 : 送り先 (キューまたはトピック) をルックアップする

送り先をルックアップするには、あらかじめ WebLogic JMS システム管理者によって送り先がコンフィグレーションされている必要があります。詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

送り先のコンフィグレーションが済んでいれば、JNDI コンテキスト (context) を定義し (4-6 ページの「手順 1 :JNDI で接続ファクトリをルックアップする」で実行済み)、以下のコマンド (PTP または Pub/Sub メッセージング用) のいずれかを実行することによって、送り先をルックアップできます。

```
Queue queue = (Queue) context.lookup(Dest_name);
```

```
Topic topic = (Topic) context.lookup(Dest_name);
```

Dest_name 引数には、コンフィグレーション時に定義された送り先の JNDI 名を指定します。

JNDI ネームスペースを使用しない場合は、以下の QueueSession または TopicSession メソッドを使用してキューまたはトピックをそれぞれ参照できます。

```
public Queue createQueue(
  String queueName
) throws JMSException
```

```
public Topic createTopic(
  String topicName
) throws JMSException
```

queueName と topicName 文字列の構文は、JMS_Server_Name/Destination_Name (たとえば、*myjmsserver/mydestination*) です。この構文を使用するソース コードを参照するには、4-53 ページの「送り先の動的作成」の findqueue( ) の例を参照してください。

注意： createQueue() メソッドと createTopic() メソッドでは送り先が動的には作成されず、既に存在する送り先への参照が作成されるだけです。送り先の動的作成については、4-53 ページの「送り先の動的作成」を参照してください。

これらのメソッドの詳細については、それぞれ javax.jms.QueueSession Javadoc と javax.jms.TopicSession Javadoc を参照してください。

送り先を定義したら、以下の Queue メソッドまたは Topic メソッドを使用してキューまたはトピックにそれぞれアクセスできます。

```
public String getQueueName(
) throws JMSException
```

```
public String getTopicName(
) throws JMSException
```

キュー名とトピック名が印刷可能なフォーマットで返されるようにするには、toString() メソッドを使用します。

Destination クラスの詳細については、2-13 ページの「Destination」、または javax.jms.Destination Javadoc を参照してください。

送り先ルックアップ時のサーバ アフィニティ

createTopic() および createQueue() メソッドでは、「./Destination_Name」構文を使用して、送り先をルックアップする場合のサーバ アフィニティを示すこともできます。これにより、JMS 接続の接続ファクトリ ホストと同じ JVM に、ローカルにデプロイされた送り先を特定できます。名前がローカル JVM にない場合は、同じ名前が別の JVM にデプロイされていても例外が送出されます。

アプリケーションでこの規約を利用すると、createTopic() メソッドおよび createQueue() メソッドを使用する場合にサーバ名をハードコード化せずに済むので、コードを変更しなくても別の JVM でコードを再利用できます。

手順 5 : セッションと送り先を使用してメッセージ プロデューサとメッセージ コンシューマを作成する

メッセージ プロデューサとメッセージ コンシューマを作成するには、以降の節で説明する Session メソッドに送り先の参照を渡します。

注意 : 各コンシューマはメッセージの独自のローカル コピーを受信します。受信が済んだら、ヘッダ フィールド値を変更することはできますが、メッセージ プロパティとメッセージ本文は読み込み専用です。(この時点でメッセージ プロパティまたは本文を変更しようとすると、MessageNotWriteableException が発生します)。メッセージ本文を変更するには、対応するメッセージ タイプの clearbody() メソッドを実行して、既存の内容を消去し、書き込みパーミッションを有効にします。

MessageProducer クラスと MessageConsumer クラスの詳細については、2-15 ページの「MessageProducer と MessageConsumer」、または javax.jms.MessageProducer Javadoc と ja vax.jms.MessageConsumer Javadoc をそれぞれ参照してください。

QueueSender と QueueReceiver の作成

QueueSession オブジェクトは、キュー センダとキュー レシーバを作成するための、以下のメソッドを定義します。

```
public QueueSender createSender(
  Queue queue
) throws JMSException

public QueueReceiver createReceiver(
  Queue queue
) throws JMSException

public QueueReceiver createReceiver(
  Queue queue,
  String messageSelector
) throws JMSException
```

作成するキュー センダまたはキュー レシーバに関連付けるキュー オブジェクトを指定しなければなりません。また、メッセージをフィルタ処理するためのメッセージ セレクタを指定できます。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

`createSender()` メソッドに null 値を渡すと、匿名プロデューサが作成されます。この場合、4-23 ページの「メッセージの送信」で説明するように、メッセージの送信時にキュー名を指定しなければなりません。

キュー センダまたはキュー レシーバの作成が済んだら、以下の `QueueSender` メソッドまたは `QueueReceiver` メソッドを使用して、そのキュー センダまたはレシーバに関連付けられているキュー名にアクセスできます。

```
public Queue getQueue(
) throws JMSException
```

`QueueSession` クラス メソッドの詳細については、`javax.jms.QueueSession` Javadoc を参照してください。`QueueSender` クラスと `QueueReceiver` クラスの詳細については、`javax.jms.QueueSender` Javadoc および `javax.jms.QueueReceiver` Javadoc をそれぞれ参照してください。

TopicPublisher と TopicSubscriber の作成

`TopicSession` オブジェクトは、トピック パブリッシャとトピック サブスクライバを作成するための、以下のメソッドを定義します。

```
public TopicPublisher createPublisher(
  Topic topic
) throws JMSException
```

```
public TopicSubscriber createSubscriber(
  Topic topic
) throws JMSException
```

```
public TopicSubscriber createSubscriber(
  Topic topic,
  String messageSelector,
  boolean noLocal
) throws JMSException
```

注意： この節で説明するメソッドでは、非恒久サブスクライバが作成されます。非恒久トピック サブスクライバは、アクティブな間だけメッセージを受信します。恒久サブスクリプションを作成して、すべてのメッセージが恒久サブスクライバに届けられるまでメッセージを保持できるようにす

るためのメソッドについては、4-57 ページの「恒久サブスクリプションの設定」を参照してください。この場合、恒久サブスクライバはサブスクライバがサブスクライブした後にパブリッシュされたメッセージのみを受信します。

作成するパブリッシャまたはサブスクライバに関連付けるトピック オブジェクトを指定しなければなりません。また、メッセージをフィルタ処理するためのメッセージ セレクタ、および noLocal フラグ (この節で後述) を指定することもできます。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

createPublisher() メソッドに null 値を渡すと、匿名プロデューサが作成されます。この場合、4-23 ページの「メッセージの送信」で説明するように、メッセージの送信時にトピック名を指定しなければなりません。

アプリケーションは、JMS 接続を使用して同じトピックに対してパブリッシュとサブスクライブの両方を行う場合があります。トピック メッセージはすべてのサブスクライバに届けられるので、アプリケーションは自身がパブリッシュしたことを示すメッセージを受信する可能性があります。この動作を防ぐために、JMS アプリケーションは noLocal フラグを true に設定できます。

トピック パブリッシャまたはトピック サブスクライバの作成が済んだら、以下の TopicPublisher メソッドまたは TopicSubscriber メソッドを使用して、そのトピック パブリッシャまたはサブスクライバに関連付けられているトピック名にアクセスできます。

```
Topic getTopic(
) throws JMSException
```

また、次の TopicSubscriber メソッドを使用すると、トピック サブスクライバに関連付けられる noLocal 変数の設定値にアクセスできます。

```
boolean getNoLocal(
) throws JMSException
```

TopicSession クラス メソッドの詳細については、javax.jms.TopicSession Javadoc を参照してください。TopicPublisher クラスと TopicSubscriber クラスの詳細については、javax.jms.TopicPublisher Javadoc および javax.jms.TopicSubscriber Javadoc をそれぞれ参照してください。

手順 6a : メッセージ オブジェクトを作成する (メッセージ プロデューサ)

注意 : この手順は、メッセージ プロデューサだけに適用されます。

メッセージ オブジェクトを作成するには、以下の Session クラス メソッドまたは WLSession クラス メソッドのいずれかを使用します。

- Session メソッド

  注意 : これらのメソッドは、QueueSession サブクラスと TopicSession サブクラスの両方によって継承されます。

```
public BytesMessage createBytesMessage(
) throws JMSException

public MapMessage createMapMessage(
) throws JMSException

public Message createMessage(
) throws JMSException

public ObjectMessage createObjectMessage(
) throws JMSException

public ObjectMessage createObjectMessage(
  Serializable object
) throws JMSException

public StreamMessage createStreamMessage(
) throws JMSException

public TextMessage createTextMessage(
) throws JMSException

public TextMessage createTextMessage(
  String text
) throws JMSException
```

- WLSession メソッド

```
public XMLMessage createXMLMessage(
  String text
) throws JMSException
```

Session クラスと WLSession クラス メソッドの詳細については、javax.jms.Session Javadoc と weblogic.jms.extensions.WLSession Javadoc をそれぞれ参照してください。Message クラスとそのメソッドの詳細については、2-17 ページの「Message」、または javax.jms.Message Javadoc を参照してください。

手順 6b : 非同期メッセージ リスナを登録する (オプション) (メッセージ コンシューマ)

注意 : この手順は、メッセージ コンシューマだけに適用されます。

メッセージを非同期的に受信するには、次の手順で非同期メッセージ リスナを登録する必要があります。

1. onMessage() メソッドが組み込まれた javax.jms.MessageListener インタフェースを実装します。

   注意 : onMessage() メソッドのインタフェースの例については、4-17 ページの「例 :PTP アプリケーションの設定」を参照してください。

   onMessage() メソッド呼び出し内で close() メソッドを発行する場合、システム管理者は接続ファクトリをコンフィグレーションするときに [メッセージの短縮を許可] チェックボックスをチェックしなければなりません。JMS のコンフィグレーションの詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

2. 次の MessageConsumer メソッドを使用してメッセージ リスナを設定し、リスナ情報を引数として渡します。

   ```
   public void setMessageListener(
     MessageListener listener
   ) throws JMSException
   ```

3. オプションで、4-50 ページの「セッション例外リスナの定義」で説明するように、例外を取得するためのセッションの例外リスナを実装します。

メッセージ リスナの設定を解除するには、null 値を指定して MessageListener() メソッドを呼び出します。

メッセージ リスナを定義したら、次の MessageConsumer メソッドを呼び出してそのリスナにアクセスできます。

```
public MessageListener getMessageListener(
) throws JMSException
```

注意： WebLogic JMS は、同じセッションの複数の onMessage() 呼び出しが同時に実行されないことを保証します。

メッセージ コンシューマが管理者またはサーバのダウンによってクローズされた場合、ConsumerClosedException がセッション例外リスナに送信されます (定義されている場合)。このように、必要な場合は新しいメッセージ コンシューマを作成できます。セッション例外リスナの定義については、4-50 ページの「セッション例外リスナの定義」を参照してください。

MessageConsumer クラスのメソッドは、QueueReceiver クラスおよび TopicSubscriber クラスによって継承されます。MessageConsumer クラス メソッドの詳細については、2-15 ページの「MessageProducer と MessageConsumer」、または javax.jms.MessageConsumer Javadoc を参照してください。

手順 7 : 接続を開始する

接続を開始するには、Connection クラスの start() メソッドを使用します。

接続の開始、停止、およびクローズの詳細については、4-49 ページの「接続の開始、停止、クローズ」、または javax.jms.Connection Javadoc を参照してください。

例 :PTP アプリケーションの設定

次の例は、*WL_HOME*\samples\server\src\examples\jms\queue ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.queue.QueueSend の例からの抜粋です。init() メソッドは、JMS アプリケーションの QueueSession をどのように設定して開始するかを示すものです。次に、その init() メソッドを示し、併せて各設定手順も述べます。

必要な変数 (JNDI コンテキストなど)、JMS 接続ファクトリ、およびキュー静的変数を定義します。

```
public final static String JNDI_FACTORY=
       "weblogic.jndi.WLInitialContextFactory";
public final static String JMS_FACTORY=
       "weblogic.examples.jms.QueueConnectionFactory";
public final static String
       QUEUE="weblogic.examples.jms.exampleQueue";

private QueueConnectionFactory qconFactory;
private QueueConnection qcon;
private QueueSession qsession;
private QueueSender qsender;
private Queue queue;
private TextMessage msg;
```

JNDI 初期コンテキストを次のとおり設定します。

```
InitialContext ic = getInitialContext(args[0]);
       .
       .
       .
private static InitialContext getInitialContext(
  String url
) throws NamingException
{
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
    env.put(Context.PROVIDER_URL, url);
    return new InitialContext(env);
}
```

注意 : EJB またはサーブレットの JNDI 初期コンテキストを設定する場合は、以下のメソッドを使用します。

```
Context ctx = new InitialContext();
```

JMS キューにメッセージを送信するのに必要なすべてのオブジェクトを作成します。`ctx` オブジェクトは、`main()` メソッドによって渡された JNDI 初期コンテキストです。

```
public void init(
  Context ctx,
  String queueName
) throws NamingException, JMSException
{
```

手順 1　JNDI で接続ファクトリをルックアップします。

```
  qconFactory = (QueueConnectionFactory) ctx.lookup(JMS_FACTORY);
```

手順 2　接続ファクトリを使用して接続を作成します。

```
qcon = qconFactory.createQueueConnection();
```

手順 3　接続を使用してセッションを作成します。次のコードでは、セッションが非トランザクションとして定義され、メッセージに対する確認応答が自動的に行われるものと指定されます。トランザクション セッションと確認応答モードの詳細については、2-10 ページの「Session」を参照してください。

```
qsession = qcon.createQueueSession(false,
  Session.AUTO_ACKNOWLEDGE);
```

手順 4　JNDI で送り先 (キュー) をルックアップします。

```
queue = (Queue) ctx.lookup(queueName);
```

手順 5　セッションと送り先 (キュー) を使用してメッセージ プロデューサ (キュー センダ) への参照を作成します。

```
qsender = qsession.createSender(queue);
```

手順 6　メッセージ オブジェクトを作成します。

```
msg = qsession.createTextMessage();
```

手順 7　接続を開始します。

```
qcon.start();
}
```

examples.jms.queue.QueueReceive の例の init() メソッドは、前記の QueueSend init() メソッドとほぼ同じですが、例外が 1 つあります。手順 5 と手順 6 は、それぞれ以下のコードに置き換えられます。

```
qreceiver = qsession.createReceiver(queue);
qreceiver.setMessageListener(this);
```

最初の行では、createSender() メソッドを呼び出してキュー センダへの参照を作成する代わりに、アプリケーションは createReceiver() メソッドを呼び出してキュー レシーバを作成します。

2 番目の行では、メッセージ コンシューマは非同期メッセージ リスナを登録します。

メッセージがキュー セッションに届くと、そのメッセージは examples.jms.QueueReceive.onMessage() メソッドに渡されます。次の QueueReceive の例からの引用コードは、onMessage() インタフェースを示したものです。

```
public void onMessage(Message msg)
{
    try {
       String msgText;
       if (msg instanceof TextMessage) {
          msgText = ((TextMessage)msg).getText();
       } else { // TextMessage ではない場合 ...
         msgText = msg.toString();
       }

       System.out.println("Message Received:"+ msgText );

       if (msgText.equalsIgnoreCase("quit")) {
          synchronized(this) {

              quit = true;
              this.notifyAll(); // メイン スレッドに終了するよう通知する
          }
       }
   } catch (JMSException jmse) {
     jmse.printStackTrace();
   }
}
```

onMessage() メソッドは、キュー レシーバを通して受信したメッセージを処理します。このメソッドは、メッセージが TextMessage であるかどうかを検証し、そうである場合は、そのメッセージのテキストを出力します。onMessage() が別のタイプのメッセージを受信した場合、そのメッセージの toString() メソッドを使用してメッセージの内容を表示します。

注意： 受信したメッセージのタイプが、メッセージ ハンドラ メソッドが予期したタイプであるかどうかを検証するようにしてください。

この例で使用した JMS クラスの詳細については、2-5 ページの「WebLogic JMS のクラス」、または javax.jms Javadoc を参照してください。

例 :Pub/Sub アプリケーションの設定

次の例は、*WL_HOME*\samples\server\src\examples\jms\topic ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.topic.TopicSend の例からの抜粋です。init() メソッドは、JMS アプリケーションのトピック セッションをどのように設定して開始するかを示すものです。次に、その init() メソッドを示し、併せて各設定手順も述べます。

必要な変数 (JNDI コンテキストなど)、JMS 接続ファクトリ、およびトピック静的変数を定義します。

```
public final static String JNDI_FACTORY=
       "weblogic.jndi.WLInitialContextFactory";
public final static String JMS_FACTORY=
       "weblogic.examples.jms.TopicConnectionFactory";
public final static String
       TOPIC="weblogic.examples.jms.exampleTopic";

protected TopicConnectionFactory tconFactory;
protected TopicConnection tcon;
protected TopicSession tsession;
protected TopicPublisher tpublisher;
protected Topic topic;
protected TextMessage msg;
```

JNDI 初期コンテキストを次のとおり設定します。

```
InitialContext ic = getInitialContext(args[0]);
       .
       .
       .
private static InitialContext getInitialContext(
  String url
) throws NamingException
{
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
    env.put(Context.PROVIDER_URL, url);
    return new InitialContext(env);
}
```

注意 : サーブレットの JNDI 初期コンテキストを設定する場合は、以下のメソッドを使用します。

```
Context ctx = new InitialContext();
```

JMS キューにメッセージを送信するのに必要なすべてのオブジェクトを作成します。ctx オブジェクトは、main() メソッドによって渡された JNDI 初期コンテキストです。

```
public void init(
  Context ctx,
  String topicName
) throws NamingException, JMSException
{
```

手順 1　JNDI を使用して接続ファクトリをルックアップします。

```
    tconFactory =
      (TopicConnectionFactory) ctx.lookup(JMS_FACTORY);
```

手順 2　接続ファクトリを使用して接続を作成します。

```
    tcon = tconFactory.createTopicConnection();
```

手順 3　接続を使用してセッションを作成します。次のコードでは、セッションが非トランザクションとして定義され、メッセージに対する確認応答が自動的に行われるものと指定されます。セッション トランザクションと確認応答モードの設定については、2-10 ページの「Session」を参照してください。

```
    tsession = tcon.createTopicSession(false,
      Session.AUTO_ACKNOWLEDGE);
```

手順 4　JNDI を使用して送り先 (トピック) をルックアップします。

```
    topic = (Topic) ctx.lookup(topicName);
```

手順 5　セッションと送り先 (トピック) を使用してメッセージ プロデューサ (トピック パブリッシャ) への参照を作成します。

```
    tpublisher = tsession.createPublisher(topic);
```

手順 6　メッセージ オブジェクトを作成します。

```
    msg = tsession.createTextMessage();
```

手順 7　接続を開始します。

```
    tcon.start();
  }
```

examples.jms.topic.TopicReceive の例の init() メソッドは、前記の TopicSend init() メソッドとほぼ同じですが、例外が 1 つあります。手順 5 と手順 6 は、それぞれ以下のコードに置き換えられます。

```
tsubscriber = tsession.createSubscriber(topic);
tsubscriber.setMessageListener(this);
```

最初の行では、createPublisher() メソッドを呼び出してトピック パブリッシャへの参照を作成する代わりに、アプリケーションは createSubscriber() メソッドを呼び出してトピック サブスクライバを作成します。

2 番目の行では、メッセージ コンシューマは非同期メッセージ リスナを登録します。

メッセージがトピック セッションに届くと、そのメッセージは examples.jms.TopicSubscribe.onMessage() メソッドに渡されます。TopicReceive の onMessage() インタフェースは、QueueReceive onMessage() インタフェース (4-17 ページの「例 :PTP アプリケーションの設定」を参照) と同じです。

この例で使用した JMS クラスの詳細については、2-5 ページの「WebLogic JMS のクラス」、または javax.jms Javadoc を参照してください。

メッセージの送信

4-4 ページの「JMS アプリケーションの設定」で説明したとおり JMS アプリケーションを設定したら、メッセージを送信することができます。メッセージを送信するには、次の手順を実行する必要があります。

1. メッセージ オブジェクトを作成する。
2. メッセージを定義する。
3. メッセージを送り先に送信する。

メッセージを送信するための JMS クラスとメッセージ タイプの詳細については、javax.jms.Message Javadoc を参照してください。メッセージの受信については、4-31 ページの「メッセージの受信」を参照してください。

手順 1 : メッセージ オブジェクトを作成する

この手順は、4-15 ページの「手順 6a : メッセージ オブジェクトを作成する (メッセージ プロデューサ)」で説明したように、クライアント設定手順の一部として既に実行されています。

手順 2 : メッセージを定義する

この手順は、4-15 ページの「手順 6a : メッセージ オブジェクトを作成する (メッセージ プロデューサ)」で説明したように、アプリケーションの設定時に実行されている場合もあります。この手順が実行済みであるかどうかは、メッセージ オブジェクトを作成するために呼び出されたメソッドによって決まります。たとえば、TextMessage タイプと ObjectMessage タイプの場合は、メッセージ オブジェクトを作成するときにオプションでメッセージを定義することができます。

既に値が指定されており、それを変更したくない場合は、そのまま手順 3 に進みます。

値が指定されていないか、または既存の値を変更する場合は、適切な `set` メソッドを使用して値を定義できます。たとえば、`TextMessage` のテキストを定義するメソッドは次のとおりです。

```
public void setText(
  String string
) throws JMSException
```

注意 : メッセージは null として定義することができます。

それ以後は、次のメソッドを使用してメッセージを消去できます。

```
public void clearBody(
) throws JMSException
```

メッセージを定義するためのメソッドの詳細については、`javax.jms.Session` Javadoc を参照してください。

手順 3 : メッセージを送り先に送信する

メッセージを送り先に送信するには、メッセージ プロデューサ — キュー センダ (PTP) またはトピック パブリッシャ (Pub/Sub) — と以下の節で説明するメソッドを使用します。`Destination` オブジェクトと `MessageProducer` オブジェクトは、4-4 ページの「JMS アプリケーションの設定」で説明したとおり、アプリケーションの設定時に作成されています。

注意 : 同じトピックに対して複数のトピック サブスクライバが定義されている場合、各サブスクライバはメッセージの独自のローカル コピーを受信します。受信が済んだら、ヘッダ フィールド値を変更することはできますが、メッセージ プロパティとメッセージ本文は読み込み専用です。メッセージ本文を変更するには、対応するメッセージ タイプの `clearbody()` メソッドを実行して、既存の内容を消去し、書き込みパーミッションを有効にします。

`MessageProducer` クラスの詳細については、2-15 ページの「MessageProducer と MessageConsumer」、または `javax.jms.MessageProducer` Javadoc を参照してください。

キュー センダを使用してメッセージを送信する

メッセージを送信するには、以下の `QueueSender` メソッドを使用します。

```
public void send(
  Message message
) throws JMSException

public void send(
  Message message,
  int deliveryMode,
  int priority,
  long timeToLive
) throws JMSException

public void send(
  Queue queue,
  Message message
) throws JMSException

public void send(
  Queue queue,
  Message message,
  int deliveryMode,
```

```
  int priority,
  long timeToLive
) throws JMSException
```

まず、メッセージを定義する必要があります。また、キュー名 (匿名メッセージ プロデューサ用)、配信モード (DeliveryMode.PERSISTENT または DeliveryMode.NON_PERSISTENT)、優先度 (0-9)、および存続時間 (ミリ秒単位) も指定する必要があります。指定しない場合、配信モード、優先度、および存続時間の各属性は以下のいずれかに設定されます。

- プロデューサに対して定義された接続ファクトリまたは送り先オーバーライド コンフィグレーション属性 (『管理者ガイド』の「JMS の管理」を参照)
- メッセージ プロデューサの set メソッドによって指定された値 (4-28 ページの「メッセージ プロデューサ属性の設定」を参照)

注意 : WebLogic JMS には、以下のような独自の属性も用意されています。詳細については、4-28 ページの「メッセージ プロデューサ属性の設定」を参照してください。

- TimeToDeliver (生成時間)。送信されたメッセージが、対象送り先で表示可能になるまでの遅延を表します。
- RedeliveryLimit。回復またはロールバックの後に、メッセージが再配信される回数を示します。

配信モードを PERSISTENT として定義した場合、『管理者ガイド』の「JMS の管理」で説明してあるように、送り先のバッキング ストアをコンフィグレーションする必要があります。

注意 : バッキング ストアがコンフィグレーションされていない場合、配信モードは NON_PERSISTENT に変更され、メッセージは永続ストレージに書き込まれません。

キュー センダが匿名プロデューサである場合 (つまり、キューが作成されたときにその名前が null に設定された場合)、キュー名を指定して (最後の 2 つのメソッドのいずれかを使用する) メッセージの配信先を指示する必要があります。匿名プロデューサの定義の詳細については、4-12 ページの「QueueSender と QueueReceiver の作成」を参照してください。

たとえば、次のコードは、永続的メッセージを優先度 4、存続時間 1 時間で送信します。

```
QueueSender.send(message, DeliveryMode.PERSISTENT, 4, 3600000);
```

QueueSender クラス メソッドの詳細については、javax.jms.QueueSender Javadoc を参照してください。

TopicPublisher を使用してメッセージを送信する

メッセージを送信するには、以下の TopicPublisher メソッドを使用します。

```
public void publish(
  Message message
) throws JMSException

public void publish(
  Message message,
  int deliveryMode,
  int priority,
  long timeToLive
) throws JMSException

public void publish(
  Topic topic,
  Message message
) throws JMSException

public void publish(
  Topic topic,
  Message message,
  int deliveryMode,
  int priority,
  long timeToLive
) throws JMSException
```

