開発者ガイド

Sun™ ONE Message Queue

Version 3.0

816-6459-10
2002 年 6 月

目 次

図目次

表目次

手順一覧

コード例一覧

はじめに

このマニュアルは、開発者が Sun™ ONE Message Queue (MQ) 環境で、メッセージングアプリケーションを開発するために必要な概念と手順に関する情報を提供します。

ここでは、次の節について説明します。

- マニュアルの対象読者
- マニュアルの構成
- マニュアルの表記規則
- 関連マニュアル

マニュアルの対象読者

このマニュアルは、主として MQ メッセージングシステムを使用してメッセージを交換するアプリケーションの開発者を対象としています。

これらのアプリケーションは、Java Message Service (JMS) アプリケーションプログラミングインタフェース (API) と Java XML Messaging (JAXI) API を使用して、メッセージの作成、送受信、および読み込みを行います。JMS と JAXM の仕様はオープンスタンダードです。

この『開発者ガイド』では、JMS API と JMS プログラミングのガイドラインに精通していることを前提としています。このマニュアルでは、MQ メッセージングシステムの機能と柔軟性を最大限に活かして JMS クライアントアプリケーションを最適化する方法を説明します。

ただし、この『開発者ガイド』は、JAXM API または JAXM プログラミングのガイドラインに精通していることを前提としていません。これに関する資料は第 5 章「SOAP メッセージの操作」で説明します。ただし、XML の基本的な知識があることを前提としています。

マニュアルの構成

このマニュアルは、読者が全編を読み通すように構成されています。各章の内容を次の表で簡単に説明します。

表 1　　マニュアルの内容

章	説明
第 1 章「概要」	Sun ONE Message Queue、JMS の概念、およびプログラミングの問題に関する高レベルな概要
第 2 章「クイックスタートチュートリアル」	簡単なクライアントアプリケーションを例として使用し、MQ 開発環境に関する知識を習得するためのチュートリアル
第 3 章「管理対象オブジェクトの使用」	MQ 管理対象オブジェクトの使用方法 (プロバイダに依存しない方法とプロバイダ固有の方法の両方) の説明
第 4 章「クライアントの最適化」	MQ クライアントランタイムの機能、およびクライアントアプリケーションを最適化するためにこれらの機能を使用する方法の説明
第 5 章「SOAP メッセージの操作」	MQ のサポートの有無に応じた、SOAP メッセージの送受信の方法の説明
付録 A「管理対象オブジェクトの属性」	管理対象オブジェクトの属性に関する概要の説明

マニュアルの表記規則

ここでは、このマニュアルで使用されている表記規則について説明します。

テキストの表記規則

表 2　　マニュアルの表記規則

書体	説明
斜体	可変部分に使用される。斜体で表記された項目や値は適宜置き換える必要がある。強調するマニュアル名や説明の対象となる語句や項目に対しても使用される
モノスペース	コード例、コマンド行に入力するコマンド、ディレクトリ、ファイルまたはパス名、エラーメッセージテキスト、クラス名、メソッド名 (署名の全要素を含む)、パッケージ名、予約語、および URL を表す

表 2　　マニュアルの表記規則 (続き)

書体	説明
[]	コマンド行の構文ステートメントのオプションの値を示す
すべて大文字	ファイルシステムタイプ (GIF、TXT、HTML など)、環境変数 (IMQ_HOME)、または頭文字 (MQ、JSP) を表す
キー + キー	複数のキーストロークはプラス記号で結合する。Ctrl+A は、両方のキーを同時に押すことを表す
キー - キー	連続するキーストロークはハイフンで結合する。Esc-S は、Esc キーを押してから離し、次に S キーを押すことを表す

環境変数の表記規則

MQ では 3 種類の環境変数が使用されますが、その使用方法は、プラットフォームによって異なります。表 3 では、これらの環境変数について説明し、Solaris、Windows、および Linux の各プラットフォームでの使用方法についても概説します。

表 3　　MQ 環境変数

環境変数	説明
IMQ_HOME	ルート MQ インストールディレクトリ。インストールされたファイルはすべてこのディレクトリ内に配置される • Solaris の場合、ルート MQ インストールディレクトリは存在しない。IMQ_HOME が MQ ソフトウェアで使用されることも、Solaris 上のファイルの場所を示すために MQ マニュアルで使用されることもない • Solaris 上の Sun ONE Application Server の Evaluation Edition では、MQ ソフトウェアが IMQ_HOME を使用することはないが、ルート MQ インストールディレクトリ (Application Server インストールの ルートディレクトリの下にある、imq サブディレクトリ) を示すために MQ マニュアルで使用される • Windows の場合、IMQ_HOME は MQ ソフトウェアでも使用され、また ルート MQ インストールディレクトリを示すために MQ マニュアルでも使用される。IMQ_HOME の値は、MQ インストーラにより設定される (デフォルトでは C:¥Program Files¥Sun Microsystems¥Message Queue 3.0) • Linux の場合、IMQ_HOME は MQ ソフトウェアでは使用されないが、ルート MQ インストールディレクトリを示すために、MQ マニュアルで使用される (デフォルトでは /opt の下の imq サブディレクトリ)
IMQ_VARHOME	MQ の一時的または動的に作成された構成ファイルやデータファイルが格納されている、/var ディレクトリを示す • Solaris の場合、IMQ_VARHOME は、デフォルトで /var/imq ディレクトリに設定されるが、ユーザはオプションでこの値を任意のディレクトリに設定できる • Solaris 上の Sun ONE Application Server の Evaluation Edition では、IMQ_VARHOME は、デフォルトで IMQ_HOME/var に設定されるが、ユーザはオプションでこの値を任意のディレクトリに設定できる • Windows の場合、IMQ_VARHOME は、デフォルトで IMQ_HOME/var に設定されるが、ユーザはオプションでこの値を任意のディレクトリに設定できる • Linux の場合、IMQ_VARHOME は、デフォルトで IMQ_HOME/var に設定されるが、ユーザはオプションでこの値を任意のディレクトリに設定できる

表 3　　MQ 環境変数 (続き)

環境変数	説明
IMQ_JAVAHOME	MQ 実行可能ファイルが必要とする Java ランタイム (JRE 1.4) の場所を示す • Solaris の場合、IMQ_JAVAHOME は、デフォルトで /usr/j2se/jre ディレクトリに設定されるが、ユーザはオプションでこの値を JRE 1.4 がインストールされている任意の値に設定できる • Windows の場合、IMQ_JAVAHOME は、デフォルトで IMQ_HOME/jre に設定されるが、ユーザはオプションでこの値を JRE 1.4 がインストールされている任意の値に設定できる • Linux の場合、IMQ_JAVAHOME は、デフォルトで /usr/java/j2sdk1.0/jre ディレクトリに設定されるが、ユーザはオプションでこの値を JRE 1.4 がインストールされてい任意の値に設定できる

このマニュアルで、IMQ_HOME、IMQ_VARHOME、および IMQ_JAVAHOME は、プラットフォーム固有の環境変数の表記法や構文 (UNIX の $IMQ_HOME など) に関係なく示されています。すべてのパス名には、UNIX のファイル区切り文字の表記法 (/) が使用されています。

関連マニュアル

このマニュアル以外にも、MQ には追加のマニュアルリソースが用意されています。

MQ マニュアルセット

表 4 に MQ マニュアルセットに含まれる各マニュアルを、通常使用する順に一覧表示します。

表 4　　MQ マニュアルセット

マニュアル	対象読者	説明
MQ インストールガイド	開発者および管理者	Solaris、Linux、および Windows の各プラットフォームで MQ ソフトウェアをインストールする方法を説明する
リリースノート	開発者および管理者	新機能、制限事項、既知のバグ、および技術的な注意点を説明する
MQ 開発者ガイド	開発者	MQ の JMS 実装に関連するクイックスタートチュートリアルおよびプログラミング情報を提供する
MQ 管理者ガイド	管理者。開発者にも推奨	MQ 管理ツールを使用して管理タスクを実行するために必要な基本情報を提供する

JavaDoc

JavaDoc 形式の JMS および MQ API マニュアルは、次の場所にあります。

`IMQ_HOME/javadoc/index.html`
(Solaris では、`/usr/share/javadoc/imq/index.html`)

このマニュアルは、Netscape または Internet Explorer などの HTML ブラウザで表示できます。このマニュアルには、標準の JMS API マニュアルおよび MQ 管理対象オブジェクト用の MQ 固有の API が含まれており (第 3 章「管理対象オブジェクトの使用」を参照)、メッセージングアプリケーションの開発者にとって有用です。

クライアントアプリケーションの例

複数のアプリケーションの例が次の場所に用意されており、サンプルのクライアントアプリケーションコードを参照できます。

`IMQ_HOME/demo` (Solaris では `/usr/demo/imq`)

このディレクトリと各サブディレクトリの中に配置されている README ファイルを参照してください。

Java Message Service (JMS) 仕様書

JMS 仕様書は、次のサイトにあります。

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

この仕様書には、サンプルのクライアントコードも掲載されています。

Java XML Messaging (JAXM) 仕様書

JAXM 仕様書は、次のサイトにあります。

`http://java.sun.com/xml/downloads/jaxm.html`

この仕様書には、サンプルのクライアントコードも掲載されています。

JMS プログラミングに関する参考文献

JMS API の基礎知識を身につける上で役立つ一般書籍を次に紹介します。

- 『Java Message Service』、Richard Morson-Haefel、David A. Chappell 共著、O'Reilly and Associates, Inc. (Sebastopol, CA)
- 『Professional JMS Programming』、Scott Grant、Michael P. Kovacs、Meeraj Kunnumpurath、Silvano Maffeis、K. Scott Morrison、Gopalan Suresh Raj、Paul Giotta、James McGovern 共著、Wrox Press Inc.、ISBN: 1861004931
- 『Practical Java Message Service』、Tarak Modi 著、Manning Publications、ISBN: 1930110138

第 1 章

概要

この章では、開発者を対象に、Sun ONE Message Queue (MQ)、JMS の概念、プログラミング関連の問題全般について概説します。

Sun ONE Message Queue とは

MQ 製品には、アプリケーション間の通信や、信頼できるメッセージ配信に関する問題の標準的なソリューションが用意されています。MQ は、企業向けメッセージングシステムで、Java Message Service (JMS) のオープンスタンダードである JMS プロバイダを実装します。

Sun ONE Message Queue ソフトウェアを使用して、異なるプラットフォームおよびオペレーティングシステム上で実行されているプロセスを、共通の MQ メッセージサービスに接続し、情報を送受信することができます。アプリケーション開発者は、ネットワークを介したアプリケーションの通信手順という低レベルの問題にとらわれることなく、アプリケーションのビジネスロジックの開発に集中できます。

MQ には、JMS 仕様の必要条件以上の機能があります。たとえば、次のような機能が含まれています。

集中管理 : MQ メッセージサービスと、送信先やセキュリティなど、メッセージングアプリケーション固有の機能を管理するために、コマンド行と GUI の両方のツールが用意されています。

スケーラブルなメッセージサービス : 連携して動作する複数の MQ メッセージサービスコンポーネント (ブローカ) 間 (マルチブローカクラスタ) でロードバランスを実行することにより、増加する JMS クライアント (コンポーネントまたはアプリケーション) の保守が可能になります。

チューニング可能なパフォーマンス : 信頼性がそれほど重要ではない配信を行う場合に、MQ メッセージサービスのパフォーマンスを向上させることができます。

複数のトランスポート： JMS クライアントが、TCP や HTTP など、複数の異なるトランスポートを介し、安全な (SSL) コネクションを使用して互いに通信する機能をサポートします。

JNDI のサポート： ファイルベースおよび LDAP ディレクトリの両方のサービスを、オブジェクトストアおよびユーザリポジトリとしてサポートします。

SOAP メッセージングのサポート： SOAP (シンプルオブジェクトアクセスプロトコル) 仕様に準拠したメッセージである SOAP メッセージの、JMS メッセージング経由の作成と配信をサポートします。SOAP により、ピア間の構造化 XML データを分散環境で交換できます。詳細は、81 ページの第 5 章「SOAP メッセージの操作」を参照してください。

JMS 準拠に関連した問題についてのドキュメントは、『MQ3.0 リリースノート』を参照してください。

製品エディション

Sun ONE Message Queue 製品は、Platform および Enterprise の 2 つのエディションで利用できます。各エディションは、次に説明するように、別々のライセンス機能に対応しています (MQ を一方のエディションから別のエディションにアップグレードするには、『MQ インストールガイド』の解説を参照)。

Platform Edition: このエディションは、Sun の Web サイトから無料でダウンロードできます。また、最新の Sun ONE Application Server プラットフォームにもバンドルされています。Platform Edition には、次のように 2 つのライセンスが付属しています。

- 基本ライセンス。このライセンスには、基本的な JMS サポート (完全な JMS プロバイダ) が用意されていますが、ロードバランス (マルチブローカのメッセージサービス)、HTTP/HTTPS コネクション、安全なコネクションサービス、スケーラブルなコネクション機能、および複数キューの配信ポリシーなど、企業向けの機能は含まれていません。このライセンスには期限が設定されていないため、比較的要求の少ない運用環境で使用できます。
- 90 日間の企業向けトライアルライセンス。このライセンスには、マルチブローカのメッセージサービス、HTTP/HTTPS コネクション、安全なコネクションサービス、スケーラブルなコネクション機能、および複数キューの配信ポリシーに対するサポートなど、基本ライセンスには含まれていない企業向けの機能がすべて含まれています。ただし、ソフトウェアに設定されたライセンスの有効期間は 90 日間です。そのため、Enterprise Edition の製品 (25 ページの「Enterprise Edition」を参照) で利用できる企業向けの機能を評価するのに最適です。

Platform Edition では、各 MQ メッセージサービスでサポートされる JMS クライアントのコネクション数に制限はありません。基本ライセンスから企業向けライセンスへの切り替えについては、『MQ 管理者ガイド』の `license` コマンド行オプションの説明を参照してください。

Enterprise Edition: このエディションは、運用環境でメッセージングアプリケーションを配置および実行するために使用されます。また Enterprise Edition は、メッセージングアプリケーションやコンポーネントの開発、デバッグ、負荷テストにも使用できます。Enterprise Edition のライセンスには有効期限がなく、マルチブローカのメッセージサービスにおけるブローカ数や、各ブローカにサポートされるクライアントコネクション数にも制限がありません。ただし、Enterprise Edition がサポートする CPU の数はライセンスで指定されています。

注 すべての MQ エディションで、製品の一部であるクライアントランタイムは商用目的での再配布が許可されています。ただし、製品のそれ以外のファイルの再配布は一切許可されていません。ライセンスを取得したあと、再配布可能な製品の一部を使用して、MQ メッセージサーバに接続できる JMS クライアントを開発し、MQ ライセンス料を支払わずにサードパーティ向けに販売することができます。サードパーティは、MQ の自社用バージョンを購入して MQ メッセージサーバにアクセスするか、または MQ メッセージサーバをインストールし実行している別の組織にコネクションする必要があります。

MQ メッセージングシステムのアーキテクチャ

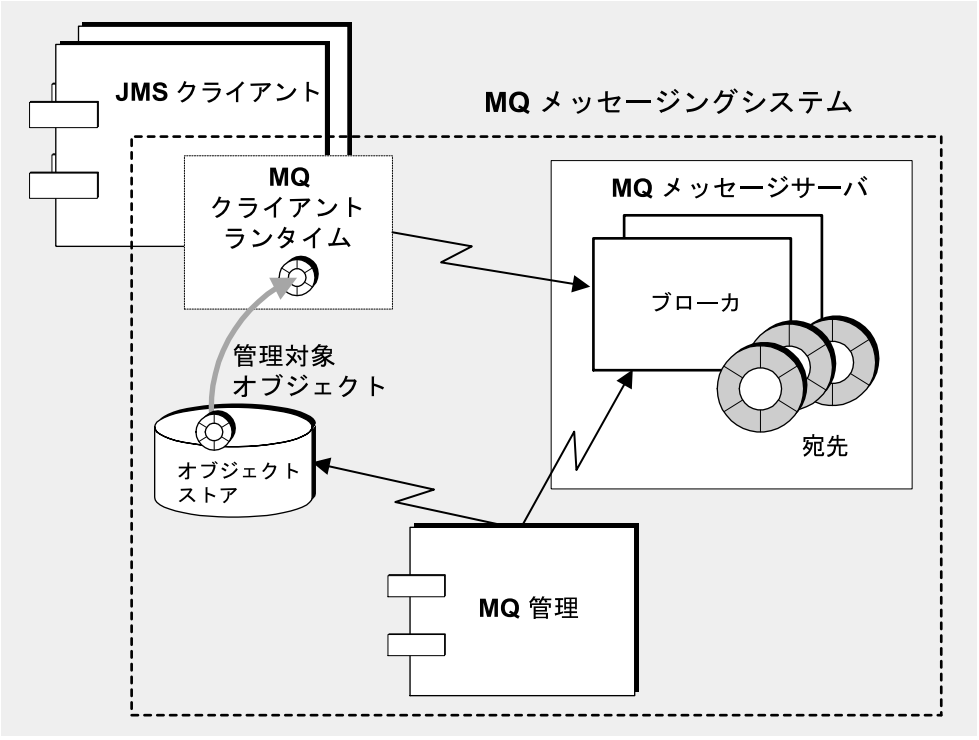
この節では、MQ メッセージングシステムの主要部分について簡単に説明します。開発者は、メッセージングシステムのあらゆる部分、またはそれらの相互作用を完全に把握しておく必要はありません。基本アーキテクチャをしっかりと把握していれば、JMS クライアントの設計および開発に影響するシステムの機能を容易に理解できます。

MQ メッセージングシステムの主要部分は次のとおりです。これらの具体的な関連性については、図 1-1 を参照してください。

MQ メッセージサーバ : MQ メッセージサーバは、メッセージングシステムの中核として機能します。メッセージサーバは、システムに配信サービスを提供する 1 つまたは複数のブローカで構成されます。配信サービスには、JMS クライアントへのコネクション、メッセージの転送と配信、持続性、セキュリティ、およびロギングなどのサービスがあります。メッセージサーバは、クライアントのメッセージ送信先であり、コンシューミングクライアントへのメッセージ配信元である、物理的な送信先を管理します。MQ メッセージサーバの詳細は、『MQ 管理者ガイド』を参照してください。

MQ クライアントランタイム： MQ クライアントランタイムは、JMS クライアントに MQ メッセージサーバへのインタフェースを提供します。つまり、クライアントランタイムによって、27 ページの「JMS プログラミングモデル」に記載されるすべての JMS プログラミングオブジェクトがクライアントに提供されます。クライアントランタイムでは、送信先にメッセージを送信し、送信先からメッセージを受信する場合に、クライアントに必要なすべての処理がサポートされます。MQ クライアントランタイムの詳細は、第 4 章「クライアントの最適化」を参照してください。

図 1-1　MQ システムアーキテクチャ



MQ 管理対象オブジェクト： 管理対象オブジェクトは、プロバイダ固有の実装および構成情報を JMS クライアントが使用するオブジェクト内にカプセル化します。通常、管理対象オブジェクトは、管理者によって構成され、ネームサービスに格納されます。クライアントは、JNDI 検索コードを使ってこうした管理対象オブジェクトにアクセスし、プロバイダに依存しない方式でこれらを使用します。クライアントが管理対象オブジェクトをインスタンス化することもできますが、その場合は、プロバイダ固有の方式で使用されます。MQ クライアントランタイムの構成は、管理対象オブジェクト属性によって行われます。詳細は、第 4 章「クライアントの最適化」を参照してください。

MQ 管理 : MQ には、MQ メッセージングシステムの管理に使用する多数の管理ツールが用意されています。これらのツールを使って、メッセージサーバの管理、管理対象オブジェクトの作成および格納、セキュリティ管理、メッセージアプリケーションリソースの管理、持続データの管理が可能です。通常、これらのツールは MQ 管理者によって使用されます。詳細は、『MQ 管理者ガイド』を参照してください。

JMS プログラミングモデル

この節では、JMS 仕様のプログラミングモデルについて簡単に説明します。ここでは、JMS クライアントのプログラミングで使用する重要な概念および用語を確認します。

JMS プログラミングインタフェース

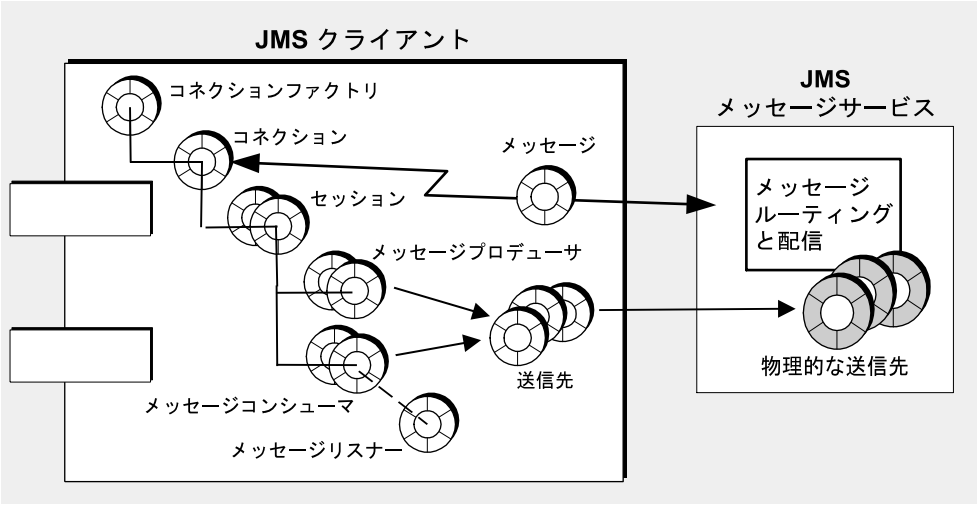
JMS プログラミングモデルでは、JMS クライアント (コンポーネントまたはアプリケーション) は JMS アプリケーションプログラミングインタフェース (API) を使ってメッセージの送受信を行います。この節では、JMS API を実装し、メッセージ配信のための JMS クライアントの設定に使用されるオブジェクトを紹介します (32 ページの「JMS クライアントの設定方法」を参照)。主なインタフェースオブジェクトは、図 1-2 に示されています。次に、各オブジェクトについて説明します。

メッセージ

MQ 製品では、JMS 仕様に準拠した JMS メッセージを使ってデータ交換を行います。JMS の仕様に従い、メッセージはヘッダー、プロパティ、本体で構成されています。

プロパティはオプションです。クライアントがメッセージのフィルタリングに使用する値を提供します。本体もオプションです。交換する実際のデータです。

図 1-2　JMS プログラミングオブジェクト



ヘッダー

すべてのメッセージにはヘッダーが必要です。ヘッダーフィールドには、メッセージのルーティングや識別に使用する値が含まれています。

ヘッダーフィールドの一部の値は、メッセージの作成および配信処理の間に MQ によって自動的に設定されます。一部の値は、クライアントにメッセージプロデューサを作成するときに指定する、メッセージプロデューサの設定に依存します。また、その他の値は JMS API を使用するユーザによってメッセージごとにメッセージ上に設定されます。次の表に、JMS 定義 (必須) のヘッダーフィールドとその設定方法を示します。

表 1-1　JMS 定義のメッセージヘッダー

ヘッダーフィールド	設定者	デフォルト値
JMSDestination	クライアント、メッセージプロデューサまたはメッセージごとに設定	
JMSDeliveryMode	クライアント、メッセージプロデューサまたはメッセージごとに設定	Persistent
JMSExpiration	クライアント、メッセージプロデューサまたはメッセージごとに設定	生存期間 0 (有効期限なし)
JMSPriority	クライアント、メッセージプロデューサまたはメッセージごとに設定	4 (標準)
JMSMessageID	プロバイダ、自動設定	

表 1-1　JMS 定義のメッセージヘッダー (続き)

ヘッダーフィールド	設定者	デフォルト値
JMSTimestamp	プロバイダ、自動設定	
JMSRedelivered	プロバイダ、自動設定	
JMSCorrelationID	クライアント、メッセージごとに設定	
JMSReplyTo	クライアント、メッセージごとに設定	
JMSType	クライアント、メッセージごとに設定	

プロパティ

2 つのプロセス間でデータをやりとりするとき、ペイロードデータ以外の情報も同時に送信できます。このような記述的フィールドまたはプロパティにより、データの作成プロセス、作成時刻、データの各部分の構造を一意に識別する情報など、データに関する追加情報を提供することができます。プロパティ (ヘッダーの拡張部分とも考えられる) は、JMS クライアントが指定したプロパティ名とプロパティ値のペアで構成されます。

コンシューミングクライアントは、特定の送信先に配信対象を登録することにより、選択基準として特定のプロパティ値を指定して、選択内容を細かく調整できます。たとえば、クライアントが施設管理に関するメッセージは除いて、給与に関するメッセージ、とくに New Jersey の非常勤社員の給与明細だけを配信対象として指定することができます。指定した基準を満たしていないメッセージは、コンシューマに配信されません。

メッセージ本体のタイプ

JMS には、JMS プロバイダがサポートする必要のある 6 つのメッセージのクラス (タイプ) が指定されています。次の表でこれらのタイプについて説明します。

表 1-2　メッセージ本体のタイプ

タイプ	説明
Message	メッセージ本体を持たないメッセージ
StreamMessage	本体に Java プリミティブ値のストリームを含むメッセージ。いっぱいになると順番に読み込まれる
MapMessage	本体に一連の名前と値のペアを含むメッセージ。エントリの順番は定義されていない
TextMessage	本体に Java 文字列 (XML メッセージなど) を含むメッセージ
ObjectMessage	本体に直列化された Java オブジェクトを含むメッセージ

表 1-2　メッセージ本体のタイプ (続き)

タイプ	説明
BytesMessage	本体に未解釈のバイトストリームを含むメッセージ

送信先

Destination は、JMS メッセージサービスの物理的な送信先を示す JMS 管理対象オブジェクトです (31 ページの「管理対象オブジェクト」を参照)。物理的な送信先は、JMS メッセージサービスエンティティの 1 つです。プロデューサはこの送信先にメッセージを送信し、コンシューマはこの送信先からメッセージを受信します。メッセージサービスは、物理的な送信先に送信されるメッセージのルーティングおよび配信の機能を提供します。Destination 管理対象オブジェクトは、物理的な送信先のプロバイダ固有の命名規則をカプセル化します。これにより、JMS クライアントがプロバイダに依存しなくなります。

コネクションファクトリ

ConnectionFactory は、プロバイダ固有のコネクション設定情報をカプセル化する JMS 管理対象オブジェクトです (31 ページの「管理対象オブジェクト」を参照)。クライアントは、このオブジェクトを使って、メッセージ配信に使用するコネクションを作成します。JMS 管理対象オブジェクトは、Java Naming and Directory Service (UNDI) 検索によって取得できますが、プロバイダ固有のクラスを使って直接インスタンス化することもできます。

