MQSeries® Everyplace

# プログラミング・ガイド

*バージョン 1*

SC88-8654-00
(英文原典：SC34-5845-00)

MQSeries® Everyplace

# プログラミング・ガイド

*バージョン 1*

SC88-8654-00
(英文原典：SC34-5845-00)

**ご注意!**

この情報およびサポートされている製品をお使いになる前に、 241ページの『付録B. 特記事項』にある一般情報をお読みください。

**ライセンスについての警告**

MQSeries Everyplace バージョン 1 ツールサーキットにより、開発者は MQSeries Everyplace アプリケーションを作成し、それを実行するための環境を作成することができます。

ツールキットのご購入の際のライセンス条件により、それを使用できる環境が決まります。

*MQSeries Everyplace をデバイス (クライアント) として使用するために購入された場合は、その MQSeries Everyplace を使用して* ***MQSeries Everyplace チャネル・マネージャー***、 ***MQSeries Everyplace チャネル・リスナー***、*または* ***MQSeries Everyplace ブリッジ*** *を作成することは****できません****。*

***MQSeries Everyplace チャネル・マネージャー***、 ***MQSeries Everyplace チャネル・リスナー***、*または* ***MQSeries Everyplace ブリッジ*** *の存在により、* ***ゲートウェイ*** *(サーバー) 環境が定義されますが、それにはゲートウェイ・ライセンスが必要です。*

この版は、MQSeries Everyplace バージョン 1、および新版において特に断りのない限り、それ以降のすべてのリリースとモディフィケーションに適用されます。

本書は、随時改訂され、内容は更新されます。最新の版については、MQSeries ファミリー・ライブラリーの Web ページ： `http://www.ibm.com/software/ts/mqseries/library/` をご覧ください。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

`http://www.ibm.com/jp/manuals/main/mail.html`

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

`http://www.ibm.com/jp/manuals/` の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)


| | |
|---|---|
| **原 典：** | SC34-5845-00<br>MQSeries® Everyplace<br>Programming Guide<br>Version 1 |
| **発 行：** | 日本アイ・ビー・エム株式会社 |
| **担 当：** | ナショナル・ランゲージ・サポート |

第1刷 2000.8


この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財) 日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

# 目次

# 本書について

本書は MQSeries Everyplace 製品のプログラミング・ガイドであり、MQSeries Everyplace プログラミング・リファレンス に記述されている MQSeries Everyplace クラス・ライブラリーを使用する方法が記載されています。共通メッセージング・タスクを実行するためにはどのクラスを使用したらよいかを判別するのに役立つ情報が載せられており、多くの場合にサンプル・コードも提供されています。

1ページの『第1章 概要』には、MQSeries Everyplace の概念とコンポーネントに精通していない方のために簡単な概要が示されています。 11ページの『第2章 概説』には、環境の設定およびサンプルを使ったアプリケーションの作成など役立つ情報があります。本書の残りの部分には、MQSeries Everyplace でのプログラミングの様々な面についてのより詳細な情報が記載されています。

本書は、MQSeries Everyplace プログラミング・リファレンス および Java® プログラミングに関する既存の資料またはマニュアルとともに使用することを想定しています。

本書は、継続的に新規または改訂情報で更新されます。最新版は、MQSeries ファミリー・ライブラリー Web ページ
`http://www.ibm.com/software/ts/mqseries/library/` (英語) にアクセスしてください。

## 本書の対象読者

本書は、MQSeries Everyplace システムと他の MQSeries ファミリーのメッセージングおよびキューイング製品のメンバーとの間でセキュア・メッセージを交換する Java ベースの MQSeries Everyplace プログラムを作成するプログラマーを対象としています。

Java 以外の環境用の開発キットが入手可能かどうかについては、MQSeries Web サイト `http://www.ibm.com/software/ts/mqseries/` にアクセスしてください。

## 前提条件となる知識

読者に Java およびオブジェクト指向プログラミング技法の実用的な知識があることを前提としています。

セキュア・メッセージングの概念の初歩の知識があると助けになります。そうでないなら、以下の MQSeries 資料をお読みになることをお勧めします。

- *MQSeries An Introduction to Messaging and Queuing*
- *MQSeries (Windows NT® 版) インストールの手引き V5.1*

上記の資料は、オンライン MQSeries ライブラリーの Book セクションからソフトコピーで入手することができます。 MQSeries Web サイト、URL アドレス
`http://www.ibm.com/software/ts/MQSeries/library/` からライブラリーにアクセスできます。

# 用語

以下に示すのは、本書で使用されている MQSeries Everyplace およびセキュア・メッセージングに特有の用語の一部です。

キュー・マネージャー: この MQSeries Everyplace コンポーネントは、キューのセットおよび他の MQSeries Everyplace キュー・マネージャーへのチャネルのセットを管理します。キュー・マネージャーは、メッセージング機能を実行するために MQSeries Everyplace が使用するインターフェースをパブリッシュします。

クライアント: 1 つのキュー・マネージャーだけでなる、最も単純な MQSeries Everyplace 構成。デフォルトのクライアントは接触不能で、他の MQSeries Everyplace 構成への接続のみ可能です。このクライアントは出力メッセージを「プッシュ」し、自分あてのメッセージを指定されたホーム・サーバーから「プル」します。ただし、クライアントに対等リスナーを組み込むよう定義することは可能で、この場合、他の MQSeries Everyplace 構成から直接クライアントに接続することができます。

チャネル: 2 つの MQSeries Everyplace キュー・マネージャーの間でデータを転送するために使用されるメカニズム。チャネル・オブジェクトは、ネットワークによって使用される基礎トランスポート・プロトコルを覆う論理ラッパーです。このチャネルをアクティブにしたキュー・マネージャーが望む限り、チャネルはオープンされたままになります。ただし、基礎接続に関しては必ずしもそうとは言えず、各ネットワーク・フローごとにオープンとクローズが行われます。

チャネルには認証プログラム、暗号化プログラム、および圧縮プログラムの各特性があります。チャネルを通して転送されたデータすべてには、それに適用された特性が付与されます。たとえば、チャネルが DES 暗号化プログラムと RLE 圧縮プログラムを持つよう定義されると、そのチャネルを通過するデータには DES 暗号化および RLE 圧縮が適用されます。

トランスポーター: トランスポーターは、チャネルとの間でのメッセージの転送および宛先キューを処理するオブジェクトです。トランスポーターはチャネルのどちらか一方の端に付加されます。多数のトランスポーターをチャネルに付加することができます。アクセスされるリモート・キューごとに 1 つのトランスポーターが作成されます。

**注:** チャネルは宛先キュー・マネージャーを指しますが、トランスポーターは宛先キューを指します。

サーバー: この MQSeries Everyplace コンポーネントは、他の MQSeries Everyplace 構成からの複数並行接続を可能にします。サーバーには関連するキュー・マネージャーがあり、このキュー・マネージャーが着信要求へのサービスを提供します。

対等チャネル: 標準の MQSeries Everyplace チャネルは単一方向です。つまりデータは双方向にフローするものの、チャネルを作成したキュー・マネージャーだけがデータの転送を開始できます。この典型的な例は、サーバー構成と通信するクライアント構成です。サーバーは要求を処理し結果を戻します。サーバー構成はクライアントへの通信を開始できず、ただクライアントの要求に応答することしかできません。対等通信は標準チャネルに対しては同一の方法で操作を行いますが、両者が

ともに通信を開始することを許可した場合は例外です。クライアント / サーバーという形で操作するのではなく、チャネルは両者を等しいもの、すなわち対等として扱います。

*対等リスナー:* 対等リスナーは別の MQSeries Everyplace 構成からの着信接続要求を検出し、その接続を確立します。その接続は対等チャネルを使用して開始されていなければなりません。対等リスナーとサーバーの違いは、一度に 1 つの接続しかアクティブにできないので、対等リスナーはチャネル・マネージャーを必要としないということです。

*ホーム・サーバー:* 多くのパーベイシブ MQSeries Everyplace クライアントへは他の MQSeries Everyplace 構成から直接接続可能ではありません。ただし、各 MQSeries Everyplace 構成はホーム・サーバーを定義できます。ホーム・サーバーは、クライアントあてのメッセージをストア・アンド・フォワード (蓄積交換) キューとして知られる特別なキューに格納できる標準 MQSeries Everyplace サーバーです。格納されたメッセージは、クライアントがホーム・サーバーに接続して自分あてのメッセージを要求するまで、ホーム・サーバー上に保持されます。

# 第1章 概要

この章では、MQSeries Everyplace オブジェクトについての要旨と使用方法について説明します。

## MQSeries Everyplace クライアント

MQSeries Everyplace クライアントは、パーベイシブまたはモバイル・デバイス上で実行するコードです。これらのプログラムは、デバイス上で使用可能な API または関数の一部あるいはすべてを使用することができます。それらは MQSeries Everyplace プログラミング・インターフェースだけに限定されません。

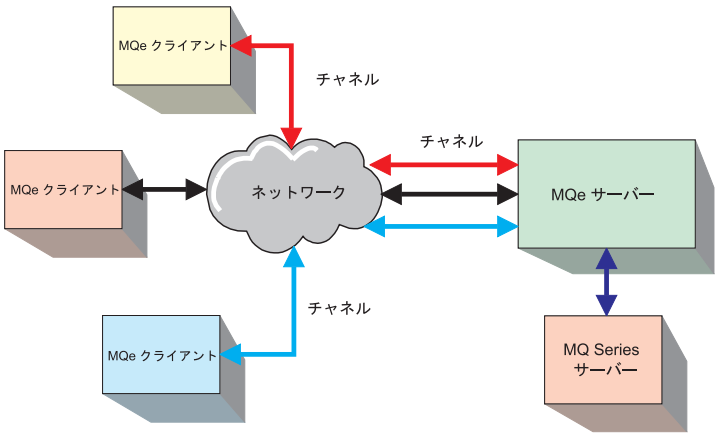


*図 1. MQSeries Everyplace クライアント*

使用できる接続のスタイルは、次のとおりです。

- 永続接続 (LAN、専用回線など)
- ダイヤルアウト接続 (インターネット・サービス・プロバイダー (ISP) に接続する標準モデムなど)
- ダイヤルアウトおよび応答機能 (携帯電話、テレビ電話など)

通信プロトコルは、アダプターのセットとして、サポートされるプロトコルごとに 1 つずつインプリメントされます。このことにより、新規プロトコルを非常に簡単に追加することができ、指定されたクライアント上のメモリー・フットプリントを、そのクライアントの構成に合わせて調整することができます。

# MQSeries Everyplace キュー・マネージャー

MQSeries Everyplace キュー・マネージャーは、MQSeries Everyplace システムの中心拠点です。これは次のものを提供します。

- MQSeries Everyplace アプリケーションのためのメッセージングおよびキューイング・ネットワークへのアクセスの中央点
- クライアント側キュー (オプション)
- 管理機能 (オプション)
- 1 回だけの確実なメッセージ送達
- 障害状態からの完全なリカバリー
- 拡張可能なルールに基づく動作

MQSeries Everyplace キュー・マネージャーは、オブジェクト指向スタイルで設計されています。オブジェクトは継承することができ、一連のルールを提供することによって、キュー・マネージャーの動作をカスタマイズすることができます。
MQSeries Everyplace キュー・マネージャーは、クライアント上で、または
MQSeries Everyplace サーバーの一部として実行することができます。

MQSeries Everyplace キュー・マネージャーはオプションで独自のキュー・セットを制御することができます。キューはキュー・マネージャーと同じマシンまたはデバイス上に存在しており、これは*ローカル・キュー* と呼ばれます。キュー・マネージャーはまた、MQSeries Everyplace/MQSeries ネットワーク内の他のキュー・マネージャーに属するキューに接続することもできます。これらのキューは、*リモート・キュー* と呼ばれ、それが所有するキュー・マネージャーは*リモート・キュー・マネージャー* と呼ばれます。 MQSeries Everyplace キュー・マネージャーは、リモート・キューの属性についていくらか知っていることがあります。なぜなら、それは属性について検出した情報を保管するからです。この情報は、*リモート・キュー定義* と呼ばれます。リモート・キューへのメッセージ伝送は同期または非同期のどちらの方式でも行えます。どちらの方式で行うかは、キュー・マネージャーがリモート・キューごとに保持するリモート・キュー定義で定義されます。

非同期通信では、MQSeries Everyplace アプリケーションは、キュー・マネージャーがオフラインのときでもメッセージを送信することができます。非同期として定義されたキューへの出力メッセージは、送信することが可能か、または送信することが適切と思われるときまで (あるいはその両方)、ローカル・キュー・マネージャー内に保管されます。 MQSeries Everyplace アプリケーションは、通常どおり継続することができます。メッセージをいつ送信するかを決めるのはキュー・マネージャー・ルールの主なタスクの 1 つであり、特に、通信の可用性が制限されている場合やコストに関する考慮事項がある場合に関係があります。

非同期通信では、宛先キュー・マネージャーおよびキューが、発信キュー・マネージャー上で事前定義されていなければなりません。これが必要なのは、メッセージの送達を確実に行うために、発信キュー・マネージャーは、宛先キュー・マネージャー / キューの対が有効となるよう設定する必要があるためです。

同期通信の場合、発信元と宛先の両方の MQSeries Everyplace キュー・マネージャーが、MQSeries Everyplace/MQSeries ネットワーク上で使用可能になっていなければなりません。

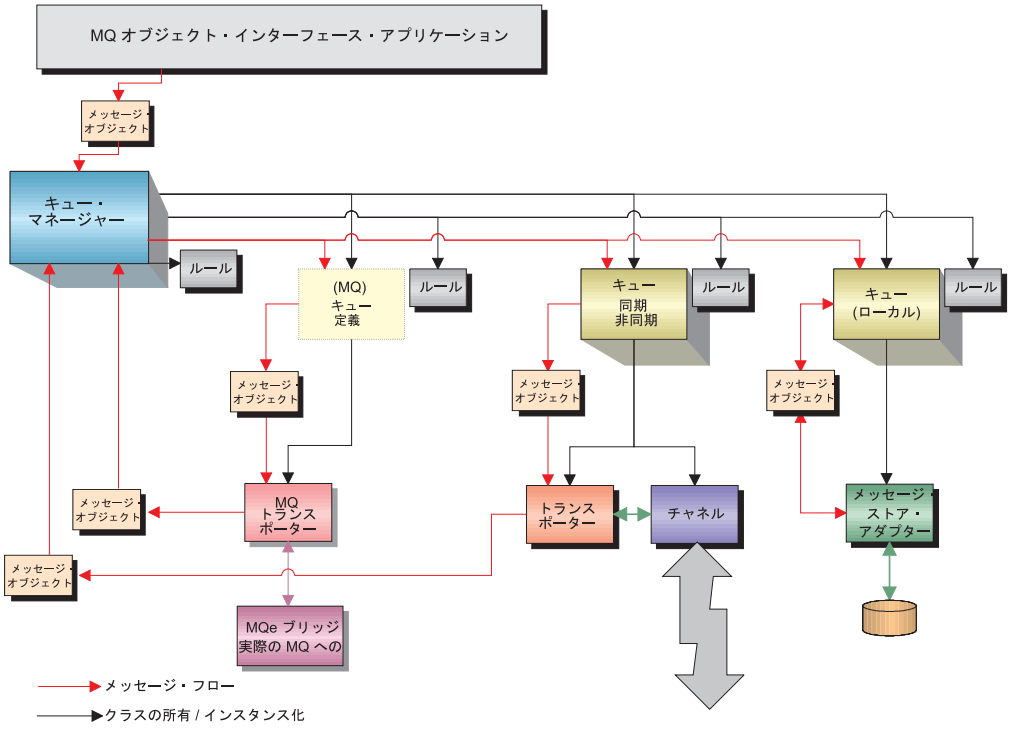


*図 2. MQSeries Everyplace メッセージ・フロー*

MQSeries Everyplace を介したメッセージのフローを、図2 に示します。

## MQSeries Everyplace キュー

クライアントがローカル・キューを持つように構成されている場合 (つまり、非同期のオフライン 作業が可能な場合)、メッセージ・オブジェクトがローカル・キューに保管されます。キューは特性を (認証、圧縮、暗号化など) を持つことができ、メッセージ・オブジェクトが移動されるときにこれらの属性が使用されます。

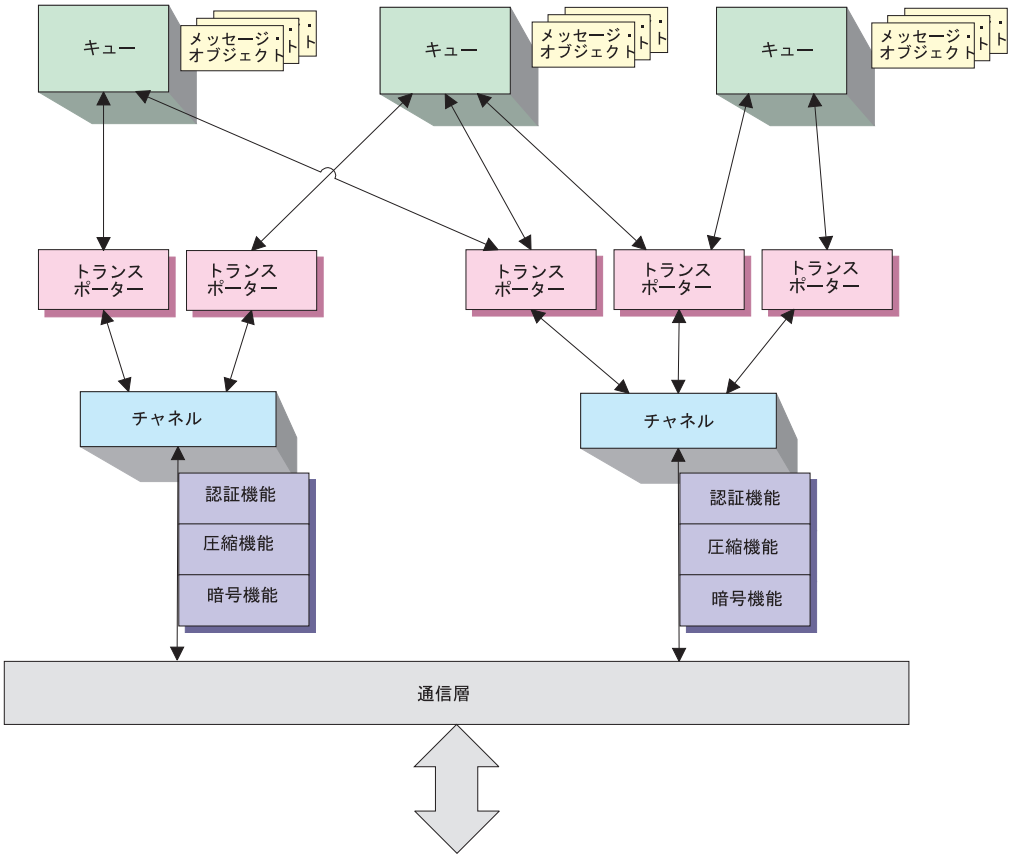


*図 3. MQSeries Everyplace キュー*

## MQSeries Everyplace 内のキュー・タイプ

MQSeries Everyplace 環境で使用できるキュー・クラスには、いくつかの異なるタイプがあります。 MQSeries Everyplace 開発パッケージで使用可能なタイプは次のとおりです。

### ローカル・キュー

このタイプのキューは、ローカル・アプリケーションが確実に (ハードウェア障害やデバイスの損失が生じた場合を除く)、そしてセキュリティー面でも安全にメッセージを保管するために使用します。このキューにはアダプター・クラス (通常はディスク・アダプター) を介してアクセスするメッセージ・ストアがありますが、適切なアダプターを作成すれば、メッセージはどこにでも保管することができます (DB2 データベースや書き込み可能 CD など)。

これらのキューは、オンラインででもオフラインでも、つまり、ネットワークに接続していてもいなくても使用することができます。

アクセスおよびセキュリティーはキューに属しており、リモート・キュー・マネージャーでの使用に関しては (ネットワークに接続する場合)、これら他のキュー・マネージャーがそのキューとメッセージをやり取りすることを許可する権限を設けることができます。ローカル・キューは常に操作モードで同期とみなされます。

## リモート・キュー

このタイプのキューは、ローカル環境にありません。定義がローカルに存在するのではなく、キューが属するキュー・マネージャーと実際のキューを識別します (つまり、その親キュー・マネージャーに対してローカルということです)。

リモート・キューは、同期的にも非同期的にもアクセスできます。リモート・キューの定義がローカルに保持されている場合、アクセスのモードはこの定義に基づいて同期または非同期になります。しかし、ローカルに保持されている定義がない場合には、キュー・ディスカバリーが発生し、特性 (認証、暗号、および圧縮) が検出されて、アクセスのモードは強制的に同期になります。

**同期キュー**は、所有するキュー・マネージャーへのパスを持つネットワークに接続されている場合にのみアクセスできるキューです。ネットワークが確立されていない場合は、取得、書き込み、ブラウズのような操作によって例外が発生します。所有するキューは、キューにアクセスするために必要なアクセス許可、およびセキュリティー要件を制御します。メッセージの送受信時のエラーや再試行を扱うのはアプリケーションの責任であり、 MQSeries Everyplace には、1 回で確実にメッセージを送達する責任はありません。

**非同期キュー**にはメッセージを書き込むことができますが、リモート非同期キューからメッセージを検索することはできません。ネットワーク接続が確立されると、メッセージはキューが属しているキュー・マネージャーとキューに送信されます。しかし、ネットワークが接続されていない場合には、メッセージはネットワーク接続が確立されるまでローカルに保管され、その後、伝送されます。このことによりアプリケーションは、クライアントがオフラインのときでもキューで動作できるようになります。ただし、そのためには、一時的にメッセージを保管するローカル・メッセージ・ストアが必要です。

## ストア・アンド・フォワード (蓄積交換) キュー

このタイプのキューは、次 (キューが属するキュー・マネージャーである必要はない) のキュー・マネージャーに転送できるようになるまでメッセージを保管します。通常、このタイプのキューはサーバー上で定義され、クライアントはネットワークに接続する際にそのメッセージを収集する必要があります。

ストア・アンド・フォワード (蓄積交換) キューは、多数のクライアントへのメッセージを保持する場合もありますし、クライアントとごに 1 つのストア・アンド・フォワード (蓄積交換) キューが設けられる場合もあります。

あるクライアントが切断されている別のクライアントにメッセージを送信する場合でも、送信側は受信側クライアントのキュー・マネージャーとローカル・キューにメッセージを書き込みます。すると中間サーバーは、受信側クライアントが接続されておらず、メッセージがそのローカル・メッセージ・ストアに保管されているこ

とを検出します。送信側アプリケーションは、クライアントが接続されているか切断されているかには関係なく、そのクライアントにメッセージを送信します。

### ホーム・サーバー・キュー

このタイプのキューは、通常、クライアント上にあり、ホーム・サーバー というサーバー上のストア・アンド・フォワード (蓄積交換) キューに関連付けられています。このキューは、クライアントがネットワーク上で接続するときには必ずホーム・サーバーからメッセージをプルします。

このキューがサーバーからメッセージをプルすると、`putMessage` および `confirmPutMessage` メソッド呼び出しによって、そのメッセージをローカル・キュー・マネージャーに渡します。その後、メッセージを適正なローカル・キューに置くのは、キュー・マネージャーの責任です。サーバーからメッセージを取得する pull メソッドは、ネットワーク上のフローの観点からすると、サーバーがメッセージをプッシュするよりも一層効果的と言えます。これは、ホーム・サーバー・キューが次のメッセージ (もしあれば) に対する要求として、最初のメッセージの肯定応答を使用するのに対し、サーバー・プッシュは 1 つの要求と応答によってメッセージを送信し、2 つ目の要求と応答によって確認フローを行う必要があるためです。

ホーム・サーバー・キューには、通常、ポーリング間隔が設定されており、その間隔で、ネットワークへの接続中にサーバー上に保留メッセージがないかをチェックします。ポーリング間隔は、管理構成オプションです。

### MQSeries － ブリッジ・キュー

このタイプのキューは常にサーバーにあり、MQSeries Everyplace 環境から MQSeries 環境へのパスを提供します。

## MQSeries Everyplace サーバー

MQSeries Everyplace サーバーは、ある MQSeries Everyplace サーバーから別のサーバーにメッセージを転送する機能を持っている必要があります。ただし、ネットワーク全体で同じプロトコル (たとえば TCP/IP) が使用されているという保証はないので、 MQSeries Everyplace サーバーはある通信層から別の通信層にメッセージを転送することができなければなりません (図4 を参照)。

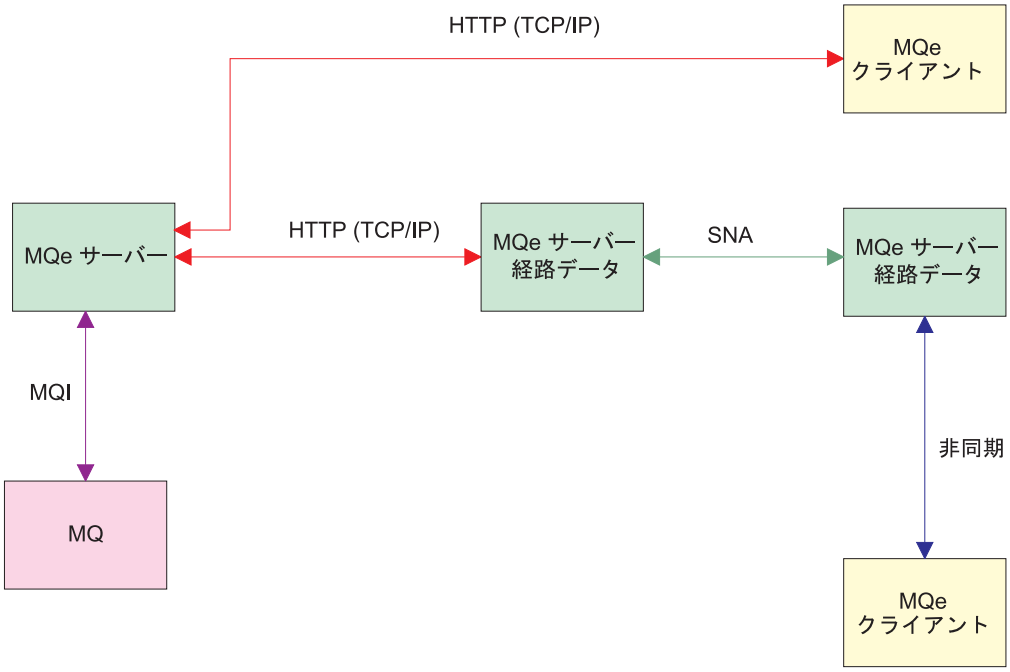


*図 4. MQSeries Everyplace サーバー*

## MQSeries への MQSeries Everyplace ブリッジ

MQSeries Everyplace サーバーは、MQSeries サーバーへのインターフェースになることができます。 2 つのシステムの間のメッセージ転送は、MQSeries ブリッジによって処理されます。このインターフェースについては、131ページの『MQSeries ブリッジの構成』で詳しく解説されています。

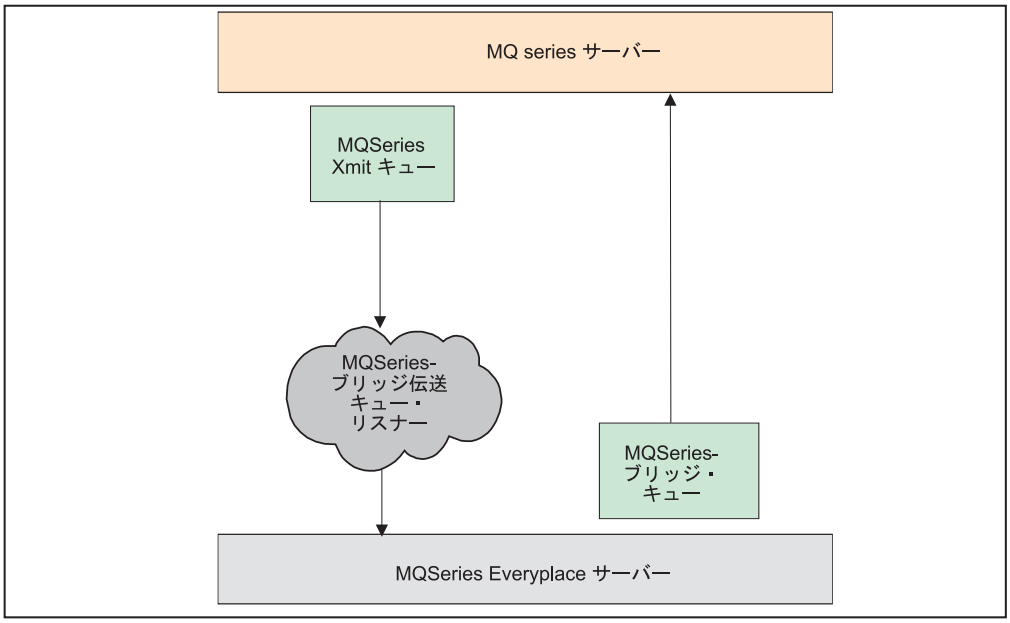


*図 5. MQSeries への MQSeries Everyplace インターフェース*

# MQSeries Everyplace チャネル

MQSeries Everyplace は、MQSeries Everyplace チャネルと呼ばれる、キュー・マネージャー間の接続を確立する方式をサポートします。チャネルは、2 つのパーティー間の論理接続であり、データをやり取りする目的で確立されます。チャネルは様々な属性または特性 (認証、暗号、圧縮、または使用されるプロトコルなど) を持つことができ、様々なチャネル上で様々なバージョンを使用することができます。それぞれのチャネルには、次の各属性ついて独自の値を設定することができます。

**認証機能**
ヌル、またはユーザー認証またはチャネル認証を実行することのできる認証オブジェクト。

**暗号機能**
ヌル、または暗号機能を実行することのできる暗号オブジェクト。

**圧縮機能**
'ヌル'、またはデータ圧縮 / 解凍を実行することのできる圧縮オブジェクト。

**宛先**
このチャネルの宛先。たとえば、 "server.xyz.com"

暗号の最も単純なタイプは XorCryptor です。これは、データの排他 OR を実行することによって送信されるデータを暗号化します。これは、セキュアな暗号化ではありませんが、データを見えないようにします。

最も単純なタイプの圧縮機能は RleCompressor です。これは、反復文字を 1 カウントに置き換えることによってデータを圧縮します。

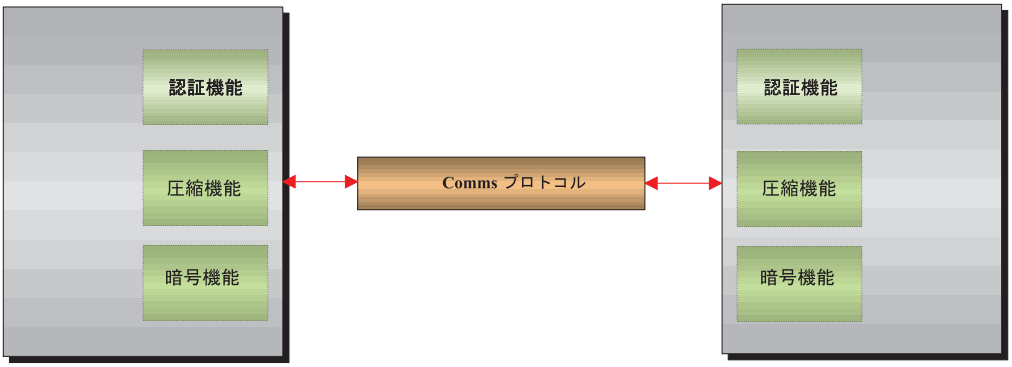


*図 6. MQSeries Everyplace チャネル*

一般に、認証機能が使用されるのは、チャネルの設定時に限られます。また、一般に、圧縮機能と暗号機能は、すべてのフローで使用されます。

**注:** 世界のいたるところで、暗号のレベルとタイプに関する政府規定が変わりつつあります。 MQSeries Everyplace で使用される暗号のレベルとタイプは必ず、該当する各地の法律規定に従っていなければなりません。これは特に、国境を越えて移動するモバイル・デバイスで使用するときに関係があります。
MQSeries Everyplace はこのための機能を提供していますが、それをインプリメントするのはアプリケーション・プログラマーの責任です。

# 第2章 概説

このセクションでは、MQSeries Everyplace 開発キット バージョン 1.0 について紹介します。開発キットは、Java 1.1 ベースのメッセージングおよびキュー・アプリケーションを作成するための開発環境です。

**注:** Java 以外の開発キットの可用性については、MQSeries Web サイト (`http://www.ibm.com/software/ts/mqseries/`) を参照してください。

開発キットのコード部分は、2 つのセクションに分けられます。

**ベース MQSeries Everyplace クラス**
メッセージおよびキュー・アプリケーションを作成するのに必要な機能すべての提供する Java クラスのセット

**例** MQSeries Everyplace の多くの機能を使用する方法を示す Java ソース・コードおよびクラスのセット

## 開発環境

MQSeries Everyplace 開発キットを使用して Java でプログラムを開発するには、Java 環境が以下のようにセットアップされていなければなりません。

- Java 開発キット (JDK) は、MQSeries Everyplace クラスを見つけることができなければなりません。

  たとえば、標準 JDK を使用する Windows® 環境では、CLASSPATH は以下のように設定できます。

  ```
  Set CLASSPATH=<MQeInstallDir>¥Java;%CLASSPATH%
  ```

- MQSeries ブリッジを使用または拡張するコードを開発する場合、MQSeries Java クライアントがインストールされており、JDK で使用できるようになっていなければなりません。 MQSeries Java クライアントの環境を設定することについての詳細は、 *MQSeries Using Java* を参照してください。

MQSeries Everyplace と共に使用できる Java 開発環境および Java ランタイム環境は数多くあります。開発とランタイムのどちらのシステム環境も、使用される環境に依存しています。 Windows システムの場合、異なる Java 開発キットについて開発環境をセットアップする方法を示すバッチ・ファイル `JavaEnv.bat` が MQSeries Everyplace で提供されています。 `JavaEnv.bat` を使用する場合は、このコピーを取って、使用するマシンの環境に適合するようにこのコピーを変更しなければなりません。このファイルは、MQSeries Everyplace の例の一部を実行するバッチ・ファイルのセットにより使用されます。このバッチ・ファイルの例を使用するには、`JavaEnv.bat` が以下のように変更されていなければなりません。

- `JDK` 環境変数を JDK のベース・ディレクトリーに設定する
- `JavaCmd` 環境変数を Java アプリケーションを実行するのに使用したコマンドに設定する
- MQSeries Java クライアントがインストールされている場合には、`MQDIR` 環境変数を MQSeries Java クライアントのベース・ディレクトリーに設定する

**注:** JavaEnv.bat の変更バージョンを使用している場合には、MQSeries Everyplace を再インストールする際にこれが上書きされる可能性があります。

JavaEnv.bat が起動すると、パラメーターが渡され、使用する Java 開発キットのタイプが判別されます。

使用できる値は以下のとおりです。

**Sun** - Sun

**JB** -Borland JBuilder

**MS** - Microsoft

**IBM** - IBM

パラメーターが渡されない場合には、デフォルトは IBM です。

JavaEnv.bat は、デフォルトでは <MQeInstallDir>¥java ディレクトリーから実行されます。これは、他のディレクトリーから実行したり、他の Java 開発キットを使用したりできるように変更することが可能です。

## アプリケーションの展開

MQSeries Everyplace アプリケーションを展開する場合には、アプリケーションに必要なクラスの最低限のセットを jar 圧縮ファイルに圧縮し、アプリケーションで必要なシステム・リソースを最小限に抑えるようにします。 MQSeries Everyplace では、MQSeries Everyplace クラスが jar ファイルに圧縮される方法について、以下の例を提供しています。これらは、<MQeInstallDir>¥Jars ディレクトリーにあります。

**MQeDevice.jar**
デバイスで使用できるベース・クラスのフル・セット

**MQeGateway.jar**
ゲートウェイで使用できるベース・クラスのフル・セット

**MQeMQBridge.jar**
MQeGateway.jar を拡張して、MQSeries と相互操作するゲートウェイを作成するのに使用できるクラス。

**MQeHighSecurity.jar ***
MQeGateway.jar および MQeDevice.jar の両方を拡張して、拡張セキュリティーを提供するのに使用できるクラスのセット。

**MQeMiniCertificateServer.jar ***
ミニ認証サーバーを実行するのに必要なすべてのクラスを提供する自己完結型の jar ファイル。

**MQeExamples.jar**
1 つの jar ファイルへのすべての MQSeries Everyplace 例のパッケージ。

* これらの jar ファイルは、MQSeries Everyplace の高セキュリティー・エディションにのみ組み込まれています。

### アプリケーションの展開

MQSeries Everyplace アプリケーションを実行するには、Java ランタイム環境が設定されており、必要な MQSeries Everyplace およびアプリケーション・クラスが組み込まれている必要があります。標準 Java ランタイム環境 (JRE) を使用して、CLASSPATH に必要な jar ファイルを組み込むように設定しなければなりません。

たとえば、Windows システムでは、次のように入力します。

```
Set CLASSPATH=<MQeInstallDir>¥Jars¥MQeDevice.jar;%CLASSPATH%
```

## インストール後のテスト

MQSeries Everyplace がインストールされたら、以下の手順で例のセットを実行して、開発キットが正常にインストールされたかどうかを判別します。

- Java 環境が、11ページの『開発環境』で説明されているとおりにセットアップされていることを確認します。以下に説明するバッチ・ファイルを実行する場合、それぞれの最初のパラメーターは、使用する Java 開発キットの名前です。何も指定されていない場合には、デフォルトである IBM が使用されます。
- <MQeInstallDir>¥Java ディレクトリーに変更します。
- キュー・マネージャーを作成します。

  バッチ・ファイル CreateExampleQM.bat <JDK> を実行して、'ExampleQM' と呼ばれるキュー・マネージャーの例を作成します。

  作成プロセスの一部として、キュー・マネージャー構成情報およびキューを保持するようにディレクトリーをセットアップする作業があります。例では、現行のディレクトリーに対応する ExampleQM というディレクトリーを使用します。このディレクトリーには、さらに 2 つのディレクトリーがあります。

  – Registry - キュー・マネージャー構成データを含むファイルを保持します。
  – Queues - 各キューについて、キューのメッセージを保持するために使用できるサブディレクトリーがあります。 (キューが活動化されるまで、ディレクトリーは作成されません。)
- 単純なアプリケーションを実行します。

  キュー・マネージャーが一度作成されると、これをアプリケーションによって始動および使用することができます。バッチ・ファイル ExamplesMQeClientTest.bat は、単純なアプリケーションの例の一部で使用することができます。

  デフォルトで、examples.application.Example1 が実行されます。この例では、テスト・メッセージをキュー・マネージャー ExampleQM に書き込んでから、この同じキュー・マネージャーからメッセージを取り出します。この 2 つのメッセージが適合すると、アプリケーションは正常に実行したことになります。

  examples.application パッケージにはアプリケーションのセットがあり、MQSeries Everyplace のあらゆる機能をデモンストレーションします。これらは、パラメーターをバッチ・ファイル ExamplesMQeClientTest <JDK> <ExampleNo> に渡すことによって実行できます。ここで、*ExampleNo* は 1 〜 6 の範囲の例の接尾部です。
- キュー・マネージャーを削除します。

キュー・マネージャーが必要でなくなったら、削除してもかまいません。キュー・マネージャーの例 ExampleQM を削除するには、バッチ・ファイル `DeleteExampleQM.bat <JDK>` を実行します。

一度削除すると、キュー・マネージャーを開始することはできなくなります。

キュー・マネージャーを削除しても、このキューから取り出されていないメッセージや、ベース・キュー・マネージャーの作成の一部として作成されなかった構成データは削除されないことに注意してください。したがって、同じ作成パラメーターを使用してキュー・マネージャーが再作成されると、以前に取り出されていないメッセージは再作成されたキュー・マネージャーで使用できることになります。

**注:** この例では、セットアップを簡単にするために相対ディレクトリー (.¥) を使用します。基本の開発およびデモンストレーションの場合以外は、絶対ディレクトリーを使用することを強くお勧めします。相対ディレクトリーが使用される場合には、現行のディレクトリーが変更される際に問題が発生する可能性があります。その場合、キュー・マネージャーは構成情報およびキューを見つけることができなくなります。

# 例

上に説明した例は、MQSeries Everyplace で提供される例のセットの一部に過ぎません。個々の例は、MQSeries Everyplace の機能の使用および拡張方法を説明しています。これらの大半は、プログラマーの手引きの関係するセクションに説明されています。以下のセクションでは、これらすべてをリストし、簡単に説明します。

## examples.application パッケージ

このパッケージには、メッセージの書き込みおよびキューからのメッセージの取得などの、キュー・マネージャーと対話するためのあらゆる方法を示す例のセットが含まれています。すべての例は、ローカル・キュー・マネージャーまたはリモート・キュー・マネージャーのいずれかを使用して作業できます。これらのアプリケーションのいずれかを実行できるようにするには、まず使用するキュー・マネージャーを作成しなければなりません。たとえば、`CreateExampleQM.bat` バッチ・ファイルを使用して、キュー・マネージャー ExampleQM を作成します。

**例 1** メッセージの単純な書き込みおよび取得

**例 2** いくつかのメッセージを書き込み、一致するフィールドを使用して 2 番目のメッセージを取得する

**例 3** メッセージ・リスナーを使用して、新しいメッセージの到着を検出する

**例 4** WaitForMessage ルーチンを使用して、指定された間隔の間に到着するメッセージを取得する

**例 5** メッセージをロック、取得、ロック解除、削除する

**例 6** 確実なメッセージ送達を使用して簡単なメッセージを書き込みおよび取得する

**ExampleBase**
すべてのアプリケーションの例が継承するベース・クラス

以下のようにしてバッチ・ファイル ExamplesMQeClientTest.bat を使用すると、これらの例を実行することができます。

```
ExamplesMQeClient.Test  <JDK> <example no> <remoteQMgrName> <localQMgr ini file>
```

ここで、

**<JDK>**
Java 環境の名前 (詳細は 11ページの『開発環境』を参照)。デフォルトは IBM です。

**<example no>**
実行する例の数 (例の名前の接尾部)。デフォルトは、1 (Example1)。

**<remoteQMgrName>**
アプリケーションで作業するキュー・マネージャーの名前。これは、ローカルまたはリモート・キュー・マネージャーの名前にすることができます。リモート・キュー・マネージャーの場合、ローカル・キュー・マネージャーがリモート・キュー・マネージャーと通信する方法を定義する接続を構成しなければなりません。デフォルトでは、ローカル・キュー・マネージャーが使用されます (ExamplesMQeClient.ini で定義されている)。

**<localQMgrIniFile>**
ローカル・キュー・マネージャーの始動パラメーターを含む ini ファイル。デフォルトでは、ExamplesMQeClient.ini が使用されます。

キュー・マネージャーと対話するアプリケーションの作成方法についての詳細は、33ページの『第4章 MQSeries Everyplace キュー・マネージャー』を参照してください。

## examples.administration.simple パッケージ

このパッケージには、MQSeries Everyplace の管理機能のいくつかを方針に基づいて使用する方法を示す例のセットが含まれています。アプリケーションの例と同じく、これらの例も、ローカルまたはリモート・キュー・マネージャーで作業できます。

**例 1** キューを作成および削除する

**例 2** リモート・キュー・マネージャーへ接続定義を追加する

**例 3** キュー・マネージャーの特性を照会し、これ自体をキューに入れる

管理の詳細については、91ページの『第5章 MQSeries Everyplace 管理』を参照してください。

## examples.administration.console パッケージ

このパッケージには、MQSeries Everyplace リソースを管理するための簡単なグラフィカル・ユーザー・インターフェースをインプリメントするクラスのセットが含まれています。

**Admin**
管理 GUI の例へのフロントエンド。

**例**

加えて、個々の MQSeries Everyplace 管理対象リソースにグラフィカル・ユーザー・インターフェースを提供するクラスのセットがあります。

GUI は、以下のいずれかの方法で呼び出すことができます。

- バッチ・ファイル `ExamplesAdminConsole.bat` を使用する
- コマンド行から以下を入力する

  ```
  java examples.administration.console.Admin
  ```

- サーバーのサンプル `examples.awt.AwtMQeServer` のボタンを使用する

管理の詳細については、91ページの『第5章 MQSeries Everyplace 管理』を参照してください。

## examples.attributes パッケージ

このパッケージには、追加のコンポーネントを作成して、MQSeries Everyplace セキュリティーを拡張する方法を示すクラスのセットが含まれています。

**NTAuthenticator**
ユーザーを Windows NT セキュリティー・データベースに認証する認証プログラム。 NT 認証プログラムは、JNI インターフェースを使用して Windows NT セキュリティーと相互作用します。このコードは、`examples.nativecode` ディレクトリーにあります。

**TableCryptor**
非常に単純な暗号機能

セキュリティーの詳細については、175ページの『第7章 セキュリティー』を参照してください。

## examples.awt パッケージ

このパッケージは、小さなグラフィカル・インターフェースを必要とするアプリケーションと、MQSeries Everyplace 機能へのグラフィカル・フロントエンドを提供するアプリケーションをまとめて構築するためのツールキットを提供します。