メッセージを指定する必要があります。また、トピック名、配信モード (DeliveryMode.PERSISTENT または DeliveryMode.NON_PERSISTENT)、優先度 (0-9)、および存続時間 (ミリ秒単位) も指定する必要があります。指定しない場合、配信モード、優先度、および存続時間の各属性は以下のいずれかに設定されます。

- プロデューサに対して定義された接続ファクトリまたは送り先オーバーライド コンフィグレーション属性 (『管理者ガイド』の「JMS の管理」を参照)
- メッセージ プロデューサの set メソッドによって指定された値 (4-28 ページの「メッセージ プロデューサ属性の設定」を参照)

注意： WebLogic JMS には、以下のような独自の属性も用意されています。詳細については、4-28 ページの「メッセージ プロデューサ属性の設定」を参照してください。

- TimeToDeliver (生成時間)。送信されたメッセージが、対象送り先で表示可能になるまでの遅延を表します。
- RedeliveryLimit。回復またはロールバックの後に、メッセージが再配信される回数を示します。

配信モードを PERSISTENT として定義した場合、『管理者ガイド』の「JMS の管理」で説明してあるようにバッキング ストアをコンフィグレーションする必要があります。

注意 : バッキング ストアがコンフィグレーションされていない場合、配信モードは NON_PERSISTENT に変更され、メッセージは保存されません。

トピック パブリッシャが匿名プロデューサである場合 (つまり、トピックが作成されたときにその名前が null に設定された場合)、トピック名を指定して (最後の 2 つのメソッドのいずれかを使用する) メッセージの配信先を指示する必要があります。匿名プロデューサの定義の詳細については、4-13 ページの「TopicPublisher と TopicSubscriber の作成」を参照してください。

たとえば、次のコードは、永続的メッセージを優先度 4、存続時間 1 時間で送信します。

```
TopicPublisher.publish(message, DeliveryMode.PERSISTENT,
  4,3600000);
```

TopicPublisher クラス メソッドの詳細については、javax.jms.TopicPublisher Javadoc を参照してください。

メッセージ プロデューサ属性の設定

前節で説明したように、メッセージを送信するときには、配信モード、優先度、および存続時間をオプションで指定できます。指定しない場合、これらの属性は接続ファクトリのコンフィグレーション属性に設定されます。詳細については、『管理者ガイド』の「JMS の管理」を参照してください

また、メッセージ プロデューサの set メソッドを使用して、配信モード、優先度、存続時間、配信時間、再配信遅延 (タイムアウト)、再配信制限の値を動的に設定することも可能です。次の表に、メッセージ プロデューサの set メソッドと get メソッドを、動的コンフィグレーション可能な属性ごとに示します。

注意： 配信モード、優先度、存続時間、配信時間、再配信遅延 (タイムアウト)、再配信制限の各属性値は、[配信モードのオーバライド]、[優先順位オーバライド]、[生存時間のオーバライド]、[配信時間のオーバライド]、[再配信遅延のオーバライド]、および [再配信の制限] の各コンフィグレーション属性を使用して、送り先によってオーバーライドできます。詳細については、Administration Console オンライン ヘルプを参照してください。

表 4-2 メッセージ プロデューサの set メソッドおよび get メソッド

属性	set メソッド	get メソッド
配信モード	`public void setDeliveryMode( int deliveryMode ) throws JMSException`	`public int getDeliveryMode( ) throws JMSException`
優先度	`public void setPriority( int defaultPriority ) throws JMSException`	`public int getPriority( ) throws JMSException`
存続時間	`public void setTimeToLive( long timeToLive ) throws JMSException`	`public long getTimeToLive( ) throws JMSException`
配信時間	`public void setTimeToDeliver( long timeToDeliver ) throws JMSException`	`public long getTimeToDeliver( ) throws JMSException`
再配信の制限	`public void setRedeliveryLimit( int redeliveryLimit ) throws JMSException`	`public int getredeliveryLimit( ) throws JMSException`

注意： JMS では、メッセージ ID とタイムスタンプ情報を無効にするための `MessageProducer` メソッドを定義することもできます。ただし、これらのメソッドは WebLogic JMS では無視されます。

`MessageProducer` クラス メソッドの詳細については、Sun の `javax.jms.MessageProducer` Javadoc または `weblogic.jms.extensions.WLMessageProducer` Javadoc を参照してください。

例 : PTP アプリケーション内でのメッセージの送信

次の例は、*WL_HOME*\samples\server\src\examples\jms\queue ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.queue.QueueSend の例からの抜粋です。この例では、TextMessage を作成し、メッセージのテキストを設定してキューに送信するために必要なコードを示してあります。

```
msg = qsession.createTextMessage();
       .
       .
       .
public void send(
  String message
) throws JMSException
{
  msg.setText(message);
  qsender.send(msg);
}
```

QueueSender クラスとメソッドの詳細については、javax.jms.QueueSender Javadoc を参照してください。

例 : Pub/sub アプリケーション内でのメッセージの送信

次の例は、*WL_HOME*\samples\server\src\examples\jms\topic ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.topic.TopicSend の例からの抜粋です。この例では、TextMessage を作成し、メッセージのテキストを設定してトピックに送信するために必要なコードを示してあります。

```
msg = tsession.createTextMessage();
       .
       .
       .
public void send(
  String message
) throws JMSException
{
  msg.setText(message);
  tpublisher.publish(msg);
}
```

TopicPublisher クラスとメソッドの詳細については、javax.jms.TopicPublisher Javadoc を参照してください。

メッセージの受信

4-4 ページの「JMS アプリケーションの設定」で説明したとおり JMS アプリケーションを設定したら、メッセージを受信することができます。

メッセージを受信するには、以下の節で説明するとおりレシーバ オブジェクトを作成し、メッセージを同期受信するか非同期受信するかを指定する必要があります。

メッセージを受信する順序は、以下の要素によって設定できます。

- コンフィグレーション時に定義されるメッセージ配信属性 (配信モードとソート条件) (『管理者ガイド』の「JMS の管理」を参照)、または send() メソッドの一部として定義されるメッセージ配信属性 (4-23 ページの「メッセージの送信」を参照)
- 送り先キーを使用して設定される送り先ソート順序 (『管理者ガイド』の「JMS の管理」を参照)

受信が済んだら、ヘッダ フィールド値を変更することはできますが、メッセージ プロパティとメッセージ本文は読み込み専用です。メッセージ本文を変更するには、対応するメッセージ タイプの clearbody() メソッドを実行して、既存の内容を消去し、書き込みパーミッションを有効にします。

メッセージを受信するための JMS クラスとメッセージ タイプの詳細については、javax.jms.Message Javadoc を参照してください。メッセージの送信については、4-23 ページの「メッセージの送信」を参照してください。

メッセージの非同期受信

この手順については、アプリケーションの設定手順の中で説明されています。詳細については、4-16 ページの「手順 6b : 非同期メッセージ リスナを登録する (オプション) (メッセージ コンシューマ)」を参照してください。

JMS 接続ファクトリのコンフィグレーション時に [最大メッセージ数] 属性を設定すると、非同期セッションの間に存在し、メッセージ リスナにまだ渡されていないメッセージの最大数を指定できます。

非同期メッセージ パイプライン

メッセージの生成が、それらを消費できる非同期メッセージ リスナ (コンシューマ) よりも速い場合、バッチ内でまだ消費されていない複数のメッセージが、使用可能な非同期メッセージ リスナが存在する別のセッションに送信されます。このようにして別のセッションに送信されるメッセージのことを、メッセージ パイプラインと呼んだり、JMS ベンダによってはメッセージ バックログと呼んだりします。パイプライン (またはバックログ) のサイズは、非同期コンシューマに蓄積されながら、まだメッセージ リスナに渡されていないメッセージの数で決まります。

メッセージ パイプラインのコンフィグレーション

クライアントのパイプラインの最大サイズを指定するには、クライアントの接続ファクトリの [最大メッセージ数] 属性をコンフィグレーションします。パイプラインの最大サイズは、「メッセージ リスナに渡されていない状態で、1 つの非同期コンシューマに蓄積できるメッセージの最大数」と定義されます。デフォルト設定は 10 です。

JMS 接続ファクトリのコンフィグレーションの詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

パイプライン化されたメッセージの動作

コンフィグレーションしたメッセージ パイプラインは以下のように動作します。

- 統計 — JMS モニタ統計により、メッセージ パイプライン内のバックログ メッセージが、コミットまたは確認応答されるまでは (キューおよび恒久サブスクライバに対して) 保留中と報告される。
- パフォーマンス — [最大メッセージ数] パイプライン サイズを増やすと、スループットの高いアプリケーションのパフォーマンスが向上する可能性がある。ただし、パイプラインを大きくすると、非同期コンシューマのリスナが呼び出されるまでにクライアント JVM に蓄積される保留中のパイプライン メッセージが増えるため、クライアントのメモリ使用量が増加します。

- ソート — 非同期コンシューマのパイプライン内のメッセージは、コンシューマ送り先にコンフィグレーションされたソート順ではソートされず、JMS サーバから送信された順序のまま蓄積される。たとえば、送り先で優先度によるソートがコンフィグレーションされていても、優先度の高いメッセージが、すでに非同期コンシューマのパイプラインに送信されている優先度の低いメッセージより優先されることはありません。

注意： 接続ファクトリで設定する [最大メッセージ数] パイプライン サイズは、JMS サーバおよび送り先で設定する [最大メッセージ数] 割り当て設定とは無関係です。

パイプライン化されたメッセージが、ネットワーク転送上の 1 つのメッセージに集約されることがあります。メッセージが大きすぎると、集約後に書き込まれるデータのサイズが転送の最大値を超え、予期しない動作が生じるおそれがあります。たとえば、`t3` プロトコルにおける最大メッセージ サイズは、デフォルトでは 10,000,000 バイトに設定され、サーバ上の `MaxT3MessageSize` 属性でコンフィグレーション可能です。この場合、2MB のメッセージが 10 個パイプライン化されると `t3` での最大値を超えてしまいます。

メッセージの同期受信

メッセージを同期的に受信するには、以下の `MessageConsumer` メソッドを使用します。

```
public Message receive(
) throws JMSException

public Message receive(
   long timeout
) throws JMSException

public Message receiveNoWait(
) throws JMSException
```

どのケースでも、アプリケーションは次に生成されるメッセージを受信します。`receive()` メソッドを引数なしで呼び出した場合、その呼び出しはメッセージが生成されるか、またはアプリケーションが閉じられるまで無期限にブロックされます。代わりに、タイムアウト値を渡してメッセージの待ち時間を指定することもできます。値 0 を指定して `receive()` メソッドを呼び出した場合、その呼

び出しは無期限にブロックされます。receiveNoWait() メソッドは、次のメッセージが存在する場合はそれを受信し、それ以外の場合は null を返します。この場合、呼び出しはブロックされません。

MessageConsumer クラスのメソッドは、QueueReceiver クラスおよび TopicSubscriber クラスによって継承されます。MessageConsumer クラス メソッドの詳細については、javax.jms.MessageConsumer Javadoc を参照してください。

例：PTP アプリケーション内でのメッセージの同期受信

次の例は、*WL_HOME*\samples\server\src\examples\jms\queue ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.queue.QueueReceive の例からの抜粋です。メッセージ リスナを設定するのではなく、各メッセージに対して qreceiver.receive() を呼び出します。次に例を示します。

```
qreceiver = qsession.createReceiver(queue);
qreceiver.receive();
```

最初の行では、キューに対するキュー レシーバが作成されます。2 番目の行では、receive() メソッドが実行されます。receive() メソッドは、ブロックしてメッセージを待ちます。

例：Pub/sub アプリケーション内でのメッセージの同期受信

次の例は、*WL_HOME*\samples\server\src\examples\jms\topic ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.topic.TopicReceive の例からの抜粋です。メッセージ リスナを設定するのではなく、各メッセージに対して tsubscriber.receive() を呼び出します。

次に例を示します。

```
tsubscriber = tsession.createSubscriber(topic);
Message msg = tsubscriber.receive();
msg.acknowledge();
```

最初の行では、トピックに対するトピック サブスクライバが作成されます。2 番目の行では、receive() メソッドが実行されます。receive() メソッドは、ブロックしてメッセージを待ちます。

受信メッセージの回復

注意： この節は、2-11 ページの表 2-5 「非トランザクション セッションで使用する確認応答モード」で説明したように、確認応答モードが CLIENT_ACKNOWLEDGE に設定されている非トランザクション セッションだけに適用されます。同期受信される AUTO_ACKNOWLEDGE メッセージは確認応答済みのため回復しないことがあります。

アプリケーションは、次のメソッドを使用して、JMS にメッセージの再配信 (未確認) を要求できます。

```
public void recover(
) throws JMSException
```

recover() メソッドは、次の手順を実行します。

- セッションのメッセージ配信を停止する。
- 確認応答されていない (ただし配信されている可能性のある) すべてのメッセージに再配信のタグを付ける。
- そのセッションの確認応答されていない最初のメッセージからメッセージの送信を再開する。

キュー内のメッセージは、必ずしも元の配信順序と同じ順序で、または同じキュー コンシューマに再配信されるとは限りません。

受信メッセージの確認応答

注意： この節は、2-11 ページの表 2-5 「非トランザクション セッションで使用する確認応答モード」で説明したように、確認応答モードが CLIENT_ACKNOWLEDGE に設定されている非トランザクション セッションだけに適用されます。

受信したメッセージの確認応答を行うには、次の Message メソッドを使用します。

```
public void acknowledge(
) throws JMSException
```

acknowledge() メソッドは、接続ファクトリの [確認応答ポリシー] 属性のコンフィグレーションによって以下のように動作が異なります。

- デフォルト ポリシーの [All] が指定されている場合、メッセージの確認応答を呼び出すと、セッションで受信してまだ確認応答されていないすべてのメッセージが確認応答される。
- [Previous] を指定した場合、メッセージの確認応答を呼び出すと、まだ確認応答されていないメッセージのうち、指定したメッセージ以前のメッセージのみが確認応答される。確認応答が行われないメッセージは、クライアントに再配信できます。

このメソッドは、確認応答モードが CLIENT_ACKNOWLEDGE に設定されている非トランザクション セッションに対してだけ有効です。それ以外の場合、このメソッドは無視されます。

オブジェクト リソースの解放

JMS アプリケーションに代わって作成した接続、セッション、メッセージ プロデューサ / コンシューマ、接続コンシューマ、またはキュー ブラウザを使い終えたら、それらを明示的にクローズしてリソースを解放する必要があります。

JMS オブジェクトをクローズするには、close() メソッドを次のとおり入力します。

```
public void close(
) throws JMSException
```

オブジェクトをクローズするときには、以下の処理が行われます。

- メソッド呼び出しが完了するか、未処理の同期レシーバの onMessage() 呼び出しが完了するまで、その呼び出しがブロックされる。
- 関連付けられているすべてのサブオブジェクトもクローズされます。たとえば、セッションをクローズすると、関連付けられているすべてのメッセージ

プロデューサおよびコンシューマもクローズされます。接続をクローズすると、関連付けられているすべてのセッションもクローズされます。

各オブジェクトの `close()` メソッドの影響については、適切な `javax.jms` Javadoc を参照してください。また、接続またはセッションの `close()` メソッドの詳細については、4-49 ページの「接続の開始、停止、クローズ」、または 4-52 ページの「セッションのクローズ」をそれぞれ参照してください。

次の例は、*`WL_HOME`*`\samples\server\src\examples\jms\queue` ディレクトリ (*`WL_HOME`* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の `examples.jms.queue.QueueSend` の例からの抜粋です。この例を見れば、メッセージ コンシューマ、セッション、および接続オブジェクトをどのようにクローズするかが分かります。

```
public void close(
) throws JMSException
{
  qreceiver.close();
  qsession.close();
  qcon.close();
}
```

この `QueueSend` の例では、`main()` の最後に `close()` メソッドが呼び出され、オブジェクトのクローズとリソースの解放が行われます。

ロールバック、回復、または期限切れとなったメッセージの管理

以下の節では、ロールバックまたは回復したメッセージを管理する以下の方法について説明します。

- メッセージの再配信遅延の設定
- メッセージの再配信制限の設定
- パッシブなメッセージの有効期限ポリシー

メッセージの再配信遅延の設定

一時的または外部的な要因でアプリケーションがメッセージを正しく処理できない場合に、メッセージの再配信を遅延させることができます。これによって、アプリケーションは、現時点では処理できない「有害な」メッセージを一時的に受信できないようにすることができます。メッセージがロールバックまたは回復される場合、再配信遅延は、メッセージが止められてから再配信が試行されるまでの間隔です。

JMS でメッセージをすぐに再配信すると、エラーの原因が解決されず、アプリケーションはメッセージを処理できないままの場合があります。ただし、アプリケーションが再配信遅延用にコンフィグレーションされている場合、メッセージがロールバックまたは回復されると、メッセージは再配信の遅延が過ぎるまで止められます。メッセージは、遅延の期間を過ぎた時点で再配信できるようになります。

セッションによって消費された結果、ロールバックまたは回復したすべてのメッセージは、ロールバックまたは回復時にそのセッションの再配信遅延を受信します。単一のユーザ トランザクションの一部として複数のセッションで消費されたメッセージは、個々のメッセージを消費したセッションの機能として異なる再配信遅延を受信します。意識的か、または失敗の結果、クライアントによる確認応答またはコミットされていないメッセージには、再配信遅延が割り当てられません。

再配信遅延の設定

セッションは、作成時に接続ファクトリから再配信遅延を継承します。接続ファクトリの `RedeliveryDelay` 属性は、Administration Console でコンフィグレーションします。詳細については、Administration Console オンライン ヘルプの「[JMS 接続ファクトリ]」を参照してください。

セッションを作成するアプリケーションは、`javax.jms.Session` インタフェースに対する WebLogic 固有の拡張を使用して接続ファクトリ設定をオーバーライドできます。セッション属性は動的なので、いつでも変更できます。セッションの再配信遅延を変更すると、変更後にそのセッションで消費およびロールバック (または回復) されるすべてのメッセージに影響します。

セッションに対して再配信遅延を設定するメソッドは、javax.jms.Session インタフェースの拡張である weblogic.jms.extensions.WLSession インタフェースを介して提供されます。セッションの再配信遅延を定義するには、以下のメソッドを使用します。

```
public void setRedeliveryDelay(
  long redeliveryDelay
) throws JMSException;

public long getRedeliveryDelay(
) throws JMSException;
```

WLSession クラスの詳細については、weblogic.jms.extensions.WLSession Javadoc を参照してください。

送り先での再配信遅延のオーバーライド

再配信遅延がセッションで設定されているかどうかに関係なく、メッセージがロールバックまたは回復される送り先で再配信遅延の設定をオーバーライドできます。メッセージの再配信に割り当てられた再配信遅延のオーバーライドは、メッセージがロールバックまたは回復されるときに有効になります。

送り先の RedeliveryDelayOverride 属性は、Administration Console でコンフィグレーションします。詳細については、Administration Console オンラインヘルプの「[JMS 送り先]」を参照してください。

メッセージの再配信制限の設定

WebLogic JMS によるアプリケーションへのメッセージ再配信の試行回数に対して制限を設けることができます。WebLogic JMS が送り先へのメッセージ再配信を指定した回数だけ試みて失敗すると、メッセージの送り先に関連付けられたエラー送り先にメッセージをリダイレクトできます。再配信の制限がコンフィグレーションされているにもかかわらずエラー送り先がコンフィグレーションされていない場合、再配信の上限に達すると永続メッセージや非永続メッセージは削除されます。

メッセージ プロデューサの set メソッドを使用して、再配信制限の値を動的に設定することもできます。詳細については、4-28 ページの「メッセージ プロデューサ属性の設定」を参照してください。

メッセージの再配信制限のコンフィグレーション

送り先によるコンシューマへのメッセージ再配信の試行が指定した再配信制限に達すると、送り先はメッセージを配信不能にします。`RedeliveryLimit` 属性は、送り先で設定され、Administration Console でコンフィグレーションできます。この設定によって、メッセージ プロデューサで設定した再配信制限がオーバライドされます。詳細については、Administration Console オンライン ヘルプの「[JMS 送り先]」を参照してください。

配信されなかったメッセージに対するエラー送り先のコンフィグレーション

配信されなかったメッセージのエラー送り先をコンフィグレーションすると、メッセージが配信不能になったときに、指定したエラー送り先にリダイレクトされます。エラー送り先は、キューでもトピックでもかまいませんが、定義した送り先と同じ JMS サーバでコンフィグレーションする必要があります。エラー送り先がコンフィグレーションされていない場合、配信できないメッセージは削除されます。

送り先の `ErrorDestination` 属性は、Administration Console でコンフィグレーションします。詳細については、Administration Console オンライン ヘルプの「[JMS 送り先]」を参照してください。

メッセージの再配信試行が既に指定した制限に達しているものの、エラー送り先も最大割り当てに達している場合、メッセージは配信不能になり、削除されます。永続的メッセージは格納され、サーバを再起動すると元の送り先 (エラー送り先ではない) に再表示される一方で、非永続的メッセージは削除されます。いずれの場合も、ログ メッセージが生成されます。ログ ファイルへの書き込みが滞るのを防ぐために、エラーが解決されるまで、ログ メッセージは、エラー送り先ごとに 5 分おきに 1 回だけ生成されます。

パッシブなメッセージの有効期限ポリシー

WebLogic JMS には、パッシブなメッセージの有効期限ポリシーしかありません。つまり、期限切れのメッセージが積極的に検索されたりシステムから削除されたりすることはありません。期限切れのメッセージは、以下の場合にのみシステムから削除されます。

- メッセージが消費されそうになったとき
- JMS サーバが再起動されたとき

期限切れのメッセージは積極的には検索されないため、システムに蓄積されてシステム リソースを過度に消費するおそれがある点に注意してください。

メッセージ配信時間の設定

アプリケーションへのメッセージ配信を、将来の指定した時間にスケジューリングできます。メッセージ配信は、短期間 (秒や分など) でも長期間 (数時間単位でバッチ処理する場合など) でもかまいません。その配信時間になって配信されるまでメッセージは表示されないため、将来の特定の時間に作業をスケジューリングできます。

メッセージが送信されるのは 1 回だけで、繰り返し送信されることはありません。メッセージが繰り返し送信されるようにするには、受信したスケジューリング済みのメッセージを元の送信先に戻す必要があります。1 度のみというセマンティクスを保証するため、受信、送信、およびこれらに関連する作業は同じトランザクションのもとで行われます。

プロデューサに対する配信時間の設定

個々のプロデューサに対する配信時間の設定および取得のサポートは、`javax.jms.MessageProducer` インタフェースの拡張である `weblogic.jms.extensions.WLMessageProducer` インタフェースを介して提供されます。個々のプロデューサに対して配信時間を定義するには、以下のメソッドを使用します。

```
public void setTimeToDeliver(
  long timeToDeliver
) throws JMSException;

public long getTimeToDeliver(
) throws JMSException;
```

WLMessageProducer クラスの詳細については、weblogic.jms.extensions.WLMessageProducer Javadoc を参照してください。

メッセージに対する配信時間の設定

DeliveryTime は、メッセージを配信できる最も早い絶対時間を定義する JMS メッセージ ヘッダです。つまり、メッセージはメッセージング システムによって保持され、その時間になるまでどのコンシューマにも配信されません。

DeliveryTime は、JMS ヘッダ フィールドとして送り先でのメッセージのソート、またはメッセージの選択に使用できます。データ型変換の目的で、配信時間は長整数として保存されます。

注意： メッセージで配信時間の値を設定しても、DeliveryTime フィールドに影響はありません。その理由は、メッセージが送信またはパブリッシュされるときに JMS が常にプロデューサの値でその値をオーバーライドするからです。ここで説明されているメッセージ配信時間メソッドは、プロデューサを通じて設定される他の JMS メッセージ フィールド (配信モード、優先順位、配信時間、存続時間、再配信遅延、および再配信制限フィールドなど) と類似しています。具体的に言うと、それらのフィールドの設定は WebLogic JMS などの JMS プロバイダ用に予約されています。

個々のメッセージに対する配信時間の設定および取得のサポートは、javax.jms.Message インタフェースの拡張である weblogic.jms.extensions.WLMessage インタフェースを介して提供されます。メッセージの配信時間を定義するには、以下のメソッドを使用します。

```
public void setJMSDeliveryTime(
  long deliveryTime
) throws JMSException;

public long getJMSDeliveryTime(
) throws JMSException;
```

WLMessage クラスの詳細については、weblogic.jms.extensions.WLMessage Javadoc を参照してください。

配信時間のオーバーライド

作成されたプロデューサは、接続の作成に使用する接続ファクトリ (プロデューサもそこに含まれます) から TimeToDeliver 属性 (ミリ秒) を継承します。配信時間がプロデューサで設定されているかどうかに関係なく、メッセージが送信またはパブリッシュされる送り先で配信時間の設定をオーバーライドできます。管理者は、相対形式またはスケジューリング済み文字列形式のいずれかで送り先に対する TimeToDeliverOverride 属性を設定できます。

存続時間の値との関係

指定した存続時間の値 (JMSExpiration) が指定した配信時間と同じかそれより少ない場合、メッセージの配信は成功します。ただし、メッセージは通知されずに期限切れになります。

相対配信時間のオーバーライドの設定

相対 TimeToDeliverOverride は、整数で指定した文字列で、Administration Console でコンフィグレーション可能です。詳細については、Administration Console オンライン ヘルプの「[JMS 送り先]」を参照してください。