コネクション

Connection は、JMS クライアントから JMS メッセージサービスへのアクティブなコネクションを表します。通信リソースの割り当てとクライアント認証の両方が、コネクションの作成時に行われます。このため、このオブジェクトは比較的重くなり、多くのクライアントはメッセージングを 1 つのコネクションだけで行います。コネクションはセッションの確立に使用されます。

セッション

Session は、メッセージのプロデュースとコンシュームのためのシングルスレッドコンテキストです。セッションを使用できるスレッドの数に制限はありませんが、同じセッションを複数のスレッドで同時に使用することはできません。このコンテキストは、メッセージの送受信を行うメッセージプロデューサとメッセージコンシューマの作成に使用され、配信するメッセージの順番を定義します。セッションは、多数の通知オプションまたはトランザクションを使って、信頼性の高い配信処理をサポートします。処理済みセッションは、一連の順次処理を、複数のプロデューサとコンシューマにまたがる単一のトランザクションに結合します。

メッセージプロデューサ

クライアントは、MessageProducer を使って、物理的な送信先へメッセージを送信します。通常、MessageProducer オブジェクトは、Destination 管理対象オブジェクトをメッセージプロデューサを作成するセッションのメソッドに渡すことによって作成されます。なお、特定の送信先を参照しないメッセージプロデューサを作成する場合は、作成するメッセージごとに送信先を指定する必要があります。クライアントは、メッセージプロデューサのデフォルトの配信モード、優先度、生存期間を指定して、これらの情報が明示的にオーバーライドされないかぎり、プロデューサから送信されるすべてのメッセージを制御します。

メッセージコンシューマ

クライアントは、MessageConsumer を使って、物理的な送信先からメッセージを受信します。通常、このオブジェクトは、Destination 管理対象オブジェクトをメッセージコンシューマを作成するセッションのメソッドに渡すことによって作成されます。メッセージコンシューマは、メッセージセレクタを使用できます。このメッセージセレクタを使用すると、メッセージサービスで、選択条件に一致するメッセージだけをメッセージコンシューマに送信できます。メッセージコンシューマは、同期または非同期のどちらかのメッセージのコンシュームをサポートしています (40 ページの「メッセージのコンシューム : 同期と非同期」を参照)。

メッセージリスナー

JMS クライアントは、MessageListener オブジェクトを使って、非同期にメッセージをコンシュームします。メッセージリスナーはメッセージコンシューマに登録されます。セッションスレッドが MessageListener オブジェクトの onMessage() メソッドを呼び出すと、クライアントはメッセージをコンシュームします。

管理対象オブジェクト

27 ページの「JMS プログラミングモデル」で説明している 2 つのオブジェクトは、JMS プロバイダがどのように JMS メッセージサービスを実行するかに依存します。コネクションファクトリオブジェクトは、基礎となっているプロトコルや、プロバイダがメッセージの送信に使用するメカニズムに依存し、送信先オブジェクトは、特定の命名規則や、プロバイダが使用する物理的送信先に依存します。

通常、これらプロバイダ固有の特性により、JMS クライアントコードは特定の JMS 実装に依存します。ただし、JMS クライアントコードがプロバイダに依存しないようにするには、JMS の仕様で、プロバイダ固有の実装と設定情報を管理対象オブジェクトと呼ばれるものにカプセル化する必要があります。これらのオブジェクトには、標準的な、プロバイダ固有ではない方法でアクセスできます。

管理対象オブジェクトは管理者によって作成、構成され、ネーミングサービスに格納されると、クライアントが標準的な Java Naming and Directory Service (JNDI) 検索コードを介して JMS アクセスします。この方法で管理対象オブジェクトを使用すると、JMS クライアントコードがプロバイダに依存しなくなります。

JMS には、コネクションファクトリと送信先の 2 種類の一般的な管理対象オブジェクトがあります。どちらもプロバイダ固有情報をカプセル化しますが、JMS クライアント内では、それぞれまったく違った方法で使用されます。コネクションファクトリは、メッセージサーバへのコネクションを確立するのに使用されますが、送信先オブジェクトは、JMS メッセージサービスに使用される物理的送信先を識別するのに使用されます。

管理対象オブジェクトの詳細については、第 3 章「管理対象オブジェクトの使用」を参照してください。

JMS クライアントの設定方法

JMS プログラミングモデルには、メッセージをプロデュースまたはコンシュームする JMS クライアントの設定に関する一般的な方法があります。この方法は、前の節で説明した JMS プログラミングインタフェースオブジェクトを使用します。

メッセージのプロデュースとコンシュームに関する一般的な手順は次のとおりです。メッセージのプロデュースとコンシュームの両方を行うクライアントの場合、多くの共通手順は複製する必要がありません。

➤ メッセージをプロデュースする JMS クライアントを設定するには

1. JNDI を使って `ConnectionFactory` オブジェクトを検索します。`ConnectionFactory` オブジェクトを直接インスタンス化し、属性値を設定することもできます。
2. `ConnectionFactory` オブジェクトを使って `Connection` オブジェクトを作成します。
3. `Connection` オブジェクトを使って 1 つまたは複数の `Session` オブジェクトを作成します。
4. JNDI を使って 1 つまたは複数の `Destination` オブジェクトを検索します。`Destination` オブジェクトを直接インスタンス化し、名前属性を設定することもできます。
5. `Session` オブジェクトと `Destination` オブジェクトを使って、必要な `MessageProducer` オブジェクトを作成します。`MessageProducer` は `Destination` オブジェクトを指定しなくても作成できますが、この場合はプロデュースするメッセージごとに `Destination` オブジェクトを指定する必要があります。

これで、クライアントがメッセージをプロデュースするために必要な基本的なセットアップが完了しました。

➤ メッセージをコンシュームする JMS クライアントをセットアップするには

1. JNDI を使って ConnectionFactory オブジェクトを検索します。ConnectionFactory オブジェクトを直接インスタンス化し、属性値を設定することもできます。
2. ConnectionFactory オブジェクトを使って Connection オブジェクトを作成します。
3. Connection オブジェクトを使って 1 つまたは複数の Session オブジェクトを作成します。
4. JNDI を使って 1 つまたは複数の Destination オブジェクトを検索します。Destination オブジェクトを直接インスタンス化し、名前属性を設定することもできます。
5. Session オブジェクトと Destination オブジェクトを使って、必要な MessageConsumer オブジェクトを作成します。
6. 必要に応じて、MessageListener オブジェクトをインスタンス化し、MessageConsumer オブジェクトに登録します。
7. Connection オブジェクトにメッセージ配信を開始するように指示します。これで、メッセージがクライアントに配信され、コンシュームされます。

これで、クライアントがメッセージをコンシュームするために必要な基本的なセットアップが完了しました。

JMS クライアントの設計に関する問題

この節では、JMS クライアントの設計に影響を及ぼすさまざまな JMS メッセージングの問題について説明します。

プログラミングドメイン

JMS では、ポイントツーポイント、およびパブリッシュ / サブスクライブの 2 つの別個のメッセージ配信モデルをサポートしています。

ポイントツーポイント (キュー送信先): メッセージは、1 つのプロデューサから 1 つのコンシューマに配信されます。この配信モデルでは、送信先がキューとなります。メッセージは最初にキュー送信先に配信され、次にキューの配信ポリシー (『MQ 管理者ガイド』の第 2 章「メッセージングシステム」を参照) に従って 1 回に 1 つずつ、キューに登録されたコンシューマの 1 つへキューから配信されます。キュー送信先に複数のプロデューサがメッセージを送信してもかまいません。しかし、個々のメッセージは、単一のコンシューマに向けて配信され、コンシュームされることになっています。キュー送信先にコンシューマが 1 つも登録されていない場合、キューは受信したメッセージを保持し、コンシューマがキューを登録したときにそのメッセージを配信します。

パブリッシュ / サブスクライブ (トピック送信先): メッセージは、1 つのプロデューサから複数のコンシューマに配信されます。この配信モデルでは、トピックが送信先になります。メッセージは最初にトピック送信先に配信され、次にトピックに加入した、すべてのアクティブなコンシューマに配信されます。トピック送信先へのメッセージの送信は、複数のプロデューサで実行でき、メッセージごとに加入済みの複数のコンシューマに配信されます。トピック送信先は、永続的サブスクリプションの概念もサポートします。永続的なサブスクリプションでは、トピック送信先とともに登録されたコンシューマを表していますが、メッセージ配信時にアクティブでないことがあります。その後コンシューマがアクティブになると、メッセージを受信します。トピック送信先に登録されたコンシューマがない場合、アクティブでないコンシューマの永続的サブスクリプションがない限り、トピックは受信したメッセージを保持しません。

この 2 つのメッセージ配信モデルは、わずかに異なるセマンティクスを持った別々の API オブジェクトを使用して処理され、表 1-3 にあるように異なるプログラミングドメインを表しています。

表 1-3　JMS プログラミングオブジェクト

基本タイプ (統一ドメイン)	ポイントツーポイントドメイン	パブリッシュ / サブスクライブドメイン
Destination (Queue または Topic)*	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

* プログラミングの手法によっては、特定の送信先タイプを指定する場合がある

JMS 1.1 仕様に準拠する統一ドメインオブジェクトを使用して、ポイントツーポイントとパブリッシュ / サブスクライブのどちらのメッセージングもプログラミングできます (表 1-3 の最初のカラムを参照)。JMS 1.1 仕様は、JMS 1.02 と比較してより簡単な JMS クライアントプログラミングの方法を提供します。特に、JMS クライアントは、同じコネクション上でまた同じセッション内でポイントツーポイントとパブリッシュ / サブスクライブのどちらのメッセージングも実行できます。また、同じトランザクションにキューとトピックの両方を含めることができます。

つまり、JMS クライアントの開発者は、より簡単な JMS 1.1 統一ドメイン手法を選択することにより、JMS 1.0.2 のポイントツーポイントとパブリッシュ / サブスクライブプログラミングドメインを個別に選択する必要がありません。この方法をお勧めしますが、JMS 1.1 仕様は個別の JMS 1.02 プログラミングドメインも引き続きサポートします。事実、MQ 製品に含まれるサンプルアプリケーションとこのガイドで提供されるサンプルコードは、すべて個別の JMS 1.02 プログラミングドメインを使用しています。

注　Sun ONE Application Server 環境で実行するアプリケーションの開発者は、JMS 1.0.2 API を使用するように制限されています。これは、Sun ONE Application Server が、JMS 1.0.2 だけをサポートする J2EE 1.3 仕様に準拠しているからです。これは、サーブレットと メッセージ駆動 Bean (42 ページの「メッセージ駆動型 Beans」を参照) を含む EJB で実行される JMS メッセージングが、ドメイン固有の JMS API に基づく必要があることを意味します。

JMS プロバイダへの非依存性

JMS は管理対象オブジェクト (31 ページの「管理対象オブジェクト」を参照) の使用を指定し、ほかの JMS プロバイダに移植可能な JMS クライアントの開発をサポートします。管理対象オブジェクトにより、クライアントはプロバイダ固有オブジェクトを検索および参照する論理名を使用できます。この方法では、クライアントコードは特定の命名構文やアドレス指定構文、またはプロバイダによって使用される設定可能なプロパティを知る必要はありません。これにより、クライアントコードはプロバイダに依存しなくなります。

管理対象オブジェクトは、MQ 管理者により作成および構成される MQ システムオブジェクトです。これらのオブジェクトは、JNDI ディレクトリサービスに配置され、JMS クライアントが JNDI 検索を使用してアクセスします。

また、MQ 管理対象オブジェクトは、JNDI ディレクトリサービスで検索されず、クライアントによってインスタンス化されます。この管理対象オブジェクトには、アプリケーション開発者がプロバイダ固有の API を使用する必要があるという欠点があります。また、MQ 管理者が MQ メッセージサーバを問題なく制御、管理する機能を損なうことにもなります。

管理対象オブジェクトの詳細については、第 3 章「管理対象オブジェクトの使用」を参照してください。

クライアント識別子

JMS プロバイダは、クライアントの代わりにメッセージサービス保持している状態の情報を使用して、JMS クライアントのコネクションをメッセージサービスに関連付けるクライアント識別子の概念をサポートする必要があります。本来、クライアント識別子は固有のもので、一度に 1 人のユーザだけに適用されます。クライアント識別子は永続的サブスクリプション名 (34 ページの「パブリッシュ / サブスクライブ (トピック送信先)」を参照) と組み合わせて使用され、それぞれの永続的サブスクリプションが 1 人のユーザだけに対応するようにします。

JMS 仕様では、クライアント識別子は、API メソッド経由でクライアントにより設定されますが、コネクションファクトリ管理対象オブジェクト (31 ページの「管理対象オブジェクト」を参照) を使用して管理者が設定することをお勧めします。ただし、コネクションファクトリに物理的に組み込まれている場合、各ユーザは個々のコネクションファクトリに固有の ID を持たせる必要があります。

MQ は、クライアント識別子を ConnectionFactory にすることも、また `ConnectionFactory` オブジェクトで設定できる特殊な変数置換構文を使用したユーザ固有のものにすることもどちらの方法も用意されています (72 ページの「クライアントの識別」を参照)。この方法を使用すると、命名の重複やセキュリティの問題を

気にせずに、永続的サブスクリプションを作成する複数のユーザが、単一の ConnectionFactory オブジェクトを使用できます。したがって、ユーザの永続的サブスクリプションは、誤って消去されることも、別のユーザが間違ったクライアント識別子を設定したために使用できなくなることもありません。

配置済みアプリケーションの場合、クライアント識別子は、JMS API を使用してクライアントがプログラム上設定するか、またはでクライアントが使用する ConnectionFactory オブジェクト内に管理上設定する必要があります。

どのような場合でも、クライアント識別子は、永続的サブスクリプションを作成するために、JMS API を使用してクライアントがプログラム上設定するか、またはクライアントが使用する ConnectionFactory オブジェクト内に管理上設定する必要があります。

信頼性のあるメッセージング

JMS は、次の 2 つの配信モードを定義します。

持続性メッセージ : 持続性メッセージは、必ず 1 回は配信され確実にコンシュームされることが保証されています。持続性のメッセージにとって、信頼性は大変重要です。

持続性のないメッセージ : 持続性のないメッセージは、1 回の配信だけが保証されています。持続性のないメッセージにとって、信頼性はそれほど重要な問題ではありません。

持続メッセージの場合、信頼性の保証には 2 つの側面があります。1 つは、メッセージサービスへの配信とメッセージサービスからの配信が問題なく行われるようにすることです。もう 1 つは、コンシューマに配信される前にメッセージサービスが持続メッセージを失うことがないようにすることです。

通知 / トランザクション

信頼性のあるメッセージングは、送信先からの持続メッセージや送信先への持続メッセージが確実に配信されることに依存します。通知またはトランザクションの MQ セッションによってサポートされている 2 つの一般メカニズムのどちらかを使用すると、信頼性のあるメッセージングが実現します。トランザクションの場合、分散トランザクションマネージャに制御されている状態では、ローカルまたは分散のどちらかになる可能性があります。

通知

信頼性の高い配信を確実にするため、通知を使用するようにセッションを構成できます。

プロデューサの場合、プロデューサの send() メソッドが返される前に、メッセージサービスにより送信先にコネクションメッセージの配信が通知されることになります。コンシューマの場合は、メッセージサービスがメッセージを削除する前に、送信先からのコネクションメッセージの配信とコンシュームが、クライアントにより通知されることになります。

ローカルトランザクション

セッションを処理済として設定することもできます。ここでは、1 つ以上のメッセージのプロデュースおよびコンシュームが、トランザクションという極小の単位にグループ化されます。JMS API には、トランザクションを起動、確定、およびロールバックするメソッドが用意されています。

メッセージがトランザクション内でプロデュースまたはコンシュームされるに従って、ブローカがさまざまな送受信を追跡し、クライアントが呼び出しを実行してトランザクションを確定したときにだけ、送受信の操作を完了させます。トランザクション内での特定の送信や受信の操作が失敗すると、例外が発生します。クライアントコードは、これを無視するか、操作を試行し直すか、またはトランザクション全体をロールバックして、例外を処理できます。トランザクションが確定して、すべての操作が正常に完了したことになります。トランザクションがロールバックされると、正常に行われたすべての操作が取り消されます。

ローカルトランザクションの範囲は、常に単一セッションです。つまり、単一セッションのコンテキストで実行された、1 つ以上のプロデューサまたはコンシューマの操作は、単一のローカルトランザクションにグループ化されます。

トランザクションは単一セッションだけに及ぶため、メッセージのプロデュースとコンシュームの両方を含む終端間トランザクションを持つことはできません。言い換えると、送信先へのメッセージの配信と、それに続くクライアントへのメッセージの配信は、単一トランザクションには置くことはできません。

分散トランザクション

MQ では、分散トランザクションもサポートしています。つまり、メッセージのプロデュースとコンシュームは、データベースシステムなど、ほかのリソースマネージャに関連した操作を含む大容量の分散トランザクションの一部となります。分散トランザクションでは、Java Transaction API (JTA)、*XA Resource* API の仕様で定義された 2 階層コミットプロトコルを使用して、メッセージサービスやデータベースマネージャといった複数のリソースマネージャによって実行される操作を、分散トランザクションマネージャが追跡および管理します。Java の世界では、リソースマネージャと分散トランザクションマネージャ間の対話は、JTA の仕様で記述されます。

分散トランザクションをサポートするということは、メッセージングクライアントが、JTA で定義される XAResource インタフェースを介して分散トランザクションに加わることができるということです。このインタフェースでは、2 階層コミットを実装するための、数多くのメソッドが定義されます。API の呼び出しがクライアント側で行われている間、MQ ブローカは分散トランザクション内のさまざまな送受信操作やトランザクションの状態を追跡し、Java Transaction Service (JTS) で提供される分散トランザクションマネージャと一致したときにだけ、メッセージング操作を完了します。

ローカルトランザクションに関しては、無視したり、操作を試行し直したり、分散トランザクション全体をロールバックしたりして、クライアントは例外を処理できます。

MQ では、XA コネクションファクトリを介した分散トランザクションのサポートが実装されています。この XA コネクションファクトリでは、次に XA セッション (27 ページの「JMS プログラミングモデル」を参照) を作成する XA コネクションを確立できます。さらに、分散トランザクションへのサポートには、サードパーティの JTS、または JTS を提供する J2EE 準拠の Application Server のいずれかが必要です。

持続ストレージ

信頼性のもう一つの重要な側面は、1 度持続メッセージが送信先に配信されると、そのメッセージがコンシューマに配信されるまで、メッセージサービスがメッセージを失わないようにすることです。つまり、送信先への持続メッセージの配信では、メッセージサービスは持続メッセージを持続データストアに配置する必要があります。何かの理由でメッセージサービスが停止した場合、持続データ格納ではメッセージが修復され、適切なコンシューマに配信されます。これにより、メッセージ配信にオーバーヘッドが発生しますが、信頼性も向上します。

メッセージサービスは、永続的サブスクリプションも格納する必要があります。これは、トピック送信先の場合の配信を保証するためで、コネクションメッセージを修復するだけでは十分ではないからです。メッセージサービスでは、トピックの永続的サブスクリプションに関する情報も修復する必要があり、そうでない場合、アクティブになったときに永続サブスクライバにメッセージを配信できません。

コネクションメッセージの配信を保証する必要があるメッセージングアプリケーションは、キュー送信先を使用するか、トピック送信先に対して永続的サブスクリプションを使用する必要があります。

パフォーマンスのかね合い

メッセージ配信の信頼性が高くなればなるほど、その信頼性を実現するために、より多くのオーバーヘッドや帯域幅が必要となります。信頼性とパフォーマンスの兼ね合いは、設計上考慮するべき重要な点です。持続性のないメッセージをプロデュースおよびコンシュームするように設定すると、最大のパフォーマンスとスループットが得られます。これに対して、処理済みセッションを使用するトランザクションで、持続的メッセージをプロデュースおよびコンシュームすれば、最大の信頼性が得られます。この 2 つの要素には複数のオプションがあり、どちらを優先させるかは MQ 固有の持続性や通知プロパティの使用をはじめ、アプリケーションの必要性によって異なります (79 ページの「パフォーマンス要因」を参照)。

メッセージのコンシューム : 同期と非同期

JMS クライアントがメッセージをコンシュームする方法には、同期または非同期のいずれか 2 つの方法があります。

同期コンシュームの場合、クライアントは、`MessageConsumer` オブジェクトの `receive()` メソッドを呼び出してメッセージを取得します。クライアントスレッドは、メソッドが復帰するまでブロックされます。これは、使用可能なメッセージが存在しない場合、メッセージが使用可能になるまでクライアントがブロックされるか、または `receive()` メソッドがタイムアウトするまで (タイムアウトを指定して呼び出された場合) クライアントがブロックされることを意味します。このモデルでは、メッセージは、クライアントスレッドによって 1 つずつコンシュームされます (同期方式)。

非同期コンシュームの場合、クライアントは、メッセージコンシューマに `MessageListener` オブジェクトを登録します。メッセージリスナーはコールバックオブジェクトのように機能します。セッションが `MessageListener` オブジェクトの `onMessage()` メソッドを呼び出すと、クライアントはメッセージをコンシュームします。このモデルでは、クライアントスレッドはブロックされず、メッセージが非同期でコンシュームされます。これは、メッセージを待機し、コンシュームするスレッドが、MQ クライアントランタイムに属しているためです。

メッセージの選択

JMS には、メッセージセレクタに設定された条件に基づくフィルタや転送を、メッセージサービスが実行できるようにするメカニズムが用意されています。プロデューシングクライアントは、アプリケーション固有のプロパティをメッセージに設定できます。コンシューミングクライアントは、設定されたプロパティに基づく選択条件を使用して、メッセージの項目を示すことができます。これにより、クライアントの作業が単純になり、不要なクライアントへの配信メッセージのオーバーヘッドがなくなります。ただし、選択条件を処理しているメッセージサービスに、一部のオーバーヘッドが追加されます。メッセージセレクタの構文とセマンティクスは、JMS の仕様書で解説されています。

メッセージの順番と優先度

一般には、単一のセッションにより送信先に設定されたすべてのメッセージは、送信された順にコンシューマへ配信されるようになっています。ただし、別々のプロパティが割り当てられている場合、メッセージングサービスは、優先度が高いほうのメッセージを先に配信しようとします。

それ以外の場合、クライアントにコンシュームされるメッセージの順番は、プロデュースされた順番と密接な関係はありません。これは、送信先へのメッセージ配信と、その送信先からのメッセージ配信が、メッセージの送信順、メッセージの送信元となるセッション、メッセージが持続的かどうか、メッセージの生存期間、メッセージの優先度、キュー送信先のメッセージ配信ポリシー (『MQ 管理者ガイド』を参照) 、およびメッセージサービスの可用性といった、タイミングに影響を与える多くの問題点に依存するためです。

JMS/J2EE プログラミング : メッセージ駆動型 Beans

27 ページの「JMS プログラミングモデル」で説明した一般的な JMS クライアントプログラミングモデルのほかに、さらに JMS に特化したプログラミングモデルがあり、Java 2 Enterprise Edition (J2EE) アプリケーションのコンテキストで使用されます。この特殊な JMS クライアントは、メッセージ駆動型 Beans メッセージ駆動型 Beans と呼ばれ、EJB 2.0 Specification (`http://java.sun.com/products/ejb/docs.html`) で指定されている Enterprise JavaBeans (EJB) コンポーネントのシリーズの 1 つです。

ほかの EJB コンポーネント (セッション Beans とエンティティ Beans) は同時に呼び出す必要があるため、メッセージ駆動型 Beans が必要となります。これらの EJB コンポーネントは、標準的な EJB インタフェースを介してしかアクセスできないため、非同期メッセージ受信のメカニズムは備わっていません。

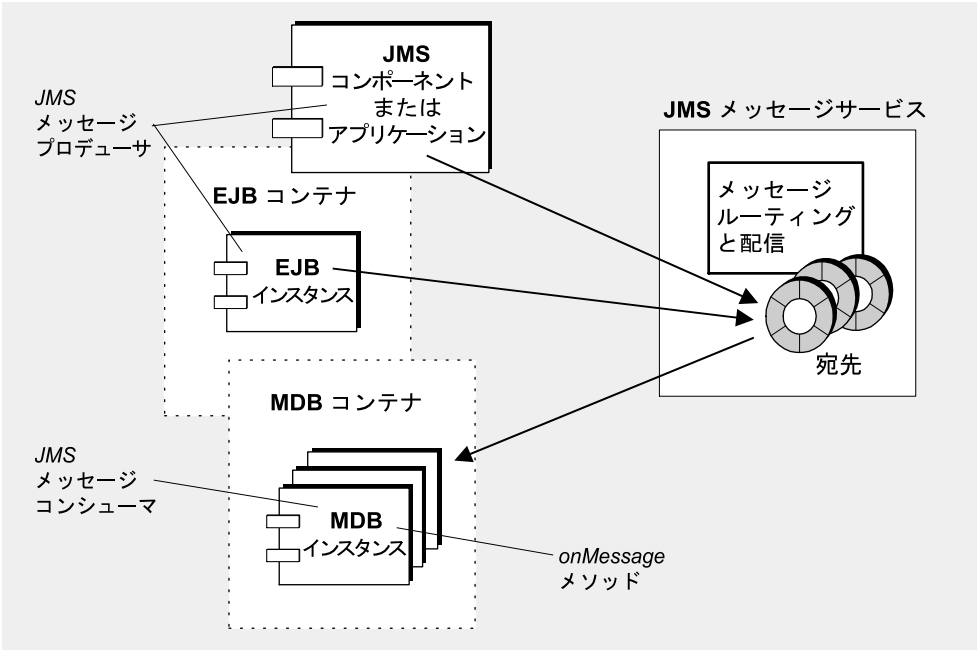
ただし、非同期メッセージングは多くのエンタープライズアプリケーションの要件となっています。これらのアプリケーションの多くは、サーバサイドコンポーネントがサーバリソースを結びつけずに、お互いに通信や応答できることが要件となっています。このため、メッセージのプロデューサにしっかり対応していなくてもメッセージを受信し、コンシュームできる EJB コンポーネントが必要となります。この機能は、サーバサイドコンポーネントがアプリケーションイベントに応答する必要のある、すべてのアプリケーションで必要となります。エンタープライズアプリケーションでは、負荷が増加する場合、この機能を拡張する必要があります。

メッセージ駆動型 Beans

メッセージ駆動型 Beans (MDB) は、特殊な EJB コンテナ (サポートするコンポーネントに分散サービスを提供するソフトウェア環境) のサポート対象となる特殊な EJB コンポーネントです。