**AwtMQeServer**
`examples.queuemanager.MQeServer.` の例へのグラフィカル・フロントエンド。クラス MQeTraceResourceGUI が、GUI で使用できる国際化されたストリングを含むリソース・バンドルを提供します。 MQeTraceResourceGUI は、パッケージ `examples.trace` にあります。

バッチ・ファイル `ExamplesAwtMQeServer.bat` を使用すると、このアプリケーションを実行できます。

サーバー環境でキュー・マネージャーを実行することについての詳細は、47ページの『サーバー』を参照してください。

**AwtMQeTrace**
`examples.trace.MQeTrace` へのグラフィカル・フロントエンド。

MQSeries Everyplace トレース機能についての詳細は、219ページの『第8章 MQSeries Everyplace でのトレース』を参照してください。

クラス AwtDialog、AwtEvent、AwtFormat、AwtFrame、AwtLayout、および AwtOutputStream は、グラフィカル・アプリケーションに基づいた小さいフットプリントを作成するためのツールキットを提供します。これらのクラスは、多くのグラフィカル MQSeries Everyplace 例により使用されます。

## examples.eventlog パッケージ

このパッケージには、異なる機能にイベントのログを記録する方法を示す例がいくつか含まれています。

**LogToDiskFile**
ディスク・ファイルにイベントを書き込む。

**LogToNTEventLog**
Windows NT イベント・ログにイベントを書き込む。このクラスは、JNI インターフェースを使用して、Windows NT イベント・ログと対話します。このコードは、examples.nativecode ディレクトリーにあります。

## examples.install パッケージ

このパッケージには、キュー・マネージャーを作成および削除するためのクラスのセットが含まれています。

**DefineQueueManager**
ユーザーがキュー・マネージャーの作成に関連したオプションを選択できるようにするための GUI。すべてのオプションが選択されると、これは、キュー・マネージャー始動パラメーターを含む ini ファイルを作成して、キュー・マネージャーを作成します。

**CreateQueueManager**
キュー・マネージャーの始動パラメーター ini ファイルの名前およびディレクトリーを使用する GUI プログラム。名前およびディレクトリーが提供されると、キュー・マネージャーが作成されます。

**SimpleCreateQM**
キュー・マネージャーの始動パラメーター ini ファイルの名前をパラメーターとして使用するコマンド行プログラム。オプションで、キューが保管されているルート・ディレクトリーであるディレクトリー。有効な ini ファイルが見つかると、キュー・マネージャーが作成されます。

**DeleteQueueManager**
キュー・マネージャーの始動パラメーター ini ファイルの名前を使用する GUI プログラム。 ini ファイルが入力されると、キュー・マネージャーは削除されます。

**SimpledDeleteQM**
キュー・マネージャーの始動パラメーター ini ファイルの名前としてパラメーターを使用し、それからこのキュー・マネージャーを削除するコマンド行プログラム。

詳細については、33ページの『第4章 MQSeries Everyplace キュー・マネージャー』を参照してください。

## examples.native パッケージ

例を使用するには、Windows NT 機能へのアクセスが必要になります。 MQSeries Everyplace は、java ネイティブ・インターフェース (JNI) を使用してこれらの機能にアクセスします。 examples¥native ディレクトリーにあるコードは、examples.attributes.NTAuthenticator および examples.eventlog.LogToNTEventLog によって必要とされる JNI インプリメンテーションを提供します。

## examples.queuemanager パッケージ

キュー・マネージャーは、数多くの異なるタイプの環境で実行できます。このパッケージには、キュー・マネージャーが、クライアント、サーバー、およびサーブレットとして実行できるようにする例のセットが含まれています。

**MQeClient**
通常はデバイスで使用される単純なクライアント

**MQePrivateClient**
セキュア・キューおよびセキュア・メッセージングで使用できるクライアント。

**MQeServer**
複数のキュー・マネージャー (クライアントまたはサーバー) に同時に接続できるサーバー。通常、ゲートウェイで使用されます。バッチ・ファイル ExamplesAwtMQeServer.bat を使用すると、このサーバーにグラフィカル・フロントエンドを提供する examples.awt.AwtMQeServer の例を実行できます。

**MQePrivateServer**
MQeServer と類似しているものの、セキュア・キューおよびセキュア・メッセージングを可能にする。

**MQeServlet .**
サーブレットでキュー・マネージャーを実行する方法を示す例。

異なる環境でキュー・マネージャーを実行することについての詳細は、42ページの『キュー・マネージャーの開始』を参照してください。セキュア・キューおよびメッセージングの環境を提供するキュー・マネージャー (MQePrivateClient および MQePrivateServer) についての詳細は、175ページの『第7章 セキュリティー』を参照してください。

## examples.rules パッケージ

ベースの MQSeries Everyplace 機能は、ルールを使用して制御および拡張できます。 MQSeries Everyplace のコンポーネントの一部は、ルール・クラスがこれらに適用できるようにします。これらのルールは、コンポーネントの機能を変更する手段を提供します。このパッケージには、以下のルール・クラスの例が含まれています。

**ExamplesQueueManagerRules**
保留メッセージを伝送するように定期的に試行するキュー・マネージャー・ルール・クラスの例。

詳細については、80ページの『ルール』を参照してください。

**AttributeRule**
属性の使用を制御する属性ルールの例。

## examples.security パッケージ

このパッケージには、MQSeries Everyplace セキュリティーを変更する例が含まれています。

**MQeSecurity**
MQSeries Everyplace の特定の機能の使用を許可するかどうかを制御する Java セキュリティー・マネージャーへの拡張機能の例。

## examples.trace パッケージ

このパッケージには、開発時のアプリケーションのデバッグ、および完了したアプリケーションのトレースの両方に使用できるトレース・ハンドラーの例が含まれています。

**MQeTrace**
ベース MQSeries Everyplace トレース・クラス。

AwtMQeTrace。これは `examples.awt` パッケージにあり、このトレース・クラスへのグラフィカル・フロントエンドを提供します。

**MQeTraceResource**
MQSeries Everyplace により出力されるトレース・メッセージを含むリソース・バンドル

## examples.mqbridge パッケージ

このパッケージには、MQSeries ブリッジの使用方法および拡張方法を示すクラスのセットが含まれています。例の一部は、他の MQSeries Everyplace の例を拡張します。

詳細については、131ページの『第6章 MQSeries ブリッジ』を参照してください。

# 第3章 MQeFields および MQeMsgObject

MQeFields は、MQSeries Everyplace メッセージを送信、受信、または操作するためにデータ項目を保持するのに使用される基本クラスです。 MQeFields クラスは次のように構成されています。

```
/* create a MQeFields object */
MQeFields fields = new MQeFields( );
```

MQeFields オブジェクト内には、正しいデータ型を保持しながらアイテムを格納および検索するための様々な put および get メソッドがあります。アイテムは名前、型、および値の形式で保持されます。名前は以下のルールに従っている必要があります。

- 1 文字以上にする
- ASCII 文字セットに準拠する。つまり、20 < value < 128 の値の文字。
- 次の文字を含めてはならない。{ } [ ] # ( ) : ; , ’ ″ =
- MQeFields オブジェクト内で固有でなければならない

名前は値の検索および更新に使用されます。 MQeFields オブジェクトがダンプされるときに名前はアイテム・データとともに組み込まれるので、名前は短くしておくのがよいでしょう。

次の例は、MQeFields オブジェクトに値を置く方法を示しています。

```
/* Place integer values in a fields object */
  fields.putInt( "Int1", 1234 );
  fields.putInt( "Int2", 5678 );
  fields.putInt( "Int3",    0 );
```

値を検索するには:

```
/* Retrieve an integer value from a fields object */
  int Int2 = fields.getInt( "Int2" );
```

表1に示されている値の型を格納および検索するためのメソッドが提供されています。

*表 1. 格納および検索メソッド*

| 値の型 | 格納メソッド | 検索メソッド |
|---|---|---|
| byte (バイト) | putByte | getByte |
| int (整数) | putInt | getInt |
| short (短精度) | putShort | getShort |
| long (長精度) | putLong | getLong |
| floating point (浮動小数点) | putFloat | getFloat |
| | putDouble | getDouble |
| boolean (ブール) | putBoolean | getBoolean |
| string (ストリング) | putAscii | getAscii |
| | putUnicode | getUnicode |

値の配列はフィールド・オブジェクト内に保持できます。配列の保持には 2 つの形式があります。

- バイト、短精度、整数、長精度、浮動小数点および倍精度の固定長配列は、putArrayOf*type* および getArrayOf*type* メソッドを使用して処理されます。ここで、*type* は Byte、Short、Int、Long、Float、または Double です。
- バイト、短精度、整数、長精度、浮動小数点、倍精度、およびストリングの可変長配列は、put*type*Array/get*type*Array を使用して処理されます。この形式を使用することにより、各要素はアイテムの名前に *:nn* を付加してアイテムを連続させた形で格納されます (*nn* は 0 で始まる配列内の要素番号です)。個々のアイテムは配列長を含めて設定されます。この配列長は整数値で、putArrayLength/getArrayLength を使用して処理されます。

フィールド・オブジェクトは、putFields/getFields メソッドを使用して別のフィールド・オブジェクト内に組み込むことができます。

通常の MQSeries Everyplace メッセージに使用されるクラスは MQeMsgObject またはこのクラスの下位クラスです。 MQeMsgObject は MQeFields クラスの下位クラスなので、すべてのフィールド・メソッドにアクセスできます。

フィールド・オブジェクトの内容は、以下の形式でダンプされます。

**バイナリー**

これは、ネットワークを介してフィールドまたは MQeMsgObject を送信する際に通常使用される形式です。データをバイナリーに変換するために用いられるメソッドは dump です。このメソッドは、オブジェクトの内容をコード化した形式を含むバイナリー・バイト配列を戻します。 Dump にはオプションとしてブール値パラメーターがあり、これはダンプされたデータを直前のフィールド・データで XOR 処理するかどうかを指定します。これは、出力配列の 0x00 であるバイト数を増やして、ネットワークを介して送信するデータ・ストリームのサイズを圧縮プログラムが小さくできるようにするという方法です。アプリケーションが他の物理メディアにバイト配列を書き出すことを意図しているのでなければ、このパラメーターの指定にはあまり利点はありません。

**コード化ストリング**

ストリング形式には様々な制限が課せられており、ストリングを使用してフィールド・オブジェクトを格納することは必ずしも可能ではありません。ストリング形式は、フィールド・オブジェクトの dumpToString メソッドを使用します。これには 2 パラメーター、テンプレートと名称が必要です。テンプレートは、次の例が示すように、フィールド項目データをどのように返還する必要があるかを示すパターン・ストリングです。

```
"(#0)#1=#2¥r¥n"
```

この場合

**#0** データ型 (たとえば ″ascii″ または ″short″)

**#1** フィールド名

**#2** 値のストリング表記

その他の文字は変更されずに出力ストリングにコピーされます。メソッドは組み込みフィールド・オブジェクトをオブジェクトに正常にダンプしますが、組み込みフィールド・データが restoreFromString メソッドを使用して復元できることの保証はありません。

フィールドの強力な機能は、次の例が示すように、ini ファイル (ASCII 値のセクションおよびキーワードを持つ ASCII テキスト・ファイル) を読み込めるということです。

[Section1]
Keyword1=value1
Keyword2=value2
[Section2]
Keyword1=value
...

次のコード例が示すように、このデータは読み込みおよび構文解析してフィールド・オブジェクトに取り込むことができます。

```
File diskFile = new File( fileName );              /*access the file*/
  byte data[] = new byte[(int) diskFile.length()];       /*file size*/
  FileInputStream inputFile = new FileInputStream( diskFile );
  inputFile.read( data );                            /*read the file*/
  inputFile.close( );                               /*finish with file*/
  MQeFields fields = new MQeFields( );          /*new Fields Object*/
  fields.restoreFromString( "¥r¥n",            /*end of line string*/
"[#0]",                                          /*section pattern*/
"#1=#2",                                          /*keyword pattern*/
byteToAscii( data ) );
```

コードの変形を使用すると、次の例のように、異なるデータ型を復元することができます。

[Section1]
(ascii)Keyword1=value1
(int)Keyword2=1234
[Section2]
(boolean)Keyword1=true
...

[

```
File diskFile = new File( fileName );               /*access the file*/
  byte data[] = new byte[(int) diskFile.length()];     /*size of file*/
  FileInputStream inputFile = new FileInputStream( diskFile );
  inputFile.read( data );                             /*read the file*/
  inputFile.close( );                               /*finish with file*/
  MQeFields fields = new MQeFields( );            /*new Fields Object*/
  fields.restoreFromString( "¥r¥n",              /*end of line string*/
"[#0]",                                            /*section pattern*/
"(#0)#1=#2",                                        /*keyword pattern*/
byteToAscii( data ) );
```

**注:** dumpToString は、前述の技法を使用して復元できない形式には、組み込みフィールド・オブジェクトをダンプしません。

ini ファイルの使用は任意ですが、使用することをお勧めします。 ini ファイルを処理するためのユーティリティーは MQSeries Everyplace とともに提供されている例にありますが、ユーザー自身が独自に作成したり、MQeFields オブジェクトを直接使用することもできます。 ini ファイルを使用する場合、キュー・マネージャーを作成および管理するために、ファイルが MQeFields オブジェクトに渡されなければなりません。この処理のために役立つ例が提供されています。

## ini ファイル・エディターに基づいたフィールドの作成

この例は、MQSeries Everyplace examples.awt ディレクトリーにあるコンポーネント例を使用して、 ini ファイル・エディターを作成します。これはすべての形式の MQeFields を包含するという意味はなく、より強力なエディターの例または出発点を意図したものです。

次の例は、各セクションを別個の組み込みフィールド・オブジェクトとして扱います。基本クラスは、ini ファイル内にあるすべてのセクションをリストする Choice ボックスの付いたウィンドウを作成します。

この例では、examples.awt ディレクトリーにある Java クラスを利用します。これらのクラスにより、基本フレームとダイアログを簡単に作成および操作することができます。

examples.awt.AwtFrame からアプリケーションが拡張するメニューの付いたフレームを作成するには、次のようにします。

```
public class Editor extends examples.awt.AwtFrame
  /*-----------------------------------------------------------------*/
  public editor( String args[] ) throws Exception
    {
    super( "Editor - " );
    format( Menu,  new String[][][] {
                          { { "File"      },
                            {  " ", "Open"  },       /* Index   0   */
                            {  " ", "Save"  },       /* Index   1   */
                            {  "-" },
                            {  " ", "Exit"  } },     /* Index   2   */
                            { { "Help"    },
                            {  " ", "Trace" } } } ); /* Index   3   */
    visible( true );
    }
```

super( "Editor - " ); は、フレームの表題を割り当て、上位を初期化します。次のステートメントは、メニュー・バーをフレームに割り当て、バーに表示される項目を定義します。

```
format( Menu,  new String[][][] {
                          { { "File"      },
                            {  " ", "Open" },          /* Index  0   */
                            {  " ", "Save" },          /* Index  1   */
                            {  "-" },
                            {  " ", "Exit"  } },       /* Index  2   */
                            { { "Help" },
                            {  " ", "Trace" } } } );   /* Index  3   */
```

format メソッド呼び出しには 2 つのパラメーターがあり、この例では最初のものが Menu、2 番目のものが String array オブジェクトです。

ストリング配列は次のように、3D 配列でなければなりません。

- 最初のディメンションが行数を定義する。
- 2 番目のディメンションが列数を定義する。
- 3 番目のディメンションが形式のコンポーネントを定義する。

たとえば、次のようにします。

```
new String[][][] { type, Data {, Data, { ... } } }
```

この場合

**type** コンポーネントのタイプ

″ ″ 標準のメニュー項目

″**C**″ CheckItem - チェックなし。修飾子 ″!″ でチェックを付けることも可能

″**-**″ 区切り文字

他のものはすべてラベルとして扱われる

**Data** コンポーネントによって使用されるテキスト

アクション・イベントを引き起こしうるメニューの各項目には、先行するコード断片の配列での位置に基づいてインデックス番号が付けられています。コメントにインデックス番号が示されています。

format メソッドの最初のパラメーターには、フレーム内のパネルの位置に対応させて North、South、East、West および Center という値を割り当てることもできます。この場合の String 配列オブジェクトは次のような構文になります。

```
new String[][][] { type, Data {, Data, { ... } } }
```

この場合

**type** コンポーネントのタイプ

″**A**″ テキスト域、次の修飾子を指定することが可能

- ″**P**″ 保護 - 編集不能
- ″**K**″ キー解放アクション・イベントを与える

″**B**″ ボタン

″**C**″ チェック・ボックス - チェックなし、次の修飾子を指定することが可能

- ″**!**″ チェックあり

″**D**″ 選択 (ドロップダウン・リスト)

″**L**″ ラベル

″**S**″ 選択リスト (リスト・ボックス)

″**T**″ テキスト・フィールド、次の修飾子を指定することが可能

- ″**K**″ キー解放アクション・イベントを与える
- ″**P**″ 保護 - 編集不能
- ″*****″ マスクされた入力

他のものはすべてラベルとして扱われる

**Data** コンポーネントによって使用されるテキスト

これらのメソッドがどのように働くかについての詳細は、examples.awt ディレクトリーの AwtDialog、AwtFormat、および AwtFrame の例を参照してください。

エディターの作成に examples.awt コンポーネントを使用することにより、次のコードは 3 つの作業変数、およびメニューと Choice ボックスが 1 つ付いたウィンドウを作成するコンストラクターを定義します。

```
public class Editor extends examples.awt.AwtFrame
  {
  protected Choice    choiceBox   = null;
  protected MQeFields fields      = null;
  protected String    currentFile = "";
  /*-------------------------------------------------------------------*/
  public editor( String args[] ) throws Exception
    {
    super( "Editor - " );
    format( Menu,  new String[][][] {
                          { { "File"      },
                            {  " ", "Open"  },       /* Index   0   */
                            {  " ", "Save"  },       /* Index   1   */
                            {  "-" },
                            {  " ", "Exit"  } },     /* Index   2   */
                            { { "Help"    },
                            {  " ", "Trace" } } } ); /* Index   3   */
    format( North, new String[][][] {
                          { { "D", "< -- No File Loaded -- >" } } } );
    choiceBox = (Choice) getObject( North, 0 );
    visible( true );
    }
  /*-------------------------------------------------------------------*/
  public static void main( String[] args )
    {
    try
      {
      new  Editor( args );
      }
    catch ( Exception e )
      {
      e.printStackTrace();
      }
    }
```

次のコードは、ユーザーがメニューと Choice ボックスで対話することによって引き起こされたイベントを処理します。

「Menu」アクション:

**Open** Action インデックスは 0。これは switch ステートメントで使用され、Load メソッドを呼び出してディスク・ファイルを読み取る

**Save.・**
Action インデックスは 1。これは switch ステートメントで使用され、Save メソッドを呼び出してディスク・ファイルに書き出す

**Exit** Action インデックスは 2。これは switch ステートメントで使用され、プログラムを終了する

**Trace** Action インデックスは 3。これは switch ステートメントで使用され、Examples.Awt.AwtMQeTrace クラスを呼び出す

Choice ボックスは North にだけ配置されているコンポーネントで、そのためインデックス 0 が付けられています。このリスト・ボックスから項目を選択すると、フィールド・オブジェクトの内容を表示するために使用されるクラスがアクティブになります。

```
public void action( Object e, int where, int index,
  String choice, boolean state )
    {
    try
      {
      switch ( where )
        {
        /* process Menu actions                                        */
        case Menu:
          switch ( index )
            {
            case  0: load( );          break;
            case  1: save( );          break;
            case  2: System.exit( 0 ); break;
            case  3: new examples.awt.AwtMQeTrace( "Edit Trace", null );
    break;
            }
          break;
        /* process North events                                        */
        case North:
          switch ( index )
            {
            case 0:
              String item = choiceBox.getSelectedItem();
              new EditorFieldsDisplay( "Editor - ["  + Item +  "]",
                                       fields.getFields( Item ) );
              break;
            }
          break;
        }
      }
    /* exception occured - show error in a modal dialog window       */
    catch ( Exception ex )
      {
      ex.printStackTrace();
      new examples.awt.AwtDialog( this,
   "Exception",
   examples.awt.AwtDialog.Show_OK,
   new String[][][] {
  { { "TP", ex.toString() }  } } );
      }
    }
```

次のコードは、Save メニュー要求を処理します。共通ファイル・ダイアログが作成および表示され、出力ファイル名を指定することができます。一度ファイル・パスとファイル名が設定されると、今度はすべてのフィールド・オブジェクトを含む String が一つ作成され、組み込まれた各フィールド・オブジェクトが String 変数にダンプされます。この String はディスクに書き込まれ、出力ファイルはクローズします。

```
protected void save( ) throws Exception
    {
    if ( fields == null )   throw new Exception( "No Fields object" );
    FileDialog fd  = new FileDialog( this, "", FileDialog.SAVE );
    fd.setFile( CurrentFile );
    fd.show( );
    if ( (fd.getDirectory() != null) && (fd.getFile() != null) )
      {
      currentFile   = fd.getDirectory() + fd.getFile();
```

```
      File diskFile = new File( currentFile );
      /* look for imbedded fields objects   */
      String buffer = "";                /*for imbedded_fields objects*/
      String base   = "";                  /*non-imbedded_fields items*/
      Enumeration keys = fields.fields();              /*get the names*/
      while ( keys.hasMoreElements() )
        {
        String key = (String) keys.nextElement();
        if  ( fields.dataType( key )  == MQeField.TypeFields )
          buffer = buffer + "[" + key + "]¥r¥n" +
                   fields.getFields( key ).dumpToString(
   "(#0)#1=#2¥r¥n" ) + "¥r¥n";
        else                                      /*... no, normal item*/
          base   = base   + fields.dumpToString( "(#0)#1=#2¥r¥n", key );
        }
      buffer = base + buffer;
      FileOutputStream outputFile = new FileOutputStream( diskFile );
      outputFile.write( MQe.asciiToByte( buffer ) );
      outputFile.close( );
      }
    }
```

ここまででアプリケーションを制御するウィンドウを定義してきました。また、ディスク・ファイルのロードと保管の処理、特定のセクションが選択された場合に EditorFieldsDisplay クラスをアクティブにするメカニズムも定義してきました。

EditorFieldsDisplay クラスは実際の編集が行われる場所です。

このクラスはエディター画面を作成します。クラスのコンストラクターはメニュー (この場合は North に Exit のみ)、組み込みフィールド・オブジェクトの名前をすべて含む選択ボックス、および中央にダンプされた項目を保持するリスト・ボックスを設定します。

次のコードは画面上にサブウィンドウを配置します。

```
public class EditorFieldsDisplay extends AwtFrame
  {
  protected MQeFields fields     = null;        /* fields object       */
  protected Choice    choiceBox  = null;        /* Fields Choice       */
  protected List      listBox    = null;        /* listbox object      */
  protected String    newItem    =
  "  <<<< Double click here to add new item >>>>";
  /* constructor                                                       */
  public EditorFieldsDisplay( String    thisTitle,
                              MQeFields theseFields ) throws Exception
    {
    super( thisTitle );
    fields = theseFields;
    format( Menu,   new String[][][] {
                              { { "Exit"    },
                                {  " ", "Exit"   } } } );/* Index  0   */
    format( North,  new String[][][] {
                              { { "D",  "<none>" } } } );/* Index  0   */
    format( Center, new String[][][] {
                              { { "S",  "" } } } );      /* Index  0   */
    choiceBox = (Choice) getObject( North,  0 );
    listBox   = (List)   getObject( Center, 0 );
    listBox.setFont( new Font( "Courier", 1, 12) );
    visible( true );
    /* re-size/re-position the edit window                             */
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    setSize    ( screenSize.width / 3, screenSize.height / 3 );
```

```
    setLocation( screenSize.width / 3, screenSize.height / 3 );
    /* initialise the various component contents                        */
    showFields( ); /* show fields contents  */
    }
```

showFields メソッド呼び出しは、フィールド・オブジェクト内のアイテムにある List ボックスに含まれるデータをリフレッシュする共通ルーチンへの呼び出しです。

```
protected void showFields( ) throws Exception
    {
    listBox.removeAll();               /* clear all entries            */
    if ( fields != null )              /* fields object ?              */
      {                                /* ... yes                      */
      Enumeration keys = fields.Fields();       /* get the key names   */
      choiceBox.removeAll();
      while ( keys.hasMoreElements() )
        {
        String key = (String) keys.nextElement();
        if  ( fields.dataType( key )  == MQeField.TypeFields )
          choiceBox.add( key );        /* ... yes, add name            */
        else
          listBox.add( format( fields.dumpToString( "#1¥t(#0)¥t = #2",
         Key), 10 ) );
        }
      listBox.add( newItem );          /* add information line         */
      }
    }
```

直前のコードにかかれている ListBox.add( format( fields.dumpToString( "#1¥t(#0)¥t = #2" は、タブ (″¥t″)、復帰 (″¥r″)、および 改行 (″¥n″) 文字でフィールド・データがダンプされるようにします。これらはリスト・ボックスに表示される前にフォーマットすることが必要です。次のコードが示しているのはフォーマッターです。

```
public static String format( String data, int tabSize )
    {
    int l = 0;                                      /*start line number*/
    char c[] = new char[data.length()];                   /*work array*/
    data.getChars( 0, data.length(), c, 0 );       /*convert to chars*/
    StringBuffer result = new StringBuffer( 512 );
    for ( int i = 0; i < c.length; i = i + 1 )
      switch ( c[i] )
        {
        case '¥r': /* ignore */
        case '¥n': /* new line */
           l = 0; /* set space count */
           result.append( c[i] );                  /*append to string*/
           break;
        case '¥t':                                    /*tab character*/
          int m = l;                               /*current position*/
          for ( l = m; l < tabSize + 1; l = l + 1 )        /*fill tab*/
            result.append( ' ' );                            /*pad*/
          l = 0;                                           /*reset*/
          break;
        default:                                        /*all others*/
          result.append( c[i] );                   /*append to string*/
          l = (l + 1) % tabSize;                /*increase the length*/
          break;
        }
    return( result.toString() );
    }
```

### ini エディターに基づいたフィールド

次のコードは、ユーザーがメニュー、選択ボックスまたはリスト・ボックスで対話することによって引き起こされたイベントを処理します。

メニュー項目は Exit だけで、これは編集ウィンドウを閉じることによって処理されます。選択ボックスは組み込みフィールド項目をすべて処理します。個々の項目はリスト・ボックス内の項目を選択することによって編集されます。

```
public void action( Object e, int where, int index,
 String choice, boolean state )
  {
  try
    {
    switch ( where )
      {
      /* process Menu events                                          */
      case Menu:
        switch ( index )
          {
          case  0:  dispose( );       break;
          }
        break;
      /* process North events                                         */
      case North:
        switch ( index )
          {
          case  0:         break;
          }
        break;
      /* process Center events                                       */
      case Center:
        switch ( index )
          {
          case 0:
            int i = listBox.getSelectedIndex();
            if  ( i > -1 )                    /* anything selected ? */
              {
              String editName  = listBox.getItem( i );
              if ( editName.equals( newItem ) )    /* add new item ? */
                editItem( "", "ascii", "" );  /* ... yes,            */
              else
                {
                editName = editName.substring( 0,
                                               editName.indexOf( ' ' ) );
                editItem( editName,
                          fields.dumpToString( "#0", editName ),
                          fields.dumpToString( "#2", editName ) );
                }
              }
            break;
          }
        break;
      }                             /* end switch( Where )          */
    }
  /* exception occured - show error in a modal dialog window        */
  catch ( Exception ex )
    {
    ex.printStackTrace();
    new AwtDialog( this,
                   "Exception",
                   AwtDialog.Show_OK,
                   new String[][][] { { { "TP", ex.toString() }  } } );
    }
  }
```

リスト・ボックス内のリストが選択されると、Edit ダイアログが表示されます。ここでは名前、型、値を編集できます。フィールド・オブジェクトからその項目を削除することも可能です。

同じダイアログを使って、フィールド・オブジェクトに新しい項目を追加することもできます。この場合、項目名と値はブランクで、型にはデフォルトで ascii が設定されます。

```
protected void editItem( String name, String type, String value )
    throws Exception
    {
    if ( fields == null )   throw new Exception( "No Fields object" );
    /* Dialog to set Field Item name type and value                    */
    AwtDialog md = new AwtDialog( this,
                                  getTitle() + " - edit item",
                                  AwtDialog.Show_OK_Cancel,
                                  new String[][][] {
        { { "L", "Field Item Name:" }, { "T", name  } },    /* Index 1 */
        { { "L", "      Data type:" }, { "D", type,         /* Index 3 */
                                         "ascii",
                                         "boolean",
                                         "byte",
                                         "double",
                                         "float",
                                         "int",
                                         "long",
                                         "short",
                                         "unicode" } },
        { { "L", "          Value:" }, { "T", value    } }, /* Index 5 */
        { { "L", "   Remove item ?" }, { "C", "Delete" } }  /* Index 7 */
                                              } );
    /* process dialog response                                        */
    if ( md.getActionIndex( South ) == md.Button_OK )
      {
      name = md.GetText( Center, 1 );
      if ( name.equals( "" ) )
        throw new Exception( "Invalid Item name" );
      fields.delete( name );
      if ( ! md.getCheckState( center, 7 ) )   /* delete this item ?   */
        {                                     /* ... no               */
        String data = "(" + md.GetText( Center, 3 ) +
                      ")" + name +
                      "=" + md.getText( Center, 5 );
        fields.restoreFromString( "(#0)#1=#2", data );
        }
      showFields( );
      }
    }
```

これで、実用的ではあるもののいくぶん素朴な ini ファイル・エディターの完成です。データが MQeAttribute メカニズムを使ってコード化されていない限り、MQeMsgObjects (MQeFields の下位) を表示および修正するのに、このエディターを使用することができます。

このプログラムはすべての形式の MQeFields を包含するという意味はなく、より強力なエディターの例または出発点を意図したものです。

# 第4章 MQSeries Everyplace キュー・マネージャー

MQSeries Everyplace キュー・マネージャーは、MQSeries Everyplace システムの中心拠点です。これは次のものを提供します。

- MQSeries Everyplace アプリケーションのための MQSeries Everyplace/MQSeries ネットワークへのアクセスの中央点
- 1 回だけの確実なメッセージ送達
- 障害状態からの完全なリカバリー
- 拡張可能なルールに基づく動作

MQSeries Everyplace キュー・マネージャーは、オブジェクト指向スタイルで設計されています。キュー・マネージャーはその機能を継承し、拡張することができます。さらに、一連のルールを提供することによって、キュー・マネージャーの動作をカスタマイズすることができます。

MQSeries Everyplace キュー・マネージャーは、クライアント上で、または MQSeries Everyplace サーバーの一部として実行することができます。

MQSeries Everyplace キュー・マネージャーは、2 つのセットのキューを制御します。すなわち、ローカル・キューとリモート・キューの定義です。ローカル・キュー・マネージャーにあるキューは、ローカル・キューと呼ばれます。一般に、ローカル・キューはそのメッセージをローカル・マシン上にある永続ストアに保管しますが、メッセージ・ストアの実際の場所は構成することが可能です。

キュー・マネージャーは、MQSeries Everyplace/MQSeries ネットワーク内の他のキュー・マネージャーに属するキューに接続することができます。これらのキューは、リモート・キューと呼ばれ、リモート・キュー・マネージャーに属します。ローカル・キュー・マネージャーは、そのキューについて検出した情報をローカルに保管しているため、それらのキューの属性についていくらか知っています。この情報は、*リモート・キュー定義* と呼ばれます。

## メッセージの伝送

リモート・キューへのメッセージ伝送は、同期または非同期のどちらの方式でも行えます。伝送スタイルは、それぞれのリモート・キューごとに、発信元キュー・マネージャーが保持するリモート・キュー定義で定義されます。

同期通信を行うには、発信元と宛先の両方の MQSeries Everyplace キュー・マネージャーが、MQSeries Everyplace ネットワーク上で同時に使用可能になっていなければなりません。

非同期通信では、MQSeries Everyplace アプリケーションは、キュー・マネージャーがオフラインのときでもメッセージを送信することができます。非同期として定義されたキューへの出力メッセージは、送信することが可能か、または送信することが適切と思われるときまで (あるいはその両方)、ローカル・キュー・マネージャー内に保管されます。 MQSeries Everyplace アプリケーションは、通常どおり継続す

ることができます。メッセージをいつ送信するかを決めるのはキュー・マネージャー・ルールの主なタスクの 1 つであり、特に、通信の可用性が制限されている場合やコストに関する考慮事項がある場合に関係があります。

非同期通信では、宛先キュー・マネージャーおよびキューが、発信元キュー・マネージャー上で事前定義されていなければなりません。これが必要なのは、メッセージの送達を確実に行うために、発信キュー・マネージャーは、宛先キュー・マネージャー / キューの対が有効となるよう設定する必要があるためです。

発信元キュー・マネージャー上の宛先キュー定義が、実際のキューの属性によって最新のものに保たれていなければ、確実な送達は行えません。実際のキューとそのリモート定義は、リモート管理を使用するステップで保持されます (113ページの図 18 を参照)。

## 確実なメッセージ送達

MQSeries Everyplace は、前述の条件を満たす場合に、メッセージの送達を 1 回だけ確実に行うことができます。確実な送達を行うには、余分なフローを伝送しなければならないので、これは必須ではありません。 MQSeries Everyplace アプリケーションは、1 つ 1 つのメッセージに対して、確実な送達を使用するかどうか指定することができます。

確実な送達を使用するときには、それに続いて確認が発行されるまで、メッセージは宛先キュー上でロックされます。確認は、受信または送信が成功したときだけ発行されるはずです。

## セキュリティー

キュー・マネージャーは、MQSeries Everyplace に提供されるセキュリティー機能を完全にサポートします。セキュリティー特性を使用して定義されたキューに保管されているメッセージは、それらの特性を使ってエンコードされます。キュー・マネージャーとセキュア・キューとの間でセットアップされた通信チャネルは、キューのセキュリティー特性を使用します。または、同等かそれ以上のセキュリティーを持つ既存のチャネルを使用することもあります。

セキュリティー特性をメッセージに直接付加することにより、メッセージを個々に保護することができます。この方法で保護されたメッセージを扱うときには必ず、正確な特性が存在していなければなりません。

MQSeries Everyplace セキュリティーの詳細については、175ページの『第7章 セキュリティー』を参照してください。

## キュー・マネージャーの MQeRegistry パラメーター

レジストリーは、キュー・マネージャーに関連した情報のための基本ストアです。キュー・マネージャーごとに 1 つのレジストリーがあります。すべてのキュー・マネージャーは、レジストリーを使用して、次のものを保持します。

- キュー・マネージャー構成データ
- キュー定義

- リモート・キュー定義
- リモート・キュー・マネージャー定義
- ユーザー・データ (構成に依存するセキュリティー情報を含む)

このセクションでは、レジストリーに関連したキュー・マネージャー始動パラメーターについて解説します。

## レジストリー・タイプ

次のパラメーターは、開いているレジストリーのタイプを指定します。現在サポートされているタイプは、*ファイル・レジストリー* と*私用レジストリー* です。いくつかのセキュリティー機能では、*私用レジストリー* が必須です。 175ページの『第7章 セキュリティー』を参照してください。

**MQeRegistry.LocalRegType (ASCII)**

ファイル・レジストリーの場合、このパラメーターは次のように設定する必要があります。

```
com.ibm.mqe.registry.MQeFileSession
```

私用レジストリーの場合、次のように設定する必要があります。

```
com.ibm.mqe.registry.MQePrivateSession
```

これらの値を表すのに、別名を使用することができます。

## ファイル・レジストリー・パラメーター

ファイル・レジストリーには次のパラメーターが必要です。

**MQeRegistry.DirName (ASCII)**
レジストリー・ファイルを保持するディレクトリーの名前。

## 私用レジストリー・パラメーター

私用レジストリーの場合、次のパラメーターを使用することができます。最初の 3 つのパラメーターは常に必要とされ、最後の 2 つのパラメーターは、ミニ認証発行サーバーから証明書を取得したい場合に、レジストリーの自動登録のために必要とされるものです。

**MQeRegistry.DirName (ascii)**
レジストリー・ファイルを保持するディレクトリーの名前。

**MQeRegistry.PIN (ascii )**
私用レジストリー用の PIN。

**MQeRegistry.KeyRingPassword (ascii)**
レジストリーの秘密鍵を保護するために使用されるパスワードまたは句。

**MQeRegistry.CAIPAddrPort (ascii)**
ミニ認証発行サーバーのアドレスおよびポート番号。

**MQeRegistry.CertReqPIN (ascii)**
レジストリーが証明書を取得できるようにするため、ミニ認証管理者によって事前に割り振られる認証要求番号。

### 共通パラメーター

どのタイプのレジストリーでも、デフォルト以外の区切り文字が使用される場合には、さらにこのパラメーターが必要です。

**MQeRegistry.Separator (ascii)**

項目名のコンポーネント間の区切り文字として使用される文字。たとえば、<QueueManager><Separator><Queue>

このパラメーターはストリングとして指定されますが、単一文字でなければなりません。複数の文字が含まれている場合には、最初の文字だけが使用されます。

レジストリーを開くときには必ず、同じ区切り文字を使用しなければなりません。一度レジストリーを使用し、そこに項目を入れたならば、区切り文字を変更してはなりません。

このパラメーターが指定されていない場合は、デフォルトの ″+″ が使用されます。

## キュー・マネージャーの作成および削除

キュー・マネージャーは少なくとも、次のものを必要とします。

- レジストリー
- キュー・マネージャー定義
- ローカル・デフォルト・キュー定義

一度これらの定義が行われると、キュー・マネージャーを実行し、管理インターフェースを使用して、構成 (キューの追加など) をさらに実行することができます。

これらの初期オブジェクトを作成するためのメソッドは、MQeQueueManagerConfigure クラスで提供されます。

インストール・プログラムの例 `examples.install.SimpleCreateQM` および `examples.install.SimpleDeleteQM` は、このクラスを使用します。

このセクションでは、MQeQueueManagerConfigure クラスの使用について詳しく説明します。

### キュー・マネージャーの作成

キュー・マネージャーを作成するために必要な基本ステップは、次のとおりです。

1. MQeQueueManagerConfigure のインスタンスを作成し、アクティブにする
2. キュー・マネージャーの特性を設定する
3. キュー管理プログラムの定義を作成する
4. デフォルト・キューの定義を作成する
5. MQeQueueManagerConfigure インスタンスをクローズする

これらのステップが正常に完了したならば、キュー・マネージャーをアクティブにして実行することができます。

## MQeQueueManagerConfigure のインスタンスを作成してアクティブにする

MQeQueueManagerConfigure クラスをアクティブにするには、空のコンストラクターを呼び出した後、activate() を実行します。

```
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( );
 qmConfig.activate( parms, "qmName¥¥Queues¥¥" );
}
catch (Exception e)
{ ... }
```

または、次のようにパラメーターを指定してコンストラクターを呼び出します。

```
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName¥¥Queues¥¥" );
}
catch (Exception e)
{ ... }
```

最初のパラメーターは、キュー・マネージャーの初期化パラメーターを含む MQeFields オブジェクトです。これらは、キュー・マネージャーが続けて実行されるときに必要な最小セットです。 MQeQueueManagerConfigure のパラメーターには、以下のものが含まれていなければなりません。

- ″QueueManager″ と呼ばれる組み込み MQeFields オブジェクト。キュー・マネージャーの名前が入ります。
- ’レジストリー’ と呼ばれる組み込み MQeFields オブジェクト。 LocalRegType および DirName が入ります。基本ファイル・レジストリーが使用される場合には、これらが必要なすべてのパラメーターです。私用レジストリーが使用される場合には、加えて PIN および KeyRingPassword も必要です。

2 番目のパラメーターは、キュー・ストアの位置です (デフォルト・キューが作成された、ファイル・システム内のディレクトリー)。ディレクトリー名は、キュー・マネージャー定義の一部として保管され、それ以降のキュー定義でキュー・ストアのデフォルト値として使用されます。レジストリーの場合と同様、ディレクトリーは必ずしも存在している必要はなく、必要なときに作成されます。

初期化パラメーターのいずれかが別名を使用する場合、または別名を使用してチャネル属性ルール名 (後述) を設定する場合には、MQeQueueManagerConfigure をアクティブにする前に別名を定義する必要があります。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
```

```
 // initialise the parameters
 MQeFields qmgrFields  = new MQeFields();
 MQeFields regFields   = new MQeFields();
 // Queue manager name is needed
 qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
 // Registry information
 regFields.putAscii(MQeRegistry.LocalRegType, "FileRegistry");
 regFields.putAscii(MQeRegistry.DirName, "qmname¥¥Registry");
 // add the embedded MQeFields objects
 parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
 parms.putFields(MQeQueueManager.Registry, regFields);
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName¥¥Queues¥¥" );
}
catch (Exception e)
{ ... }
```

## キュー・マネージャー特性の設定

MQeQueueManagerConfigure がアクティブになっており、キュー・マネージャー定義がまだ作成されていなければ、一部のキュー・マネージャー特性を設定することができます。

- setDescription() を使用すれば、キュー・マネージャーに説明を追加することができます。
- setChannelTimeout() を使用すれば、チャネル・タイムアウト値を設定することができます。
- setChnlAttributeRuleName() を使用すれば、チャネル属性ルールの名前を設定することができます。

## キュー・マネージャー定義の作成

これは、defineQueueManager( ) を呼び出すことによって行われます。これは、以前に設定された特性を含んでいる、キュー・マネージャー用のレジストリー定義を作成します (defineQueueManager() を呼び出す前にそれらの特性を設定しなければならないのは、このためです)。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName¥¥Queues¥¥" );
 qmConfig.setDescription("a test queue manager");
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmConfig.defineQueueManager();
}
catch (Exception e)
{ ... }
```

この時点で、MQeQueueManagerConfigure をクローズして、キュー・マネージャーを実行することができます。ただし、キュー・マネージャーにはキューがないので、多くのことは行えません。管理インターフェースを使ってキューを追加することもできません。なぜなら、管理メッセージを保守するための管理キューがないからです。

次のセクションでは、キューの作成方法およびキュー・マネージャーを役立てる方法を説明します。

## デフォルト・キューの定義の作成

MQeQueueManagerConfigure により、キュー・マネージャーの 4 つの標準キューを定義することができます。

- 管理キュー : defineDefaultAdminQueue()
- 管理応答キュー : defineDefaultAdminReplyQueue()
- 送達不能キュー : defineDefaultDeadLetterQueue()
- デフォルトのローカル・キュー : defineDefaultSystemQueue()

これらのメソッドはすべて、キューがすでに存在する場合には、例外を出します。

管理キューおよび管理応答キューは、キュー・マネージャーが管理メッセージに応答できるようにするため (たとえば、新規の接続定義およびキューを作成するため) に必要です。

正しい送信先に送達できないメッセージについては、送達不能キューを使用することができます (試行されているルールに基づきます)。

デフォルトのローカル・キュー SYSTEM.DEFAULT.LOCAL.QUEUE は、MQSeries Everyplace 自体の中では特別な意味はありませんが、MQSeries Everyplace が MQSeries メッセージ機能とともに使用されるときに役に立ちます。なぜなら、それはすべての MQSeries メッセージング・キュー・マネージャーに存在するからです。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName¥¥Queues¥¥" );
 qmConfig.setDescription("a test queue manager");
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmconfig.defineDefaultAdminQueue();
 qmconfig.defineDefaultAdminReplyQueue();
 qmconfig.defineDefaultDeadLetterQueue();
 qmconfig.defineDefaultSystemQueue();
}
catch (Exception e)
{ ... }
```

### MQeQueueManagerConfigure インスタンスのクローズ

キュー・マネージャーおよび必要なキューを定義したら、MQeQueueManagerConfigure をクローズして、キュー・マネージャーを実行することができます。次はその完全な例です。

```
import com.ibm.mqe.*;
import com.ibm.mqe.registry.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 MQeFields qmgrFields  = new MQeFields();
 MQeFields regFields   = new MQeFields();
 // Queue manager name is needed
 qmgrFields.putAscii(MQeQueueManager.Name, "qmName");
 // Registry information
 regFields.putAscii(MQeRegistry.LocalRegType, "FileRegistry");
 regFields.putAscii(MQeRegistry.DirName, "qmname¥¥Registry");
 // add the embedded MQeFields objects
 parms.putFields(MQeQueueManager.QueueManager, qmgrFields);
 parms.putFields(MQeQueueManager.Registry, regFields);
 // set aliases
 MQe.alias("FileRegistry", "com.ibm.mqe.registry.MQeFileSession");
 MQe.alias("ChannelAttrRules", "examples.rules.AttributeRule");
 // activate the configure object
 qmConfig = new MQeQueueManagerConfigure( parms, "qmName¥¥Queues¥¥" );
 qmConfig.setDescription("a test queue manager");
 qmConfig.setChnlAttributeRuleName("ChannelAttrRules");
 qmConfig.defineQueueManager();
 qmconfig.defineDefaultAdminQueue();
 qmconfig.defineDefaultAdminReplyQueue();
 qmconfig.defineDefaultDeadLetterQueue();
 qmconfig.defineDefaultSystemQueue();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