スケジューリング済み配信時間のオーバーライドの設定

スケジューリング済み TimeToDeliverOverride は weblogic.jms.extensions.schedule クラスを使用して指定できます。このクラスは、スケジュールを取得し、指定されたメッセージ配信時間を返すメソッドを提供します。

表 4-3 配信時間のスケジュールの例

例	説明
0 0 0,30 * * * *	次の最も近い 30 分

表 4-3 配信時間のスケジュールの例 (続き)

例	説明
`* * 0,30 4-5 * * *`	午前 4 時から午前 5 時までの 30 分の任意の最初の分
`* * * 9-16 * * *`	午前 9 時と午後 5 時の間 (9:00.00 A.M. ～ 4:59.59 P.M.)
`* * * * 8-14 * 2`	その月の第 2 火曜日
`* * * 13-16 * * 0`	日曜日の午後 1 時と午後 5 時の間
`* * * * * 31 *`	その月の最後の日
`* * * * 15 4 1`	次に日曜日になる 4 月 15 日
`0 0 0 1 * * 2-6;0 0 0 2 * * 1,7`	平日の午前 1 時および週末の午前 2 時

cron に似た文字列がスケジュールの定義に使用されます。書式は、次の BNF 構文で定義されます。

```
schedule := millisecond second minute hour dayOfMonth month
            dayOfWeek
```

second フィールドを指定するための BNF 構文は次のとおりです。

```
second      := * | secondList
secondList  := secondItem [, secondList]
secondItem  := secondValue | secondRange
SecondRange := secondValue - secondValue
```

ミリ秒、分、時、日、月、曜日に対する同様の BNF 文を 2 番目の構文から取得できます。各フィールドの値は、以下の範囲の正の整数で指定します。

```
milliSecondValue := 0-999
milliSecondValue := 0-999
secondValue      := 0-59
minuteValue      := 0-59
hourValue        := 0-23
dayOfMonthValue  := 1-31
monthValue       := 1-12
dayOfWeekValue   := 1-7
```

注意 : これらの値は、`monthValue` を除いて、`java.util.Calendar` クラスで使用するのと同じ範囲です。`monthValue` の `java.util.Calendar` の範囲は、1-12 ではなく 0-11 です。

この構文を使用すると、2 つの時間の間のすべての時間を示す値の範囲で各フィールドを表すことができます。たとえば、dayOfWeek フィールドの 2-6 は、月曜から金曜まで (双方の曜日を含む) を示します。また、カンマ区切りのリストで各フィールドを指定することもできます。たとえば、分フィールドの 0,15,30,45 は、1 時間内の 15 分おきの値を示します。さらに、個々の値と値の範囲の組み合わせで各フィールドを定義することもできます。たとえば、時フィールドの 9-17,0 は、午前 9 時と午後 5 時の間と深夜 12 時を示します。

補足のセマンティクスは以下のとおりです。

- セミコロン (;) で区切って複数のスケジュールが指定されている場合、次のスケジュール時間は、最も早い値を返すスケジュールを基に決定されます。この使い方としては、曜日を基に変更されるスケジュールを指定する場合などがあります (下記の最後の例を参照してください)。
- dayOfWeek の値 1 は日曜日です。
- 値 * は、そのフィールドのあらゆる時間を示します。たとえば、月フィールドの * は、毎月という意味です。時フィールドの * は、毎時という意味です。
- 値 l、つまり last (大文字と小文字の区別はありません) は、そのフィールドで使用できる値で最も大きな値を示します。
- 日に対して月の最大値を超える値を指定すると、その月の最大値が指定されます。たとえば、うるう年の 2 月に対して 31 を指定した場合、スケジューラは、29 と見なしてスケジューリングします。これは、月フィールドを 31 に設定すると、その月の最後の日になるということです。
- ミリ秒が指定される場合、1 秒内で最も近い 50 番目の値に丸められます。値は 0、19、39、59、...、979、および 999 です。したがって、0 ～ 40 は 0 ～ 39 に丸められ、50 ～ 999 は 39 ～ 999 に丸められます。

注意： このクラスの静的メソッドのいずれか 1 つに対するメソッド パラメータとして Calendar が指定されない場合、使用されるカレンダーは、デフォルトの java.util.TimeZone およびデフォルトの java.util.Locale がある java.util.GregorianCalendar です。

JMS スケジュール インタフェース

weblogic.jms.extensions.schedule クラスには、反復時間式に一致する次のスケジュール時間を返すメソッドがあります。この式は、TimeToDeliverOverride と同じ構文を使用します。ミリ秒で返される時間は、相対でも絶対でもかまいません。

WLSession クラスの詳細については、weblogic.jms.extensions.Schedule Javadoc を参照してください。

次のメソッドを使用すると、指定した時間の後の次のスケジュール時間を定義できます。

```
public static Calendar nextScheduledTime(
    String schedule,
    Calendar calendar
  ) throws ParseException {
```

次のメソッドを使用すると、現在の時間の後の次のスケジュール時間を定義できます。

```
public static Calendar nextScheduledTime(
    String schedule,
  ) throws ParseException {
```

次のメソッドを使用すると、指定した時間の後の次のスケジュール時間を絶対ミリ秒で定義できます。

```
public static long nextScheduledTimeInMillis(
    String schedule,
    long timeInMillis
  ) throws ParseException
```

次のメソッドを使用すると、指定した時間の後の次のスケジュール時間を相対ミリ秒で定義できます。

```
public static long nextScheduledTimeInMillisRelative(
    String schedule,
    long timeInMillis
  ) throws ParseException {
```

次のメソッドを使用すると、現在の時間の後の次のスケジュール時間を相対ミリ秒で定義できます。

```
public static long nextScheduledTimeInMillisRelative(
    String schedule
  ) throws ParseException {
```

接続の管理

以下の節では、接続を管理する方法について説明します。

- 接続例外リスナの定義
- 接続メタデータへのアクセス
- 接続の開始、停止、クローズ

接続例外リスナの定義

例外リスナは、接続に問題が発生するとアプリケーションに非同期的に通知します。このメカニズムは、通知されない限り接続がメッセージの消費を待ち続ける場合に役立ちます。

注意： 例外リスナの目的は、接続によって送出されたすべての例外をモニタすることではなく、本来なら配信されない例外を配信することです。

接続に対する例外リスナを定義するには、次の `Connection` メソッドを使用します。

```
public void setExceptionListener(
  ExceptionListener listener
) throws JMSException
```

接続に対する `ExceptionListener` オブジェクトを指定しなければなりません。

JMS プロバイダは、接続の問題を発見すると、次の `ExceptionListener` メソッドを使用して例外リスナ (定義されている場合) に通知します。

```
public void onException(
  JMSException exception
)
```

JMS プロバイダは、このメソッドを呼び出すときに、問題を説明する例外を指定します。

接続に対する例外リスナにアクセスするには、次の Connection メソッドを使用します。

```
public ExceptionListener getExceptionListener(
) throws JMSException
```

接続メタデータへのアクセス

特定の接続に関連付けられているメタデータにアクセスするには、次の Connection メソッドを使用します。

```
public ConnectionMetaData getMetaData(
) throws JMSException
```

このメソッドは、JMS メタデータへのアクセスに使用する ConnectionMetaData オブジェクトを返します。次の表に、JMS メタデータのタイプと、それらにアクセスするために使用できる get メソッドを示します。

JMS メタデータ	get メソッド
バージョン	`public String getJMSVersion( ) throws JMSException`
メジャー バージョン	`public int getJMSMajorVersion( ) throws JMSException`
マイナー バージョン	`public int getJMSMinorVersion( ) throws JMSException`
プロバイダ名	`public String getJMSProviderName( ) throws JMSException`
プロバイダ バージョン	`public String getProviderVersion( ) throws JMSException`
プロバイダ メジャー バージョン	`public int getProviderMajorVersion( ) throws JMSException`
プロバイダ マイナー バージョン	`public int getProviderMinorVersion( ) throws JMSException`
JMSX プロパティ名	`public Enumeration getJMSXPropertyNames( ) throws JMSException`

ConnectionMetaData クラスの詳細については、javax.jms.ConnectionMetaData Javadoc を参照してください。

接続の開始、停止、クローズ

メッセージの流れを制御するために、以下の start() メソッドと stop() メソッドをそれぞれ使用して、接続を一時的に開始および停止できます。

start() メソッドと stop() メソッドの詳細は以下のとおりです。

```
public void start(
) throws JMSException
```

```
public void stop(
) throws JMSException
```

新しく作成された接続は停止しています。ー接続が開始されるまで、メッセージは受信されません。一般に、他の JMS オブジェクトは、4-4 ページの「JMS アプリケーションの設定」で説明するとおり、接続が開始する前からメッセージを処理するよう設定されます。メッセージは、停止している接続上に作成することはできますが、停止している接続に届けることはできません。

接続が開始されていれば、stop() メソッドを使用して接続を停止できます。このメソッドは、次の手順を実行します。

- すべてのメッセージの配信を中断する。接続が再開されるか、そのメッセージに関連付けられている存続時間に達するまで、メッセージの受信を待っているアプリケーションは何も返しません。
- 現在メッセージを処理しているすべてのメッセージ リスナが完了するまで待機する。

一般に、JMS プロバイダは接続を作成するときに大量のリソースを割り当てます。接続が使用されなくなったら、その接続をクローズしてリソースを解放する必要があります。接続をクローズするには、次のメソッドを使用します。

```
public void close(
) throws JMSException
```

このメソッドは、次の手順を実行して系統的にシャットダウンを行います。

- 保留中のすべてのメッセージの受信を終了させる。アプリケーションは、クローズ時にメッセージを受信できない場合はメッセージまたは null を返す場合があります。
- 現在メッセージを処理しているすべてのメッセージ リスナが完了するまで待機する。
- 処理中のトランザクションを、そのトランザクション セッション上でロールバックする (こうしたトランザクションが外部 JTA ユーザ トランザクションの一部である場合を除く)。JTA ユーザ トランザクションの詳細については、5-5 ページの「JTA ユーザ トランザクションの使い方」を参照してください。
- クライアントが確認応答を行うセッションの確認応答は強制しない。確認応答を強制しないことにより、キューおよび信頼性の高い処理が要求される恒久サブスクリプション用のメッセージが失われなくなります。

接続をクローズすると、関連付けられているすべてのオブジェクトがクローズされます。受信メッセージの `acknowledge()` メソッドを除いて、接続で作成または受信されたメッセージ オブジェクトは引き続き使用できます。閉じた接続をクローズしても影響はありません。

注意 : クローズされた接続のセッションから受信したメッセージを確認応答しようとすると、`IllegalStateException` が送出されます。

セッションの管理

以下の節では、セッションを管理する方法について説明します。

- セッション例外リスナの定義
- セッションのクローズ

セッション例外リスナの定義

例外リスナは、セッションに問題が発生すると、クライアントに非同期的に通知します。これは、通知されない限りセッションがメッセージの消費を待ち続ける場合に役立ちます。

注意： 例外リスナの目的は、セッションによって送出されたすべての例外をモニタすることではなく、本来なら通知されない例外を通知することです。

セッションに対する例外リスナを定義するには、次の `WLSession` メソッドを使用します。

```
public void setExceptionListener(
  ExceptionListener listener
) throws JMSException
```

セッションに対する `ExceptionListener` オブジェクトを指定しなければなりません。

JMS プロバイダは、セッションの問題を発見すると、次の `ExceptionListener` メソッドを使用して例外リスナ (定義されている場合) に通知します。

```
public void onException(
  JMSException exception
)
```

JMS プロバイダは、このメソッドを呼び出すときに、問題を説明する例外を指定します。

セッションに対する例外リスナにアクセスするには、次の `WLSession` メソッドを使用します。

```
public ExceptionListener getExceptionListener(
) throws JMSException
```

注意： 1 つのセッションに対して 1 つのスレッドしか存在しないので、例外リスナとメッセージ リスナ (非同期メッセージ配信用に使用される) を同時に実行することはできません。そのため、問題が発生したときにメッセージ リスナが実行されている場合、そのメッセージ リスナが実行を完了するまで例外リスナはブロックされます。メッセージ リスナの詳細については、4-31 ページの「メッセージの非同期受信」を参照してください。

セッションのクローズ

接続と同じように、JMS プロバイダはセッションを作成するときに大量のリソースを割り当てます。セッションが使用されなくなったら、そのセッションをクローズしてリソースを解放することをお勧めします。セッションをクローズするには、次の Session メソッドを使用します。

```
public void close(
) throws JMSException
```

注意： close() メソッドは、セッション スレッドとは別個のスレッドから呼び出すことができる唯一の Session メソッドです。

このメソッドは、次の手順を実行して系統的にシャットダウンを行います。

- 保留中のすべてのメッセージの受信を終了させる。アプリケーションは、クローズ時にメッセージを受信できない場合はメッセージまたは null を返す場合があります。
- 現在メッセージを処理しているすべてのメッセージ リスナが完了するまで待機する。
- 処理中のトランザクションをロール バックする (こうしたトランザクションが外部 JTA ユーザ トランザクションの一部である場合を除く)。JTA ユーザ トランザクションの詳細については、5-5 ページの「JTA ユーザ トランザクションの使い方」を参照してください。
- クライアントが確認応答を行うセッションの確認応答は強制しない。これにより、キューおよび信頼性の高い処理が要求される恒久サブスクリプション用のメッセージが失われなくなります。

セッションをクローズすると、関連付けられているすべてのプロデューサとコンシューマもクローズされます。

注意： onMessage() メソッド呼び出し内で close() メソッドを発行する場合、システム管理者は接続ファクトリをコンフィグレーションするときに [メッセージの短縮を許可] チェック ボックスを選択しなければなりません。詳細については、Administration Console オンライン ヘルプの「[JMS 接続ファクトリ]」を参照してください。

送り先の動的作成

以下のいずれかを使用して、送り先を動的に作成できます。

- `weblogic.jms.extensions.JMSHelper` クラス メソッド
- 一時的な送り先

送り先の動的作成に関する手順については、以降の節で説明します。

JMSHelper クラス メソッドの使い方

以下の各 `JMSHelper` メソッドを使用してキューまたはトピックを作成する非同期リクエストを動的に送信できます。

```
static public void createPermanentQueueAsync(
  Context ctx,
  String jmsServerName,
  String queueName,
  String jndiName
) throws JMSException

static public void createPermanentTopicAsync(
  Context ctx,
  String jmsServerName,
  String topicName,
  String jndiName
) throws JMSException
```

JNDI 初期コンテキスト、送り先に関連付けられる JMS サーバの名前、送り先 (キューまたはトピック) の名前、および JNDI ネームスペース内で送り先をルックアップする場合に使用する名前を指定する必要があります。

各メソッドによって、以下のものが更新されます。

- 動的に作成された送り先を含む、指定されたドメインに関連付けられているコンフィグレーション ファイル
- 送り先を公開する JNDI ネームスペース

注意： いずれのメソッド呼び出しも、例外を送出せずに失敗する場合があります。また、例外が送出されても、それが必ずしもメソッド呼び出しの失敗を示しているとは限りません。

JMS サーバでの送り先の作成と JNDI ネームスペースへの情報の伝播には、時間がかかる場合があります。複数のサーバを使用している環境では、伝播の遅延が増大します。JNDI ルックアップを実行するよりも、`createQueue()` メソッドまたは `createTopic()` メソッドを使用して、それぞれキューまたはトピックの存在をテストすることをお勧めします。この方法によって、伝播固有の遅延を、ある程度回避できます。

たとえば、次に示す `findQueue()` メソッドは、動的に作成されたキューにアクセスしようとしますが、アクセスに失敗すると再試行まで、指定された間隔スリープします。無限ループを回避するために、再試行の最大回数が設定されています。

```
private static Queue findQueue (
  QueueSession queueSession,
  String jmsServerName,
  String queueName,
  int retryCount,
  long retryInterval
) throws JMSException
{
   String wlsQueueName = jmsServerName + "/" + queueName;
   String command = "QueueSession.createQueue(" +
       wlsQueueName + ")";
   long startTimeMillis = System.currentTimeMillis();
   for (int i=retryCount; i>=0; i--) {
       try {
         System.out.println("Trying " + command);
         Queue queue = queueSession.createQueue(wlsQueueName);
         System.out.println(command + "succeeded after " +
              (retryCount - i + 1) + " tries in " +
              (System.currentTimeMillis() - startTimeMillis) +
              " millis.");
         return queue;
       } catch (JMSException je) {
         if (retryCount == 0) throw je;
       }
       try {
         System.out.println(command + “> failed, pausing “ +
              retryInterval + " millis.");
         Thread.sleep(retryInterval);
       } catch (InterruptedException ignore) {}

   }
   throw new JMSException("out of retries");
}
```

この場合、JMSHelper クラス メソッドの後に findQueue() メソッドを呼び出すことで、動的に作成されたキューを使用可能になりしだい、取り出すことができます。次に例を示します。

```
JMSHelper.createPermanentQueueAsync(ctx, domain, jmsServerName,
  queueName, jndiName);
Queue queue = findQueue(qsess, jmsServerName, queueName,
  retry_count, retry_interval);
```

JMSHelper クラスの詳細については、weblogic.jms.extensions.JMSHelper Javadoc を参照してください。

一時的な送り先の使い方

一時的な送り先を使用することで、サーバ定義の送り先のコンフィグレーションと作成に伴うシステム管理のオーバーヘッドを発生させずに、必要に応じてアプリケーションで送り先を作成できます。

WebLogic JMS サーバでは、JMSReplyTo ヘッダ フィールドを使用して、アプリケーションに応答を返すことができます。アプリケーションでは、オプションとして、メッセージの JMSReplyTo ヘッダ フィールドをその一時的な送り先に設定することで、使用している一時的な送り先を別のアプリケーションに対して公開できます。

一時的な送り先は、4-56 ページの「一時的な送り先の削除」にある説明のとおりに delete() メソッドを使用して削除されない限り、現在の接続が継続している間だけ存在します。

サーバが再起動されると、メッセージは使用できなくなるので、すべての PERSISTENT メッセージは自動的に NON_PERSISTENT メッセージになります。そのため、一時的な送り先は、再起動によるデータの消失が許されないビジネス ロジックには適していません。

注意： 一時的な送り先 (キューまたはトピック) を作成する前に、Administration Console を使用して、一時的な送り先を使用する JMS サーバをコンフィグレーションする必要があります。そのためには、JMS サーバの Temporary Template 属性を使用して、同じドメイン内でコンフィグレーションされる JMS テンプレートを選択します。JMS サーバのコンフィグレーションの詳細については、Administration Console オンライン ヘルプの「[JMS サーバ]」を参照してください。

以降の節では、一時的なキュー (PTP) または一時的なトピック (Pub/Sub) の作成方法について説明します。

一時的なキューの作成

次の QueueSession メソッドを使用して、一時的なキューを作成できます。

```
public TemporaryQueue createTemporaryQueue(
) throws JMSException
```

たとえば、現在の接続が継続している間だけ存在する TemporaryQueue への参照を作成するには、次のメソッド呼び出しを使用します。

```
QueueSender = Session.createTemporaryQueue();
```

一時的なトピックの作成

次の TopicSession メソッドを使用して、一時的なトピックを作成できます。

```
public TemporaryTopic createTemporaryTopic(
) throws JMSException
```

たとえば、現在の接続が継続している間だけ存在する一時的なトピックへの参照を作成するには、次のメソッド呼び出しを使用します。

```
TopicPublisher = Session.createTemporaryTopic();
```

一時的な送り先の削除

一時的な送り先の使用が終了したら、次の TemporaryQueue メソッドまたは TemporaryTopic メソッドを使用して送り先を削除し、関連リソースを解放できます。

```
public void delete(
) throws JMSException
```

恒久サブスクリプションの設定

WebLogic JMS では、恒久サブスクリプションおよび非恒久サブスクリプションがサポートされています。

恒久サブスクリプションの場合、WebLogic JMS では、メッセージはサブスクライバに配信されるか、または期限切れになるまで永続ファイルまたはデータベースに格納されます。この場合、メッセージの配信時にサブスクライバがアクティブな状態でなくてもかまいません。サブスクライバを表す Java オブジェクトが存在していれば、サブスクライバはアクティブであるとみなされます。恒久サブスクリプションは、Pub/Sub メッセージングのみでサポートされています。

注意： 恒久サブスクリプションは、分散トピックに対しては作成できません。ただし、分散トピックのメンバーに対して恒久サブスクリプションを作成することはできます。こうすると、他のトピック メンバーは、恒久サブスクリプションを持つメンバーにメッセージを転送します。分散トピックの使い方の詳細については、4-94 ページの「分散送り先の使用」を参照してください。

非恒久サブスクリプションの場合、WebLogic JMS では、メッセージはアクティブ セッションを持つアプリケーションにのみ配信されます。アプリケーションがリスンしていない間にトピックへ送信されたメッセージは、二度とそのアプリケーションに配信されません。つまり、非恒久サブスクリプションは、そのサブスクライバ オブジェクトが存在している間だけ存続します。デフォルトでは、サブスクライバは非恒久です。

以降の節で説明する内容は、次のとおりです。

- 永続ストアの定義
- クライアント ID の定義
- 恒久サブスクリプション用のサブスクライバの作成
- 恒久サブスクリプションの削除
- 恒久サブスクリプションの変更
- 恒久サブスクリプションの管理

永続ストアの定義

メッセージがサブスクライバに配信されるまで、またはメッセージの期限が切れるまで格納しておくには、永続ファイル ストアまたは永続データベース ストアをコンフィグレーションして JMS サーバに割り当てる必要があります。

- [ストア] ノードを使用して、JMS ファイル ストアまたは JMS JDBC バッキング ストアを作成する。
- [JMS サーバ | コンフィグレーション | 一般] タブの [ストア] フィールドのドロップダウン リストで、コンフィグレーションしたストアを選択して JMS サーバに割り当てる。

  注意 : 2 つの JMS サーバで同じバッキング ストアを使用することはできません。

JMS ストアのコンフィグレーションの詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

クライアント ID の定義

恒久サブスクリプションをサポートするには、接続に対してクライアント ID を定義する必要があります。

注意 : JMS クライアント ID は、WebLogic セキュリティ レルムでのユーザ認証で使用される WebLogic Server ユーザ名とは必ずしも一致しません。JMS アプリケーションに適合していれば、JMS クライアント ID を WebLogic Server ユーザ名に設定することは当然可能です。

クライアント ID は、以下の 2 つの方法で設定できます。

- 1 つ目の方法は、クライアント ID を使用する接続ファクトリをコンフィグレーションする方法です。WebLogic JMS では、この方法は各クライアント ID のコンフィグレーション時に別々の接続ファクトリの定義を追加することになります。アプリケーションでは、JNDI でそれ自身のトピック接続ファクトリがルックアップされ、それを使用して自身のクライアント ID を含む接続が作成されます。クライアント ID を使用する接続ファクトリのコンフィ

グレーションの詳細については、Administration Console オンライン ヘルプの「[JMS 接続ファクトリ]」を参照してください。

- もう 1 つのより望ましい方法は、接続を作成してから、アプリケーションで次の Connection メソッドを呼び出して、接続にクライアント ID を設定する方法です。

```
public void setClientID(
  String clientID
) throws JMSException
```

ユニークなクライアント ID を指定する必要があります。この代替方法を使用する場合は、デフォルトの接続ファクトリを使用すると (アプリケーションに適合している場合)、コンフィグレーション情報を変更する必要がありません。ただし、恒久サブスクリプションに対応しているアプリケーションの場合は、トピック接続を作成したらすぐに `setClientID()` を呼び出すようにする必要があります。デフォルトの接続ファクトリについては、『管理者ガイド』の「JMS の管理」を参照してください。

クライアント ID が接続に対して既に定義されている場合は、`IllegalStateException` が送出されます。指定したクライアント ID が別の接続に対して既に定義されている場合は、`InvalidClientIDException` が送出されます。

注意： `setClientID()` メソッドを使用してクライアント ID を指定する場合には、重複したクライアント ID が例外の送出なしに指定されてしまう危険性があります。たとえば、2 つの異なる接続に対して、同じ値を持つクライアント ID が同時に設定されると、競合状態のまま、同じ値が両方の接続に割り当てられる場合があります。このような重複の危険性を回避するには、コンフィグレーション時にクライアント ID を指定します。

クライアント ID を表示し、クライアント ID が既に定義されているかどうかをテストするには、次の Connection メソッドを使用します。

```
public String getClientID(
) throws JMSException
```

注意： 恒久サブスクリプションのサポートは、Pub/Sub メッセージング モデル固有の機能なので、クライアント ID はトピック接続でしか使用できません。キュー接続にもクライアント ID がありますが、JMS では使用されません。

恒久サブスクリプションは、一時的なトピックに対しては作成しないでください。一時的なトピックは、現在の接続が継続している間だけ存在するように設計されているからです。

恒久サブスクリプション用のサブスクライバの作成

以下の TopicSession メソッドを使用して、恒久サブスクリプション用のサブスクライバを作成できます。

```
public TopicSubscriber createDurableSubscriber(
  Topic topic,
  String name
) throws JMSException

public TopicSubscriber createDurableSubscriber(
  Topic topic,
  String name,
  String messageSelector,
  boolean noLocal
) throws JMSException
```

サブスクライバを作成するトピックの名前と恒久サブスクリプションの名前を指定する必要があります。また、メッセージをフィルタ処理するためのメッセージセレクタ、および noLocal フラグ (この節で後述) を指定することもできます。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。messageSelector を指定しない場合、デフォルトではすべてのメッセージが検索されます。