メッセージ駆動型 Beans: MDB とは、JMS `MessageListener` インタフェースを実装する JMS メッセージコンシューマです。MDB 開発者が書き込んだ `onMessage` メソッドは、MDB コンテナがメッセージを受信したときに呼び出されます。`onMessage()` メソッドは、標準的な `MessageListener` オブジェクトの `onMessage()` メソッドがコンシュームするのと同じように、メッセージをコンシュームします。ほかの EJB コンポーネメッセージ駆動型 Beans ントの場合とは異なり、メソッドを MDB でリモートに呼び出さないため、これと関連付けられているホームまたはリモートのインタフェースはありません。MDB は単一の送信先からのメッセージをコンシュームできます。43 ページの図 1-3 にあるように、スタンドアロン JMS アプリケーション、JMS コンポーネント、EJB コンポーネントにより、メッセージをプロデュースできます。

図 1-3　MDB を使用したメッセージング



MDB コンテナ : MDB コンテナは、MDB のインスタンスを作成し、メッセージの非同期のコンシュームのためにそのインスタンスを設定する役割を担います。これには、メッセージサービスを使用したコネクションの設定 (認証を含む)、特定の送信先に関するセッションのプールの作成、セッションのプールや関連する MDB インスタンスでメッセージが受信されるときのメッセージ分散の管理が含まれます。コンテナは MDB インスタンスのライフサイクルを制御するため、受信メッセージの読み込みに対応できるように、MDB インスタンスのプールを管理します。

コネクションファクトリと送信先のメッセージコンシュームを設定するときに、コンテナに使用される管理対象オブジェクトの JNDI 検索名を指定する配置記述子は、MDB に関連付けられています。また、配置記述子には、MDB コンテナを設定するため配置ツールに使用されるほかの情報も含まれます。各 MDB コンテナでは、1 つの MDB のインスタンスだけをサポートします。

アプリケーションサーバのサポート

J2EE アーキテクチャ (`http://java.sun.com/j2ee/download.html#platformspec` にある「J2EE Platform Specification」を参照) では、EJB コンテナ (MDB コンテナを含む) がアプリケーションサーバにホストされます。アプリケーションサーバには、トランザクションマネージャ、持続マネージャ、ネーミングサービス、および JMS プロバイダ (メッセージングの場合) などのさまざまなコンテナで必要とされるリソースが用意されています。

Sun ONE Application Server の場合、メッセージングリソースは Sun ONE Message Queue に用意されています。つまり、MQ メッセージングシステム (25 ページの「MQ メッセージングシステムのアーキテクチャ」を参照) は Sun ONE Application Server に統合され、アプリケーションサーバ環境で実行している MDB やほかの JMS メッセージングコンポーネントへ JMS メッセージを送信するために必要なサポートを提供しています。

第 2 章

クイックスタートチュートリアル

この章では、MQ 環境における JMS クライアントプログラミングの概要を簡単に紹介します。具体的には、HelloWorldMessage という簡単なサンプルアプリケーションの作成からコンパイル、実行までの手順をチュートリアル方式で説明します。

この章で説明する手順は次のとおりです。

- 環境設定
- ブローカの起動およびテスト
- 簡単なクライアントアプリケーションの開発
- クライアントアプリケーションのコンパイルおよび実行

このチュートリアルの目的は、デフォルト設定の MQ メッセージサーバを実行することです。MQ メッセージサーバの設定手順については、81 ページの第 5 章「SOAP メッセージの操作」を参照してください。

環境設定

JMS クライアントのコンパイルおよび実行時には、多くの環境変数を使用します。この節では、これらの環境変数の設定方法を説明します。

➤ MQ 関連の環境変数を設定するには

1. 必要に応じて、`IMQ_HOME` 環境変数を設定します (Windows プラットフォームのみ)。

   `IMQ_HOME` 環境変数は、JMS クライアントのコンパイルおよび実行時に使用します。Windows プラットフォームでは、MQ インストーラが `IMQ_HOME` を設定します。

   インストール時に設定されるディレクトリおよび環境変数は次のとおりです。

プラットフォーム	設定
Solaris	インストールディレクトリなし IMQ_HOME 環境変数は使用しない
Solaris : Sun ONE Application Server, EE	Sun ONE Application Server、Evaluation Edition (EE) の場合、デフォルトのインストールディレクトリは *Application Server installation directory*/imq IMQ_HOME 環境変数はソフトウェアでは使用されない。MQ ドキュメント内で、ルートインストールディレクトリを参照するために使用される
Windows	デフォルトのインストールディレクトリは、C:¥Program Files¥Sun Microsystems¥Message Queue 3.0¥ インストーラは、IMQ_HOME 環境変数の値として MQ インストールディレクトリを設定する
Linux	デフォルトのインストールディレクトリは /opt/imq IMQ_HOME 環境変数はソフトウェアでは使用されない。MQ ドキュメント内で、ルートインストールディレクトリを参照するために使用される

2. JAVA_HOME 環境変数を設定します。

   JAVA_HOME 環境変数の値として、J2SE SDK (Java 2 Standard Edition ソフトウェア開発キット) のインストール先ディレクトリを指定します。

3. サンプルアプリケーションが格納されているディレクトリに移動します。

プラットフォーム	設定
Solaris	/usr/demo/imq/jms
Solaris : Sun ONE Application Server, EE	IMQ_HOME/demo/jms
Windows	IMQ_HOME/demo/jms
Linux	IMQ_HOME/demo/jms

この章で使用するサンプルアプリケーションは、このディレクトリ内にあります。

4. CLASSPATH 環境変数を設定します。

   CLASSPATH を設定して、現在のディレクトリおよび IMQ_HOME/lib ディレクトリ (Solaris の場合は /usr/share/lib/imq) 内の jar ファイル (jms.jar、imq.jar jndi.jar) を取り込みます。これらはクライアントのコンパイルおよび実行時に必要になります。

注 JDK 1.4 には、JNDI の jar ファイルと JSSE の jar ファイル (安全なコネクションに必要) が付属しています。これらの jar ファイルを CLASSPATH で取り込む必要があるのは、JDK 1.3 または 1.2 を使ってアプリケーションを開発する場合のみです。

プラットフォーム	設定と詳細

Solaris (csh)	`% cd /usr/demo/imq/jms`
	`% setenv CLASSPATH .:`
	`/usr/share/lib/imq/jms.jar:`
	`/usr/share/lib/imq/imq.jar:`
	`/usr/share/lib/imq/jndi.jar`

Solaris : Sun ONE Application Server, EE	`% cd $IMQ_HOME/demo/jms`
	`% setenv CLASSPATH .:`
	`$IMQ_HOME/lib/jms.jar:`
	`$IMQ_HOME/lib/imq.jar:`
	`$IMQ_HOME/lib/jndi.jar`

Windows	`C:¥>cd %IMQ_HOME%¥demo¥jms`
	`C:¥Program Files¥Sun Microsystems¥`
	`Message Queue 3.0¥demo> set CLASSPATH=.;`
	`%IMQ_HOME%¥lib¥jms.jar;`
	`%IMQ_HOME%¥lib¥imq.jar;`
	`%IMQ_HOME%¥lib¥jndi.jar`

Linux (csh)	`% cd $IMQ_HOME/demo/jms`
	`% setenv CLASSPATH .:`
	`$IMQ_HOME/lib/jms.jar:`
	`$IMQ_HOME/lib/imq.jar:`
	`$IMQ_HOME/lib/jndi.jar`

MQ メッセージサーバの起動およびテスト

このチュートリアルでは、現時点でまだ MQ メッセージサーバを実行していないものとして説明を進めます。メッセージサーバは、メッセージの転送と配信を行う 1 つまたは複数のソフトウェアコンポーネント、すなわちブローカによって構成されます。

UNIX の起動プロセスまたは Windows サービスとしてすでにブローカを実行している場合、次の手順「ブローカをテストするには」は省略できます。

➤ ブローカを起動するには

1. 端末ウィンドウを使って、`IMQ_HOME/bin` ディレクトリ (Solaris の場合は `/usr/bin`) に移動します。
2. 次のようにして、ブローカコマンド (`imqbrokerd`) を実行します。

   `-tty` オプションを指定すると、ログファイルとそこに記録されたすべてのメッセージが端末コンソールに表示されます。

プラットフォーム	起動コマンド
Solaris	`% usr/bin/imqbrokerd -tty`
Solaris : Sun ONE Application Server, EE	`% IMQ_HOME/bin/imqbrokerd -tty`
Windows	`C:¥Program Files¥Sun Microsystems¥ Message Queue 3.0¥bin> imqbrokerd -tty`
Linux	`% IMQ_HOME/bin/imqbrokerd -tty`

ブローカが起動し、いくつかのメッセージに続いて「imqbroker@host:7676 ready.」というメッセージが表示されます。このメッセージは、ブローカをクライアントで使用する準備ができたことを表します。

➤ ブローカをテストするには

ブローカが起動したかどうかを簡単にチェックしたい場合は、MQ コマンドユーティリティ (`imqcmd`) を使ってブローカ情報を表示します。

1. 別の端末ウィンドウを開いて、`IMQ_HOME/bin` ディレクトリ (Solaris の場合は `/usr/bin`) に移動します。
2. 次の引数を指定して、`imqcmd` を実行します。

プラットフォーム	設定と詳細
Solaris	`% /usr/bin/imqcmd query bkr -u admin -p admin`
Solaris : Sun ONE Application Server, EE	`% IMQ_HOME/bin/imqcmd query bkr -u admin -p admin`
Windows	`C:¥Program Files¥Sun Microsystems¥ Sun One Message Queue 3.0¥bin> imqcmd query bkr -u admin -p admin`
Linux	`% IMQ_HOME/bin/imqcmd query bkr -u admin -p admin`

次のような出力が得られます。

```
指定されたブローカを照会中 :
-------------------------
ホスト           プライマリポート
-------------------------
localhost       7676


インスタンス名                              imqbroker
クラスタ URL
クラスタブローカリスト （有効）             localhost/111.222.333.444:7676 (imqbroker)
クラスタブローカリスト （設定済み）
クラスタマスターブローカ
システム内のメッセージの最大サイズ          0 （無制限）
システム内のメッセージの最大数              0 （無制限）
システム内の現在のメッセージのサイズ （バイト単位）     0
システム内の現在のメッセージ数              0
プライマリポート                            7676
ログレベル                                  INFO
ログロールオーバー 間隔 （秒単位）          604800
ログロールオーバーサイズ （バイト単位）     0 （無制限）
最大メッセージサイズ                        70m
自動キュー作成                              true
自動トピック作成                            true
自動的に作成されたキュー配信ポリシー        シングル

ブローカの照会に成功しました。
```

簡単なクライアントアプリケーションの開発

この節では、キューの送信先にメッセージを送信したあと、このメッセージをキューから取得する、「Hello World」という簡単なアプリケーションの作成手順を紹介します。このアプリケーションは、`IMQ_HOME/demo/jms` (Solaris の場合 `/usr/demo/imq/jms`) にあります。

次の手順では、クライアントでメッセージの送受信を行うための Java プログラミング言語コードに注目します。

➤ サンプルアプリケーション HelloWorldMessage を作成するには

1. JMS API のインタフェースおよび MQ 実装クラスをインポートします。

   `javax.jms` パッケージには、JMS クライアントの開発に必要な JMS インタフェースがすべて定義されています。

   ```
   import javax.jms.*;
   ```

2. MQ `QueueConnectionFactory` 管理対象オブジェクトをインスタンス化します。

   `QueueConnectionFactory` オブジェクトにより、MQ メッセージサーバへの `QueueConnection` コネクションを作成するために使用する MQ 固有の設定プロパティすべてがカプセル化されます。

   ```
   QueueConnectionFactory myQConnFactory =
      new com.sun.messaging.QueueConnectionFactory();
   ```

   `ConnectionFactory` 管理対象オブジェクトへのアクセスには、JNDI 検索も利用できます (59 ページの「ConnectionFactory オブジェクトの検索」を参照)。この方法により、クライアントコードが JMS プロバイダに依存しなくなります。また、メッセージングシステムを集中管理できるようになります。

3. MQ メッセージサーバへのコネクションを作成します。

   `QueueConnection` は、ポイントツーポイントプログラミングドメイン内の MQ メッセージサーバへのアクティブなコネクションです。

   ```
   QueueConnection myQConn =
      myQConnFactory.createQueueConnection();
   ```

4. コネクション内にセッションを作成します。

   `QueueSession` オブジェクトは、メッセージのプロデュースおよびコンシュームに使用されるシングルスレッドコンテキストです。クライアントは、このオブジェクトを利用して、キュー送信先のメッセージのプロデューサとコンシューマを作成します。

   ```
   QueueSession myQSess = myQConn.createQueueSession(false,
      Session.AUTO_ACKNOWLEDGE);
   ```

ここで作成された myQSess オブジェクトは未処理です。このオブジェクトは、メッセージがコンシューマによってコンシュームされた時点で、これを自動的に通知します。

5. MQ メッセージサーバ内のキュー送信先に対応する MQ queue 管理対象オブジェクトをインスタンス化します。

   送信先管理対象オブジェクトは、プロバイダ固有の送信先の命名構文および動作をカプセル化します。次のコードは、「world」という物理的なキュー送信先の queue 管理対象オブジェクトをインスタンス化します。

   ```
   Queue myQueue = new.com.sun.messaging.Queue("world");
   ```

   送信先管理対象オブジェクトへのアクセスには、JNDI 検索も利用できます (60 ページの「Destination オブジェクトの検索」を参照)。この方法により、クライアントコードが JMS プロバイダに依存しなくなります。また、メッセージングシステムを集中管理できるようになります。

6. QueueSender メッセージプロデューサを作成します。

   このメッセージプロデューサは myQueue に関連付けられており、キュー送信先「world」にメッセージを送信する際使用されます。

   ```
   QueueSender myQueueSender = myQSess.createSender(myQueue);
   ```

7. メッセージを作成し、キューに送信します。

   QueueSession オブジェクトを使って TextMessage オブジェクトを作成し、このオブジェクトにメッセージのデータを表す文字列を追加します。さらに、QueueSender オブジェクトを使って、キュー送信先「world」にメッセージを送信します。

   ```
   TextMessage myTextMsg = myQSess.createTextMessage();
   myTextMsg.setText("Hello World");
   System.out.println("Sending Message:" + myTextMsg.getText());
   myQueueSender.send(myTextMsg);
   ```

8. QueueReceiver メッセージコンシューマを作成します。

   このメッセージコンシューマは myQueue に関連付けられており、キュー送信先「world」からメッセージを受信する際使用されます。

   ```
   QueueReceiver myQueueReceiver =
      myQSess.createReceiver(myQueue);
   ```

9. 手順 3 で作成した QueueConnection を起動します。

   クライアントによってコンシュームされるメッセージは、すでに起動しているコネクションでしか配信できません。これに対して、クライアントによってプロデュースされたメッセージは、コネクションを起動しなくても送信先に配信できます (手順 7 を参照)。

   ```
   myQConn.start();
   ```

10. キューからメッセージを受信します。

    QueueReceiver オブジェクトを使って、キュー送信先「world」からメッセージを受信します。次のコードは、メッセージの同期コンシュームの例です (40 ページの「メッセージのコンシューム : 同期と非同期」を参照)。非同期コンシュームの例については、54 ページの表 2-1 を参照してください。

    ```
    Message msg = myQueueReceiver.receive();
    ```

11. メッセージのコンテンツを取得します。

    メッセージが正常に受信された場合は、そのコンテンツを取得できます。

    ```
    if (msg instanceof TextMessage) {
       TextMessage txtMsg = (TextMessage) msg;
       System.out.println("Read Message:" + txtMsg.getText());
    }
    ```

12. セッションとコネクションリソースを終了します。

    ```
    myQSess.close();
    myQConn.close();
    ```

クライアントアプリケーションのコンパイルおよび実行

MQ 環境で JMS クライアントをコンパイルして実行するには、Java2 SDK Standard Edition v1.4 の使用をお勧めします (ただし、バージョン 1.3 および 1.2 もサポートしています)。推奨 SDK は次の Web サイトからダウンロードできます。

```
http://java.sun.com/j2se/1.4
```

47 ページの手順 4 で説明したように、CLASSPATH 環境変数が `jms.jar` ファイルと `imq.jar` ファイルをポイントするように設定してから、クライアントアプリケーションのコンパイルまたは実行を開始してください。

次の手順では、50 ページの「簡単なクライアントアプリケーションの開発」で作成した HelloWorldMessage アプリケーションを使用します。このアプリケーションは、MQ 3.0 のサンプルアプリケーションディレクトリに格納されています。

```
IMQ_HOME/demo/jms (Solaris では /usr/demo/imq/jms)
```

➤ HelloWorldMessage アプリケーションをコンパイルして実行するには

1. アプリケーションを含むディレクトリを現在のディレクトリにします。

   Solaris の MQ 3.0 サンプルアプリケーションディレクトリは、ユーザが書き込みできないため、HelloWrodMessage アプリケーションを書き込み可能なディレクトリにコピーして、そのディレクトリを現在のディレクトリにします。

2. 次のようにして、HelloWorldMessage アプリケーションをコンパイルします。

プラットフォーム	設定と詳細
Solaris (csh) の場合	`% $JAVA_HOME/bin/javac HelloWorldMessage.java`
Windows の場合	`C:¥Program Files¥Sun Microsystems¥Message Queue 3.0¥demo¥jms>%JAVA_HOME%¥bin¥javac HelloWorldMessage.java`
Linux (csh) の場合	`% $JAVA_HOME/bin/javac HelloWorldMessage.java`

   これで、現在のディレクトリ内に `HelloWorldMessage.class` ファイルが作成されます。

3. 次のようにして、HelloWorldMessage アプリケーションを実行します。

   HelloWorldMessage アプリケーションを実行したときに表示される出力内容は、examples/jms ディレクトリに格納されています。

プラットフォーム	設定と詳細
Solaris (csh) の場合	`% $JAVA_HOME¥bin¥java HelloWorldMessage`
Windows の場合	`C:¥Program Files¥Sun Microsystems¥Message Queue 3.0¥demo¥jms>%JAVA_HOME%¥bin¥java HelloWorldMessage`
Linux (csh) の場合	`% $JAVA_HOME¥bin¥java HelloWorldMessage`

HelloWorldMessage を実行したときに表示される出力内容は次のとおりです。

```
Sending Message:Hello World
Read Message:Hello World
```

サンプルアプリケーションコード

MQ 3.0 には、サンプルアプリケーションとして、JMS メッセージングアプリケーションと JAXM メッセージングのサンプルが付属しています。詳細については、81 ページの「SOAP メッセージの操作」を参照してください。

JMS のサンプル

HelloWorldMessage のサンプルのコードは、その他のサンプルアプリケーションのコードとともに、次の場所に格納されています。

`IMQ_HOME/demo/jms` (Solaris では `/usr/demo/imq/jms`)

このディレクトリには、各サンプルアプリケーションの概要と実行方法を示した `README` ファイルも格納されています。サンプルには、JMS サンプルプログラムと MQ のサンプルアプリケーションがあります。これらのサンプルについては、次の 2 つの表を参照してください。

表 2-1 は JMS サンプルプログラムの概要です。

表 2-1　JMS サンプルプログラム

サンプルアプリケーション名	説明
SenderToQueue	キューを使ってテキストメッセージを送信する
SynchQueueReceiver	キューを使って同期方式でテキストメッセージを受信する
SynchTopicExample	テキストメッセージをパブリッシュし、トピックを使って同期方式で受信する
AsyncQueueReceiver	メッセージリスナーを使って複数のメッセージを非同期方式で受信する
AsynchTopicExample	トピックに対して 5 つのテキストメッセージをパブリッシュし、メッセージリスナーを使って非同期方式で取得する
MessageFormats	サポートされている 5 つのメッセージ形式でメッセージの書き込みおよび読み取りを行う

表 2-1　JMS サンプルプログラム (続き)

サンプルアプリケーション名	説明
MessageConversion	メッセージ形式によっては、あるデータタイプでメッセージを書き込み、別のデータタイプで読み取ることができることを示す
ObjectMessages	オブジェクトが参照によって渡されるのではなくメッセージにコピーされることを示す
BytesMessages	長さが不定のバイトメッセージを書き込んで読み取る方法を示す
MessageHeadersTopic	JMS メッセージヘッダーフィールドの使用方法を示す
TopicSelectors	メッセージのプロパティをメッセージセレクタとして使用する方法を示す
DurableSubscriberExample	サブスクライバが非アクティブな状態で、トピックに対してパブリッシュされたメッセージを確保するような持続的サブスクライバの作成方法を示す
AckEquivExample	処理が完了するまでメッセージの通知を行わないように設定する方法を示す
TransactedExample	電子商取引アプリケーションのシミュレーションにより、トランザクションの使用方法を示す
RequestReplyQueue	JMS 要求 / 応答機能の使用方法を示す

表 2-2 は MQ のサンプルアプリケーションの概要です。

表 2-2　MQ のサンプルアプリケーション

サンプルアプリケーション名	説明
HelloWorldMessage	「Hello World」メッセージの送受信を行う
XMLMessageExample	ファイルから XML 文書を読み取り、キューに送信する。その後、XML 文書の内容に従ってキュー内のメッセージを処理し、変換して DOM オブジェクトを生成する
SimpleChat	MQ を使って簡単な GUI チャットアプリケーションを作成する方法を示す
SimpleJNDIClient	管理者によって作成され、オブジェクトストアに配置された管理対象オブジェクトに、JNDI 検索を使ってクライアントからアクセスする方法を示す (『MQ 管理者ガイド』の管理コンソールのチュートリアルを参照)

JAXM のサンプル

SOAP メッセージの送受信の方法を示すサンプルも多数用意されています。これらのサンプルの格納場所は次のとおりです。

`IMQ_HOME/demo/jaxm` (Solaris では `/usr/demo/imq/jaxm`)

このディレクトリには、各サンプルアプリケーションの概要と実行方法を示した `README` ファイルも格納されています。これらのサンプルアプリケーションについては、表 2-3 を参照してください。

表 2-3　SOAP メッセージングサンプルアプリケーション

サンプルアプリケーション名	説明
SendSOAPMessage	SOAP メッセージを送信するスタンドアロンクライアント
SOAPEchoServlet	SOAP メッセージをエコーするサーブレット
SendSOAPMessageWithJMS	SOAP メッセージを構築し、JMS メッセージとしてラップし、このメッセージをトピックに対してパブリッシュするスタンドアロンクライアント
ReceiveSOAPMessageWithJMS	JMS でラップされた SOAP メッセージを受信するトピックに登録していて、このメッセージを SOAP メッセージに変換する JMS メッセージリスナー
SOAPtoJMSServlet	SOAP メッセージを受信し、JMS メッセージとしてラップし、トピックに対してパブリッシュするサーブレット

第 3 章

管理対象オブジェクトの使用

管理対象オブジェクトは、プロバイダ固有の実装および構成情報を JMS クライアントが使用するオブジェクト内にカプセル化します。

MQ は、コネクションファクトリと送信先という 2 種類の JMS 管理対象オブジェクトと、JAXM 管理対象オブジェクトを提供します。どちらの管理対象オブジェクトもプロバイダ固有の情報をカプセル化しますが、その用途は異なっています。`ConnectionFactory` オブジェクトと `XAConnectionFactory` (分散トランザクション) オブジェクトは MQ メッセージサーバへのコネクションの作成に使用されます。これに対して、Destination オブジェクト (物理的送信先を表す) は、JMS メッセージコンシューマと JMS メッセージプロデューサの作成に使用されます (50 ページの「簡単なクライアントアプリケーションの開発」を参照)。

JAXM エンドポイント管理対象オブジェクトは、SOAP メッセージの送信に使用されます (第 5 章「SOAP メッセージの操作」を参照)。

管理対象オブジェクトの使用方法は、次の 2 通りです。

- 管理者によって構成され、ネームサービスに格納されます。クライアントは、JNDI 検索コードを使ってこうした管理対象オブジェクトにアクセスし、プロバイダに依存しない方式でこれらを使用します。
- クライアントコードの作成時、開発者によってインスタンス化され、設定されます。この場合、プロバイダ固有の方法で使用されます。

管理対象オブジェクトの使用方法は、アプリケーションの実行環境や、クライアントの MQ 固有の設定内容をどの程度まで制御するかによって決まります。この章では、この 2 通りの方法と、それぞれに対する JMS クライアントコードの作成方法を説明します。

管理対象オブジェクトの JNDI 検索

集中管理型のメッセージ環境の制御下でアプリケーションを実行する場合は、管理者が MQ 管理対象オブジェクトを作成し構成する必要があります。これにより、管理者は、次のことができるようになります。

- クライアントに対して、JNDI 検索を使って事前構成済みの `ConnectionFactory` (および `XAConnectionFactory`) オブジェクトにアクセスするように要求することにより、コネクションの動作を制御する
- クライアントに対して、既存の物理的送信先に対応する `Destination` オブジェクトだけにアクセスするように要求することにより、物理送信先の増加を抑制する

この方法によって、管理者はメッセージサーバとクライアントランタイムの構成内容を管理することが可能となり、同時に、クライアントはプロバイダに依存しない JMS コードを使えるようになります。このため、クライアントは、プロバイダ固有の構文およびオブジェクトの命名規則、またはプロバイダ固有の構成プロパティを把握する必要がなくなります。

管理者は、MQ 管理ツールを使って管理対象オブジェクトを作成します。詳細は、『MQ 管理者ガイド』を参照してください。管理者は、作成する管理対象オブジェクトを読み取り専用に指定できます。この場合、このオブジェクトの作成時に指定した MQ 固有の構成値をクライアント側で変更することができなくなります。つまり、クライアントコードでは、読み取り専用の管理対象オブジェクトに属性値を設定することはできません。また、クライアント起動オプションを使ってこれらをオーバーライドすることもできません。詳細は、64 ページの「オーバーライド指定をしてクライアントアプリケーションを起動」を参照してください。

クライアント側で、クライアント自身の `ConnectionFactory` (および `XAConnectionFactory`) や送信先管理対象オブジェクトをインスタンス化することは可能ですが、その場合は、管理者が、アプリケーションが要求するブローカのリソースを制御し、アプリケーションのパフォーマンスを調整するという管理対象オブジェクトの本来の目的を果たせなくなります。また、管理対象オブジェクトをインスタンス化すると、クライアントアプリケーションはプロバイダ依存になります。

ConnectionFactory オブジェクトの検索

➤ ConnectionFactory オブジェクトの JNDI 検索を行うには

1. JNDI 検索の初期コンテキストを作成します。

   初期コンテキストの作成方法は、ファイルシステムストアを使用するか、MQ 管理対象オブジェクトの LDAP サーバを使用するかによって異なります。次に示すのは、ファイルシステムストアを使用する場合のコードです。対応する LDAP サーバプロパティについては、『MQ 管理者ガイド』を参照してください。

   ```
   Hashtable env = new Hashtable();
   env.put (Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.fscontext.RefFSContextFactory");
   env.put (Context.PROVIDER_URL,
      "file:///c:/imq_admin_objects");
   Context ctx = new InitialContext(env);
   ```