キュー・マネージャーと必要なキューのレジストリー定義が作成されました。キューが初めてアクティブになるとき、キューそのものが作成されます。

## キュー・マネージャーの削除

キュー・マネージャーを削除するために必要な基本ステップは、次のとおりです。

1. 管理インターフェースを使用して、定義を削除する
2. MQeQueueManagerConfigure のインスタンスを作成し、アクティブにする
3. 標準キューの定義を削除する
4. キュー・マネージャーの定義を削除する
5. MQeQueueManagerConfigure インスタンスをクローズする

これらのステップが完了すると、キュー・マネージャーは削除され、それ以上実行することができなくなります。キュー定義は削除されますが、キューそのものは削除されません。キューに残ったメッセージは、アクセス不能になります。

**注:** キューにメッセージがある場合、自動的には削除されません。アプリケーション・プログラムには、キュー・マネージャーを削除する前に、残ったメッセージがあるかをチェックしてそれらを処理するためのコードが含まれている必要があります。

### 定義の削除

MQeQueueManagerConfigure は、作成した標準キューを削除することができます。その他のキューはいずれも、MQeQueueManagerConfigure を使用する前に、管理インターフェースを使って削除しておく必要があります。

### MQeQueueManagerConfigure のインスタンスを作成してアクティブにする

このプロセスは、キュー・マネージャーを作成するときと同じです。 37ページの『MQeQueueManagerConfigure のインスタンスを作成してアクティブにする』を参照してください。

### 標準キューの定義の削除

デフォルト・キューは、それぞれ次のものを呼び出すことによって削除することができます。

- 管理キューを削除するには、`deleteAdminQueueDefinition()`
- 管理応答キューを削除するには、`deleteAdminReplyQueueDefinition()`
- 送達不能キューを削除するには、`deleteDeadLetterQueueDefinition()`
- デフォルトのローカル・キューを削除するには、`deleteSystemQueueDefinition()`

これらのメソッドは、キューが存在していなくても正常に作動します。

### キュー・マネージャー定義の削除

キュー・マネージャーの定義は、`deleteQueueManagerDefinition()` を呼び出すことによって削除されます。

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteAdminQueueDefinition();
 qmConfig.deleteAdminReplyQueueDefinition();
 qmConfig.deleteDeadLetterQueueDefinition();
 qmConfig.deleteSystemQueueDefinition();
 qmConfig.deleteQueueManagerDefinition();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

### デフォルト・キューおよびキュー・マネージャーの同時削除

デフォルト・キューとキュー・マネージャーの定義は、
deleteStandardQMDefinitions() を 1 回呼び出すだけで削除することができます。
このメソッドは便宜上提供されており、次のものと同等です。

```
deleteDeadLetterQueueDefinition();
deleteSystemQueueDefinition();
deleteAdminQueueDefinition();
deleteAdminReplyQueueDefinition();
deleteQueueManagerDefinition();
```

### MQeQueueManagerConfigure インスタンスのクローズ

キューおよびキュー・マネージャーの定義を削除したら、
MQeQueueManagerConfigure インスタンスをクローズすることができます。

```
import com.ibm.mqe.*;
import examples.queuemanager.MQeQueueManagerUtils;
try
{
 MQeQueueManagerConfigure qmConfig;
 MQeFields parms = new MQeFields();
 // initialise the parameters
 ...
 // Establish any aliases defined by the .ini file
 MQeQueueManagerUtils.processAlias(parms);
 qmConfig = new MQeQueueManagerConfigure( parms );
 qmConfig.deleteStandardQMDefinitions();
 qmconfig.close();
}
catch (Exception e)
{ ... }
```

## キュー・マネージャーの開始

キュー・マネージャーは、次のように実行することができます。

- クライアントとして
- サーバー内で
- サーブレット内で

次のセクションでは、examples.queuemanager パッケージ内にある、クライアント、サーバー、およびサーブレットの例を説明しています。それらは基本的に、同じ基本 MQSeries Everyplace コンポーネントで構成され、それぞれに固有の特性を付与するものがいくらか加えられています。たくさんの機能を共用するため、それらの機能をカプセル化したユーティリティー・クラス MQeQueueManagerUtils が提供されます。

すべての例は、始動時にパラメーターを必要とします。これらのパラメーターは、標準 ini ファイルに保管されています。 ini ファイルは読み取ることができ、それらの中のデータは MQeFields オブジェクトに変換されます。これについては、21ページの『第3章 MQeFields および MQeMsgObject』を参照してください。
MQeQueueManagerUtils クラス内の loadConfigFile() メソッドがこの機能を実行します。

## クライアント

一般に、あるデバイス上のアプリケーションがキュー・マネージャーを使用する場合、そのデバイス上ではクライアントが実行します。クライアントは、他のキュー・マネージャー (一般に、サーバー) への多数の接続を開くことができ、対等チャネルを使用するよう構成されていれば、他のキュー・マネージャーからの着信要求を受信することができます。

一般に長時間実行するサーバーとは異なり、クライアントは必要に応じて、それらを使用するアプリケーションにより開始および停止されます。キュー・マネージャーが複数のアプリケーションによって共用される場合には、アプリケーションはクライアントの開始および停止を調整する必要があります。

前述のとおり、クライアントを開始するために必要なパラメーターは、ini ファイルに保管されています。典型的なクライアントの始動パラメーターは、次のとおりです。

```
*
* ExamplesMQeClient.ini
*   An example ini file for a simple MQe client
*
[Alias]
*
*    Event log class
*
(ascii)EventLog=examples.log.LogToDiskFile
*
*    Network adapter class
*
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*    Queue Manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
*    Trace handler (if any)
*
(ascii)Trace=examples.trace.MQeTrace
*
*    Message Log file interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
*    Class name for File registry
*
(ascii)FileRegistry=com.ibm.mqe.registry.MQeFileSession
*
*    Class name for Private registry
*
(ascii)PrivateRegistry=com.ibm.mqe.registry.MQePrivateSession
*
*    Default Channel class
*
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
*
*    Default Transporter class
*
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
*
*    Channel Attribute Rules
*
(ascii)ChannelAttrRules=examples.rules.AttributeRule
*
```

```
*    Name of Base Key
*
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
*
*    Name of Shared Key
*
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MQeSharedKey
*-------------------------------------------------------------------------*
*
*  Registry ( configuration data store )
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=FileRegistry
*
*   Location of the registry
*   (Only use relative directory for development/demo)
*
(ascii)DirName=.¥ExampleQM¥Registry¥
*-------------------------------------------------------------------------*
*
*  Queue manager details
*
[QueueManager]
*
*   Name for this Queue Manager
*
(ascii)Name=ExampleQM
```

QueueManager および Registry セクションの詳しい説明は、34ページの『キュー・マネージャーの MQeRegistry パラメーター』にあります。別名セクションには、別名を設定できる場所があります。一般に、別名を使用して、クラス名の別名を提供します。別名は MQSeries Everyplace によって使用されますが、アプリケーションと実際のクラス名との間の間接レベルを提供するためにアプリケーション・プログラムが使用することもあります。そのため、別名が参照するクラスを変更するときに、アプリケーションがそれを行う必要はありません。別名を使用することで、構成を容易に変更することができます。たとえば、どのトレース・ハンドラーを使用するかを指定する Trace に、別名を設定します。 MQSeries Everyplace はいくつかの異なったトレース・ハンドラーを提供します。 Trace の別名を変更するだけで、MQSeries Everyplace が使用するトレース・ハンドラーが変更されます。

次のコード・セクションは、クライアントを開始します。

```
/*-------------------------------------------------------------------------*/
/* Init - first stage setup                                                */
/*-------------------------------------------------------------------------*/
public void init( MQeFields parms ) throws Exception
{
  if ( queueManager != null )               /* One queue manager at a time   */
  {
    throw new Exception( "Client already running" );
  }
  sections = parms;                         /* Remember startup parms        */
  MQeQueueManagerUtils.processAlias( sections ); /* set any alias names      */
// Uncomment the following line to start trace before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeClient Trace", null); /* Turn trace on   */
  /* Display the startup parameters                                         */
  System.out.println( sections.dumpToString( "#1¥t=¥t#2¥r¥n" ) );
  /* Start the queue manager                                                */
  queueManager = MQeQueueManagerUtils.processQueueManager( sections, null );
}
```

クライアントの開始には、次の事柄が関係しています。

1. すでに実行しているクライアントがないことを確認します。 Java 仮想計算機 1 台につき 1 つのクライアントだけが許可されます。
2. システムに別名を追加します。
3. 必要であれば、トレースを使用可能にします。
4. キュー・マネージャーを開始します。

クライアントが一度開始されると、キュー・マネージャー・オブジェクトへの参照は、静的クラス変数 MQeClient.queueManager から、または静的メソッド MQeQueueManager.getReference(queueManagerName) を使用することによって、取得することができます。

次のコード断片は、システムに別名をロードします。

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )   /* section present ?            */
  {                                           /* ... yes                      */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );    /* get all the keywords         */
    while ( keys.hasMoreElements() )          /* as long as there are keywords*/
    {
      String    key   = (String) keys.nextElement();     /* get the Keyword   */
      MQe.alias( key, section.getAscii( key ).trim( ) ); /* add               */
    }
  }
}
```

それぞれの別名ごとに、システムに別名を追加するための MQSeries Everyplace 別名メソッドが使用されます。 MQSeries Everyplace とアプリケーションのどちらもロードされると、別名を使用することができます。 MQSeries Everyplace が正しく機能するためには、上記の ini ファイルで示されたたくさんの別名が必要になるので、それらを除去しないでください。

次のコード断片は、キュー・マネージャーを開始します。

```
public static MQeQueueManager processQueueManager( MQeFields sections,
 Hashtable ght ) throws Exception
{                                             /*                            */
  MQeQueueManager queueManager = null;        /* work variable              */
  if ( sections.contains( Section_QueueManager) )     /* section present ?    */
  {                                           /* ... yes                      */
    queueManager = (MQeQueueManager) MQe.loader.loadObject(Section_QueueManager);
    if ( queueManager != null )               /* is there a Q manager ?       */
    {
      queueManager.setGlobalHashTable( ght );
      queueManager.activate( sections );      /* ... yes, activate            */
    }
  }
  return( queueManager );                     /* return the alloated mgr      */
}
```

キュー・マネージャーの開始には、次の事柄が関係しています。

1. キュー・マネージャーのインスタンス化。ロードするキュー・マネージャー・クラスの名前は、別名 QueueManager で指定されます。 MQSeries Everyplace クラス・ローダーを使用して、クラスをロードし、空コンストラクターを呼び出します。

2. その後、ini ファイルのフィールド・オブジェクトの表現を渡す活動化メソッドを使用して、キュー・マネージャーがアクティブにされます。キュー・マネージャーは、始動パラメーターから QueueManager および Registry セクションだけを使用します。

## MQePrivateClient の例

MQePrivateClient は、MQeClient の拡張で、キュー・マネージャーとレジストリーをセキュア・キューで使用できるように構成する機能が追加されています。セキュア・クライアントの場合、始動パラメーターの Registry セクションは、次のように拡張されています。

```
* Extract from MQePrivateClient.ini
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=PrivateRegistry
*
*   Location of the registry
*
(ascii)DirName=.¥ExampleQM¥PrivateRegistry
*
* PIN
*
(ascii)PIN=not set
*
* Certificate request pin
*
(ascii)CertReqPIN=not set
*
* Key ring password
*
(ascii)KeyRingPassword=not set
*
* Network address of certificate authority
*
(ascii)CAIPAddrPort=9.20.7.219:8081er
```

これらのフィールドについては、34ページの『キュー・マネージャーの MQeRegistry パラメーター』で解説します。セキュア・キューおよび MQePrivateClient に詳細については、 175ページの『第7章 セキュリティー』に記載されています。

MQSeries 標準版が使用されている場合は、すべてのセキュリティー機能が使用可能であるとは限りません。 MQePrivateClient (および MQePrivateServer) を作動させるには、始動パラメーターにパラメーター CertReqPIN、KeyRingPassword および CAIPAddrPort が含まれていてはなりません。 MQSeries Everyplace 標準版を使用した MQePrivateClient のレジストリー・セクションは、次のようになります。

```
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=PrivateRegistry
*
*   Location of the registry
*
(ascii)DirName=.¥ExampleQM¥PrivateRegistry
```

```
*
* PIN
*
(ascii)PIN=not set
```

## サーバー

一般に、サーバーはゲートウェイ上で実行します。サーバーは、サーバー側のアプリケーションを実行することができますが、必要であれば、クライアント側のアプリケーションも実行することもできます。クライアントの場合と同じように、サーバーは、サーバー、クライアント、およびゲートウェイを含む、他の多くのキュー・マネージャーへの接続を開くことができます。クライアントと異なる主な特性の 1 つは、たくさんの同時着信要求を処理できるという点です。一般に、サーバーはたくさんのクライアントのための MQSeries Everyplace ネットワークへのエントリー・ポイントとして働きます。 MQSeries Everyplace は、以下の例のように、いくつかの異なったサーバーを提供します。

**MQeServer**
: 単純な、コンソール・ベースのサーバー

**MQePrivateServer**
: 拡張セキュリティーを持つ、単純な、コンソール・ベースのサーバー

**AwtMQeServer**
: MQeServer へのグラフィカル・フロントエンド

**MQBridgeServer**
: パッケージ examples.mqbridge.queuemanager にあるサーバー。これは、他のサーバーに加えて、MQSeries ファミリーの他のメンバーとの間でメッセージを送受信することができます。このサーバーについては、131ページの『第6章 MQSeries ブリッジ』を参照してください。

### MQeServer の例

単純なサーバーのインプリメンテーションが MQeServer です。

このサーバーは、次のコマンドによって開始することができます。

```
<javaCommand> examples.queuemanger.awt.MQeServer  <startupIniFile>
```

この場合

**javaCommand:**
: Java アプリケーションを開始するために使用されるコマンド (たとえば、java)

**startupIniFile:**
: キュー・マネージャーおよびサーバーの始動パラメーターを含む ini ファイル (たとえば、.¥ExamplesMQeServer.ini)

バッチ・ファイル ExamplesMQeServer.bat は、ini ファイル .¥ExamplesMQeServer.ini を使ってサーバーを開始するためのショートカットを提供します。クライアントの場合と同様、ini ファイルを使用して、サーバーを開始するために必要なパラメーターが保持されます。サーバーと一緒に使用する場合、

標準クライアント ini ファイルを拡張して、ChannelManager および Listener セクションを含める必要があります。サーバー始動パラメーターを拡張した典型的な例は、次のとおりです。

```
* Extract from ExamplesMQeServer.ini
*
[ChannelManager]
*
*   Maximum number of channels allowed
*
(int)MaxChannels=0
*--------------------------------------------------------------------------*
[Listener]
*
*   FileDescriptor for listening adapter
*
(ascii)Listen=Network::8081
*
*   FileDescriptor for Network read/write
*
(ascii)Network=Network:
*
*   Channel timeout interval in seconds
*
(int)TimeInterval=300
```

2 つのキュー・マネージャーが相互に通信するとき、MQSeries Everyplace はその 2 つのキュー・マネージャーの間のチャネルを開きます。チャネルは、キュー・マネージャー・パイプに対するキュー・マネージャーとして使用される論理エンティティーです。いつでも複数のチャネルを開くことができます。

ini ファイルの新規セクションは、チャネルの使用法を制御します。
ChannelManager セクションの MaxChannels パラメーターは、いつでも開くことのできるチャネルの最大数を制御します。 Listener セクションには、着信ネットワーク要求が処理される方法に関するパラメーターが含まれています。

**Listen** 着信ネットワーク要求を処理するために使用されるネットワーク・アダプター。たとえば、http アダプターまたは純正の tcpip アダプター。アダプター名だけでなく、アダプターが聴取を行う方法を示すパラメーターも渡すことができます。たとえば、'Listen=Network::8081' は、Network アダプターを使用することを意味します。ここで、Network は、ポート 8081 上で聴取を行う別名です。 (これは、Network 別名が HTTP または TCPIP アダプターのいずれかに設定されていることを前提としています。)

**Network**
このパラメーターを使用して、初期ネットワークが受け入れられたら、ネットワーク読み取り / 書き込み要求に使用されるアダプターを指定します。一般に、これは Listen パラメーターで使用されるアダプターと同じです。

**TimeInterval**
使用されていないチャネルがタイムアウトになるまでの時間 (秒)。チャネルは、単一のキュー・マネージャー要求よりも長く持続する永続的な論理エンティティーであり、ネットワークの破損後も存続することが可能なので、一定の時間、活動状態にないチャネルをタイムアウトにしなければならない場合があります。

MQeServer の構造は、MQeClient の構造の後に続きます。 init メソッドで一回初期化が実行されると、 activate メソッドを使用して、サーバーを活動化また非活動化が行われます。次のコードは、サーバーを開始するときに使用される init メソッドを示しています。

```
public void init( MQeFields parms ) throws Exception
{
  if ( initialised )                          /* Only one server at a time     */
    throw new Exception( "Server already running" );
  sections = parms;                           /* Remeber startup parms         */
  MQeQueueManagerUtils.processAlias( sections ); /* set any alias names       */
// Uncomment the following line to start trace before the queue manager is started
//  MQeQueueManagerUtils.traceOn("MQeServer Trace", null); /* Turn trace on   */
  /* Display the startup parameters                                           */
  System.out.println( sections.dumpToString( "#1¥t=¥t#2¥r¥n" ) );
}
```

1. init メソッドがサーバー始動パラメーターに渡されます。
2. JVM ごとに 1 台のサーバーだけが実行されるように検査が行われます。
3. 別名がロードされ、必要であればトレースが使用可能になります。

サーバーが初期化されると、パラメーターを true に設定した activate メソッドを使って、サーバーをアクティブにすることができます。アクティブにされたならば、パラメーターを false に設定して activate メソッドを呼び出すことにより、非活動化することができます。

```
public void activate( boolean Start ) throws Exception
{
  if ( Start )                                /* activate ?                    */
  {                                           /* ... yes                       */
    if ( ! initialised )                      /* been here before              */
    {                                         /* ... no                        */
      /* allocate Chan Mgr                                                     */
      channelManager  = MQeQueueManagerUtils.processChannelManager( sections );
      /* assign any class aliases                                              */
      MQeQueueManagerUtils.processAlias( sections );
      /* check for any pre-loaded classes                                      */
      loadTable = MQeQueueManagerUtils.processPreLoad( sections );
      initialised = true;                     /* only once                     */
    }                                         /*                               */
    /* setup and activate the queue manager                                    */
    queueManager = MQeQueueManagerUtils.processQueueManager( sections,
   channelManager.getGlobalHashtable( ) );
    /* setup and activate the listener for incomming connections               */
    channelListener = MQeQueueManagerUtils.processListener(
   sections, channelManager );
  }
  else                                        /* ... no                        */
  {                                           /*                               */
    if ( channelListener != null )  channelListener.stop( );
    if ( queueManager    != null )  queueManager.close( );
    channelListener = null;                   /* release object                */
    queueManager    = null;                   /* release object                */
  }
}
```

サーバーを活動化すると、次の事柄が生じます。

1. チャネル・マネージャーが開始されます。
2. ユーザー指定のクラスが追加してロードされ、空コンストラクターが呼び出されます。
3. キュー・マネージャーが開始されます。

4. チャネル・リスナーが開始されます。

リスナーが開始されると、サーバーが始動し、ネットワーク要求を受け入れる準備が整います。

サーバーを非活動化すると、次のようになります。

1. チャネル・リスナーが停止し、新規着信要求が妨げられます。
2. キュー・マネージャーがクローズされます。

MQeQueueManagerUtils クラスのコードの次のセクションは、それぞれのコンポーネントを処理します。まず、チャネル・マネージャーを開始するセクションは次のとおりです。

```
public static MQeChannelManager processChannelManager( MQeFields sections )
throws Exception
{
  MQeChannelManager channelManager = null;    /* work variable               */
  if ( sections.contains( Section_ChannelManager ) )  /* section present ?    */
  {                                           /* ... yes                     */
    MQeFields section = sections.getFields( Section_ChannelManager );
    channelManager    = new MQeChannelManager( );     /* load the manager     */
    channelManager.numberOfChannels( section.getInt( "MaxChannels" ) );
  }                                           /*                             */
  return( channelManager );                   /* return the allocated mgr    */
}
```

このメソッドは、チャネル・マネージャーをインスタンス化した後、始動パラメーターの ChannelManager セクションの MaxChannels パラメーターを使って、許可されるチャネルの最大数を設定します。

また、キュー・マネージャーがロードされるときにロードすることのできるクラスのセットを指定することもできます。これらは、ini ファイルの PreLoad セクションに追加することができます。ここで、ロードされるクラスは (ascii)uniqueName=class という形式でなければなりません。たとえば、次のようにします。

```
[PreLoad]
*
*   Classes to load at server startup
*
(ascii)StartClass1=test.ServerTest1
(ascii)StartClass2=test.ServerTest2
```

コードの次のセクションを使用して、プリロード・クラスをロードします。

```
public static Hashtable processPreLoad( MQeFields sections ) throws Exception
{
  Hashtable loadTable = new Hashtable();      /* allocate load table         */
  if ( sections.contains( Section_PreLoad ) ) /* section present ?           */
  {                                           /* ... yes                     */
    MQeFields section = sections.getFields( Section_PreLoad );
    Enumeration keys  = section.fields();     /* get all the keywords        */
    while ( keys.hasMoreElements() )          /* as long as there are keywords*/
    try                                       /* incase of error             */
    {                                         /*                             */
      String key = section.getAscii( (String) keys.nextElement() ).trim( );
      loadTable.put( key, MQe.loader.loadObject( key ) );
    }
    catch ( Exception e )                     /* error occured               */
    {
      e.printStackTrace();                    /* show the error              */
```

```
    }
  }
  return( loadTable );                        /* return the table            */
}
```

PreLoad セクションで指定されるそれぞれのクラスごとに、次のようになります。

1. MQeLoader を使用して、クラスがロードされます。これはクラスの空コンストラクターを呼び出すので、初期化コードまたは始動コードは、このコンストラクターに入れる必要があります。
2. ロードされると、クラスの参照がハッシュ・テーブルに置かれます。その後、このテーブルは、サーバー内の他のメソッドが使用することができます。たとえば、サーバーがクローズするときに、サーバーの close メソッドを拡張して、プリロードされたすべてのクラスの close メソッドを実行することができます。

## MQePrivateServer の例

MQePrivateServer は、MQeServer の拡張で、キュー・マネージャーとレジストリーをセキュア・キューで使用できるように構成する機能が追加されています。 175ページの『第7章 セキュリティー』を参照してください。

## AwtMQeServer の例

パッケージ examples.awt にある AwtMQeServer は、コンソール・ベースのサーバーに対するグラフィカル・フロントエンドを提供します。

サーバーは、次の呼び出しによって開始することができます。

```
<javaCommand> examples.awt.AwtMQeServer  <startupIniFile>
```

この場合

**javaCommand:**
Java アプリケーションを開始するために使用されるコマンド (たとえば、java)

**startupIniFile:**
キュー・マネージャーおよびサーバーの始動パラメーターを含む ini ファイル (たとえば、.¥ExamplesAwtMQeServer.ini)

バッチ・ファイル ExamplesAwtMQeServer.bat は、ini ファイル .¥ExamplesAwtMQeServer.ini を使ってサーバーを開始するためのショートカットを提供します。

AwtMQeServer は、次の別名を追加して使用します。

**Sever** このクラスがグラフィカル・フロントエンドを提供するサーバー・クラス。

**Admin**
管理コンソールを提供するクラスの名前。

ini ファイル例 .¥ExamplesAwtMQeServer.ini は、次のように別名を設定します。

```
*
*    Admin console (if any)
*
(ascii)Admin=examples.administration.console.Admin
*
```

```
*    Base Server class
*
(ascii)Server=examples.queuemanager.MQeServer
```

開始すると、次のウィンドウが表示されます。


*図 7. AWT MQe サーバー・ウィンドウ*

ボタンの機能は次のとおりです。

**「Exit (終了)」**
サーバーをクローズし、System.exit() を実行します。

**「Stop (停止)|Run (実行)」**
サーバーが実行中の場合、「Stop (停止)」はそれを停止します。サーバーが停止すると、ボタンは「Run (実行)」を表示します。これを使って、サーバーを開始します。

開始は、次のコードによって実行されます。

```
if ( running )                    /* running ?                    */
{
  setText( North, index, "Run"  );        /* ... yes,             */
  server.activate( false );               /* stop server          */
}
else
{
  setText( North, index, "Stop" );        /* ... no,  i.e start   */
  if ( server == null )                   /* initialized before ? */
  {                                       /* yes,                 */
    /* Load the startup parms and setup class aliases             */
    MQeFields sections
      = MQeQueueManagerUtils.loadConfigFile( iniName );
    MQeQueueManagerUtils.processAlias( sections );
    /* Load the server and initialise if first pass               */
    server = (MQeServer)MQe.loader.loadObject( "Server" );
    server.init( sections );
  }
  server.activate( true );        /* Activate the server          */
}
running = ! running;              /* change state                 */
```

**「Trace (トレース)」**
トレース別名が設定されている場合、それが現在アクティブでなければ、「Trace (トレース)」はトレースを活動化し、それがすでに実行中であれば、トレースを非活動化します。これは、次のコードによって実行されます。

```
/* Get current trace handler if any ..                            */
MQeTraceInterface    trace = MQe.getTraceHandler( );
if ( trace == null )              /* If trace is not on,start it  */
  MQeQueueManagerUtils.traceOn( this.getTitle() + " - Trace", null );
else                              /* otherwise stop it            */
  MQeQueueManagerUtils.traceOff();
```

**「Admin (管理)」**
管理別名が設定されている場合、管理コンソールがまだ開始されていなければ開始され、すでに実行していれば停止します。次のコードがこの機能をインプリメントします。

```
if ( adminGUI != null && adminGUI.active )
{                                   /* GUI active so               */
  adminGUI.close();                 /* close it                    */
  adminGUI = null;
}
else if ( adminGUI == null ||
        ( adminGUI != null && !adminGUI.active ) )
{                                   /* GUI not running or not active*/
  adminGUI = (Admin)MQe.loader.loadObject( "Admin" );
  adminGUI.activate();              /* so load and actiate it      */
}
```

**「Help (ヘルプ)」**
ダイアログについて表示します。

さらに、イベント・ロギングをオン / オフに切り替えて、使用するイベント・ロガーをドロップダウン・リスト・ボックスから選択することができます。次の選択が可能です。

- ログ記録しない
- examples.eventlog.LogToDiskFile
- examples.eventlog.LogToNTEventLog

## サーブレット

キュー・マネージャーは、スタンドアロン・サーバーとして実行するだけでなく、サーブレット内にカプセル化することによって、Web サーバー内で実行することができます。サーブレットとして実行するとき、キュー・マネージャーは、サーバーとして実行するときとほぼ同じ機能を持ちます。 MQeServlet はサーブレットのインプリメンテーション例を提供します。サーバーの場合と同様、サーブレットは ini ファイルを使用して始動パラメーターを保持しますが、その代わりにサーバー ini ファイルを使用することもできます。

サーブレットは、サーバーと同じ MQSeries Everyplace コンポーネントを多数使用します。サーブレットで必要でない主なコンポーネントはチャネル・リスナーで、この機能は Web サーバー自体が実行します。 Web サーバーは HTTP データ・ストリームしか処理しないので、MQSeries Everyplace サーブレットと通信する MQSeries Everyplace クライアントは、HTTP アダプターを使用する必要があります (com.ibm.mqe.adapters.MQeTcpipHttpAdaper)。サーブレット内で実行しているキュー・マネージャーへの接続を構成するとき、サーブレットの名前は、接続のパラメーター・フィールドで指定する必要があります。たとえば、サーブレット /servlet/MQSeries Everyplace 上のキュー・マネージャー PayrollQM と通信するには、次のように接続を構成する必要があります。

***接続名 :***
PayrollQM

***チャネル :***
com.ibm.mqe.MQe

## サーブレット・キュー・マネージャー

***チャネル・アダプター：***

com.ibm.mqe.adapters.MQe

***TcpipHttpAdaper:***

192.168.0.10:80

***パラメーター：***

/servelet/MQe

***オプション***

あるいは、関係のある別名が設定されている場合は、次のように接続を構成することができます。

***接続名：***

PayrollQM

***チャネル：***

DefaultChannel

***アダプター：***

Network:192.168.0.10:80

***パラメーター：***

/servelet/MQe

***オプション：***

Web サーバーは複数のサーブレットを実行することができます。 Web サーバー内では複数の異なる MQSeries Everyplace サーブレットを実行することができます。ただし、次のような制限があります。

- それぞれのサーブレットには固有の名前が必要です。
- 1 つのサーブレットあたり 1 つのキュー・マネージャーだけが許可されます。
- それぞれの MQSeries Everyplace サーブレットは、別々の Java 仮想計算機 (JVM) で実行しなければなりせん。

MQSeries Everyplace サーブレットは、`javax.servlet..http.HttpServlet` を拡張したものであり、新規要求を開始、停止、および処理するためのメソッドをオーバーライドします。次のコード断片は、サーブレットを開始します。

```
/**
 * Servlet Initialisation......
 */
public void init(ServletConfig sc) throws ServletException
{
  // Ensure supers constructor is called.
  super.init(sc);
  try
  {
    // Get the the server startup ini file
    String startupIni;
    if ( ( startupIni = getInitParameter("Startup")) == null )
      startupIni = defaultStartupInifile;
    // Load it
    MQeFields sections = MQeQueueManagerUtils.loadConfigFile(startupIni);
    // assign any class aliases
    MQeQueueManagerUtils.processAlias( sections );
    // Uncomment the following line to start trace before the queue
    // manager is started
    //      MQeQueueManagerUtils.traceOn("MQeServlet Trace", null);
    // Start channel manager
    channelManager  = MQeQueueManagerUtils.processChannelManager( sections );
    // check for any pre-loaded classes
    loadTable = MQeQueueManagerUtils.processPreLoad( sections );
```

```
    // setup and activate the queeu manager
    queueManager = MQeQueueManagerUtils.processQueueManager( sections,
   channelManager.getGlobalHashtable( ) );
    // Start ChannelTimer  (convert timeout from secs to millsecs)
    int tI =
      sections.getFields(MQeQueueManagerUtils.Section_Listener).getInt( "TimeInterval" );
    long timeInterval = 1000 * tI;
    channelTimer = new MQeChannelTimer( channelManager, timeInterval );
    // Servlet initialisation complete
    mqe.trace( 1300, null );
  }
  catch (Exception e)
  {
    mqe.trace( 1301, e.toString() );
    throw new ServletException( e.toString() );
  }
}
```

サーバー始動と比較した場合の主な違いは、次のとおりです。

- サーブレットはスーパークラス `init` メソッドをオーバーライドします。このメソッドは、Web サーバーがサーブレットを開始するために呼び出されます。一般にこれが生じるのは、サーブレットの最初の要求が到着したときです。
- 始動パラメーター `ini` ファイルの名前は、サーバーの場合と同様にコマンド行から渡すことができません。例では、サーブレット・メソッド `getInitParameter()` を使用して、その名前を取得することになっています。このメソッドは、パラメーターの名前を取得して、値を戻すものです。 MQSeries Everyplace サーブレットは、`ini` ファイル名が入っていると予期される、`Startup` パラメーター名を使用します。 Web サーバー内でパラメーターを構成するメカニズムは、Web サーバーに依存しています。
- Web サーバーがサーブレットのためにすべてのネットワーク要求を処理するので、チャネル・リスナーは開始されません。
- チャネル・リスナーが存在しないため、長時間活動状態にないチャネルをタイムアウトにするためのメカニズムが必要です。この機能を実行するために、単純なタイマー・クラス MQeChannelTimer がインスタンス化されます。 `TimeInterval` 値が、`ini` ファイルの Listener セクションから使用される唯一のパラメーターです。

前述のとおり、サーブレットは着信要求の受信および処理を Web サーバーに依存しています。 Web サーバーが、要求が MQSeries Everyplace サーブレットに対するものであると判断すると、`doPost()` メソッドを使って、要求を MQSeries Everyplace に渡します。次のコードがこの要求を処理します。

```
/**
 * Handle POST......
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) throws IOException
{
  // any request to process ?
  if (request == null)
    throw new IOException("Invalid request");
  try
  {
    int max_length_of_data = request.getContentLength();     // data length
    byte[] httpInData = new byte[max_length_of_data];        // allocate data area
    ServletOutputStream httpOut = response.getOutputStream();// output stream
    ServletInputStream  httpIn  = request.getInputStream();  // input stream
    // get the request
    read( httpIn, httpInData, max_length_of_data);
    // process the request
    byte[] httpOutData = channelManager.process(null, httpInData);
```

```
    // appears to be an error in that content-length is not being set
    // so we will set it here
    response.setContentLength(httpOutData.length);
    response.setIntHeader("content-length", httpOutData.length);
    // Pass back the response
    httpOut.write(httpOutData);
  }
  catch (Exception e)
  {
    // pass it on ...
    throw new IOException( "Request failed" + e );
  }
}
```

このメソッドには、以下の処理が含まれます。

1. HTTP 入力データ・ストリームをバイト配列に読み取ります。入力データ・ストリームをバッファーに入れ、`read()` メソッドによって、処理を継続する前にデータ・ストリーム全体が確実に読み取られるようにすることもできます。

   **注:** MQSeries Everyplace は `doPost()` メソッドによってのみ要求を処理します。 `doGet()` メソッドを使用して要求を受け入れることはありません。
2. 要求はチャネル・マネージャーを介して MQSeries Everyplace に渡されます。この時点から、要求のすべての処理は、キュー・マネージャーなどのコア MQSeries Everyplace クラスによって処理されます。
3. MQSeries Everyplace が要求の処理を完了すると、HTTP ヘッダーにバイト配列として含められた結果を戻します。この結果は、要求を発信したクライアントに戻すために、Web サーバーに渡されます。

# 基本クラスを使用したキュー・マネージャーの構成

キュー・マネージャーを作成したり削除したりするには、MQeQueueManagerConfigure を使用することをお勧めしますが、このセクションでは、同じ機能を基本クラスから作成する方法を説明します。

## キュー・マネージャーの活動化

キュー・マネージャーが適正に活動化するためには、次の 2 つのものが必要です。すなわち、事前に構成されたレジストリー、およびそのレジストリーを活動化する方法をキュー・マネージャーに通知する活動化パラメーターのセットです。キュー・マネージャーが活動化されると、活動化パラメーターがそれに渡されます。これらのパラメーターは、別の MQeFields オブジェクトの内部に組み込まれた MQeFields オブジェクトから成っています。

使用される組み込み MQeFields オブジェクトの名前は、MQeQueueManager クラスで定義されます。

**キュー・マネージャー・セットアップ・データ**

MQeQueueManager.QueueManager

活動化されるキュー・マネージャーの名前。

**レジストリーの場所**

MQeQueueManager.Registry.

キュー・マネージャーの事前定義されたレジストリーの場所。

**MQSeries Everyplace 別名**
MQeQueueManagerUtils.Section_Aliases

キュー・マネージャーのセットアップ・フィールドは、活動化されるキュー・マネージャーの名前から成っています。レジストリーの場所フィールドは、キュー・マネージャーの事前定義されたレジストリーの場所を、キュー・マネージャーに通知します。

レジストリーには、キュー・マネージャーが所有するキューの定義、その他の既知のキュー・マネージャーの定義、および構成可能なキュー・マネージャーのセットアップ・データの一部が含まれます。このセットアップ・データは次のものです。

**Queue manager description**
このキュー・マネージャーについての記述を含むストリング。

**Queue manager rules**
キュー・マネージャーのルールとして使用される、クラスの名前を含むストリング (80ページの『キュー・マネージャー・ルール』を参照)。

**Default queue store**
デフォルトのキュー・ストアの場所であるパス名。これは、まだキュー・ストア・フィールドを含んでいないキューが、キュー・マネージャーに追加される場合にのみ使用されます。この場合、キューはデフォルトのキュー・ストアを割り当てられます。キューの名前は、デフォルトのストリングに追加されて、固有のストア・パス名をキューに与えます。

**注:** キュー・ストアとは、メッセージのキュー・ストアの場所です。

**Channel attribute rules**
チャネル属性ルールとして使用されるクラスの名前を含むストリング。これらのルールは、非空属性を持つリモート・キューを処理する際に求められる動作を記述します。

**Channel Timeout -**
チャネル・タイムアウトとなる長精度の値 (ミリ秒単位)。 2 つのキュー・マネージャー間のチャネルが、この期間よりも長く使用されていない場合には、チャネルがクローズされます。

これらの値はすべて MQSeries Everyplace 管理を使って更新したり、またキュー・マネージャーの作成時に構成したりすることができます。

MQSeries Everyplace 別名は、標準およびカスタム MQSeries Everyplace クラスの別名です。クラスを参照するときには、完全なクラス名の代わりに、別名を使用することができます。次は、典型的な別名リストです (ini ファイル・フォーマット)。

```
[Alias]
*
* Channel manager & Commands class(s)
*
(ascii)Echo=Test.Echo
*
* Event log class
*
(ascii)EventLog=examples.eventlog.LogToDiskFile
*
* Network adapter class
*
```

```
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*  queue manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
* Trace handler (if any)
*
(ascii)Trace=examples.awt.AwtMQeTrace
*
* Disk fields interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
* Admin console (if any)
*
(ascii)Admin=examples.administration.console.Admin
*
* Base Server class
*
(ascii)Server=examples.queuemanager.MQeServer
*
* Default Channel class
*
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
*
* Default Transporter class
*
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
```

別名は、等号の左側にあり、完全なクラス名は右側にあります。たとえば、この例では、examples.awt.AwtMQeTrace の代わりに、'Trace' という名前を使用することができます。別名の前の "(ascii)" は、エントリーのタイプ (この場合は、ASCII ストリング) を示します。

別名リストには、ソリューションの所有するクラスを組み込むことができます。

別名リストは、キュー・マネージャー自体によっては処理されません。別名のいくつかはキュー・マネージャーが正常に活動化するために必要とされるため、この別名リストはキュー・マネージャーの活動化の前に処理されていなければなりません。たとえば、キュー・ストア・アダプターでは、キューがメッセージを保持するためのストレージ域を持つことができるように定義する必要があります (68ページの『キュー』を参照)。 MsgLog は、デフォルトのキュー・ストア・アダプターであり、これが存在しない場合には、'MsgLog not found' 例外が出されます。

MQSeries Everyplace は、事前定義された構成でキュー・マネージャーを開始する 2 つのクラスを提供します。 (これらのクラスは examples ディレクトリーにあります。)

- MQeClient クラスはキュー・マネージャーを、クライアントとして開始します。
- MQeServer クラスはキュー・マネージャーを、MQSeries Everyplace サーバーの一部として開始します。

  **注:** MQeServer は他の MQSeries Everyplace 構成からの複数の同時要求を受け入れ、これらの要求は、そのサーバーに関連したキュー・マネージャーによって保守されます。

必要な処理はすべて、キュー・マネージャーが開始する前に、これらのクラスによって処理されます。

これらのクラスのどちらも使用せずに、別名リストを処理してキュー・マネージャーを活動化することができます。別名リストは、MQe.alias メソッドを使って処理されます。下の例では、別名 'Trace' が 'examples.awt.AwtMQeTrace' に設定されます。

```
alias( "Trace", "examples.awt.AwtMQeTrace" );
```

MQeClient と MQeServer の両方が、キュー・マネージャー・パラメーターを含む .ini ファイルを受け入れます。 .ini ファイルのエントリーは、必要な組み込み MQeFields オブジェクトに変換されます。この処理は、examples.queuemanager.MQeQueueManagerUtils クラスによって行われ、そのクラスは MQe.alias メソッドを使って別名リストを処理します。

次のコード断片は、上記の手順を示しています。

```
public static void processAlias( MQeFields sections ) throws Exception
{
  if ( sections.contains( Section_Alias ) )   /* section present ?          */
  {                                           /* ... yes                    */
    MQeFields section = sections.getFields( Section_Alias );
    Enumeration keys  = section.fields( );    /* get all the keywords       */
    while ( keys.hasMoreElements() )          /* as long as there are       */
                                              /* keywords                   */
    {                                         /*                            */
      /* get the Keyword    */
      String key = (String) keys.nextElement();
      /* add alias          */
      MQe.alias( key, section.getAscii( key ).trim( ) );
    }                                         /*                            */
  }                                           /*                            */
}
```

このメソッドへの入力データである MQeFields オブジェクト sections は、MQeFields フォームの .ini ファイルです。 ini ファイルは、MQeQueueManagerUtils の loadConfigFile() メソッドで、MQeFields オブジェクト・フォームに変換されます (これは MQeFields.restoreFromString() メソッドを使用します)。

まず最初に、sections に別名リストが含まれているかを調べるテストが行われます。別名リスト ini ファイル のセクション名は、定数 Section_Alias で定義されます。別名リストが使用可能な場合は、getFields() が sections に対して実行され、別名リストを戻します (これも MQeFields オブジェクトです)。その後、別名リストの内容が列挙され、コードが列挙全体をループして、それぞれの別名ごとに別名コマンドを呼び出します。

# キュー・マネージャーの使用

## MQSeries Everyplace アプリケーションおよび Java 仮想計算機

MQSeries Everyplace キュー・マネージャーの Java 版は、Java 仮想計算機 (JVM) のインスタンスの内部で実行します。現在、MQSeries Everyplace は、1 つの JVM につき 1 つのキュー・マネージャーだけしか起動させません。しかし、同じデバイス上で JVM の複数インスタンス (Java コマンドが呼び出されるたびに、新規の Java 仮想計算機が作成される)、つまり複数の MQSeries Everyplace キュー・マネー

ジャーを起動することができます。これらのキュー・マネージャーにはそれぞれ固有の名前が必要であり、固有の名前がなければ、予期しない動作が生じることがあります。

Java MQSeries Everyplace アプリケーションは、それらが使用しているキュー・マネージャーと同じ JVM の内部で実行しなければなりません。これを行う洗練された方法は、アプリケーション・ランチャーを使用することです。これは、別個のスレッド上にあるキュー・マネージャーと多数の MQSeries Everyplace アプリケーションを開始するクラスです。そのようなクラスの例が、次のコード断片に示されています。

```
/* extends from   MQe base class */
public class appLauncher extends   MQe implements Runnable
{
  Thread[] threads    = null; /* thread references */
  String[] appList    = null; /* list of   MQSeries Everyplace apps */
  int      appCount   = 0;
  String lock = new String();
  MQeQueueManager qmgr = null; /* reference to QMgr */
  public static void main( String args[] )
  {

    try
    {
      (new appLauncher()).startApplications();
    }
    catch ( Exception e )
    {
      System.err.println( "Exception on starting applications" );
      e.printStackTrace( System.err );
    }
  }
  public void startApplications( String args[] ) throws Exception
  {
    boolean active = false; /* any active threads? */
    /* create an array of the thread references of the applications  */
    /* being launched */
    threads = new Thread[ args.length ];
    appList = args; /* keep the list of the applications to be launched */
    /* loop through the list of apps being launched & start a new  */
    /* thread for each one */
    for ( int i = 0; i < appList.length; i++ )
    {
      Thread th = new Thread( this ); /* create a new thread */
      threads[i] = th; /* keep reference */
      th.start(); /* start new thread */
      /* loop until queue manager is active then start rest of apps */
      if ( i == 0 )
        while( qmgr == null );
    }
    /* keep appLauncher thread alive until all other apps have finished */
    while( active )
    {
      active = false;
      /* loop through thread references, starting at element 1 */
      /* remember first element in appList is QMgr ini file path name */
      for( int j=1; j < appList.length; j++ )
        if ( threads[j] != null )
          active = true; /* thread still active */
    }
    if ( qmgr != null )
      qmgr.close(); /* close queue manager */
  }
  /* this method called for each application being launched, plus the  */
```

```
  /*  queue manager */
  public void run()
  {
    int currentApp; /* which element in threads table */
    synchronized( lock )
    {
      currentApp = appCount;
      appCount++; /* update count */
    }
    try
    {
      /* first element is QMgr ini file path name */
      if ( currentApp == 0 ) /* start  queue manager */
      {
        MQeClient client = new MQeClient( appList[0] );
        qmgr = client.queueManager; /* QMgr now active */
      }
      else  /* load application */
                                        /* (this invokes default constructor) */
        loader.loadObject( appList[currentApp] );
    }
    catch ( Exception e )
    {
      e.printStackTrace( System.err );
    }
    finally
    { /* get thread reference for this app */
      Thread th = threads[currentApp];
      threads[currentApp] = null; /* nullify reference */
      th.stop(); /* stop thread */
    }
  }
}
```

このクラスに提供される引き数は、キュー・マネージャーの ini ファイルのパス名で、その後に、起動される MQSeries Everyplace アプリケーションのリストが続きます。すべてのアプリケーションは、デフォルトのコンストラクターを使用して呼び出されます。

アプリケーション・ランチャーは、次のコマンドで起動されます。

```
java appLauncher
<ini ファイル・パス名><アプリケーション・クラス名><アプリケーション・クラス名>...
```

たとえば、次のようにします。