アプリケーションでは、JMS 接続を使用して同じトピックに対してパブリッシュとサブスクライブの両方を行うことができます。トピック メッセージはすべてのサブスクライバに配信されるので、アプリケーションは自身がパブリッシュしたメッセージを受信する可能性があります。これを防ぐために、JMS アプリケーションは noLocal フラグを true に設定できます。noLocal 値は、デフォルトでは false になっています。

恒久サブスクリプション名は、クライアント ID ごとにユニークである必要があります。接続に対するクライアント ID の定義については、4-58 ページの「クライアント ID の定義」を参照してください。

特定の恒久サブスクリプション用のサブスクライバをいつでも定義できるのは、1 つのセッションだけです。複数のサブスクライバが恒久サブスクリプションにアクセスできますが、同時にはアクセスできません。恒久サブスクリプションはファイルまたはデータベースに格納されます。

恒久サブスクリプションの削除

恒久サブスクリプションを削除するには、次の `TopicSession` メソッドを使用します。

```
public void unsubscribe(
  String name
) throws JMSException
```

削除する恒久サブスクリプションの名前を指定する必要があリます。

以下の条件のいずれかに当てはまる場合は、恒久サブスクリプションを削除できません。

- `TopicSubscriber` がセッションでまだアクティブな場合
- 恒久サブスクリプションで受信したメッセージがトランザクションの一部であるか、またはセッションでまだ確認応答されていない場合

注意： Administration Console を使用して恒久サブスクリプションを削除することもできます。恒久サブスクリプションの管理の詳細については、4-62 ページの「恒久サブスクリプションの管理」を参照してください。

恒久サブスクリプションの変更

恒久サブスクリプションを変更するには、以下の手順を実行します。

1. 4-61 ページの「恒久サブスクリプションの削除」にある説明に従って、恒久サブスクリプションを削除します。

   この手順は省略可能です。この手順が明示的に実行されない場合は、次の手順で恒久サブスクリプションが再作成されるときに暗黙的に削除が行われます。

2. 4-60 ページの「恒久サブスクリプション用のサブスクライバの作成」で説明されているメソッドを使用して、同じ名前の恒久サブスクリプションを再作成します。ただし、トピック名、メッセージ セレクタ、`noLocal` のいずれかには異なる値を指定します。

   新しい値に基づいて、恒久サブスクリプションが再作成されます。

注意： 恒久サブスクリプションを再作成する場合には、重複した名前を持つ恒久サブスクリプションを作成しないよう注意してください。たとえば、使用できない JMS サーバから恒久サブスクリプションを削除しようとすると、削除は失敗します。続いて、別の JMS サーバで同じ名前の恒久サブスクリプションを作成すると、最初の JMS サーバが使用可能になったときに予期しない結果が生じることがあります。元の恒久サブスクリプションが削除されていないため、最初の JMS サーバが再び使用可能になると、重複した名前の 2 つの恒久サブスクリプションが存在することになります。

恒久サブスクリプションの管理

Administration Console を使用して恒久サブスクリプションをモニタおよび削除することができます。詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの設定と参照

WebLogic JMS には、メッセージの識別および転送を定義できる一連の標準ヘッダ フィールドが用意されています。さらに、プロパティ フィールドを使用すると、標準セットを拡張した、アプリケーション固有のヘッダ フィールドをメッセージ内に含めることができます。メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドを使用して、通信しているプロセス間で情報をやり取りできます。

データをメッセージ本文ではなくプロパティ フィールドに含める主要な理由は、メッセージ セレクタを使用したメッセージのフィルタ処理をサポートするためです。メッセージ セレクタからは、メッセージ本文のデータ (XML メッセージ拡張子を除く) にはアクセスできません。たとえば、プロパティ フィールドを使用して、あるメッセージに高い優先度を割り当てると仮定します。その場合、このプロパティ フィールドにアクセスし、至急の優先度が指定されたメッセージだけを選択するメッセージ セレクタを含むメッセージ コンシューマを作成できます。セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

メッセージ ヘッダ フィールドの設定

JMS メッセージには、常にメッセージと共に送信されるヘッダ フィールドの標準セットが含まれます。これらはメッセージを受信するメッセージ コンシューマで利用でき、また、一部のフィールドはメッセージを送信するメッセージ プロデューサで設定できます。メッセージを受信したら、ヘッダ フィールドの値は変更できます。

標準メッセージ ヘッダ フィールドの詳細については、2-17 ページの「メッセージ ヘッダ フィールド」を参照してください。

次の表に、Message クラスの set メソッドと get メソッドを、サポートされているデータ型ごとに示します。

注意： `set()` メソッドを使用して設定されたヘッダ フィールド値は、`send()` メソッドによってオーバーライドされる場合もあります (次表参照)。

ヘッダ フィールド	set メソッド	get メソッド

`JMSCorrelationID`	`public void setJMSCorrelationID( String correlationID ) throws JMSException`	`public String getJMSCorrelationID( ) throws JMSException`
		
		`public byte[] getJMSCorrelationIDAsBytes( ) throws JMSException`

ヘッダ フィールド	set メソッド	get メソッド
JMSDestination[1]	public void setJMSDestination(Destination destination) throws JMSException	public Destination getJMSDestination( ) throws JMSException
JMSDeliveryMode[1]	public void setJMSDeliveryMode(int deliveryMode) throws JMSException	public int getJMSDeliveryMode( ) throws JMSException
JMSDeliveryTime[1]	public void setJMSDeliveryTime(long deliveryTime) throws JMSException	public long getJMSDeliveryTime( ) throws JMSException
JMSDeliveryMode[1]	public void setJMSDeliveryMode(int deliveryMode) throws JMSException	public int getJMSDeliveryMode( ) throws JMSException

JMSMessageID[1]	public void setJMSMessageID(
	  String id
	) throws JMSException
	set メソッドだけでなく、weblogic.jms.extensions.JMSHelper クラスには、WebLogic JMS 6.0 以前の JMSMessageID 形式を WebLogic JMS 6.1 の形式に (または WebLogic JMS 6.1 の形式をそれ以前の形式に) 変換する、以下のメソッドも用意されている。
	public void
	oldJMSMessageIDToNew(
	  String id,
	  long timeStamp
	) throws JMSException
	
	public void
	newJMSMessageIDToOld(
	  String id,
	  long timeStamp
	) throws JMSException
		public String getJMSMessageID(
		) throws JMSException

ヘッダ フィールド	set メソッド	get メソッド
JMSPriority[1]	public void setJMSPriority(int priority) throws JMSException	public int getJMSPriority( ) throws JMSException
JMSRedelivered[1]	public void setJMSRedelivered(boolean redelivered) throws JMSException	public boolean getJMSRedelivered( ) throws JMSException
JMSRedeliveryLimit[1]	public void setJMSRedeliveryLimit(int redelivered) throws JMSException	public int getJMSRedeliveryLimit( ) throws JMSException
JMSReplyTo	public void setJMSReplyTo(Destination replyTo) throws JMSException	public Destination getJMSReplyTo( ) throws JMSException
JMSTimeStamp[1]	public void setJMSTimeStamp(long timestamp) throws JMSException	public long getJMSTimeStamp( ) throws JMSException
JMSType	public void setJMSType(String type) throws JMSException	public String getJMSType( ) throws JMSException

1. send() メソッドが実行されている場合、対応する set() メソッドは、メッセージ ヘッダ フィールドに影響を与えません。ヘッダ フィールド値が設定されている場合は、send() メソッドの処理中に、このヘッダ フィールド値がオーバーライドされます。

WL_HOME\samples\server\src\examples\jms\sender ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) に収められている WebLogic Server 付属の examples.jms.sender.SenderServlet の例では、送信するメッセージのヘッダ フィールドを設定する方法、および送信後にメッセージのヘッダ フィールドを表示する方法を示します。

たとえば、send() メソッドの後に置く、次のコードは、WebLogic JMS によってメッセージに割り当てられたメッセージ ID を表示します。

```
System.out.println("Sent message " +
       msg.getJMSMessageID() + " to " +
       msg.getJMSDestination());
```

メッセージ プロパティ フィールドの設定

プロパティ フィールドを設定するには、適切な set メソッドを呼び出して、プロパティ名と値を指定します。プロパティ フィールドを参照するには、適切な get メソッドを呼び出して、プロパティ名を指定します。

送信側アプリケーションはメッセージにプロパティを設定でき、受信側アプリケーションはそのプロパティを表示できます。受信側アプリケーションがプロパティを変更するには、まず次の `clearProperties()` メソッドを使用してプロパティを消去する必要があります。

```
public void clearProperties(
) throws JMSException
```

このメソッドは、メッセージ ヘッダ フィールドおよびメッセージ本文は消去しません。

注意： JMS 用に `JMSX` というプロパティ名のプレフィックスが予約されています。接続メタデータには、JMSX プロパティのリストが含まれています。`getJMSXPropertyNames()` メソッドを使用して、列挙リストとして、このリストにアクセスできます。詳細については、4-48 ページの「接続メタデータへのアクセス」を参照してください。

プロバイダ固有のプロパティ用に `JMS_` というプロパティ名のプレフィックスが予約されています。このプレフィックスは標準の JMS メッセージングでは使用できません。

プロパティ フィールドは、boolean、byte、double、float、int、long、short、string の各データ型のいずれかに設定できます。次の表に、Message クラスの set メソッドと get メソッドを、サポートされているデータ型ごとに示します。

表 4-4 メッセージ プロパティのデータ型ごとの set メソッドおよび get メソッド

データ型	set メソッド	get メソッド
boolean	`public void setBooleanProperty( String name, boolean value ) throws JMSException`	`public boolean getBooleanProperty( String name ) throws JMSException`

表 4-4 メッセージ プロパティのデータ型ごとの set メソッドおよび get メソッド (続き)

データ型	set メソッド	get メソッド
byte	`public void setByteProperty( String name, byte value ) throws JMSException`	`public byte getByteProperty( String name ) throws JMSException`
double	`public void setDoubleProperty( String name, double value ) throws JMSException`	`public double getDoubleProperty( String name ) throws JMSException`
float	`public void setFloatProperty( String name, float value ) throws JMSException`	`public float getFloatProperty( String name ) throws JMSException`
int	`public void setIntProperty( String name, int value ) throws JMSException`	`public int getIntProperty( String name ) throws JMSException`
long	`public void setLongProperty( String name, long value) throws JMSException`	`public long getLongProperty( String name ) throws JMSException`
short	`public void setShortProperty( String name, short value ) throws JMSException`	`public short getShortProperty( String name ) throws JMSException`
String	`public void setStringProperty( String name, String value ) throws JMSException`	`public String getStringProperty( String name ) throws JMSException`

上記の表で説明した set メソッドおよび get メソッド以外にも、`setObjectProperty()` メソッドおよび `getObjectProperty()` メソッドを使用して、プロパティのデータ型の具体的なプリミティブ値を使用できます。具体的な値が使用されている場合、プロパティのデータ型はコンパイル時ではなく、実行時に決定されます。有効なオブジェクトのデータ型は、boolean、byte、double、float、int、long、short、および string です。

次の Message メソッドを使用して、すべてのプロパティ フィールド名にアクセスできます。

```
public Enumeration getPropertyNames(
) throws JMSException
```

このメソッドは、すべてのプロパティ フィールド名を列挙して返します。その後、プロパティ フィールドのデータ型に基づいて、上記の表で説明されている適切な get メソッドにプロパティ フィールド名を渡すことで、各プロパティ フィールドの値を取り出すことができます。

次の表は、メッセージ プロパティの変換表です。この表から、読み込み可能なデータ型を、書き込まれたデータ型に基づいて識別できます。

表 4-5 メッセージ プロパティの変換表

書き込まれたプロパティのデータ型	読み込み可能なデータ型							
	boolean	**byte**	**double**	**float**	**int**	**long**	**short**	**String**
boolean	X							X
byte		X			X	X	X	X
double			X					X
float			X	X				X
int					X	X		X
long						X		X
Object	X	X	X	X	X	X	X	X
short					X	X	X	X
String	X	X	X	X	X	X	X	X

次の Message メソッドを使用して、プロパティの値が設定されているかどうかをテストできます。

```
public boolean propertyExists(
  String name
) throws JMSException
```

プロパティ名を指定すると、メソッドはそのプロパティが存在するかどうかを示すブール値を返します。

たとえば、次のコードは、2 種類の String プロパティと 1 種類の int プロパティを設定します。

```
msg.setStringProperty("User", user);
msg.setStringProperty("Category", category);
msg.setIntProperty("Rating", rating);
```

メッセージ プロパティ フィールドの詳細については、2-22 ページの「メッセージ プロパティ フィールド」、または javax.jms.Message Javadoc を参照してください。

メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの参照

注意： 参照できるのは、キューのメッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドだけです。トピックのメッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドは参照できません。

以下の QueueSession メソッドを使用して、キューにあるメッセージのヘッダ フィールドおよびプロパティ フィールドを参照できます。

```
public QueueBrowser createBrowser(
  Queue queue
) throws JMSException

public QueueBrowser createBrowser(
  Queue queue,
  String messageSelector
) throws JMSException
```

参照するキューを指定する必要があります。また、参照するメッセージをフィルタ処理するメッセージ セレクタを指定することもできます。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

キューを定義すると、以下の QueueBrowser メソッドを使用して、キュー ブラウザに関連付けられたキュー名およびメッセージ セレクタにアクセスできるようになります。

```
public Queue getQueue(
) throws JMSException

public String getMessageSelector(
) throws JMSException
```

さらに、次の QueueBrowser メソッドを使用して、メッセージを参照するための列挙値にアクセスできます。

```
public Enumeration getEnumeration(
) throws JMSException
```

WL_HOME\samples\server\src\examples\jms\queue ディレクトリ (*WL_HOME* は WebLogic Platform のインストール先の最上位ディレクトリ) にある WebLogic Server 付属の examples.jms.queue.QueueBrowser の例では、受信されたメッセージのヘッダ フィールドにアクセスする方法を示します。

たとえば、次の QueueBrowser の例からの引用コードでは、QueueBrowser オブジェクトを作成します。

```
qbrowser = qsession.createBrowser(queue);
```

次のコードは、QueueBrowser の例で定義されている displayQueue() メソッドからの引用です。この例では、QueueBrowser オブジェクトを使用して、キューのメッセージをスキャンする場合に使用される列挙値を取得します。

```
   public void displayQueue(
) throws JMSException
  {
    Enumeration e = qbrowser.getEnumeration();
    Message m = null;

    if (! e.hasMoreElements()) {
      System.out.println("There are no messages on this queue.");
    } else {

      System.out.println("Queued JMS Messages: ");
      while (e.hasMoreElements()) {
        m = (Message) e.nextElement();
        System.out.println("Message ID " + m.getJMSMessageID() +
                      " delivered " + new Date(m.getJMSTimestamp())
                          " to " + m.getJMSDestination());
      }
    }
```

キュー ブラウザが使用されていない場合は、キュー ブラウザを閉じてリソースを解放する必要があります。詳細については、4-36 ページの「オブジェクト リソースの解放」を参照してください。

QueueBrowser クラスの詳細については、javax.jms.QueueBrowser Javadoc を参照してください。

メッセージのフィルタ処理

多くの場合、アプリケーションでは、配信されるすべてのメッセージが通知される必要はありません。メッセージ セレクタを使用すると、不要なメッセージをフィルタ処理できるので、ネットワーク トラフィックへの影響が最小限になり、パフォーマンスが向上します。

メッセージ セレクタは、以下のように動作します。

- 送信側アプリケーションでは、標準化された方法でメッセージを説明したり分類したりするためのメッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドが設定される。
- 受信側アプリケーションでは、単純なクエリ文字列を指定することで、アプリケーションで受信するメッセージがフィルタ処理される。

メッセージ セレクタはメッセージの内容 (本文) を参照することができないため、情報の一部をメッセージ プロパティ フィールドに複製することもできます (XML メッセージの場合を除く)。

キュー レシーバまたはトピック サブスクライバの作成時に、それぞれ QueueSession.createReceiver() メソッドまたは TopicSession.createSubscriber() メソッドの引数としてセレクタを指定します。キュー レシーバまたはトピック サブスクライバの作成の詳細については、4-12 ページの「手順 5 : セッションと送り先を使用してメッセージ プロデューサとメッセージ コンシューマを作成する」を参照してください。

以降の節では、SQL 文と XML セレクタ メソッドを使用してメッセージ セレクタを定義する方法、およびメッセージ セレクタを更新する方法について説明します。ヘッダ フィールドおよびプロパティ フィールドの設定の詳細については、それぞれ 4-62 ページの「メッセージ ヘッダ フィールドおよびメッセージ プロパティ フィールドの設定と参照」、または 4-66 ページの「メッセージ プロパティ フィールドの設定」を参照してください。

SQL 文を使用したメッセージ セレクタの定義

メッセージ セレクタはブール式であり、SQL の select 文の where 句と似た構文を持つ文字列です。

次の引用は、セレクタ式の例を示します。

```
salary > 64000 and dept in ('eng','qa')
```

```
(product like 'WebLogic%' or product like '%T3')
       and version > 3.0
```

```
hireyear between 1990 and 1992
       or fireyear is not null
```

```
fireyear - hireyear > 4
```

次の例では、キュー レシーバの作成時に、優先度が 6 未満のメッセージをフィルタで除外するセレクタを設定する方法を示します。

```
String selector = "JMSPriority >= 6";
qsession.createReceiver(queue, selector);
```

次の例では、トピック サブスクライバの作成時に、同様のセレクタを設定する方法を示します。

```
String selector = "JMSPriority >= 6";
qsession.createSubscriber(topic, selector);
```

メッセージ セレクタの構文の詳細については、javax.jms.Message Javadoc を参照してください。

XML セレクタ メソッドを使用した XML メッセージ セレクタの定義

XML メッセージ タイプの場合、メッセージ セレクタの定義には、前節で説明した SQL セレクタ式だけでなく、次のメソッドを使用することもできます。

```
String JMS_BEA_SELECT(String type, String expression)
```

JMS_BEA_SELECT は、WebLogic JMS SQL 構文の組み込み関数です。構文タイプと XPath 式を指定します。構文タイプは、xpath (XML Path Language) に設定する必要があります。XML Path Language は、XML Path Language (XPath) のドキュメントで定義されており、XML Path Language の Web サイト http://www.w3.org/TR/xpath で入手できます。

注意： XML メッセージの構文には十分に注意を払ってください。不正な XML メッセージ (終了タグがないなど) はどの XML セレクタとも一致しません。

以下の環境では、メソッドは null 値を返します。

- メッセージによって解析されない場合
- メッセージによって解析されるが、要素がない場合
- メッセージによって解析され、要素も存在するが、メッセージに値が含まれていない場合 (例 : <order></order>)

たとえば、次のような XML の引用があります。

```
<order>
       <item>
              <id>007</id>
              <name>Hand-held Power Drill</name>
              <description>Compact, assorted colors.</description>
              <price>$34.99</price>
       </item>
       <item>
              <id>123</id>
              <name>Mitre Saw</name>
              <description>Three blades sizes.</description>
              <price>$69.99</price>
       </item>
       <item>
              <id>66</id>
              <name>Socket Wrench Set</name>
              <description>Set of 10.</description>
              <price>$19.99</price>
       </item>
</order>
```

次の例では、上記の引用にある 2 番目の項目の名前を取り出す方法を示します。メソッド呼び出しは、Mitre Saw という文字列を返します。

```
String sel = "JMS_BEA_SELECT('xpath',
'/order/item[2]/name/text()') = 'Mitre Saw'";
```

二重引用符と単一引用符、およびスペースの使い方に注意してください。xpath、XML タブ、および文字列値が単一引用符で囲まれていることに注意してください。

次の例では、上記の引用にある 3 番目の項目の ID を取り出す方法を示します。メソッド呼び出しは、66 という文字列を返します。

```
String sel = "JMS_BEA_SELECT('xpath',
'/order/item[3]/id/text()') = '66'";
```

メッセージ セレクタの表示

次の MessageConsumer メソッドを使用して、メッセージ セレクタを表示できます。

```
public String getMessageSelector(
) throws JMSException
```

このメソッドは、現在定義されているメッセージ セレクタ、またはメッセージ セレクタが定義されていない場合は null のいずれかを返します。

トピック サブスクライバのメッセージ セレクタにインデックスを付けることによるパフォーマンスの最適化

アプリケーションの一部のクラスでは、WebLogic JMS でトピック サブスクライバのメッセージ セレクタにインデックスを付けることで最適化できます。一般的にこれらのアプリケーションは、各々がユニークな識別子 (ユーザ名など) を付けられた多数のサブスクライバを備えており、単一のサブスクライバ、またはサブスクライバのリストに対し、迅速にメッセージを送信できる必要があります。一般的な例としては、各サブスクライバが異なるユーザに対応し、各メッセージに 1 つまたは複数の対象ユーザのリストが含まれるインスタント メッセージング アプリケーションがあります。

最適化されたサブスクライバのメッセージ セレクタをアクティブ化するには、サブスクライバはセレクタに対し次の構文を使用する必要があります。

```
"identifier IS NOT NULL"
```

ここで、*identifier* はあらかじめ定義された JMS メッセージ プロパティではない (例 : JMSCorrelationID でも JMSType でもない) 任意の文字列を表します。複数のサブスクライバで同一の識別子を共有できます。

WebLogic JMS はこのメッセージ セレクタ構文を、内部サブスクライバ インデックスを構築する手掛かりとして使用します。この構文に従わないメッセージ セレクタ、またはさらに OR 句および AND 句を含むメッセージ セレクタにも対応はしていますが、これらは最適化にはつながりません。

サブスクライバがこのメッセージ セレクタ構文を使って登録を行った後は、トピックに公開されたメッセージが、次の例のようにメッセージのユーザ プロパティに 1 つまたは複数の識別子を含めることによって、特定のサブスクライバを対象とできるようになります。

```
// 指定されたサブスクライバを設定する。ここでは "wilma" が
// サブスクライバ名、subscriberSession は JMS TopicSession。
// 使用されているセレクタ構文により、最適化が促進される。

TopicSubscriber topicSubscriber =
    subscriberSession.createSubscriber(
       (Topic)context.lookup("IMTopic"),
       "Wilma IS NOT NULL",
       /* noLocal= */ true);

// サブスクライバ "Fred" および "Wilma" へメッセージを送信する。
// ここでは publisherSession は JMS TopicSession。メッセージ
// セレクタ式が "Wilma IS NOT NULL" または "Fred IS NOT NULL" である
// サブスクライバが、このメッセージを受信する。

TopicPublisher topicPublisher =
  publisherSession.createPublisher(
    (Topic)context.lookup("IMTopic");

TextMessage msg =
  publisherSession.createTextMessage("Hi there!");
msg.setBooleanProperty("Fred", true);
msg.setBooleanProperty("Wilma", true);

topicPublisher.publish(msg);
```

注意 : 最適化されたメッセージ セレクタおよびメッセージ構文は、標準 JMS API に基づいています。したがって、この構文を使用するアプリケーションは、最適化されたメッセージ セレクタのないバージョンの

WebLogic JMS および非 WebLogic JMS 製品でも機能します。ただし、これらのバージョンでは、この機能強化が含まれているバージョンほどのパフォーマンスは得られません。

メッセージ セレクタの最適化は、MULTICAST_NO_ACKNOWLEDGE 確認応答モードを使用するアプリケーションには効果がありません。メッセージ選択はサーバ サイドではなくクライアント サイドで行われるため、これらのアプリケーションはいずれにしても機能強化を必要としません。

サーバ セッション プールの定義

注意 : セッション プールは、現在はあまり使用されません。その理由としては、J2EE 仕様で必要なくなったこと、JTA ユーザ トランザクションをサポートしていないこと、よりシンプルで管理がしやすく機能性の高いメッセージ駆動型 Bean (MDB) で代用されるようになったことなどが挙げられます。MDB の設計の詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』の「メッセージ駆動型 Bean の設計」を参照してください。

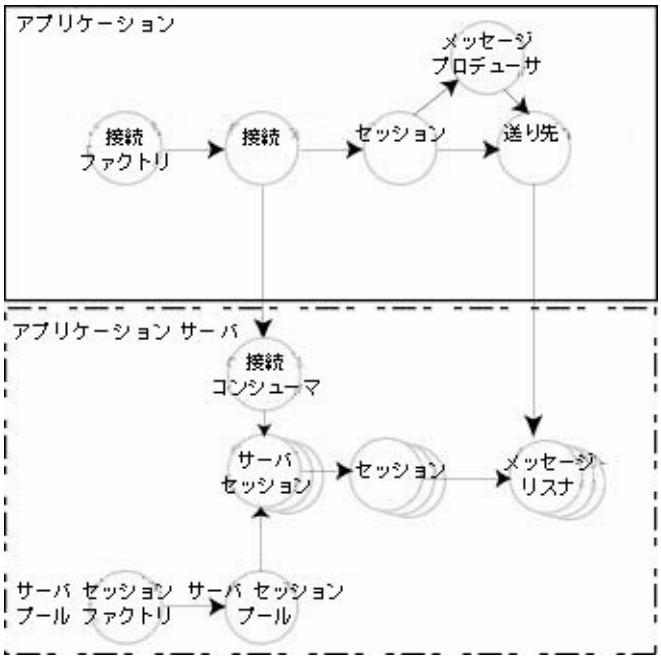
WebLogic JMS には、サーバ セッションのサーバ管理プールを定義するためのオプションの JMS 機能が実装されています。この機能を使用すると、アプリケーションで複数のメッセージを並行して処理できます。

サーバ セッション プールの機能は次のとおりです。

- 送り先からメッセージを受信し、そのメッセージを、メッセージ処理用に用意したサーバ側のメッセージ リスナに渡す。メッセージ リスナ クラスには、メッセージを処理する onMessage() メソッドがあります。
- JMS セッションのプールを管理することで、メッセージを並行して処理する。各セッションでは、シングル スレッドの onMessage() メソッドが実行されます。