   この例のように、プログラムを使用するのではなく、コマンド行からシステムプロパティを指定して環境設定を行うこともできます。具体的な手順については、サンプルアプリケーションディレクトリのサブディレクトリ `jms` 内の `README` ファイルを参照してください。

   `IMQ_HOME/demo/jms` (Solaris では `/usr/demo/imq/jms`)

   システムプロパティを使って環境設定を行う場合は、`env` パラメータを指定せずにコンテキストを初期化します。

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

2. `ConnectionFactory` オブジェクトや `XAConnectionFactory` オブジェクトが JNDI オブジェクトストアに格納された「検索」名を使って JNDI 検索を行います。

   ```
   QueueConnectionFactory myQConnFactory = (QueueConnectionFactory)
      ctx.lookup("cn=MyQueueConnectionFactory");
   ```

   このコネクションファクトリをオリジナルの構成として使用することをお勧めします。`ConnectionFactory` および `XAConnectionFactory` オブジェクト構成プロパティについては、69 ページの「MQ クライアントランタイムの設定可能なプロパティ」を参照してください。全プロパティの一覧は、125 ページの「ConnectionFactory 管理対象オブジェクト」のとおりです。

3. ConnectionFactory を使って Connection オブジェクトを作成します。

   ```
   QueueConnection myQConn =
      myQConnFactory.createQueueConnection();
   ```

コード例 3-1 に、前の手順で使用したコードを示します。

コード例 3-1　　ConnectionFactory オブジェクトの検索

```
Hashtable env = new Hashtable();
env.put (Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
env.put (Context.PROVIDER_URL,
    "file:///c:/imq_admin_objects");
Context ctx = new InitialContext(env);
QueueConnectionFactory myQConnFactory = (QueueConnectionFactory)
    ctx.lookup("cn=MyQueueConnectionFactory");
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

Destination オブジェクトの検索

➤ Destination オブジェクトの JNDI 検索を行うには

1. ConnectionFactory の検索時と同じ初期コンテキストで、JNDI 検索を行います。JNDI 検索には、Destination オブジェクトが JNDI オブジェクトストアに格納された「検索」名を使用します。

   ```
   Queue myQ =
   (Queue) ctx.lookup("cn=MyQueueDestination");
   ```

管理対象オブジェクトのインスタンス化

アプリケーションを集中管理環境の制御下で実行する必要がない場合は、クライアントコード内の管理対象オブジェクトのインスタンス化および設定が可能です。

この方法では、開発者が、メッセージサーバとクライアントランタイムの設定情報を制御します。この場合、クライアントは、その他の JMS プロバイダではサポートされません。通常は、次のような場合にクライアントコード内の管理対象オブジェクトをインスタンス化します。

- まだ開発の初期段階で、管理対象オブジェクトの作成、設定、格納の必要がない場合。この場合は、JNDI 検索を省略してアプリケーションの開発およびデバッグを行うことができます。
- その他の JMS プロバイダでクライアントをサポートする必要がない場合。

クライアントコード内の管理対象オブジェクトをインスタンス化するということは、設定値をアプリケーションにハードコーディングするということです。この場合、アプリケーションの配置後に、管理者が管理対象オブジェクトを再構成してパフォーマンスやスループットを改善するという柔軟性は得られません。

ConnectionFactory オブジェクトのインスタンス化

MQ の `ConnectionFactory` 管理対象オブジェクトをインスタンス化するオブジェクトコンストラクタは、プログラミングドメインごとに 1 つずつ、合計で 2 個用意されています。

- **パブリッシュ / サブスクライブ (トピック) ドメイン**

  ```
  new com.sun.messaging.TopicConnectionFactory();
  ```

  `TopicConnectionFactory` をデフォルト構成でインスタンス化します (ポート番号 `7676` の「`localhost`」で実行されているブローカに対して Topic TCP ベースのコネクションを作成)。

- **ポイントツーポイント (キュー) ドメイン**

  ```
  new com.sun.messaging.QueueConnectionFactory();
  ```

  `QueueConnectionFactory` をデフォルト設定でインスタンス化します (ポート番号 `7676` の「`localhost`」で実行されているブローカに対してキュー TCP ベースのコネクションを作成)。

➤ ConnectionFactory オブジェクトを直接インスタンス化し、設定するには

1. 適切なコンストラクタを使って、Topic または Queue ConnectionFactory オブジェクトをインスタンス化します。

   ```
   com.sun.messaging.QueueConnectionFactory myQConnFactory =
      new com.sun.messaging.QueueConnectionFactory();
   ```

2. ConnectionFactory オブジェクトを設定します。

   ```
   myQConnFactory.setProperty("imqBrokerHostName", "new_hostname");
   myQConnFactory.setProperty("imqBrokerHostPort", "7878");
   ```

   ConnectionFactory 設定プロパティについては、69 ページの「MQ クライアントランタイムの設定可能なプロパティ」を参照してください。全プロパティの一覧は、125 ページの「ConnectionFactory 管理対象オブジェクト」のとおりです。

3. ConnectionFactory を使って Connection オブジェクトを作成します。

   ```
   QueueConnection myQConn =
      myQConnFactory.createQueueConnection();
   ```

コード例 3-2 に、前の手順で使用したコードを示します。

コード例 3-2 ConnectionFactory オブジェクトのインスタンス化

```
com.sun.messaging.QueueConnectionFactory myQConnFactory =
    new com.sun.messaging.QueueConnectionFactory();
try {
    myQConnFactory.setProperty("imqBrokerHostName", "new_host");
    myQConnFactory.setProperty("imqBrokerHostPort", "7878");
} catch (JMSException je) {
}
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

Destination オブジェクトのインスタンス化

MQ の Destination 管理対象オブジェクトをインスタンス化するオブジェクトコンストラクタは、プログラミングドメインごとに 1 つずつ、合計で 2 個用意されています。

- **パブリッシュ / サブスクライブ (トピック) ドメイン**

  ```
  new com.sun.messaging.Topic();
  ```

  デフォルトの送信先名「Untitled_Destination_Object」で Topic をインスタンス化します。

- **ポイントツーポイント (キュー) ドメイン**

  ```
  new com.sun.messaging.Queue();
  ```

  デフォルトの送信先名「Untitled_Destination_Object」で Queue をインスタンス化します。

➤ Destination オブジェクトを直接インスタンス化し、設定するには

1. 適切なコンストラクタを使って、Topic または Queue Destination オブジェクトをインスタンス化します。

   ```
   com.sun.messaging.Queue myQueue = new com.sun.messaging.Queue();
   ```

2. Destination オブジェクトを設定します。

   ```
   myQueue.setProperty("imqDestinationName", "new_queue_name");
   ```

3. セッションの作成後、Destination オブジェクトを使って MessageProducer または MessageConsumer オブジェクトを作成します。

   ```
   QueueSender qs = qSession.createSender((Queue)myQueue);
   ```

コード例 3-3 にコードを示します。

コード例 3-3 Destination オブジェクトのインスタンス化

```
com.sun.messaging.Queue myQueue = new com.sun.messaging.Queue();
try {
    myQueue.setProperty("imqDestinationName", "new_queue_name");
} catch (JMSException je) {
}
...
QueueSender qs = qSession.createSender((Queue)myQueue);
...
```

オーバーライド指定をしてクライアントアプリケーションを起動

Java アプリケーションだけでなく、メッセージングアプリケーションの起動時にも、コマンド行からシステムプロパティを指定できます。このメカニズムを利用して、クライアントコード内の MQ 管理対象オブジェクトの属性値をオーバーライドすることができます。クライアントコード内の `setProperty()` メソッドでインスタンス化および設定された MQ 管理対象オブジェクトだけでなく、JNDI 検索によってアクセスされる MQ 管理対象オブジェクトの構成もオーバーライドできます。

管理対象オブジェクトの設定をオーバーライドするには、次のコマンド行構文を使用します。

```
java [[-Dattribute=value ]...] clientAppName
```

`attribute` は、69 ページの「MQ クライアントランタイムの設定可能なプロパティ」に記載されている任意の `ConnectionFactory` 管理対象オブジェクトの属性です。

たとえば、クライアントを、クライアントコード内でアクセスされる ConnectionFactory 管理対象オブジェクト内に指定されたブローカ以外のブローカに接続する場合は、そのブローカの `imqBrokerHostName` と `imqBrokerHostPort` が設定されるようにコマンド行で指定して、クライアントを起動します。

`System.setProperty()` メソッドを使って、クライアントコード内のシステムプロパティを設定することもできます。このメソッドは、コマンド行オプションと同じようにして、MQ 管理対象オブジェクトの属性値をオーバーライドします。

ただし、読み取り専用の管理対象オブジェクトの属性値は、コマンド行オプションでも `System.setProperty()` メソッドでも変更できません。オーバーライドの指定は無視されます。

第 4 章

クライアントの最適化

JMS クライアントのパフォーマンスは、クライアントアプリケーション固有の設計と MQ クライアントランタイムの機能と能力に依存します。

この章では、MQ クライアントランタイムが JMS クライアントのメッセージング機能をサポートする仕組み、特にパフォーマンスとメッセージスループットの改善に役立つプロパティおよび動作の設定について説明します。

この章には次のトピックがあります。

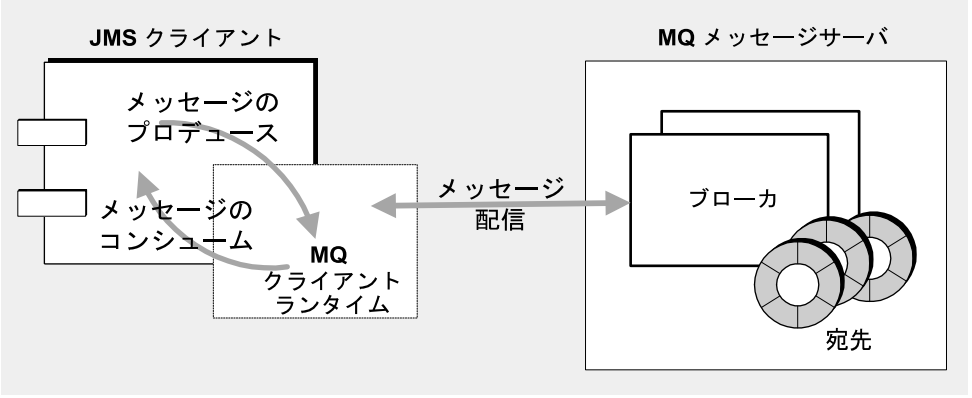
- メッセージのプロデュースとコンシューム
- MQ クライアントランタイムの設定可能なプロパティ
- パフォーマンスに影響を及ぼす要素

メッセージのプロデュースとコンシューム

MQ クライアントランタイムは、JMS クライアントに MQ メッセージサーバへのインタフェースを提供します。つまり、クライアントランタイムによって、27 ページの「JMS プログラミングモデル」に記載されるすべての JMS プログラミングオブジェクトがクライアントに提供されます。クライアントランタイムでは、送信先にメッセージを送信し、送信先からメッセージを受信する場合に、クライアントに必要なすべての処理がサポートされます。

この節では、MQ クライアントランタイムがどのようにメッセージをプロデュースしコンシュームするかについて詳しく説明します。66 ページの図 4-1 は、クライアントと MQ クライアントランタイム間の対話におけるメッセージのプロデュースとコンシュームの様子と、MQ クライアントランタイムと MQ メッセージサーバ間の対話におけるメッセージの配信の様子について示しています。

図 4-1　メッセージングの処理



クライアントによってブローカへのコネクションが作成されると、メッセージ配信用のシングルスレッドコンテキストとしてセッションが作成されます。さらに、メッセージサーバ内の特定の送信先にアクセスするために必要な MessageProducer と MessageConsumer オブジェクトが作成され、メッセージのプロデュース (送信) およびコンシューム (受信) が行われます。

メッセージのプロデュース

メッセージのプロデュースでは、クライアントによってメッセージが作成され、ブローカ上の送信先へのコネクションを介して送信されます。MessageProducer オブジェクトのメッセージ配信モードが、持続的 (配信を 1 回だけ保証する) に設定されている場合、メッセージが送信先に配信されて、ブローカの持続データストアに保存されたことをブローカが通知するまで、クライアントスレッドがブロックされます。メッセージが持続的でない場合、ブローカの通知メッセージ (「Ack」というプロパティ名で呼ばれる) は返されず、クライアントスレッドはブロックされません。

持続メッセージの場合、スループットを改善するため、ブローカの通知を必要としないコネクションを設定できます (78 ページの表 4-7 の `imqAckOnProduce` プロパティを参照)。ただし、この設定では、持続的メッセージの確実な配信を保証できません。

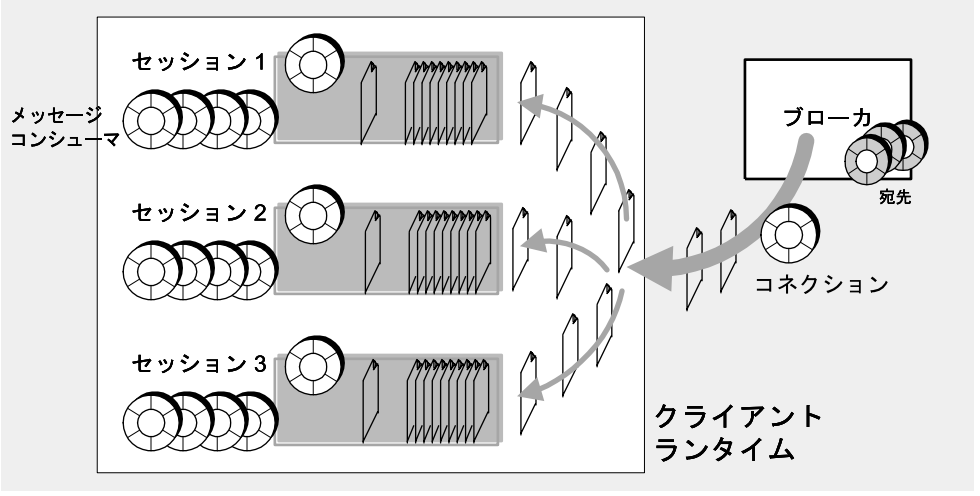
メッセージのコンシューム

メッセージのコンシュームは、プロデュースより複雑です。ブローカの送信先に到着したメッセージは、次の条件に基づいて MQ クライアントランタイムへのコネクションで配信されます。

- クライアントが、特定の送信先のコンシューマを設定している
- コンシューマの選択基準が、特定の送信先に到着したメッセージの基準と一致している
- メッセージの配信を開始するように、コネクションが指示されている

図 4-2 に示されるように、コネクションで配信されるメッセージは、適切な MQ セッションに分散されます。ここで、メッセージは適切な MessageConsumer オブジェクトによってコンシュームされるために、キューに入れられます。メッセージは、1 つずつ各セッションキューからフェッチされ (セッションはシングルスレッドになる)、`receive` メソッドを呼び出すクライアントスレッドによって同期的、または MessageListener オブジェクトの `onMessage` メソッドを呼び出すセッションスレッドによって非同期的にコンシュームされます。

図 4-2　MQ クライアントランタイムへのメッセージの配信



ブローカがクライアントランタイムにメッセージを配信する場合、ブローカはそれに応じてメッセージをマークしますが、メッセージが受信、またはコンシュームされたかどうかは、実際には把握していません。このため、ブローカは、ブローカの送信先からメッセージを削除する前に、クライアントがメッセージの受信を通知するのを待ちます。

JMS 仕様に基づいて、クライアント開発者は、クライアントセッションに次の 3 つの通知オプションを設定できます。

- AUTO_ACKNOWLEDGE: クライアントがコンシュームした個々のメッセージが、セッションによって自動的に通知されます。
- CLIENT_ACKNOWLEDGE: 1 つまたは複数のメッセージがコンシュームされたあと、クライアントによって明示的に通知が行われます。このオプションを指定すると、ほとんどの制御をクライアントで行うことができます。この通知は、メッセージオブジェクトの acknowledge() メソッドを呼び出すことによって行われます。セッションは、現時点までにコンシュームされたすべてのメッセージを通知します。これには、セッション内の複数のメッセージリスナーによって非同期方式でコンシュームされたメッセージも含まれます。なお、メッセージがコンシュームされた順番は考慮されません。
- DUPS_OK_ACKNOWLEDGE: 設定可能な複数のメッセージがコンシュームされたあと、セッションによって通知されます。これらのメッセージが必ず 1 回配信され確実にコンシュームされることを保証しません。クライアントは、メッセージが複数回処理されるかどうかを考慮する必要がない場合に、このモードを使用します。

これら 3 つの通知オプションでは、それぞれ異なった処理レベルと帯域幅オーバーヘッドが必要になります。最もオーバーヘッドの高い自動通知では、メッセージごとにその信頼性が保証されます。これに対して、最もオーバーヘッドの低い DUPS_OK_ACKNOWLEDGE では、メッセージの重複配信が可能です。

AUTO_ACKNOWLEDGE オプションまたは CLIENT_ACKNOWLEDGE オプションを指定した場合、通知を行うか、またはトランザクションをコミットするスレッドはブロックされ、クライアントの通知の受信を知らせる制御メッセージがブローカから返されるまで待機します。このブローカの通知 (プロパティ名は「Ack」) により、対応するコネクションメッセージがブローカによって削除され、二度と送信されないことが保証されます。ただし、クライアントやブローカに障害が発生したりコネクションに障害が発生したような故障時を除きます。

スループットを改善するため、クライアントの通知の受信を知らせるブローカの通知を必要としないコネクションを設定できます (75 ページの表 4-5 の imqAckOnAcknowledge プロパティを参照)。ただし、この設定では、コネクションメッセージの確実な配信を保証できません。

注 DUPS_OK_ACKNOWLEDGE モードでは、セッションはブローカの通知を待機しません。このオプションは、重複メッセージが問題とならない JMS クライアントで使用されます。このほか、JMS API (recover Session) も用意されています。クライアントは、これを使って、受信されたがまだクライアントの通知がないメッセージの配信を明示的に要求できます。このようなメッセージを再配信するときに、ブローカは、これらに Redeliver フラグを付けます。

MQ クライアントランタイムの設定可能なプロパティ

MQ クライアントランタイムは、65 ページの「メッセージのプロデュースとコンシューム」で説明されるすべての操作をサポートします。また、リソース、パフォーマンス、メッセージスループットの最適化に役立つ多くの設定可能なプロパティも提供します。これらのプロパティは、JMS クライアントと MQ メッセージサーバ間の物理コネクションを作成するために使用する ConnectionFactory オブジェクトの属性に対応しています。

ConnectionFactory オブジェクトは、クライアントからブローカへのコネクションを確立するために使用されるだけであり、実際にブローカ内に存在するわけではありません。また、ConnectionFactory オブジェクトは、コネクションの動作や、ブローカにアクセスするためにコネクションを使用するクライアントランタイムの動作を指定する場合にも使用します。ConnectionFactory は、クライアントが設定するメッセージヘッダーの値をオーバーライドする機能により、MQ メッセージサーバの管理にも使用されます。

分散トランザクション (38 ページの「ローカルトランザクション」を参照) をサポートする場合は、分散トランザクションをサポートする特殊な XAConnectionFactory オブジェクトを使用する必要があります。

第 3 章「管理対象オブジェクトの使用」で示されるように、ConnectionFactory 管理対象オブジェクトは、管理者が作成するか、クライアントコード内でインスタンス化します。

ConnectionFactory 管理対象オブジェクトを構成することによって、オブジェクトがプロデュースするすべてのコネクションに共通する属性値 (プロパティ) を指定します。ConnectionFactory オブジェクトと XAConnectionFactory オブジェクトは、一連の同じ属性値を共有します。これらの属性値は、影響を及ぼす動作に基づいて、いくつかのカテゴリに分類されます。

- コネクションの指定
- 自動再コネクションの動作
- クライアントの識別
- メッセージヘッダーのオーバーライド
- 信頼性およびフローの制御
- キューブラウザの動作
- アプリケーションサーバのサポート
- JMS 定義されたプロパティのサポート

各カテゴリについては、それぞれの ConnectionFactory (または XAConnectionFactory) 属性とともに、次の節で説明します。『MQ 管理者ガイド』で示されているように、属性値の設定には MQ 管理ツールを使用します。

コネクションの指定

コネクションは、ブローカのホスト名、ポートマッパー (または特定のコネクションサービス) があるポートの番号、サポートしているコネクションサービスの種類によって指定されます。コネクションタイプ (コネクションサービスが使用するプロトコル) によっては、追加の属性値を設定しないとコネクションが動作しない場合もあります。

表 4-1 に、コネクションの動作に影響を及ぼす属性を示します。

表 4-1 コネクションファクトリ属性 : コネクションの指定

属性 / プロパティ名	説明
inqConnectionType	クライアントが使用するコネクションサービスの転送プロトコルを指定する。サポートするタイプは、TCP、TLS、HTTP。デフォルトは TCP
imqAckTimeout	クライアントランタイムがブローカの通知を待機する最大時間を、ミリ秒単位で指定する。この時間が過ぎると例外がスローされる。値が 0 の場合、タイムアウトはなし。デフォルトは 0 たとえば、ブローカが安全な (SSL) コネクションを介して LDAP ユーザリポジトリに対して初めてユーザを認証するような場合、認証の完了には 30 秒以上かかる可能性がある。このため、imqAckTimeout の設定値が小さすぎると、クライアントランタイムがタイムアウトになる可能性がある
imqBrokerHostName	imqConnectionType が TCP か TLS の場合、接続先のブローカホスト名を指定する。デフォルトは、localhost
imqBrokerHostPort	imqConnectionType が TCP か TLS の場合、ブローカのホストポートを指定する。デフォルトは 7676

表 4-1　コネクションファクトリ属性 : コネクションの指定 (続き)

属性 / プロパティ名	説明
imqBrokerServicePort	imqConnectionType が TCP か TLS の場合、ブローカのホストポート (ポートマッパーのポート) を使用するコネクションを回避してコネクションを試みるポートを指定する。この属性は、主としてファイアウォールを介したコネクションに使用される。ファイアウォールを介したコネクションでは、オープンポートの数を最小限に抑える必要がある。この機能を使用するには、ブローカのコネクションサービス構成プロパティを使って、特定のポート上で特定のサービスを開始する必要がある (『MQ 管理者ガイド』を参照)。デフォルトは 0 (使用しない)
imqSSLIsHostTrusted	imqConnectionType が TLS の場合、ホストが信頼されているかどうかを指定する。デフォルトは true
imqConnectionURL	imqConnectionType が HTTP の場合、MQ メッセージサーバへの接続に使用する URL を指定する。通常の値 (HTTPS コネクション) は https://*hostName:port*/imq/tunnel デフォルトは http://localhost/imq/tunnel

自動再コネクションの動作

MQ には自動再コネクション機能があります。コネクションに障害が発生した場合、MQ は、クライアントランタイムが提供するオブジェクト (セッション、メッセージコンシューマ、メッセージプロデューサなど) を維持したまま、再コネクションを試みます。処理済みセッションの場合、再コネクションによる影響は不明であり、その際の動作は予測できません。一方、処理済みセッションでない場合、再コネクションによる影響はメッセージのプロデュースとメッセージのコンシュームで異なります。

メッセージのプロデュース

再コネクションの間、プロデューサはメッセージを送信できません。この場合、例外がスローされます。プロデューサは、再度コネクションが確立されるまでコネクションを再試行する必要があります。

一時的な送信先は、コネクションが確立されている間だけ存在します。これらは、再度コネクションが確立されたときには失われます。

メッセージのコンシューム

コンシューマは、自動再コネクションの試行中も、クライアントランタイムに配信されたメッセージをコンシュームできます。ただし、再コネクションの動作は次のいくつかの要因によって異なります。

- CLIENT_ACKNOWLEDGE を使ったセッションの場合、持続的サブスクライバまたはキューの受信側に通知されていないメッセージが再配信される
- AUTO_ACKNOWLEDGE を使ったセッションの場合、持続的サブスクライバまたはキューの受信側に最後に配信されたメッセージが再配信される (まだ通知されていない可能性があるため)
- 持続的サブスクライバではない場合、コネクションの修復処理の間、次のようなメッセージは配信されない
  - コネクションの修復処理の間に作成されたメッセージ
  - コネクションの修復処理の間に作成され、配信されなかったメッセージ

表 4-2 に、自動接続機能の動作に影響を及ぼす属性を示します。

表 4-2 コネクションファクトリ属性 自動再コネクションの動作

属性 / プロパティ名	説明
`imqReconnect`	コネクションが失われたとき、クライアントランタイムがブローカへの再コネクションを試行するかどうかを指定する。デフォルトは `false`
`imqReconnectDelay`	`imqReconnect` が `true` の場合、クライアントランタイムが MQ メッセージサーバへの再コネクションを継続的に試行する間隔を指定する。デフォルトは `30,000 milliseconds`
`imqReconnectRetries`	`imqReconnect` が `true` の場合、クライアントランタイムがブローカへの再コネクションを試行する回数を指定する。値 0 は再試行回数に制限がないことを表す。デフォルトは `0`

クライアントの識別

クライアントは、認証と持続的サブスクリプションの追跡のためにブローカに識別される必要があります (36 ページの「クライアント識別子」を参照)。

認証のため、MQ はデフォルトのユーザ名とパスワードを提供します。ユーザリポジトリを明示的に設定する必要がない場合、開発者はこれらを便利に使用できます (『MQ 管理者ガイド』を参照)。

MQ は、永続的サブスクリプションを追跡するため、一意のクライアント ID (ClientID) を使用します。トピック送信先にメッセージが配信されたとき、永続的サブスクライバが非アクティブになっていると、ブローカはそのサブスクライバ宛てのメッセージを保存し、サブスクライバがアクティブになったときに配信します。ブローカがサブスクライバを識別する唯一の方法は ClientID を使用する方法です。

コネクションのために ClientID を設定する方法には、さまざまな方法があります。たとえば、クライアントコードでは、Connection オブジェクトの setClientID() メソッドを使用できます。どのような方法を使用する場合でも、ClientID は、コネクションを使用する前に設定しなければなりません。コネクションの使用後は ClientID の設定または再設定はできません。

クライアントコード内に ClientID を設定する方法が最適というわけではありません。各ユーザには一意の ID が必要です。つまり、調整が一元的に行われることを意味します。このため、MQ は ConnectionFactory オブジェクトに imqConfiguredClientID 属性を提供します。この属性を使用して、各ユーザに一意の ClientID を提供できます。この機能を使用するには、次のように imqConfiguredClientID の値を設定します。

```
imqConfiguredClientID=${u}string
```

特殊な予約文字 ${u} により、コネクションを確立するためのユーザ認証時に一意のユーザ識別子が提供されます。*string* は ConnectionFactory オブジェクト固有のテキスト値です。適切に使用した場合、MQ メッセージサーバは u を u:*username* に置き換え、ユーザ固有の ClientID を生成します。