```
java appLauncher
           e:¥¥MQe¥¥TestQMgr¥¥TestQMgr.ini examples.queuemanager.TestMQeApp
```

すべてのアプリケーションは、MQeQueueManager.getReference() を使用して、すでに JVM 内部で実行しているキュー・マネージャーへのオブジェクト参照を取得する必要があります。

## RunList を使ってアプリケーションを立ち上げる

MQSeries Everyplace アプリケーションを立ち上げる代替方法は、RunList メカニズムを使用することです。キュー・マネージャーの活動化パラメーターの一部として、MQSeries Everyplace アプリケーションの 2 つのリストを提供することができます。最初のリストには、キュー・マネージャーが活動化された後で立ち上げられ

るアプリケーションが含まれています。 2 番目のリストには、キュー・マネージャーがクローズ要求を受け取ると立ち上げられるアプリケーションが含まれています。

キュー・マネージャー・パラメーターに含まれるアプリケーションが MQeRunListInterface をインプリメントする場合、キュー・マネージャーは インターフェースで定義された activate() メソッドを呼び出してアプリケーションを活動化します。そして、キュー・マネージャー・パラメーターに含まれるアプリケーションのセットアップ情報を渡します。

アプリケーションは MQeRunListInterface をインプリメントしなくても構いませんが、インプリメントしなければ、アプリケーションはただ起動されるだけで、セットアップ情報は渡されません。

ini ファイル内の section [AppRunList] には、キュー・マネージャーの活動化時に立ち上げられるアプリケーションの名前が含まれています。アプリケーションの記号名は等号の左側にあり、アプリケーションの完全なクラス名はその右側にあります。 キュー・マネージャーの *ini ファイル例* に示されているとおりです。

アプリケーションのセットアップ・データは、セクション内で、アプリケーションの記号名を使用して提供することができます。

**キュー・マネージャーの ini ファイルの例**

```
* Sample  queue manager ini file
*  queue manager setup info
[QueueManager]
* Name for this  queue manager
(ascii)Name=ServerQMgr8082
* Registry setup info
[Registry]
* QueueManager Registry type (ascii)LocalRegType=com.ibm.mqe.registry.MQePrivateSession
* Location of the registry
(ascii)DirName=d:¥development¥Rename¥Classes¥ServerQMgr8082¥Registry
* Registry access PIN
(ascii)PIN=12345678
* List of applications to launched at  queue manager activation-time
[AppRunList]
(ascii)App1=examples.queuemanager.TestMQeApp
(ascii)App2=examples.administration.AdminApp
* Setup info for App1 - the data in this section is passed to the application
[App1]
(ascii)DefaultMsgPriority = 7
(long)Timeout = 30000
* Setup info for App2 - the data in this section is passed to the application
[App1]
(ascii)DefaultQueueName=AdminReplyQueue
```

キュー・マネージャーの活動化時に起動されるアプリケーションは、キュー・マネージャーが活動化を継続できるように、できるだけ即座にこのメソッドから戻る必要があります。アプリケーションが長時間実行するタスクである場合には、呼び出されたものとは別のスレッド上でそれ自体を初期化する必要があります。アプリケーションは、このスレッドを管理する責任があります。

キュー・マネージャー close 上で呼び出されるアプリケーションは、メソッドから戻らないことによって、キュー・マネージャーがシャットダウンしないようにすることができます。

**キュー・マネージャーの活動化時に立ち上げられるアプリケーションの例**

```
public class ExampleApp extends   MQe implements MQeRunListInterface,
                                                  Runnable,
                                                  MQeMessageListenerInterface
{
  Thread th = null;
  MQeQueueManager qmgr = null;
  ...
  /*Called by the  queue manager to activate the application          */
  public Object activate( Object owner, Hashtable loadTable,
                          MQeFields setupData )
  {
    qmgr = (MQeQueueManager)owner;   /*QMgr is owner of the application*/
    processSetupData( setupData );      /*Process the setup information*/
    th = new Thread( this );            /*Create a new thread to listen*/
    th.start();                             /*for incoming messages*/
    return (null);                     /*return control to the QMgr*/
  }
  public void run()
  {
    try
    {
       /*Create a message listener for incoming messages*/
       qmgr.addMessageListener( this, "MyQueue", null );
       /* Loop indefintely keeping application alive */
       while( true );
    }
    catch ( Exception e )
    {
      e.printStackTrace( System.err );
    }
  }
  ...
}
```

この例では、activate() メソッドを使用してアプリケーションが起動されます。このメソッドは、そのセットアップ・データを処理し、別個のスレッド上でメッセージ・リスナーを作成します。アプリケーションは、キュー・マネージャーがその活動化プロセスを継続できるように、できるだけ早くキュー・マネージャーに制御を戻します。アプリケーションが作成したスレッドはアクティブのままです。

### キュー・マネージャーがクローズ要求を受け取るときに立ち上げられるアプリケーションの例

```
public class ExampleCloseApp extends   MQe implements MQeRunListInterface
{
  MQeQueueManager qmgr = null;
  ...
  /* Called by the  queue manager to activate the application  */
  public Object activate( Object owner, Hashtable loadTable,
                          MQeFields setupData )
  {
    qmgr = (MQeQueueManager)owner; /* QMgr is owner of the application */
    performAction(); /* Perform some action */
    /* don't return control to the QMgr until application has finished */
    return (null);
  }
}
```

この例では、キュー・マネージャーが close 要求を受け取るとき、activate() メソッドを使用してアプリケーションが活動化されます。アプリケーションは、その処理を完了するまでキュー・マネージャーに制御を戻してはなりません。なぜなら、キュー・マネージャーが一度制御を取ると、そのクローズ・プロセスを継続するからです。

# メッセージ

MQSeries Everyplace メッセージ・オブジェクトは MQeFields の下位オブジェクトなので、フィールド・パラダイムの能力を継承します。アプリケーションは、データを <名前、データ> の対としてメッセージに入れることができます。 MQSeries Everyplace は、メッセージング・アプリケーションにとって有用ないくつかの定数フィールド名を定義します。これらのフィールドは、次のとおりです。

**固有 ID**
MQe.Msg_OriginQMgr + MQe.Msg_Time

**MQSeries メッセージ ID**
MQe.Msg_ID

**MQSeries 相関 ID**
MQe.Msg_CorrelID

**優先順位**
MQe.Msg_Priority

固有 ID は、メッセージの作成時にメッセージ・オブジェクトによって生成された固有のタイム・スタンプ (JVM あたり 1 つ) と、メッセージが最初に送られたキュー・マネージャーの名前の組み合わせです。固有 ID は、アプリケーションによって変更することができませんが、アプリケーションがメッセージを検索するために使用することはできます。

MQSeries Everyplace ネットワーク内のすべてのキュー・マネージャーが一意的に命名されている限り、固有 ID を使用して、MQSeries Everyplace ネットワーク内のメッセージを一意的に識別することができます。

**注:** MQSeries Everyplace はこの要件を満たしません。キュー・マネージャーに固有の名前を付けるのは、個々のソリューションの責任です。

アプリケーションは、メッセージによって生成された固有 ID に対する制御を持っていません。しかし、アプリケーションは固有 ID にアクセスし、それをメッセージ・フィルター内で使用することができます (下記を参照)。 `getMsgUIDFields()` メソッドは、メッセージの固有 ID にアクセスします。

```
MQeFields msgUID = msgObj.getMsgUIDFields();
```

`getMsgUIDFields()` メソッドによって戻される MQeFields オブジェクトには、次の 2 つのフィールドがあります。

- MQe.Msg_OriginQMgr
- MQe.Msg_Time

これらのフィールドは、次のようにして個々に検索することができます。

```
long timeStamp    = msgUID.getLong( MQe.Msg_Time );
String originQMgr = msgUID.getAscii( MQe.Msg_OriginQMgr );
```

MQSeries メッセージ ID フィールドと相関 ID フィールドにより、アプリケーションの指定する ID 値を提供することができます。これらの 2 つのフィールドはまた、MQSeries ファミリーの残りのメンバーとのインターオペラビリティーを容易にします。

```
MQeMsgObject msgObj = new MQeMsgObject;
msgObj.putArrayOfByte( MQe.Msg_ID, MQe.asciiToByte( "1234" ) );
```

優先順位フィールドには、メッセージ優先順位値が入ります。メッセージ優先順位は、MQSeries ファミリーの他のメンバーの場合と同様の方法で定義されます。優先順位の範囲は 9 (最高) ～ 0 (最低) です。アプリケーションはこのフィールドを使用して、優先順位に応じて適切にメッセージを処理することができます。

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putByte( MQe.Msg_Priority, (byte)8 );
```

アプリケーションは、それぞれ独自のデータ用のフィールドを、メッセージ内に作成することができます。

```
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "PartNo", "Z301" );
msgObj.putAscii( "Colour", "Blue" );
msgObj.putInt( "Size", 350 );
```

代替方法として、MQeMsgObject を拡張して、メッセージを作成するのに助けとなるメソッドをいくつか組み込むことができます。

```
package messages.order;
import com.ibm.mqe.*;
/**
 * This class defines the Order Request format
 */
public class OrderRequestMsg extends MQeMsgObject
{
  public OrderRequestMsg() throws Exception
  {
  }
 /**
  * This method sets the client number
  */
  public void setClientNo(long aClientNo) throws Exception
  {
    putLong("ClientNo", aClientNo);
  }
 /**
  * This method returns the client number
  */
  public long getClientNo() throws Exception
  {
    return getLong("ClientNo");
  }
 /**
  * This method sets the name of the item to be ordered
  */
  public void setItem(String anItem) throws Exception
  {
    putUnicode("Item", anItem);
  }
 /**
  * This method returns the name of the item to be ordered
  */
  public String getItem() throws Exception
  {
    return getUnicode("Item");
  }
 /**
  * This method sets the quantity required
  */
  public void setQuantity(int aQuantity) throws Exception
  {
```

```
    putInt("Quantity", aQuantity);
  }
 /**
  * This method returns the quantity required
  */
  public int getQuantity() throws Exception
  {
    return getInt("Quantity");
  }
 /**
  * This method sets the name of the queue to which to send an order reply
  */
  public void setReplyToQ(String aMyReplyToQ) throws Exception
  {
    putAscii("Msg_ReplyToQ", aMyReplyToQ);
  }
 /**
  * This method returns the name of the queue to which an order reply
  * will be sent
  */
  public String getReplyToQ() throws Exception
  {
    return getAscii("Msg_ReplyToQ");
  }
 /**
  * This method sets the name of the queue manager to which an order
  * reply will be sent
  */
  public void setReplyToQMgr(String aMyReplyToQMgr) throws Exception
  {
    putAscii("Msg_ReplyToQMgr", aMyReplyToQMgr);
  }
 /**
  * This method returns the name of the queue manager to which an order
  * reply will be sent
  */
  public String getReplyToQMgr() throws Exception
  {
    return getAscii("Msg_ReplyToQMgr");
  }
}
```

追加のメソッドは、メッセージ・オブジェクトに出入りするデータの書き込みおよび取得を処理します。アプリケーション・プログラマーは、送信されるデータのタイプも、メッセージ内で使用されるフィールド名も知る必要がありません。

```
OrderRequestMsg orderRequest = new OrderRequestMsg();
orderRequest.setClientNo( 1234 ); /* client ref. number */
orderRequest.setItem( "  MQSeries Everyplace Programmers Guide" ); /* item being ordered */
orderRequest.setQuantity( 12 ); /* quantity */
/* send the order reply to QMgr1.OrderReplyQueue */
orderRequest.setReplyToQMgr( "QMgr1" );
orderRequest.setReplyToQ( "OrderReplyQueue" );
```

## フィルター

フィルターの概念により、MQSeries Everyplace は強力なメッセージ検索を実行することができます。キュー・マネージャーの主な操作のほとんどは、フィルターの使用をサポートします。フィルターは、MQeFields オブジェクトにフィールドを配置することによって作成されます。たとえば、単純な get message 操作では空のフィルターを使用することができます。この操作の結果、キュー上にある使用可能な最初のメッセージが戻されます。

```
qmgr.getMessage( "myQMgr", "myQueue", null, null, 0 );
```

フィルターの使用により、アプリケーションは、フィルターと同じフィールドおよび値を含む、使用可能な最初のメッセージを戻すことができます。たとえば、アプリケーションがメッセージ ID '1234' を含む最初のメッセージを取得したい場合、次のようなフィルターを作成します。

```
MQeFields filter = new MQeFields();
filter.putArrayOfByte( MQe.Msg_MsgID, MQe.AsciiToByte( "1234" ) );
```

次に、フィルターは、get message 操作に渡されます。

```
qmgr.getMessage( "myQMgr", "myQueue", filter, null, 0 );
```

この操作の結果、メッセージ ID '1234' を含むキュー上にある、使用可能な最初のメッセージが戻されます。

## メッセージ索引フィールド

MQSeries Everyplace はそのメッセージをキューごとに提供される永続ストアに保管します (アプリケーションがメモリー・キュー・ストア・アダプターを使用していない場合)。メモリー・サイズの制約のため、メモリー内にメッセージ全体を保持することはしません。しかし、各メッセージ内の特定のフィールドをメッセージ索引に保持して、より高速なメッセージ検索を行えるようにします。キャッシュに入れられるフィールドは、次のものです。

**固有 ID**
: MQe.Msg_OriginQMgr + MQe.Msg_Time

**MQSeries メッセージ ID**
: MQe.Msg_ID

**MQSeries 相関 ID**
: MQe.Msg_CorrelID

**優先順位**
: MQe.Msg_Priority

メッセージング機能に提供されるフィルターでこれらのフィールドを使用すると、MQSeries Everyplace はすべてのメッセージをメモリーにロードする必要がないため、検索が一層効果的に行われます。

## メッセージの有効期限

キューには有効期限を定義することができます。メッセージの保管期間がこの期限を超えた場合には、有効期限切れのマークが付けられます。有効期限切れというマークが付いたときにメッセージがどのように処理されるかは、キュー・ルールによって決まります。

メッセージには、それ自体の有効期限が指定されることもあります。これは、メッセージに MQe.Msg_ExpireTime フィールドを追加することによって定義されます。有効期限には、相対的なもの (例、メッセージの作成後 2 日で満了) もあれば、絶対的なもの (例、2000 年 11 月 25 日 08:00 時) もあります。

下記の例では、メッセージは作成後 60 秒で有効期限が切れます。 (60000 ミリ秒 == 60 秒。)

```
/* create a new message */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
```

```
/* expiry time of sixty seconds after message was created */
msgObj.putInt( MQe.Msg_ExpireTime, 60000 );
/* put message onto queue */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

下記の例では、メッセージは 2001 年 5 月 15 日 15:25 に有効期限が切れます。

```
/* create a new message */
MQeMsgObject msgObj = new MQeMsgObject();
msgObj.putAscii( "MsgData", getMsgData() );
/* create a Date object for 15th May 2001, 15:25 hours */
Calendar calendar = Calendar.getInstance();
calendar.set( 2001, 04, 15, 15, 25 );
Date expiryTime = calendar.getTime();
/* add expiry time to message */
msgObj.putLong( MQe.Msg_ExpireTime, expiryTime.getTime() );
/* put message onto queue */
qmgr.putMessage( null, "MyQueue", msgObj, null, 0 );
```

## キュー

キュー・マネージャーは、メッセージを保持するキューを管理します。それぞれのキュー・マネージャーには、それが管理し、所有するキューを持つ機能があります。これらのキューはローカル・キューと呼ばれます。

MQSeries Everyplace により、アプリケーションは別のキュー・マネージャーに属するキュー上のメッセージにアクセスすることができます。これらのキューはリモート・キューと呼ばれます。ローカル・キューの場合と同じセットの操作 (ただし、メッセージ・リスナーの定義は除く) を、リモート・キューでも行うことができます (下記を参照)。

キュー・エンティティーは、アプリケーションには直接見えません。キューとの対話はすべてキュー・マネージャーを介して行われます。

キュー上にあるメッセージは、キューの永続ストアに書き込まれます。キューで使用される支援記憶装置を使ってシステム障害時にメッセージを回復することができる場合、 MQSeries Everyplace はメッセージの送達を確実に実行することができます。

キューは、キュー・ストア・アダプターを介して永続ストアにアクセスします。アダプターは、MQSeries Everyplace とハードウェア・デバイス (ディスク、ネットワーク、またはデータベースなどのソフトウェア) との間のインターフェースです。アダプターは、交換可能なコンポーネントとして設計されているので、デバイスとの対話に使用されるプロトコルを簡単に交換することができます。 (227ページの『第9章 MQSeries Everyplace アダプター』を参照してください。)

キューで使用される支援記憶装置は、MQSeries Everyplace 管理メッセージを使って変更することができます。ただし、キューがアクティブになっているか、またはそれにメッセージが含まれている場合には、支援記憶装置の変更は許可されていません。

### キューの順序付け

キュー上のメッセージの順序は、基本的に優先順位に依存しています。メッセージ優先順位の範囲は 9 (最高) 〜 0 (最低) です。優先順位値が同じメッセージはキュ

ーに到達した時刻によって配列されます。したがって、最も長時間キュー上にあるメッセージは、その優先順位グループの先頭に配置されます。

## メッセージの取得

キューが空のとき、get message コマンドが発行された場合、キューは Except_Q_NoMatchingMsg 例外を出します。これにより、キュー上の使用可能なすべてのメッセージを読み取りをインプリメントすることができます。

```
try
{
  while( true )
  { /* keep getting messages until an exception is thrown */
    MQeMsgObject msg = qmgr.getMessage( "myQMgr", "myQueue", null, null, 0 );
    processMessage( msg );
  }
}
catch ( Exception e )
{
  if ( e.code() != MQe.Except_Q_NoMatchingMsg )
    throw e;
}
```

getMessage() 呼び出しを try..catch ブロック内部に入れることにより、発生する例外コードをテストすることができます。これは、MQeException クラスの code() メソッドを使用して行われます。 code() メソッドの結果は、MQSeries Everyplace クラスによって発行される例外定数のリストと比較することができます。例外のタイプが Except_Q_NoMatchingMsg でない場合、例外がもう一度出されます。

**注:** この機能は、MQ-ブリッジ・キュー上ではサポートされません。

## ブラウズおよびロック

メッセージのグループのブラウズおよびロックは有用な技法です。というのは、あるアプリケーションは、メッセージがロックされている間、他のアプリケーションによってメッセージが処理されないようにすることができるからです。アプリケーションによってロックが解除されるまで、メッセージはロックされたままの状態になります。他のアプリケーションがメッセージのロックを解除することはできません。

```
MQeEnumeration msgEnum = qmgr.browseMessagesAndLock( null, "MyQueue", null,
                                                     null, 0, false );
```

このコマンドは、ローカル・キュー・マネージャー上に存在する ″MyQueue″ キュー上にあるすべてのメッセージをロックします (ヌルは、ローカル・キュー・マネージャーの別名です)。現在、これらのメッセージには、それらをロックしたアプリケーションだけがアクセスすることができます。 (ブラウズおよびロック操作後にキューに届いたメッセージは、ロックされないことにご注意ください。)

MQeEnumeration オブジェクトには、ブラウズに提供されたフィルターに一致するすべてのメッセージが入ります。 MQeEnumeration を、標準 Java Enumeration と同じ仕方で使用することができるので、ブラウズされたすべてのメッセージを次のように列挙することができます。

```
while( msgEnum.hasMoreElements() )
{
  MQeMsgObject msg = (MQeMsgObject)msgEnum.nextElement();
  System.out.println( "Message from  queue manager: " +
                      msg.getAscii( MQe.Msg_OriginQMgr ) );
}
```

アプリケーションは、メッセージに対して get または delete のいずれかの操作を実行して、それらをキューから除去することができます。これを行うには、アプリケーションは列挙したメッセージとともに戻されるロック ID を提供する必要があります。ロック ID を指定することにより、最初にそれらのロックを解除しなくても、アプリケーションはロックされたメッセージを処理することができます。次のコードは、列挙で戻されるすべてのメッセージに対して、delete を実行します。メッセージの固有 ID は、ロック ID とともに、delete 操作でフィルターとして使用されます。

```
while( msgEnum.hasMoreElements() )
{
  MQeMsgObject msg = (MQeMsgObject)msgEnum.getNextMessage( null,0 );

  processMessage( msg );

  MQeFields filter = msg.getMsgUIDFields();
  filter.putLong( MQe.Msg_LockID, msgEnum.getLockId() );

  qmgr.deleteMessage( null, "MyQueue", filter );
}
```

標準の java.util.Enumeration nextElement() メソッドを使用する代わりに、MQeEnumeration は getNextMessage() メソッドを提供します。このメソッドは、browseMessages() メソッドの justUID パラメーターに応じた仕方で作動します。このパラメーターは、ブラウズ操作がメッセージ内の一致するすべてのフィールドを戻すか、あるいは固有 ID フィールドだけを戻すかを指定します。

justUID パラメーターが 'false' に設定されている場合、ブラウズによって戻される MQeEnumeration には、メッセージ内の一致するすべてのフィールドが含まれています。この場合、getNextMessage() メソッドは nextElement() のように作動します。

justUID パラメーターが 'true' に設定されている場合、ブラウズによって戻される MQeEnumeration には、メッセージ内の一致する固有 ID フィールドだけが含まれます (MQe.Msg_OriginQMgr & MQe.Msg_TimeStamp)。この場合、getNextMessage() メソッドは、nextElement() と異なる仕方で作動します。ブラウズ・コマンドによって作成された MQeEnumeration に対して getNextMessage() を使用すると、justUID が 'true' に設定されていれば、列挙されているメッセージに対してメッセージの取得が実際に実行されます。これでキューからメッセージが除去されます。

メッセージを取得する際には確実なメッセージ送達を使用することができます。ゼロ以外の確認 ID を指定するとは、取得の確認が必要であるという意味です (確実なメッセージ送達の詳細については、73ページの『同期の確実なメッセージ送達』を参照してください)。

メッセージをキューから除去する代わりに、メッセージのロックを解除することもできます。これで、メッセージが再びすべての MQSeries Everyplace アプリケーションに見えるようになります。このことは、`unlockMessage()` メソッドを使用して行われます。

**注:** この機能は、MQ-ブリッジ・キュー上ではサポートされません。

## 同期および非同期のメッセージング

MQSeries Everyplace により、アプリケーションはメッセージを柔軟に処理することができます。アプリケーションは、そのメッセージがいつどのようにして送信されるかを知る必要はありません。しかし望むならば、*同期* メッセージングを使用して、このプロセスを制御することができます。同期メッセージングは、put message コマンドが発行されるとすぐに、メッセージが送信されるということです。このタイプのメッセージングは、ローカルと宛先の両方のキュー・マネージャーが同時にオンラインであるときにのみ行われます。キュー・マネージャーがネットワークに接続されていない場合には作動しません。これは、インスタント接続のパフォーマンス上の利点を提供し、メッセージがその宛先に到達したことを認識します。

*非同期* メッセージングでは、デバイスがネットワークに接続されているかどうかにかかわりなく、アプリケーションはメッセージの処理を継続することができます。アプリケーションはメッセージをリモート・キューに書き込み、そのメッセージはキュー・マネージャーによって保管され、後で接続が確立されたときにリモート・キュー・マネージャーに送信されます。アプリケーションは、これがいつ生じるかに気付く必要はありません。非同期メッセージングの典型的な例が IBM 技術員または販売員用のアプリケーションです。それらのアプリケーションは注文や在庫を送信することができ、デバイスが再びネットワークに物理的に接続されるまでキューに置かれます。この時点で、注文を送信することができます。非同期伝送が行われる場合、キュー・マネージャーは起動される必要があります。このことは、アプリケーションがキュー・マネージャーの `triggerTransmission()` メソッドを呼び出すアプリケーションか、またはキュー・マネージャーの伝送ルールを使用するアプリケーションのいずれかによって行われます (82ページの『伝送ルール』を参照)。

同期メッセージングまたは非同期メッセージングを使用するかどうかは、リモート・キューがどのように定義されているかに依存しています。メッセージをリモート・キューに送信するキュー・マネージャーは、そのキューの定義を保持します。この定義は、リモート・キュー定義と呼ばれます。リモート・キュー定義は、同期または非同期のいずれかとして定義することができます。メッセージがリモート・キューに書き込まれると、ローカル・キュー・マネージャーはリモート・キュー定義を使用して、メッセージの送信方法を判別します。

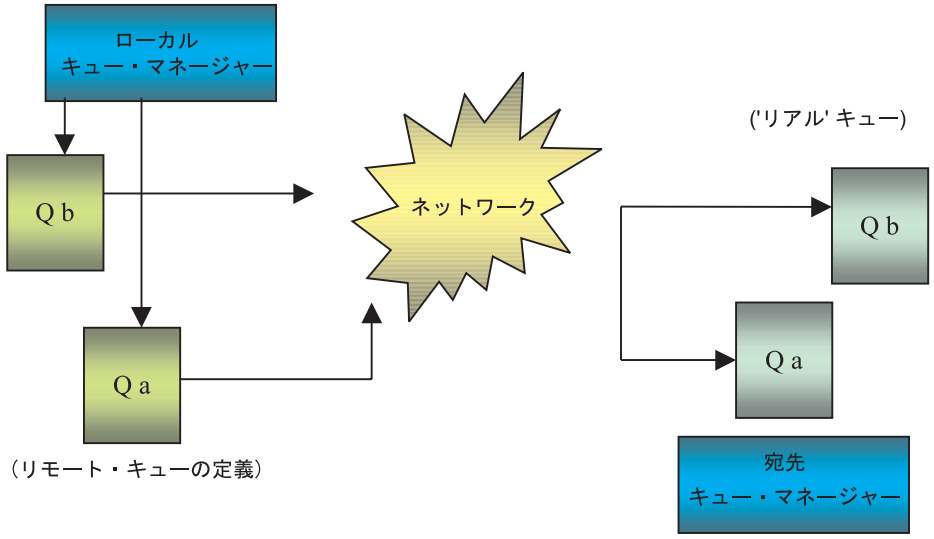


*図 8. MQSeries Everyplace メッセージ・フロー*

メッセージはローカル・キュー・マネージャーからリモート・キュー・マネージャーに、リモート・キュー上で定義された認証機能、暗号機能、および圧縮機能を使用して送信されます。 2 つのキュー・マネージャーの間でメッセージ・チャネルを作成できるようにするには、ローカル・キュー・マネージャーは、伝送に先立って必要な属性を検出できなければなりません。ローカル・キュー・マネージャーは、この情報をそのリモート・キュー定義の一部として保管します。

2 つの伝送スタイルは、これを別々の方法で処理します。

同期スタイルの伝送では、宛先キュー・マネージャーがオンラインで使用可能なので、ローカル・キュー・マネージャーは、宛先キューの属性を要求することができます。また、同期メッセージングは、ローカル・キュー・マネージャーがリモート・キュー定義を保持していないキューに、メッセージを送信することもできます。メッセージを送信する前に、ローカル・キュー・マネージャーはリモート・キュー・マネージャーからリモート・キューの特性を要求します。すると、メッセージを送信するときに、チャネル上で正しい属性を使用することができます。このプロセスは、キュー・ディスカバリーと呼ばれます。ローカル・キュー・マネージャーが検出した情報は、新規のリモート・キュー定義に保管されます。

しかし、非同期スタイルの伝送は、宛先キュー・マネージャーから情報を要求することができません。したがって、このスタイルの伝送では、非同期伝送が可能になる前に、リモート・キュー定義が存在していなければなりません。リモート・キュー定義は、MQSeries Everyplace 管理メッセージを使用して定義することができます。

非同期伝送はまた、確実なメッセージ送達の概念を紹介します。メッセージを非同期に送達すると、MQSeries Everyplace はメッセージを一回限り、その宛先キューに確実に送達します。しかし、この保証は、リモート・キューとリモート・キュー・マネージャーの定義が、リモート・キューとリモート・キュー・マネージャーの現行の特性と一致する場合にのみ有効です。

同期メッセージングと非同期メッセージングを組み合わせることによって、MQSeries Everyplace は信頼できない通信リンクに対処することができます。リンクが切断されたためにメッセージをすぐに送達できない場合には、そのメッセージはキューに入れられ、後ほど送達されます。

この例を以下に示します。 2 つのキューを定義することによって、アプリケーションは同期伝送を行えない状態に対処することができます。

```
try
{
  qmgr.putMessage( "RemoteQMgr", "TransactionQueue", msgObj, null, 0 );
}
catch ( Exception e )
/* reset message UID */
msgObj.resetMsgUIDFields();
{ /* if connection cannot be made, put message on asynchronous queue */
  if ( e.getMessage().equals( "Connection Refused" )
    qmgr.putMessage( "RemoteQMgr", "AsynchTransactionQueue",
                     msgObj, null, 0 );
}
```

## 同期の確実なメッセージ送達

***メッセージの書き込み:*** 同期メッセージ伝送を使用して確実なメッセージ送達を実行することができますが、エラー処理については、アプリケーションが実行する責任があります。

メッセージの確実ではない送達は、単一のネットワーク・フローで生じます。メッセージを送信するキュー・マネージャーは、宛先キュー・マネージャーへのチャネルを作成し、そのチャネルに宛先キューを指すトランスポーターを接続します。(適切なチャネルおよびトランスポーターは、以前の操作から存在していることがあります。その場合には、代わりに既存のものが使用されます。)

送信されるメッセージはダンプされてバイト・ストリームを作成します。そして、これが伝送用のチャネルに送られます。プログラム制御が一度チャネルから戻されると、送信側キュー・マネージャーは、メッセージが正常に宛先キュー・マネージャーに送られ、その宛先がメッセージのログをキュー上に記録したこと、およびメッセージが MQSeries Everyplace アプリケーションに見えるようになっていることを認識します。

しかし、送信側が宛先からチャネルを介して例外を受け取る場合、問題が生じることがあります。送信側は、例外の発生が、メッセージがログに記録されて見えるようになる前なのか、それともその後なのかを知る方法がありません。例外が発生したのがメッセージが見えるようになる前であった場合は、送信側が安全にメッセージを再送することができますが、それがメッセージが見えるようになった後の場合には、メッセージを重複してシステムに送信する恐れがあります。なぜなら、MQSeries Everyplace アプリケーションは、送信側が再送する前に、メッセージを処理してしまった可能性があるからです。

この問題の解決策には、追加の確認フローを伝送することが関係しています。送信側が宛先からこのフローが成功したという応答を受け取るならば、メッセージが一回だけ送達されたことが分かります。

putMessage メソッドの confirmId パラメーターは、確認フローを送信するかどうかを示します。ゼロの値は、メッセージ伝送が 1 つのフローで生じることを意味し、ゼロ以外の値は、確認フローが予期されていることを意味します。宛先キュー・マネージャーは、通常どおりメッセージを宛先キューに記録しますが、確認フローを受け取るまでは、メッセージはロックされ、MQSeries Everyplace アプリケーションには見えません。

MQSeries Everyplace アプリケーションは、confirmPutMessage メソッドを使用して、 put message confirmation を発行することができます。宛先キュー・マネージャーがこのコマンドによって生成されるフローを受け取ると、問題のメッセージのロックを解除し、それを MQSeries Everyplace アプリケーションに見えるようにします。ただし、一度に 1 つのメッセージだけしか確認することができず、メッセージのバッチを確認することはできません。

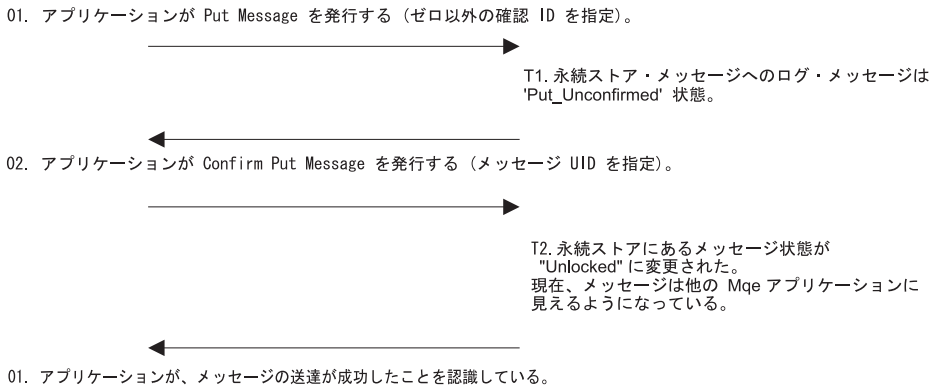


*図 9. 同期メッセージの確実な書き込み*

confirmPutMessage() メソッドでは、前の put message コマンドで使用された Confirm ID ではなく、メッセージの UID を指定する必要があります。 (Confirm ID も、伝送失敗の後ロックされたままになっているメッセージを復元するために使用されます。これは、77ページで詳しく説明されています。)

以下に、必要なコードの骨組みを示します。

```
long confirmId = MQe.uniqueValue();

try
{
  qmgr.putMessage( "RemoteQMgr", "RemoteQueue", msg, null, confirmId );
}
catch( Exception e )
{
  /* handle any exceptions */
}

try
{
  qmgr.confirmPutMessage( "RemoteQMgr", "RemoteQueue",
                          msg.getMsgUIDFields() );
}
catch ( Exception e )
{
  /* handle any exceptions */
}
```

74ページの図9 のステップ 1 で障害が発生した場合には、メッセージを再送してはなりません。この場合、重複したメッセージが MQSeries Everyplace ネットワークに送信される恐れはありません。なぜなら、宛先キュー・マネージャーのメッセージは、確認フローが処理されるまでは、アプリケーションには見えないからです。

MQSeries Everyplace アプリケーションがメッセージを再送する場合、それを行うことを宛先キュー・マネージャーに通知する必要もあります。宛先キュー・マネージャーは、すでに持っているメッセージの重複コピーを削除します。アプリケーションはこれを行うために MQe.Msg_Resend フィールドを設定します。

74ページの図9 のステップ 2 で障害が発生した場合、アプリケーションは確認フローを再送する必要があります。このとき、宛先キュー・マネージャーは、すでに確認したメッセージについて受け取った確認フローは無視するため、安心して再送することができます。

以下のコードは、examples.application.example6 からの抜粋です。

```
boolean msgPut     = false; /* put successful? */
boolean msgConfirm = false; /* confirm successful? */
int maxRetry       = 5; /* maximum number of retries */

long confirmId = MQe.uniqueValue();

int retry = 0;
while( !msgPut && retry < maxRetry )
{
  try
  {
    qmgr.putMessage( "RemoteQMgr", "RemoteQueue", msg, null, confirmId );
   msgPut = true;  /* message put successful             */
  }
  catch( Exception e )
  {
    /* handle any exceptions */
    /* set resend flag for retransmission of message */
    msg.putBoolean( MQe.Msg_Resend, true );
    retry ++;
  }
}

if ( !msgPut )  /* was put message successful?   */
    /* Number of retries has exceeded the maximum allowed, so abort the put*/
    /* message attempt                                                     */
  return;

retry = 0;
while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmPutMessage( "RenoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true;   /* message confirm successful        */
  }
  catch ( Exception e )
  {
    /* handle any exceptions */
    /* An Except_NotFound exception means that the message has already   */
    /* been confirmed                                                    */
    if ( e instanceof MQeException &&
         ((MQeException)e).code() == Except_NotFound )
      putConfirmed = true;            /* confirm successful            */
```

```
    /* another type of exception - need to reconfirm message            */
    retry ++;
  }
}
```

### メッセージの取得:

確実なメッセージ送達のシステムは、get message 操作についても同様に作動します。 get message コマンドが、ゼロより大きい confirmId パラメーターを指定して発行されると、確認フローが宛先キュー・マネージャーによって処理されるまで、メッセージはそれが存在するキュー上でロックされたままになります。その後、メッセージはキューから削除されます。

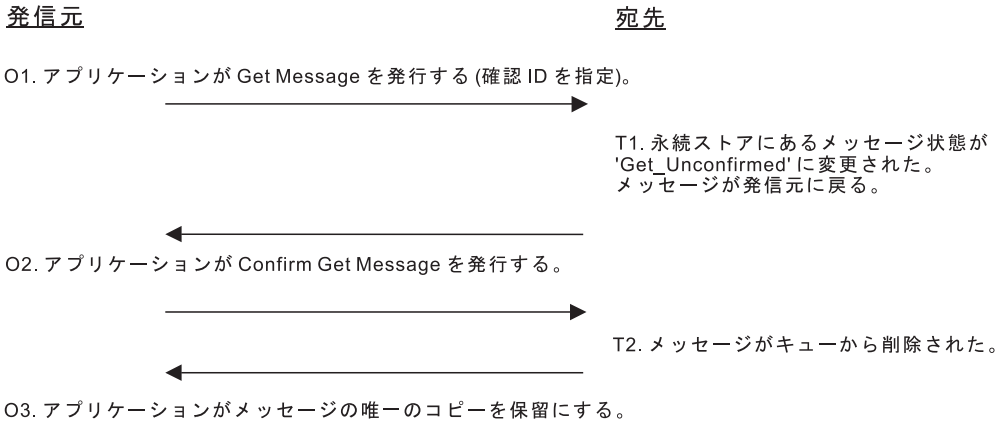


*図 10. 同期メッセージの確実な取得*

以下のコードは、examples.application.example6 からの抜粋です。

```
boolean msgGet     = false; /* get successful? */
boolean msgConfirm = false; /* confirm successful? */
MQeMsgObject msg   = null;
int maxRetry       = 5; /* maximum number of retries */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr", "RemoteQueue", filter, null,
                           confirmId );
    msgGet = true; /* get succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions         */
    /* if the exception is of type Except_Q_NoMatchingMsg, meaning that  */
    /* the message is unavailable then throw the exception               */
    if ( e instanceof MQeException )
      if ( ((MQeException)e).code()  == Except_Q_NoMatchingMsg )
        throw e;
    retry ++; /* increment retry count */
  }
}

if ( !msgGet )                        /* was the get successful?       */
    /* Number of retry attempts has exceeded the maximum allowed, so abort  */
    /* get message operation                                                */
```

```
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true; /* confirm succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions  */
    retry ++; /* increment retry count */
  }
}
```

confirmId パラメーターとして渡される値には、別の使用法もあります。この値は、ロックされて確認を待っている間、メッセージを識別するために追加されます。get 操作中にエラーが発生すると、キュー上でメッセージがロックされたままになることがあります。これが発生するのは、get コマンドの応答としてメッセージがロックされ、アプリケーションがメッセージを受け取る前にエラーが発生するためです。アプリケーションが例外の応答として get を再発行した場合には、メッセージがロックされて MQSeries Everyplace アプリケーションに見えなくなるため、同じメッセージを受け取ることができなくなります。

しかし、get コマンドを発行したアプリケーションは、undo() メソッドを使用して、メッセージを復元することができます。アプリケーションは、get message コマンドに提供したのと同じ confirmId 値を提供する必要があります。 undo コマンドは、提供した confirmId を含むすべてのメッセージを、get コマンドの前の状態に復元します。

```
boolean msgGet     = false; /* get successful? */
boolean msgConfirm = false; /* confirm successful? */
MQeMsgObject msg   = null;
int maxRetry       = 5; /* maximum number of retries */

long confirmId = MQe.uniqueValue();
int retry = 0;
while( !msgGet && retry < maxRetry )
{
  try
  {
    msg = qmgr.getMessage( "RemoteQMgr", "RemoteQueue", filter, null,
                           confirmId );
    msgGet = true; /* get succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions         */
    /* if the exception is of type Except_Q_NoMatchingMsg, meaning that  */
    /* the message is unavailable then throw the exception               */
    if ( e instanceof MQeException )
      if ( ((MQeException)e).code()  == Except_Q_NoMatchingMsg )
        throw e;
    retry ++; /* increment retry count */
    /* As a precaution, undo the message on the queue. This will remove  */
    /* any lock that may have been put on the message prior to the       */
    /* exception occurring                                               */
    myQM.undo( qMgrName, queueName, confirmId );
  }
}
```

```
if ( !msgGet )                          /* was the get successful?       */
    /* Number of retry attempts has exceeded the maximum allowed, so abort  */
    /* get message operation                                                */
  return;

while( !msgConfirm && retry < maxRetry )
{
  try
  {
    qmgr.confirmGetMessage( "RemoteQMgr", "RemoteQueue",
                            msg.getMsgUIDFields() );
    msgConfirm = true; /* confirm succeeded */
  }
  catch ( Exception e )
  {
    /* handle any exceptions  */
    retry ++; /* increment retry count */
  }
}
```

undo コマンドも、putMessage および browseMessagesAndLock コマンドと関係があります。 get メッセージの場合と同様に、undo コマンドは browseMessagesandLock コマンドによってロックされたメッセージを、以前の状態に復元します。

putMessage コマンドが失敗した後で undo コマンドを発行すると、宛先キュー上でロックされて確認を待っているメッセージが削除されます。

undo コマンドは、ローカル・キューとリモート・キューの両方の操作に影響します。

## メッセージ・リスナー

MQSeries Everyplace は、アプリケーションにキュー上で発生したイベントの聴取を行う機能を提供します。通知は標準 Java イベントの形式を取り、聴取しているアプリケーションは、イベントの発生時に呼び出されるメソッドを提供するインターフェースをインプリメントします。アプリケーションは、関係のあるメッセージを識別するためのメッセージ・フィルターを指定することができます。

```
/* Create a filter for "Order" messages of priority 7 */
MQeFields filter = new MQeFields();
filter.putAscii( "MsgType", "Order" );
filter.putByte( MQe.Msg_Priority, (byte)7 );
/* activate a listener on "MyQueue" */
qmgr.addMessageListener( "MyQueue", this, filter );
```

addMessageListener() メソッドに渡されるパラメーターは、次のとおりです。

- メッセージ・イベントの聴取を行うキューの名前。
- MQeMessageListenerInterface をインプリメントするコールバック・オブジェクト。
- メッセージ・フィルターを含むフィールド・オブジェクト。

メッセージがリスナーの付加されたキューに達すると、キュー・マネージャーは、メッセージ・リスナーの作成時に与えられた callback オブジェクトを呼び出します。

以下の例は、アプリケーションがメッセージ・イベントを処理する通常の方法を示しています。

```
public void messageArrived( MQeMsgEvent msgEvent )
{
  String queueName = msgEvent.getQueueName()
  if ( queueName.equals( "MyQueue"))
  {
    /* get message from queue */
    MQeMsgObject msg = qmgr.getMessage( null, queueName,
                                        msgEvent.getMsgUIDFields(), null, 0 );
    /* ...and process it */
    processMessage( msg );
  }
}
```

messageArrived() は、MQeMessageListenerInterface にインプリメントされたメソッドの 1 つです。 msgEvent パラメーターにはメッセージに関する情報が含まれています。これには次のものが含まれます。

- メッセージが達するキューの名前
- メッセージの UID
- メッセージ ID
- 相関 ID
- メッセージ優先順位

メッセージ・フィルターは、ローカル・キュー上でのみ作動します。ポーリングと呼ばれる別の技法により、メッセージがリモート・キューに達するとすぐにそれを取得することができます。

## メッセージ・ポーリング

メッセージ・ポーリングは waitForMessage() メソッドを使用します。このコマンドは、リモート・キューに対して定期的に getMessage() コマンドを実行します。指定されたフィルターに一致するメッセージが使用可能になるとすぐに、呼び出し側アプリケーションにそのメッセージが戻されます。

次は、メッセージ呼び出し待機の典型的な例です。

```
qmgr.waitForMessage( "RemoteQMgr", "RemoteQueue", filter, null, 0, 60000 );
```

waitForMessage() メソッドは、最後のパラメーターで指定された時間だけリモート・キューをポーリングします。この時間はミリ秒単位で指定されます。それで上記の例では、ポーリングは 60 秒間続きます。コマンドが実行されているスレッドは、指定された時間より前にメッセージが戻らない限り、その時間中はブロックされます。

メッセージ・ポーリングは、ローカル・キューとリモート・キューの両方の操作に影響します。

**注:** この技法を使用すると、多重要求がネットワーク上で送信されることになります。

### メッセージング操作

| 操作 | ローカル・キュー | リモート・キュー | |
|---|---|---|---|
| | | 同期 | 非同期 |
| ブラウズ (& ロック) | はい | はい | |
| 削除 | はい | はい | |
| 取得 | はい | はい | |
| 聴取 | はい | | |
| 書き込み | はい | はい | はい |
| 待機 | はい | はい | |

## ルール

MQSeries Everyplace は、その主なコンポーネントの動作を制御するために、ルールの概念を使用します。ルールを使用すると、アプリケーションは MQSeries Everyplace の内部動作をある程度制御することができます。ルールは、MQSeries Everyplace コンポーネントが初期化されたときに、それらによってロードされる Java クラスの形式を取ります。

それぞれのコンポーネントのルールは、コンポーネントの実行段階の特定の時点で呼び出されます。コンポーネントは、特定の署名のメソッドが使用可能になっていることを予期するため、基本ルールを拡張する際に、正確なメソッド署名を使用するように注意してください。

デフォルトまたは例のルールがすべての MQSeries Everyplace コンポーネントに提供されていますが、ソリューションの要件に適合するように、ソリューションは MQSeries Everyplace の動作をカスタマイズする独自のルールを提供することが期待されています。

### キュー・マネージャー・ルール

キュー・マネージャー・ルールが呼び出されるのは、次の場合です。

- キュー・マネージャーが活動化された。
- キュー・マネージャーがクローズされた。
- キューがキュー・マネージャーに追加された。
- キューがキュー・マネージャーから除去された。
- メッセージの書き込み操作が発生した。
- メッセージの取得操作が発生した。
- メッセージの削除操作が発生した。
- メッセージの取り消し操作が発生した。
- キュー・マネージャーが保留メッセージをすべて送信するよう起動された (送信ルール)。
- 着信対等接続が確立された。

## キュー・マネージャー・ルールの使用

以下に示すのは、キュー・マネージャー・ルールの使用法のいくつかの例です。

最初の例は、*メッセージの書き込みルール* を示しています。ここでは、このキュー・マネージャーを使用してキューに書き込まれるすべてのメッセージには、MQSeries メッセージ ID フィールドが含まれていなければならないことが強調されています。