次の図に、サーバ セッション プール機能、およびアプリケーションとアプリケーション サーバのコンポーネントの関係を示します。

図 4-3 サーバ セッション プール機能

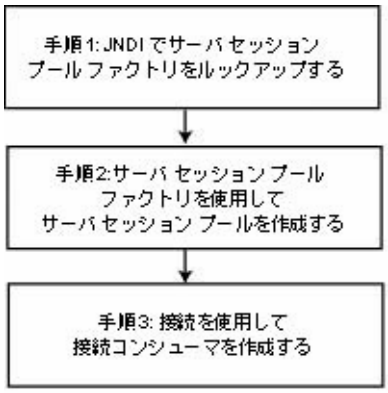


図に示されているように、アプリケーションにはシングル スレッドのメッセージ リスナが用意されています。JMS によって実装された、アプリケーション サーバ上の接続コンシューマによって、以下のタスクが実行され、1 つまたは複数のメッセージが処理されます。

1. サーバ セッション プールからサーバ セッションを取得する。
2. サーバ セッションのセッションを取得する。
3. セッションに 1 つまたは複数のメッセージをロードする。
4. サーバ セッションを開始して、メッセージを受信する。
5. メッセージの処理が終了したら、サーバ セッションを解放してプールに戻す。

次の図に、メッセージの並行処理を行うための準備に必要な手順を示します。

図 4-4 メッセージの並行処理を行うための準備



この手順では、アプリケーションで、他のアプリケーション サーバ プロバイダのセッション プール実装を使用できます。サーバ セッション プールはメッセージ駆動型 Bean を使用して実装することもできます。メッセージ駆動型 Bean によるサーバ セッション プールの実装については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

コンフィグレーション時にセッション プールと接続コンシューマが定義された場合は、この手順を省略することができます。サーバ セッション プールと接続コンシューマのコンフィグレーションの詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

WebLogic JMS は現在、オプションの `TopicConnection.createDurableConnectionConsumer()` 操作をサポートしていません。この高機能 JMS の操作の詳細については、Sun Microsystems の JMS 仕様を参照してください。

手順 1 : JNDI でサーバ セッション プール ファクトリをルックアップする

サーバ セッション プール ファクトリを使用して、サーバ セッション プールを作成します。

WebLogic JMS は、デフォルトでは `weblogic.jms.ServerSessionPoolFactory:<name>` (<*name*> は、セッション プールが作成される JMS サーバ名) という `ServerSessionPoolFactory` オブジェクトを 1 つ定義します。

コンフィグレーションが終了したら、サーバ セッション プール ファクトリをルックアップするために、まず `NamingManager.InitialContext()` メソッドを使用して JNDI コンテキスト (`context`) を定義します。サーブレット アプリケーション以外のアプリケーションの場合は、初期コンテキストの作成に使用する環境を渡す必要があります。詳細については、`NamingManager.InitialContext()` Javadoc を参照してください。

コンテキストが定義されたら、次のコードを使用して、JNDI でサーバ セッション プール ファクトリをルックアップします。

```
factory = (ServerSessionPoolFactory) context.lookup(<ssp_name>);
```

`<ssp_name>` には、サーバ セッション プール ファクトリの修飾名または非修飾名を指定します。

サーバ セッション プール ファクトリの詳細については、2-24 ページの「ServerSessionPoolFactory」、または `weblogic.jms.ServerSessionPoolFactory` Javadoc を参照してください。

手順 2 : サーバ セッション プール ファクトリを使用してサーバ セッション プールを作成する

以降の節で説明する `ServerSessionPoolFactory` メソッドを使用して、キュー (PTP) またはトピック (Pub/Sub) の接続コンシューマで使用するサーバ セッション プールを作成できます。

サーバ セッション プールの詳細については、2-24 ページの「ServerSessionPool」、または `javax.jms.ServerSessionPool` Javadoc を参照してください。

キュー接続コンシューマで使用するサーバ セッション プールを作成する

ServerSessionPoolFactory には、キュー接続コンシューマ用のサーバ セッション プールを作成する、次のメソッドが用意されています。

```
public ServerSessionPool getServerSessionPool(
  QueueConnection connection,
  int maxSessions,
  boolean transacted,
  int ackMode,
  String listenerClassName
) throws JMSException
```

サーバ セッション プールに関連付けられるキュー接続、接続コンシューマ (手順 3 で作成予定) で取得できる並行セッションの最大数、セッションをトランザクション処理するかどうか、確認応答モード (トランザクション処理されないセッションの場合にのみ適用可能)、およびインスタンス化され、メッセージの受信および並行処理に使用されるメッセージ リスナ クラスを指定する必要があります。

ServerSessionPoolFactory クラス メソッドの詳細については、weblogic.jms.ServerSessionPoolFactory Javadoc を参照してください。ConnectionConsumer クラスの詳細については、javax.jms.ConnectionConsumer Javadoc を参照してください。

トピック接続コンシューマで使用するサーバ セッション プールを作成する

ServerSessionPoolFactory には、トピック接続コンシューマ用のサーバ セッション プールを作成する、次のメソッドが用意されています。

```
public ServerSessionPool getServerSessionPool(
  TopicConnection connection,
  int maxSessions,
  boolean transacted,
  int ackMode,
  String listenerClassName
) throws JMSException
```

サーバ セッション プールに関連付けられるトピック接続、接続コンシューマ (手順 3 で作成予定) で取得できる並行セッションの最大数、セッションをトランザクション処理するかどうか、確認応答モード (トランザクション処理されないセッションの場合にのみ適用可能)、およびインスタンス化され、メッセージの受信および並行処理に使用されるメッセージ リスナ クラスを指定する必要があります。

`ServerSessionPoolFactory` クラス メソッドの詳細については、`weblogic.jms.ServerSessionPoolFactory` Javadoc を参照してください。`ConnectionConsumer` クラスの詳細については、`javax.jms.ConnectionConsumer` Javadoc を参照してください。

手順 3 : 接続コンシューマを作成する

以下の方法のいずれかを使用して、サーバ セッションを取得し、メッセージを並行処理するための接続コンシューマを作成できます。

- 『管理者ガイド』の「JMS の管理」にある説明に従って、コンフィグレーション時にサーバ セッション プールと接続コンシューマをコンフィグレーションします。
- 以降の節で説明されている Connection メソッドをアプリケーションに含めます。

`ConnectionConsumer` クラスの詳細については、2-25 ページの「ConnectionConsumer」、または `javax.jms.ConnectionConsumer` Javadoc を参照してください。

キュー用の接続コンシューマを作成する

`QueueConnection` には、キュー用の接続コンシューマを作成する、次のメソッドが用意されています。

```
public ConnectionConsumer createConnectionConsumer(
  Queue queue,
  String messageSelector,
  ServerSessionPool sessionPool,
  int maxMessages
) throws JMSException
```

関連付けられるキューの名前、メッセージをフィルタ処理するためのメッセージ セレクタ、サーバ セッションにアクセスするためのサーバ セッション プール、およびサーバ セッションに同時に割り当てることができるメッセージの最大数を指定する必要があります。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

`QueueConnection` クラス メソッドの詳細については、`javax.jms.QueueConnection` Javadoc を参照してください。
`ConnectionConsumer` クラスの詳細については、`javax.jms.ConnectionConsumer` Javadoc を参照してください。

トピック用の接続コンシューマを作成する

`TopicConnection` には、トピック用の接続コンシューマを作成する、以下の 2 種類のメソッドが用意されています。

```
public ConnectionConsumer createConnectionConsumer(
  Topic topic,
  String messageSelector,
  ServerSessionPool sessionPool,
  int maxMessages
) throws JMSException
```

```
public ConnectionConsumer createDurableConnectionConsumer(
  Topic topic,
  String messageSelector,
  ServerSessionPool sessionPool,
  int maxMessages
) throws JMSException
```

各メソッドには、関連付けられるトピックの名前、メッセージをフィルタ処理するためのメッセージ セレクタ、サーバ セッションにアクセスするためのサーバ セッション プール、およびサーバ セッションに同時に割り当てることができるメッセージの最大数を指定する必要があります。メッセージ セレクタの詳細については、4-71 ページの「メッセージのフィルタ処理」を参照してください。

いずれのメソッドも接続コンシューマを作成しますが、後者のメソッドは、恒久サブスクライバで使用する恒久接続コンシューマも作成します。恒久サブスクライバの詳細については、4-57 ページの「恒久サブスクリプションの設定」を参照してください。

TopicConnection クラス メソッドの詳細については、
javax.jms.TopicConnection Javadoc を参照してください。
ConnectionConsumer クラスの詳細については、
javax.jms.ConnectionConsumer Javadoc を参照してください。

例：PTP クライアントのサーバ セッション プールの設定

次の例では、JMS クライアント用のサーバ セッション プールを設定する方法を示します。startup() メソッドは、4-17 ページの「例 :PTP アプリケーションの設定」で説明されている examples.jms.queue.QueueSend の例の init() メソッドとほぼ同じです。このメソッドでもサーバ セッション プールを設定できます。

次の例に startup() メソッドを示し、併せて各設定手順も述べます。

サーバ セッション プール アプリケーションを実装するには、次のパッケージをインポート リストに追加します。

```
import weblogic.jms.ServerSessionPoolFactory
```

セッション プールの作成に必要なセッション プール ファクトリの静的変数を定義します。

```
private final static String SESSION_POOL_FACTORY=
       "weblogic.jms.ServerSessionPoolFactory:examplesJMSServer";

private QueueConnectionFactory qconFactory;
private QueueConnection qcon;
private QueueSession qsession;
private QueueSender qsender;
private Queue queue;
private ServerSessionPoolFactory sessionPoolFactory;
private ServerSessionPool sessionPool;
private ConnectionConsumer consumer;
```

必要な JMS オブジェクトを作成します。

```
public String startup(
  String name,
  Hashtable args
) throws Exception
```

```
{
  String connectionFactory = (String)args.get("connectionFactory");
  String queueName = (String)args.get("queue");
  if (connectionFactory == null || queueName == null) {
       throw new
IllegalArgumentException("connectionFactory="+connectionFactory+
                                          ", queueName="+queueName);
    }
  Context ctx = new InitialContext();
  qconFactory = (QueueConnectionFactory)
       ctx.lookup(connectionFactory);
  qcon =qconFactory.createQueueConnection();
  qsession = qcon.createQueueSession(false,
       Session.AUTO_ACKNOWLEDGE);
  queue = (Queue) ctx.lookup(queueName);
  qcon.start();
```

手順 1　JNDI でサーバ セッション プール ファクトリをルックアップします。

```
sessionPoolFactory = (ServerSessionPoolFactory)
                ctx.lookup(SESSION_POOL_FACTORY);
```

手順 2　次のように、サーバ セッション プール ファクトリを使用してサーバ セッション プールを作成します。

```
sessionPool = sessionPoolFactory.getServerSessionPool(qcon, 5,
     false, Session.AUTO_ACKNOWLEDGE,
     examples.jms.startup.MsgListener);
```

このコードでは、以下のように定義されています。

- qcon は、サーバ セッション プールに関連付けられるキュー接続を示します。
- 5 は、接続コンシューマ (手順 3 で作成予定) で取得できる並行セッションの最大数を示します。
- false は、セッションをトランザクション処理しないことを示します。
- AUTO_ACKNOWLEDGE は、確認応答モードを示します。
- examples.jms.startup.MsgListener は、インスタンス化され、メッセージの受信および並行処理に使用されるメッセージ リスナを示します。

手順 3　次のように、接続コンシューマを作成します。

```
consumer = qcon.createConnectionConsumer(queue, "TRUE",
     sessionPool, 10);
```

このコードでは、以下のように定義されています。

- queue は、関連付けられるキューを示します。
- TRUE は、メッセージをフィルタ処理するためのメッセージ セレクタを示します。
- sessionPool は、サーバ セッションにアクセスするためのサーバ セッション プールを示します。
- 10 は、サーバ セッションに同時に割り当てることができるメッセージの最大数を示します。

この例で使用した JMS クラスの詳細については、2-5 ページの「WebLogic JMS のクラス」、または javax.jms Javadoc を参照してください。

例 : Pub/Sub クライアントのサーバ セッション プールの設定

次の例では、JMS クライアント用のサーバ セッション プールを設定する方法を示します。startup() メソッドは、4-21 ページの「例 :Pub/Sub アプリケーションの設定」で説明されている examples.jms.topic.TopicSend の例の init() メソッドとほぼ同じです。このメソッドでもサーバ セッション プールを設定できます。

次の例に startup() メソッドを示し、併せて各設定手順も述べます。

サーバ セッション プール アプリケーションを実装するには、次のパッケージをインポート リストに追加します。

```
import weblogic.jms.ServerSessionPoolFactory
```

セッション プールの作成に必要なセッション プール ファクトリの静的変数を定義します。

```
private final static String SESSION_POOL_FACTORY=
       "weblogic.jms.ServerSessionPoolFactory:examplesJMSServer";

private TopicConnectionFactory tconFactory;
private TopicConnection tcon;
private TopicSession tsession;
private TopicSender tsender;
private Topic topic;
private ServerSessionPoolFactory sessionPoolFactory;
```

```
private ServerSessionPool sessionPool;
private ConnectionConsumer consumer;
```

必要な JMS オブジェクトを作成します。

```
public String startup(
  String name,
  Hashtable args
) throws Exception

{
  String connectionFactory = (String)args.get("connectionFactory");
  String topicName = (String)args.get("topic");
  if (connectionFactory == null || topicName == null) {
       throw new
IllegalArgumentException("connectionFactory="+connectionFactory+
                                        ", topicName="+topicName);
    }
  Context ctx = new InitialContext();
  tconFactory = (TopicConnectionFactory)
       ctx.lookup(connectionFactory);
  tcon = tconFactory.createTopicConnection();
  tsession = tcon.createTopicSession(false,
       Session.AUTO_ACKNOWLEDGE);
  topic = (Topic) ctx.lookup(topicName);
  tcon.start();
```

手順 1　JNDI でサーバ セッション プール ファクトリをルックアップします。

```
  sessionPoolFactory = (ServerSessionPoolFactory)
                  ctx.lookup(SESSION_POOL_FACTORY);
```

手順 2　次のように、サーバ セッション プール ファクトリを使用してサーバ セッション プールを作成します。

```
  sessionPool = sessionPoolFactory.getServerSessionPool(tcon, 5,
       false, Session.AUTO_ACKNOWLEDGE,
       examples.jms.startup.MsgListener);
```

このコードでは、以下のように定義されています。

- `tcon` は、サーバ セッション プールに関連付けられるトピック接続を示します。
- `5` は、接続コンシューマ (手順 3 で作成予定) で取得できる並行セッションの最大数を示します。
- `false` は、セッションをトランザクション処理しないことを示します。
- `AUTO_ACKNOWLEDGE` は、確認応答モードを示します。

- `examples.jms.startup.MsgListener` は、インスタンス化され、メッセージの受信および並行処理に使用されるメッセージ リスナを示します。

手順 3　次のように、接続コンシューマを作成します。

```
consumer = tcon.createConnectionConsumer(topic, "TRUE",
     sessionPool, 10);
```

このコードでは、以下のように定義されています。

- `topic` は、関連付けられるトピックを示します。
- `TRUE` は、メッセージをフィルタ処理するためのメッセージ セレクタを示します。
- `sessionPool` は、サーバ セッションにアクセスするためのサーバ セッション プールを示します。
- `10` は、サーバ セッションに同時に割り当てることができるメッセージの最大数を示します。

この例で使用した JMS クラスの詳細については、2-5 ページの「WebLogic JMS のクラス」、または `javax.jms` Javadoc を参照してください。

マルチキャストの使い方

マルチキャストを使用することによって、後でメッセージをサブスクライバに転送する、指定したホストのグループにメッセージを配信できます。

マルチキャストには、次のような利点があります。

- ホスト グループにメッセージをほとんどリアルタイムに配信できる。
- メッセージをサブスクライバに配信する場合に JMS サーバで必要になるリソース量が削減されるので、スケーラビリティが向上する。

マルチキャストには、次のような制限があります。

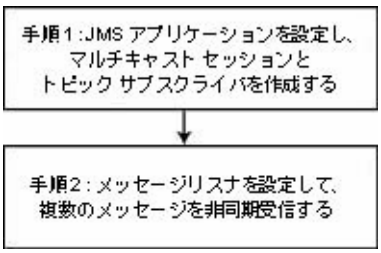
- マルチキャストでは、ホスト グループの全メンバーに対するメッセージの配信は保証されない。確実な配信と回復が必要なメッセージについては、マルチキャストを使用しないでください。

- 異なるバージョンの WebLogic Server との相互運用性のため、クライアントにはホストよりもバージョンの古いリリースの WebLogic Server はインストールできない。少なくとも同じバージョンか、それ以降のものにする必要があります。

マルチキャストを使用すると便利な例としては、株価表示があります。最新の株価を入手する場合に重要になるのは、信頼性よりもタイムリーな配信です。全部または一部の内容が配信されなくても、リアルタイムの株価情報にアクセスするときに、クライアントは簡単に情報の再送信を要求できます。クライアントでは情報の回復は必要とされません。回復された情報が再配信される頃には、その情報は古くて価値のないものになっています。

次の図に、マルチキャストの設定に必要な手順を示します。

図 4-5 マルチキャストの設定



注意 : マルチキャストは、Pub/Sub メッセージング モデル、および非恒久サブスクライバのみでサポートされています。

マルチキャスト セッションおよびマルチキャスト コンシューマに関するモニタ統計は提供されません。

マルチキャストを設定する前に、マルチキャストをサポートするよう、次のように接続ファクトリおよび送り先をコンフィグレーションする必要があります。

- 各接続ファクトリについては、システム管理者が、マルチキャスト セッションに存在できる未処理のメッセージの最大数と、最大数に達した場合に最新のメッセージと最古のメッセージのどちらを破棄するかをコンフィグレーションする。メッセージが最大数に達すると、`DataOverrunException` が送出され、メッセージは自動的に破棄されます。これらの属性は、動的にコンフィグレーションすることもできます (4-91 ページの「マルチキャストのコンフィグレーション属性の動的コンフィグレーション」参照)。

- 各送り先については、マルチキャスト アドレス (IP)、ポート、TTL (存続時間) の各属性を指定する。TTL 属性の設定の詳細については、4-92 ページの「例 : マルチキャスト TTL (存続時間)」を参照してください。

注意 : マルチキャスト IP アドレス、ポート、存続時間の各属性をコンフィグレーションする場合は、ネットワーク管理者に相談してから、適切な値を設定することをお勧めします。

マルチキャストのコンフィグレーション属性の詳細については、Administration Console オンライン ヘルプを参照してください。マルチキャストのコンフィグレーション属性は、付録 A「コンフィグレーション チェックリスト」でも簡単に説明されています。

手順 1 : JMS アプリケーションを設定し、マルチキャスト セッションとトピック サブスクライバを作成する

JMS アプリケーションの設定については、4-4 ページの「JMS アプリケーションの設定」で説明されています。ただし、セッションを作成する際は、4-8 ページの「手順 3 : 接続を使用してセッションを作成する」に説明されているように、*acknowledgeMode* の値を `MULTICAST_NO_ACKNOWLEDGE` に設定することで、セッションがマルチキャスト メッセージを受信するように指定します。

注意 : マルチキャストは、Pub/Sub メッセージング モデル、および非恒久サブスクライバのみでサポートされています。マルチキャスト セッションで恒久サブスクライバを作成しようとすると、`JMSException` が送出されます。

たとえば、次のメソッドは、Pub/Sub メッセージング モデル用のマルチキャスト セッションの作成方法を示します。

```
tsession = tcon.createTopicSession(
     false,
  WLSession.MULTICAST_NO_ACKNOWLEDGE
);
```

注意： クライアント サイドでは、マルチキャスト セッションごとに、ソケットからメッセージを取り出すための専用のスレッドが 1 つ必要になります。そのため、JMS クライアントサイドのスレッド プール サイズを増やして調整する必要があります。スレッド プール サイズの調整に関する詳細については、「WebLogic JMS Performance Guide」ホワイト ペーパーの「Tuning Thread Pools and EJB Pools」の項を参照してください。JMS クライアントサイドのスレッド プールのチューニングについて書かれています。

さらに、4-13 ページの「TopicPublisher と TopicSubscriber の作成」にある説明に従って、トピック サブスクライバを作成します。

たとえば、次のコードは、トピック サブスクライバの作成方法を示します。

```
tsubscriber = tsession.createSubscriber(myTopic);
```

注意： 指定した送り先がマルチキャストをサポートするようコンフィグレーションされていない場合、`createSubscriber()` メソッドは失敗します。

手順 2：メッセージ リスナを設定する

マルチキャストのトピック サブスクライバでは、メッセージを非同期に受信することしかできません。マルチキャスト セッションで同期メッセージを受信しようとすると、`JMSException` が送出されます。

4-31 ページの「メッセージの非同期受信」にある説明に従って、トピック サブスクライバに対してメッセージ リスナを設定します。

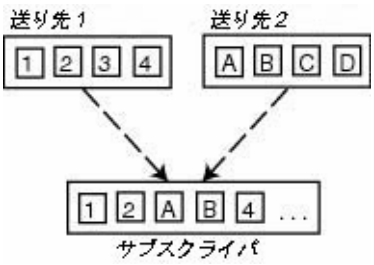
たとえば、次のコードは、メッセージ リスナの設定方法を示します。

```
tsubscriber.setMessageListener(this);
```

メッセージを受信すると、WebLogic JMS では、送り先から送信されたメッセージのシーケンスがトラッキングされます。マルチキャスト サブスクライバのメッセージ リスナがシーケンスが異なるメッセージを受信すると、その結果、1 つまたは複数のメッセージがスキップされ、そのセッションの

ExceptionListener に SequenceGapException が送出されます。スキップされたメッセージは、その後配信されても破棄されます。たとえば、次の図では、サブスクライバは 2 つの送り先から同時にメッセージを受信しています。

図 4-6　マルチキャストにおけるシーケンスのずれ



送り先 1 からメッセージ「4」を受信すると、SequenceGapException が送出され、シーケンスが異なるメッセージが受信されたことがアプリケーションに通知されます。その後、メッセージ「3」が配信されても、それは破棄されます。

注意： やり取りされるメッセージ数が多くなるほど、SequenceGapException が発生する危険性も大きくなります。

マルチキャストのコンフィグレーション属性の動的コンフィグレーション

システム管理者は、コンフィグレーション時に、マルチキャストをサポートするよう、各接続ファクトリに対して以下の情報をコンフィグレーションします。

- マルチキャスト セッションに存在できる未処理のメッセージの最大数を指定する最大メッセージ数
- メッセージが最大数に達した場合に、最新のメッセージと最古のメッセージのどちらを破棄するかを指定する超過時のポリシー

メッセージが最大数に達すると、DataOverrunException が送出され、メッセージは超過時のポリシーに基づいて自動的に破棄されます。また、Session クラスの set メソッドを使用して、最大メッセージ数と超過時のポリシーを設定する方法もあります。

次の表に、Session クラスの set メソッドと get メソッドを、動的コンフィグレーションが可能な属性ごとに示します。

表 4-6 メッセージ プロデューサの set メソッドおよび get メソッド

属性	set メソッド	get メソッド
最大メッセージ数	`public void setMessagesMaximum( int messagesMaximum ) throws JMSException`	`public int getMessagesMaximum( ) throws JMSException`
超過時のポリシー	`public void setOverrunPolicy ( int overrunPolicy ) throws JMSException`	`public int getOverrunPolicy( ) throws JMSException`

注意： set メソッドを使用して設定された値は、コンフィグレーションされた値よりも優先されます。

Session クラス メソッドの詳細については、weblogic.jms.extensions.WLSession Javadoc を参照してください。マルチキャストのコンフィグレーション属性の詳細については、Administration Console オンライン ヘルプの「JMS の送り先」を参照してください。

例：マルチキャスト TTL (存続時間)

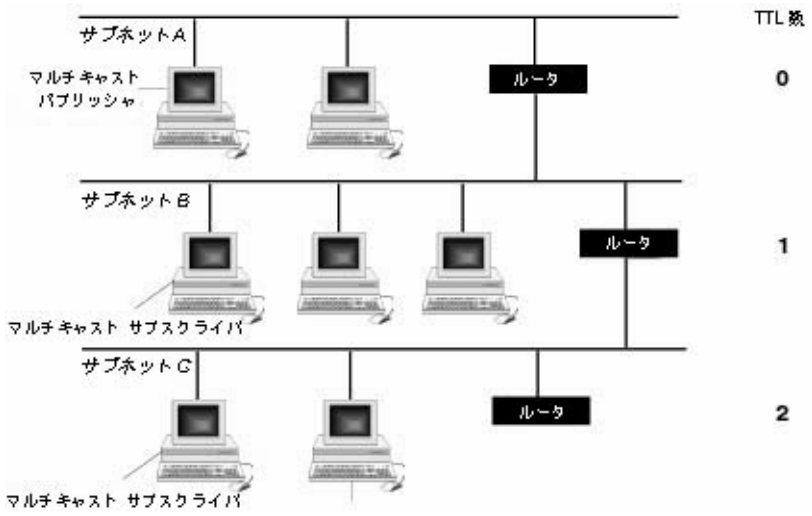
注意： 次の図は、マルチキャスト TTL (存続時間) 送り先コンフィグレーション属性が、ルータを経由したメッセージの配信に影響する様子を説明するための単純な例です。マルチキャスト TTL 属性をコンフィグレーションする場合は、ネットワーク管理者に相談してから、適切な値を設定することをお勧めします。