${u} は、必ず属性値の最初の 4 文字から始める必要があります。「u」以外の文字が検出された場合、コネクションの作成時に JMS 例外がスローされます。属性値の先頭以外の文字で ${} を使用した場合、この文字列はプレーンテキストと見なされ、変数置換は行われません。

追加属性 imqDisableSetClientID を true に設定して、コネクションファクトリを使用するクライアントが Connection オブジェクトの setClientID() メソッドを使用して構成可能な ClientID を変更できないようにすることができます。

配置済みアプリケーション内で永続的サブスクリプションを使用する場合は、プログラム内で setClientID() メソッドを使用するか、ConnectionFactory オブジェクトの imqConfiguredClientID 属性を使用する方法でクライアント識別子を設定する必要があります。

表 4-3 に、クライアント ID に影響を及ぼす属性を示します。

表 4-3 コネクションファクトリ属性 : クライアント ID

属性 / プロパティ名	説明
imqDefaultUsername	ブローカによる認証で使用されるデフォルトのユーザ名を指定する。デフォルトは guest
imqDefaultPassword	ブローカによる認証で使用されるデフォルトのパスワードを指定する。デフォルトは guest
imqConfiguredClientID	管理上設定された ClientID の値を指定する。デフォルトは null

表 4-3　コネクションファクトリ属性 : クライアント ID (続き)

属性 / プロパティ名	説明
imqDisableSetClientID	クライアントが JMS API 内の setClientID() メソッドを使って ClientID を変更できないようにするかどうかを指定する。デフォルトは false

メッセージヘッダーのオーバーライド

MQ 管理者は、メッセージの持続性、有効期間、優先度を指定する JMS メッセージヘッダーフィールドをオーバーライドできます。特に、次のフィールドの値をオーバーライドできます (21 ページの「Java XML Messaging (JAXM) 仕様書」を参照)。

- JMSDeliveryMode (メッセージの持続性 / 非持続性)
- JMSExpiration (メッセージの有効期間)
- JMSPriority (メッセージの優先度。0 ～ 9 の整数値)

MQ 管理者は、メッセージヘッダーの値をオーバーライドすることにより、MQ メッセージサーバのリソースをより厳密に管理できます。ただし、オーバーライドする内容とアプリケーション固有の要件 (メッセージの持続性など) が衝突する可能性もあります。したがって、この機能は、必ず適切なアプリケーションユーザか設計者と相談して使用してください。

MQ では、コネクションレベルでメッセージヘッダーをオーバーライドできます。オーバーライドは、指定されたコネクションのコンテキストで作成されたすべてのメッセージに適用され、対応するコネクションファクトリ管理対象オブジェクトの属性を設定することにより構成されます。これらの属性は、表 4-4 に示されています。

表 4-4　コネクションファクトリ属性 : メッセージヘッダーのオーバーライド

属性 / プロパティ名	説明
imqOverrideJMSDeliveryMode	クライアントによって設定された JMSDeliveryMode フィールドをオーバーライドできるかどうかを指定する。デフォルトは false
imqJMSDeliveryMode	JMSDeliveryMode のオーバーライド値を指定する。値は 1 が非持続、2 が持続を表す。デフォルトは 2
imqOverrideJMSExpiration	クライアントによって設定された JMSExpiration フィールドをオーバーライドできるかどうかを指定する。デフォルトは false
imqJMSExpiration	JMSExpiration のオーバーライド値 (ミリ秒) を指定する。デフォルトは 0 (有効期限なし)

表 4-4　コネクションファクトリ属性 : メッセージヘッダーのオーバーライド (続き)

属性 / プロパティ名	説明
imqOverrideJMSPriority	クライアントによって設定された JMSPriority フィールドをオーバーライドできるかどうかを指定する。デフォルトは false
imqJMSPriority	JMSPriority のオーバーライド値 (0 ～ 9 の整数値) を指定する。デフォルトは 4
imqOverrideJMSHeadersTo TemporaryDestinations	オーバーライド値を一時的な送信先に適用するかどうかを指定する。デフォルトは false

信頼性およびフローの制御

クライアントランタイムによる MQ 制御メッセージ、特にブローカの通知 (属性名内の「Ack」で表される) の使用方法とフローは、さまざまな属性によって決定されます。

表 4-5 に、信頼性とフローの制御に影響を及ぼす属性を示します。

表 4-5　コネクションファクトリ属性 : 信頼性およびフローの制御

属性 / プロパティ名	説明
imqAckOnProduce	true の場合、ブローカは、プロデューシングクライアントからのすべての JMS メッセージ (持続および非持続メッセージ) の受信を通知する。プロデューシングクライアントスレッドはブロックされ、この通知 (プロパティ名の「Ack」で表される) が得られるまで待機する false の場合、ブローカはプロデューシングクライアントからの JMS メッセージ (持続または非持続メッセージ) の受信を一切通知しない。プロデューシングクライアントスレッドはブロックされず、ブローカの通知を待機することもない 値を指定しなかった場合、ブローカは 持続メッセージの受信のみを通知する。プロデューシングクライアントスレッドはブロックされ、これらの通知が得られるまで待機する デフォルトは未指定

表 4-5　コネクションファクトリ属性 : 信頼性およびフローの制御 (続き)

属性 / プロパティ名	説明
imqAckOnAcknowledge	true の場合、ブローカはすべてのコンシューミングクライアントの通知の受信を通知する。コンシューミングクライアントスレッドはブロックされ、ブローカの通知 (プロパティ名の「Ack」で表される) が得られるまで待機する false の場合、ブローカはコンシューミングクライアントの通知の受信を一切通知しない。コンシューミングクライアントスレッドはブロックされず、ブローカの通知を待機することもない 値を指定しなかった場合、ブローカは AUTO_ACKNOWLEDGE モードと CLIENT_ACKNOWLEDGE モードのコンシューミングクライアントの通知の受信は通知するが (コンシューミングクライアントスレッドはブロックされ、これらのブローカの通知が得られるまで待機する)、DUPES_OK_ACKNOWLEDGE モードのコンシューミングクライアントの通知の受信は通知しない (コンシューミングクライアントスレッドはブロックされない) デフォルトは未指定
imqFlowControlCount	測定バッチ内の JMS メッセージ数を指定する。クライアントランタイムに配信された JMS メッセージ数がこの値に達すると、配信が一時的に中断され、それまで配信待ち状態だった制御メッセージがある場合はそのメッセージが配信される。クライアントランタイムからの通知を受け取るとペイロードメッセージ配信が再開され、再度指定の数値に達するまで配信処理が行われる 設定値が 0 の場合、測定バッチ内の JMS メッセージ数は無制限。0 以外の場合、大量の JMS メッセージの配信によって MQ 制御メッセージがブロックされることがないように、クライアントランタイムによってメッセージフローが測定される。デフォルトは 100
imqFlowControlIsLimited	imqFlowControlLimit (imqFlowControlCount の設定値が 0 以外の場合にのみアクティブ) を有効にするかどうかを指定する。デフォルトは false

表 4-5　コネクションファクトリ属性 : 信頼性およびフローの制御 (続き)

属性 / プロパティ名	説明
imqFlowControlLimit	クライアントランタイムに配信できるコンシュームされないメッセージ数の制限値を指定する (imqFlowControlIsLimited の値が true の場合にのみ使用される) imqFlowControlCount によって制御されるフロー測定に従って、クライアントランタイムに配信される JMS メッセージ数が制限値を超えると、メッセージ配信処理が停止する。コンシュームされないメッセージ数が、このプロパティで設定された値より少なくなったときに、配信処理が再開される コンシューミングクライアント側の保留メッセージ数が多くなりすぎると、メッセージ処理に時間がかかりメモリー不足が発生するが、この制限値により、問題を防ぐことができる。デフォルトは 1000

キューブラウザの動作

表 4-6 に、クライアントランタイムのキューブラウザの動作に影響を及ぼす属性を示します。

表 4-6　コネクションファクトリ属性 : キューブラウザの動作

属性 / プロパティ名	説明
imqQueueBrowserMax MessagesPerRetrieve	キュー送信先のコンテンツを検索する場合に、クライアントランタイムが 1 回で取得できる最大メッセージ数を指定する。デフォルトは 1000
imqQueueBrowserRetrieve Timeout	キュー送信先のコンテンツを検索する場合に、クライアントランタイムが例外をスローするまでメッセージの取得を待機する最長時間を指定する。デフォルトは 60,000 milliseconds

アプリケーションサーバのサポート

アプリケーションサーバ環境のセッションの動作は、表 4-7 に示す属性による影響を受けます。基本的な情報は、JMS 仕様書を参照してください。

表 4-7　コネクションファクトリ属性 : アプリケーションサーバのサポート

属性 / プロパティ名	説明
imqLoadMaxToServerSession	JMS アプリケーションサーバ機能専用 MQ ConnectionConsumer が ServerSession のセッションに最大 maxMessages 数のメッセージを読み込むか (値が true)、1 回につき 1 つのメッセージを読み込むか (値が false) を指定する。デフォルトは true

JMS 定義のプロパティのサポート

JMS 定義のプロパティの名前は、JMS によって予約されています。JMS プロバイダは、これらのプロパティをサポートするかどうかを選択できます (21 ページの「Java XML Messaging (JAXM) 仕様書」を参照)。これらのプロパティは、クライアントのプログラミング機能を拡張します。

表 4-8 に MQ がサポートする JMS 定義のプロパティを示します。

表 4-8　コネクションファクトリ属性 : JMS 定義のプロパティのサポート

属性 / プロパティ名	説明
imqSetJMSXUserID	MQ が、プロデュースされたメッセージに JMS 定義のプロパティ JMSXUserID (メッセージの送信元ユーザ ID) を設定するかどうかを指定する。デフォルトは false
imqSetJMSXAppID	MQ が、プロデュースされたメッセージに JMS 定義のプロパティ JMSXAppID (メッセージの送信元アプリケーションの ID) を設定するかどうかを指定する。デフォルトは false
imqSetJMSXProducerTXID	MQ が、プロデュースされたメッセージに JMS 定義のプロパティ JMSXProducerTXID (メッセージがプロデュースされたトランザクションのトランザクション ID) を設定するかどうかを指定する。デフォルトは false
imqSetJMSXConsumerTXID	MQ が、コンシュームされたメッセージに JMS 定義のプロパティ JMSXConsumerTXID (メッセージが消費されたトランザクションのトランザクション ID) を設定するかどうかを指定する。デフォルトは false

表 4-8　コネクションファクトリ属性 : JMS 定義のプロパティのサポート (続き)

属性 / プロパティ名	説明
imqSetJMSXRcvTimestamp	MQ が、コンシュームされたメッセージに JMS 定義のプロパティ JMSXRcvTimestamp (メッセージがコンシューマに配信される時刻) を設定するかどうかを指定する。デフォルトは false

パフォーマンス要因

ブローカとのメッセージのやりとりのメカニズムと、信頼性の高い配信を確保するために使用されるさまざまな MQ 制御メッセージにより、いくつかの要因がメッセージのフローとコンシュームに影響します。

メッセージのスループットとパフォーマンスに影響を与える要因には、配信モード、メッセージフローの測定、メッセージフローの制限、通知モード、およびセッション数があります。これらの要因はまったく別個のものですが、相互作用があるため、特定のクライアントのパフォーマンスにどの要因が影響を与えているかを判別するのは困難です。

配信モード : メッセージは、配信モードによって、1 回だけ配信される場合と (非持続)、必ず 1 回は確実に配信される場合 (持続) があります。配信モードにより信頼性の要件も異なるため、オーバーヘッドの大きさが変わります。特に、ブローカ、クライアントに障害が発生したり、コネクションが失われたときなどに、持続メッセージが失われないようにしたり、2 回配信したりして配信を保証する場合にオーバーヘッドは大きくなります。たとえば、クライアントとブローカの制御メッセージがコネクション全体にフローする数、およびクライアントとブローカの通知の処理が、メッセージのスループットとパフォーマンスに大きな影響を与えます。

メッセージフロー測定 : JMS クライアントによって送受信されるメッセージ (JMS メッセージ) と MQ 制御メッセージは、同じクライアントとブローカのコネクションを使って伝送されます。これが、制御メッセージのフローのボトルネックになる可能性があります。たとえば、ブローカのコネクションがクライアントに配信される JMS メッセージでいっぱいになっているとします。この場合には、ブローカの通知など制御メッセージの配信遅延が発生する可能性があります。このように、遅延が発生する場合は、大量の JMS メッセージ配信によって制御メッセージの配信が停滞している可能性があります。このようなネットワークの混雑を防止するため、MQ はコネクション全体の JMS メッセージのフローを測定します。JMS メッセージはバッチ処理されるので、一定数の JMS メッセージが配信されると配信処理が一時的に停止します。JMS メッセージの配信処理が再開される前に、それまで待ち状態だった制御メッセージが

クライアントに配信されます。JMS メッセージのこのような測定バッチ内のメッセージ数を指定できます (75 ページの表 4-5 の imqFlowControlCount プロパティを参照)。大量の JMS メッセージが配信される場合は、この数を小さくすると、制御メッセージがコネクション全体にフローされる遅延時間が短縮できます。

メッセージフロー制限 : MQ クライアントランタイムコードは、メモリーなどのローカルリソースが限界に達するまでに、処理できる JMS メッセージの配信数を制限できます。この数に達すると、パフォーマンスに悪影響が出ます。このため、MQ では、クライアントへの JMS メッセージのフローを制御することで、セッション内のコンシューム待ちメッセージ数を制限できるようになっています。具体的には、しきい値を指定します (75 ページの表 4-5 の FlowControlLimit を参照)。バッチ処理で配信される JMS メッセージ (79 ページの「メッセージフロー測定」を参照) の数がこのしきい値を超過すると、クライアントランタイムは待機し、コンシュームされないメッセージ数がしきい値を下回ったときに JMS メッセージの配信要求を再開します。その後、再度しきい値に達するまで配信処理が続行されます。

クライアント通知モード : コネクション上で送受信されるクライアントおよびブローカ通知メッセージの数は、クライアント通知モードによって異なります。

- AUTO_ACKNOWLEDGE モードでは、コンシュームされるメッセージごとにクライアントとブローカの通知が要求される。セッションスレッドはブロックされ、ブローカの通知が得られるまで待機する
- CLIENT_ACKNOWLEDGE モードでは、クライアントとブローカの通知がバッチ処理される (通知はメッセージごとに送信されない)。このモードでは、コネクション帯域幅を節約できる。また、通常ブローカ通知の待ち時間も短縮される
- DUPS_OK_ACKNOWLEDGE モードでは、クライアントの通知がバッチ処理で行われ、クライアントスレッドがブロックされない (ブローカの通知が要求されない) ため、スループットが一層改善される。ただし、同じメッセージを 1 回以上配信し、コンシュームすることはできない

セッション数とコネクション数 : セッション内で待ち状態になっているメッセージの数と、これらの処理にかかる時間によって、セッション内のメッセージコンシューマの数と、各コンシューマに読み込めるメッセージの数が決まります。クライアント側のメッセージのプロデュースまたはメッセージのコンシュームに遅延が発生する場合は、アプリケーションを再設計し、より多くのセッションにメッセージプロデューサとメッセージコンシューマを分散し、またはより多くのコネクションにセッションを分散してパフォーマンスを改善できます。

第 5 章

SOAP メッセージの操作

Sun ONE MQ では、SOAP ペイロードを含む JMS メッセージを送信できます。このため、高い信頼性で SOAP メッセージを転送できます。また、SOAP メッセージを JMS サブスクライバにパブリッシュすることもできます。この章では、次の手順を説明します。

- MQ を使用しない SOAP メッセージの送受信
- SOAP ペイロードを含む JMS メッセージの送受信

この章では、最初に SOAP 処理の概要を紹介し、添付ファイル付き SOAP メッセージ用の Java API (JAXM) について説明します。SOAP メッセージ処理を行うには、これらに関する知識が必要です。この章の最後の部分では、SOAP メッセージペイロードを含む JMS メッセージの作成方法について説明します。

SOAP 仕様について十分に理解している場合は、導入部分をとばして 89 ページの「Java での SOAP メッセージング」から読み始めてもかまいません。

SOAP とは

SOAP (Simple Object Access Protocol) は、分散環境内の複数のピア間で構造化データを交換するためのプロトコルです。交換されるデータの構造は、XML スキーマによって指定されます。

SOAP メッセージは、データをシステムに依存しない移植可能な方法で表現できる XML 形式でコード化されているため、移植可能です。XML でコード化されたデータには、レガシーシステムからアクセスできます。また、複数の企業間で同じデータを共有することもできます。SOAP テクノロジは、XML によるデータの統合という特徴を活かして、Web サービスなどの Web ベースのコンピューティングでも一般的に使用されます。ファイアウォールは、SOAP パケットをそのコンテンツタイプ (`text/xml-SOAP`) に基づいて認識します。また、SOAP メッセージヘッダーに含まれる情報に基づいて元にメッセージをフィルタリングします。

SOAP 仕様には、XML メッセージ交換のための規則が記載されています。SOAP 仕様は、XML でコード化された情報を交換する必要がある Web サービスの基盤を形成します。通信を行う双方が、コード化された情報の交換に使用する固有のプロトコルを定義してもかまいません。しかし、SOAP のような標準を採用すれば、開発者は情報交換をサポートする汎用の部品を構築できます。こうした部品は、基本的な SOAP 交換に新たな機能を追加するソフトウェアであったり、SOAP メッセージングを管理するツールであったり、SOAP 処理をサポートするオペレーティングシステムの一部分であったりします。このサポートが得られれば、その他の開発者は Web サービスの作成作業に集中できます。

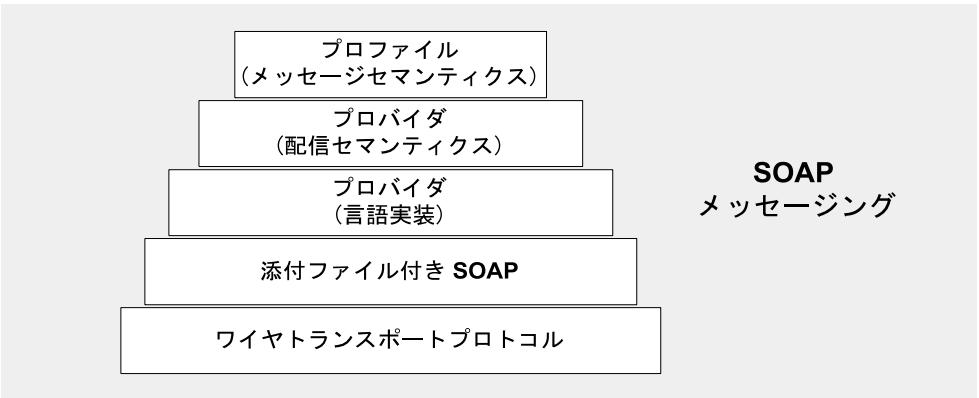
SOAP プロトコルについては、Web サイト `http://www.w3.org/TR/SOAP` にすべて説明されています。この節では、SOAP を使用する理由と JAXM API の操作をより簡単にする基本的概念の一部の説明だけを行います。

SOAP と Java for XML Messaging API

Java API for XML Messaging (JAXM) は、SOAP 標準に準拠させるための Java ベースの API です。この API で SOAP メッセージをアセンブルまたは逆アセンブルすると、構文的に正しい SOAP メッセージが構築されます。また、メッセージを次の送り先に転送する前にその部分部分を複数のアプリケーションで処理する必要がある場合、JAXM を使用することによってメッセージ処理を自動化できます。

図 5-1 に、SOAP メッセージングの実装に使用される層を示します。この章では、SOAP と言語実装層に注目します。

図 5-1　SOAP メッセージング層



次の節では、先ほどの図に示した各層について詳しく説明します。この章の以降の部分では、SOAP と言語実装層に注目します。

トランスポート層

基礎となるメッセージングシステムは、回線上に送信されるメッセージの直列化と、送信先に着信したメッセージビットを解釈するトランスポートプロトコルまたはワイヤプロトコルです。SOAP メッセージの送信には、プロトコルをいくつ使ってもかまいません。しかし、SOAP 仕様に定義されているのは、HTTP とのバインディングだけです。SOAP は HTTP 要求 / 応答メッセージモデルを使用します。これにより、HTTP 要求内には SOAP 要求パラメータ、HTTP 応答内には SOAP 応答パラメータが設定されます。HTTP バインディングには、SOAP メッセージがファイアウォールを通過できるようにするという利点があります。

SOAP 層

トランスポート層の上には SOAP 層があります。この層は、SOAP 仕様書に定義されているとおり、メッセージの構成部分 (エンベロープ、ヘッダー、本体、添付ファイル) の識別に使用する XML スキームを指定します。添付ファイルを除く SOAP メッセージの部分とコンテンツは、すべて XML で書かれています。XML タグを使って SOAP タグを定義する方法については、次のサンプル SOAP メッセージを参照してください。

```
<SOAP-ENV: Envelope
   xmlns: SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV: encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">
      <SOAP-ENV: Body>
          <m: GetLastTradePrice xmlns:m="Some-URI">
              <symbol>DIS</symbol>
          </m: GetLastTradePrice>
      </SOAP-ENV: Body>
</SOAP-ENV: Envelope>
```

SOAP メッセージングを行うために実際に必要なのは、ワイヤトランスポート層と SOAP 層です。送信するメッセージを定義する XML ドキュメントの作成や、メッセージを一方から送信し、もう一方で受信するための HTTP コマンドの作成も可能です。この場合、クライアントは、指定された URL へ同期方式でメッセージを送信することしかできません。残念ながら、このようなメッセージングの有効範囲と信頼性はかなり制限されています。このような制限を克服するため、SOAP メッセージングには、プロバイダ層とプロファイル層が追加されています。

プロバイダ層

図 5-1 では、プロバイダ層を言語実装と配信セマンティクスの 2 つの機能部分として示しています。

プロバイダ言語実装では、API 呼び出しを使って、SOAP に準拠した XML メッセージを作成できます。任意の JAXM が実装されていれば、Java クライアントは、SOAP メッセージとそのすべての部分を Java オブジェクトとして定義できます。クライアントは、JAXM を使ってコネクションを作成し、これを使ってメッセージを送信することもできます。同様に、Java で書かれた Web サービスも同じ (または別の) JAXM API 実装を使って、メッセージの受信、逆アセンブル、メッセージの受信の通知を行うことができます。

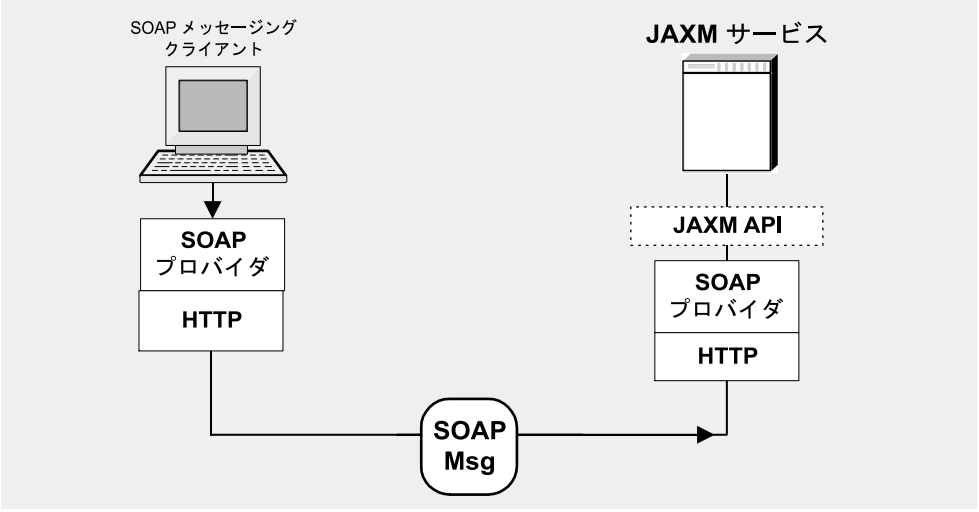
メッセージングのセマンティクス

SOAP プロバイダは、言語実装のほかに、メッセージ配信に関連するサービスを提供できます。たとえば、信頼性、持続性、セキュリティ、管理制御などのサービスがあります。MQ の将来のリリースでは、これらのサービスが SOAP メッセージング用として提供されます。

相互運用性

SOAP プロバイダは、SOAP 仕様に定義されているようにすべてのメッセージを構築または解体する必要があるため、SOAP を使用するクライアントとサービスには相互運用性があります。図 5-2 のように、SOAP メッセージングを行うクライアントとサービスは、同じ言語で書かれていなくてもかまいません。また、同じ SOAP プロバイダを使用していなくてもかまいません。標準化の必要があるのは、メッセージのパッケージのみです。

図 5-2　SOAP の相互運用性



JAXM クライアントまたはサービスを、異なるプロバイダを使用するサービスまたはクライアントと相互運用するには、双方が次の 2 点について合意する必要があります。

- 同じトランスポートバインディング (同じワイヤプロトコル) を使用する
- 送信される SOAP メッセージの構築には同じプロファイルを使用する

次の節で説明するように、プロファイルには追加の処理情報が含まれています。

プロファイル層

SOAP メッセージングの最後の層、プロファイル層は、SOAP プロバイダ で SOAP メッセージングを使用するビジネスパートナー間のメッセージングセマンティクスを管理します。プロファイルは「ebxml」のような業界標準であり、メッセージ処理の追加の規則を定義します。プロバイダは、メッセージファクトリがメッセージを作成するとき、メッセージのヘッダーにプロファイル情報を追加できます。SOAP メッセージヘッダーは SOAP メッセージングの主要な拡張手段です。ebxml プロファイルのサポートは、MQ の将来のリリースで追加される予定です。

SOAP メッセージ

SOAP メッセージング層の概略を把握したところで、今度は SOAP メッセージ自体に注目します。XML で記述された SOAP メッセージの描画は JAXM ライブラリによって行われます。しかし、正しい順序で JAXM 呼び出しを行うには、その構造を把握しておく必要があります。

SOAP メッセージは、SOAP エンベロープ、オプションの SOAP ヘッダー、SOAP 本体からなる XML ドキュメントです。SOAP メッセージヘッダーには、1 つまたは複数の中間ノードを経て最終的な送信先までメッセージを配信するための情報が含まれています。

- エンベロープは、メッセージを表す XML ドキュメントのルート要素です。これにより、メッセージが誰によってどのように処理されるかというフレームワークが定義されます。SOAP プロセッサは、エンベロープ要素を検出した時点で、その XML が SOAP メッセージであることを認識し、メッセージの各部分を検索できるようになります。
- ヘッダーは、SOAP メッセージに機能を追加するための汎用メカニズムです。ヘッダーには、基本プロトコルの拡張機能を定義する子要素を含めることができます。子要素の数に制限はありません。たとえば、ヘッダーの子要素によって定義されるものとして、認証情報、トランザクション情報、ロケール情報などがあります。メッセージを処理するソフトウェアをアクターと呼びます。アクターは、ヘッダーのメカニズムを使って、ある機能が必須機能であるのかオプション機能であるのか、また誰によって使用されるのかを事前の合意なしで定義します。
- 本体は、メッセージの最終的な受信者向けの必須情報を格納するコンテナです。