```
/* Only allow msgs containing an ID field to be placed on the Queue */
public void putMessage( String destQMgr, String destQ, MQeMsgObject msg,
                        MQeAttribute attribute, long confirmId )
{
  if ( !(msg.Contains( MQe.Msg_MsgId )) )
    throw new MQeException( Except_Rule, "Msg must contain an ID" );
}
```

次のルールの例は、*メッセージの取得ルール* です。ここでは、″OutboundQueue″ と呼ばれるキュー上でメッセージの取得要求を処理できるようにするために、前もってパスワードを提供しなければならないことが強調されています。パスワードは、`getMessage()` メソッドに渡されるメッセージ・フィルターに、フィールドとして組み込まれます。

```
/* This rule only allows GETs from 'OutboundQueue', if a password is  */
/* supplied as part of the filter                                     */
public void getMessage( String destQMgr, String destQ, MQeFields filter,
                        MQeAttribute attr, long confirmId )
{
  super.getMessage( destQMgr, destQ, filter, attr, confirmId );
  if ( destQMgr.equals( Owner.GetName() && destQ.equals( "OutboundQueue" ) )
  {
    if ( !(filter.Contains( "Password" ) )
      throw new MQeException( Except_Rule, "Password not supplied" );
    else
    {
      String pwd = filter.getAscii( "Password" );
      if ( !(pwd.equals( "1234" )) )
        throw new MQeException( Except_Rule, "Incorrect password" );
    }
  }
}
```

このルールは、キューを保護するたいへん単純な例です。さらにセキュリティーを拡張するために、ソリューションで認証機能を使用することをお勧めします。これにより、ソリューションはアクセス制御リストを作成し、キューからメッセージを取得できる人物を管理することができます。

次のルールの例は、キュー・マネージャー管理要求がキューを除去しようとしたときに呼び出されます。ルールには、問題のキューへのオブジェクト参照が渡されます。次の例では、ルールは渡されるキューの名前をチェックし、キューが ″PayrollQueue″ という名前の場合には、キューの除去要求が拒絶されます。

```
/* This rule prevents the removal of the Payroll Queue */
public void removeQueue( MQeQueue queue ) throws Exception
{
  if ( queue.getQueueName().equals( "PayrollQueue" ) )
    throw new MQeException( Except_Rule, "Can't delete this queue" );
}
```

キュー・マネージャーは、それ独自の対等チャネル・リスナーを定義することができます。リスナーは、対等チャネルを使用して行われた他のキュー・マネージャーからの着信接続試行を検出します。以下のルールは、接続要求が検出されると呼び出されます。ルールには、接続を試行しているキュー・マネージャーの名前が渡されます。

```
public void peerConnection( String qmgrName )
{
  /* block any connection attempt from 'RogueQMgr' */
  if ( qmgrName.equals( "RogueQMgr" ) )
    throw new MQeException( Except_Rule, "Connection not allowed" );
}
```

## 伝送ルール

リモート・キューに書き込まれ、同期として定義されるメッセージは、即時に伝送されます。非同期として定義され、リモート・キューに書き込まれるメッセージは、キュー・マネージャーがその宛先への伝送に起動されるようなときまで、ローカル・キュー・マネージャーに保管されます。キュー・マネージャーはアプリケーションによって直接起動することができますが、処理はキュー・マネージャーの伝送ルールによっても制御されます。

伝送ルールは、メッセージがキュー・マネージャーから伝送される仕方を制御するキュー・マネージャー・ルールのサブセットです。

ルール・クラス内には、メッセージ伝送を制御するための次の 2 つのメソッドがあります。

**triggerTransmission()**
: ルールが呼び出されるときにメッセージ伝送を許可するかどうかを指定します。

**transmit()**
: それぞれのキューごとに、伝送を許可するかどうかを決定します。たとえば、優先順位が高いとみなされるキューからのメッセージだけを伝送できるようにします。 transmit() ルールは、triggerTransmission() ルールが正常に戻る場合にのみ呼び出されます。

***トリガー伝送ルール:*** トリガー伝送ルールは、メッセージがリモート非同期キューに書き込まれるときに呼び出されます。キュー・マネージャーの `triggerTransmission` メソッドは、このルールを変更し、保留メッセージの伝送を試行します。

```
/* default transmission rule - always allow transmission */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  return true;
}
```

このルールからの戻りコードは、保留メッセージを伝送するかどうかをキュー・マネージャーに知らせます。戻りコードが true であれば伝送し、false であれば、この時点では伝送しません。したがって、上記のルールでは、すべてのメッセージを即時に伝送しようとします。これは、基本キュー・マネージャー・ルール class com.ibm.mqe.MQeQueueManagerRule に含まれている、デフォルトの `triggerTransmission()` ルールです。ルールはメッセージがキューに書き込まれるとすぐにメッセージを伝送しようとします。この同期モードに近い操作は、あまり

望ましくはありません。なぜなら、すべてのメッセージを別々に送信するので、効率が悪いからです。通常は、メッセージはグループとして送信し、ネットワークをより効率的に使用するほうが勝っています。

もっと複雑なルールでは、メッセージの優先順位に基づいて、即時に伝送するかどうかを指定することができます。次の例は、優先順位が 5 より大きいメッセージが到着した場合に、キュー・マネージャーを起動するルールを示しています。

次の例は、メッセージ伝送を開始するに、アプリケーションによって送信されるトリガー・メッセージを使用できるようにしています。

```
/* Decide to transmit based on priority of message */
public boolean triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  if ( msgFields == null ) /* msg fields may be null */
    return false;
  if ( !(msgFields.contains( MQe.Msg_Priority )) )
    return false; /* no priority field in message */
  byte priority = msg.GetByte( MQe.Msg_Priority );
  if ( priority > 5 ) /* if message priority greater than 5 */
    return true; /* then transmit */
  else
    return false; /* else do not transmit */
}
```

msgFields パラメーターには、メッセージから選択したフィールドが含まれます。これらのフィールドは、次のとおりです。

- Unique ID (固有 ID)
- Message ID (メッセージ ID)
- Correlation ID (相関 ID)
- Priority (優先順位)

ルールが伝送を許可することを決定した場合、非同期リモート・キューに書き込まれたメッセージだけでなく、すべての保留メッセージが伝送されます。

noOfMsgs パラメーターには、伝送を待機しているメッセージの数が含まれています。ソリューションは、保留メッセージが一定の数になるまで、伝送をブロックするルールをインプリメントするよう指定することができます。このようなルールは、ネットワーク接続をより効率的に使用するのに役立ちます。

以下のルールは、伝送を待機しているメッセージが最低 10 個になるまで、伝送をブロックします。

```
public void triggerTransmission( int noOfMsgs, MQeFields msgFields )
{
  if ( noOfMsgs >= 10 ) /* if more than 10 msgs are waiting */
    return true; /* then transmit */
  else
    return false;
}
```

***伝送ルール:*** transmit() ルールは、triggerTransmission() ルールが伝送を許可する場合にのみ呼び出されます (戻り値 true)。 transmit() ルールは、伝送を待機しているメッセージを保持するすべてのリモート・キュー定義ごとに呼び出されます。つまり、キュー単位でどのメッセージが伝送されるかをルールが決定できるという意味です。

## キュー・マネージャー・ルール

以下のルールがキューからのメッセージ伝送を許可するのは、キューのデフォルトの優先順位が 5 より大きい場合だけです。 (メッセージがキューに置かれる前に優先順位が割り当てられていない場合は、キューのデフォルトの優先順位が与えられます。)

```
public boolean transmit( MQeQueue queue )
{
  if ( queue.getDefaultPriority() > 5 )
    return (true);
  else
    return (false);
}
```

このルールを拡張して、すべてのメッセージをオフピーク時に伝送されるようにし、優先順位の高いキューのメッセージだけをピーク時に伝送するようにするのは、実際的な方法です。同様のアイデアをインプリメントしたいくつかのルールについては、次のセクションを参照してください。

このルールは、キューに含まれるメッセージが 10 を超えた場合にのみ、メッセージの伝送を許可します。

```
public boolean transmit( MQeQueue queue )
{
  if ( queue.getNumberOfMessages() >= 10 )
    return (true);
  else
    return (false);
}
```

次の複雑な例は、伝送に要する時間に対して課金する通信ネットワーク上で、メッセージの伝送が行われていることを想定しています。また、単位時間のコストが比較的に低い低料金時間があることも想定しています。ルールは、低料金時間になるまで、メッセージの伝送をブロックします。低料金時間中は、キュー・マネージャーが定期的に起動されます。

```
import com.ibm.mqe.*;
import java.util.*;

/**
  * Example set of  queue manager Rules which trigger the transmission
  * of any messages waiting to be sent.
  *
  * These rules only trigger the transmission of messages if the current
  * time is between the values defined in the variables cheapRatePeriodStart
  * and cheapRatePeriodEnd

  * (This example assumes that transmission will take place over a
  * communication network which charges for the time taken to transmit)
 */
public class ExampleQueueManagerRules extends MQeQueueManagerRule
                                      implements Runnable
{
  /* default interval between triggers is 10 minutes                        */
  public final int triggerInterval   = 600000;
  /* cheap rate transmission period start and end times        */
  public final int cheapRatePeriodStart = 18; /* 18:00 hrs        */
  public final int cheapRatePeriodEnd   = 9; /* 09:00 hrs        */

  /* background thread reference        */
  protected Thread    th                = null;
}
```

ルールの例は、基本キュー・マネージャー・ルール・クラス com.ibm.mqe.MQeQueueManagerRule から拡張したものです。

定数 cheapRatePeriodStart および cheapRatePeriodEnd は、この低料金時間の範囲を定義します。この例では、低料金は夜の 18:00 時から次の日の朝の 09:00 時までと定義されています。

```
/* cheap rate transmission period start and end times         */
public final int cheapRatePeriodStart = 18; /* 18:00 hrs */
public final int cheapRatePeriodEnd = 9; /* 09:00 hrs */
```

定数 triggerInterval は、キュー・マネージャーが毎回起動されるまでの間の時間を定義します (ミリ秒)。

```
public final int triggerInterval = 600000;
```

この例では、トリガー間隔は 600,000 ミリ秒と定義されています。これは、600 秒つまり 10 分と同じです。

キュー・マネージャーのトリガーは、triggerInterval 期間の最後に「覚せい」するバックグラウンド・スレッドによって処理されます。現在の時刻が低料金時間内であれば、それは MQeQueueManager.triggerTransmission() ルールを呼び出して、伝送を待機しているすべてのメッセージの伝送試行を開始します。

バックグラウンド・スレッドは、queueManagerActivate() ルールで作成され、queueManagerClose() ルールで停止します。キュー・マネージャーがこれらのルールを呼び出すのは、それが活動化され、クローズされたときです。

```
 /**
  * Overrides MQeQueueManagerRule.queueManagerActivate()
  * Starts a timer thread
  */
  public void queueManagerActivate()
  {
    /* background thread which triggers XmitQ */
    th = new Thread( this );
    th.start();                     /* start timer thread */
  }
 /**
  * Overrides MQeQueueManagerRule.queueManagerClose()
  * Stops the timer thread
  */
  public void queueManagerClose()
  {
    th.stop();                      /* stop timer thread                    */
  }
```

次は、バックグラウンド・スレッドを処理するコードの例です。

```
/**
 * Timer thread
 * Triggers queue manager every interval until thread is stopped
 */
 public void run()
 {
   try
   {
     while ( true )
     { /* sleep for specified interval                                    */
       Thread.sleep( triggerInterval );
       /* if cheap rate period call queue manager to trigger transmission */
       if ( timeToTransmit() )
```

```
            ((MQeQueueManager)owner).triggerTransmission();
        }
      }
      catch ( Exception e )
      {
        e.printStackTrace( System.err );
      }
    }
```

変数の所有者はクラス MQeRule によって定義されます。これは、MQeQueueManagerRule の祖先です。その始動プロセスの一部として、キュー・マネージャーはキュー・マネージャー・ルールを活動化し、それ自体への参照をルール・オブジェクトに渡します。その後、これは変数所有者に保管されます。

スレッドは絶えずループし (queueManagerClose() ルールによって停止されることに注意)、トリガー間隔期間の最後になるまでスリープし、その後、現在の時刻が低料金伝送時間内であるかをチェックします。このことは、timeToTransmit() メソッドを呼び出して実行します。このメソッドが成功すると、キュー・マネージャーの triggerTransmission() ルールが呼び出されます。

timeToTransmit メソッドは、次のコードで示されます。

```
protected boolean timeToTransmit()
{
  /* get current time */
  long currentTimeLong = System.currentTimeMillis();

  Date date = new Date( currentTimeLong );
  Calendar calendar = Calendar.getInstance();
  calendar.setTime( date );

  /* get hour */
  int hour = calendar.get( Calendar.HOUR_OF_DAY );

  if ( hour >= cheapRatePeriodStart || hour < cheapRatePeriodEnd )
    return true; /* cheap rate */
  else
    return false; /* not cheap rate */
}
```

## 非同期リモート・キュー定義の活動化

キュー・マネージャーは起動時に、その非同期リモート・キュー定義を活動化することができます。キューを活動化するとは、キューに含まれているメッセージの伝送を試行することを意味します。この動作は、activateQueues() ルールによって構成することができます。

基本的なルールでは、true または false を戻すだけです。

```
public boolean activateQueues()
{
  return true; /* always transmit on activate */
}
```

前述のセクションの例にある他のルールと同様に、現在の時刻が低料金時間内かどうかをチェックすることができます。

```
public boolean activateQueues()
{
  if ( timeToTransmit() )
```

```
    return true;
  else
    return false;
}
```

このルールは、起動時にホーム・サーバー・キューおよびストア・アンド・フォワード (蓄積交換) キューが活動化されるかどうかも判別します。

activateQueues() が false を戻す場合には、メッセージがリモート・キュー定義に書き込まれるときに限り、リモート・キュー定義が活動化されます。ホーム・サーバー・キューは、キュー・マネージャーの triggerTransmission() ルールを呼び出すことによって、活動化することができます。

## キュー・ルール

それぞれのキューには、独自のルールのセットがあります。ソリューションは、これらのルールの動作を拡張することができます。すべてのキュー・ルールは、com.ibm.mqe.MQeQueueRule の子孫でなければなりません。

キュー・ルールが呼び出されるのは、次の場合です。

- キューが活動化された。
- キューがクローズされた。
- メッセージがキューに置かれた (Put)。
- メッセージがキューから除去された (Get)。
- メッセージがキューから削除された (Delete)。
- キューがブラウズされた。
- 取り消し操作がキュー上のメッセージに対して実行された。
- メッセージ・リスナーがキューに追加された。
- メッセージ・リスナーがキューから除去された。
- メッセージの有効期限が切れた。
- キューの使用回数が変更された。
- キューの属性 (認証機能、暗号機能、圧縮機能) を変更しようとした。
- メッセージの索引項目が作成された。

### 索引項目ルール

キューはメモリーにすべてのメッセージを保持するわけではありません。メッセージをキュー・ストアに保管し、必要なときにメモリーに復元します。キューはそのキュー・ストア内に保持しているメッセージごとに索引項目を保守します。索引項目には、ロックされているかロックが解除されているかといった、メッセージの状態情報が含まれています。また、メッセージからの特定のフィールド (*索引フィールド* と呼ばれる) が、索引項目に保管されます。デフォルトの索引フィールドは、メッセージ固有 ID、 MQSeries メッセージ ID、MQSeries 相関 ID、およびメッセージ優先順位です。これらのフィールドが保管されるのは、ほとんどのメッセージに存在するためであり、それらのフィールドをメモリーに保管することにより、より高速なメッセージ検索が可能になります。

索引項目が作成されると、indexEntry() ルールが呼び出されます。新規メッセージがキューに書き込まれるときには必ずこれが行われます。また、キューが活動化さ

れるときや、キューが以前のセッションからキュー・ストアに残っているメッセージを読み取るときにも行われます。このルールにより、ソリューションは、作成時に索引項目を更新することができます。この 1 つの使用法は、共通に使用される追加フィールド (1 つまたは複数) を索引に追加して、メッセージ検索時間を短縮させることです。

```
/* if the message contains a customer number field - then add this field */
/* to the message's index entry.  */
/* This will enable faster message searching */
public void indexEntry( MQeFields entry,
                        MQeMsgObject msg ) throws Exception
{
  if ( msg.contains( "Cust_No" ) )
    entry.copy( msg, true, "Cust_No" );
}
```

パラメーター entry には、メッセージのブランク索引項目が入ります (indexEntry ルールが戻った後、デフォルトの索引フィールドが、キューによって追加されます)。上記の例では、メッセージに Cust_No というフィールドが含まれる場合、これがメッセージの索引項目に追加されます。

取得やブラウズのような、後続のメッセージング操作において、アプリケーションは Cust_No フィールドを、操作に提供されるフィルターの一部として使用することができます。アプリケーションが値 75 を持つ Cust_No フィールド、および値 115 を持つ Order_No フィールドを含むメッセージを検出しようとしているとします。この場合、キューは、まず値が '75' の Cust_No フィールドを含むメッセージをメモリーにロードし、次に、それらが指定された値を持つ Order_No フィールドを含むかどうかを調べるだけですみます。 Cust_no フィールドが索引の一部でない場合には、キュー上の最初のメッセージから始めて、すべてのメッセージがメモリーにロードされ、それがフィルターに一致するフィールドを含んでいるかどうかが調べられます。

もちろん、索引フィールドの使用は妥協策にすぎません。それらを使用してメッセージ検索時間を短縮することができます (これは、パーベイシブ・デバイスにおいて奨励されます) が、索引フィールドがメモリーに保持されます。

## メッセージ有効期限切れルール

キューとメッセージのどちらにも、有効期限を設定することができ、この期限を超過すると、メッセージに '有効期限切れ' というフラグが立てられます。その時点で、messageExpired() ルールが呼び出されます。その場合、このルールが、メッセージをどのように処理するかを決定します。一般に、メッセージを削除するか、送達不能キューに置くかのいずれかです。しかし、たとえば、ルールはキュー上にメッセージを完全な状態で残して、MQSeries Everyplace アプリケーションに見えるようにしておくよう決定することができます。

```
/* This rule puts a copy of any expired messages to a Dead Letter Queue */
public boolean messageExpired( MQeFields entry,
                               MQeMsgObject msg ) throws Exception
{
  /* Get the reference to the Queue Manager */
  MQeQueueManager qmgr = MQeQueueManager.getReference(
                                    ((MQeQueue)owner).getQueueManagerName() );
  /* need to set re-send flag so that put of message to new queue isn't */
  /* rejected */
  msg.putBoolean( MQe.Msg_Resend, true );
  /* if the message contains an expiry interval field - remove it */
```

```
  if ( msg.contains( MQe.Msg_ExpireTime )
    msg.delete( MQe.Msg_ExpireTime );
  /* put message onto dead letter queue */
  qmgr.putMessage( null, MQe.DeadLetter_Queue_Name, msg, null, 0 );
  /* return true & the message will be deleted from the queue */
  return (true);
}
```

前述の例では、有効期限が切れたメッセージがキュー・マネージャーの送達不能キューに送信されます。そのキューの名前は、定数 MQe.DeadLetter_Queue_Name によって定義されています。ここでの注意事項として、キュー・マネージャーは、すでに別のキューに書き込まれているメッセージの書き込みを拒否します (これは、重複したメッセージが MQSeries Everyplace ネットワークに送信されるのを防ぐためです)。したがって、送達不能キューにメッセージを移す前に、その再送フラグを設定する必要があります。これは、メッセージに MQe.Msg_Resend フィールドを追加することによって行われます。また、メッセージを送達不能キューに移す前に、その有効期限フィールドを削除する必要があります。

'true' の値を戻すと、ルールがメッセージの有効期限が切れたと判別したことがキューに通知されます。

***追加のメッセージ・リスナー・イベントのログ記録:*** 次の例は、キュー上で発生するイベントをログに記録する方法を示しています。この例で発生するイベントは、メッセージ・リスナーの作成ですが、メッセージの書き込みやメッセージ要求のブラウズといった他のキュー・イベントの場合も、基本的なことは同じです。

例では、キューに独自のログ・ファイルがありますが、すべてのキューで使用される中央ログ・ファイルを持っているのと効果は同じです。キューは、活動化されたときにはログ・ファイルを開く必要があり、またキューがクローズされるときにはログ・ファイルをクローズする必要があります。キュー・ルール queueActivate および queueClose を使用して、このことを行うことができます。どちらのルールもログ・ファイルにアクセスできるように、変数 logFile はクラス変数である必要があります。

```
/* This rule logs the activation of the queue */
public void queueActivate()
{
  try
  {
    logFile = new LogToDiskFile( ¥¥log.txt );
    log( MQe_Log_Information, Event_Queue_Activate, "Queue " +
         ((MQeQueue)owner).getQueueManagerName() + " + " +
         ((MQeQueue)owner).getQueueName() + " active" );
  }
  catch( Exception e )
  {
    e.printStackTrace( System.err );
  }
}
/* This rule logs the closure of the queue */
public void queueClose()
{
  try
  {
    log( MQe_Log_Information, Event_Queue_Closed, "Queue " +
         ((MQeQueue)owner).getQueueManagerName() + " + " +
         ((MQeQueue)owner).getQueueName() + " closed" );
    /* close log file */
    logFile.close();
```

```
  }
  catch ( Exception e )
  {
    e.printStackTrace( System.err );
  }
}
```

addListener rule は、次のコードで示されます。これは MQe.log メソッドを使用して、Event_Queue_AddMsgListener イベントを追加します。

```
/* This rule logs the addition of a message listener */
public void addListener( MQeMessageListenerInterface listener,
                         MQeFields filter ) throws Exception
{
  log( MQe_Log_Information, Event_Queue_AddMsgListener,
       "Added listener on queue " +
       ((MQeQueue)owner).getQueueManagerName() + "+" +
       ((MQeQueue)owner).getQueueName() );
}
```

# 第5章 MQSeries Everyplace 管理

キュー・マネージャーやキューのような MQSeries Everyplace リソースの管理は、専用の MQSeries Everyplace メッセージを使用して行われます。メッセージを使用して管理を行うと、 MQSeries Everyplace リソースのローカルおよびリモート管理を明快に把握することができます。

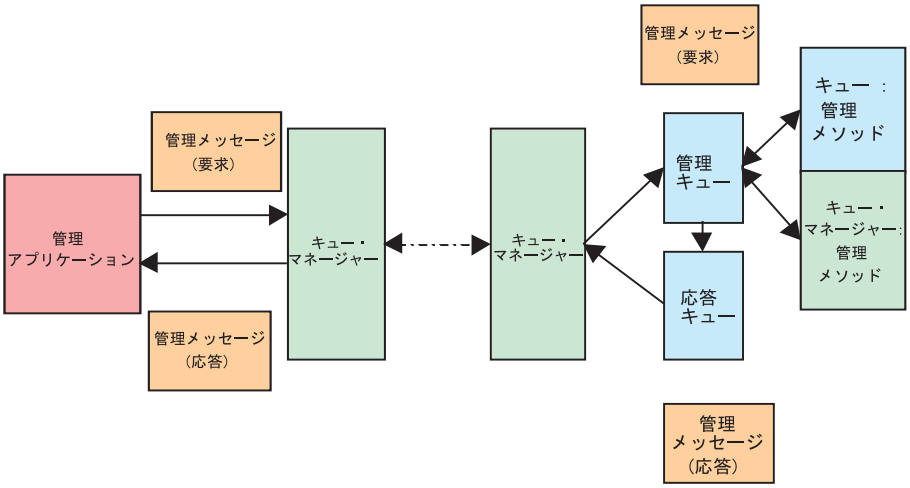


*図 11. MQSeries Everyplace 管理*

キュー・マネージャーやそのリソース上で何らかの管理を実行するためには、まずキュー・マネージャーが正常に開始されていなければならず、その上に専用の管理キューが構成されている必要があります。管理キューには、キューが受け取った管理メッセージを順番に処理するという役割があります。 1 度に処理できる要求は 1 つだけです。キューは、 MQeQueueManagerConfigure クラスの `defineDefaultAdminQueue()` メソッドを使用して作成することができます。キューの名前は AdminQ で、プログラムからは定数 `MQe.Admin_Queue_Name` を使用して参照することができます。

一般的な管理アプリケーションでは、まず MQeAdminMsg のサブクラスをインスタンス化し、これを必要な管理要求と共に構成して宛先キュー・マネージャーの AdminQ に渡します。アクションの結果を知りたい場合は、応答を要求することができます。要求が処理されると、要求メッセージで指定した応答先キュー / キュー・マネージャーに、要求の結果を示すメッセージが返されます。

応答は任意のキュー・マネージャーまたはキューに送ることができますが、デフォルトの応答先キューを構成して、管理応答メッセージ専用のキューにすることもできます。このキューは、MQeQueueManagerConfigure クラスの `defineDefaultAdminReplyQueue()` メソッドを使用して作成することができます。キューの名前は "AdminReplyQ" で、プログラムからは定数 `MQe.Admin_Reply_Queue_Name` を使用して参照することができます。

管理キューには、個々のリソースの管理を実行する方法についての情報はありません。この情報は、各リソースおよびそれに対応する管理メッセージにカプセル化されます。 MQSeries Everyplace リソースの管理においては、次のメッセージが使用されます。

| メッセージ名 | 目的 |
|---|---|
| **MQeAdminMsg** | すべての管理メッセージの基底クラスとして使用される要約クラス |
| **MQeAdminQueueAdminMsg** | 管理キューの管理をサポートする |
| **MQeConnectionAdminMsg** | キュー・マネージャー間の接続の管理をサポートする |
| **MQeHomeServerQueueAdminMsg** | ホーム・サーバー・キューの管理をサポートする |
| **MQeQueueAdminMsg** | ローカル・キューの管理をサポートする |
| **MQeQueueMangerAdminMsg** | キュー・マネージャーの管理をサポートする |
| **MQeRemoteQueueAdminMsg** | リモート・キューの管理をサポートする |
| **MQeStoreAndForwardQueueAdminMsg** | ストア・アンド・フォワード (蓄積交換) キューの管理をサポートする |
| **MQeMQBridgeQueueAdminMsg** | MQSeries システムに接続するキューの管理をサポートする |

これらのベースとなる管理メッセージは、 `com.ibm.mqe.administration` パッケージに組み込まれています。管理対象リソースの各タイプには、それぞれ独自の管理メッセージがあります。追加のタイプやリソースは、 MQeAdminMsg またはいずれかの既存の管理メッセージにサブクラス化することによって管理できます。たとえば、`com.ibm.mqe.mqbridge` パッケージには、MQSeries ブリッジを管理するための管理メッセージの追加のセットが含まれています。

## 基本となる管理要求メッセージ

MQSeries Everyplace リソースの管理要求には、すべて同じ基本の形式があります。基本的なインプリメンテーションは、MQeAdminMsg クラスから提供されます。次の図は、すべての管理要求メッセージに使用される基本的な構造を示しています。

要求は次のものから成っています。

1. すべての管理要求に共通する、管理固有のフィールド
2. 管理されるリソースに固有な、管理のフィールド
3. 管理メッセージの処理を容易にする、オプションのフィールド

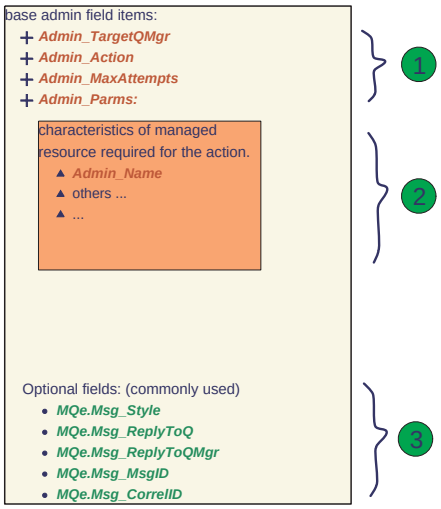


*図 12. 管理要求メッセージ*

## 固有のフィールドの管理

管理要求では、要求の宛先キュー・マネージャー (Admin_TargetQMgr、要求が実行されるキュー・マネージャー) について詳細な情報を提供しなければなりません。これは、ローカル・キュー・マネージャーの場合にも、リモート・キュー・マネージャーの場合にも同じです。 Java 仮想マシンでは、1 度に 1 つのキュー・マネージャーしかアクティブにすることができないため、メッセージは宛先キュー・マネージャーにしか置くことができません。

各管理対象リソースでは、一連の管理アクションを実行することができますが、管理メッセージで要求できるのは、実行される 1 つのアクションだけです。このアクションは、Admin_Action フィールドで指定します。定義されている一連の共通アクションは次のとおりです。

| | |
|---|---|
| Action_Create | 管理対象リソースの新しいインスタンスを作成する |
| Action_Delete | 既存の管理対象リソースを削除する |
| Action_Inquire | 管理対象リソースの 1 つまたは複数の特性を照会する |
| Action_InquireAll | 管理対象リソースのすべての特性を照会する |
| Action_Update | 管理対象リソースの 1 つまたは複数の特性を更新する |

これらのアクションは、必ずしもすべてのリソースで実装する必要はありません。たとえば、管理メッセージを使用してキュー・マネージャーを作成することはできません。特定の管理メッセージでは、基本のセットを拡張して、リソースに固有の付加的なアクションを実行できるようにすることもできます。それぞれの共通アクションは、次のメソッドで Admin_Action フィールドに設定することができます。

| | |
|---|---|
| Action_Create | create( MQeFields parms ) |
| Action_Delete | delete( MQeFields parms ) |
| Action_Inquire | inquire( MQeFields parms ) |
| Action_InquireAll | inquireAll( MQeFields parms ) |
| Action_Update | update( MQeFields parms ) |

アクションが実行されると、それは正常に完了するかあるいは失敗します。リソースが適切な状況にない (たとえば、キューに対して delete が発行されたがキューは使用中だった) ためにアクションが失敗した場合は、Admin_MaxAttempts を 1 より大きな値に設定しておけば、自動的にアクションを再試行することができます。再試行は、キュー・マネージャーを次に再始動する際に実行することもできますし、管理キューに設定されている次のインターバルで実行することもできます。

ほとんどの失敗では、応答メッセージの中で詳細な情報を確認することができます。失敗の情報の読み取りと処理は、要求を出したアプリケーションで行われます。

次の一連のメソッドを使って、いくつかの要求のフィールドを設定および使用することができます。

| Admin_Parms | MQeFields | MQeFields getInputFields() |
|---|---|---|
| Admin_Action | int | setAction(int action) |
| | | int getAction() |
| Action_TargetQMgr | ascii | setTargetQMgr(String qmgr) |
| | | String getTargetQMgr() |
| Action_MaxAttempts | int | setMaxAttempts(int attempts) |
| | | int getMaxAttempts() |

## 管理対象ノードに固有のフィールド

各リソースには、そのリソースに固有な一連の特性があります。それぞれの特性には名前とタイプと値があります。各特性の名前は管理メッセージの中で定数によって定義されます。これらの特性の中には、すべての管理対象リソースに共通する特性が 1 つありますが、それはリソースの名前という特性です。リソースの名前を定義するフィールドの名前は Admin_Name で、これは ascii タイプのフィールドです。一連の特性をすべて判別するには、管理メッセージのインスタンスに対して `characteristics()` メソッドを使用します。このメソッドを使用すると、各特性ごとに 1 つのフィールドを含む MQeFields オブジェクトが返されます。各特性の名前、タイプ、およびデフォルト値を確認するには、MQeFields メソッドを使用して一連の特性を列挙することができます。

アクションにパラメーターとして渡すことのできる一連の特性は、要求されるアクションによって異なりますが、リソースの名前 Admin_Name だけは、どんな場合でも必ず渡されなければなりません。なお、Action_InquireAll が要求される場合は、このパラメーターが唯一の必須パラメーターとなります。

パラメーターを指定するには、管理メッセージ内の組み込み MQeFields オブジェクトにこれを書き込みます。パラメーターを書き込むフィールドの名前は Admin_Parms です。例として、管理メッセージで管理するリソースの名前を設定する場合は、次のようなコードが使用されます。

```
SetResourceName( MQeAdminMsg msg, String name )
{
  MQeFields parms;
  if ( msg.contains( Admin_Parms ) )
    parms = msg.getFields( Admin_Parms );
  else
    parms = new MQeFields();
  parms.putAscii( Admin_Name, name );
  msg.putFields( Admin_Parms, name );
}
```

あるいは別の方法として、メッセージから Admin_Parms フィールドを返す getInputFields() メソッドか、メッセージに Admin_Name フィールドを設定する setName() を使用するなら、コードを簡単にすることができます。この場合、上のコードは次のようになります。

```
SetResourceName( MQeAdminMsg msg, String name )
{
  msg.SetName( name );
}
```

## 他の便利なフィールド

デフォルトでは、管理要求が処理されたときに応答は行われません。要求の結果が必要な場合は、応答のメッセージを求めるように要求のメッセージをセットアップする必要があります。応答の要求には、いくつかのフィールドが関係しています。これらのフィールドは、すべて MQSeries Everyplace クラスで定義されています。

**Msg_Style**
: 次の 3 つの値のいずれかをとれる int タイプのフィールド

  **Msg_Style_Datagram**
  : 応答を要求しないコマンド

  **Msg_Style_Request**
  : 応答を求める要求

  **Msg_Style_Reply**
  : 要求に対する応答

Msg_Style を Msg_Style_Request (応答を要求する) に設定した場合は、要求メッセージの中で応答先の位置を設定する必要があります。応答先の設定には、次の 2 つのフィールドを使用します。

**Msg_ReplyToQ**
: 応答が書き込まれるキューの名前を保持する ascii フィールド

**Msg_ReplyToQMgr**
: 応答を受け取るキュー・マネージャーの名前を保持する ascii フィールド

## 管理要求メッセージ

応答先のキュー・マネージャーと要求を処理するキュー・マネージャーが異なる場合、要求を処理するキュー・マネージャーには応答先のキュー・マネージャーへの接続を定義する必要があります。

管理要求メッセージをその応答メッセージと相互に関連付ける場合は、要求を一意的に識別し、かつ応答メッセージにコピーすることのできるフィールドをその要求メッセージに含める必要があります。 MQSeries Everyplace には、この目的で使用することのできる、 Msg_MsgID および Msg_CorrelID という 2 つのフィールドがあります。要求メッセージに含めるフィールド自体は他のどんなフィールドでも構いませんが、この 2 つのフィールドが持つ大きな利点として、キュー・マネージャーは、これらのフィールドを使用した場合に最も効率的にキューやメッセージを検索することができます。いずれのフィールドにおいても、含まれるデータのタイプは byte array でなければなりません。以下のコード・フラグでは、要求メッセージを作成する方法の例を示しています。

```
public class LocalQueueAdmin extends MQe
{
  public String   targetQMgr = "ExampleQM";  // target queue manager

public MQeFields primeAdminMsg(MQeAdminMsg msg) throws Exception
{
  /*
   * Set the target queue manager that will process this message
   */
  msg.setTargetQMgr( targetQMgr );

  /*
   * Ask for a reply message to be sent to the queue
   * manager that processes the admin request
   */
  msg.putInt  (MQe.Msg_Style,       MQe.Msg_Style_Request);
  msg.putAscii(MQe.Msg_ReplyToQ,    MQe.Admin_Reply_Queue_Name);
  msg.putAscii(MQe.Msg_ReplyToQMgr, targetQMgr);

  /*
   * Setup the correl id so we can match the reply to the request.
   * - Use a value that is unique to the this queue manager.
   */
  byte[] correlID = Long.toHexString( (MQe.uniqueValue()).getBytes() );
  msg.putArrayOfByte( MQe.Msg_CorrelID, correlID );

  /*
   * Ensure matching response message is retrieved
   * - set up a fields object that can be used as a match parameter
   *   when searching and retrieving messages.
   */
  MQeFields msgTest = new MQeFields();
  msgTest.putArrayOfByte( MQe.Msg_CorrelID, correlID );

  /*
   * Return the unique filter for this message
   */
  return msgTest;
}
```

管理要求のメッセージを作成したなら、通常の MQSeries Everyplace メッセージ処理 API を使用してこれを宛先キュー・マネージャーに送ることができます。メッセージを同期で送達できるか非同期で送達できるかは、宛先の管理キューをどのように定義するかによって異なります。

次いで、通常の MQSeries Everyplace メッセージ処理 API を使用して、応答のメッセージ、あるいはその通知を待つことができます。メッセージ・キューイング・システムによって、要求の送信と応答メッセージの受信との間には時間のずれが生じます。この時間のずれは、要求がローカルに処理される場合には短く、要求も応答メッセージも共に非同期で送達される場合には長くなります。たとえば、次のコード・フラグを使用して、要求メッセージを送信し、その応答を待つことができます。

```
public class LocalQueueAdmin extends MQe
{
  public String   targetQMgr = "ExampleQM";  // target queue manager
  public int      waitFor    =   10000;      // millsecs to wait for reply

/*
 * Send a completed admin message.
 *   Uses the simple putMessage method which is not assured if the
 *   the queue is defined for synchronous operation.
 */
public void sendRequest( MQeAdminMsg msg ) throws Exception
{
  myQM.putMessage( targetQMgr,
                   MQe.Admin_Queue_Name,
                   msg,
                   null,
                   0 );
}

/*
 * Wait a while for a reply message. This method will wait for
 * a limited time on either a local or a remote reply to queue.
 * Parameters:
 *   msgTest: a filter for the reply message to wait for
 * Returns:
 *   respMsg: a reply message matching the msgTest filter.
 */
public MQeAdminMsg waitForReply( MQeFields msgTest ) throws Exception
{
  MQeAdminMsg respMsg = null;
  respMsg = (MQeAdminMsg)myQM.waitForMessage(targetQMgr,
                                             MQe.Admin_Reply_Queue_Name,
                                             msgTest,
                                             null,
                                             0,
                                             waitFor);
  return respMsg;
}
```

## 基本となる管理応答メッセージ

管理要求が処理されると、応答 (要求された場合) が応答先のキュー・マネージャー / キューに送られます。応答メッセージは、要求メッセージと同じ基本の形式を持っていて、要求メッセージよりもいくつか多くのフィールドを含んでいます。

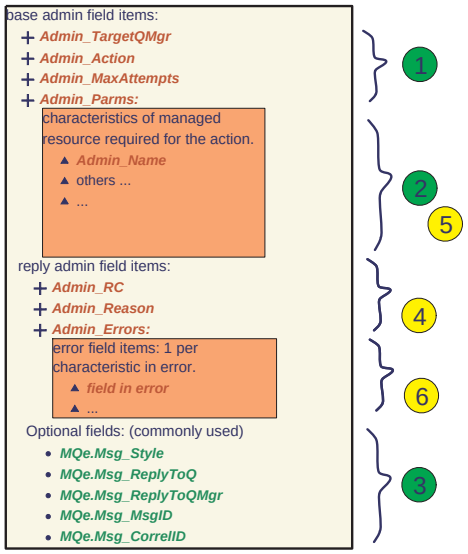


*図 13. 管理応答メッセージ*

応答は次のものから成っています。

1. 管理固有のフィールド (要求メッセージからコピーされる)
2. 管理のフィールド
3. 管理メッセージの処理を容易にする、オプションのフィールド (要求メッセージからコピーされる)
4. 要求の結果を詳細に記述する管理のフィールド
5. 管理されるリソースに固有な、要求の結果を詳細に記述する管理のフィールド
6. 管理されるリソースに固有な、エラーの詳細を示す管理のフィールド

## 応答メッセージに固有な管理のフィールド

要求の全体の結果は、Admin_RC フィールドを調べることによって判別できます。これは `int` タイプのフィールドで、次の 3 つのうちのいずれかの値をとります。

- MQeAdminMsg.RC_Success
- MQeAdminMsg.RC_Failed
- MQeAdminMsg.RC_Mixed

'Success' は、アクションが正常に完了したことを意味します。 'Failed' は、要求が失敗したことを意味し、'Mixed' は、要求が部分的に成功したことを意味します。たとえば、4 つのキューの属性を更新する要求が出され、3 つの属性で更新が正常に完了したがその 1 つで要求が失敗した場合は、戻りコード Mixed が返されます。Mixed や Failed が返された場合には、`Unicode` タイプの Admin_Reason フィールドで、大まかな失敗の理由を確認することができます。

次の一連のメソッドを使用して、いくつかの応答のフィールドを確認することができます。

| | | |
|---|---|---|
| Admin_RC | int | int getAction() |
| Action_Reason | unicode | String getReason() |
| Action_Parms | MQeFields | MQeFields getOutputFields() |
| Action_Errors | MQeFields | MQeFields getErrorFields() |

実行されるアクションによっては、関係するフィールドだけが、戻りコードや理由として返されます。たとえば、delete などの場合がそうです。一方 inquire のような他のアクションの場合、要求は、応答メッセージで返されるもっと詳細な情報に対するものとなります。これらの詳細は、Action_Parms フィールドに返され、getOutputFields() メソッドで検索することができます。 Action_arms は MQeFields タイプのフィールドで、このフィールドには、管理対象リソースの各特性ごとに 1 つのフィールド項目が含まれています。たとえば、フィールド Queue_Description および Queue_FileDesc に対して inquire 要求が出された場合、結果の MQeFields オブジェクトには、実際のキューの値を含むその両方のフィールドの項目が含まれます。次の表は、要求の Admin_Parms フィールドと、キューのいくつかのパラメーターでの inquire に対する応答メッセージのフィールドを示したものです。

| **要求メッセージ: Admin_Parms フィールド** | | | **応答メッセージ: Admin_Parms フィールド** | | |
|---|---|---|---|---|---|
| **フィールド名** | **タイプ** | **値** | **フィールド名** | **タイプ** | **値** |
| Admin_Name | ascii | ″TestQ″ | Admin_Name | ascii | ″TestQ″ |
| Queue_QMgrName | ascii | ″ExampleQM″ | Queue_QMgrName | ascii | ″ExampleQM″ |
| Queue_Description | Unicode | ヌル | Queue_Description | Unicode | ″A test queue″ |
| Queue_FileDesc | ascii | ヌル | Queue_FileDesc | ascii | ″c:¥queues¥″ |

応答で付加的なデータが予期されないアクションでは、応答の Admin_Parms フィールドと要求メッセージの同じフィールドが一致します。これは、create アクションや update アクションの場合に当てはまります。

create や update のようないくつかのアクションでは、管理対象リソースの複数の特性を設定または更新するよう要求する場合があります。このような場合には、一部の更新は成功するものの、それ以外の更新が失敗する可能性があります。一部の更新は失敗したが、アクション全体が失敗しなかった場合は、戻りコード RC_Mixed が返されます。それぞれの更新がなぜ失敗したかを示す追加の詳細は、Action_Errors フィールドで確認することができます。このフィールドは、getErrorFields() メソッドで検索することができます。 Action_Errors は MQeFields タイプのフィールドで、失敗した各更新ごとに 1 つのフィールドが含まれています。 Action_Errors フィールドに含まれる各項目は、ascii タイプか asciiArray タイプです。次の表は、キューの更新の要求に対する Admin_Parms フィールドと結果の Admin_Errors フィールドの例を示したものです。

*表 2. キューの更新の要求メッセージ*

| **要求メッセージ: Admin_Parms フィールド** | | |
|---|---|---|
| **フィールド名** | **タイプ** | **値** |
| Admin_Name | ascii | ″TestQ″ |
| Queue_QMgrName | ascii | ″ExampleQM″ |
| Queue_Description | Unicode | ヌル |
| Queue_FileDesc | ascii | ヌル |

*表 3. キューの更新に対する応答メッセージ*

| **フィールド名** | **タイプ** | **値** |
|---|---|---|
| **応答メッセージ: Admin_Parms フィールド** | | |
| Admin_Name | ascii | ″TestQ″ |
| Queue_QMgrName | ascii | |
| Queue_Description | Unicode | ″ExampleQM″ ″A new description″ |
| Queue_FileDesc | Unicode | ″D:¥queues″ |
| **応答メッセージ: Admin_Errors フィールド** | | |
| Queue_FileDesc | ascii | 値 ″Code=4;com.ibm.mqe.MQeException: wrong field type″ |

更新や設定が正常に行われたフィールドについては、Admin_Errors フィールドに項目は存在しません。

各エラーに対しては、エラーについての詳細な記述を含む ascii ストリングが返されます。値は、設定か更新が試行された際に発生した例外です。例外が MQeException であった場合、実際の例外コードは、例外の toString 表示と共に返されます。したがって、MQeException の場合の値の形式は次のようになります。

```
"Code=nnnn;toString representation of the exception"
```

次のコード・フラグは、管理要求の結果を調べ、すべてのエラーを System.out に送る方法を示しています。

```
/**
 * Check to see if a good reply was received.
 * If not detail the error(s) that occurred
 * @return boolean  true if good
 * @param replyMsg reply message to check
 * Throws an Exception if the request failed.
 */
public boolean checkReply( MQeAdminMsg replyMsg ) throws Exception
{
  // Was a reply received ?
  if (replyMsg == null)
  {
    System.out.println("..No response received to the request");
    throw new Exception("No response message received");
  }
  // If the reply was not successful output details for failure
  if ( replyMsg.getRC() != MQeAdminMsg.RC_Success)
  {
    System.out.println("..Action Failed: "+replyMsg.getReason());