マルチキャスト TTL は、メッセージの存続時間に依存しません。

次の例では、マルチキャスト TTL 送り先コンフィグレーション属性が、ルータを経由したメッセージの配信に影響する様子を示します。マルチキャストのコンフィグレーション属性の詳細については、Administration Console オンライン ヘルプの「JMS の送り先」を参照してください。

次のようなネットワーク図があります。

図 4-7　マルチキャスト TTL の例



この図では、ネットワークは 3 つのサブネットで構成されています。サブネット A にはマルチキャスト パブリッシャがあり、サブネット B および C にはそれぞれ 1 台ずつマルチキャスト サブスクライバがあります。

マルチキャスト TTL 属性が 0 に設定されている (メッセージはルータを経由できず、現在のサブネットにしか配信されないことを示す) 場合は、サブネット A にあるマルチキャスト パブリッシャでメッセージがパブリッシュされても、メッセージはどのマルチキャスト サブスクライバにも配信されません。

マルチキャスト TTL 属性が 1 に設定されている (メッセージは 1 台のルータを経由できることを示す) 場合、サブネット A にあるマルチキャスト パブリッシャでメッセージがパブリッシュされると、サブネット B にあるマルチキャスト サブスクライバでメッセージが受信されます。

同様に、マルチキャスト TTL 属性が 2 に設定されている (メッセージは 2 台のルータを経由できることを示す) 場合、サブネット A にあるマルチキャスト パブリッシャでメッセージがパブリッシュされると、サブネット B および C にあるマルチキャスト サブスクライバでメッセージが受信されます。

分散送り先の使用

複数の物理的送り先 (キューとトピック) を単一の分散送り先セットのメンバーとしてコンフィグレーションできるようにすることで、WebLogic JMS は、クラスタ内の 1 台のサーバに障害が発生した場合でもサービスを継続します。適切に構成すると、プロデューサとコンシューマがその分散送り先に対してメッセージを送受信できるようになります。WebLogic JMS は、分散送り先内で利用可能な全送り先メンバーにメッセージの負荷を分散します。サーバの障害で送り先メンバーがアクセスできなくなった場合、トラフィックはセット内の他のアクセス可能な送り先メンバーにリダイレクトされます。

Administration Console を使用して分散送り先をコンフィグレーションする手順については、『管理者ガイド』の「分散送り先のコンフィグレーション」を参照してください。

以下の節では、JMS アプリケーションで分散送り先を使用する方法について説明します。

- 分散送り先へのアクセス
- 分散送り先メンバーへのアクセス
- 分散送り先におけるメッセージのロード バランシング
- 分散送り先の移行

分散送り先へのアクセス

分散送り先は、実際には一連の物理的な JMS 送り先メンバー (キューまたはトピック) となっており、これには単一の JNDI 名を使用してアクセスします。このため、分散送り先は JNDI を使用してルックアップできます。実装されるインタフェースは `javax.jms.Destination` で、これはプロデューサ、コンシューマ、およびブラウザの作成に使用できます。

宛先はクラスタ内の複数の WebLogic Server からサービスを受けることができるため、`createQueue()` メソッドまたは `createTopic()` メソッドのどちらかを使って送り先の参照を作成するときに提供される名前は、単純に

JMSDistributedQueueMBean または JMSDistributedTopicMBean コンフィグレーション MBean 名となります。JMS サーバ名や区切り用のフォワード スラッシュ (/) は必要ありません。

たとえば、次のコードは、分散送り先トピック メンバーのルックアップ方法を示します。

```
topic = myTopicSession.createTopic("myDistributedTopic");
```

注意： createQueue() メソッドまたは createTopic() メソッドを呼び出す際、フォワード スラッシュ (/) を含む文字列は、送り先ではなく、分散送り先メンバーの名前であると見なされます。このような分散送り先メンバーが存在しない場合、呼び出しは InvalidDestinationException で失敗します。

分散キューのルックアップ

分散キューは、一連の物理的な JMS キューのメンバーです。したがって、分散キューは QueueSender、QueueReceiver、および QueueBrowser を作成するために使用できます。分散キューが複数の物理的キューを表すという事実は、アプリケーションには全く意識されません。

キューのメンバーはどこにでも配置できますが、単一のサーバ クラスタ内にある JMS サーバからサービスを受ける必要があります。分散キューに送信されるメッセージは、分散キューの一連のメンバー内にある物理的キューの 1 つだけに送られます。キューのメンバーに届いたメッセージは、そのキューのメンバーのコンシューマのみが受信できるようになります。

注意： キューのメンバーがメッセージをキューの他のメンバーに転送できるようにするには、Administration Console の [転送の遅延] 属性をコンフィグレーションします。この属性はデフォルトでは無効になっています。この属性は、コンシューマではなく、メッセージだけを持つ分散キューのメンバーが、コンシューマを持つキューの他のメンバーにメッセージを転送するときに待機する時間を秒単位で定義します。

QueueSender

キュー センダを作成した後、作成時に提供したキューが分散キューであった場合は、センダを使用してメッセージが作成されるたびに、どのキューのメンバーがそのメッセージを受信するかの判断が下されます。各メッセージは単一の物理的キューのメンバーに送信されます。

メッセージが複製されることは一切ありません。このため、メッセージは送信元のキューのメンバーのみから受信できます。メッセージの受信前にその物理的キューが使用できなくなった場合は、そのキューのメンバーがオンラインに復帰するまでこのメッセージを受信できなくなります。

メッセージを分散キューに送信して、その分散キューのキュー レシーバがメッセージを受信することを期待するだけでは十分とは言えません。メッセージは 1 つの物理的キューのメンバーだけに送信されるため、そのキューのメンバーで受信またはリスンするキュー レシーバが存在する必要があります。

注意 : コンシューマを持たない分散キューに対するロード バランシングのヒューリスティックについては、4-104 ページの「ロード バランシングのヒューリスティック」を参照してください。

QueueReceiver

キュー レシーバを作成する際は、提供したキューが分散キューの場合、単一の物理的キューのメンバーが作成時にレシーバとして選ばれます。作成された `QueueReceiver` は、キュー レシーバがキュー メンバーのアクセスを失うまでそのキュー メンバーに固定されます。この時点で、コンシューマは次のように `JMSException` を受信します。

- キュー レシーバが同期レシーバである場合は、例外がユーザに直接返される。
- キュー レシーバが非同期レシーバである場合は、例外が `ConsumerClosedException` の内部に配信され、それがコンシューマ セッションに定義されている `ExceptionListener` (存在する場合) に配信される。

このような例外を受信した場合、アプリケーションは自分のキュー レシーバを終了し、再作成できます。分散キュー内で他のキュー メンバーを使用できる場合は、新しいキューレシーバが作成され、これらのキュー メンバーのいずれかに固定されます。他のキュー メンバーを使用できない場合、キュー レシーバの再作成はできないため、後で作成を試みる必要があります。

注意： コンシューマを持たない分散キューに対するロード バランシングのヒューリスティックについては、4-104 ページの「ロード バランシングのヒューリスティック」を参照してください。

QueueBrowser

キュー ブラウザを作成する際は、提供されているキューが分散キューの場合、単一の物理的キューのメンバーが作成時にブラウザとして選ばれます。作成されたキュー ブラウザは、レシーバがキュー メンバーのアクセスを失うまでそのキューメンバーに固定されます。その時点でキュー ブラウザを呼び出した場合は、`JMSException` が発行されます。列挙を呼び出した場合は、`NoSuchElementException` が返されます。

注意： キュー ブラウザは固定されているキュー メンバーのみを閲覧できます。分散キューが作成時に指定されている場合でも、キュー ブラウザが分散送り先内の他のキュー メンバーのメッセージを表示または閲覧することはできません。これは、サーバ アフィニティが分散送り先で有効にされている場合でも同じです。

分散トピックのルックアップ

分散トピックは、一連の物理的な JMS トピックのメンバーです。このため、分散トピックは `TopicPublisher` と `TopicSubscriber` を使用して作成できます。分散トピックが複数の物理的トピックを表すという事実は、アプリケーションには全く意識されません。

注意： 恒久サブスクライバ (`DurableTopicSubscriber`) は、分散トピックに対しては作成できません。ただし、分散トピックのメンバーに対して恒久サブスクリプションを作成することはできます。こうすると、他のトピック メンバーは、恒久サブスクリプションを持つトピック メンバーにメッセージを転送します。

トピック メンバーはどこにでも配置できますが、単一の WebLogic Server、またはクラスタ内の任意の数のサーバによってサービスを受ける必要があります。分散トピックに送信されたメッセージは、分散トピック セット内のトピック メンバーすべてに送信されます。このため、分散トピックのすべてのサブスクライバは、分散トピック用にパブリッシュされたメッセージを受信できます。

送り先のトピック メンバーに直接パブリッシュされたメッセージ (つまりパブリッシャが送り先を指定しなかったメッセージ) も、その分散トピックのメンバーすべてに転送されます。これには、最初に分散トピックにサブスクライブしたサブスクライバと、その特定のトピック メンバーにたまたま割り当てられているサブスクライバが含まれます。つまり、特定の分散トピックのメンバーにメッセージをパブリッシュすると、それが分散トピックの他のメンバーすべてに自動的に転送されます。これは、分散トピックにメッセージをパブリッシュすると、その分散トピックのメンバーすべてにメッセージが自動的に転送されることと同じです。特定の分散送り先のメンバーのルックアップの詳細については、4-101 ページの「分散送り先メンバーへのアクセス」を参照してください。

分散トピックへのメッセージ駆動型 Bean のデプロイ

MDB を分散トピックにデプロイし、JMS サーバ上の分散トピックの 2 つのメンバーをホストしているクラスタ内の WebLogic Server インスタンスを対象とした場合、MDB は分散トピックのいずれのメンバーにもデプロイされます。これは、MDB が分散トピックのメンバーの送り先名に固定されているからです。

したがって、WebLogic Server インスタンスにデプロイされている分散トピックのメンバー数に応じて、MDB ごとに [送信メッセージ数] * [分散トピックのメンバー数] のメッセージを受信することになります。たとえば、JMS サーバに 2 つの分散トピックのメンバーが含まれている場合、各メンバーに 1 つ、合わせて 2 つの MDB がデプロイされるので、2 倍の数のメッセージを受信することになります。

TopicPublisher

トピック パブリッシャを作成する際、提供されている送り先が分散送り先である場合、その送り先に送信されるメッセージはすべて次のように、分散トピックのアクセス可能なトピック メンバーすべてに送信されます。

- 1 つまたは複数の分散トピック メンバーにアクセスできず、送信するメッセージが非永続的である場合、メッセージはアクセス可能なトピック メンバーのみに送信される。
- 1 つまたは複数の分散トピック メンバーにアクセスできず、送信するメッセージが永続的である場合、メッセージは格納され、アクセスできなかったメンバーにアクセスできるようになったときに転送される。ただし、トピック メンバーに JMS ストアがコンフィグレーションされている場合は、メッセージの永続的な保管のみが行われます。

  注意： 永続ストレージを利用している分散メンバーに対しては、最初にメッセージを転送するためにあらゆる努力がなされます。ただし、分散メンバーがいずれもストレージを利用していない場合でも、メッセージは 4-101 ページの「分散送り先におけるメッセージのロード バランシング」で説明されるように、選択されているロード バランシング アルゴリズムに従って、メンバーのいずれかに送信されます。
- 分散トピック メンバーすべてにアクセスできない場合 (メッセージが永続的であるか非永続的であるかにかかわらず) は、パブリッシャがメッセージを送信しようとしたときに `JMSException` が発行される。

TopicSubscriber

トピック サブスクライバの作成時に、提供されているトピックが分散トピックの場合、トピック サブスクライバはその分散トピックにパブリッシュされたメッセージを受信します。トピック サブスクライバが分散トピックのトピック メンバーの 1 つまたは複数にアクセスできない場合は、メッセージが永続的であるか非永続的であるかに応じて次の事態が発生します。

- アクセスできない 1 つまたは複数の分散トピック メンバーにパブリッシュされた永続メッセージは、これらのメンバーがアクセス可能になったときに、これらのメンバーのトピック サブスクライバによって受信される。ただし、トピック メンバーに JMS ストアがコンフィグレーションされている場合は、メッセージの永続的な保管のみが行われます。
- アクセスできない分散トピック メンバーにパブリッシュされた非永続メッセージは、そのトピック サブスクライバには送信されない。

  注意： 分散トピック メンバーをホストしている JMS サーバで JMS ストアがコンフィグレーションされている場合は、そのメンバー送り先と関連付けられているすべての分散トピック システム サブスクライバが恒久サブスクリプションとして扱われます (トピック メンバーで JMS

ストアが明示的にコンフィグレーションされていない場合も同様)。そのような状況で、それらの分散トピック サブスクライバに送信されたすべてのメッセージを記憶すると、予期せずにメモリとディスクが消費されることになります。したがって、分散送り先をデプロイするときに推奨される設計上のベスト プラクティスは、すべてのメンバー送り先で、恒久メッセージで JMS ストアあり、または非恒久メッセージで JMS ストアなし、という一貫性のあるコンフィグレーションをすることです。たとえば、すべての分散トピック サブスクライバを非恒久にしたいが、関連付けられている JMS サーバで JMS ストアが使用されるために一部のメンバー送り先で JMS ストアが暗黙的にコンフィグレーションされている場合は、各メンバー送り先で明示的に StoreEnabled 属性を False に設定して JMS サーバの設定をオーバーライドする必要があります。

最終的に、トピック サブスクライバは物理的なトピック メンバーに固定されます。そのトピック メンバーにアクセスできなくなった場合は、次のようにトピック サブスクライバに `JMSException` が発行されます。

- トピック サブスクライバが同期サブスクライバである場合は、例外がユーザに直接返される。
- トピック サブスクライバが非同期サブスクライバである場合は、例外が `ConsumerClosedException` の内部に配信され、それがコンシューマ セッションに定義されている `ExceptionListener` (存在する場合) に配信される。

このような例外を受信した場合、アプリケーションはトピック サブスクライバを終了して再作成できます。分散トピック内のその他のトピック メンバーにアクセス可能な場合は、新しいトピック サブスクライバが作成され、これらのトピック メンバーのいずれかに固定されます。他のトピック メンバーにアクセスできない場合、トピック サブスクライバの再作成はできないため、後で作成を再試行する必要があります。

分散送り先メンバーへのアクセス

分散送り先内の分散メンバーにアクセスするには、コンフィグレーションされた JNDI 名を使用して分散メンバーをルックアップするか、JMS サーバ名と `JMSQueueMBean` または `JMSTopicMBean` コンフィグレーション MBean 名をフォワード スラッシュ (`/`) で区切って、`createQueue()` メソッドまたは `createTopic()` メソッドのどちらかに提供する必要があります。

たとえば、次のコードは、JMS サーバ (`myServer`) 上の分散キュー (`myQueue`) の特定のメンバーをルックアップする方法を示します。

```
queue = myQueueSession.createQueue("myServer/myQueue");
```

注意： `createQueue()` メソッドまたは `createTopic()` メソッドを呼び出す際、フォワード スラッシュ (`/`) を含む文字列は、送り先ではなく、分散送り先メンバーの名前であると見なされます。このような分散送り先メンバーが存在しない場合、呼び出しは `InvalidDestinationException` で失敗します。

分散送り先におけるメッセージのロード バランシング

WebLogic JMS は分散送り先を使用して、複数の物理的送り先にメッセージの負荷を分散し、バランスを取ることができます。これによって、リソースがさらに効率的に利用され、応答時間が改善されます。WebLogic JMS のロード バランシング アルゴリズムでは、メッセージの送信先となる物理的送り先に加えて、コンシューマの割り当て先の物理的送り先も決定されます。

分散送り先のロード バランシングのコンフィグレーションの詳細については、『管理者ガイド』の「メッセージのロード バランシングのコンフィグレーション」を参照してください。

ロード バランシング オプション

WebLogic JMS は、任意の送り先セットに含まれている複数の物理的送り先間でメッセージのロード バランシングを行うための 2 種類のアルゴリズムをサポートしています。これらのロード バランシング オプションのどちらかを選択するには、Adminstration Console で分散トピックまたは分散キューをコンフィグレーションします。

- ラウンドロビン分散
- ランダム分散

ラウンドロビン分散

ラウンドロビン アルゴリズムでは、WebLogic JMS は送り先内の物理的送り先の序列を管理します。メッセージの負荷は、WebLogic Server コンフィグレーション (`config.xml`) ファイルに定義されている順番で、一度に 1 つずつ物理的送り先に分散されます。各 WebLogic Server は、同一の序列を維持しますが、その序列内の位置は異なります。特定の分散送り先を使用して単一のサーバ内で複数のスレッドを実行すると、スレッドからメッセージが生成されるたびに、メンバーが割り当てられている物理的送り先に関連して、これらのスレッドが互いに影響し合うことになります。ラウンドロビンはデフォルトのアルゴリズムで、コンフィグレーションの必要はありません。

特定の分散送り先に対し、セット内の物理的送り先のいずれかに重みが割り当てられている場合、それらの物理的送り先は序列内に複数回発生します。たとえば、送り先 A、B、および C の重みがそれぞれ 2、5、および 3 である場合、序列は A、B、C、A、B、C、B、C、B、B となります。つまり、いくつものパスが基本的な序列 (A、B、C) 内に組み込まれます。パスの数は、セット内の送り先の最大の重みと等しくなります。それぞれのパスでは、パスの序数値以上の重みを持つ送り先のみが序列に含められます。このロジックに従うと、ここに挙げた例では次の結果が生じます。

- A は 2 回のパスの後に序列から除外される。
- C は 3 回のパスの後に除外される。
- B だけは 4 回目と 5 回目のパスにも残る。

ランダム分散

ランダム分散アルゴリズムでは、物理的送り先に割り当てられている重みを使用して、一連の物理的送り先の重み付き送り先を計算します。メッセージの負荷は、疑似ランダム的に送り先にアクセスすることで、物理的送り先に分散されます。短期的には、負荷は重みには直接比例しません。長期的には、極限に近い分散が行われます。純粋なランダム分散は、重みすべてに同じ値を設定することで達成できます。この値は通常 1 です。

メンバーが追加または除去された (管理者が行うか、WebLogic Server がシャットダウンまたは再起動した結果発生したもの) 場合は、分散を再計算する必要があります。ただし、このようなイベントは頻繁に発生せず、計算は通常単純で、O(n) 時間で実行されます。

コンシューマのロード バランシング

アプリケーションがコンシューマを作成するときは、送り先を提供する必要があります。その送り先が分散送り先を表す場合、WebLogic JMS ではコンシューマのメッセージ受信元となる物理的送り先を見つける必要があります。どの送り先メンバーを使用するかは、4-102 ページの「ロード バランシング オプション」に説明されているロード バランシング アルゴリズムのいずれかを用いて選択します。選択は一度だけ、つまりコンシューマが作成されたときに行います。その後、コンシューマはそのメンバーのみからメッセージを受信するようになります。

プロデューサのロード バランシング

プロデューサがメッセージを送信するとき、WebLogic JMS はメッセージが送信される送り先を確認します。送り先が分散送り先である場合は、メッセージの送信先が WebLogic JMS によって決定されます。つまり、プロデューサは 4-102 ページの「ロード バランシング オプション」で説明されているロード バランシング アルゴリズムのいずれかに従って、送り先のメンバーのいずれかに送信します。

プロデューサはメッセージを送信するたびにこのような決定を行います。ただし、コンシューマとプロデューサ間の序列の保証が脅かされることはありません。コンシューマのロード バランシングがいったん行われると、コンシューマは単一の送り先メンバーに固定されるためです。

注意： プロデューサが永続メッセージを分散送り先に送信しようとした場合、永続ストレージを利用している分散メンバーにまずメッセージが転送されるよう、あらゆる努力をします。ただし、分散メンバーがいずれも永続ストレージを利用していない場合でも、メッセージは選択されているロードバランシング アルゴリズムに従って、メンバーのいずれかに送信されます。

ロード バランシングのヒューリスティック

4-102 ページの「ロード バランシング オプション」に説明されているアルゴリズムに加えて、WebLogic JMS では送り先のインスタンスを選択するときに次のヒューリスティックが使用されます。

- トランザクション アフィニティ
- サーバ アフィニティ
- コンシューマのないキュー

トランザクション アフィニティ

トランザクション セッション内で複数のメッセージを作成するときは、作成されたメッセージすべてを同じ WebLogic Server に送信しようとします。つまり、セッションが単一の分散送り先に複数のメッセージを送信した場合は、すべてのメッセージが同じ物理的送り先に転送されます。セッションによって複数のメッセージが複数の異なる分散送り先に送信された場合は、同じ WebLogic Server がサービスを提供する一連の物理的送り先が選択されるようになります。

サーバ アフィニティ

分散送り先でサーバ アフィニティ オプションが有効になっている場合、ドメイン内の分散送り先のすべてのメンバーでコンシューマまたはプロデューサのロード バランシングを試行する前に、WebLogic Server インスタンスはまず同じ WebLogic Server インスタンスで動作しているローカル メンバーの間でロード バランシングを試行します。

JMS 接続ファクトリの [サーバ アフィニティを有効化] オプションを使用した分散送り先のサーバ アフィニティのコンフィグレーションに関する詳細については、『管理者ガイド』の「分散送り先のサーバ アフィニティのコンフィグレーション」を参照してください。

注意： [Server Affinity Enabled サーバ アフィニティを有効化] 属性は、キューブラウザに影響を与えません。したがって、分散キューで作成されたキューブラウザは、サーバ アフィニティが有効な場合でもリモートの分散キュー メンバーに固定できます。

コンシューマのないキュー

複数のリモートの物理的キュー間でコンシューマのロード バランシングを行う場合、1 つまたは複数のキューにコンシューマがないとき、それらのキューだけがロード バランシングの対象となります。セット内の物理的キューすべてに少なくとも 1 つのコンシューマが含まれるようになると、標準のアルゴリズムが適用されます。

また、プロデューサがメッセージを送信するとき、コンシューマのないキューはメッセージ作成の対象にはなりません。ただし、任意のキューのインスタンスのいずれにもコンシューマが含まれていない場合を除きます。

ロード バランシングの回避

アプリケーションは、個別の物理的送り先に直接アクセスすることで、ロード バランシングを回避できます。つまり、物理的送り先に JNDI 名がない場合でも、`createQueue()` メソッドまたは `createTopic()` メソッドを使用して参照することができます。

- JNDI ルックアップ
- CreateQueue() and CreateTopic()
- 接続ファクトリ

JNDI ルックアップ

物理的送り先に JNDI 名がある場合は、JNDI を使用してルックアップできます。それによって返された送り先を使用すると、コンシューマまたはレシーバを作成できます。

CreateQueue() and CreateTopic()

アプリケーションでは、`createQueue()` メソッドや `createTopic()` メソッドを使用してトピックまたはキューの参照を取得することもできます。これらのメソッドを使用するときは、参照先の送り先を識別するベンダ固有の文字列を提供する必要があります。WebLogic JMS 用のベンダ固有の文字列は、*server/destination* の形式になっています。ここで、「server」は JMS サーバ名で、「destination」はその JMS サーバ上のキューまたはトピックの名前を表します。

接続ファクトリ

アプリケーションにおいて、分散送り先を使用して、複数の物理的送り先間でプロデューサやコンシューマを分散、つまりバランスさせることは行っても、メッセージが作成されるたびにロード バランシングの決定を行いたくない場合は、[ロード バランスを有効化] 属性を無効 (False に設定) にした状態で、接続ファクトリを使用します。

分散送り先のロード バランシングのコンフィグレーションの詳細については、『管理者ガイド』の「メッセージのロード バランシングのコンフィグレーション」を参照してください。

[サーバ アフィニティを有効化] 属性を使用した場合の分散送り先のロード バランシングへの影響

次の表では、JMS 接続ファクトリの [サーバ アフィニティを有効化] 属性の設定が、分散送り先メンバーのロード バランシングの優先順位にどのように影響するかを説明します。ロード バランシングの優先順位は、操作のタイプや、恒久サブスクリプションまたは永続メッセージが関係するかどうかによって異なります。

分散送り先の [サーバ アフィニティを有効化] 属性は、`ClusterMBean` の [デフォルトのロード バランス アルゴリズム] 属性によって提供されるサーバ アフィニティとは異なります。後者は、JMS 接続ファクトリによって、クライアント接続の初期コンテキスト アフィニティの作成に使用されます。詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「EJB と RMI オブジェクトのロード バランシング」を参照してください。

表 4-7 サーバ アフィニティのロード バランシングの優先順位

実行する操作	[サーバ アフィニティを有効化] 属性	ロード バランシングの優先順位
■ キューの `createReceiver()` ■ トピックの `createDurableSubscriber()`	True	1. コンシューマを持たないローカル メンバー 2. ローカル メンバー 3. コンシューマを持たないリモート メンバー 4. リモート メンバー
キューの `createReceiver()`	False	1. コンシューマを持たないメンバー 2. メンバー
トピックの `createSubscriber()` (**注意 :** 非恒久サブスクライバ)	True または False	1. コンシューマを持たないローカル メンバー 2. ローカル メンバー 3. ローカル プロキシ トピック (『管理者ガイド』の「分散送り先のシステム サブスクリプションとプロキシ トピック メンバーのモニタ」を参照)
■ キューの `createSender()` ■ トピックの `createPublisher()`	True または False	JMS プロデューサの作成におけるロード バランシングは、別のマシンで行われるわけではない。JMS プロデューサは、JMS 接続のロード バランシングを行うサーバまたは JMS 接続の固定先となるサーバで作成される。. 接続ファクトリを使用して作成した JMS 接続のロード バランシングについては、『WebLogic Server クラスタ ユーザーズ ガイド』の「EJB と RMI オブジェクトのロード バランシング」を参照。