SOAP メッセージには添付ファイルが付属している場合もあります。添付ファイルは XML で記述されていなくてもかまいません。詳細は、次の SOAP パッケージ化モデルを参照してください。

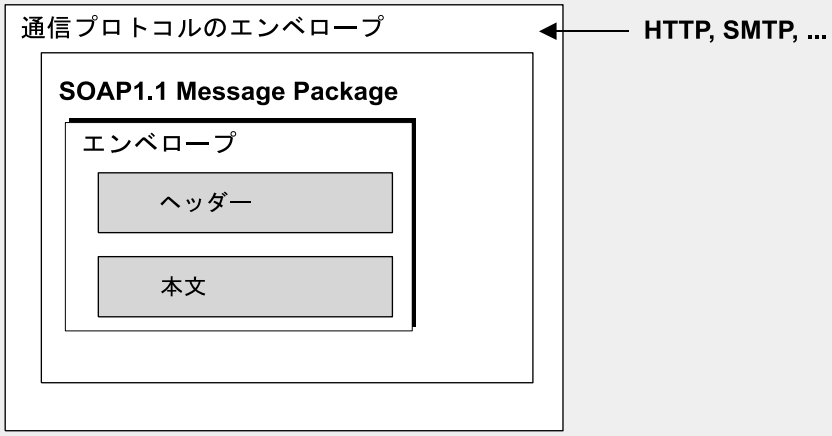
SOAP メッセージは入れ子構造になっています。JAXM を使ってメッセージのアセンブルまたは逆アセンブルを行うとき、必要なメッセージ部分を取得するには、適切な順序で API 呼び出しを行う必要があります。たとえば、メッセージに新しいコンテンツを追加する場合、メッセージの本体部分を取得する必要があります。このためには、入れ子になった SOAP 部分、SOAP エンベロープ、SOAP 本体を使って、データの指定に使用する SOAP 本体要素を取得する必要があります。詳細は、89 ページの「SOAP メッセージオブジェクト」を参照してください。

SOAP パッケージ化モデル

SOAP 仕様には、SOAP メッセージの 2 つのモデルが記述されています。1 つは完全に XML でエンコードされたモデルであり、1 つは XML 以外のデータの含む添付ファイルを送信元が追加できるようにするモデルです。次の 2 つの図を参照して、各モデルの SOAP メッセージの部分を確認してください。JAXM を使って SOAP メッセージとその部分を定義すると、この情報を理解しやすくなります。

図 5-3 は、添付ファイルのない SOAP モデルを示します。このパッケージには、SOAP エンベロープ、ヘッダー、本体が含まれています。ヘッダーはオプションです。

図 5-3　添付ファイルのない SOAP メッセージ

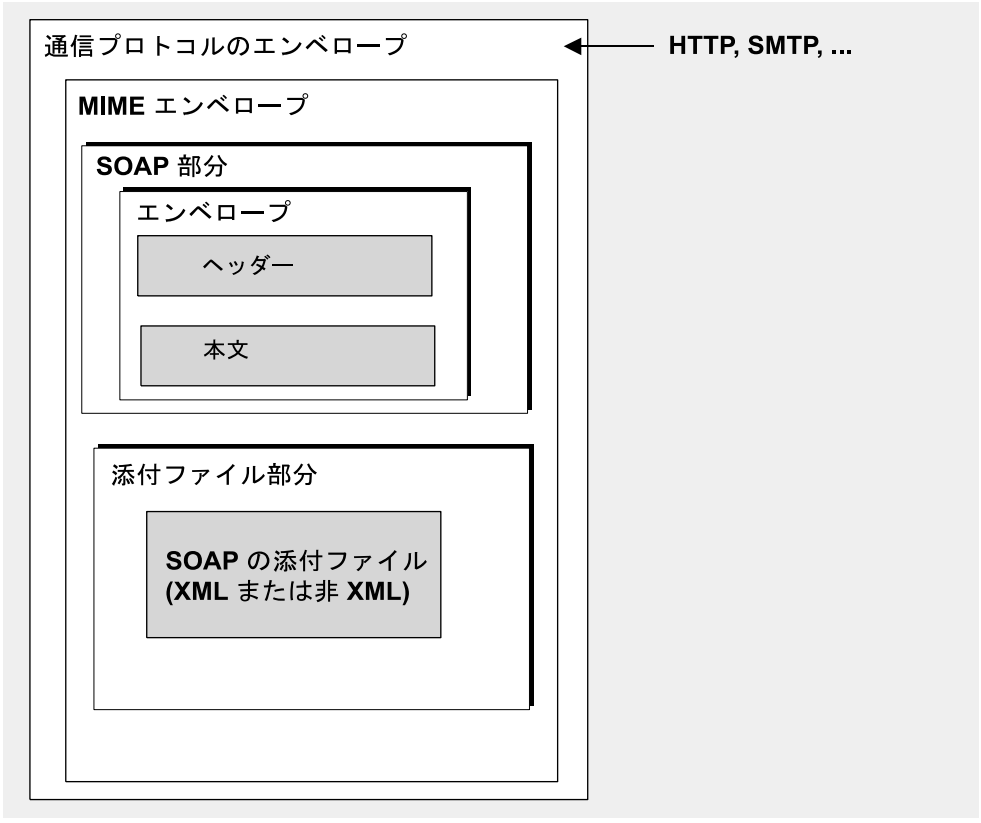


JAXM を使って SOAP メッセージを構築するとき、どちらのモデルを採用するかを指定する必要はありません。添付ファイルを追加すると、図 5-4 のようなメッセージが構築されます。追加しない場合は、図 5-3 のようなメッセージが構築されます。

図 5-4 は、添付ファイル付きの SOAP メッセージを示します。添付ファイルの部分には、イメージファイル、プレーンテキストなどどのような種類のコンテンツでも含めることができます。メッセージの送信元は、添付ファイル付きの SOAP メッセージを作成するかどうかを指定できます。メッセージの受信先も、添付ファイルをコンシュームするかどうかを指定できます。

1 つまたは複数の添付ファイルを含むメッセージは、メッセージのすべての部分を含む MIME エンベロープに同封されます。JAXM では、クライアントが添付ファイル部分を作成すると、MIME エンベロープが自動的に生成されます。メッセージに添付ファイルを追加した場合、MIME ヘッダーに、その添付ファイル内のデータのタイプを指定する必要があります。

図 5-4　添付ファイル付き SOAP メッセージ



Java での SOAP メッセージング

SOAP 仕様には、プログラミングモデルや SOAP メッセージを構築するための API は定義されていません。SOAP メッセージのパッケージ化に使用できる XML スキーマが定義されているだけです。

JAXM は、アプリケーションプログラミングインタフェースの 1 つです。JAXM を実装することにより、SOAP メッセージングのプログラミングモデルをサポートできます。また、SOAP メッセージの構築、送受信、確認のためにアプリケーションやツールの作成者が使用する Java オブジェクトを供給することができます。JAXM は、次の 2 つのパッケージを定義します。

- `javax.xml.soap`: このパッケージのオブジェクトを使って、SOAP メッセージの各部分を定義したり、SOAP メッセージのアセンブルや逆アセンブルを行うことができます。プロバイダのサポートなしでも、このパッケージを使って SOAP メッセージを送信できます。
- `javax.xml.messaging`: このパッケージのオブジェクトでは、プロバイダを使って SOAP メッセージを送信したり、SOAP メッセージを受信したりできます。

この章では、次の操作を行うために、`javax.xml.soap` パッケージとこれによって定義されているオブジェクトおよびメソッドの使用方法について説明します。

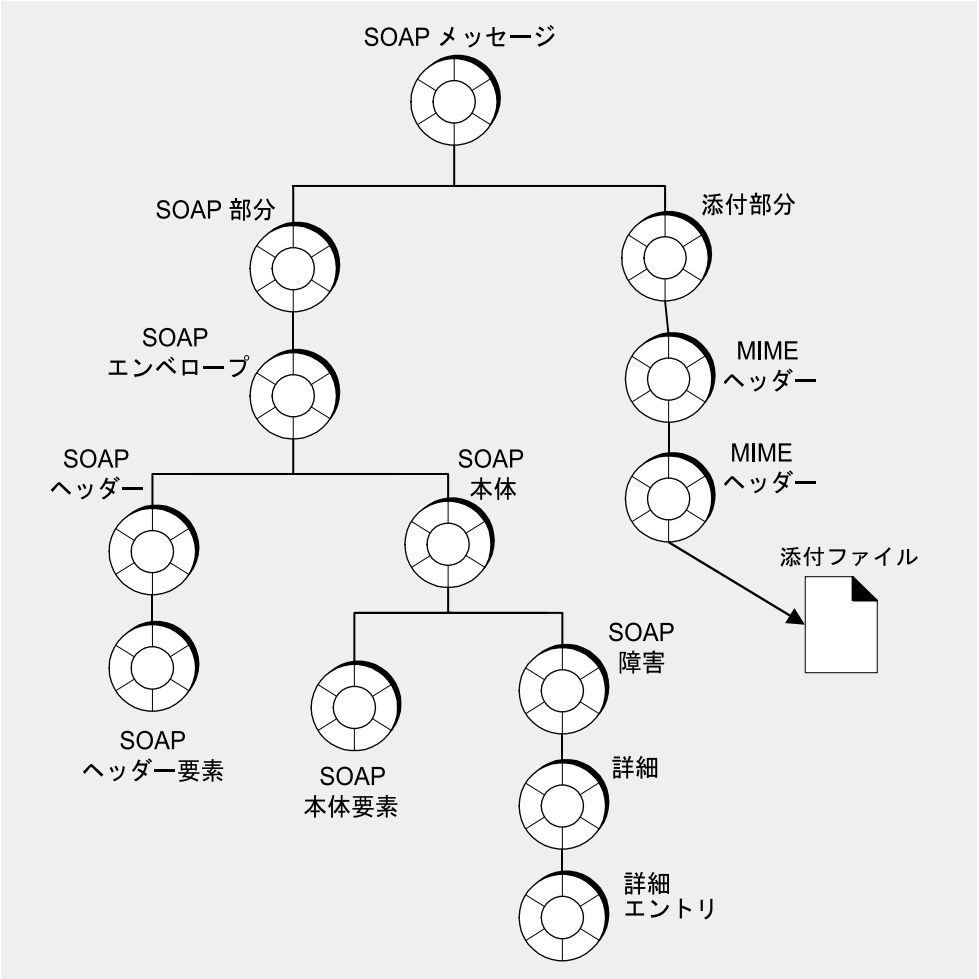
- SOAP メッセージのアセンブルおよび逆アセンブル
- SOAP メッセージの送受信

また、JMS API と MQ を使って SOAP メッセージペイロードを伝送する JMS メッセージを送受信する方法についても説明します。

SOAP メッセージオブジェクト

図 5-5 に示すように、SOAP メッセージオブジェクトはオブジェクトツリーを形成します。`javax.xml.soap` パッケージには、ツリーオブジェクトの派生元となるクラスやインタフェースがすべて定義されています。

図 5-5　SOAP メッセージオブジェクト



上図に示されるように、`SOAPMessage` オブジェクトは SOAP 部分と添付ファイル部分の 2 つの部分に分けられるオブジェクトのコネクションです。重要なことは、添付ファイル部分に XML 以外のデータが含まれている可能性があることです。

メッセージの SOAP 部分のエンベロープには、本体 (データや障害情報を含む) とオプションのヘッダーが含まれています。JAXM を使って SOAP メッセージを作成すると、本体要素だけを作成する必要がある場合でも、SOAP 部分、エンベロープ、および本体が作成されます。このためには、本体要素の親、つまり SOAP 本体を取得する必要があります。

SOAPMessage ツリー内のオブジェクトに移動するには、次のコードに示されるように、ルートからツリー内を移動していく必要があります。たとえば、SOAPMessage MyMsg を作成したとします。この場合、次のような呼び出しで SOAP 本体を取得できます。

```
SOAPPart MyPart = MyMsg.getSOAPPart();

SOAPEnvelope MyEnv = MyPart.getEnvelope();

SOAPBody MyBody = envelope.getBody();
```

ここで、本体要素の名前を作成し (93 ページの「ネームスペース」を参照)、SOAPMessage に本体要素を追加します。

たとえば、次のコードで本体要素の名前 (XML タグ表現) を作成します。

```
Name bodyName = envelope.createName("Temperature");
```

次のコードで本体要素を本体に追加します。

```
SOAPBodyElement myTemp = MyBody.addBodyElement(bodyName);
```

最後に、次のコードで本体要素 bodyName のデータを定義します。

```
myTemp.addTextNode("98.6");
```

継承されたメソッド

SOAP メッセージの要素はツリーを形成します。ツリー内の各ノードは Node インタフェースを実装します。エンベロープレベルから始まる各ノードは、さらに SOAPElement インタフェースも実装します。その結果、表 5-1 のような共有メソッドが得られます。

表 5-1 継承されたメソッド

継承元	メソッド名	用途
SOAPElement	addAttribute(Name, String)	特定の Name オブジェクトと文字列値を持つ属性を追加する
	addChildElement(Name) addChildElement(String, String) addChildElement(String, String, String)	指定の Name オブジェクトで初期化された新しい SOAPElement オブジェクトを作成し、追加する (Name オブジェクトの作成には Envelope.createName メソッドを使用)
	addNameSpaceDeclaration (String, String)	特定のプレフィックスおよび URI を持つネームスペース宣言を追加する

表 5-1 継承されたメソッド (続き)

継承元	メソッド名	用途
	addTextnode(String)	指定の String で初期化された新しい Text オブジェクトを作成し、この SOAPElement オブジェクトに追加する
	getAllAttributes()	このオブジェクト内のすべての属性名に反復子を返す
	getAttributeValue(Name)	特定の属性の値を返す
	getChildElements()	この要素の直系のコンテンツすべてに反復子を返す
	getChildElements(Name)	特定の名前の子要素すべてに反復子を返す
	getElementName()	このオブジェクトの名前を返す
	getEncodingStyle()	このオブジェクトのコード化形式を返す
	getNameSpacePrefixes()	ネームスペースプレフィックスの反復子を返す
	getNamespaceURI(String)	特定のプレフィックスを持つネームスペースの URI を返す
	removeAttribute(Name)	特定の属性を削除する
	removeNamespaceDeclaration (String)	特定のプレフィックスに対応するネームスペース宣言を削除する
	setEncodingStyle(String)	このオブジェクトのコード化形式を String によって指定されたコード化形式に設定する
Node	detachNode()	ツリーからこの Node オブジェクトを削除する
	getParentElement()	この Node オブジェクトの親要素を返す
	getValue	この Node オブジェクトの直接の子の値を返す。ただし、子要素が存在し、その値が text である場合
	recycleNode()	この Node オブジェクトがもう使用されておらず、再利用できることを実装に通知する
	setParentElement (SOAPElement)	このオブジェクトの親を SOAPElement パラメータによって指定された親に設定する

ネームスペース

XML ネームスペースでは、要素名や属性名を同じドキュメント内のその他の名前と区別できます。この節では、XML ネームスペースの概要と SOAP での使用方法を簡単に説明します。詳細は、`http://www.w3.org/TR/REC-xml-names/` を参照してください。

明示的な XML のネームスペース宣言は、次の形式をとります。

<prefix:myElement

`xmlns:`*prefix* `="`*URI*`">`

この宣言は、*prefix* を特定の URI の別名として定義します。`myElement` 内で *prefix* と任意の要素または属性を使って、この要素名または属性名が URI で指定されたネームスペースに属するように指定できます。

次に、ネームスペース宣言の例を示します。

```
<SOAP-ENV: Envelope
```

```
xmlns: SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

この宣言は、ネームスペースの別名として `SOAP_ENV` を定義します。

```
http://schemas.xmlsoap.org/soap/envelope/
```

定義済みの別名は、`Envelope` 要素内の任意の属性または要素のプレフィックスとして使用できます。コード例 5-1 では、要素 `<Envelope>` と `<Body>` および属性 `encodingStyle` は、すべて URI `"http://schemas.xmlsoap.org/soap/envelope/"` で指定された SOAP ネームスペースに属します。

コード例 5-1 明示的なネームスペース宣言

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
                      "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
        <HeaderA
 xmlns="HeaderURI"
 SOAP-ENV:mustUnderstand="0">
    ヘッダーのテキスト
     </HeaderA>
 </SOAP-ENV:Header>
   <SOAP-ENV:Body>
.
.
```

コード例 5-1 明示的なネームスペース宣言 (続き)

```
.
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

ネームスペースを定義する URI が実際の場所をポイントしている必要はありません。URI の目的は、属性名と要素名を区別することです。

事前定義済みの SOAP ネームスペース

SOAP には、次の 2 つのネームスペースが定義されています。

- 次のネームスペース識別子を持つ SOAP エンベロープ (SOAP メッセージのルート要素)

  `"http://schemas.xmlsoap.org/soap/envelope"`

- 次のネームスペース識別子を持つ SOAP 直列化 (SOAP の直列化の規則を定義する URI)

  `"http://schemas.xmlsoap.org/soap/encoding"`

JAXM を使ってメッセージを構築またはコンシュームする場合は、ネームスペースの設定および処理を正しく行い、不正なネームスペースを持つメッセージを破棄する必要があります。

ネームスペースによる SOAP 名の作成

SOAP メッセージの本体要素やヘッダー要素を作成するときは、`Name` オブジェクトを使って、その要素に適した名前を指定する必要があります。`Name` オブジェクトは、`SOAPEnvelope.createName` メソッドを呼び出すことで取得できます。

このメソッドを呼び出すとき、パラメータとしてローカル名を指定するか、またはローカル名、プレフィックス、URI を指定できます。たとえば、次のコードは Name オブジェクト `bodyName` を定義します。

```
Name bodyName = MyEnvelope.createName("TradePrice",
                                      "GetLTP",
                                      "http://foo.eztrade.com");
```

これは、次のネームスペース宣言と同等です。

```
<GetLTP:TradePrice xmlns:GetLTP= "http://foo.eztrade.com">
```

次のコードでは、名前を作成し、`SOAPBody` 要素と関連付ける方法を示します。`createName` メソッドの使用方法と配置に注目してください。

```
SoapBody body = envelope.getBody(); //get body from envelope
```

```
Name bodyName = MyEnvelope.createName("TradePrice",
                                      "http://foo.eztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

Name オブジェクトのパース

特定の Name オブジェクトの名前をパースするときは、次の Name メソッドを使用して名前をパースします。

- getQualifiedName は「*prefix:LocalName*」を返す。特定の名前を使用する場合、GetLTP:TradePrice を返す
- getURI は "http://foo.eztrade.com" を返す
- getLocalName は「TradePrice」を返す
- getPrefix は「GetLTP」を返す

送信先、メッセージファクトリ、コネクションオブジェクト

SOAP メッセージングは、メッセージファクトリによって作成された SOAP メッセージがコネクションを介してエンドポイントに送信されるとき行われます。

- プロバイダを使用しない場合は、次の作業を行う必要があります。
  - SOAPConnectionFactory オブジェクトの作成
  - SOAPConnection オブジェクトの作成
  - メッセージの送信先を表す Endpoint オブジェクトの作成
  - MessageFactory オブジェクトの作成と、このオブジェクトによるメッセージの作成
  - メッセージの生成
  - メッセージの送信
- プロバイダを使用する場合は、次の作業を行う必要があります。
  - ProviderConnectionFactory オブジェクトの作成
  - プロバイダコネクションファクトリからの ProviderConnection オブジェクトの取得
  - プロバイダコネクションからの MessageFactory オブジェクトの取得と、このオブジェクトによるメッセージの作成
  - メッセージの生成
  - メッセージの送信

次の 3 つの項では、エンドポイント、メッセージファクトリ、コネクションオブジェクトについて詳しく説明します。

エンドポイント

エンドポイントは、メッセージの最終的な送信先を表します。エンドポイントは、プロバイダを使用する場合 Endpoint クラス、使用しない場合 URLEndpoint クラスによって定義されます。

エンドポイントの構築

エンドポイントを初期化するときは、そのコンストラクタを呼び出すか、ネーミングサービス内でエンドポイントを検索します。エンドポイント用の管理対象オブジェクトの作成方法については、98 ページの「JAXM 管理対象オブジェクトの使用」を参照してください。

次のコードでは、コンストラクタを使って URLEndpoint を作成します。

```
myEndpoint = new URLEndpoint("http://somehost/myServlet");
```

エンドポイントによるメッセージのアドレス指定

プロバイダを使用する場合、メッセージを作成するメッセージファクトリにより、メッセージヘッダー内にエンドポイントが指定されます。

プロバイダを使用しない場合、SOAP メッセージの送信に使用する SOAPConnection.call メソッドのパラメータとしてエンドポイントを指定できます。

複数のエンドポイントへのメッセージ送信

管理対象オブジェクトを使ってエンドポイントを定義する場合は、その管理対象オブジェクトと複数の URL を関連付けることができます。この場合、各 URL が着信 SOAP メッセージを処理することができます。次のサンプルコードは、検索名が myEndpoint というエンドポイントと 2 つの URL (http://www.myServlet1/ および http://www.myServlet2/) を関連付けています。

```
imqobjmgr add
    -t e
    -l "cn=myEndpoint"
    -o "imqSOAPEndpointList=http://www.myServlet1/
                    http://www.myServlet2/"
```

この構文により、SOAP コネクションを使って複数のエンドポイントに SOAP メッセージをパブリッシュできます。エンドポイント管理対象オブジェクトの詳細は、98 ページの「JAXM 管理対象オブジェクトの使用」を参照してください。

メッセージファクトリ

メッセージファクトリを使って SOAP メッセージを作成できます。

- プロバイダを使用する場合は、プロバイダコネクションの `createMessageFactory` メソッドを使ってメッセージファクトリを作成します。たとえば、プロバイダコネクションが `con` の場合、次のコードでメッセージファクトリ `mf` を作成できます。

  ```
  MessageFactory mf = con.createMessageFactory(xProfile);
  ```

  `createMessageFactory` メソッドに *profile* パラメータを指定することにより、メッセージファクトリによって作成されたメッセージのヘッダーに配置するアドレス指定やその他の情報を決定します。

- プロバイダを使用しない場合は、次のようにメッセージファクトリを直接インスタンス化します。

  ```
  MessageFactory mf = MessageFactory.newInstance();
  ```

コネクション

JAXM を使って SOAP メッセージを送信する場合は、`SOAPConnection` または `ProviderConnection` を取得する必要があります。さらに、MQ を使って SOAP メッセージを転送することもできます。詳細は、114 ページの「SOAP と MQ の統合」を参照してください。

SOAP コネクション

`SOAPConnection` では、リモートロケーションにメッセージを直接送信できます。`SOAPConnection` オブジェクトを取得するには、単に静的メソッド `SOAPConnectionFactory.newInstance()` を呼び出します。このタイプのコネクションでは、信頼性も安全性も保証されません。

プロバイダコネクション

`ProviderConnectionFactory` から取得した `ProviderConnection` を使って、特定のメッセージングプロバイダへのコネクションを作成します。プロバイダを使って SOAP メッセージを送信する場合、このメッセージはプロバイダに転送されます。このメッセージは、プロバイダによって最終的な送信先に配信されます。プロバイダは信頼性の高い、安全なメッセージングを保証します。現在、MQ は SOAP プロバイダサポートを提供していません。

JAXM 管理対象オブジェクトの使用

管理対象オブジェクトは、プロバイダ固有の設定およびネーミング情報をカプセル化するオブジェクトです。エンドポイントオブジェクトの場合、このようなオブジェクトをインスタンス化するか、管理対象オブジェクトを作成してエンドポイントオブジェクトのインスタンスに関連付けることができます。

JNDI 検索でエンドポイントを作成する方法には、コードからエンドポイントの URL を切り離すことができるという利点があります。この場合、アプリケーションは、コードの再コンパイルなしで送信先を変更できます。さらに、プロバイダへの非依存性という利点も得られます。

SOAP 要素の管理対象オブジェクトの作成方法は、MQ 内で管理対象オブジェクトを作成する方法と同じです。具体的には、オブジェクトマネージャ (`imqobjmgr`) ユーティリティを使ってオブジェクトの検索名、属性、タイプを指定します。

エンドポイント管理対象オブジェクトの作成時に指定する必要がある属性やその他の情報は、表 5-2 のとおりです。すべての属性を文字列として指定してください。

表 5-2 SOAP 管理対象オブジェクトに関する情報

オプション	説明
-o "*attribute=val*"	次に示す 3 つのエンドポイント管理対象オブジェクトの属性を指定するオプション • URL のリスト -o "`imqSOAPEndpointList` = "*url1 url2 ....urln*" このリストには、スペースで区切られた 1 つまたは複数の URL が含まれる。複数の URL が含まれる場合は、メッセージはすべての URL にブロードキャストされる。各 URL を、SOAP メッセージの受信と処理が可能なサーブレットに関連付ける • 名前 -o "`imqSOAPEndpointName=`*SomeName*" 名前を指定しない場合、デフォルトで `Untitled_Endpoint_Object` が使用される • 説明 `-o "imqSOAPEndpointDescription=my endpoints for broadcast"` 説明を指定しない場合、デフォルト値 `"A description for the endpoint object"` が指定される

表 5-2　SOAP 管理対象オブジェクトに関する情報 (続き)

オプション	説明
-l "cn=*lookupName*"	エンドポイントの検索名を指定するオプション
-t *type*	オブジェクトのタイプを指定するオプション。エンドポイントの場合、常に e
-i *filename*	imqobjmgr コマンドが記述されている入力ファイルの名前を指定するオプション。通常、このような入力ファイルは、オブジェクトストア属性の指定に使用される
-j "*attribute=val*"	オブジェクトストア属性を指定するオプション。入力ファイル内にも指定できる。入力ファイルの指定には、-i オプションを使用する

コード例 5-2 では、imqobjmgr コマンドを使ってエンドポイントの管理対象オブジェクトを作成し、オブジェクトストアに追加する方法を示します。-i オプションで、オブジェクトストア属性 (-j オプション) を定義する入力ファイル名を指定します。

コード例 5-2　エンドポイント管理対象オブジェクトの追加

```
imgobjmgr add
    -t ep
    -l "cn=myEndpoint"
    -o "imqSOAPEndpointList=http://www.myServlet/
                    http://www.myServlet2/"
    -o "imqSOAPEndpointName=MyBroadcastEndpoint"
    -i MyObjStoreAttrs
```