    // If mixed then detail each error that occurred
    if ( replyMsg.getRC() == MQeAdminMsg.RC_Mixed)
```

```
    {
      MQeFields errors = replyMsg.getErrorFields();
      Enumeration en = errors.fields();
      // process each error
      while( en.hasMoreElements() )
      {
        String value[];
        String name = (String)en.nextElement();
        // Field in error may be an array
        if ( errors.dataType( name ) == MQeField.TypeArrayElements )
          value = errors.getAsciiArray( name );
        else
          value = new String[] { errors.getAscii( name ) };
        for (int j=0; j<value.length; j++)
          System.out.println("Field in error: "+name+"  "+value[j]);
      }
    }
    // Request failed so throw exception
    throw new MQeException(replyMsg.getReason());
  }
  return true;    // All is OK
}
```

## 管理対象リソースの管理

前のセクションで扱ったように、 MQSeries Everyplace には、管理メッセージで管理できる一連のリソースがあります。これらのリソースは、*管理対象リソース* として知られています。続くセクションでは、これらのリソースのいくつかを管理する方法について扱います。各リソースのアプリケーション・プログラミング・インターフェースに関する詳細は、 *MQSeries Everyplace プログラミング・リファレンス* を参照してください。

### キュー・マネージャー

管理対象リソースのライフ・サイクルの完全な管理は、管理メッセージによって制御することができます。これはつまり、管理対象リソースは、管理メッセージによって存在するようになり、管理され、削除されることを意味します。ただし、キュー・マネージャーはこれに該当しません。キュー・マネージャーを管理するためには、キュー・マネージャーがすでに作成され、開始されている必要があります。キュー・マネージャーの作成と開始については、 36ページの『キュー・マネージャーの作成および削除』を参照してください。

キュー・マネージャーそのものにはほとんど特性がありませんが、キュー・マネージャーは他の MQSeries Everyplace リソースを制御します。キュー・マネージャー上で照会を実行すると、他のキュー・マネージャーへの接続のリストや、そのキュー・マネージャーが作業できるキューのリストを確認することができます。それぞれのリスト項目には、接続またはキューの名前が示されます。リソースの名前が確認できたなら、関係する管理メッセージを使用してリソースを管理することができます。たとえば、MQeConnectionAdmin を使用して接続を管理することができます。

### 接続

接続では、特定のキュー・マネージャーが別のキュー・マネージャー (あるいは、宛先キュー・マネージャーともいう) にどのように接続するかを定義します。接続を定義すると、キュー・マネージャーからリモート・キュー・マネージャーのキュ

ーにメッセージを書き込むことができるようになります。次の図は、別のキュー・マネージャーにあるキューと通信する上で、1 つのキュー・マネージャーのリモート・キューに必要なコンポーネントを示しています。

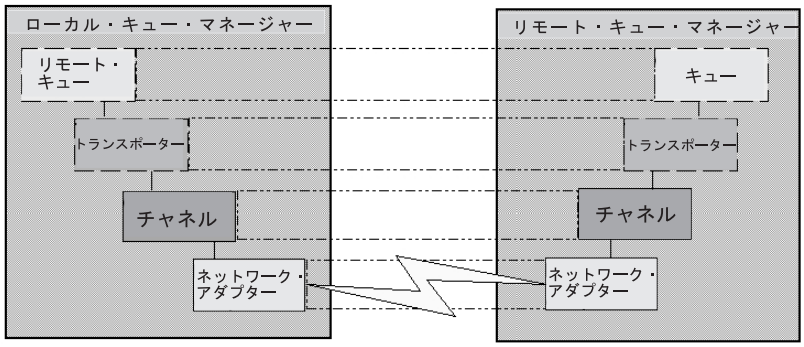


*図 14. キュー・マネージャー接続*

通信は、様々なレベルで行われます。

**トランスポーター:**
2 つのキューの間の論理接続

**チャネル:**
2 つのシステムの間の論理接続

**アダプター:**
プロトコル固有の接続

チャネルとアダプターは、接続の定義を作成する際に指定します。トランスポーターは、リモート・キューの定義を作成する際に指定します。次のサンプル・コードは、接続を確立できる状態の MQeConnectionAdminMsg をインスタンス化および準備するメソッドを示しています。

```
/**
 * Setup an admin msg to create a connection definition
 */
public MQeConnectionAdminMsg addConnection( remoteQMgr
  adapter,
               parms,
               options,
               channel,
               desc ) throws Exception
{
  String remoteQMgr  = "ServerQM";
  /*
   * Create an empty queue manager admin message and parameters field
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue manager to add routes to
```

```
   */
  msg.setName( remoteQMgr );

  /*
   * Set the admin action to create a new queue
   * The connection is setup to use a default channel. This is an alias
   * which must have be setup on the queue manager for the connection to
   * work.
   */
  msg.create( adapter,
              parms,
              options,
              channel,
              desc );
  return msg;
}
```

MQSeries Everyplace では、チャネルとアダプターのいずれについても、標準で 1 つ以上の項目を選択できるようになっています。選択の内容に従って、キュー・マネージャーはいくつかの方法で接続することができます。

- クライアント / サーバー
- 対等通信

## クライアント / サーバー

クライアント / サーバー構成では、1 つのキュー・マネージャーがクライアントとして機能し、別のキュー・マネージャーがサーバー環境として稼働します。サーバーでは、複数の着信接続 (チャネル) を同時に受けることができます。このため、サーバーには複数の着信要求を処理できるコンポーネントが必要です。サーバーでキュー・マネージャーを稼働させる方法については、 47ページの『サーバー』を参照してください。

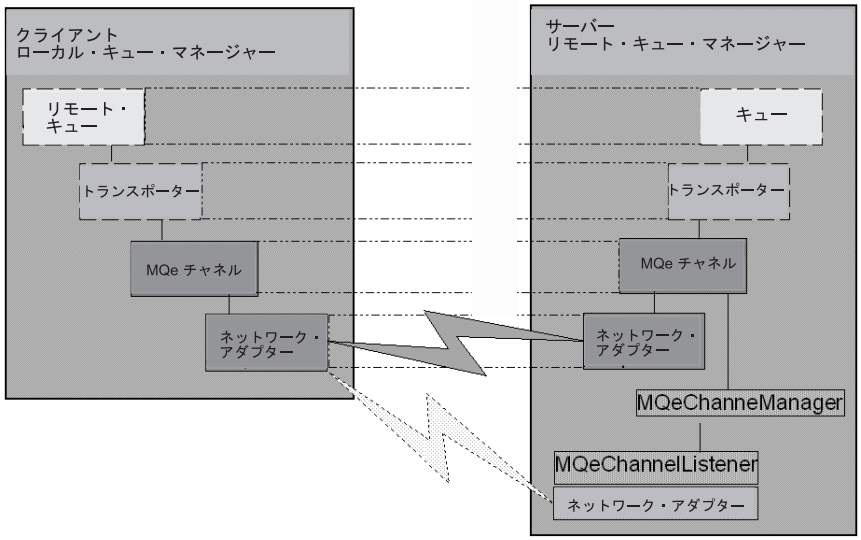


*図 15. クライアント / サーバー接続*

図15 は、クライアント / サーバー構成のコンポーネントに関連した一般的な接続を示したものです。

クライアント部分の接続を構成するには、MQeConnectionAdminMsg を使用します。チャネルのタイプは com.ibm.mqe.MQeChannel です。通常、DefaultChannel の別名は MQeChannel に構成されます。次のコード・フラグは、HTTP プロトコルを使用してサーバーと通信できるクライアントで接続を構成する方法を示しています。

```
/**
 * Create a connection admin message which will create a connection
 * definition to a remote queue manager using the HTTP protocol. Then
 * send the message to the client queue manager.
 */
public addClientConnection( MQeQueueManager myQM,
  String targetQMgr ) throws Exception
{
  String remoteQMgr  = "ServerQM";
  String adapter     = "Network:127.0.0.1:80";
// This assumes that an alias called Network has been setup for
// network adapter com.ibm.mqe.adapters.MQeTcpipHttpAdapter
  String parameters  = null;
  String options     = null;
  String channel     = "DefaultChannel";
  String description = "client connection to ServerQM";

  /*
   * Setup the admin msg
   */
```

```
  MQeConnectionAdminMsg msg = addConnection( remoteQMgr,
                                             adapter,
                                             parameters,
                                             options,
                                             channel,
                                             desc );
  /*
   *  Put the admin message to the admin queue (not using assured flows)
   */
  myQM.putMessage(targetQMgr,
  MQe.Admin_Queue_Name,
  msg,
  null,
  0 );

}
```

## 対等通信

対等通信構成において、キュー・マネージャーは、他の多くの対等機能に対して同時に発信を行うことができるが、どんな場合にも、1 つの対等機能からしか着信を受けることができない対等機能として稼働します。対等機能の中から 1 つがマスターまたは起動側として構成され、それ以外はスレーブまたは受信側として構成されます。

マスターは、クライアント接続の定義とほとんど同じ方法で構成されます。クライアント接続と唯一異なるのは、使用されるチャネルのタイプです。チャネル・タイプは、 com.ibm.mqe.adapters.MQePeerChannel (または別名) に設定されなければなりません。

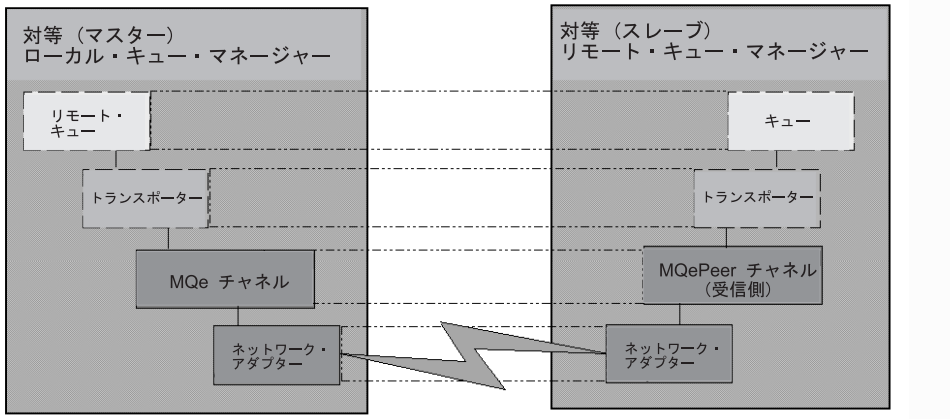


*図 16. 対等通信接続*

スレーブまたは受信側も同様の方法で構成されますが、以下の点が異なっています。

- 接続定義の名前が、接続を定義するキュー・マネージャーの名前と一致していなければならない
- チャネル・タイプが、com.ibm.mqe.adapters.MQePeerChannel でなければならない

- アダプターをリスナーとして構成しなければならない

次のコード・フラグでは、HTTP プロトコルを使用してポート 8081 で聴取を行い、″PeerQM1″ というキュー・マネージャーを対等通信の受信側として構成します。

```
/**
 * Create a connection admin message which will create a connection
 * definition to a remote queue manager using the HTTP protocol. Then
 * send the message to the client queue manager.
 */
public addClientConnection( MQeQueueManager myQM,
  String targetQMgr ) throws Exception
{
  String remoteQMgr  = "PeerQM1";
// To be a receiver the connection definition called "PeerQM1" must
// be configured on queue manager "PeerQM1"
  String adapter     = "Network::8081";
// This assumes that an alias called Network has been setup for
// network adapter com.ibm.mqe.adapters.MQeTcpipHttpAdapter
  String parameters  = null;
  String options     = null;
  String channel     = "com.ibm.mqe.adapters.MQePeerChannel";
  String description = "peer receiver on PeerQM";

  /*
   * Setup the admin msg
   */
  MQeConnectionAdminMsg msg = addConnection( remoteQMgr,
  adapter,
               parameters,
               options,
               channel,
               desc );

  /*
   *  Put the admin message to the admin queue (not using assured flows)
   */
  myQM.putMessage(targetQMgr,
  MQe.Admin_Queue_Name,
  msg,
  null,
  0 );
}
```

次の表は、″PeerQM1″ の受信側と、これに接続しようとする他のすべての対等キュー・マネージャーに使用される、接続定義パラメーターを示したものです。

| | マスター (起動側) | スレーブ (受信側) |
|---|---|---|
| キュー・マネージャー上の名前 | 任意 | ″PeerQM1″ |
| 接続名 | ″PeerQM1″ | ″PeerQM1″ |
| チャネル | com.ibm.mqe.MQePeerChannel | com.ibm.mqe.MQePeer |
| アダプター | Network:192.168.0.10:8081 | Channel Network::8081 |
| パラメーター | | |
| オプション | | |

## アダプター

MQSeries Everyplace で提供されている様々なアダプターについての詳細は、227ページの『第9章 MQSeries Everyplace アダプター』を参照してください。

接続は、キュー・マネージャーが中間キュー・マネージャーを経由してメッセージを送れるようにセットアップすることができます。これには、2 つの接続、すなわちこのセクションの前の部分で説明した中間キュー・マネージャーに対する接続と、宛先キュー・マネージャーに対する接続を確立する必要があります。この場合は、ネットワーク・アダプターを指定するのではなく、中間キュー・マネージャーの名前を指定します。このような構成において、アプリケーションは宛先キュー・マネージャーにメッセージを書き込みますが、その際、メッセージは 1 つ以上の中間キュー・マネージャーを介して宛先に送られます。

## 別名

接続には、複数の名前、つまり別名を割り当てることができます。アプリケーションで MQeQueueManager クラスのメソッドを呼び出し、キュー・マネージャー名の指定が要求された際には、別名を使用することもできます。この機能を使用すれば、アプリケーション・プログラムと実際のキュー・マネージャー名との間を間接的につなぐ、1 つの段階を設けることができます。たとえば、'QM12A26'1 というキュー・マネージャーに、'PAYROLLQM' という別名があるとします。アプリケーション・プログラムは、'PAYROLLQM' に対してメッセージの書き込み要求を実行します。するとこれは、自動的に 'QM12A345' に対するメッセージの書き込み要求に変換されます。別名から実際の名前への変換は、要求を出すキュー・マネージャーで行われます。

別名は、ローカル・キュー・マネージャーでも、リモート・キュー・マネージャーでも使用することができます。ローカル・キュー・マネージャーに別名を付けるためには、ローカル・キュー・マネージャーと同じ名前の接続定義が確立されている必要があります。これは、すべてのパラメーターをヌルに設定できる論理接続です。

別名の追加と除去は、MQeConnectionAdminMsg クラスの Action_AddAlias アクションと Action_RemoveAlias アクションを使用して、容易に行うことができます。1 つのメッセージの中で複数の別名を追加または除去することができます。追加または除去する別名は、ascii array タイプの Con_Aliases フィールドを設定して直接メッセージに書き込むこともできますし、 addAlias() メソッドや removeAlias() メソッドを使用することもできます。各メソッドはそれぞれ 1 つの別名しか扱えませんが、メソッドを繰り返し呼び出して 1 つのメッセージに複数の別名を追加できます。次のコードの抜粋は、メッセージに接続の別名を追加する方法を示すものです。

```
/**
 * Setup an admin msg to add aliases to a queue manager (connection)
 */
public MQeConnectionAdminMsg addAliases(  String queueManagerName
                                          String aliases[] ) throws Exception
{
  /*
   * Create an empty connection admin message
   */
  MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg();

  /*
```

```
     * Prime message with who to reply to and a unique identifier
     */
    MQeFields msgTest = primeAdminMsg( msg );

    /*
     * Set name of the connection to add aliases to
     */
    msg.setName( queueManagerName );

    /*
     * Use the addAlias method to add aliases to the message.
     */
    for ( int i=0; i<aliases.length; i++ )
    {
      msg.addAlias( aliases[i] );
    }

    return msg;
}
```

## キュー

MQSeries Everyplace には、いくつかのタイプのキューがあります。その中で最も単純なのがローカル・キューで、このキューは、MQeQueue クラスに実装され、MQeQueueAdminMsg の Inherit クラスによって管理されます。他のすべてのタイプのキューは、MQeQueue を継承します。各タイプのキューには、それに対応して、MQeQueueAdminMsg を継承する管理メッセージが存在します。次のセクションでは、様々なタイプのキューについて説明します。

### ローカル・キュー

キューの中で最も単純なタイプは、ローカル・キューです。ローカル・キューは、すべてのメッセージの最終的な宛先となる、実際のキューです。このタイプのキューは、特定のキュー・マネージャーに対してローカルに存在し、そのキュー・マネージャーに属します。キュー・マネージャー上のアプリケーションは、そのキュー・マネージャーのキューと直接対話し、確実に (ハードウェアの障害やデバイスの損失が生じた場合を除く)、そしてセキュリティー面でも安全にメッセージを保管することができます。これらのキューは、オンラインでもオフラインでも、つまり、ネットワークに接続してもネットワークに接続しなくても使用することができます。ローカル・キューは、ローカル・キュー・マネージャーからのアクセスを受けるとき、必ず同期モードで動作します。

ローカル・キューとそれに属するキューでは、通常の管理アクションを使用して作成、更新、削除、および照会を行うことができます。ローカル・キューで使用される基本的な管理のメカニズムは、MQeAdminMsg から継承されます。

キューの名前は、宛先キュー・マネージャーの名前 (ローカル・キューの場合は、そのキューが属しているキュー・マネージャーの名前) からとられ、そのキュー・マネージャーのキューに固有な名前が用いられます。キューは、管理メッセージ内の 2 つのフィールド、すなわち Admin_Name フィールドと Queue_QMgrName フィールドによって一意的に識別されます。これらのフィールドはいずれも ascii タイプのフィールドです。これら管理メッセージ内の 2 つのフィールドは、メソッド setName( queueManagerName, queueName) を使用して設定することができます。

次の図は、ローカル・キューが含まれているキュー・マネージャーの例を示しています。キュー・マネージャー 'qm1' には、'invQ' というキューがあります。キュー

のキュー・マネージャー名特性が ’qm1’ であり、キュー・マネージャーの名前と一致しているのが分かります。

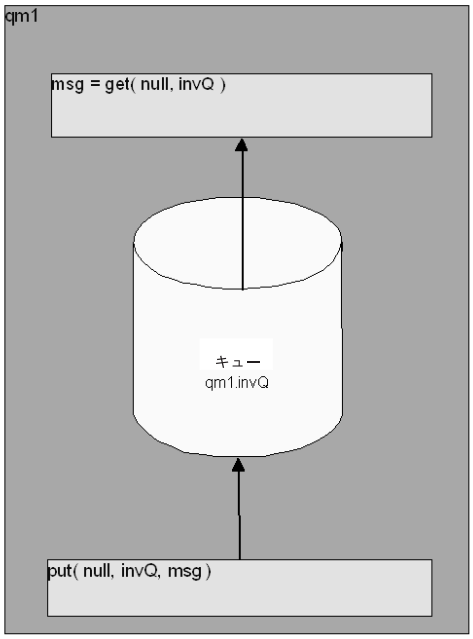


*図 17. ローカル・キュー*

***メッセージ・ストア:*** ローカル・キューでメッセージを保管するには、メッセージ・ストアが必要です。メッセージ・ストアは、キューを基準としてキューの上で変更することができます。使用するメッセージ・ストアのタイプは、キューの特性 `Queue_FileDesc` とそのパラメーターによって指定することができます。このフィールドは `ascii` タイプのフィールドで、値は、次の形式のファイル記述子でなければなりません。

adapter class:adapter parameters
または
adapter alias:adapter parameters

たとえば、次のように入力します。

```
MsgLog:d:¥QueueManager¥ServerQM12¥Queues
```

MQSeries Everyplace バージョン 1 では、2 つのアダプター、すなわちディスクへの書き込みを行うためのアダプターと、ストレージのメモリーを使用するためのアダプターが提供されています。適切なアダプターを作成することにより、適当であればどんな場所あるいはメディア (DB2 データベースや CD-R など) にでも、メッセージを保管できるようになります。

メッセージの永続性や回復力は、どのアダプターを選択するかによって決まります。たとえば、メモリー・アダプターを使用する場合、メッセージの回復力はメモリーのそれと同じだけになります。メモリーは、ディスクよりもかなり高速なメデ

ィアですが、ディスクに比べて非常に不安定なメディアでもあります。それで、アダプターの選択が重要であるということが分かります。

キューの作成時にメッセージ・ストアが作成されない場合は、キュー・マネージャーを作成する際に指定されたデフォルトのメッセージ・ストアが使用されます。詳細については、33ページの『第4章 MQSeries Everyplace キュー・マネージャー』を参照してください。

Queue_FileDesc フィールドを設定する際には、以下の点を考慮に入れる必要があります。

- 使用している構文が、キューが置かれているシステムに対して適切なものであることを確認してください。たとえば、Windows システムではファイル区切りに ″¥″ を使用していますが、 UNIX® システムでは ″/″ を使用します。ある場合には、このどちらを使用しても構わないこともありますが、それは、キュー・マネージャーが稼働する JVM (Java 仮想マシン) のサポートによって異なります。また、ファイル区切りの違いに加え、Windows NT のような一部のシステムでは、UNIX などの他のシステムでは使用されないドライブ名を使用している場合があります。
- 一部のシステムでは、他のシステムでは使用できない、相対的なディレクトリー (例、″.\″) を指定できる場合があります。相対的なディレクトリーを指定できるシステムにおいても、相対的なディレクトリーを指定する場合は厳重な注意が必要です。現行ディレクトリーは JVM の存続時間の間に変更される可能性があり、それによって、相対的なディレクトリーを使用するキューとの対話に問題を引き起こす恐れがあります。

***ローカル・キューの作成:*** 次のコード・フラグは、ローカル・キューをどのように作成するかを示したしたものです。

```
/**
 * Create a new local queue
 */
protected void createQueue(MQeQueueManager localQM,
                           String          qMgrName,
                           String          queueName,
       String          description,
                           String          queueStore
  ) throws Exception
{
  /*
   * Create an empty queue admin message and parameters field
   */
  MQeQueueAdminMsg msg = new MQeQueueAdminMsg();
  MQeFields parms = new MQeFields();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue to manage
   */
  msg.setName( qMgrName, queueName );

  /*
   * Add any characteristics of queue here, otherwise
   * charateristics will be left to default values.
   /
```

```
  if ( description != null ) // set the description ?
    parms.putUnicode( MQeQueueAdminMsg.Queue_Description,
                      description);
  if ( queueStore != null )  // Set the queue store ?
    // If queue store includes directory and file info then it
    // must be set to the correct style for the system that the
    // queue will reside on e.g ¥ or /
    parms.putAscii( MQeQueueAdminMsg.Queue_FileDesc,
                    queueStore );
  /*
   * Other queue characteristics like queue depth, message expiry
   * can be set here ...
   */

  /*
   * Set the admin action to create a new queue
   */
  msg.create( parms );

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   */
  localQM.putMessage( qMgrName,
                      MQe.Admin_Queue_Name,
                      msg,
                      null,
                      0);
}
```

***キューのセキュリティー:*** アクセスとセキュリティーはキューに属しており、リモート・キュー・マネージャーでの使用に関しては (ネットワークに接続する場合)、これら他のキュー・マネージャーがそのキューとメッセージをやり取りすることを許可する権限を設けることができます。キューのセキュリティーのセットアップでは、次の特性が使用されます。

- Queue_Cryptor
- Queue_Authenticator
- Queue_Compressor
- Queue_TargetRegistry
- Queue_AttrRule

セキュリティーに基づくキューのセットアップについての詳細は、175ページの『第7章 セキュリティー』を参照してください。

***その他のキューの特性:*** キューを構成する際には、キューが受け取ることのできるメッセージの最大数などを含め、他の多くの特性を使用することができます。これらについては、 *MQSeries Everyplace プログラミング・リファレンス* の *MQeQueueAdminMsg* のセクションを参照してください。

***別名:*** キューの名前には、 107ページの『別名』で扱われた接続の別名と同様の方法で別名を定義することができます。接続のセクションにある別名の例のコード・フラグでは、接続に対して別名をセットアップする方法が示されています。キューに対して別名をセットアップする場合には、 MQeConnectionAdminMsg の代わりに MQeQueueAdminMsg を使用する点を除き、これと同じ方法を用いることができます。

***アクションの制限:*** ある特定の管理アクションは、キューが適切な状況にない場合には実行することができません。次のアクションがそうです。

**Action_Update**

- キューが使用されているときは、そのキューの特性を変更できません
- キューのセキュリティー特性は、キューが使用されている場合、つまりキューにメッセージが含まれている場合には変更することができません
- キューのメッセージ・ストアは、1 度設定されたなら変更できません

**Action_Delete**

キューは、使用されている場合、つまりそのキューにメッセージが含まれている場合には変更できません

キューが使用されていない、つまりメッセージがまったく含まれていない状況になければ実行できない要求は、キュー・マネージャーが再始動される際か一定の時間間隔の後に再試行されます。管理要求の再試行をセットアップすることについての詳細は、92ページの『基本となる管理要求メッセージ』を参照してください。

## リモート・キュー

このタイプのキューは、その名前が示すとおり、ローカル環境にありません。ローカルには、キューが属するキュー・マネージャーと実際のキューを識別する定義が保持されます。リモート・キューは、同期的にも非同期的にもアクセスできます。リモート・キューの定義がローカルに保持されている場合、アクセスのモードはこの定義に基づいて同期または非同期になります。ローカル環境に定義が保持されていない場合は、キュー・ディスカバリー (115ページの『ディスカバリー』を参照) が行われます。これによってキューの特性が特定され、アクセスのモードは強制的に同期になります。

リモート・キューは MQeRemoteQueue クラスによって実装され、MQeAdminMsg のサブクラスである MQeRemoteQueueAdminMsg クラスによって管理されます。

キューの名前は、宛先キュー・マネージャーの名前 (リモート・キューの場合、これはそのキューがローカルに存在するキュー・マネージャーの名前) と、そのキュー・マネージャーでのキューの実名からとられます。キューは、管理メッセージ内の 2 つのフィールド、すなわち `Admin_Name` フィールドと `Queue_QMgrName` フィールドによって一意的に識別されます。これらのフィールドはいずれも `ascii` タイプのフィールドです。これら管理メッセージ内の 2 つのフィールドは、メソッド `setName( queueManagerName, queueName)` を使用して設定することができます。リモート・キューの定義では、そのキューのキュー・マネージャー名と、定義が置かれているキュー・マネージャーの名前とが一致することはありません。

キューのリモート定義は、ほとんどの場合、実際のキューの定義と一致します。そうでない場合、そのキューと対話する際には異なった結果が得られます。たとえば、次のような場合があります。

- 非同期キューでは、リモート定義の `max message size` が実際のキューの同じ定義より大きい場合に、定義はリモート・キューのストレージで受け入れられますが、実際のキューに移動されると拒否されます。なおメッセージは、失われずにリモート・キューに残されます。ただし、これを送達することはできません。
- 同期キューでは、セキュリティー特性が一致しない場合に、MQSeries Everyplace は、実際のキューと折衝してどのセキュリティー特性を使用するべきかを決定します。メッセージの書き込みは正常に行われる場合もありますが、それ以外の場合には、属性の不一致による例外が返されます。

***同期キュー:*** 同期キューは、所有するキュー・マネージャーへのパスを持つネットワークに接続されている場合にのみアクセスされます。接続が確立できない場合は、取得、書き込み、ブラウズなどの操作によって例外が発生します。所有するキューは、キューにアクセスするために必要なアクセス許可、およびセキュリティー要件を制御します。メッセージの送受信時にエラーや再試行を扱うのはアプリケーションの責任であり、 MQSeries Everyplace には、1 回で確実にメッセージを送達する責任はありません。

同期的な操作のためにキューを設定するには、`Queue_Mode` フィールドを `Queue_Synchronous` に設定します。

***非同期キュー:*** 非同期キューでは、メッセージを書き込むことはできても、それを検索することはできません。ネットワーク接続が確立されると、メッセージはキューが属しているキュー・マネージャーとキューに送信されます。しかし、ネットワークが接続されていない場合には、メッセージはネットワーク接続が確立されるまでローカルに保管され、その後、伝送されます。これによりアプリケーションは、クライアントがオフラインのときでもキューで動作できるようになります。ただし、そのためには、一時的にメッセージを保管するメッセージ・ストアが非同期キューに必要です。メッセージ・ストアの定義は、ローカル・キューの場合と同じです。

非同期的な操作のためにキューを設定するには、 `Queue_Mode` フィールドを `Queue_Asynchronous` に設定します。

図18 は、同期的な操作のためのリモート・キューのセットアップと、非同期的な操作のためののりモード・キューのセットアップを示したものです。

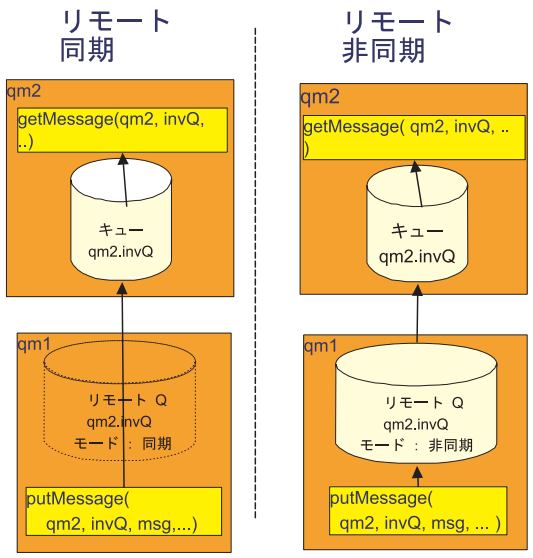


*図 18. リモート・キュー*

- 同期、非同期のいずれの例でも、キュー・マネージャー qm2 には qm2 というローカル・キューが存在します。
- 同期の例では、キュー・マネージャー qm1 上に、キュー・マネージャー qm2 に置かれているキュー invQ のリモート・キュー定義があり、同期的な操作のために設定されています。キュー・マネージャー qm1 を使用するアプリケーションと、キュー qm2.invQ への書き込みメッセージによって、キュー・マネージャー qm2 に対するネットワーク接続が確立され (まだ接続が存在していない場合)、すぐにメッセージが実際のキューに書き込まれます。ネットワーク接続を確立できない場合、アプリケーションは例外を受け取り、これを処理しなければなりません。
- 非同期の例では、キュー・マネージャー qm1 上に、キュー・マネージャー qm2 に置かれているキュー invQ のリモート・キュー定義があり、非同期的な操作のために設定されています。キュー・マネージャー qm1 を使用するアプリケーションと、キュー qm2.invQ への書き込みメッセージでは、qm1 のリモート・キューに一時的にメッセージを保管します。伝送の基準にかなうなら、メッセージはキュー・マネージャー qm2 上の実際のキューに移動されます。メッセージは、伝送が正常に行われるまでリモート・キューに保管されます。

***リモート・キューの作成:*** 次のコード・フラグは、リモート・キューを作成するための管理メッセージをどのようにセットアップするかを示したものです。

```
/**
 * Create a remote queue
 */
protected void createQueue(MQeQueueManager localQM,
                           String          targetQMgr,
                           String          qMgrName,
                           String          queueName,
       String          description,
                           String          queueStore,
       byte            queueMode
  ) throws Exception
{
  /*
   * Create an empty queue admin message and parameters field
   */
  MQeRemoteQueueAdminMsg msg = new MQeRemoteQueueAdminMsg();
  MQeFields parms = new MQeFields();

  /*
   * Prime message with who to reply to and a unique identifier
   */
  MQeFields msgTest = primeAdminMsg( msg );

  /*
   * Set name of queue to manage
   */
  msg.setName( qMgrName, queueName );

  /*
   * Add any characteristics of queue here, otherwise
   * charateristics will be left to default values.
   /
  if ( description != null ) // set the description ?
    parms.putUnicode( MQeQueueAdminMsg.Queue_Description,
                      description);

  // set the queue access mode if mode is valid
  if ( queueStore != MQeQueueAdminMsg.Queue_Asynchronous &&
       queueStore != MQeQueueAdminMsg.Queue_Synchronous )
```

```
    throw new Exception ("Invalid queue store");

  parms.putByte( MQeQueueAdminMsg.Queue_Mode,
                 queueMode);

  if ( queueStore != null )  // Set the queue store ?
    // If queue store includes directory and file info then it
    // must be set to the correct style for the system that the
    // queue will reside on e.g ¥ or /
    parms.putAscii( MQeQueueAdminMsg.Queue_FileDesc,
                    queueStore );
  /*
   * Other queue characteristics like queue depth, message expiry
   * can be set here ...
   */

  /*
   * Set the admin action to create a new queue
   */
  msg.create( parms );

  /*
   *  Put the admin message to the admin queue (not assured delivery)
   *  on the target queue manager
   */
  localQM.putMessage( targetQMgr,
                      MQe.Admin_Queue_Name,
                      msg,
                      null,
                      0);
}
```

***ディスカバリー:*** アプリケーションがリモート・キューにメッセージを書き込む際、リモート・キューの定義がローカルに保持されていると、キューの特性はそのリモート・キュー定義を使って判別されます。定義がローカルに保持されていない場合は、キュー・ディスカバリー が行われます。このローカル・キュー・マネージャーは、同期的にリモート・キュー・マネージャーと交信し、キューの特性の確認を試みます。これによって次のような特性が検出されます。

- Queue_Description
- Queue_Expiry
- Queue_MaxQSize
- Queue_MaxMsgSize
- Queue_Priority
- Queue_Cryptor
- Queue_Authenticator
- Queue_Compressor
- Queue_TargetRegistry
- Queue_AttrRule

キューのディスカバリーが正常に行われた後、このキューの定義は、ディスカバリーが開始されたキュー・マネージャーにリモート・キュー定義として保管されます。このようにして検出されたキューの定義は、通常のリモート・キュー定義と同じように扱われます。なお、Queue_Mode は、検出されたキューがすべて同期操作のために設定されている場合には検出されません。

## ストア・アンド・フォワード (蓄積交換) キュー

このタイプのキューは、その名前が示すとおり、次の (宛先である必要はない) キュー・マネージャーに転送できるようになるまでメッセージを保管します。通常、このタイプのキューはサーバー上で定義され、次の 2 とおりに構成することができます。

- 次のキュー・マネージャーにメッセージを転送する。次のキュー・マネージャーは宛先キュー・マネージャーでなくても構いません。このようなキューの構成では、メッセージがネクスト・ホップにプッシュされます。
- 宛先キュー・マネージャーがストア・アンド・フォワード (蓄積交換) キューからメッセージを収集できるようになるまでメッセージを保持する。このような構成は、ホーム・サーバー ・キューを使用することによって可能になります (119ページの『ホーム・サーバー・キュー』を参照してください)。このような構成を使用する場合、メッセージはストア・アンド・フォワード (蓄積交換) キューからプルされます。

ストア・アンド・フォワード (蓄積交換) キューは、多数の宛先キュー・マネージャーへのメッセージを保持する場合もありますし、宛先キュー・マネージャーごとに 1 つのストア・アンド・フォワード (蓄積交換) キューが設けられる場合もあります。

ストア・アンド・フォワード (蓄積交換) キューは、MQeStoreAndForwardQueue クラスによって実装され、MQeRemoteQueueAdminMsg のサブクラスである MQeStoreAndForwardQueueAdminMsg クラスによって管理されます。主な追加の管理機能としては、ストア・アンド・フォワード (蓄積交換) キューがメッセージを保持できるキュー・マネージャーの名前を追加および除去することができます。これは、`Action_AddQueueManager` アクションと `Action_RemoveQueueManager` アクションによって行われます。 1 つのメッセージで複数のキュー・マネージャー名を追加または除去することができます。名前は、`ascii array` タイプの `Queue_QMgrNameList` フィールドを設定して直接メッセージに書き込むこともできますし、 `addQueueManager()` メソッドや `removeQueueManager()` メソッドを使用することもできます。各メソッドはそれぞれ 1 つのキュー・マネージャー名しか扱えませんが、メソッドを繰り返し呼び出して 1 つのメッセージに複数のキュー・マネージャー名を追加することができます。

次のコード・フラグは、どのようにしてメッセージに宛先キュー・マネージャーの名前を追加するかを示しています。

```
/**
 * Setup an admin msg to add target queue managers to
 * a store and forward queue.
 */
public MQeStoreAndForwardQueueAdminMsg addQueueManager( String queueName
                                                         String queueManagerName
                                                         String qMgrNames[] )
                                                         throws Exception
{
  /*
   * Create an empty admin message
   */
  MQeStoreAndForwardQueueAdminMsg msg =
  new MQeStoreAndForwardQueueAdminMsg();

  /*
   * Prime message with who to reply to and a unique identifier
```

```
 */
MQeFields msgTest = primeAdminMsg( msg );

/*
 * Set name of the store and forward queue
 */
msg.setName( queueManagerName, queueName );

/*
 * Use the addAlias method to add aliases to the message.
 */
for ( int i=0; i<qMgrNames.length; i++ )
{
  msg.addQueueManager(qMgrNames[i] );
}