表 4-7 サーバ アフィニティのロード バランシングの優先順位

実行する操作	[サーバ アフィニティを有効化] 属性	ロード バランシングの優先順位
永続メッセージの ■ `QueueSender.send()`	True	1. コンシューマとストアを持つローカル メンバー 2. コンシューマとストアを持つリモート メンバー 3. ストアを持つローカル メンバー 4. ストアを持つリモート メンバー 5. コンシューマを持つローカル メンバー 6. コンシューマを持つリモート メンバー 7. ローカル メンバー 8. リモート メンバー
永続メッセージの ■ `QueueSender.send()`	False	1. コンシューマとストアを持つメンバー 2. ストアを持つメンバー 3. コンシューマを持つメンバー 4. メンバー
非永続メッセージの ■ `QueueSender.send()`	True	1. コンシューマを持つローカル メンバー 2. コンシューマを持つリモート メンバー 3. ローカル メンバー 4. リモート メンバー

表 4-7 サーバ アフィニティのロード バランシングの優先順位

実行する操作	[サーバ アフィニティを有効化] 属性	ロード バランシングの優先順位
非永続メッセージの ■ `QueueSender.send()`	False	1. コンシューマを持つメンバー 2. メンバー
セッション プール キューおよびトピックの `createConnectionConsumer()`	True または False	1. ローカル メンバーのみ **注意：**セッション プールは、現在はあまり使用されない。その理由としては、J2EE 仕様で必要なくなったこと、JTA ユーザ トランザクションをサポートしていないこと、よりシンプルで管理がしやすく機能性の高いメッセージ駆動型 Bean (MDB) で代用されるようになったことなどが挙げられる。

分散送り先の移行

WebLogic Server 7.0 サービス移行機能を利用する JMS の実装では、JMS サーバに障害が発生した場合、クラスタ内の別の WebLogic Server に、分散送り先のメンバーとともにサーバを移行できます。ただし、移行先の WebLogic Server は物理的送り先すべてを持つ JMS サーバを既にホストしている可能性があります。このような場合は、同じ WebLogic Server が単一の送り先に対して 2 つの物理的送り先をホストするような状況が生じます。WebLogic Server はその送り先に対して複数の物理的送り先をホストできるため、これは短期的には許容できますが、このような状況でのロード バランシングは効果が半減します。

このような状況では、移行先の WebLogic Server 上の JMS サーバは独立して動作します。これは、2 つの送り先インスタンスの結合や、1 つのインスタンスの無効化を避けるためです。このようなことが発生すると、メッセージが長時間使用できなくなる場合があります。ただし、長期的な目的は、移行された JMS サーバをクラスタ内のさらに別の WebLogic Server に最終的に再移行することにあります。

JMS 移行可能先のコンフィグレーションの詳細については、3-7 ページの「JMS 移行可能対象のコンフィグレーション」を参照してください。

分散送り先のフェイルオーバ

JMS プロデューサと JMS コンシューマの JMS 接続をホストしているサーバ インスタンスで障害が発生した場合、その接続を使用するすべてのプロデューサとコンシューマが閉じられ、それらがクラスタ内の別のサーバ インスタンスで再作成されることはありません。JMS 送り先をホストしているサーバ インスタンスで障害が発生した場合にも、その送り先の JMS コンシューマがすべて閉じられ、それらがクラスタ内の別のサーバ インスタンスで再作成されることはありません。

キュー プロデューサが作成された分散キュー メンバーに障害が発生したにもかかわらず、プロデューサの JMS 接続が存在する WebLogic Server インスタンスが引き続き稼動している場合、プロデューサはそのまま稼動し、[ロード バランス] オプションが有効になっているかどうかに関係なく、WebLogic JMS によって別の分散キュー メンバーにフェイルオーバされます。

WebLogic Server の障害の回復手順の詳細については、『管理者ガイド』の「JMS の管理」を参照してください。

5 WebLogic JMS によるトランザクションの使い方

以下の節では、WebLogic JMS でトランザクションを使用する方法について説明します。

- トランザクションの概要
- JMS トランザクション セッションの使い方
- JTA ユーザ トランザクションの使い方
- メッセージ駆動型 Bean を使用した JTA ユーザ トランザクション内の非同期メッセージング
- 例 : JTA ユーザ トランザクションにおける JMS と EJB

注意： この節で説明する JMS クラスの詳細については、Sun Microsystems の Java Web サイトにある以下の最新の JMS 仕様と Javadoc を参照してください。（http://java.sun.com/products/jms/docs.html）

トランザクションの概要

トランザクションを使用すると、アプリケーションでは生成および消費されるメッセージのグループを調整し、送受信する複数のメッセージを基本単位として処理できます。

アプリケーションがトランザクションをコミットすると、トランザクション内で受信した全メッセージはメッセージング システムから削除され、トランザクション内で送信したメッセージが実際に配信されます。アプリケーションによってトランザクションがロールバックされた場合、トランザクション内で受信したメッセージはメッセージング システムに戻され、送信したメッセージは破棄されます。

トピック サブスクライバによってロールバックされた受信メッセージはサブスクライバに再配信されます。キュー レシーバによってロールバックされた受信メッセージは、コンシューマではなくキューに再配信されます。それによって、キュー内の他のコンシューマがメッセージを受信できるようにします。

たとえばオンライン ショッピングでは、品物を選択し、それをオンライン ショッピング カートに入れます。注文した品物はトランザクションの一部として格納されますが、チェックアウトして注文を確定するまでユーザの支払い義務は発生しません。ユーザはいつでも注文をキャンセルし、カートを空にすることができます。キャンセルによって、現在のトランザクション内で注文がロールバックされます。

JMS でトランザクションを使用する方法には以下の 3 種類があります。

- トランザクションで JMS のみを使用する場合は、JMS トランザクション セッションを作成できます。
- EJB などの他の処理と JMS を混在させる場合は、JMS 非トランザクション セッションで Java Transaction API（JTA）ユーザ トランザクションを使用します。
- メッセージ駆動型 Bean を使用します。

1 つの JTA ユーザ トランザクションで複数の JMS サーバを有効にする場合、または JMS の処理と 非 JMS の処理（EJB など）を組み合わせる場合は、2 フェーズ コミット ライセンスが必要です。詳細については、5-5 ページの「JTA ユーザ トランザクションの使い方」を参照してください。

以降の節では、JMS トランザクション セッションと JTA ユーザ トランザクションの使い方について説明します。

注意： トランザクションを使用する場合、トランザクションがコミットまたはロールバックされる前に発生する問題に対処するために、4-50 ページの「セッション例外リスナの定義」で説明しているようにセッション例外リスナを定義しておくことをお勧めします。

`acknowledge()` メソッドは、トランザクション内で呼び出されても無視されます。`recover()` メソッドがトランザクション内で呼び出されると、JMSException が送出されます。

JMS トランザクション セッションの使い方

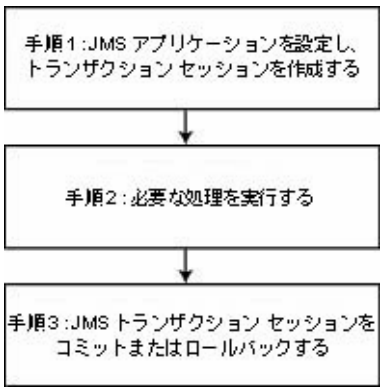
JMS トランザクション セッションは、セッション内にあるトランザクションをサポートします。JMS トランザクション セッションのトランザクションはセッション外には影響を及ぼしません。たとえば、セッションをロールバックしても、そのセッションの送受信がロールバックされるだけで、データベース更新はロールバックされません。JTA ユーザ トランザクションは JMS トランザクション セッションでは無視されます。

JMS トランザクション セッションのトランザクションは、最初の送受信処理が発生した後で暗黙的に開始され、互いに結び付けられます。トランザクションをコミットまたはロールバックすると、他のトランザクションが自動的に始まります。

『管理者ガイド』の「JMS の管理」の説明に従って、システム管理者は、JMS トランザクション セッションを使用する前に、アプリケーション開発環境の必要性に応じて、接続ファクトリの属性（トランザクション タイムアウト）とセッション プールの属性（トランザクション）を調整する必要があります。

JMS トランザクション セッションの設定および使用に必要な手順を次の図に示します。

図 5-1 JMS トランザクション セッションの設定と使用



手順 1 : JMS アプリケーションを設定し、トランザクション セッションを作成する

4-4 ページの「JMS アプリケーションの設定」の説明に従って JMS アプリケーションを設定しますが、4-8 ページの「手順 3 : 接続を使用してセッションを作成する」でセッションを作成する際に、ブール値 `transacted` を `true` に設定してセッションをトランザクション処理されるように指定します。

たとえば、PTP および Pub/sub メッセージング モデルのトランザクション セッションを作成する方法を次の各メソッドで示します。

```
qsession = qcon.createQueueSession(
  true,
  Session.AUTO_ACKNOWLEDGE
);

tsession = tcon.createTopicSession(
  true,
  Session.AUTO_ACKNOWLEDGE
);
```

定義したら、次のセッション メソッドでセッションをトランザクション処理するかどうかを決定できます。

```
public boolean getTransacted(
) throws JMSException
```

注意 : acknowledge 値はトランザクション セッションでは無視されます。

手順 2 : 必要な処理を実行する

現在のトランザクションで必要な処理を実行します。

手順 3 : JMS トランザクション セッションをコミットまたはロールバックする

必要な処理を実行したら、以下のメソッドのいずれかを実行してトランザクションをコミットまたはロールバックします。

トランザクションをコミットするには、次のメソッドを実行します。

```
public void commit(
) throws JMSException
```

`commit()` メソッドでは、現在のトランザクションの送受信メッセージがすべてコミットされます。受信メッセージはメッセージング システムから削除されますが、送信メッセージは表示されるようになります。

トランザクションをロールバックするには、次のメソッドを実行します。

```
public void rollback(
) throws JMSException
```

`rollback()` メソッドでは、現在のトランザクションの送信メッセージがキャンセルされ、受信メッセージがメッセージング システムに戻されます。

`commit()` メソッドまたは `rollback()` メソッドが JMS トランザクション セッション以外で発行された場合、`IllegalStateException` が送出されます。

JTA ユーザ トランザクションの使い方

Java Transaction API（JTA）は、複数のデータ リソースにわたるトランザクションをサポートします。JTA は WebLogic Server の一部として実装され、トランザクション管理を実装するための標準 Java インタフェースを提供します。

トランザクションを開始、コミット、ロールバックするための `javax.transaction.UserTransaction` オブジェクトを使用して JTA ユーザ トランザクション アプリケーションをプログラミングします。JTA ユーザ トラン

ザクション内に JMS と EJB を混在させる場合、『WebLogic JTA プログラマーズ ガイド』の「EJB アプリケーションのトランザクション」で説明しているとおりに EJB からトランザクションを開始することもできます。

トランザクション セッションの開始後に JTA ユーザ トランザクションを開始できます。ただし、JTA ユーザ トランザクションはトランザクション セッションに無視され、トランザクション セッションは JTA ユーザ トランザクションに無視されます。

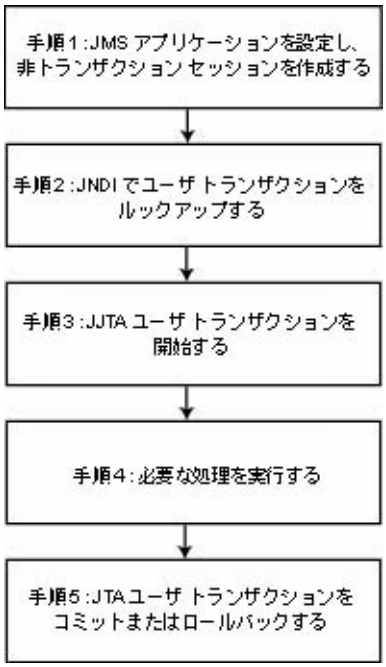
WebLogic Server は 2 フェーズ コミット（2PC）プロトコルをサポートしています。2PC では、複数のリソース マネージャ間で 1 つの JTA トランザクションを効率的に調整できるようになります。これにより、トランザクションによる更新を関連するリソース マネージャのすべてでコミットするか、またはすべてのリソース マネージャから完全にロール バックし、トランザクション開始前の状態に戻すことで、データの完全性が保証されます。

注意： このプロトコルをサポートするには、独自の 2PC トランザクション ライセンスが必要です。2PC が関連するトランザクションの移行の詳細については、6-1 ページの「WebLogic JMS アプリケーションの移植」を参照してください。

『Administration Console オンライン ヘルプ』の「JMS 接続ファクトリ」の説明に従って、システム管理者は、JTA トランザクション セッションを使用する前に、[ユーザ トランザクションを有効化] チェック ボックスをオンにして、JTA ユーザ トランザクションをサポートするように接続ファクトリをコンフィグレーションする必要があります。

JTA ユーザ トランザクションの設定および使用に必要な手順を次の図に示します。

図 5-2 JTA ユーザ トランザクションの設定と使用



手順 1 : JMS アプリケーションを設定し、非トランザクション セッションを作成する

4-4 ページの「JMS アプリケーションの設定」の説明に従って JMS アプリケーションを設定しますが、4-8 ページの「手順 3 : 接続を使用してセッションを作成する」でセッションを作成する際に、ブール値 `transacted` を `false` に設定してセッションをトランザクション処理されないように指定します。

たとえば、PTP および Pub/sub メッセージング モデルの非トランザクション セッションを作成する方法を次の各メソッドで示します。

```
qsession = qcon.createQueueSession(
  false,
```

```
  Session.AUTO_ACKNOWLEDGE
);
tsession = tcon.createTopicSession(
  false,
  Session.AUTO_ACKNOWLEDGE
);
```

注意： ユーザ トランザクションがアクティブな場合、確認応答モードは無視されます。

手順 2 : JNDI でユーザ トランザクションをルックアップする

アプリケーションは、JNDI を使用して、WebLogic Server ドメインの `UserTransaction` オブジェクトに対するオブジェクト参照を返します。

JNDI コンテキスト（`context`）を確立して次のようなコードを実行すると、`UserTransaction` オブジェクトをルックアップできます。

```
UserTransaction xact =
ctx.lookup("javax.transaction.UserTransaction");
```

手順 3 : JTA ユーザ トランザクションを開始する

`UserTransaction.begin()` メソッドを使用して JTA ユーザ トランザクションを開始します。次に例を示します。

```
xact.begin();
```

手順 4 : 必要な処理を実行する

現在のトランザクションで必要な処理を実行します。

手順 5 : JTA ユーザ トランザクションをコミットまたはロールバックする

必要な処理を実行したら、UserTransaction オブジェクトの commit() メソッドまたは rollback() メソッドを実行して、JTA ユーザ トランザクションをコミットまたはロールバックします。

トランザクションをコミットするには、次のメソッドを実行します。

```
xact.commit();
```

commit() メソッドを実行すると、WebLogic Server はトランザクション マネージャを呼び出してトランザクションを完了し、現在のトランザクションで実行されている全処理をコミットします。トランザクション マネージャの役割は、リソース マネージャと協力してデータベースを更新することです。

トランザクションをロールバックするには、次のメソッドを実行します。

```
xact.rollback();
```

rollback() メソッドを実行すると、WebLogic Server はトランザクション マネージャを呼び出してトランザクションをキャンセルし、現在のトランザクションで実行されている全処理をロールバックします。

commit() または rollback() メソッドを呼び出したら、xact.begin() を呼び出して別のトランザクションを開始することもできます。

メッセージ駆動型 Bean を使用した JTA ユーザ トランザクション内の非同期メッセージング

JMS では非同期的に配信されるメッセージでどのトランザクションを使用するかを決定できないため、JMS 非同期メッセージ配信は JTA ユーザ トランザクションではサポートされません。

ただし、メッセージ駆動型 Bean による代わりの方法が提供されます。メッセージ駆動型 Bean では、メッセージ配信の直前にユーザ トランザクションを自動的に開始できます。

メッセージ駆動型 Bean による非同期メッセージ配信については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』の「メッセージ駆動型 Bean の設計」を参照してください。

例：JTA ユーザ トランザクションにおける JMS と EJB

以下の例では、JNDI を使用して javax.transaction.UserTransaction をルックアップすることにより、JTA ユーザ トランザクション内に EJB と JMS の処理が混在するようにアプリケーションを設定し、JTA ユーザ トランザクションを開始してからコミットする方法を示します。この例を実行するには、システム管理者が接続ファクトリをコンフィグレーションするときに、[ユーザ トランザクションを有効化] チェック ボックスをオンにする必要があります。

注意： この単純な JTA ユーザ トランザクションの例に加えて、*WL_HOME*\samples\server\src\examples\jta\jmsjdcb ディレクトリにある WebLogic JTA 付属の例も参照してください。*WL_HOME* は、WebLogic Platform のインストール先の最上位ディレクトリです。

javax.transaction.UserTransaction パッケージを含む適切なパッケージをインポートします。

```
import java.io.*;
import java.util.*;
import javax.transaction.UserTransaction;
import javax.naming.*;
import javax.jms.*;
```

JTA ユーザ トランザクション変数などの必要な変数を定義します。

```
public final static String JTA_USER_XACT=
       "javax.transaction.UserTransaction";
       .
       .
       .
```

手順 1　JMS アプリケーションを設定し、非トランザクション セッションを作成します。JMS アプリケーションの設定については、4-4 ページの「JMS アプリケーションの設定」を参照してください。

```
// JMS アプリケーションの設定手順の例は次のとおり
  qsession = qcon.createQueueSession(false,
                  Session.CLIENT_ACKNOWLEDGE);
```

手順 2　JNDI で UserTransaction をルックアップします。

```
UserTransaction xact = (UserTransaction)
                ctx.lookup(JTA_USER_XACT);
```

手順 3　JTA ユーザ トランザクションを開始します。

```
xact.begin();
```

手順 4　必要な処理を実行します。

```
// JMS および EJB 処理をここで実行
```

手順 5　JTA ユーザ トランザクションをコミットします。

```
xact.commit()
```

6 WebLogic JMS アプリケーションの移植

以下の節では、新しいバージョンの WebLogic Server に WebLogic JMS アプリケーションを移植する方法について説明します。

- 既存の機能の変更点
- 既存のアプリケーションの移植
- JDBC データベース ストアの削除

既存の機能の変更点

Sun Microsystem の JMS 仕様に従って既存の機能が変更されました。移植手順を開始する前に、次の表で変更点を確認してください。

- 5.1 と 6.0 の既存の機能の変更点
- 6.0 と 6.1 の既存の機能の変更点

5.1 と 6.0 の既存の機能の変更点

次の表に、WebLogic Server バージョン 5.1 の既存の機能の変更点を示します。また、その結果としてコードを変更する必要がある場合は、そのコードも併せて示します。JMS 仕様のバージョン変更履歴については、仕様の第 11 章「Change History」を参照してください。

カテゴリ	説明	コードの変更点
接続ファクトリ	2 つのデフォルト接続ファクトリが非推奨になっている。該当するファクトリの JNDI 名は次のとおり。 ■ `javax.jms.QueueConnectionFactory` ■ `javax.jms.TopicConnectionFactory` 下位互換性を維持するため、これら 2 つの接続ファクトリの JNDI 名は現在でも定義およびサポートされている。	新しいデフォルトまたはユーザ定義の接続ファクトリ クラスを使用するために、非推奨のクラスを使用している既存のコードを変更することが望ましい。 例として、デフォルトのキュー接続ファクトリを使用して次の定数を指定した場合を示す。 `public final static String JMS_FACTORY="javax.jms.QueueConnectionFactory"`
	WebLogic JMS 6.*x* 以降では、`weblogic.jms.ConnectionFactory` という接続ファクトリがデフォルトで定義されている。 Administration Console を使用して、ユーザ定義の接続ファクトリを指定することもできる。 **注意：** デフォルトの接続ファクトリを使用する場合は、接続ファクトリがデプロイされる可能性のある WebLogic Server を限定できない。特定の WebLogic Server を対象にする場合は、新しい接続ファクトリを作成し、適切な WebLogic Server の対象を指定する。	この場合、新しいユーザ定義の接続ファクトリを使用するためには定数を次のように変更する。 `public final static String JMS_FACTORY="weblogic.jms.QueueConnectionFactory"` 旧バージョンとの下位互換性を保つためには、接続ファクトリのコンフィグレーション時に [メッセージの短縮を許可] チェック ボックスおよび [ユーザ トランザクションを有効化] チェック ボックスをオンにする必要がある。 接続ファクトリの定義の詳細については、Administration Console オンライン ヘルプの「JMS 接続ファクトリ」を参照。
	特定の WebLogic Server でデフォルト接続ファクトリをインスタンス化するには、WebLogic Server のコンフィグレーション時に [デフォルト JMS 接続ファクトリを有効化] チェックボックスをオンにする。	変更不要。コンフィグレーションで必要とされる。詳細については、Administration Console オンライン ヘルプの「JMS サーバ」を参照。

カテゴリ	説明	コードの変更点
接続	接続を閉じると、未処理の同期呼び出しと非同期リスナの処理が完了するまで呼び出しがブロックされる。	変更不要。
セッション	セッションを閉じると、未処理の同期呼び出しおよび非同期リスナの処理が完了するまで、呼び出しがブロックされる。	変更不要。
メッセージコンシューマ	1 つのセッションで 1 つのトピックに対して複数のトピック サブスクライバが定義されている場合、各コンシューマはメッセージのコピーを受信する。	変更不要。
	メッセージ コンシューマをクローズすると、メソッド呼び出しが完了し、未処理の同期アプリケーションがすべてキャンセルされるまで、呼び出しはブロックされる。	変更不要。
	JMS 仕様に準拠するために、接続ファクトリのコンフィグレーション時に [メッセージの短縮を許可] チェック ボックスをオンにしない限り、アプリケーションは、`onMessage()` メソッド内で `close()` メソッドが呼び出されても応答しない。確認応答モードが `AUTO_ACKNOWLEDGE` の場合、現在のメッセージはまだ自動的に確認応答される。	変更不要。コンフィグレーションで必要とされる。詳細については、Administration Console オンライン ヘルプの「JMS 接続ファクトリ」を参照。

カテゴリ	説明	コードの変更点

メッセージ ヘッダ フィールド	JMSMessageID ヘッダ フィールドの形式は変更されている。	JMSMessageID で既存のメッセージにアクセスする場合は、以下の weblogic.jms.extensions.JMSHelper メソッドのいずれかを実行して、JMSMessageID の形式を WebLogic JMS 5.1 以前から JMS 6.*x* に変換しなければならない場合がある。
		5.1 以前の JMSMessageID の形式を 6.*x* に変換する場合には、以下を実行する。
		`public void oldJMSMessageIDToNew( String id, long timeStamp ) throws JMSException`
		6.1 の JMSMessageID の形式を 6.1 以前に変換する場合には、以下を実行する。
		`public void newJMSMessageIDToOld( String id, long timeStamp ) throws JMSException`

カテゴリ	説明	コードの変更点
送り先	`createQueue()` メソッドと `createTopic()` メソッドでは、送り先が動的には作成されず、ベンダ固有の送り先名で既に存在する送り先への参照が作成されるのみ。	`createQueue()` または `createTopic()` を使用しているコードの一部を変更し、`createPermanentQueueAsync()` および `createPermanentTopicAsync()` という JMSHelper クラス メソッドをそれぞれ使用して動的に送り先を作成する。 例として、動的にキューを作成するために次のメソッドを使用する場合を示す。 `queue=qsession.createQueue(queueName);` キューを動的に作成するには、4-53 ページの「JMSHelper クラス メソッドの使い方」のサンプル findQueue() メソッドで説明されているようにコードを変更する。 JMSHelper クラスの詳細については、4-53 ページの「送り先の動的作成」を参照。
	一時的な送り先を作成する場合は、一時的なテンプレートを指定する必要がある。	変更不要。コンフィグレーションで必要とされる。詳細については、Administration Console オンライン ヘルプの「JMS テンプレート」を参照。
	その一時的な送り先のメッセージ コンシューマを作成するには、接続のオーナであることが必要。	一時的な送り先にメッセージ コンシューマを作成する場合は、自分が接続のオーナであることを確認する。

カテゴリ	説明	コードの変更点
恒久サブスクライバ	恒久サブスクライバ用に JDBC テーブルを手動で作成する必要はない。自動的に作成される。	変更不要。
	恒久サブスクライバは必要な数だけ作成可能。	変更不要。
	クライアント ID をプログラムで定義する場合は、接続を作成した直後に定義する必要がある。それ以外の場合、例外が送出され、その接続では他の JMS 呼び出しができなくなる。	`setClientID()` メソッドが、接続作成の直後に発行されるようにする。詳細については、4-58 ページの「クライアント ID の定義」を参照。
セッションプール	セッション プール ファクトリ、セッション プール、参照される接続ファクトリ、参照される送り先、関連する接続コンシューマは、すべて同じ JMS サーバを対象にする必要がある。	全オブジェクトが同じ JMS サーバを対象としていることを確認すること。
	WebLogic JMS バージョン 5.1 の Javadoc の一部として提供されていた `SessionPoolManager` インタフェースと `ConnectionConsumerManager` インタフェースはシステム インタフェースであり、クライアント アプリケーションでは使用しないため、バージョン 6.*x* 以降の Javadoc では削除されている。	使用している場合は、これらのオブジェクトへの参照をクライアント アプリケーションから削除すること。