管理対象オブジェクトを作成してオブジェクトストアに追加しておけば、JAXM アプリケーション内でエンドポイントを使用するとき、この管理対象オブジェクトを使用できます。コード例 5-3 では、JNDI 検索の初期コンテキストを作成してから、必要なオブジェクトを検索します。

コード例 5-3 エンドポイント管理対象オブジェクトの検索

```
Hashtable env = new Hashtable();
env.put (Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
env.put (Context.PROVIDER_URL,
    "file:///c:/imq_admin_objects");
Context ctx = new InitialContext(env);
Endpoint mySOAPEndpoint = (Endpoint)
     ctx.lookup("cn=myEndpoint");
```

管理対象オブジェクトの一覧表示、削除、更新も可能です。詳細は、『MQ 管理者ガイド』を参照してください。

SOAP メッセージングモデルと例

この節では、JAXM を使って SOAP メッセージを送受信する方法を説明します。JAXM を使って SOAP メッセージを構築し、JMS メッセージのペイロードとして送信することもできます。詳細は、114 ページの「SOAP と MQ の統合」を参照してください。

JAXM は、SOAP メッセージングの実行に使用できる 2 つのモデルを提供します。1 つは `SOAPConnection` オブジェクトを使用し、1 つは `ProviderConnection` オブジェクトを使用します。現在、MQ は `ProviderConnection` オブジェクトをサポートしていません。

SOAP メッセージングプログラミングモデル

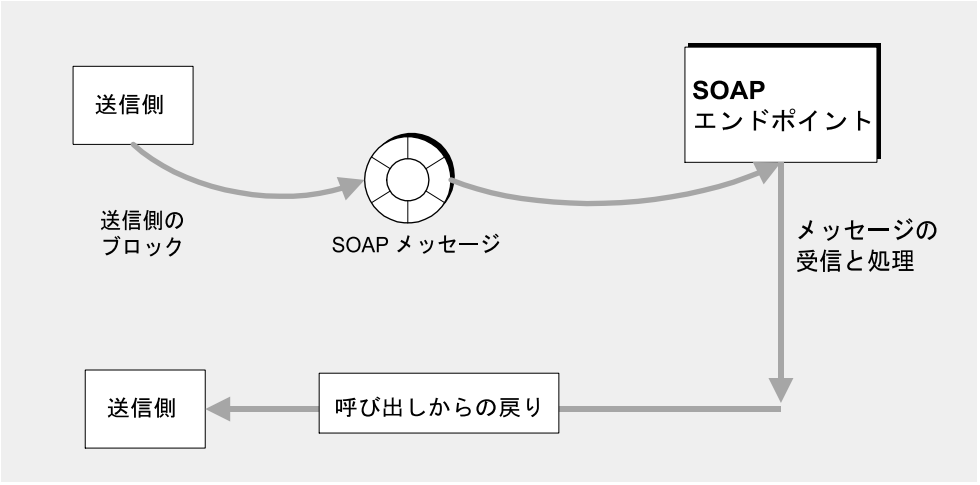
この項では、JAXM を使った SOAP メッセージングで使用するプログラミングモデルの概要を説明します。

SOAP メッセージは、コネクションを経由してエンドポイントへ送信されます。コネクションには、ポイントツーポイントコネクション (`SOAPConnection` クラスによって実装される) とプロバイダコネクション (`ProviderConnection` クラスによって実装される) の 2 種類があります。

ポイントツーポイントコネクション

ポイントツーポイントコネクションを使って、要求 - 応答メッセージングモデルを確立します。要求 - 応答モデルは 図 5-6 に示すとおりです。

図 5-6　要求 - 応答メッセージング



このモデルを使って、クライアントは次の処理を実行できます。

- エンドポイントを作成して、メッセージを送信する `SOAPConnection.call` メソッドに渡される URL を指定する

  エンドポイントのさまざまな作成方法については、96 ページの「エンドポイント」を参照

- SOAPConnection ファクトリを作成し、SOAP コネクションを取得する
- メッセージファクトリを作成し、これを使って SOAP メッセージを作成する
- メッセージのコンテンツ名を作成し、メッセージにコンテンツを追加する
- `SOAPConnection.call` メソッドを使ってメッセージを送信する

クライアントは、呼び出しメソッドから返される `SOAPMessage` オブジェクトを無視します。このオブジェクトはクライアントのブロックを解除するために返されるだけです。

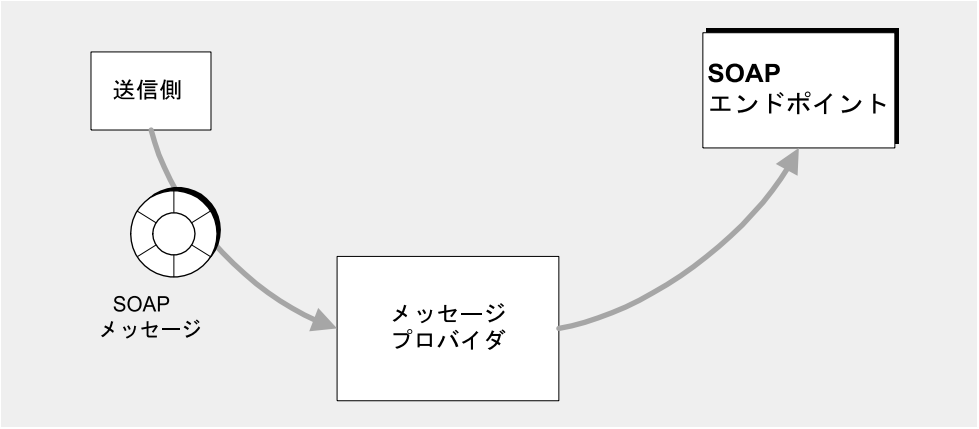
要求 - 応答メッセージを待機する JAXM サービスは `ReqRespListener` オブジェクトを使ってメッセージを受信します。

ポイントツーポイントメッセージングを行うクライアントの詳しい例については、104 ページの「SOAP クライアントの作成」を参照してください。

プロバイダコネクション

プロバイダコネクションを使って、単方向メッセージングを実装できます。単方向メッセージングモデルの仕組みは 図 5-7 に示すとおりです。

図 5-7 単方向メッセージング



ポイントツーポイントモデルとは対照的に、メッセージの最終的な送信先はプロバイダによってメッセージヘッダー内に記述されます。(管理者がメッセージングプロバイダを構成する場合は、Endpoint オブジェクトのリストを提供できます。クライアントがプロバイダを使ってメッセージを送信する場合は、プロバイダはメッセージングプロバイダの Endpoint オブジェクトに記載された送信先に対してのみメッセージを送信します。)

プロバイダコネクション経由で送信されるメッセージは、常にプロバイダ内の中間送信先を経て最終的な送信先に転送されます。プロバイダは、転送の信頼性とメッセージの機密性に対しても責任を負います。

このモデルを使って、クライアントは次の処理を実行できます。

- プロバイダコネクションファクトリを作成してコネクションを取得する
- メッセージファクトリを作成して新しいメッセージを作成する
- コンテンツ名を作成してコンテンツをメッセージに追加する
- メッセージを送信する。送信メソッドは非同期方式であり、即座に返信する

単方向メッセージを待機する JAXM サービスは、`OnewayListener` オブジェクトを使って非同期方式でメッセージを受信します。

添付ファイルの操作

メッセージに XML 以外のデータを含める場合は、添付ファイルとして追加する必要があります。1 つのメッセージに任意の数の添付ファイル部分を追加できます。添付ファイル部分の内容は、プレーンテキストからイメージファイルまで多岐にわたります。

添付ファイルを作成するには、URL オブジェクトを作成して、SOAP メッセージに添付するファイルの場所を指定します。さらに、添付ファイル内のデータを解釈するためのデータハンドラを作成します。最後に、添付ファイルを SOAP メッセージに追加します。

添付ファイル部分を作成し、メッセージに追加するには、JavaBeans Activation Framework (JAF) API を使用する必要があります。この API では、データの任意の部分のタイプを判断し、そのアクセスをカプセル化できます。また、使用可能な操作を検出して、この操作を実行する Beans をアクティブ化することもできます。JavaBeans Activation Framework を使用するためには、クライアントコードに `activation.jar` ライブラリを追加する必要があります。

➤ 添付ファイルを作成し、追加するには

1. URL オブジェクトを作成し、初期化して、SOAP メッセージに添付するファイルの場所を追加します。

   ```
   URL url = new URL("http://wombats.com/img.jpg");
   ```

2. データハンドラを作成し、デフォルトハンドラで初期化して、ハンドラのデータソースの場所を表す URL を渡します。

   ```
   DataHandler dh = new DataHandler(url);
   ```

3. 添付ファイル部分を作成し、イメージの URL を含むデータハンドラで初期化します。

   ```
   AttachmentPart ap1 = message.createAttachmentPart(dh);
   ```

4. 添付ファイル部分を SOAP メッセージに追加します。

   ```
   myMessage.addAttachmentPart(ap1);
   ```

添付ファイルを作成し、メッセージに追加したあとは、通常どおりメッセージを送信できます。

JMS を使ってメッセージを送信する場合は、`SOAPMessageIntoJMSMessage` 変換ユーティリティを使って、添付ファイル付きの SOAP メッセージを JMS メッセージに変換できます。その後、MQ を使って、このメッセージをトピックの JMS キューに送信できます。

例外および障害処理

SOAP アプリケーションは、SOAP 例外と SOAP 障害の 2 種類のエラー報告メカニズムを使用できます。

- SOAP 例外では、SOAP 要求の生成時や応答の非整列化時にクライアントサイドで発生したエラーを処理できます。
- SOAP 障害では、要求の非整列化時、メッセージの処理時、応答の整列化時にサーバサイドで発生したエラーを処理できます。こうしたエラーに対して、サーバサイドのコードは、本体要素ではなく障害要素を含む SOAP メッセージを作成し、このメッセージをその送信元へ送信する必要があります。メッセージの受信側がメッセージの最終的な送信先ではない場合は、受信側自体を `soapactor` として特定し、メッセージの送信側がエラーの発生場所を特定できるようにします。詳細は、110 ページの「SOAP 障害の処理」を参照してください。

SOAP クライアントの作成

ポイントツーポイントメッセージングの SOAP クライアントを作成するには、次の手順に従って呼び出しを行います。

1. `SOAPConnectionFactory` のインスタンスを取得します。

```
SOAPConnectionFactory myFct = SOAPConnectionFactory.newInstance();
```

2. `SOAPConnectionFactory` オブジェクトから SOAP コネクションを取得します。

   ```
   SOAPConnection myCon = myFct.createConnection();
   ```

   `myCon` オブジェクトが返されます。このオブジェクトは、メッセージの送信に使用されます。

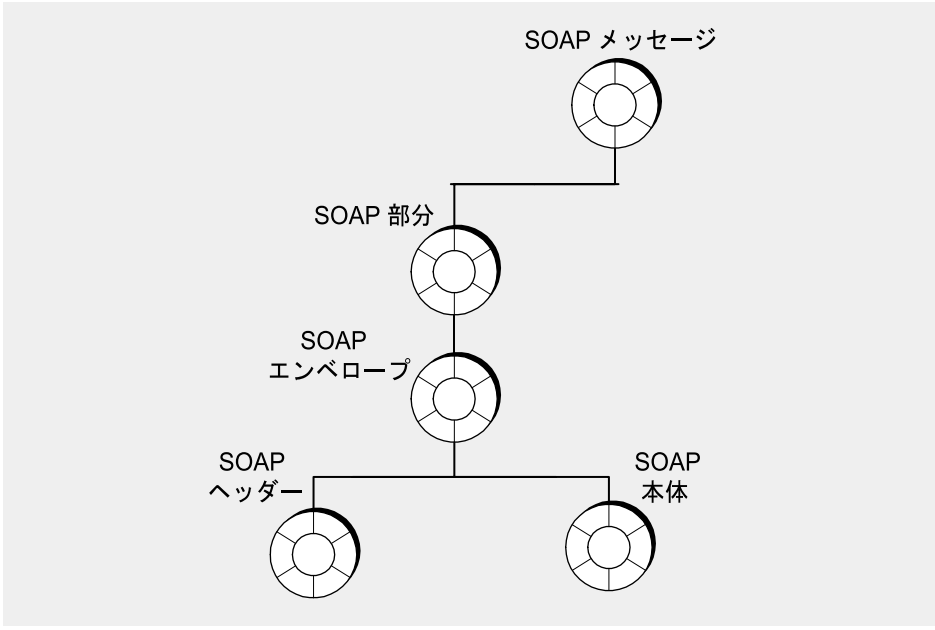
3. メッセージの作成に使用する `MessageFactory` オブジェクトを取得します。

   ```
   MessageFactory myMsgFct = MessageFactory.newInstance();
   ```

4. メッセージファクトリを使ってメッセージを作成します。

   ```
   SOAPMessage message = myMsgFct.createMessage();
   ```

   作成されたメッセージには、次の図に示す部分がすべて含まれます。



この時点では、メッセージのコンテンツはありません。コンテンツを追加するには、SOAP 本体要素を作成し、その名前とコンテンツを定義して、SOAP 本体に追加する必要があります。

メッセージの各部分にアクセスするには、親要素上で get メソッドを呼び出して子要素を取得することにより、ツリー内を移動する必要があります。たとえば、SOAP 本体に移動するには、SOAP 部分と SOAP エンベロープを取得することから開始します。

```
SOAPPart mySPart = message.getSOAPPart();

SOAPEnvelope myEnvp = mySPart.getEnvelope();
```

5. ここで myEnvp オブジェクトから本体要素を取得できます。

```
SOAPBody body = myEnvp.getBody();
```

本体要素に追加した子要素はメッセージのコンテンツを定義します。同じようにヘッダーに SOAP コンテンツを追加できます。

6. SOAP 本体 (またはヘッダー) に要素を追加するときは、まず envelope.createName メソッドを呼び出し、その要素の名前を作成します。このメソッドが返す Name オブジェクトを、本体要素 (またはヘッダー要素) を作成するメソッドにパラメータとして渡します。

```
Name bodyName = envelope.createName("GetLastTradePrice", "m",
                                    "http://eztrade.com")
```

```
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

7. ここで、gltp 要素に追加するもう一つの本体要素を作成します。

```
Name myContent = envelope.createName("symbol");

SOAPElement mySymbol = gltp.addChildElement(myContent);
```

8. これで、本体要素 mySymbol のデータを定義できるようになります。

```
mySymbol.addTextNode("SUNW");
```

生成される SOAP メッセージオブジェクトは、次の XML スキーマと同等になります。

```
<SOAP-ENV:Envelope
  xmlns:SOAPENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="http://eztrade.com">
        <symbol>SUNW</symbol>
      </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

9. メッセージを送信したり、メッセージに書き込みを加えたりするたびに、その内容が自動保存されます。ただし、受信済みのメッセージや送信済みのメッセージに変更を加えた場合は、その内容をすべて保存して、手作業でメッセージを更新する必要があります。たとえば、次のように指定します。

```
message.saveChanges();
```

10. メッセージを送信する前に、送信先エンドポイントの URL を持つ URLEndpoint オブジェクトを作成する必要があります。ただし、メッセージヘッダーにアドレス指定情報を追加するようなプロファイルを使用する場合、この処理は不要です。

```
URLEndpoint endPt = new
                    URLEndpoint("http://eztrade.com//quotes");
```

11. ここで、メッセージを送信できます。

```
SOAPMessage reply = myCon.call(message, endPt);
```

同じコネクションで、リプライメッセージ (reply) が受信されます。

12. 最後に、不要な SOAPConnection オブジェクトを終了します。

```
myCon.close();
```

SOAP サービスの作成

SOAP サービスは、SOAP メッセージの最終的な送信先になります。現時点では、サーブレットとして実装する必要があります。独自のサーブレットを作成できますが、必要に応じて、soap.messaging パッケージ内に提供される JAXMServlet クラスを拡張することもできます。この項では、JAXMServlet クラスを使って SOAP サービスを作成する方法を説明します。

サーブレットは、ReqRespListener インタフェースか OneWayListener インタフェースを実装する必要があります。ReqRespListener が応答を要求するという点を除けば、2 つのインタフェースは同じです。

どちらのインタフェースを使用する場合でも、onMessage(SOAPMsg) メソッドを実装する必要があります。JAXMservlet は、HTTP POST メソッドでメッセージを受信したあと、onMessage を呼び出します。この場合、着信メッセージを SOAP メッセージに変換する doPost() メソッドを実装する必要はありません。

コード例 5-4 は、JAXM サーブレットユーティリティクラスを使用する SOAP サービスの基本構造です。

コード例 5-4 メッセージコンシューマのスケルトン

```
public class MyServlet extends JAXMServlet implements
                  ReqRespListener
{
  public SOAPMessage onMessage(SOAP Message msg)
  { // 処理メッセージ
  }
}
```

コード例 5-5 は、簡単な ping メッセージサービスの例です。

コード例 5-5 簡単な ping メッセージサービス

```
public class SOAPEchoServlet extends JAXMServlet
                  implements ReqRespListener{

  public SOAPMessage onMessage(SOAPMessage mySoapMessage) {
  return mySoapMessage
  }
}
```

表 5-3 では、JAXM サーブレットが使用するメソッドについて説明します。独自のサーブレットを作成する場合は、同様の処理を行うメソッドを用意する必要があります。JAXMServlet を拡張する場合は、Init メソッドと SetMessageFactory メソッドをオーバーライドして、onMessage メソッドを実装しなければなりません。

表 5-3　JAXMServlet メソッド

メソッド	説明
void init (ServletConfig)	ServletConfig オブジェクトを親オブジェクトのコンストラクタに渡し、デフォルトの messageFactory オブジェクトを作成する 特定のプロファイルに従って着信メッセージを構築したい場合は、SetMessageFactory メソッドを呼び出して、SOAP メッセージの構築時に使用されるプロファイルを指定する必要がある
void doPost (HTTPRequest, HTTPResponse	HTTP 要求の本体を取得し、デフォルトまたは特定の MessageFactory プロファイルに従って SOAP メッセージを作成する SOAP メッセージをパラメータとして渡して、適切なリスナーの onMessage() メソッドを呼び出す このメソッドはオバーライドしないようにする
void setMessageFactory (MessageFactory)	MessageFactory オブジェクトを設定する。このオブジェクトを使って、onMessage メソッドに渡される SOAP メッセージを作成する
MimeHeaders getHeaders (HTTPRequest)	特定の HTTPRequest オブジェクト内にヘッダーを格納する MimeHeaders オブジェクトを返す
void putHeaders (mimeHeaders, HTTPresponse)	特定の HTTPResponse オブジェクトに特定の MimeHeaders オブジェクト内のヘッダーを設定する
onMessage (SOAPMesssage)	SOAP メッセージの受信時にサーブレットによって呼び出されるユーザ定義のメソッド。通常、このメソッドは、受け取った SOAP メッセージを逆アセンブルし、クライアントに応答を返す必要がある (サーブレットが ReqRespListener インタフェースを実装している場合)

メッセージの逆アセンブル

onMessage メソッドは、サーブレットから受け取った SOAP メッセージを逆アセンブルし、そのコンテンツを正しく処理する必要があります。メッセージの処理中に問題が発生した場合、サービスは、SOAP 障害オブジェクトを作成し、クライアントに送信する必要があります (詳細は、110 ページの「SOAP 障害の処理」を参照)。

SOAP メッセージ処理では、本体要素の場所を特定してそのコンテンツを処理するとともに、ヘッダーも操作する必要があります。次のサンプルコードでは、onMessage メソッドの本体に含まれる SOAP メッセージを逆アセンブルします。基本的に、SOAP メッセージの解析には、Document Object Model (DOM) API を使用する必要があります。

DOM API の詳細は、http://xml.coverpages.org/dom.html を参照してください。

コード例 5-6 SOAP メッセージの処理

```
{http://xml.coverpages.org/dom.html
  SOAPEnvelope env = reply.getSOAPPart().getEnvelope();
  SOAPBody sb = env.getBody();

  // 検索中の XElement に新しい Name オブジェクトを作成
  Name ElName = env.createName("XElement");

  //XElement の名前で子要素を取得
  Iterator it = sb.getChildElements( ElName );

  // 最初に一致した子要素を取得
  // 1 つしか存在しない
  SOAPBodyElement sbe = (SOAPBodyElement) it.next();

  //XElement の値を取得
  MyValue =   sbe.getValue();
}
```

添付ファイルの処理

SOAP メッセージには、添付ファイルを付けることができます。添付ファイルの作成および追加方法を示すサンプルコードについては、119 ページのコード例 5-7 を参照してください。添付ファイルの受信および処理方法を示すサンプルコードについては、121 ページのコード例 5-8 を参照してください。

添付ファイルの処理には、Java Activation Framework API を使用する必要があります。詳細は、http://java.sun.com/products/javabeans/glasgow/jaf.html を参照してください。

メッセージへの応答

メッセージに応答するときは、サーバサイドでクライアントの役割をします。

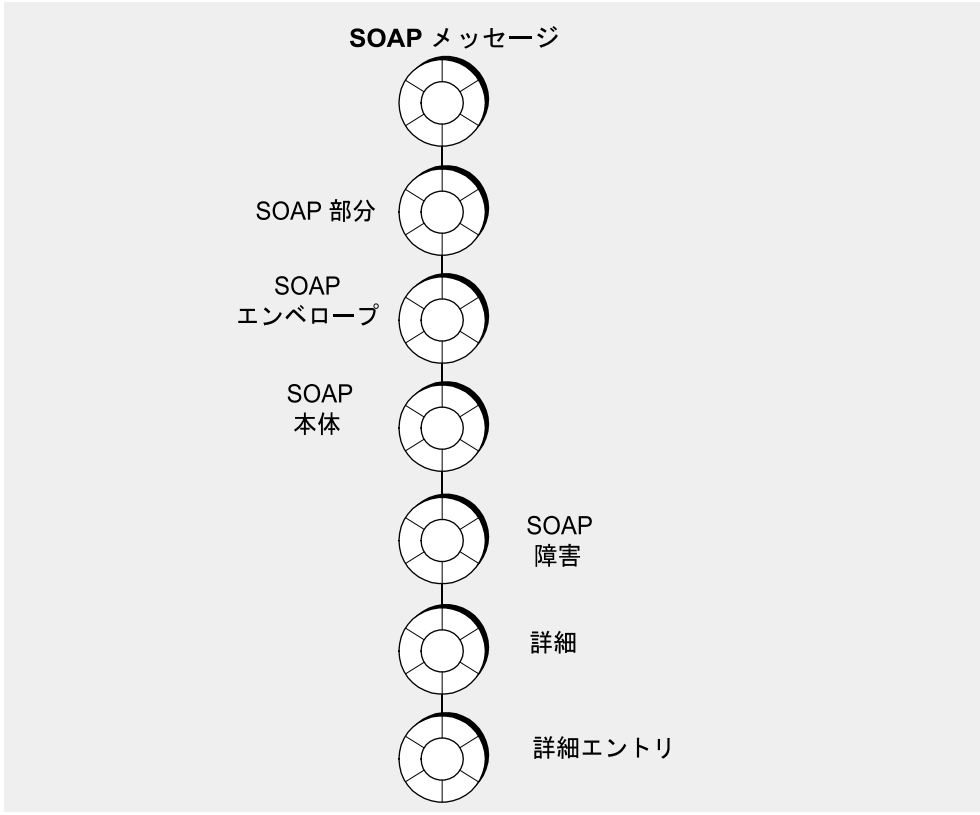
SOAP 障害の処理

サーバサイドコードは、SOAP 障害オブジェクトを使って、要求の非整列化時、メッセージの処理時、応答の整列化時にサーバサイドで発生したエラーを処理する必要があります。`SOAPFault` インタフェースは `SOAPBodyElement` インタフェースを拡張します。

SOAP メッセージは、サーバサイド上でエラー報告を行う特殊な要素および形式を備えています。要求処理時に発生したエラーを報告するため、SOAP メッセージ本体に SOAP 障害要素が含まれることがあります。`SOAPFault` オブジェクトを含む SOAP メッセージは、サーバサイドで作成され、サーバからクライアントに返されます。この SOAP メッセージは、メッセージの送信元に対して、予想外の動作をすべて報告します。

図 5-8 に示されるように、SOAP 障害オブジェクトは、SOAP 本体の子オブジェクトとして、SOAP メッセージオブジェクトに含まれています。詳細オブジェクトおよび詳細エントリオブジェクトは、受信済みのメッセージの本体の形式が不正である場合と、受信済みのメッセージの本体に不適切なデータが含まれている場合以外は不要です。この場合、詳細エントリオブジェクトは、不正な形式のデータを説明するために使用されます。

図 5-8　SOAP 障害要素



SOAP 障害要素は、次の 4 つのサブ要素を定義します。

- `faultcode`

  エラーを識別するコード (修飾名)。ソフトウェアは、このコードを使って、障害を検出するアルゴリズム的メカニズムを提供します。事前定義済みの障害コードは、112 ページの表 5-4 のとおりです。この要素は必須要素です。

- `faultstring`

  障害コードで検出された障害を説明する文字列。この要素は、人間が理解できる形式でエラーの説明を提供します。この要素は必須要素です。

- `faultactor`

  障害のソースを指定する URI。メッセージのパス間で障害の原因となったアクター。メッセージが中間ノードを経由せずに最終的な送信先に送信される場合、この要素は不要です。中間ノードで障害が発生した場合、その障害には `faultactor` 要素が含まれることになります。

- `detail`

  この要素は、本体要素に関連した特殊な情報を伝送します。本体要素のコンテンツが正常に処理されなかった場合に生成されます。つまり、この要素が存在しなければ、クライアントは、本体要素が処理されたと見なします。この要素は不正な形式のペイロード以外のエラーでは不要ですが、クライアントに追加情報を提供するために使用することもできます。

事前定義済みの障害コード

SOAP 仕様には、4 つの事前定義済み `faultcode` 値が規定されています。これらのネームスペース識別子は、`http://schemas.xmlsoap.org/soap/envelope/` です。

表 5-4 SOAP の faultcode 値

faultcode 名	意味
`VersionMismatch`	処理側が、SOAP エンベロープ要素のネームスペースが無効である、つまり SOAP エンベロープのネームスペースが `http://schemas.xmlsoap.org/soap/envelope/` 以外であることを検出した
`MustUnderstand`	受信側が SOAP ヘッダー要素の直系の子要素を認識しなかった、または適切に処理できなかった。この要素の `mustUnderstand` 属性の値は 1 (true) に設定されている
`Client`	メッセージの形式が不正であるか、メッセージに適切な情報が含まれていなかった。たとえば、メッセージに適切な認証情報または支払い情報が含まれていなかった。クライアントは、このコードから、メッセージを再送信する前に変更が必要であると判断する 戻り値としてこのコードが返される場合は、`SOAPFault` オブジェクトに、不正な形式のメッセージに関する追加情報を提供する `detailEntry` オブジェクトが含まれていると考えられる
`Server`	コンテンツとコネクションしていないためメッセージが処理されなかった。たとえば、あるメッセージハンドラがアップストリームの別のメッセージハンドラと通信できなかったため、応答しなかった。または、サーバからアクセスする必要があるデータベースがダウンしている。クライアントは、このエラーから、しばらくしたら転送に成功する可能性があると判断する