return msg;
}
```

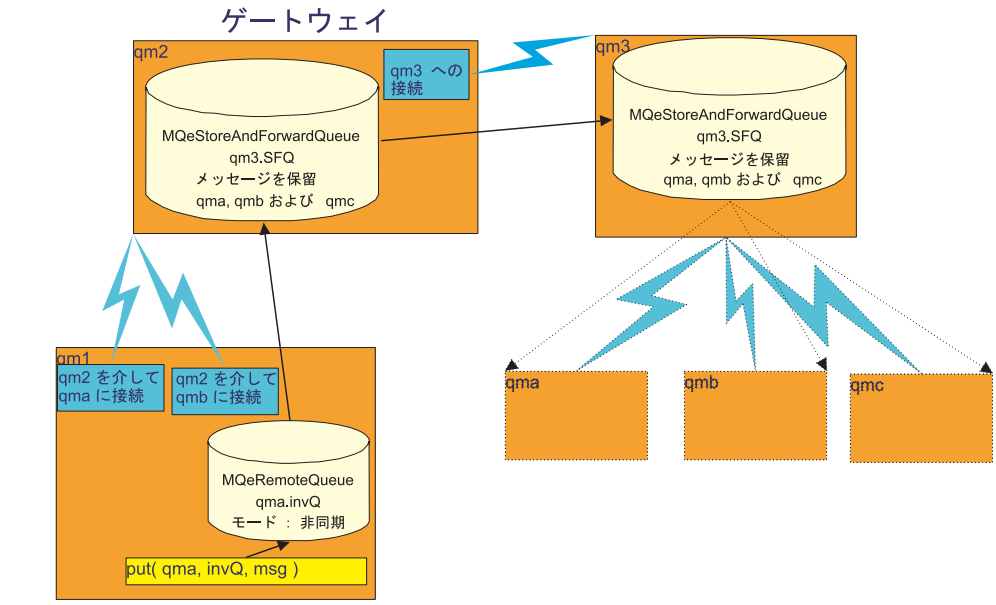


*図 19. ストア・アンド・フォワード (蓄積交換) キュー*

次のキュー・マネージャーにメッセージを転送するようストア・アンド・フォワード (蓄積交換) キューを構成する場合、ストア・アンド・フォワード (蓄積交換) キューのキュー・マネージャー名属性は、次のキュー・マネージャーの名前でなければなりません。また、次のキュー・マネージャーへの接続を構成する必要があります。

また、メッセージの収集 (プル) を待つようにストア・アンド・フォワード (蓄積交換) キューを構成する場合には、ストア・アンド・フォワード (蓄積交換) キューのキュー・マネージャー名属性は意味を持ちません。そのため、これはストア・アンド・フォワード (蓄積交換) キューが置かれているキュー・マネージャーの名前でも構いません。ただし、キューの接続先に、キューのキュー・マネージャー名属性と

同じ名前のキュー・マネージャーを含めることはできません。そのような接続先が存在すると、キューはその接続を使用してメッセージを転送しようとします。

117ページの図19 は、別々のキュー・マネージャーにある 2 つのストア・アンド・フォワード (蓄積交換) キューの例を示しています。 1 つは次のキュー・マネージャーにメッセージをプッシュするようセットアップされたキューで、他方はメッセージが収集されるのを待つようにセットアップされています。

- キュー・マネージャー qm2 では、キュー・マネージャー qm3 に対する接続が構成されています。
- キュー・マネージャー qm2 では、接続 qm3 を使用してキュー・マネージャー qm3 にメッセージをプッシュするよう構成されたストア・アンド・フォワード (蓄積交換) キューがあります。ストア・アンド・フォワード (蓄積交換) キューのキュー・マネージャー名属性は、接続名と同じ qm3 です。
- ストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ と qm2 は、キュー・マネージャー qma、qmb、および qmc を宛先とするメッセージを扱うように構成されています。
- キュー・マネージャー qm3 には、ストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ があります。キュー名 qm3 のキュー・マネージャー名部分には、これに対応する qm3 という接続がありません。したがってすべてのメッセージは、収集されるまでキューに保管されます。
- qm3 のストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ には、キュー・マネージャー qma、qmb、および qmc に代わってメッセージが保管されます。メッセージは、収集されるか有効期限が切れるまで保管されます。

キュー・マネージャーが、中間キュー・マネージャーにあるストア・アンド・フォワード (蓄積交換) キューを使用して他のキュー・マネージャーにメッセージを送ろうとする場合、メッセージが最初に置かれるキュー・マネージャーには以下のものが必要です。

- 中間キュー・マネージャーに対して構成された接続
- 中間キュー・マネージャーを介し、宛先キュー・マネージャーに対して構成された接続
- 宛先キューのリモート・キュー定義

これらの条件がすべて満たされていれば、アプリケーションは、キュー・マネージャー・ネットワークのレイアウトを理解していなくても、宛先キュー・マネージャーの宛先キューにメッセージを書き込むことができます。これにより、アプリケーション・プログラムを変更する必要なくして、基底にあるキュー・マネージャー・ネットワークを変更することができます。

117ページの図19 の中で、キュー・マネージャー qm1 は、キュー・マネージャー qma のキュー invQ にメッセージを書き込めるよう構成されれています。この構成は、次のものから成っています。

- 中間キュー・マネージャー qm2 への接続
- 宛先キュー・マネージャー qma への接続
- qma 上のリモート非同期キュー invQ

アプリケーションが、キュー・マネージャー qm1 を使用してキュー・マネージャー qma のキュー invQ にメッセージを書き込む場合、メッセージの流れは次のようになります。

1. アプリケーションが非同期キュー qma.invQ にメッセージを書き込みます。メッセージは、伝送の基準に従ってネクスト・ホップに移されるまで、qm1 にローカルに保管されます。
2. 伝送の基準にかなえば、メッセージが移されます。 qma の接続の定義に基づいて、メッセージはキュー・マネージャー qm2 を経由します。
3. キュー・マネージャー qma のキュー invQ に対するメッセージを扱うように構成されているキューは、qm3 のストア・アンド・フォワード (蓄積交換) キュー SFQ だけです。メッセージはこのキューに一時的に保管されます。
4. このストア・アンド・フォワード (蓄積交換) キューには、ネクスト・ホップ、つまりキュー・マネージャー qm3 にメッセージをプッシュするための接続が構成されています。
5. キュー・マネージャー qm3 には、キュー・マネージャー qma を宛先としたメッセージを保持できるストア・アンド・フォワード (蓄積交換) キュー qm3.SFQ があり、メッセージはこのキューに保管されます。
6. qma へのメッセージは、キュー・マネージャー qma によって収集されるまで、ストア・アンド・フォワード (蓄積交換) キューで保管されます。このセットアップを行う方法については、『ホーム・サーバー・キュー』を参照してください。

## ホーム・サーバー・キュー

このタイプのキューは、通常、クライアント上にあり、ホーム・サーバー というサーバー上のストア・アンド・フォワード (蓄積交換) キューに関連付けられています。このキューは、配置先のキュー・マネージャーがアクティブにされたときにホーム・サーバーからメッセージをプルし、オプションで、一定のポーリング間隔に従ってメッセージを確認します。このキューは、サーバーからメッセージをプルすると、確実なメッセージ送達を使用してローカル・キュー・マネージャーにそのメッセージを書き込みます。次いで、そのメッセージは宛先キューに保管されます。

ホーム・サーバー・キューは、MQeHomeServerQueue クラスによって実装され、MQeRemoteQueueAdminMsg のサブクラスである MQeHomeServerQueueAdminMsg クラスで管理されます。このキューが持つ、唯一の追加の特性は、Queue_QTimerInterval フィールドです。このフィールドは int タイプのフィールドで、ここにはミリ秒単位の時間間隔が設定されます。このフィールドに 0 以上の値が設定されると、ホーム・サーバー・キューは、ホーム・サーバー上に収集を待っているキューがないかどうかを、設定されたミリ秒ごとに確認します。待機中のメッセージは、すべて宛先キューに送達されます。

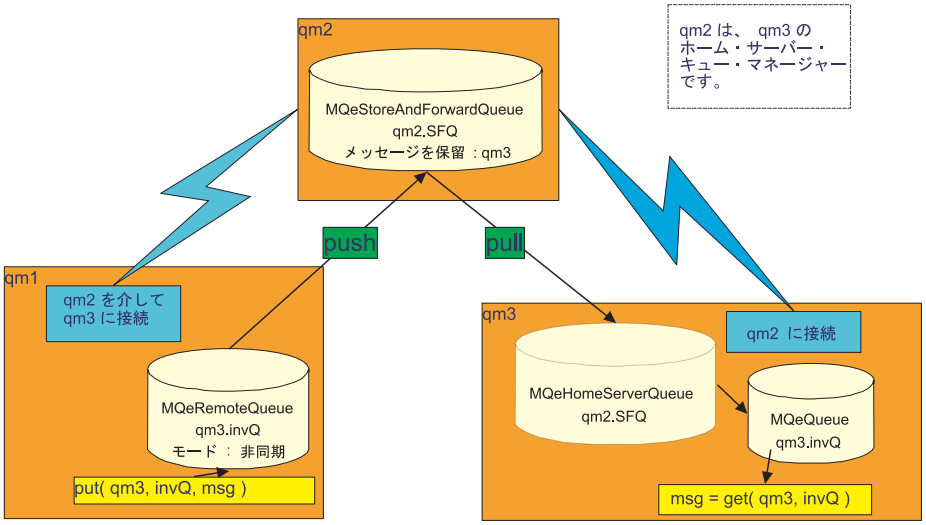


*図 20. ホーム・サーバー・キュー*

ホーム・サーバー・キューの名前は次のように設定します。

- キュー名はストア・アンド・フォワード (蓄積交換) キューの名前と一致していなければならない
- キュー・マネージャー名はホーム・サーバー・キュー・マネージャーの名前と一致していなければならない

ホーム・サーバー・キューがあるキュー・マネージャーには、ホーム・サーバー・キュー・マネージャーに対して構成された接続が必要です。

図20 は、ホーム・サーバー・キュー SFQ を持つキュー・マネージャー qm3 の例を示しています。このキューは、ホーム・サーバー・キュー・マネージャー qm2 からメッセージを収集するように構成されています。

この構成は、次のものから成っています。

- ホーム・サーバー・キュー・マネージャー qm2
- キュー・マネージャー qm3 へのメッセージを保持する、キュー・マネージャー qm2 のストア・アンド・フォワード (蓄積交換) キュー SFQ
- 通常はキュー・マネージャー qm2 から切断された状態で稼働し、またそれから接続を受けることができないキュー・マネージャー qm3
- キュー・マネージャー qm3 から qm2 に対して構成された接続
- ホーム・サーバーとしてキュー・マネージャー qm2 を使用するホーム・サーバー・キュー SFQ

qm2 を通してキュー・マネージャー qm3 に送信されるすべてのメッセージは、qm3 のホーム・サーバー・キューによって収集されるまで、qm2 のストア・アンド・フォワード (蓄積交換) キュー SFQ に保管されます。

## MQSeries ブリッジ・キュー

MQSeries ブリッジ・キューとは、MQSeries キュー・マネージャーにあるキューを参照する、リモート・キュー定義のことをいいます。メッセージを保持するキューは、ローカル・キュー・マネージャーではなく、MQSeries キュー・マネージャーにあります。

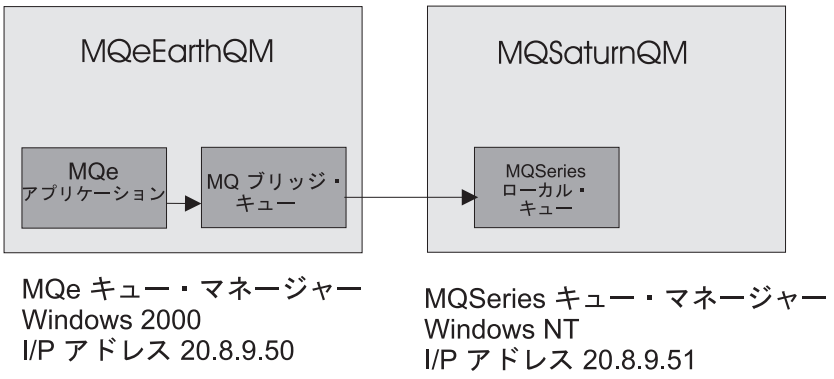


*図 21. MQSeries ブリッジ・キュー*

- MQeSaturnQM MQSeries キュー・マネージャーには、ローカル・キューを定義することができます (MQSaturnQ)。
- MQeEarthQM では、MQSaturnQM キュー・マネージャー上に MQSaturnQ という MQSeries ブリッジ・キューを定義する必要があります。
- MQeEarthQM キュー・マネージャーに付加されたアプリケーションは MQSeries ブリッジ・キューにメッセージを書き込み、ブリッジ・キューは MQSaturnQM キュー・マネージャーの MQSaturnQ にそのメッセージを送達します。

ブリッジ・キューの定義では、ブリッジ・オブジェクト階層でクライアント接続を固有に識別するため、ブリッジ (MQSeries キュー・マネージャー・プロキシー) とクライアント接続の名前を指定する必要があります (132ページの図26 を参照してください)。この情報は、MQSeries ブリッジが MQSeries キューを操作するために MQSeries キュー・マネージャーにアクセスする方法を識別します。

MQSeries ブリッジ・キューには、MQSeries Everyplace システムの MQSeries ブリッジ・キューとは異なる名前を持つ MQSeries キュー・マネージャーのキューに書き込みを行う機能があります (これは、MQSeries リモート・キュー定義の RNAME と同様です)。同様に、MQSeries ブリッジ・キュー定義のキュー・マネージャー名には、MQSeries ネットワークにおいて MQSeries メッセージが書き込まれる宛先キュー・マネージャーの名前を使用します。なおこれは、ブリッジ・キューが接続される MQSeries キュー・マネージャーである必要はありません。これにより、他の MQSeries キュー・マネージャー (MQSeries キュー・マネージャー・プロキシーで名前を付けられたもの) を介して MQSeries キュー・マネージャー (宛先) にメッセージを送信することが可能になります。

MQSeries ブリッジ・キューで使用されるすべての特性のリストは、 *MQSeries Everyplace プログラミング・リファレンス* の *com.ibm.mqe.bridge* のセクションにある、 *MQeMQBridgeQueueAdminMsg* の項を参照してください。次の 122ページの表4 は、MQbridge キューが構成された後、このキューでサポートされる操作をリストしています。

*表 4. MQSeries－ブリッジ・キューでサポートされているメッセージ操作*

| 操作のタイプ | MQSeries－ブリッジ・キューでのサポート |
|---|---|
| getMessage() | なし |
| putMessage() | あり |
| browseMessage() | なし |
| browseAndLockMessage | なし |

サポートされていない操作の使用がアプリケーションで試行された場合は、`Except_NotSupported` の MQeException が返されます。

アプリケーションがブリッジ・キューにメッセージを書き込むと、ブリッジ・キューは、そのブリッジのクライアント接続オブジェクトで保守されている接続のプールから、MQSeries キュー・マネージャーに対する論理接続を使用します。MQSeries への論理接続は、MQSeries キュー・マネージャー・プロキシーの設定でホスト名のフィールドに設定された値に従い、MQSeries Java バインディング・クラスか MQSeries Java クライアント・クラスでサポートされます。 MQSeries ブリッジ・キューは、MQSeries キュー・マネージャーに接続されると、その MQSeries キューに対してメッセージの書き込みを試行します。

MQSeries ブリッジ・キューは、非同期キューとして構成することができません。このブリッジ・キューのアクセス・モードは、常に同期でなければなりません。結果として、書き込み操作が直接 MQSeries ブリッジ・キューを操作し、正常に完了すれば、メッセージは、プロセスが書き込み操作の完了を待っている間に MQSeries システムに渡されています。

ブリッジ・キューに対して同期操作を使用することがアプリケーションにとって望まない場合は、MQSeries ブリッジ・キューを参照する非同期のリモート・キュー定義をセットアップする (113ページの『非同期キュー』を参照) か、あるいはストア・アンド・フォワード (蓄積交換) キュー、およびホーム・サーバー・キューをセットアップすることができます。このような構成を使用すれば、アプリケーションから非同期の方法でキューにメッセージを書き込むことができます。この構成で `putMessage()` メソッドが返された場合は、必ずしもメッセージが MQSeries キュー・マネージャーに渡されているとは限らない場合があります。

MQSeries ブリッジ・キューの使用の例は、136ページの『構成の例』で取り上げられています。

## 管理キュー

管理キューとは、管理メッセージを処理する方法が収められた特別なキューのことをいいます。このキューは、クラス MQeAdminQueue で実装される MQeQueue のサブクラスであるため、ローカル・キューと同じ機能を持っています。このキューは、管理クラス MQeAdminQueueAdminMsg を使用して管理されます。

管理キューに書き込まれるメッセージは、内部で処理されます。この理由で、アプリケーションが管理キューから直接メッセージを受け取ることはありません。 1 度に処理されるメッセージは 1 つだけです。 1 つのメッセージが処理されている間に着信する他のメッセージは、キューに入り、着信した順に処理されます。

管理されるリソースが使用されているためにメッセージが失敗した場合は、そのメッセージを再試行するように要求することができます。試行回数の最大値のセットアップについては、92ページの『基本となる管理要求メッセージ』で詳細に説明しています。管理対象リソースが使用可能でないためにメッセージが失敗し、試行回数がまだ最大値に達していない場合は、後日処理するためにメッセージはキューに残されます。試行回数が最大値に達した場合には、要求は失敗します。デフォルトでは、次にキュー・マネージャーが再始動される際にメッセージが再試行されますが、キューにタイマーを設定し、指定した間隔の後にキューでメッセージを処理することもできます。タイマーの間隔は、管理メッセージの `Queue_QTimerInterval` フィールドで指定することができます。この間隔のフィールドは `long` タイプのフィールドで、値はミリ秒で指定します。

## セキュリティーと管理

デフォルトでは、管理対象リソースの管理はすべての MQSeries Everyplace アプリケーションで可能です。アプリケーションは、管理されるキュー・マネージャーに対してローカル・アプリケーションとして稼働することもできますし、別のキュー・マネージャーで稼働することもできます。 MQSeries Everyplace を含め、ソリューションを展開する際には、そのソリューションに対する管理のセキュリティーが重要になります。セキュリティーが保護されなければ、システムは誤用される恐れがあります。 MQSeries Everyplace には、管理のセキュリティーを保護するための基本的な機能が備わっています。 175ページの『第7章 セキュリティー』は、キューに対してセキュリティーを適用する方法を示しています。このセクションで扱われる内容は、管理キューを保護するのに役立つでしょう。

同期的なセキュリティーが使用される場合は、管理キューにセキュリティー特性を設定することによってこれを保護することができます。たとえば、認証機能を設定し、管理アクションの許可を得るユーザーが、必ず Windows NT に対する認証を受けるようにすることができます。また、さらにこれを拡張して、特定の 1 ユーザーしか管理を実行できないようにすることもできます。

管理キューでは、アプリケーションが直接これにアクセスしてメッセージを書き込むことは許されておらず、メッセージは内部的に処理されます。これはつまり、メッセージ・レベルのセキュリティーによって保護されているキューに書き込まれたメッセージは、読み取りやブラウズの要求において属性を提供する通常のメカニズムでは開くことができないということを意味します。そのようなメッセージを管理キューで処理するためには、まず、キュー・ルール・クラスを管理キューに適用して、保護されているすべてのメッセージを開くことができます。メッセージを開いて管理を行えるようにするためには、キュー・ルール、`browseMessage()` ルールをコード化する必要があります。

キュー・ルールの実装に関する追加情報は、87ページの『キュー・ルール』を参照してください。

# 管理コンソールの例

MQSeries Everyplace で提供されている例の 1 つは、管理のグラフィカル・ユーザー・インターフェースです。この例では、本書の管理のセクションの中でこれまでに扱った多くの技法や機能を利用します。すべてのクラスはパッケージ `examples.administration.console` に含まれています。この例では、次のような多くの MQSeries Everyplace 管理機能を示します。

- ローカル・キュー・マネージャーとリモート・キュー・マネージャーの両方の管理
- すべての MQSeries Everyplace 管理対象リソースの管理
- 各管理対象リソースのすべてのアクションに対するアクセス
- ほとんどの基本的な MQeAdminMsg 機能の使用
- キュー・ブラウザー
- 管理応答キューでのキュー・ブラウザーの特殊な使用

ここで扱われる内容はすべて、プログラミングの例として紹介されており、***開発およびテスト環境外で使用するようには意図されていません***。この例の機能は、トレースやクライアント・キュー・マネージャーといった他の例と関連して使用されていること、および、MQSeries ブリッジの管理の例を示すためにサブクラス化されたものであることにご注意ください。 (142ページの『管理 GUI アプリケーションの例』を参照してください。)

## メイン・コンソール・ウィンドウ

コンソールを開始するには、次のコマンドを使用します。

```
java examples.administration.console.Admin
```

これにより、次のウィンドウが表示されます。

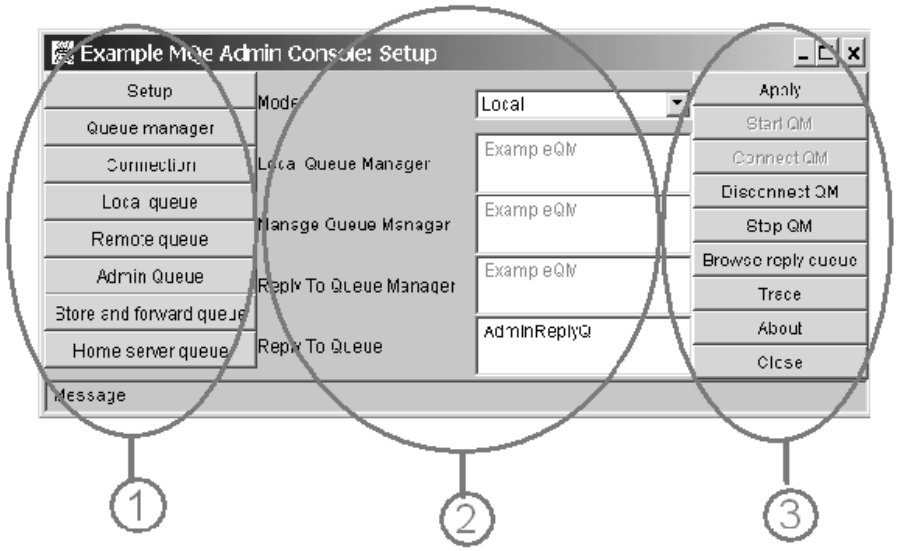


*図 22. 管理コンソール・ウィンドウ*

これは中心となるウィンドウで、他のすべての対話はこのウィンドウから開始されます。このウィンドウは 3 つのセクションに分かれています。

**1. 管理するリソースのタイプ**

ウィンドウの左側にある一連のボタンは、管理するリソースの選択を制御します。ここには、MQSeries Everyplace 管理対象リソースのタイプごとのボタンが 1 つずつと、「Setup (セットアップ)」という特別なボタンが 1 つあります。「Setup (セットアップ)」ボタンからは、応答先キューのブラウズやトレースのオン / オフといった、一連の基本的な管理機能にアクセスすることができます。

**2. 基本的な管理のパラメーター**

ウィンドウの中央部分では、基本的な管理のパラメーターを調整することができます。

**Mode (モード):**

管理するキュー・マネージャーがローカルであるかリモートであるか。

**Local queue manager (ローカル・キュー・マネージャー):**

管理機能にアクセスするローカル・キュー・マネージャーの名前。「Start QM (QM の開始)」ボタンでキュー・マネージャーが開始された場合、これは自動的に設定されます。

**Remote queue manager (リモート・キュー・マネージャー):**

管理するキュー・マネージャーの名前 (モードがリモートに設定されている場合)。モードがローカルに設定されている場合、キュー・マネージャー名は常にローカル・キュー・マネージャーと同じになります。

**Reply-to queue manager (応答先キュー・マネージャー):**
管理応答メッセージが送られるキュー・マネージャーの名前。

**Reply-to queue (応答先キュー):**
管理応答メッセージが送られるキューの名前。

3. **管理対象リソースに固有のアクション**
各管理対象リソースには、そこで実行できる一連のアクションというものが決まっています。メイン・ウィンドウの右側に表示される一連のボタンは、リソース (ウィンドウ左側の「resource to manage (管理するリソース)」ボタンで選択された) で使用できるアクションを示しています。いずれかのアクションのボタンを選択すると、そのアクションを実行する機能が導かれます (通常はそのアクションに関連する別のウィンドウが表示されます)。

選択されたキュー・マネージャーがまだ JVM で稼働していない場合は、コンソールが実行される JVM でこれを始動する必要があります。これには、「Start QM (QM の開始)」ボタンを使用することができます。このボタンを選択すると、キュー・マネージャーの始動パラメーターが含まれている ini ファイルの名前とパスが要求されます。キュー・マネージャーがすでに始動している場合は、「Connect QM (QM に接続)」ボタンを選択することができます (これは、管理が例のサーバー ExampleAwtMQeServer から開始されている場合です)。

キュー・マネージャーが開始されたなら、任意のリソースを選択および管理することができます。

## キュー・ブラウザー

ここでは、簡単なキュー・ブラウザーの例 (AdminQueueBrowser) を通して、キューをブラウズし、キューにあるメッセージの内容を表示する方法を紹介します。ブラウズできるのは、同期的にアクセスでき、かつユーザーがアクセスに必要な権限を持っているキューだけです。キューのメッセージがメッセージ・レベルのセキュリティーを使用して保護されている場合は、サンプル・コードを使用してそのメッセージを表示することはできません。

基本的なキュー・ブラウザーは、拡張機能を使用して管理応答キューをブラウズすることができるようにサブクラス化されています。これは、クラス AdminLogBrowser に実装されます。キュー・ブラウザーにアクセスする 1 つの方法としては、「Setup (セットアップ)」ボタンを選択してから「Browse reply queue (応答キューのブラウズ)」を選択します。次の図は、管理応答キュー・ウィンドウの概観を示したものです。

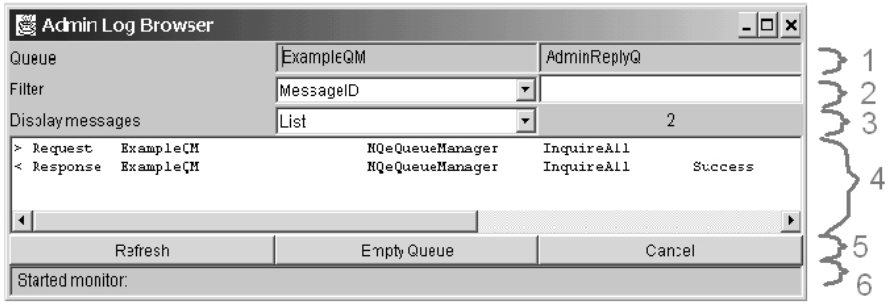


*図 23. 応答先キュー・ウィンドウ*

ウィンドウは、次のようないくつかのセクションに分かれています。

1. **キュー・マネージャーとキューに対する管理応答の名前**
2. **メッセージ・フィルター**

   フィルターを使用して、表示するメッセージのセットに制限を設けることができます。この例では、メッセージの MsgID フィールドと CorrelID フィールドでフィルターを行うことができます。また例では、 `byte array` でコード化されたストリングを含むフィールドを表示させるという前提も設けています。

   管理メッセージが例のコンソールから送信されると、MsgID は管理されるキュー・マネージャーの名前に設定されます。これにより、特定のキュー・マネージャーに対する管理メッセージだけを表示させることが可能になります。
3. **メッセージを表示する際の表示のタイプ**

   メッセージ表示パネル内のメッセージは、次のように、いくつかの異なった方法で表示することができます。

   **List (リスト):**

   1 行に要約されたキュー内のメッセージをリストします。

   **Full (全表示):**

   すべてのメッセージの内容を 1 つのパネルに表示します。

   **Both (リストと全表示):**

   2 つのパネルを使用し、1 方には各メッセージを要約する行のリストを、他方にはメッセージ・パネルで選択されたメッセージの内容を表示ます。

   ここには、現在見ている複数のメッセージを表示することができます。
4. **メッセージ表示パネル**

   3 でも説明した通り、このパネルには様々な形式でメッセージが表示されます。リスト表示されている中からメッセージをダブルクリックすると、そのメッセージの詳細表示が新しいウィンドウに表示されます。

**5. アクション**

次のようないくつかのボタンを通して、そのキュー・ブラウザーに固有のアクションを使用できます。

**Refresh (最新表示):**

画面をクリアし、キューの最新の内容を表示します。ブラウズしているキューがローカル・キューである場合は、自動的にモニターが開始されます。このモニターは、キューに新しいメッセージが追加されると画面を最新表示します。ブラウズされているキューがリモート・キューである場合は、新しいメッセージが追加されても自動的にウィンドウを最新表示することはできません。この場合には、「Refresh (最新表示)」ボタンを使用してキューの最新の内容を表示することができます。

**Empty Queue (キューのクリア):**

キューからすべてのメッセージを削除します。

**Cancel (取消):**

キュー・ブラウザーのウィンドウをクローズします。

**6. メッセージ**

ここには、エラー・メッセージや状況メッセージが表示されます。

## アクション・ウィンドウ

管理するリソースのタイプを選択し、アクションのボタンを選択すると、ウィンドウがオープンされ、そのアクションで使用できるパラメーターのリストが表示されます。これらのパラメーターには、必須のものとオプションとして使用されるものがあります。次の図は、接続における追加アクションの選択の例を示しています。

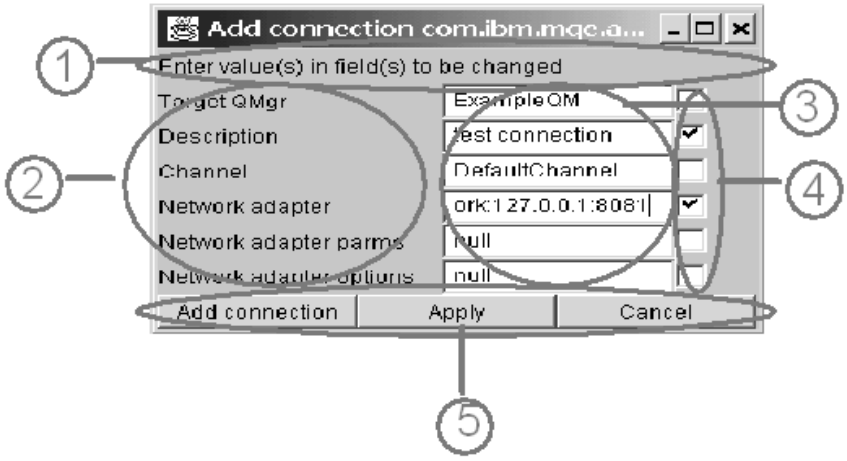


*図 24. アクション・ウィンドウ*

アクション・ウィンドウは、ほとんどのアクションで共通しています。このウィンドウは、次の部分から成っています。

1. **メッセージ領域**
   ここには、エラー・メッセージや状況メッセージが表示されます。
2. **パラメーターの名前**
3. **パラメーターの値**
   各パラメーター (フィールド) について、その名前と共に入力フィールドが表示され、そこから値を変更することができます。入力フィールドに表示される初期値は、そのフィールドのデフォルト値です。
4. **送信フィールド**
   各フィールドでは、値が変更されると自動的にこのチェック・ボックスが選択されます。このフィールド (チェック・ボックス) が選択されていると、それは、そのフィールドが管理メッセージに組み込まれるフィールドであることを示します。デフォルトでは、変更されている値だけが管理メッセージに組み込まれ、デフォルトの値は組み込まれません。管理メッセージは、メッセージのサイズを最小限に保つために、デフォルトの値を認識してこれをメッセージに含めません。なお、値をデフォルトに戻す要求を出す場合には、手動で送信フィールドのチェック・ボックスを選択する必要があります。
5. **アクション・ボタン**
   各アクションごとに、次の 3 つのボタンがあります。

   **Action (アクション):**
   そのアクションを要求する管理メッセージが作成され、宛先のキュー・マネージャーに送信されます。アクション・ウィンドウはクローズします。

   **Apply (適用):**
   そのアクションを要求する管理メッセージが作成され、宛先のキュー・マネージャーに送信されます。アクション・ウィンドウはオープンされたまま残り、同じメッセージを複数回送信したり、メッセージを変更して送信することができます。

   **Cancel (取消):**
   管理メッセージを送信せずにアクション・ウィンドウをクローズします。

## 応答ウィンドウ

管理要求の結果は、先に説明した通り、管理ログ・ブラウザーを使って見ることができます。要求の結果について詳細を表示する場合には、リスト表示されている中から応答メッセージをダブルクリックします。

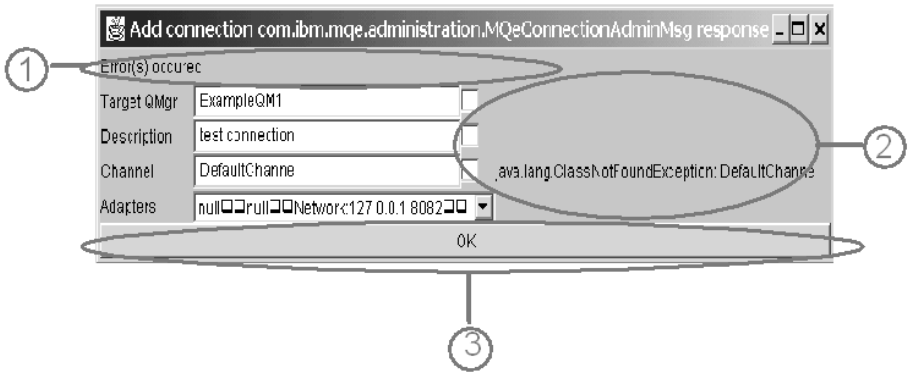


*図 25. 応答ウィンドウ*

このウィンドウの基本的な構造は管理要求のアクション・ウィンドウと同じですが、次の点が異なっています。

1. **メッセージ:**
   戻りコードとアクションの結果が表示されます。
2. **エラーの詳細:**
   戻りコードが RC_Mixed である場合は、特定のフィールドに関連するすべてのエラーがそのフィールドの隣りに表示されます。
3. **アクション・ボタン 「OK (了解)」:**
   アクション応答ウィンドウをクローズします。

# 第6章 MQSeries ブリッジ

MQSeries ブリッジは、MQSeries Everyplace ネットワークと MQSeries ネットワークがメッセージおよび内部作業を交換できるようにするためのソフトウェアの一部です。それぞれの目的を満たすための要件が異なるため、この 2 つのシステムがメッセージを渡す方法にも相違があります。ブリッジは、これらの相違を解決し、異なるシステム間でメッセージがフローするようにするものです。

## 導入

ブリッジ・コードは、MQeMQBridge.jar ファイルにパッケージされています。また、クラス・ファイルも com¥ibm¥mqe¥mqbridge ディレクトリーから入手可能です。クラス・パスは、MQSeries Everyplace サーバーの始動時に、ブリッジ・クラスがアクセス可能になるようにセットアップしなければなりません。ブリッジ・コードは、デバイスではなく MQSeries Everyplace ゲートウェイ・プラットフォームでのみ実行します。

### MQSeries Java クライアント

ブリッジを使用するには、MQSeries Java クライアント (バージョン 5.1 以降) が MQSeries Everyplace システムに導入されている必要があります。 Java クライアントは、Web サイト (http://www-4.ibm.com/software/ts/mqseries/txppacs/ma88.html) からサポート・パック MA88 として無料でダウンロードできます。 (NT クライアントは MQSeries バージョン 5.1 NT 版に付属しています。)

## MQSeries ブリッジの構成

ゲートウェイを構成するには、MQSeries キュー・マネージャーおよび MQSeries Everyplace キュー・マネージャーでいくつかの操作を行う必要があります。理論的には、ゲートウェイはメッセージの送信方向ごとに 2 つの部分に分かれています (MQSeries Everyplace から MQSeries へ、および MQSeries から MQSeries Everyplace へ)。

132ページの図26 が示すように、ブリッジ・オブジェクトは階層として定義されます。

さまざまなオブジェクトの関係は、以下のルールによって制御されます。

- 1 つの MQSeries ブリッジは単一の MQSeries Everyplace キュー・マネージャーに関連付けられます。
- 単一の MQSeries Everyplace キュー・マネージャーには、複数のブリッジ・オブジェクトを関連付けることができます。それぞれ異なる経路指定を使って複数の MQSeries ブリッジ・インスタンスを構成することもできます。
- 各ブリッジには、複数の MQSeries キュー・マネージャーを定義することができます。

- それぞれの MQSeries キュー・マネージャー定義には、MQSeries Everyplace と通信可能なクライアント接続を複数設定することができます。
- それぞれのクライアント接続は、単一の MQSeries キュー・マネージャーに接続します。ただし、各サービスごとに MQSeries キュー・マネージャーとの間で別々のサーバー接続 (つまり、異なるセキュリティー、送信出口と受信出口、ポートその他のパラメーターの組み合わせ) を使用することもできます。
- 1 つのゲートウェイ・クライアント接続につき、そのゲートウェイ・サービスを使って MQSeries キュー・マネージャーへ接続する複数のリスナーを設定できます。
- 1 つのリスナーは、接続を確立するためにクライアント接続を 1 つだけ使用します。
- 各リスナーは MQSeries システム上の単一の伝送キューに接続します。
- それぞれのリスナーは、(ゲートウェイに関連した MQSeries Everyplace キュー・マネージャーを使って) 単一の MQSeries 伝送キューから MQSeries Everyplace ネットワークの任意の位置にメッセージを移動します。したがって、各ゲートウェイでは、1 つの MQSeries Everyplace キュー・マネージャーを介して複数の MQSeries メッセージ・ソースを MQSeries Everyplace ネットワークに送ることができます。
- MQSeries Everyplace メッセージを MQSeries ネットワークに移動するとき、MQSeries Everyplace キュー・マネージャーは複数のアダプター・オブジェクトを作成します。それぞれのアダプター・オブジェクトは、(構成されていれば) 任意の MQSeries キュー・マネージャーに直接に接続して、任意のキューにメッセージを送信することができます。このように 1 つのゲートウェイを使って、単一の MQSeries Everyplace キュー・マネージャーを経由して、任意の MQSeries キュー・マネージャーへ MQSeries Everyplace メッセージを送ることができます。

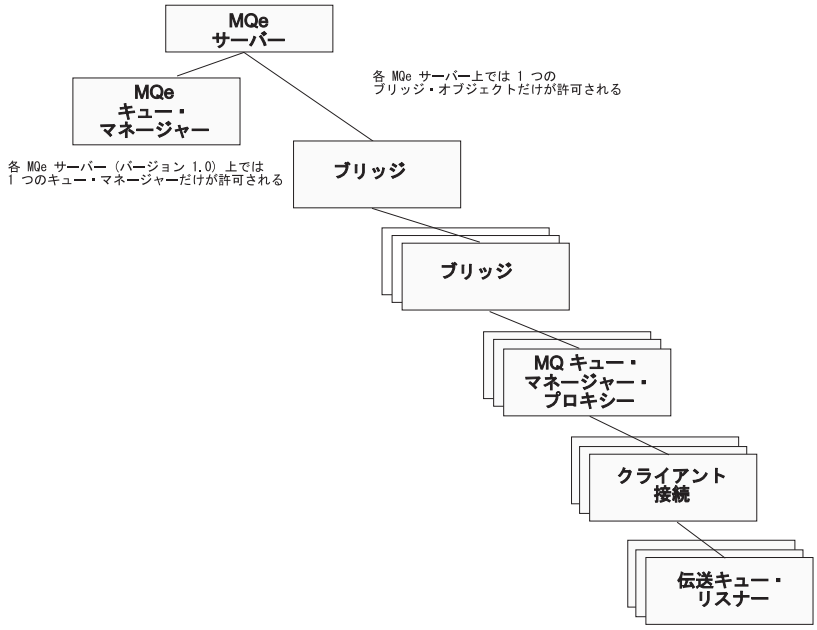


*図 26. ブリッジのオブジェクト階層*

## 基本インストールの構成

MQSeries ブリッジの基本的なインストールを構成するには、以下のステップを完了する必要があります。

1. MQSeries システムがインストールされていることを確認し、ローカルの経路指定規則とシステム構成方法を理解していることを確かめます。
2. MQSeries Everyplace をシステムにインストールします (MQSeries システムと同じシステムにインストールすることができます)。
3. MQSeries Java クライアントをまだインストールしていない場合は、Web からダウンロードしてインストールします (131ページの『MQSeries Java クライアント』を参照)。
4. MQSeries への接続を試行する前に、MQSeries Everyplace システムをセットアップして正常に作動することを確かめます。
5. MQe_java¥Classes¥JavaEnv.bat ファイルを更新して、MQSeries Java クライアントに含まれる Java クラス、および JRE (Java Runtime Environment) へのクラス・パスを指示するようにします。クラス com.ibm.mqbind.jar および com.ibm.mq.jar を必ずクラスパスに含め、ディレクトリー java¥lib および ¥bin がパスに含まれるようにしてください。
6. 実装する経路指定について計画します。どの MQSeries キュー・マネージャーがどの MQSeries Everyplace キュー・マネージャーと通信するかを決める必要があります。
7. MQSeries Everyplace オブジェクトと MQSeries オブジェクトの命名規則を決定して、将来の利用のために書き留めておきます。
8. MQSeries Everyplace システムを変更して、選択した MQSeries Everyplace サーバー上のブリッジを定義します。管理用 GUI (examples.mqbridge.awt.AwtMQBridgeServer) を使ってブリッジを定義することもできます。
9. 選択された MQSeries キュー・マネージャーおよび MQSeries Everyplace サーバー上のブリッジに接続します。

   MQSeries キュー・マネージャー上で 1 つまたは複数の Java サーバー接続チャネルを定義して、このキュー・マネージャーと通信する Java クライアント / バインディングを MQSeries Everyplace が使用できるようにします。これには、以下のステップが含まれます。

   a. (1 つまたは複数の) サーバー接続チャネルの定義。
   b. MQSeries Everyplace が使用する同期キューを定義して、MQSeries Everyplace が確実に MQSeries システムに送信できるようにする。サーバーごとのこれらの接続チャネルのうち、MQSeries Everyplace システムの使用するチャネルが 1 つ必要です。

   MQSeries Everyplace サーバー上で、MQSeries キュー・マネージャーに関する情報を収める MQSeries キュー・マネージャー・プロキシー・オブジェクトを 1 つ定義します。 MQSeries キュー・マネージャーのホスト名を取得して、それを MQSeries キュー・マネージャー・プロキシー・オブジェクトの中に含めます。

   Java クライアント / バインディングを使用して MQSeries システムのサーバー接続チャネルに接続する方法に関する情報を収めたクライアント接続オブジェク

トを、MQSeries Everyplace サーバー上に 1 つ定義します。ポート番号その他のサーバー接続チャネル・パラメーターを取得して、それらが MQSeries キュー・マネージャーと適合するようにします。

10. MQSeries Everyplace および MQSeries の構成を変更して、メッセージが MQSeries から MQSeries Everyplace へ送られるようにします。
    a. MQSeries から MQSeries Everyplace ネットワークへの経路の数を決定します。必要な経路の数は、それぞれの経路で予定しているメッセージ・トラフィック (負荷) の量によって決まります。メッセージ負荷が大きい場合には、トラフィックを多数の経路に分割することができます。
    b. 以下のようにして経路を定義します。
       - MQSeries システムに定義された伝送キューが、それぞれの経路ごとに 1 つ必要です。これらの伝送キューに対して、チャネルを定義しないでください。
       - それぞれの経路ごとに、 MQSeries Everyplace システム上に伝送キュー・リスナーが 1 つ必要です。
       - 一連のリモート・キュー定義 (リモート・キュー・マネージャー別名、キュー別名など) を選択して、すでに定義した MQSeries Everyplace 向けのさまざまな伝送キューに MQSeries メッセージが経路指定されるようにします。
11. MQSeries Everyplace の構成を変更して、メッセージが MQSeries Everyplace から MQSeries へ送られるようにします。
    a. MQSeries Everyplace ネットワークからメッセージを送る必要のある送信先 MQSeries ネットワーク上の、すべての MQSeries キュー・マネージャーに関する詳細情報を公表します。各 MQSeries キュー・マネージャーごとに、MQSeries Everyplace サーバー上で接続を定義する必要があります。この MQSeries キュー・マネージャーとの通信に通常のネットワークが使用されないことを示すために、キュー・マネージャー名を除くすべてのフィールドをヌルにする必要があります。
    b. MQSeries Everyplace ネットワークからメッセージを送る必要のある送信先 MQSeries ネットワーク上の、すべての MQSeries キューに関する詳細情報を公表します。各 MQSeries キューごとに、MQSeries Everyplace サーバー上で MQSeries ブリッジ・キューを定義する必要があります。 (これは DEFINE QREMOTE に対応する MQSeries Everyplace 上の定義です。)
       - キュー名は、この MQSeries ブリッジ・キューに到着するすべてのメッセージをブリッジが送る送信先 MQSeries キューの名前にします。
       - キュー・マネージャー名は、キューが最終的に格納される MQSeries キュー・マネージャーの名前です。
       - ブリッジ名は、MQSeries ネットワークへのメッセージ送信に使用されるブリッジを示します。
       - MQSeries キュー・マネージャー・プロキシー名は、MQSeries Everyplace 構成における、MQSeries キュー・マネージャーに接続可能な MQSeries キュー・マネージャー・プロキシー・オブジェクトの名前です。この MQSeries キュー・マネージャーには、経路を定義する必要があります。こうすると、メッセージがキュー・マネージャー名上のキュー名に通知されて、最終的な宛先に送られるようになります。

12. MQSeries システムおよび MQSeries Everyplace システムを開始して、メッセージが流れるようにします。 MQSeries システムでは、クライアント・チャネル・リスナーが開始されなければなりません。 MQSeries Everyplace システムでは、定義済みのすべてのオブジェクトが開始されなければなりません。これらのオブジェクトを開始するには、管理用 GUI を使用して明示的に開始するか、または開始時の状態 (実行または停止) を示すようにルール・クラスを構成して MQSeries Everyplace サーバーを再始動します。あるいは、この 2 つの方法を組み合わせます。手動でオブジェクトを立ち上げる最も簡単な方法は、関連するブリッジ・オブジェクトに開始コマンドを送ることです。その際、すべての子および子の子もまた開始するよう指示します。
    - メッセージが MQSeries Everyplace から MQSeries に渡るようにするには、MQSeries Everyplace 内のクライアント接続オブジェクトを開始する必要があります。
    - メッセージが MQSeries から MQSeries Everyplace に渡るようにするには、クライアント接続オブジェクト、および関連する伝送キュー・リスナーの両方を開始する必要があります。
13. 変換機能クラスを作成し、MQSeries Everyplace がそれらを使用するように構成を変更します。変換機能クラスは、特定の MQSeries メッセージ・フォーマットを MQSeries Everyplace メッセージ・フォーマットに、またはその逆に変換します。これらのフォーマット変換機能は Java で作成して、ブリッジ構成内の数か所に配置する必要があります。
    a. Java の変換機能クラスを作成する方法
        - ブリッジを介して渡す必要のある MQSeries メッセージのメッセージ・フォーマットを、アプリケーション開発者に問い合わせます。
        - MQSeries メッセージ・フォーマットを MQSeries Everyplace メッセージに変換する、一連の小さな変換機能クラスを作成します (詳しくは、変換機能の例を参照)。または、すべてのメッセージ・フォーマットを認識できて、MQSeries フォーマットと MQSeries Everyplace フォーマットの間の変換を実行できる大規模な変換機能を 1 つ作成します。あるいは、MQSeries メッセージ・フォーマットを認識できて、そのメッセージを変換する小さな変換機能をロードおよび起動できる 1 つの変換機能を作成します。 162ページの『変換機能』を参照してください。
    b. デフォルトの変換機能クラスを置換することもできます。そうするには、管理用 GUI を使用して、ブリッジ・オブジェクト構成内のデフォルト変換機能クラス・パラメーターを更新します。
    c. それぞれの MQSeries ブリッジ・キュー定義ごとに、デフォルト以外の変換機能を 1 つずつ指定することもできます。そうするには、管理用 GUI を使用して、構成内に存在するそれぞれの MQSeries ブリッジ・キューの「transformer (変換機能)」フィールドを「update (更新)」します。
    d. それぞれの MQSeries 伝送キュー・リスナーごとに、デフォルト以外の変換機能を 1 つずつ指定することもできます。そうするには、管理用 GUI を使用して、構成内に存在するそれぞれのリスナーの「transformer (変換機能)」フィールドを「update (更新)」します。
    e. ブリッジおよびすべてのリスナーを再始動します。

# サンプル構成ツール

MQSeries Everyplace システムおよび MQSeries ブリッジは複雑な環境であり、これらの構成は難しい場合があります。初期状態の構成を作成するのに役立つサンプル構成ツールが、 MQSeries ブリッジに付属しています。このツールのソース・コードが提供されています。必要に応じてこれをサブクラス化、修正、または動作変更することができます。

ここでは、このサンプル・ツールの機能と使用方法を説明します。

## 制限

このサンプル構成ツールは、多数の MQSeries Everyplace キュー・マネージャー接続が定義されているサーバーでは使用できません。たとえば、それぞれ別個のキュー・マネージャーに関連付けられた多数の携帯電話が存在して、サーバーにそれぞれの接続が定義されている場合には、このツールが一連の接続を照会することがあるため、正常には動作しません。このような状況では、メモリー不足のためにツールが停止して、ウィザードを実行している JVM が停止します。他の MQSeries Everyplace キュー・マネージャーへの接続が多数存在するサーバーを管理しようとしている場合には、代わりに `examples.mqbridge.administration.console.AdminGateway` アプリケーションを使用することをお勧めします。

## ブリッジの構成に必要なステップ

MQSeries ブリッジの基本的なインストールを構成するには、133ページの『基本インストールの構成』のステップを完了する必要があります。サンプル・ツールは、このリストのステップ 8 からステップ 12 までを簡素化するために提供されています。

# 構成の例

このセクションでは、4 つのシステムにおける構成の例を示します。

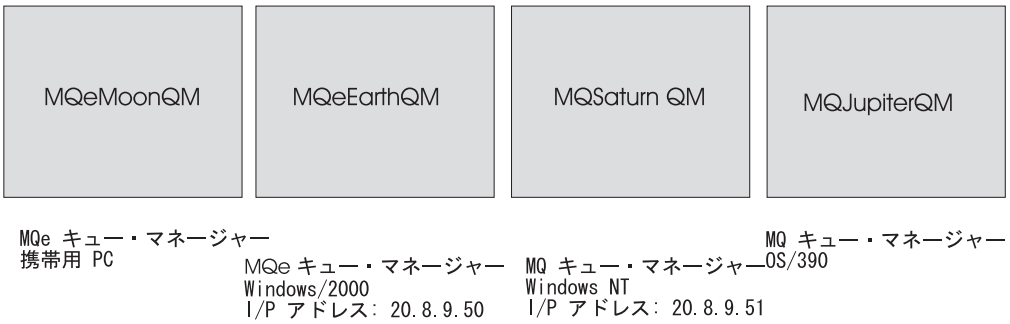


*図 27. 構成の例*

4 つのシステムは次のとおりです。

**MQeMoonQM**

携帯用 PC をサイトとする MQSeries Everyplace クライアント・キュー・マネージャーです。ユーザーは携帯用 PC を定期的にネットワークに接続して、MQeEarthQM MQSeries Everyplace ゲートウェイと連動させます。

**MQeEarthQM**

I/P アドレスが 20.8.9.50 である Windows 2000 マシン上で実行されます。これは MQSeries Everyplace ゲートウェイ (サーバー) キュー・マネージャーです。

**MQSaturnQM**

Windows NT プラットフォームにインストールされた MQSeries キュー・マネージャーです。 I/P アドレスは 20.8.9.51 です。

**MQJupiterQM**

System/390 プラットフォームにインストールされた MQSeries キュー・マネージャーです。

## 要件

この例の要件は、すべてのマシンが他の任意のマシン上のキューにメッセージを通知できることです。

断続的に接続される MQeMoonQM マシンを除いて、すべてのマシンがネットワークに継続的に接続されていることを想定します。

## 初期セットアップ

この例の場合、メッセージを収めることのできるローカル・キューが、すべてのキュー・マネージャーに存在することを想定しています。これらのキューは次のとおりです。

- MQeMoonQ (MQeMoonQM 上)
- MQeEarthQ (MQeEarthQM 上)
- MQSaturnQ (MQSaturnQM 上)
- MQJupiterQ (MQJupiterQM 上)

## MQeMoonQM が MQeEarthQM キュー・マネージャーとの間でメッセージを送受信できるようにする

**MQeMoonQM 上で**

**次の接続を定義します。**

*宛先キュー・マネージャー名*: MQeEarthQM
*アダプター*: Network:20.8.9.50

これで、MQeMoonQM がネットワークに接続されている間、アプリケーションは MQeEarthQM キュー・マネージャー上に定義された任意のキューを直接使用できます。要件によると、MQeMoonQM 上のアプリケーションはメッセージを MQeEarthQ に非同期的に送信できなければなりません。したがって、MQeEarthQ キューへの非同期リンクを設定するために、(自動検出に頼るのではなく) リモート・キューを定義する必要があります。

**リモート・キューを以下のように定義します。**

*キュー名*: MQeEarthQ
*キュー・マネージャー名*: MQeEarthQM
*アクセス・モード*: Asynchronous

これで、MQeMoonQM 上のアプリケーションは MQeMoonQ (ローカル・キュー) に同期的にアクセスし、MQeEarthQ に非同期的にアクセスすることができます。

## MQeEarthQM が MQeMoonQM キュー・マネージャーにメッセージを送信できるようにする

MQeMoonQM はまれにしかネットワークに接続されないため、MQeEarthQM 上でストア・アンド・フォワード (蓄積交換) キューを使用して、MQeMoonQM あてのメッセージを収集します。

**MQeEarthQM 上で**

**ストア・アンド・フォワード (蓄積交換) キューを定義します。**

キュー名: TO.HANDHELDS
キュー・マネージャー名: MQeEarthQM
(キューがキュー・マネージャー
MQeEarthQM 上に保管されるため)

**ストア・アンド・フォワード (蓄積交換) キューにキュー・マネージャーを追加します。**

キュー名: TO.HANDHELDS
キュー・マネージャー: MQeMoonQM

さらに、(同様の名前の) ホーム・サーバー・キューを MQeMoonQM キュー・マネージャー上にセットアップする必要があります。 MQeMoonQM キュー・マネージャーはストア・アンド・フォワード (蓄積交換) キューからメッセージを取り出して、それらを MQeMoonQM キュー・マネージャー上のキューに書き込みます。

**MQeMoonQM 上で**

**ホーム・サーバー・キューを定義します。**

キュー名: TO.HANDHELDS
キュー・マネージャー名: MQeEarthQM
(ホーム・サーバー・キューとして)

MQeMoonQM あてのメッセージが MQeEarthQM に到着すると、すべてストア・アンド・フォワード (蓄積交換) キュー TO.HANDHELDS に一時的に保管されます。MQeMoonQM が次にネットワークに接続した時点で、ホーム・サーバー・キュー TO.HANDHELDS はストア・アンド・フォワード (蓄積交換) キューにあるメッセージを取り出して MQeMoonQM キューに送ります。その後、メッセージはローカル・キューに保管されます。

これで、MQeEarthQM 上のアプリケーションはメッセージを MQeMoonQ に非同期的に送ることができます。

## MQeEarthQM が MQSaturnQ にメッセージを送信できるようにする

**MQeEarthQM 上で**

**以下のブリッジを定義します。**

ブリッジ名: MQeEarthQMBridge

**MQ キュー・マネージャー・プロキシーを定義します。**

*ブリッジ名*: MQeEarthQMBridge
*MQ QMgr プロキシー名*: MQSaturnQM
*ホスト名*: 20.8.9.51

**クライアント接続を定義します。**

*ブリッジ名*: MQeEarthQMBridge
*MQ QMgr プロキシー名*: MQSaturnQM
*ClientConnectionName*: MQeEarth.CHANNEL
*SyncQName*: MQeEarth.SYNC.QUEUE

**次の接続を定義します。**

*ConnectionName*: MQeSaturnQM
*チャネル*: ヌル
*アダプター*: ヌル

**MQBridge キューを定義します。**

*キュー名*: MQSaturnQ
*MQ キュー・マネージャー名*: MQSaturnQM
*ブリッジ名*: MQeEarthQMBridge
*MQ QMgr プロキシー名*: MQSaturnQM
*ClientConnectionName*: MQeEarth.CHANNEL

**MQSaturnQM 上で**

**サーバー接続チャネルを定義します。**

*名前*: MQeEarth.CHANNEL

**ローカル同期キューを定義します。**

*名前*: MQeEarth.SYNC.QUEUE

確実な送信のためには、同期キューが必要です。

これで、MQeEarthQM 上のアプリケーションは MQSaturnQM 上の MQSaturnQ にメッセージを送信できます。

## MQeEarthQM が MQJupiterQ にメッセージを送信できるようにする

**MQeEarthQM 上で**

**次の接続を定義します。**

*ConnectionName*: MQeJupiterQM
*チャネル*: ヌル
*アダプター*: ヌル

**MQBridge キューを定義します。**

*キュー名*: MQJupiterQ
*MQ キュー・マネージャー名*: MQJupiterQM

*ブリッジ名*: MQeEarthQMBridge
*MQ QMgr プロキシー名*: MQSaturnQM
*ClientConnectionName*: MQeEarth.CHANNEL

**MQSaturnQM 上で**

**リモート・キューを定義します。**

*キュー名*: MQJupiterQ
*伝送キュー*: MQJupiterQM.XMITQ

**MQSaturnQM および MQJupiterQM の両方で**

MQSaturnQM 上の MQJupiterQM.XMITQ から MQJupiterQM にメッセージを移動するためのチャネルを定義します。

これで、MQeEarthQM 上のアプリケーションは MQSaturnQM を介して MQJupiterQM 上の MQJuputerQ にメッセージを送信できます。

## MQeMoonQM が MQJupiterQ および MQSaturnQ にメッセージを送信できるようにする

**MQeMoonQM 上で**

**次の接続を定義します。**

*宛先キュー・マネージャー名*: MQSaturnQM
*アダプター*: MQeEarthQM

**リモート・キューを定義します。**

*キュー名*: MQSaturnQ
*キュー・マネージャー名*: MQSaturnQM
*アクセス・モード*: Asynchronous

MQSaturnQM キュー・マネージャー向けのすべてのメッセージは MQeEarthQM キュー・マネージャーを介して送られることが、この接続によって示されています。

**次の接続を定義します。**

*宛先キュー・マネージャー名*: MQJupiterQM
*アダプター*: MQeEarthQM

**リモート・キューを定義します。**

*キュー名*: MQJupiterQ
*キュー・マネージャー名*: MQJupiterQM
*アクセス・モード*: Asynchronous

これで、MQeMoonQM に接続したアプリケーションは、携帯用 PC がネットワークから切断されていても、MQeMoonQ、MQeEarthQ、MQSaturnQ、および MQJupiterQ のすべてに向けてメッセージを発信できるようになりました。

## MQSaturnQM が MQeEarthQ にメッセージを送信できるようにする

**MQSaturnQM 上で**

**ローカル・キューを定義します。**

キュー名: MQeEarth.XMITQ
キュー・タイプ: 伝送キュー

**キュー・マネージャー別名 (リモート・キュー定義) を定義します。**

キュー名: MQeEarthQM
リモート・キュー・マネージャー名: MQeEarthQM
伝送キュー: MQeEarth.XMITQ

**MQeEarthQM 上で**

**伝送キュー・リスナーを定義します。**

ブリッジ名: MQeEarthQMBridge
*MQ QMgr* プロキシー名: MQSaturnQM
*ClientConnectionName*: MQeEarth.CHANNEL
リスナー名: MQeEarth.XMITQ

これで、MQSaturnQM 上のアプリケーションは MQeEarthQM キュー・マネージャー別名を使って MQeEarthQ へメッセージを送信できます。各メッセージは MQeEarth.XMITQ に経路指定され、そこで MQe 伝送キュー・リスナー MQeEarth.XMITQ がメッセージを収集した後、MQSeries Everyplace ネットワークに向けて移動します。

## MQSaturnQM が MQeMoonQ にメッセージを送信できるようにする

**MQSaturnQM 上で**

**キュー・マネージャー別名 (リモート・キュー定義) を定義します。**

キュー名: MQeMoonQM
リモート・キュー・マネージャー名: MQeMoonQM
伝送キュー: MQeEarth.XMITQ

これで、MQSaturnQM 上のアプリケーションは MQeMoonQM キュー・マネージャー別名を使って MQeMoonQ へメッセージを送信できます。各メッセージは MQeEarth.XMITQ に経路指定され、そこで MQe 伝送キュー・リスナー MQeEarth.XMITQ がメッセージを収集した後、MQSeries Everyplace ネットワークに向けて通知します。

ストア・アンド・フォワード (蓄積交換) キュー TO.HANDHELDS がメッセージを収集し、MQeMoonQM が次にネットワークに接続した時点で、ホーム・サーバー・キューがストア・アンド・フォワード (蓄積交換) キューからメッセージを取得して、それらを MQeMoonQ に送ります。

### MQJupiterQM が MQeMoonQ にメッセージを送信できるようにする

**MQJupiterQM 上で**

MQeEarthQM および MQeMoonQM のリモート・キュー・マネージャー別名を設定して、メッセージが通常の MQSeries 経路指定手法によって MQSaturnQM に経路指定されるようにします。

これで、すべてのキュー・マネージャーに接続するすべてのアプリケーションは、MQeMoonQ、MQeEarthQ、MQSaturnQ、または MQJupiterQ のいずれに対してもメッセージを通知できるようになりました。

## 追加のブリッジ構成

通常、基本 MQSeries Java クラスのトレースは必要ではないため、デフォルトでは使用不可になっています。ただし、MQSeries トレースの初期設定はアクティブ・トレース・ハンドラー・クラスで行う必要があり、初期設定方法の例が MQSeries Everyplace クラスに付属して提供されています。ブリッジ・トレース・クラスの例は `examples.mqbridge.awt.AwtBridgeTrace` です。このクラスは、ブリッジ管理用 GUI によって自動的にインスタンス化されます。ブリッジ・トレース・メッセージは、いくつかの言語で `examples.mqbridge.trace` に収められています。

さらに、com.ibm.mq.MQException.log (デフォルトでは System.err) に定義された OutputStreamWriter に MQExceptions が記録されます。基本 MQSeries トレースの初期設定および構成について、詳しくは MQSeries Using Java の資料を参照してください。

# MQSeries ブリッジの管理

このセクションには、MQSeries ブリッジの管理に関連した作業についての情報が含まれています。

## 管理 GUI アプリケーションの例

MQSeries ブリッジには管理 GUI の例が提供されています。これは、124ページの『管理コンソールの例』で説明されている `examples.administration.console.Admin` のサブクラスの例です。

サブクラスは `examples.mqbridge.administration.console.AdminGateway` と呼ばれます。

ブリッジ機能はクライアント・キュー・マネージャーでは実行することができません。そのため、このクラスをクライアント・キュー・マネージャーと共に使用しても、そのクライアント・キュー・マネージャー上でブリッジ・オブジェクトを管理することはできません。しかし、リモート MQSeries Everyplace ブリッジが使用可能なサーバー・キュー・マネージャーの管理は行えます。

ローカル・キュー・マネージャーに接続されたブリッジを管理するには、サーバー・プログラムの例 `<java> examples.mqbridge.awt.AwtMQBridgeServer <server_ini_file>` を使用して、MQSeries Everyplace サーバーを開始します。


*図 28. MQSeries ブリッジ管理 GUI サーバー・ウィンドウ*

サーバー・ウィンドウから、次のどちらかのオプションを使用できます。

- 「Admin (管理)」ボタンをクリックして、`examples.mqbridge.administration.console.AdminGateway` クラスを使用し、ローカル・サーバー・キュー・マネージャーを管理する。これがブリッジ・オブジェクトです。
- 「MQBridge setup (MQBridge のセットアップ)」ボタンをクリックして、136ページの『サンプル構成ツール』で説明されているように `examples.mqbridge.setup.MQBridgeWizard` クラスの例を呼び出す。

どちらの例も、ブリッジ固有の管理メッセージのサブクラス (MQeMQBridgesAdminMsg、 MQeMQBridgeAdminMsg、 MQeMQQMgrProxyAdminMsg、 MQeClientConnectionAdminMsg、 MQeListenerAdminMsg、および MQeMQBridgeQueueAdminMsg) を使って、ブリッジ構成オブジェクトを方針に基づいて操作する方法を示しています。

## ブリッジ管理アクション

### 実行状態

管理オブジェクトには、それぞれ実行状態があります。そのオブジェクトがアクティブであれば実行中であり、そうでなければ停止になります。

管理オブジェクトが停止状態になっているときには、それを使用することはできません。しかしそのオブジェクトの構成パラメーターを照会したり更新したりすることはできます。

MQSeries ブリッジのキューが停止状態のブリッジ管理オブジェクトを表す場合、ブリッジ、MQSeries キュー・マネージャー・プロキシー、およびクライアント接続管理オブジェクトがすべて開始されるまでは、MQSeries Everyplace メッセージを MQSeries ネットワークに送ることはできません。

管理オブジェクトの実行状態は、 MQeMQBridgeAdminMsg、 MQeMQQMgrProxyAdminMsg、 MQeClientConnectionAdminMsg、 MQeListenerAdminMsg のいずれかの管理メッセージ・クラスから開始 / 停止アクションを使って変更することができます。

この後のセクションでは、ブリッジ管理オブジェクトがサポートするアクションについて説明します。

### 開始アクション

管理者は開始アクションをどの管理オブジェクトにも送ることができます。

'affect children' ブール・フラグは、このアクションの結果に影響を与えます。'affect children' ブール・フィールドがメッセージに含まれており、それが 'true' に設定されている場合、開始アクションによって管理オブジェクトとその子 (および子の子) がすべて開始されます。このフラグがメッセージに含まれていないか、それが 'false' に設定されている場合、開始アクションを受け取る管理オブジェクトだけがその実行状態を変更します。たとえば、ブリッジに開始アクションを送り、'affect children' を true に指定すると、すべてのプロキシー、クライアント接続、および親元であるリスナーがすべて開始されます。 'affect children' が指定されていない場合、ブリッジだけが開始されます。管理オブジェクトの親管理オブジェクトがまだ開始されていない場合、管理オブジェクトを開始することはできません。そのため、開始イベントを管理オブジェクトに送ると、階層の上位にあるすべてのオブジェクトがまだ実行されていなければ、それらが開始されます。

### 停止アクション

管理オブジェクトに停止アクションを送ると、そのオブジェクトを停止することができます。受信側の管理オブジェクトは、必ず階層の下位にあるすべてのオブジェクトが停止していることを確かめてから、自分自身を停止させます。

### 問い合わせアクション

問い合わせアクションは、送信先の管理オブジェクトからの値を照会します。

管理オブジェクトが実行中状態になっている場合、問い合わせによって戻される値は現在使用されているものです。停止状態のオブジェクトから戻される値は、更新アクションによって値に加えられた最新の変更を反映します。このように、開始、更新、問い合わせという一連のアクションでは、更新*前* に構成された値が戻されますが、開始、更新、停止、問い合わせという一連のアクションでは、更新*後* に構成された値が戻されます。

可変値を更新する前に管理オブジェクトを停止すれば、混乱は少なくて済むでしょう。

### 更新アクション

更新アクションでは、管理オブジェクトの特性に関する 1 つまたは複数の値を更新します。更新アクションで設定された値は、管理オブジェクトが次に停止されるまで、現在の値として使用されません。 (『問い合わせアクション』を参照してください。)

### 削除アクション

削除アクションでは、管理オブジェクトに関する現在の情報と永続的な情報をすべて削除します。 'affect children' ブール・フラグは、このアクションの結果に影響を与えます。 'affect children' フラグを使用しており、それが 'true' に設定されている場合、このアクションを受け取る管理オブジェクトは停止アクションを出します。次に、階層内で管理オブジェクトの下位にあるすべてのオブジェクトに対して削除アクションがとられ、こうして 1 つのアクションで階層の全部分が削除されます。フラグを使用していないか、それが false に設定されている場合、管理オブジェクトだけが削除されます。しかし、このアクションは、階層内の現行のオブジェクトの下位にあるすべてのオブジェクトが削除されるまで実行できません。

### 作成アクション

作成アクションは、管理オブジェクトを作成します。作成された管理オブジェクトの実行状態は最初は停止に設定されます。

## MQSeries キュー・マネージャーのシャットダウン

MQSeries キュー・マネージャーを停止する前に、すべての MQSeries キュー・マネージャー・プロキシーのブリッジ管理オブジェクトに対して STOP 管理メッセージを発行し、MQSeries キュー・マネージャーを使って MQSeries Everyplace ネットワークを停止することをお勧めします。 MQSeries キュー・マネージャー・プロキシーのブリッジ・オブジェクトを停止すると、MQSeries Everyplace アクティビティーは MQSeries キュー・マネージャーのシャットダウンによる悪影響を受けません。(このことは、STOP 管理メッセージを MQSeries Everyplace ブリッジ・オブジェクトに対して 1 回発行することによって行うこともできます。)

MQSeries キュー・マネージャーをシャットダウンする前に MQSeries キュー・マネージャー・プロキシーのブリッジ・オブジェクトを停止しない場合、 MQSeries シャットダウンと MQSeries ブリッジの動作は、選択する MQSeries キュー・マネージャー・シャットダウンのタイプ、つまり即時シャットダウンと制御シャットダウンによって異なります。

### 即時シャットダウン

即時シャットダウンを使って (強制的に) MQSeries キュー・マネージャーを停止すると、ブリッジと MQSeries キュー・マネージャーとの間のすべての接続が切断されます (これは、 Java バインディングまたは Java クライアント・チャネルのいずれかを使って構成される接続に当てはまります)。 MQSeries システムは通常どおりシャットダウンを行います。

即時シャットダウンを行うと、すべてのブリッジ伝送キュー・リスナーは即時に停止されます。このとき、各リスナーには MQSeries キュー・マネージャーの即時停止によって、伝送がシャットダウンされることが警告されます。

アクティブになっている MQSeries ブリッジ・キューは、以下の状態になるまで、MQSeries キュー・マネージャーへの (中断した) 接続を保存します。

- アイドル・タイムアウト期間 (クライアント接続ブリッジ・オブジェクトで指定された期間) の間アイドル状態になった後の接続タイムアウト。この時点で、中断した接続はクローズされます。
- MQSeries ブリッジ・キューには、中断した接続を使用しようとする何らかのアクション (たとえば、MQSeries に対してメッセージを送る) をとるように通知が出されます。 putMessage 操作は失敗し、中断した接続はクローズされます。

MQSeries ブリッジ・キューに接続がない場合、そのキューに対して次の操作を行うと、新しい接続が獲得されます。 MQSeries キュー・マネージャーが使用可能になっていない場合、そのキューに対する操作はすぐに失敗します。 MQSeries キュー・マネージャーをシャットダウン後に再始動しており、ブリッジ・キューに対してキュー操作 (putMessage など) を行う場合、アクティブな MQSeries キュー・マネージャーに対する新しい接続が確立され、操作は予測どおりに実行されます。

### 制御シャットダウン

制御シャットダウンを使って MQSeries キュー・マネージャーを停止しても、接続が強制的に切断されることはありません。すべての接続がクローズされるまで待機します (これは、Java バインディングまたは Java クライアント・チャネルを使って構成される接続に当てはまります)。アクティブなブリッジ伝送キュー・リスナーは、MQSeries システムが静止していることを通知し、関連する警告を出して停止します。

アクティブになっている MQSeries ブリッジ・キューは、以下の状態になるまで、MQSeries キュー・マネージャーへの接続を保存します。

- アイドル・タイムアウト期間 (クライアント接続ブリッジ・オブジェクトで指定された期間) の間アイドル状態になった後の接続タイムアウト。この時点で中断した接続はクローズされ、MQSeries キュー・マネージャーの制御シャットダウンは完了します。
- MQSeries ブリッジ・キューには、中断した接続を使用しようとする何らかのアクション (たとえば、MQSeries に対してメッセージを送る) をとるように通知が出されます。 `putMessage` 操作は失敗し、中断した接続はクローズされます。そして、MQSeries キュー・マネージャーの制御シャットダウンが完了します。

ブリッジ・クライアント接続オブジェクトは接続のプールを保守し、それらは使用されるのを待機します。ブリッジ・アクティビティーがない場合、接続アイドル時間がアイドル・タイムアウト時間 (クライアント接続オブジェクトの構成で指定された時間) を超えるまで、プールには MQSeries クライアント・チャネル接続が保存されます。アイドル・タイムアウト時間を超えた時点で、プール内のチャネルはクローズされます。

MQSeries キュー・マネージャーへの最後のクライアント・チャネル接続がクローズされると、 MQSeries 制御シャットダウンは完了します。

## 管理オブジェクトとその特性

このセクションでは、MQSeries Everyplace と MQSeries の間のブリッジに関連した様々な管理オブジェクトのタイプについて、それぞれの特性を説明します。特性とは、`inquireAll()` 管理メッセージを使って照会できるオブジェクト属性です。アプリケーションでは結果を読み取って使用することができます。あるいは、特性の値を設定するために管理メッセージの更新または作成時に送信することもできます。特性によっては、管理メッセージの作成および更新を使って設定することもできます。それぞれの特性には固有のラベルが関連付けられており、このラベルを使って特性の値を設定および入手することができます。

以下のリストは、それぞれの管理オブジェクトに当てはまる属性を示しています。属性については、148ページの『属性の詳細』でアルファベット順に詳しく説明されています。ラベル定数は com.ibm.mqe.mqbridge.MQeCharacteristicLabels で定義されます。

**ブリッジ・オブジェクトの特性**

- Run-state
- Children
- Child

### ブリッジ・オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- HeartBeatInterval
- DefaultTransformer

### MQSeries キュー・マネージャー・プロキシー・オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- HostName

### クライアント接続オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- Port
- AdapterClass
- MQUserID
- MQPassword
- SendExit
- ReceiveExit
- SecurityExit
- CCSID
- SyncQName
- SyncQPurgerRulesClass
- MaxConnectionIdleTime
- SyncQPurgeInterval

### MQSeries 伝送キュー・リスナー・オブジェクトの特性

- Run-state
- Children
- Child
- AdministerObjectClass
- StartupRuleClass
- BridgeName
- MQQMgrProxyName
- ClientConnectionName
- ListenerName
- DeadLetterQName
- ListenerStateStoreAdapter
- UndeliveredMessageRuleClass
- TransformerClass

## 属性の詳細

*属性:* AdapterClass

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_ADAPTER_CLASS
```