カテゴリ	説明	コードの変更点
トランザクション	JMS と EJB のデータベース呼び出しを同じトランザクション内で組み合わせて使用するには、2 フェーズ コミット（2PC）が必要である。WebLogic Server の以前のリリースでは、同じデータベース接続プールを使用すれば、この 2 つを組み合わせて使用することができた。	変更不要。
	受信キュー メッセージを回復またはロールバックすると、キューにある全コンシューマに対してそのメッセージが使用可能になる。WebLogic Server の以前のリリースでは、ロールバックされたメッセージは、そのメッセージをロールバックしたセッションでのみ（そのセッションが終了するまで）使用可能だった。	変更不要。

6.0 と 6.1 の既存の機能の変更点

次の表に、WebLogic Server 6.0 の既存の機能の変更点を示します。また、その結果としてコードを変更する必要がある場合は、そのコードも併せて示します。JMS 仕様の変更履歴については、Sun Microsystem の JMS 仕様の第 11 章「Change History」を参照してください。

カテゴリ	説明	コードの変更点
接続ファクトリ	Administration Console の [確認応答ポリシー] 属性の新しいデフォルト値 [All] は、JMS 仕様の変更に従った回避策。この新しいデフォルト設定は、JMS の旧バージョンからの変更を表す。旧バージョンでは、内部で [Previous] がデフォルトとなっており、Administration Console のオプションとしては表示されていなかった。 接続ファクトリのメッセージの確認応答ポリシーとしては、非トランザクション セッションに `CLIENT_ACKNOWLEDGE` モードを使用する実装にのみ、[確認応答ポリシー] 属性が適用される。 ■ [All] - どのメッセージが確認応答メソッドを呼び出すかにかかわらず、指定したセッションで受信したすべてのメッセージを確認応答する。 ■ [Previous] - 指定したセッションで受信したすべてのメッセージを確認応答する。ただし、確認応答メソッドを呼び出したメッセージを最後に、確認応答を中止する。 メッセージの確認応答モードの詳細については、2-11 ページの「非トランザクション セッション」を参照。 **注意：** メッセージ駆動型 Bean（MDB）で使用される接続ファクトリでは、[確認応答ポリシー] フィールドを常に [Previous] に設定する。デフォルト MDB 接続ファクトリでは既に設定されているが、外部の接続ファクトリでは設定されていない可能性がある。	確認応答メソッドを呼び出すメッセージ以前の受信メッセージを確認応答する場合は、Administration Console の [JMS 接続ファクトリ] タブで、デフォルトの [確認応答ポリシー] 設定を [All] から [Previous] に変更する。

カテゴリ	説明	コードの変更点
送り先	WLS 6.0 では、JMS マニュアルは `JMSDestinationMBean` の `StoreEnabled` 属性の `default`、`true`、および `false` の値を正しく指定しているが、ソフトウェアは大文字と小文字を区別しなかった。しかし、バージョン 6.1 以降では、`StoreEnabled` 設定をすべて小文字にする必要がある。	変更不要。コンフィグレーションで必要とされる。詳細については、Administration Console オンライン ヘルプの「JMS テンプレート」を参照。

既存のアプリケーションの移植

WebLogic Server のこのリリースでは、Sun Microsystem の JMS 仕様をサポートしています。既存の JMS アプリケーションを使用するには、WebLogic Server のバージョンを確認してから、この節で説明されている適切な移植手順を実行する必要があります。

- 4.5 および 5.1 アプリケーションのバージョン 6.x への移植手順
- 6.0 アプリケーションの 6.1 への移植手順
- 6.x アプリケーションの 7.0 への移植手順

始める前に

移植手順を開始する前に、以下のリストをチェックして、現在の WebServer JMS のバージョンで移植がサポートされているかどうか、およびそのバージョンに特殊な移植ルールが適用されているかどうかを確認する必要があります。

- Weblogic Server 4.5.1 - 移植は、SP15 に対してのみサポートされています。すべてのサービス パックを実行している場合は、BEA カスタマ サポートまでお問い合わせください。
- Weblogic Server 5.1 - SP07 または SP08 の場合、既存の JDBC ストアをバージョン 7.0 に移植する前に BEA カスタマ サポートに連絡する必要があります。

- オブジェクト メッセージを移植するには、オブジェクト クラスが、Weblogic Server 7.0 のサーバ CLASSPATH 内になければなりません。
- Weblogic Server 7.0 でコンフィグレーションされていない送り先では、移植されたメッセージは失敗し、イベントがログに記録されます。

- WebLogic Server 6.*x* - すべてのアプリケーションがバージョン 7.0 でサポートされています。ただし、可用性の高い新しい JMS 機能をアプリケーションで利用したい場合は、既存の物理的な送り先（キューおよびトピック）を単一の分散送り先の一部として構成する必要があります。詳細については、『WebLogic JMS プログラマーズ ガイド』の「分散送り先の使用」を参照してください。

4.5 および 5.1 アプリケーションのバージョン 6.x への移植手順

WebLogic JMS 6.*x* アプリケーションを使用するには、その前に WebLogic Server バージョン 4.5 および 5.1 のコンフィグレーション データとメッセージ データを次の手順で移植する必要があります。

1. 移植手順を開始する前に、WebLogic Server の旧バージョンを正しくシャットダウンします。

   警告： メッセージの処理中に WebLogic Server の旧バージョンを突然停止すると、移植の際に問題が発生することがあります。旧バージョンのサーバをシャットダウンして WebLogic Server バージョン 6.*x* に移植する前に、処理が非アクティブになっている必要があります。

2. 『インストール ガイド』で説明されているとおりに、WebLogic Server 環境をアップグレードします。
3. コンフィグレーション変換機能を使用してコンフィグレーション情報を移植します。

   コンフィグレーションの移植時に、以下のデフォルト キュー接続ファクトリおよびデフォルト トピック接続ファクトリが有効になります。

   - `javax.jms.QueueConnectionFactory`
   - `javax.jms.TopicConnectionFactory`
   - `weblogic.jms.ConnectionFactory`

最初の 2 つの接続ファクトリは非推奨になっていますが、下位互換性のためにこのリリースでも定義されており、使用できます。新しいデフォルト接続ファクトリの詳細については、6-2 ページの「5.1 と 6.0 の既存の機能の変更点」の表を参照してください。

JMS の管理者は、コンフィグレーションの変換結果を見直して、アプリケーションのニーズが満たされているかどうかを確認する必要があります。この場合、バージョン 5.1 と同様に、JMS 属性はすべて単一のノードにマップされます。

注意： バージョン 6.0 以降では、JMS キューはコンフィグレーション時に定義され、データベース テーブル内には保存されません。メッセージ データと恒久サブスクリプションは、2 つの JDBC テーブルまたはファイル システムのディレクトリに格納されます。

4. 既存の JDBC データベース ストアの自動移植作業を準備します。
   a. 既存の JDBC データベースのバックアップを作成します。
   b. 移植されたコンフィグレーション情報（手順 2 を参照）に、既存の JDBC データベース ストアと全く同じ属性を持つ JDBC データベース ストアがあること、また、そのストアを使用する新しい JMS サーバで、既存の JMS サーバと同じ送り先とその送り先の属性が定義されていることを確認します。
   c. 新しい JDBC データベース ストアが既にある場合、中身が空であることを確認します。

      新しい JDBC データベース ストアが必要に応じて自動移植中に作成されます。
   d. JDBC データベース ストアで必要な量の 2 倍のディスク スペースがシステムにあることを確認します。

      移植中には、既存のデータベース情報と新しいデータベース情報がディスク上に併存するため、2 倍のディスク スペースが必要になります。移植が完了したら、6-13 ページの「JDBC データベース ストアの削除」の説明に従って古い JDBC データベース ストアを削除できます。
5. 必要に応じて既存のコードを更新し、6-2 ページの「5.1 と 6.0 の既存の機能の変更点」で説明されている機能の変更点を反映させます。
6. WebLogic Server を起動すると、既存の JDBC データベース ストアが自動的に移植されます。

注意： 何らかの理由で自動移植が失敗した場合、自動移植は次に WebLogic Server が起動したときに再試行されます。

6.0 アプリケーションの 6.1 への移植手順

WebLogic JMS 6.*x* アプリケーションを使用するには、その前に WebLogic Server 6.0 のコンフィグレーション データとメッセージ データを次の手順で移植する必要があります。

1. バージョン 6.0 での接続ファクトリのコンフィグレーションを確認します。バージョン 6.1 のデフォルト接続ファクトリを呼び出すプログラムを修正して、以下の接続ファクトリのいずれかがロードされるようにする必要がある場合があります。
   - バージョン 6.0 のデフォルト接続ファクトリのいずれか
   - カスタム接続ファクトリ
2. 移植手順を開始する前に、バージョン 6.0 の WebLogic Server を正しくシャットダウンします。

   警告： メッセージの処理中に WebLogic Server の旧バージョンを突然停止すると、移植の際に問題が発生することがあります。旧バージョンのサーバをシャットダウンして WebLogic Server バージョン 6.*x* に移植する前に、処理が非アクティブになっている必要があります。
3. 『インストール ガイド』で説明されているとおりに、WebLogic Server 環境をアップグレードします。
4. 必要に応じて既存のコードを更新し、6-2 ページの「5.1 と 6.0 の既存の機能の変更点」で説明されている機能の変更点を反映させます。

   警告： バージョン 6.1 の WebLogic Server を起動する前に、バージョン 6.0 のストアをバックアップします。これは、バージョン 6.0 のサーバでは 6.1 のストアを使用できないためです。使用すると、データが破損するおそれがあります。
5. バージョン 6.1 の WebLogic Server を起動します。このサーバでは、6.0 以前のストアがそのまま使用されます。

6.x アプリケーションの 7.0 への移植手順

すべての WebLogic JMS 6.*x* アプリケーションがバージョン 7.0 でサポートされています。ただし、可用性の高い新しい JMS 機能をアプリケーションで利用したい場合は、既存の物理的な送り先（キューおよびトピック）を単一の分散送り先の一部として構成する必要があります。

JMS 分散送り先の使用方法の詳細については、『WebLogic JMS プログラマーズ ガイド』の「分散送り先の使用」を参照してください。

JDBC データベース ストアの削除

移植が完了したら、付録 B「JDBC データベース ユーティリティ」の説明に従って、`utils.Schema` ユーティリティで古い JDBC データベース テーブルを削除する必要があります。

移植中に、ローカル作業ディレクトリで DDL ファイルが生成および保存されます。DDL ファイルには、`drop_<jmsServerName>_oldtables.ddl` という名前が付けられます。`<jmsServerName>` は JMS サーバ名を示します。JDBC データベース ストアを削除するには、`utils.Schema` ユーティリティでこの DDL ファイルを引数として指定します。

たとえば、*MyJMSServer* という JMS サーバから古い JDBC データベース ストアを削除するには、次のコマンドを実行します。

```
java utils.Schema jdbc:weblogic:oracle weblogic.jdbc.oci.Driver -s
server -u user1 -p foobar -verbose drop_MyJMSServer_oldtables.ddl
```

`utils.Schema` ユーティリティの詳細については、付録 B「JDBC データベース ユーティリティ」を参照してください。

A コンフィグレーション チェックリスト

以下の節では、さまざまな WebLogic JMS の機能に関するモニタ用チェックリストを示します。

- サーバ クラスタ
- JTA ユーザ トランザクション
- JMS トランザクション
- メッセージの配信
- 非同期メッセージの配信
- 永続的メッセージ
- メッセージの並行処理
- マルチキャスト
- 恒久サブスクリプション
- 送り先のソート順
- 一時的な送り先
- しきい値と割り当て

コンフィグレーション属性の設定については、『管理者ガイド』を参照してください。各コンフィグレーション属性の詳細については、『Administration Console オンライン ヘルプ』を参照してください。

サーバ クラスタ

サーバ クラスタをサポートするには、以下のコンフィグレーションを行います。

- [接続ファクトリ] ノードの [対象] タブで、対象となる WebLogic Server を指定します。
- [サーバ] ノードの [対象] タブで、対象となる WebLogic Server を指定します。

JTA ユーザ トランザクション

JTA ユーザ トランザクションをサポートするには、以下のコンフィグレーションを行います。

- [接続ファクトリ] ノードの [コンフィグレーション | トランザクション] タブにある [ユーザ トランザクションを有効化] チェック ボックスをオンにして、接続ファクトリの JTA ユーザ トランザクション モードを設定します。

JMS トランザクション

JMS トランザクション セッションをサポートするには、以下のコンフィグレーションを行います。

- [接続ファクトリ] ノードの [コンフィグレーション | トランザクション] タブにある [トランザクション タイムアウト] 属性で、接続ファクトリのトランザクション タイムアウト値を設定します。
- [セッション プール] ノードの [コンフィグレーション] タブにある [処理済] チェックボックスをオンにして、セッション プールのトランザクション モードを設定します。

メッセージの配信

メッセージ配信の属性を定義するには、以下のコンフィグレーションを行います。

- [接続ファクトリ] ノードの [コンフィグレーション | 一般] タブで、接続ファクトリの優先順位、存続時間、配信時間、配信モードの属性を設定します。
- [送り先] ノードの [コンフィグレーション | オーバライド] タブで、送り先の優先度、存続時間、配信時間、配信モードのオーバーライドの属性を設定します。
- [送り先] ノードの [コンフィグレーション | 再配信] タブで、送り先の再配信遅延、再配信の制限、エラー送り先の属性を設定します。

注意： 以上の設定は、4-23 ページの「メッセージの送信」で説明されているように、メッセージの送信時または set メソッドの使用時にメッセージ プロデューサで動的に設定することもできます。

送り先のコンフィグレーション属性は他のすべての設定に優先します。

非同期メッセージの配信

非同期セッションの間に存在し、メッセージ リスナにまだ渡されていないメッセージの最大数を定義するには、以下のコンフィグレーションを行います。

- [接続ファクトリ] ノードの [コンフィグレーション | 一般] タブで、[最大メッセージ数] 属性を設定します。

永続的メッセージ

注意： 恒久サブスクリプションがあるトピックでのみ、送り先は永続的になります。恒久サブスクリプションの詳細については、4-57 ページの「恒久サブスクリプションの設定」を参照してください。

永続メッセージングをサポートするには、以下のコンフィグレーションを行います。

- [ストア] ノードでファイルまたは JDBC ストアを作成します。
- [サーバ] ノードの [コンフィグレーション | 一般] タブにある [ストア] を設定して、JMS サーバのバッキング ストアを指定します。

  注意： 2 つの JMS サーバで同じバッキング ストアを使用することはできません。
- 以下の属性のいずれかを [永続] または [非永続] に設定してデフォルト メッセージ配信モードを指定します。
  - [接続ファクトリ] ノードの [コンフィグレーション | 一般] タブにある [デフォルト配信モード] 属性
  - [送り先] ノードの [コンフィグレーション | オーバライド] タブにある [配信モードのオーバライド] 属性

  注意： 4-23 ページの「メッセージの送信」の説明に従って、メッセージ送信の配信モードを永続的に指定できます。

メッセージの並行処理

メッセージの並行処理をサポートするには、以下のコンフィグレーションを行います。

- [セッション プール] ノードの [コンフィグレーション] タブで、サーバ セッション プールの属性を指定します。
- [コンシューマ] ノードの [コンフィグレーション] タブで、接続コンシューマの属性を指定します。

注意： メッセージの並行処理に使用するサーバ セッション プール ファクトリはコンフィグレーションできません。WebLogic JMS は、デフォルトでは `weblogic.jms.ServerSessionPoolFactory:<name>`(<*name*> は、セッション プールが作成される JMS サーバ名) という

ServerSessionPoolFactory オブジェクトを 1 つ定義します。サーバ セッション プール ファクトリの使い方については、4-76 ページの「サーバ セッション プールの定義」を参照してください。

マルチキャスト

注意： マルチキャストはトピックでのみ有効です。

トピックのマルチキャストに対しては、以下のコンフィグレーションを行います。

- [送り先] ノードの [コンフィグレーション | マルチキャスト] タブで、アドレス、ポート、存続時間 (TTL) を設定します。
- [接続ファクトリ] ノードの [コンフィグレーション | 一般] タブにある [最大メッセージ数] 属性で、未処理メッセージの最大数を設定します。
- [接続ファクトリ] ノードの [コンフィグレーション | 一般] タブにある [超過時のポリシー] 属性で、未処理メッセージが [最大メッセージ数] の値に達したときに使用するポリシーを指定します。

恒久サブスクリプション

恒久トピック サブスクリプションをサポートするには、以下に従ってコンフィグレーションしてください。

- [ストア] ノードを使用して、JMS ファイル ストアまたは JMS JDBC 永続ストアを作成します。
- [JMS サーバ|コンフィグレーション|一般] タブの [ストア] 属性のドロップダウン リストで、コンフィグレーションしたストアを選択して JMS サーバに割り当てます。

  注意： 2 つの JMS サーバで同じ永続ストアを使用することはできません。

- 以下に従って、JMS 接続ファクトリまたはトピック接続のいずれかでクライアント ID を設定します。
  - 1 つは、接続ファクトリをクライアント ID でコンフィグレーションする方法です。WebLogic JMS では、クライアント ID ごとに別々の接続ファクトリ インスタンスを作成することになります。作成した個々の接続ファクトリには、[接続ファクトリ | コンフィグレーション | 一般] タブの [クライアント ID] 属性を使用して、恒久サブスクリプションを持つクライアントのユニークなクライアント ID を指定します。アプリケーションでは、JNDI 内でクライアントのトピック接続ファクトリをルックアップし、これらを使用してクライアント ID を含んだ接続を作成します。
  - もう 1 つの「より望ましい」方法は、接続メソッドを呼び出して接続が作成された後に、アプリケーションによってトピック接続にクライアント ID を設定する方法です。詳細については、4-57 ページの「恒久サブスクリプションの設定」を参照してください。この方法を使用すると、デフォルトの接続ファクトリを使用でき (アプリケーションに適合している場合)、コンフィグレーション情報を変更する必要がありません。

送り先のソート順

送り先のソート順をサポートするには、以下のコンフィグレーションを行います。

- [送り先キー] ノードの [コンフィグレーション] タブで、キーの属性を設定します。
- [送り先] ノードの [コンフィグレーション | 一般] タブで [送り先キー] を設定します。

一時的な送り先

一時的な送り先 (キューまたはトピック) をサポートするには、以下をコンフィグレーションします。

- 同じドメインにある JMS サーバ用の JMS テンプレート。[テンプレート] ノードの [コンフィグレーション | 一般] タブを使用します。
- JMS サーバが一時的な送り先として使用する JMS テンプレート。[サーバ] ノードの [コンフィグレーション | 一般] タブにある JMS サーバの [一時的なテンプレート] 属性を使用します。

しきい値と割り当て

しきい値と割り当てに対しては、以下のコンフィグレーションを行います。

- [サーバ] ノードの [コンフィグレーション | しきい値と割当] タブで、メッセージおよびバイトのしきい値と割り当て (最大数、最大しきい値、最小しきい値) を設定します。
- [送り先] ノードの [コンフィグレーション | しきい値と割当] タブで、メッセージおよびバイトのしきい値と割り当て (最大数、最大しきい値、最小しきい値) を設定します。
- [セッション プール] ノードの [コンフィグレーション] タブにある [最大セッション] 属性で、セッション プールから取得可能なセッションの最大数を設定します。
- [コンシューマ] ノードの [コンフィグレーション] タブにある [最大メッセージ数] 属性で、接続コンシューマで蓄積可能なメッセージの最大数を設定します。

B JDBC データベース ユーティリティ

以下の節では、WebLogic JMS の JDBC データベース ストア、および JDBC データベース ユーティリティを使用して既存の JDBC データベース ストアを再生成する方法について説明します。

- 概要
- JMS テーブルについて
- JDBC データベース ストアの再生成

概要

JDBC `utils.Schema` ユーティリティを使用すると、既存のバージョンを削除することで新しい JDBC データベース ストアを再生成できます。JMS はこれらのストアを自動的に作成するので、このユーティリティを実行する必要は通常ありません。しかし、既存の JDBC データベース ストアに障害が発生した場合は、`utils.Schema` ユーティリティを使用して再生成できます。

警告： `utils.Schema` コマンドは、既存のデータベース テーブルをすべて削除して、新しいものを再作成するので、実行する際には注意してください。

JMS テーブルについて

JMS データベースには、自動的に生成され、JMS 内部で使用されるシステム テーブルが 2 つあります。

- `<prefix>JMSStore`
- `<prefix>JMSState`

プレフィックス名は、このバッキング ストア内の JMS テーブルを識別します。ユニークなプレフィックスを指定すると、同一データベース内に複数のストアが存在できます。プレフィックスは、JDBC ストアをコンフィグレーションする際に Administration Console でコンフィグレーションされます。プレフィックスは、以下の場合にテーブルに付加されます。

- DBMS で完全修飾名が必要な場合
- 複数のテーブルを 1 つの DBMS に格納できるようにして、2 つの WebLogic Server の JMS テーブルを区別する必要がある場合

プレフィックスは、JMS テーブル名に付加されたときに有効なテーブル名になるように、次の形式で指定する必要があります。

```
[[catalog.]schema.]prefix
```

注意： データに障害が発生するので、2 つの JMS ストアを同じデータベース テーブルで使用することはできません。

WebLogic JMS の JDBC データベース ストアのコンフィグレーションの詳細は、『管理者ガイド』の「JMS の管理」を参照してください。

JDBC データベース ストアの再生成

`utils.Schema` ユーティリティは Java プログラムで、以下の項目を指定するコマンドライン引数をとります。

- JDBC ドライバ
- データベース接続情報
- データベース テーブルを作成する SQL データ定義言語（DDL）コマンド（セミコロンで終わる）を含むファイルの名前

通常、DDL ファイルには .ddl 拡張子が付いています。DDL ファイルは、Pointbase、Cloudscape、Informix、Sybase、Oracle、MS SQL Server、IBM DB2 および Times Ten データベース用に提供されています。

utils.Schema を実行するには、CLASSPATH に weblogic.jar ファイルを指定する必要があります。

utils.Schema コマンドを次のように入力します。

```
java utils.Schema url JDBC_driver [options] DDL_file
```

次の表に、utils.Schema コマンドライン引数を示します。

引数	説明
url	データベース接続 URL。この値は、JDBC 仕様の定義に従ってコロン区切りの URL にする。
JDBC_driver	JDBC ドライバ クラスの完全パッケージ名。

options	省略可能なコマンド オプション。
	データベースの必要に応じて、以下のものを指定する。
	■ ユーザ名とパスワード。次のように指定する。
	`-u <username> -p <password>`
	■ JDBC データベース サーバのドメイン ネーム サーバ（DNS）名。次のように指定する。
	`-s <dbserver>`
	-verbose オプションも指定可能。このオプションを指定すると、utils.Schema は実行時に SQL コマンドをエコーする。

引数	説明
DDL_file	実行する SQL コマンドが記されたテキスト ファイルの絶対パス名。SQL コマンドは複数行にわたって指定できる。末尾にセミコロン（;）を付ける。ポンド 記号（#）で始まる行はコメント。 `weblogic.jar` ファイル内の `weblogic/jms/ddl` ディレクトリには、Pointbase、Cloudscape、Informix、Sybase、Oracle、MS SQL Server、IBM DB2、および Times Ten データベース用の JMS DLL ファイルがある。別のデータベースを使用するには、このファイルのいずれかをコピーおよび編集する。 JDK で用意されている jar ユーティリティを使用すると、次のコマンドで DDL ファイルを `weblogic/jms/ddl` ディレクトリに抽出できる。 `jar xf weblogic.jar weblogic/jms/ddl` **注意：** `weblogic/jms/ddl` パラメータを省略すると、jar ファイル全体が抽出される。

たとえば、次のコマンドでは、ユーザ名 `user1`、パスワード `foobar` で、`DEMO` という Oracle サーバに JMS テーブルを再作成します。

```
java utils.Schema jdbc:weblogic:oracle:DEMO \
  weblogic.jdbc.oci.Driver -u user1 -p foobar -verbose \
  weblogic/jms/ddl/jms_oracle.ddl
```

WebLogic Server に付属する Pointbase デモ データベースでは、ユーザ名とパスワードは必要ではありません。しかし、次の手順に従って Pointbase サーバで JMS テーブルを作成する必要があります。

1. WLS サンプル環境を設定します。

   ```
   %SAMPLES_HOME%\server\config\examples\setExamplesEnv.cmd
   ```

2. `%WL_HOME%\server\lib\` ディレクトリに移動し、`jms_pointbase.ddl` ファイルを `weblogic.jar` ファイルからカレント ディレクトリに抽出します。

3. 次のコマンドを実行して、JMS テーブルを作成します。

```
java utils.Schema jdbc:pointbase:server://localhost/demo
  com.pointbase.jdbc.jdbcUniversalDriver
  -u examples -p examples -verbose jms_pointbase.ddl
```

Pointbase JDBC URL では、WebLogic JMS サンプルにあるデモ データベースを指定します。このデータベースではサンプルとして JMS テーブルが既に作成されています。

4. Pointbase サーバを起動して Pointbase Console を開きます。

   JMS テーブルのモニタおよび操作のために Pointbase Server console を使用する詳細については、*WL_HOME*\samples\server\src\examples ディレクトリの Pointbase.html ファイルを参照してください。ここで *WL_HOME* は、WebLogic Platform のインストール先の最上位ディレクトリを意味します。

索引

J

S

U

X

あ

い

え

お

か

き

く

こ

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

り

れ

ろ