これらの標準障害コードは、障害のクラスを表します。これらのコードは、末尾にピリオドと名前を追加することで拡張できます。たとえば、`Server.OutOfMemory` コード、`Server.Down` コードなどと定義できます。

SOAP 障害の定義

JAXM では、SOAPFault オブジェクトのメソッドを使って、faultcode、faultstring、faultactor の値を指定できます。次のコードでは、SOAP 障害オブジェクトを作成し、faultcode、faultstring、faultactor の各属性を設定します。

```
SOAPFault fault;
reply = factory.createMessage();
envp = reply.getSOAPPart().getEnvelope(true);
someBody = envp.getBody();
fault = someBody.addFault():
fault.setFaultCode("Server");
fault.setFaultString("Some Server Error");
fault.setFaultActor(http://xxx.me.com/list/endpoint.esp/)
reply.saveChanges();
```

サーバは、サーバエラーが発生した場合、着信 SOAP メッセージへの応答としてこのオブジェクトを返すことができます。

次のサンプルコードは、詳細および詳細エントリオブジェクトを定義する方法を示します。詳細エントリオブジェクトの名前を作成する必要があります。

```
SOAPFault fault = someBody.addFault();
fault.setFaultCode("Server");
fault.setFaultActor("http://foo.com/uri");
fault.setFaultString ("Unkown error");
Detail myDetail = fault.addDetail();
Name bodyName = envelope.createName("GetLastTradePrice", "m",
    "Someuri")).addTextNode("the message cannot contain
    the string //");
reply.saveChanges();
```

SOAP と MQ の統合

この節では、SOAP ペイロードを含む JMS メッセージの送受信および処理方法を説明します。

MQ には、JMS API を使って SOAP メッセージを送受信するときに役立つユーティリティが付属しています。このユーティリティでは、MQ の高信頼メッセージングサービスを活用するため、SOAP メッセージを JMS メッセージに変換することができます。さらに、受信側では、JMS メッセージを SOAP メッセージに変換し直し、JAXM API を使って処理することができます。

SOAP ペイロードを含む JMS メッセージを送受信または処理するには、次の処理を行います。

- `com.sun.messaging.xml.MessageTransformer` ライブラリをインポートします。このユーティリティのメソッドを使って、SOAP メッセージと JMS メッセージ間の変換を行います。
- SOAP メッセージを転送する前に、`MessageTransformer.SOAPMessageIntoJMSMessage` メソッドを呼び出す必要があります。このメソッドは SOAP メッセージを JMS メッセージに変換します。変換後の JMS メッセージを通常の JMS メッセージと同様に送信します。プログラムの簡便化のため、SOAP メッセージの受信専用の送信先を選択しておくことをお勧めします。つまり、SOAP メッセージの送信先となる特定のキューまたはトピックを作成し、この送信先には SOAP メッセージだけを送信するようにします。

  SOAP メッセージを JMS メッセージに変換するときは、次のような呼び出しを行う必要があります。

  ```
  Message myMsg= MessageTransformer.SOAPMessageIntoJMSMessage
                          (SOAPMessage, Session);
  ```

  `Session` 引数は、`Message` をプロデュースする時に使用するセッションを指定します。
- 受信側では、通常の JMS メッセージの場合と同様にして、SOAP ペイロードを含む JMS メッセージを取得します。その後、`MessageTransformer.SOAPMessageFromJMSMessage` ユーティリティを使用して SOAP メッセージを抽出し、JAXM を使って SOAP メッセージを逆アセンブルしたあと、その他の処理を行います。たとえば、SOAPMessage を取得する場合は、次のような呼び出しを行います。

  ```
  SOAPMessage myMsg=  MessageTransformer.SOAPMessageFromJMSMessage
                            (Message, MessageFactory);
  ```

  `MessageFactory` 引数は、ユーティリティが特定の JMS `Message` から `SOAPMessage` を構築するとき使用するメッセージファクトリを指定します。

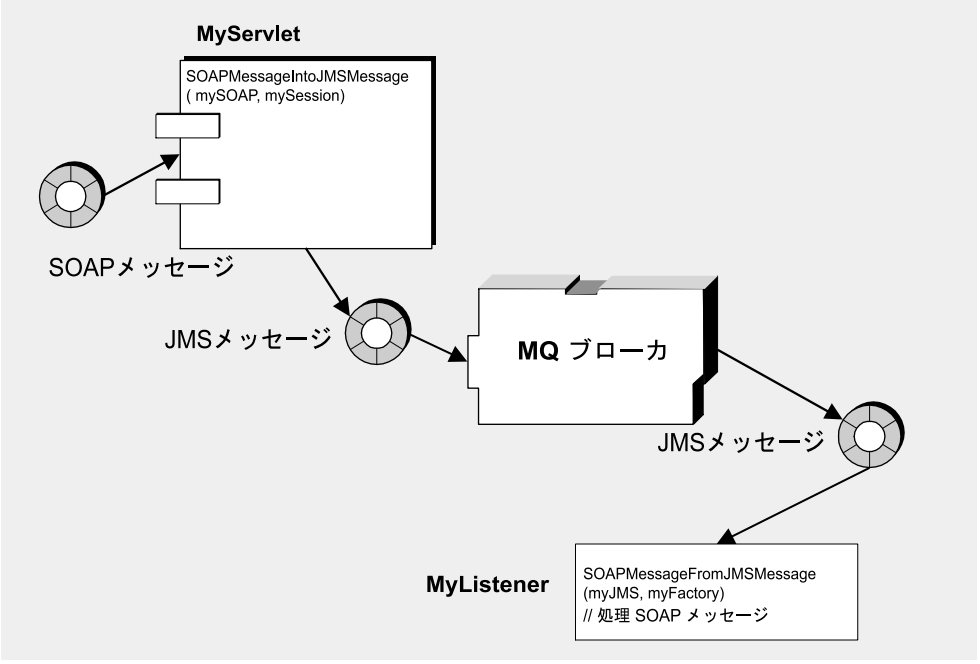
次に、このプロセスを表す使用例とサンプルコードの例をいくつか紹介します。

例 1 : SOAP 処理の延期

最初の例 (図 5-9) では、着信 SOAP メッセージはサーブレットによって受信されます。SOAP メッセージを受信したサーブレット `MyServlet` は、`MessageTransformer` ユーティリティを使って、このメッセージを JMS メッセージに変換し、高い信頼性でアプリケーションに転送します。このアプリケーションは、受信した JMS メッセージを SOAP メッセージに再変換し、SOAP メッセージのコンテンツを処理します。

サーブレットが SOAP メッセージを受信する仕組みについては、107 ページの「SOAP サービスの作成」を参照してください。

図 5-9　SOAP 処理の延期



➤ SOAP メッセージを JMS メッセージに変換して、JMS メッセージを送信するには

1. ConnectionFactory オブジェクトをインスタンス化し、その属性を設定します。次に例を示します。

```
QueueConnectionFactory myQConnFact =
        new com.sun.messaging.QueueConnectionFactory();
```

2. ConnectionFactory オブジェクトを使って Connection オブジェクトを作成します。

```
QueueConnection myQConn =
        myQConnFact.createQueueConnection();
```

3. Connection オブジェクトを使って Session オブジェクトを作成します。

```
QueueSession myQSess = myQConn.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
```

4. MQ メッセージサービス内の物理的な送信先に対応する MQ Destination 管理対象オブジェクトをインスタンス化します。この例の場合、管理対象オブジェクトは mySOAPQueue、このオブジェクトが参照する物理的な送信先は myPSOAPQ です。

```
Queue mySOAPQueue = new com.sun.messaging.Queue("myPSOAPQ");
```

5. MessageTransformer ユーティリティを使って SOAP メッセージを JMS メッセージに変換します。なお、ここでは、MySOAPMsg という名前の SOAP メッセージを使用するものとします。

```
Message MyJMS = MessageTransformer.SOAPMessageIntoJMSMessage
                                (MySOAPMsg, MyQSess);
```

6. QueueSender メッセージプロデューサを作成します。

   このメッセージプロデューサは mySOAPQueue に関連付けられており、キュー送信先 myPSOAPQ にメッセージを送信する際使用されます。

```
QueueSender myQueueSender = myQSess.createSender(mySOAPQueue);
```

7. キューへメッセージを送信します。

```
myQueueSender.send(myJMS);
```

➤ 受信した JMS メッセージを SOAP メッセージに変換して処理するには

1. ConnectionFactory オブジェクトをインスタンス化し、その属性値を設定します。

```
QueueConnectioFactory myQConnFact = new
        com.sun.messaging.QueueConnectionFactory();
```

2. ConnectionFactory オブジェクトを使って Connection オブジェクトを作成します。

```
QueueConnection myQConn = myQConnFact.createQueueConnection();
```

3. Connection オブジェクトを使って 1 つまたは複数の Session オブジェクトを作成します。

```
QueueSession myRQSess = myQConn.createQueueSession(false,
        session.AUTO_ACKNOWLEDGE);
```

4. Destination オブジェクトをインスタンス化し、その名前属性を設定します。

```
Queue myRQueue = new com.sun.messaging.Queue("mySOAPQ");
```

5. Session オブジェクトと Destination オブジェクトを使って、必要な MessageConsumer オブジェクトを作成します。

```
QueueReceiver myQueueReceiver =
   myRQSess.createReceiver(myRQueue);
```

6. 必要に応じて、MessageListener オブジェクトをインスタンス化し、MessageConsumer オブジェクトに登録します。

7. 手順 2 で作成した QueueConnection を起動します。クライアントによるコンシュームのためのメッセージは、確立されているコネクションを必ず経由して配信されます。

```
myQConn.start();
```

8. キューからメッセージを受信します。

   次のコードは、メッセージの同期コンシュームの例です。

```
Message myJMS = myQueueReceiver.receive();
```

9. Message Transformer を使って JMS メッセージを SOAP メッセージに再変換します。

```
SOAPMessage MySoap =
       MessageTransformer.SOAPMessageFromJMSMessage
         (myJMS, MyMsgFactory);
```

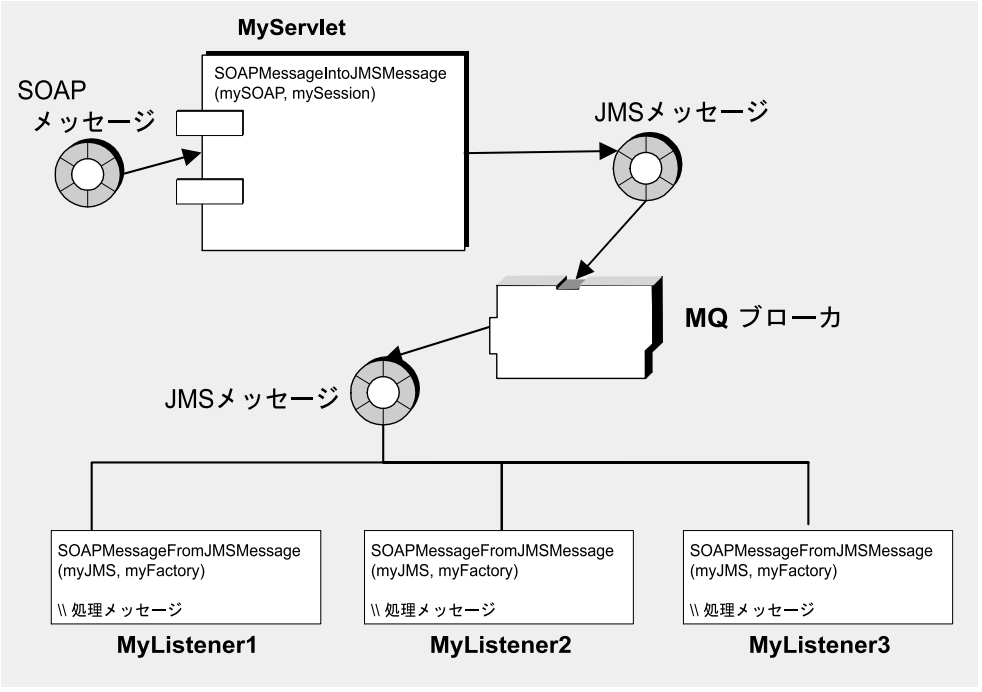
   MessageFactory 引数に NULL を指定した場合、デフォルトのメッセージファクトリを使って SOAP メッセージが構築されます。

10. 将来の処理に備えて、SOAP メッセージを逆アセンブルします。詳細は、89 ページの「SOAP メッセージオブジェクト」を参照してください。

例 2 : SOAP メッセージのパブリッシュ

次の例 (図 5-10) では、着信 SOAP メッセージはサーブレットによって受信されます。サーブレットは、SOAP メッセージを JMS メッセージとしてパッケージ化し、トピックに転送 (高信頼) します。このトピックに加入している各アプリケーションは、JMS メッセージを受信し、SOAP メッセージに再変換して、そのコンテンツを処理します。

図 5-10　SOAP メッセージのパブリッシュ



JMS メッセージをキューではなくトピックに送信する点を除けば、この処理は、前の例とまったく同じ方法で行われます。119 ページのコード例 5-7 に、MQ を使って SOAP メッセージをパブリッシュする例を示します。

サンプルコード

この項では、2 つのサンプルコードを取り上げて説明します。最初に取り上げるのは、SOAP ペイロードを含む JMS メッセージを送信する例です。次に取り上げるのは、JMS/SOAP メッセージを受信し、SOAP メッセージを処理する例です。

コード例 5-7 では、JMS API、JAXM API、JAF API を使って、添付ファイル付きの SOAP メッセージを JMS メッセージのペイロードとして送信します。SendSOAPMessageWithJMS のコードには、次のメソッドが含まれています。

- init メソッドを呼び出して、メッセージのパブリッシュに必要なすべての JMS オブジェクトを初期化するコンストラクタ
- SOAP メッセージと添付ファイルを作成し、SOAP メッセージを JMS メッセージに変換して、この JMS メッセージをパブリッシュする send メソッド
- コネクションを終了する close メソッド
- send メソッドと close メソッドを呼び出す main メソッド

コード例 5-7 SOAP ペイロードを持つ JMS メッセージの送信

```
//SOAP メッセージの構築に必要なライブラリ
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.Name

// 添付ファイルの操作に必要なライブラリ (Java Activation Framework API)
import java.net.URL;
import javax.activation.DataHandler;

//SOAP メッセージを JMS メッセージに変換して送信するために必要なライブラリ
import com.sun.messaging.xml.MessageTransformer;
import com.sun.messaging.BasicConnectionFactory;

//JMS コネクションを設定してメッセージを送信するために必要なライブラリ
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.JMSException;
import javax.jms.Session;
import javax.jms.Message;
import javax.jms.TopicSession;
import javax.jms.Topic;
import javax.jms.TopicPublisher;

//SOAP ペイロードを含む JMS メッセージを送信するクラスを定義する
public class SendSOAPMessageWithJMS{
```

コード例 5-7 SOAP ペイロードを持つ JMS メッセージの送信 (続き)

```
  TopicConnectionFactory tcf = null;
  TopicConnection tc = null;
  TopicSession session = null;
  Topic topic = null;
  TopicPublisher publisher = null;

// デフォルトのコンストラクタメソッド
public SendSOAPMessageWithJMS(String topicName){
  init(topicName);
  }

//JMS のコネクション、セッション、トピックおよびパブリッシャを初期化するメソッド
public void init(String topicName) {
  try {
  tcf = new com.sun.messaging.TopicConnectionFactory();
  tc = tcf.createTopicConnection();
  session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
  topic = session.createTopic(topicName);
  publisher = session.createPublisher(topic);
  }

//SOAP/JMS メッセージを作成および送信するメソッド
public void send() throws Exception{
  MessageFactory mf = MessageFactory.newInstance(); // デフォルトファクトリの作成
  SOAPMessage soapMessage=mfcreateMessage(); //SOAP メッセージオブジェクトの作成
  SOAPPart soapPart = soapMessage.getSOAPPart();// 本体へのドリルダウンの開始
  SOAPEnvelope soapEnvelope = soapPart.getEnvelope(); // 最初にエンベロープ
  SOAPBody soapBody = soapEnvelope.getBody();
  Name myName = soapEnvelope.createName("HelloWorld", "hw",
                  http;//www.sun.com/imq'); // 本体要素の名前
  SOAPElement element = soapBody.addChildElement(myName); // 本体要素を追加
  element.addTextNode("Welcome to SUnOne Web Services."); // テキスト値を追加

  //Java Framework Activation API で添付ファイルを作成する
  URL url = new URL("http://java.sun.com/webservices/");
  DataHandler dh = new DataHnadler (url);
  AttachmentPart ap = soapMessage.createAttachmentPart(dh);

  // コンテンツタイプと ID を設定する
  ap.setContentType("text/html");
  ap.setContentID('cid-001");

  //SOAP メッセージに添付ファイルを追加する
  soapMessage.addAttachmentPart(ap);
  soapMessage.saveChanges();

  //SOAP メッセージを JMS メッセージに変換する
  Message m = MessageTransformer.SOAPMessageIntoJMSMessage(soapMessage,
                  session);

//JMS メッセージをパブリッシュする
  publisher.publish(m);
```

コード例 5-7 SOAP ペイロードを持つ JMS メッセージの送信 (続き)

```
//JMS コネクションを終了する
  public void close() throws JMSException {
  tc.close();
  }

//SOAP メッセージを JMS で送信するメインプログラム
public static void main (String[] args) {
  try {
    String topicName = System.getProperty("TopicName");
    if(topicName == null) {
      topicName = "test";
    }

    SendSOAPMEssageWithJMS ssm = new SendSOAPMEssageWithJMS(topicName);
    ssm.send();
    ssm.close();
  }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

コード例 5-8 では、JMS API、JAXM API、DOM API を使って、添付ファイル付きの SOAP メッセージを JMS メッセージのペイロードとして受信します。
`ReceiveSOAPMessageWithJMS` のコードには、次のメソッドが含まれています。

- `init` メソッドを呼び出して、メッセージの受信に必要なすべての JMS オブジェクトを初期化するコンストラクタ
- メッセージを配信し、リスナーに呼び出される `onMessage` メソッド。`onMessage` メソッドは、Message Transformer ユーティリティを使用し JMS メッセージを SOAP メッセージに変換し、JAXM API を使って SOAP 本体を処理し、JAXM および DOM API を使ってメッセージの添付ファイルを処理する
- `ReceiveSOAPMessageWithJMS` クラスを初期化する `main` メソッド

コード例 5-8 SOAP ペイロードを持つ JMS メッセージの受信

```
//SOAP の処理をサポートするライブラリ
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.AttachmentPart

//JMS から SOAP への transformer を含むライブラリ
import com.sun.messaging.xml.MessageTransformer;
```

コード例 5-8　　SOAP ペイロードを持つ JMS メッセージの受信 (続き)

```
//JMS メッセージングサポート用のライブラリ
import com.sun.messaging.TopicConnectionFactory

//JMS メッセージング用のインターフェース
import javax.jms.MessageListener;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.JMSException;
import javax.jms.TopicSubscriber

// 添付ファイル部分のパースをサポートするライブラリ (DOM API から )
import java.util.iterator;

public class ReceiveSOAPMessageWithJMS implements MessageListener{
  TopicConnectionFactory tcf = null;
  TopicConnection tc = null;
  TopicSession session = null;
  Topic topic = null;
  TopicSubscriber subscriber = null;
  MessageFactory messageFactory = null;

//デフォルトのコンストラクタ
public ReceiveSOAPMessageWithJMS(String topicName) {
  init(topicName);
}
//Set up JMS connection and related objects
public void init(String topicName){
  try {
    //デフォルトの SOAP メッセージファクトリを構築する
    messageFactory = MessageFactory.newInstance();

    //JMS の設定
    tcf = new. com.sun.messaging.TopicConnectionFactory();
    tc = tcf.createTopicConnection();
    session = tc.createTopicSesstion(false, Session.AUTO_ACKNOWLEDGE);
    topic = session.createTopic(topicName);
    subscriber = session.createSubscriber(topic);
    subscriber.setMessageListener(this);
    tc.start();

    System.out.println("ready to receive SOAP m essages...");
  }catch (Exception jmse){
    jmse.printStackTrace();
    }
  }

//JMS メッセージが onMessage メソッドに配信される
public void onMessage(Message message){
  try {
    //JMS メッセージを SOAP メッセージに変換する
```

コード例 5-8 SOAP ペイロードを持つ JMS メッセージの受信 (続き)

```
    SOAPMessage soapMessage = MessageTransformer.SOAPMessageFromJMSMessage
              (message, messageFactory);


    // 添付ファイル数を印刷
    System.out.println("message received!Attachment counts:
              " + soapMessage.countAttachments());

    //SOAP メッセージの添付ファイル部分を取得
    Iterator iterator = soapMessage.getAttachments();
    while (iterator.hasNext()) {
      // 次の添付ファイルを取得
      AttachmentPart ap = (AttachmentPart) iterator.next();

      // コンテンツタイプを取得
      String contentType = ap.getContentType();
      System.out.println("content type:" + conent TYpe);

      // コンテンツ ID を取得
      String contentID = ap.getContentID();
      System.out.println("content Id:"+ contentId);

      // テキストであるかを確認する
      if(contentType.indexOf"text")>=0 {
        // 添付ファイルがテキストであれば文字列のコンテンツを取得して印刷する
        String content = (String) ap.getContent();
        System.outprintln("*** attachment content:" + content);
      }
    }
  }catch (Exception e) {
    e.printStackTrace();
  }
}

// 受信先のサンプルを開始するメインメソッド
public static void main (String[] args){
  try {
    String topicName = System.getProperty("TopicName");
    if( topicName == null) {
      topicName = "test";
    }
    ReceiveSOAPMessageWithJMS rsm = new ReceiveSOAPMessageWithJMS(topicName);
  }catch (Exception e) {
    e.printStackTrace();
    }
  }
}
```

付録 A

管理対象オブジェクトの属性

この付録では、ConnectionFactory、XAConnectionFactory、および送信先の管理対象オブジェクトの属性の一覧表を提供します。

ConnectionFactory 管理対象オブジェクト

表 A-1 は、ConnectionFactory と XAConnectionFactory の管理対象オブジェクトの設定可能なプロパティの概要を示します。参照しやすいようにアルファベット順で表示します。これらの属性の機能別カテゴリの分類とそれぞれの説明については、69 ページの「MQ クライアントランタイムの設定可能なプロパティ」を参照してください。

表 A-1 コネクションファクトリ属性

属性 / プロパティ名	データ型	デフォルト値	参照先
imqAckOnAcknowledge	文字列型	未指定	77 ページの表 4-6
imqAckOnProduce	文字列型	未指定	77 ページの表 4-6
imqAckTimeout	文字列型	0 milliseconds	77 ページの表 4-6
imqBrokerHostName	文字列型	localhost	70 ページの表 4-1
imqBrokerHostPort	文字列型	7676	70 ページの表 4-1
imqBrokerServicePort	文字列型	0	70 ページの表 4-1
imqConfiguredClientID	文字列型	未指定	73 ページの表 4-3
inqConnectionType	文字列型	TCP	70 ページの表 4-1
imqConnectionURL	文字列型	http://localhost/imq/tunnel	70 ページの表 4-1
imqDefaultPassword	文字列型	guest	73 ページの表 4-3

表 A-1 コネクションファクトリ属性 (続き)

属性 / プロパティ名	データ型	デフォルト値	参照先
imqDefaultUsername	文字列型	guest	73 ページの表 4-3
imqDisableSetClientID	文字列型	false	73 ページの表 4-3
imqFlowControlCount	文字列型	100	75 ページの表 4-5
imqFlowControlIsLimited	文字列型	false	75 ページの表 4-5
imqFlowControlLimit	文字列型	1000	75 ページの表 4-5
imqJMSDeliveryMode	文字列型	2 (持続的)	74 ページの表 4-4
imqJMSExpiration	整数型	0 (有効期限なし)	74 ページの表 4-4
imqJMSPriority	整数型	4 (標準)	74 ページの表 4-4
imqLoadMaxToServerSession	文字列型	true	78 ページの表 4-8
imqOverrideJMSDeliveryMode	ブール型	false	74 ページの表 4-4
imqOverrideJMSExpiration	ブール型	false	74 ページの表 4-4
imqOverrideJMSPriority	ブール型	false	74 ページの表 4-4
imqOverrideJMSHeadersTo TemporaryDestinations	ブール型	false	74 ページの表 4-4
imqQueueBrowserMaxMessages PerRetrieve	文字列型	1000	78 ページの表 4-7
imqQueueBrowserRetrieveTimeout	文字列型	60,000 milliseconds	78 ページの表 4-7
imqReconnect	文字列型	false	72 ページの表 4-2
imqReconnectDelay	文字列型	30,000 milliseconds	72 ページの表 4-2
imqReconnectRetries	文字列型	0	72 ページの表 4-2
imqSetJMSXAppID	文字列型	false	78 ページの表 4-8
imqSetJMSXConsumerTXID	文字列型	false	78 ページの表 4-8
imqSetJMSXProducerTXID	文字列型	false	78 ページの表 4-8
imqSetJMSXRcvTimestamp	文字列型	false	78 ページの表 4-8
imqSetJMSXUserID	文字列型	false	78 ページの表 4-8
imqSSLIsHostTrusted	文字列型	true	70 ページの表 4-1

ConnectionFactory 管理対象オブジェクトの使用方法は、第 3 章「管理対象オブジェクトの使用」を参照してください。

送信先管理対象オブジェクト

送信先管理対象オブジェクトは、公的に名前の付けられた送信先オブジェクトが対応するブローカの物理的な送信先 (キュー、またはトピック) を表しています。その唯一の属性は、物理的送信先の内部プロバイダ固有名です。送信先オブジェクトを作成することによって、クライアントの MessageConsumer オブジェクトまたは MessageProducer オブジェクト、あるいはその両方が、該当する物理的な送信先にアクセスできるようになります。

表 A-2　送信先の属性

属性 / プロパティ名	データ型	デフォルト値
imqDestinationDescription	文字列型	送信先オブジェクトの説明
imqDestinationName	文字列型 *	Untitled_Destination_Object

* 送信先名には、英数字 (空白文字は含まない) だけを使用できます。送信先名は、英字や "_" または "$"、あるいはその両方の文字で開始する必要があります。

送信先管理対象オブジェクトの詳細については、第 3 章「管理対象オブジェクトの使用」を参照してください。

索引

J

M

O

P

せ

そ

つ

と

ね

は

ふ

ほ

め

ゆ

ら

り