*有効なアクション*

照会、作成、更新

*説明* これは、java クラス名か、または java クラス名に解析できる別名です。ゲートウェイ・スレーブによって使用されます。

これが指定されていない場合、デフォルトの com.ibm.mqe.mqbridge.MQeMQAdapter が使用されます。このパラメーターの妥当性検査は行われません。

*属性:* AdministeredObjectClass

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_ADMINISTERED_OBJECT_CLASS
```

*有効なアクション*

照会、作成、更新

*説明* ブリッジの名前。

有効な文字は、0〜9、A〜Z、a〜z、-、.、%、/ です。

*属性:* BridgeName

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_BRIDGE_NAME
```

*有効なアクション*

照会、作成、更新、削除、開始、停止

*説明* 記号名を使用する場合、マシンのスイッチが入っていないかを検出したり、ネーム・サーバーが作動していないかを検出したりするのに時間がかかることがあります。これが問題の原因となっている場合は、このフィールドで代わりとしてドット 10 進 I/P アドレスを使用することができます。

**注:** 管理メッセージの作成が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

*属性:* CCSID

*タイプ:*

Int

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CCSID
```

*有効なアクション*

照会、作成、更新

*説明* このパラメーターの説明については、MQSeries Using Java の資料を参照してください。

有効な値は 0 ～ MAXINT です。デフォルトは 0 です。

*属性:* Child

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CHILD
```

*有効なアクション*

照会

*説明* MQSeries ブリッジ管理オブジェクトの名前が入っているフィールド。

*属性:* Children

*タイプ:*

MQeFields 配列

*ラベル:*

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CHILDREN

*有効なアクション*
照会

*説明* Child フィールドの配列。それぞれのエレメントには Child 属性が含まれます。

*属性:* ClientConnectionName

*タイプ:*
Unicode

*ラベル:*

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_CLIENT_CONNECTION_NAME

*有効なアクション*
照会、作成、更新、削除、開始、停止

*説明*

**注:** 管理メッセージの作成が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

*属性:* DeadLetterQName

*タイプ:*
Unicode

*ラベル:*

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_DEAD_LETTER_Q_NAME

*有効なアクション*
照会、作成、更新

*説明* ゲートウェイが MQSeries から MQSeries Everyplace にメッセージを送達できないことを検出する場合 (原因としては、宛先 MQSeries Everyplace キューに設定されているサイズ制限が考えられる)、ゲートウェイはメッセージを処理することができません。そのため、そのメッセージは、MQSeries システムの送達不能キューに入れられます。このパラメーターは、エラーが起きているメッセージの送達先のキューを定義します。

この値が指定されていない場合、SYSTEM.DEAD.LETTER.QUEUE という値が使用されます。

*属性:* DefaultTransformer

*タイプ:*
Unicode

*ラベル:*

com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_DEFAULT_TRANSFORMER

*有効なアクション*
照会、作成、更新

*説明* ここで指定されるクラス名がデフォルトの変換機能クラスとして使用されます。 MQSeries から MQSeries Everyplace にメッセージが送信されるとき、宛先キューが変換機能クラスを定義する場合があります。定義されない場合、このクラスが MQSeries メッセージを MQSeries Everyplace フォーマットに変換するために使用されます。

MQSeries Everyplace から MQSeries にメッセージが送信されるとき、メッセージを MQSeries Everyplace に移動させる伝送キュー・リスナーに再び変換機能クラスが定義される場合があります。定義されない場合、このクラスが MQSeries メッセージを MQSeries Everyplace フォーマットに変換するために使用されます。

このフィールドの値の妥当性検査は行われません。

デフォルト値は `com.ibm.mqebridge.MQeBaseTransformer` です。

*属性:* HeartBeatInterval

*タイプ:*
Int

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_HEARTBEAT_INTERVAL
```

*有効なアクション*
照会、作成、更新

*説明* 1 分単位で表現した時間間隔。 1<= 値 <=60。ブリッジは内部的な心拍を使用して、他の管理オブジェクトに定期的な刺激を伝えます。心拍イベント (「クライアント接続は古くなった MQSeries をリープします」や「同期キューは除去されます」など) が届くと、管理オブジェクトは小さいタスクを実行します。心拍はタイマーを分割できない単位に細分化するので、この値を低く設定すれば、現在の時刻と比較してとられるアクションはより正確になります。たとえば、「アイドル時間が 10 分を超えた場合はすべての MQSeries 接続をリープします」と決定しても、心拍間隔が 3 分に設定されている場合、アイドル状態になっている MQSeries 接続は、3、6、9、12 分後にそれぞれ検査されますが、リープされるのは 12 分後だけです。この値を低く設定すると、タイマー関連の心拍イベントの正確度は高くなりますが、作業効率が犠牲になります。作成される心拍イベントが増えるほど、実行する作業も増えるからです。

デフォルト値は 5 分です。

*属性:* Hostname

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_HOST_NAME
```

*有効なアクション*
照会、作成、更新

*説明* MQSeries Java クラスを使ってこの MQSeries キュー・マネージャーへの接続を作成するために使用されます。これを指定しない場合、MQSeries キュー・マネージャーは JVM と同じマシン上にあるものと見なされ、MQSeries システムとの対話には java バインディングが使用されます。

**注:** 値をブランクのままにしても、″localhost″ を指定したことにはなりません。ブランク値を使用すると、ブリッジは、MQSeries と直接対話を行う MQSeries java バインディングを使用します (この処理のほうが高速です)。 ″localhost″ を指定してもまったく同じ結果になりますが、ブリッジは MQSeries java クライアント・クラスを使用します。これは MQSeries とのすべての対話がネットワーク (TCP/IP) スタック通信を介して行われることを意味します。

ここで指定された値の妥当性検査は行われません。記号名を使用すると、マシンのスイッチが入っていないかを検出したり、ネーム・サーバーが作動していないかを検出したりするのに時間がかかることがあります。これが問題の原因となっている場合は、このフィールドで代わりとしてドット 10 進 I/P アドレスを使用することができます。

*属性:* ListenerName

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_NAME
```

*有効なアクション*
照会、作成、更新、削除、開始、停止

*説明* このリスナーの名前。リスナー名は、リスナーがメッセージを受け取る MQSeries 上の伝送キューの名前です。 MQ_queue_manager_name と MQ_transmission_queue_name のペアの組み合わせは、存在するすべてのゲートウェイの間で固有でなければなりません。

**注:** 管理メッセージの作成が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

*属性:* ListenerStateStoreAdapter

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_LISTENER_STATE_STORE_ADAPTER
```

*有効なアクション*
照会、作成、更新

*説明* 永続メッセージを確実に送達するために、リスナー・クラスでは、アダプターを使用して状況情報を保管します。これは、ディスクとの間の状況情報の保管および回復を管理するためにロードされたアダプターのクラス名 (またはクラス名の別名) です。現在、2 つのアダプターがサポートされています。それは com.ibm.mqe.adapters.MQeDiskFieldsAdapter (ローカル・ファイル・システムに状況情報を保管) と com.ibm.mqe.mqbridge.MQeMQAdapter (MQSeries サーバーに状況情報を保管) です。一般に、ディスク・アダプターを使うと MQSeries ベースのアダプターを使う場合よりも処理速度が速くなります。クラス名の後には、コロンで区切った引き数のリストを続けることができます。ただし、これを使用するのは MQeDiskFieldsAdapter だけです。この場合、MQeDiskFieldsAdapter の後にコロンと、状況情報を含むファイルへの完全修飾パス名を続けることができます。たとえば、ディスク・フィールド・アダプターを使ってリスナーの状況情報をファイル `c:¥folder¥state.sta` に保管するには、listener-state-store-adapter フィールドに ″com.ibm.mqe.Adapters.MQeDiskFieldsAdapter:c:¥folder¥state.sta″ という値が含まれていなければなりません。このパラメーターで指定されたファイルはすでに存在している必要はありません。提供されたパス名がフォルダー区切り文字 (たとえば、DOS では '¥') で終わっている場合には、提供されたパラメーターがディレクトリーであると想定されます。そして、'<ListenerName>-listener.sta' という状況ファイルがそのディレクトリーに作成されます (ここで、<ListenerName> はレジストリー項目にあるリスナーの名前です)。パス名が提供されていない場合、リスナーは現行の Java 作業ディレクトリーにある '<ListenerName>-listener.sta' というファイルを使用します。 MQeMQAdapter が使用されている場合、追加の引き数を指定する必要はありません。

ListenerStateStoreAdapter フィールドのデフォルト値は ″com.ibm.mqe.Adapters.MQeDiskFieldsAdapter″ です。

*属性:* MaxConnectionIdleTime

*タイプ:*

Int

*ラベル:*

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_MAX_CONNECTION_IDLE_TIME`

*有効なアクション*

照会、作成、更新

*説明* ブリッジにあるそれぞれのクライアント接続オブジェクトは、その MQSeries システムへの MQSeries Java クライアント接続のプールで保守されます。

MQSeries 接続が使用されずにアイドル状態になると、プールに入っているかどうかにかかわらず、タイマーが開始されます。タイマーがこのパラメーターの現行値になると、アイドル状態の接続はスローされます。これを接続のリープ と言います。これは、接続がアイドル状態のときにリソースを保管するために行われます。接続プールは、MQSeries ブリッジで使用される有用なデバイスです。新しい MQSeries クライアント接続を作成することは費用のかかる操作です。プール内にアイドル状態の接続がある場合、その 1

つを再利用すると、費用のかかる新規接続の入手操作を行う時間と CPU の節約になります。 MaxConnectionIdleTime の値が高くなると、アイドル状態の接続が接続プールで待機している可能性も大きくなります。しかし、何も実行しないで JVM 内のリソースを消費するクライアント接続が増えます。この値を低く設定すると、アイドル状態の接続が使用可能になっている可能性は小さくなりますが、それと同時にアイドル状態の接続の数も少なくなります。そのため、消費されるリソースが減ります。

時間は 1 分単位で表現されます。

有効な範囲は 0 <= 値 <= 720 (12 時間) で、デフォルトは 5 (分) です。

この値を 0 に設定すると、事実上、接続プールを使用しないことを意味し、MQSeries クライアント接続がアイドル状態になっているときには、リープまたは破棄が行われます。これは、このパラメーターを使用するうえで効果的な方法ではありません。

ブリッジで指定されるタイムアウトの細分度は、タイマーの細分度が `heartbeatInterval` ブリッジ・パラメーターと等しい場合に限り検査されます。

MaxConnectionIdleTime は、MQSeries Everyplace システムのシャットダウンにかかる時間に直接的な影響を与えることがあります。詳細については、145ページの『MQSeries キュー・マネージャーのシャットダウン』を参照してください。

*属性:* MQPassword

*タイプ:*

Unicode

*ラベル:*

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_PASSWORD`

*有効なアクション*

照会、作成、更新

*説明* java クライアントによって使用されます。 MQSeries 呼び出しのパスワード・フィールドが指定されていない場合は、″ ″ (ブランク) に設定されます。ここで指定する値は、使用されているデフォルトをオーバーライドします。このパラメーターの妥当性検査は行われません。

*属性:* MQQMgrProxyName

*タイプ:*

Int

*ラベル:*

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_MQ_Q_MGR_PROXY_NAME`

*有効なアクション*

照会、作成、更新、削除、開始、停止

*説明*

**注:** 管理メッセージの作成が使用されているときには、この特性は 1 回しか設定できません。その後、この特性はどのブリッジ管理オブジェクトの照会、更新、削除、開始、または停止を行う必要があるのか識別するために使用されます。

*属性:* MQUserID

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_USER_ID
```

*有効なアクション*

照会、作成、更新

*説明* java クライアントによって使用されます。 MQSeries 呼び出しのユーザー ID フィールドが指定されていない場合は、″ ″ (ブランク) に設定されます。ここで指定する値は、使用されているデフォルトをオーバーライドします。このパラメーターの妥当性検査は行われません。

*属性:* Port

*タイプ:*

Int

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_PORT
```

*有効なアクション*

照会、作成、更新

*説明* MQSeries Java クラスを使ってこの MQSeries キュー・マネージャーへの接続を作成するために使用されます。これを指定しない場合、MQSeries キュー・マネージャーは JVM と同じマシン上にあるものと見なされ、MQSeries システムとの対話には java バインディングが使用されます。

有効な範囲は 0 ～ MAXINT です。

*属性:* ReceiveExit

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_RECEIVE_EXIT
```

*有効なアクション*

照会、作成、更新

*説明* クライアント・チャネルの他方の終端で使われる出口を突き合わせるために使用されます。

このパラメーターの妥当性検査は行われません。

*属性:* Run-state

*タイプ:*
Int

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_RUN_STATE
```

*有効なアクション*
照会

*説明* 管理オブジェクトが実行中つまり使用されているか (値 =1)、それとも停止つまり使用されていないか (値 =0) を示します。オブジェクトが停止されるときには、そのプロパティーを動的に変更することができます。

*属性:* SecurityExit

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SECURITY_EXIT
```

*有効なアクション*
照会、作成、更新

*説明* クライアント・チャネルの他方の終端で使われる出口を突き合わせるために使用されます。

このパラメーターの妥当性検査は行われません。

*属性:* SendExit

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SEND_EXIT
```

*有効なアクション*
照会、作成、更新

*説明* クライアント・チャネルの他方の終端で使われる出口を突き合わせるために使用されます。

このパラメーターの妥当性検査は行われません。

*属性:* StartupRuleClass

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_STARTUP_RULE_CLASS
```

*有効なアクション*
照会、作成、更新

*説明* これは、管理オブジェクトがシステム始動時または最初の作成時にロードされたときに使用されるルール・クラスです。ルール・クラスの名前の妥当性検査は行われません。指定されたルール・クラスは、管理オブジェクトが開始されているか、およびその子が開始されているかどうかを示します。デフォルトのルールは com.ibm.mqe.mqbridge.MQeStartupRule です。このデフォルトを指定すると、管理オブジェクトが開始され、そのすべての親が始動します。このフィールドを ″ ″ (ブランク) に設定すると、管理オブジェクトは開始されません。 169ページの『MQeStartupRule』を参照してください。

*属性:* SyncQName

*タイプ:*

Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_NAME
```

*有効なアクション*

照会、作成、更新

*説明* この MQSeries キュー・マネージャーで MQSeries ブリッジによって使用される同期キューの名前。名前を構成する有効な文字は、 0〜9、 A〜Z、 a〜z、 '_'、 '.'、 '%'、 '/' です。同期キューとは、メッセージを MQSeries Everyplace から MQSeries に移動させるプロセスにおいて、メッセージの追跡のために使用される MQSeries キューです。メッセージが 1 回限りのメッセージ送達を保証する処理の途中にある場合、そのメッセージが論理のどの程度まで処理されているかを示す別のメッセージが同期キューにあります。 MQSeries Everyplace システムが完全にシャットダウンされる場合、同期キューは空になっているはずです。 MQSeries Everyplace システムが破損した場合、一部の永続状況情報は同期キューに残されます。この情報は MQSeries Everyplace システムの再始動時に使用され、システムが停止した時点から処理を継続します。同期キューの名前は、同じブリッジ上のクライアント接続については同じものにすることができます。また、その同期キューとの対話時に使用される送信、受信、セキュリティー出口が同じ場合は、別のブリッジ上にあっても同じ名前にすることができます。同期キューは、MQSeries Everyplace -> MQSeries へのメッセージ転送を処理するために、MQSeries キュー・マネージャー上になければなりません。リスナー状態クラスが MQeMQAdapter の場合、その同期キューがリスナーに関する永続状況情報を保管するためにも使用されることを意味します。そのため、メッセージを MQSeries から MQSeries Everyplace に移動させるために、同期キューはリスナー用の MQSeries に存在していなければなりません。状況情報が MQeDiskFieldsAdapter で保管されている場合、リスナーはこのパラメーターを使用しません。どのクライアント接続がどの同期キューを使用するかを理解しておくために、MQE.SYNCQ.<ClientConnectionName> という命名体系を使用することをお勧めします。

デフォルトは ″MQE.SYNCQ.DEFAULT″ です。

*属性:* SyncQPurgeInterval

*タイプ:*

int

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_PURGE_INTERVAL
```

*有効なアクション*
照会、作成、更新

*説明*

*属性:* SyncQPurgerRulesClass

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_SYNC_Q_PURGER_RULES_CLASS
```

*有効なアクション*
照会、作成、更新

*説明* 同期キューにあるメッセージが MQSeries Everyplace の障害を示すときに、メッセージを確認するために使用されるルール・クラスの名前。

デフォルトは、MQSeries Everyplace トレースの状態を報告するクラス名です。

このパラメーターの妥当性検査は行われません。

*属性:* TransformerClass

*タイプ:*
Unicode

*ラベル:*

```
com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_TRANSFORMER
```

*有効なアクション*
照会、作成、更新

*説明* これは、MQSeries メッセージを MQSeries Everyplace メッセージに変換するために使用される java クラスの名前です。メッセージがリスナーによって MQSeries から取られると、指定された変換機能を使って MQSeries Everyplace フォーマットのメッセージに変換されます。変換機能クラスが ″null″ またはブランクのストリングに指定されている場合、ブリッジ構成パラメーターで提供された DefaultTransformerClass パラメーターが変換機能として使用されます。デフォルトも null またはブランクに設定されている場合、メッセージは転送されません。

デフォルト値は ″ ″ (ブランク) です。

詳細については、162ページの『変換機能』を参照してください。

*属性:* UndelivedMessageRuleClass

*タイプ:*
Unicode

*ラベル:*

`com.ibm.mqe.mqbridge.MQeCharacteristicLabels.MQE_FIELD_LABEL_UNDELIVERED_MESSAGE_RULE_CLASS`

*有効なアクション*
照会、作成、更新

*説明* MQeUndeliveredMessageRule クラスの名前。 MQSeries から MQSeries Everyplace に移動されるメッセージを送達できないと、このルール・クラスが調べられ、リスナーがとるべきアクションが決定されます。リスナーに通知されるアクションは、待機および再試行、シャットダウン、または MQMessage レポート・オプションで定義されたメッセージ処理です。

デフォルト値は `com.ibm.mqe.mqbridge.MQeUndeliveredMessageRule` です。
168ページの『MQeUndeliveredMessageRule』を参照してください。

## テスト・メッセージを MQSeries から MQSeries Everyplace に送信する方法

メッセージの伝送をテストするために、MQSeries システムで経路指定を調整するには、多くの方法があります。 1 つの方法は、ブリッジ・セットアップ・ウィザード・ツールを使って、認知されている MQSeries Everyplace キュー・マネージャーごとにキュー・マネージャー別名を定義することです。本書では、MQSeries Everyplace キューにメッセージを送信するために、結果の構成を使用する方法を説明します。

1. MQSeries クライアント v 5.1 から、MQSeries First Steps プログラムを選択する
2. 「First Steps (最初のステップ)」画面から、API エクササイザーを選択する
3. 「API Exerciser Queue Managers (API エクササイザー・キュー・マネージャー)」画面で次のことを実行する
   - ブリッジの接続先の MQSeries キュー・マネージャーを選択する (例では MQA)
   - 「Advanced mode (拡張モード)」チェック・ボックスを選択する
   - 「MQCONN」ボタンを選択する
   - 「Queues（キュー)」タブを選択して「Queues (キュー)」画面を表示する
   - MQOPEN を選択して「MQOPEN Selectable Options (MQOPEN 選択可能オプション)」画面を表示する
4. 「MQOPEN Selectable Options (MQOPEN 選択可能オプション)」画面で次のことを実行する
   - MQOO_INPUT_AS_Q_DEF が選択されていないことを確認する
   - MQO_OUTPUT が選択されていることを確認する
   - メッセージの宛先にする MQSeries Everyplace キュー・マネージャーで、「ObjectName」フィールドにキューの名前を入力する (例では Q1)
   - メッセージの宛先にする MQSeries Everyplace キュー・マネージャーの名前を、「ObjectQMgrName」フィールドに入力する (例では ExampleQM)
   - 「OK」をクリックしてキューへの経路をオープンする

5. 「API Exerciser Queues (API エクササイザー・キュー)」画面で次のことを実行する
   - 「MQPUT」ボタンを選択して「MQPUT - Argument Options (MQPUT - 引き数オプション)」画面を表示する
6. 「MQPUT - Argument Options (MQPUT - 引き数オプション)」画面で次のことを実行する
   - メッセージを入力する
   - 「OK」をクリックしてメッセージを MQSeries Everyplace システム上の ExampleQM の Q1 に送信する

## 送達不能キュー

MQSeries Everyplace には MQSeries と同様の送達不能キューという概念があります。このキューには送達できなかったメッセージが保管されます。ただし、これらにはその使われ方に重要な違いがあります。

MQSeries では、メッセージがキュー・マネージャー A からキュー・マネージャー B へ移動しようとしたところ、A を B に接続するチャネルがそのメッセージを送達できない場合に、メッセージは*受信側*キュー・マネージャーの (B の) 送達不能キューに入れられることがあります。

MQSeries Everyplace アーキテクチャーの性質上、メッセージがキュー・マネージャー A からキュー・マネージャー B に送信されているものの、実際には送達できなかった場合に、メッセージは*送信側*キュー・マネージャーの (A の) 送達不能キューに入れられます。この動作はカスタマイズ可能なルールによって制御されます。詳細については、168ページの『MQeUndeliveredMessageRule』を参照してください。

MQSeries ブリッジの伝送キュー・リスナーは MQSeries チャネルに似ており、MQSeries 伝送キューからメッセージをプルしたり、メッセージを MQSeries Everyplace ネットワークに送達することができます。これは MQSeries Everyplace ルールに準拠しており、メッセージが送達できない場合は、*未配布メッセージ・ルール* を参照して伝送キュー・リスナーがどのように反応すべきかを判別します。ルールがメッセージ・ヘッダーのレポート・オプションを指し示し、レポート・オプションがメッセージを送達不能キューに書き込むよう指示すると、メッセージは MQSeries 送達不能キューに (送信側のキュー・マネージャーに) 入れられます。

## MQSeries ブリッジの putMessage() に関する考慮事項

アプリケーションが `putMessage` を使用しており、`confirmputMessage()` を使ってこのメッセージを確認してはならないことを指定している場合 (confirm parameter==false)、MQSeries ブリッジはメッセージを MQSeries に渡すための確実な送達論理を使用しません。むしろ、メッセージの経路指定先の MQSeries キューに対して単純な ″MQPut″ を実行します。メッセージの呼び出し側との間の MQSeries システム、ブリッジ、または MQSeries Everyplace システムのいずれか、および MQSeries システムに障害が起きた場合には、アプリケーションはメッセー

ジが送信されたかどうかを判別することはできません。その場合、アプリケーションはメッセージを再送することができるので、結果として、MQSeries キューに同じメッセージが 2 つ届くことがあります。

これが原因で問題が起きる場合、アプリケーション・プログラマーは、代わりに putMessage() および confirmputMessage() 呼び出しを使用することを選択する必要があります。 confirm parameter=true を指定した putMessage() を使用すると、ブリッジは確実な送達論理を使用して、メッセージを MQSeries システムに書き込みます。

MQSeries システムと送信側アプリケーションとの間のパスのコンポーネントに障害が発生した場合、アプリケーション・プログラマーは、メッセージが送信先に達したかどうかを判別することができなくなります。この場合、アプリケーションは元のメッセージを取って、それにブール MQeField を追加する必要があります。たとえば、次のようにします。

```
msg.putBoolean( MQe.Qos_Retry)
```

このメッセージが以前に送信されていることを示すには、 putMessage() メソッドを使用して、メッセージを (もう一度) 発行することができます。 MQSeries ブリッジはその確実な送達論理を使用して、2 つの putMessage() 呼び出しのうちの 1 つだけが、実際にメッセージを MQSeries に書き込むようにすることができます。

2 番目の putMessage() を MQe.Qos_Retry を指定せずに発行すると、同じデータを含む 1 つまたは 2 つのメッセージが宛先キュー内に達することがあります。したがって、以前にそのメッセージの送信を試みたことがある場合には、必ず Qos_Retry ブール・フィールドを使用する必要があります。

確認フラグを設定しないで putMessage() を使用し、正常な戻りコードを受け取った場合には、アプリケーションはメッセージが MQSeries キューに渡されたことを確証できたことになります。

確認フラグを設定して putMessage() を使用し、正常な戻りコードを受け取った場合、アプリケーションは、メッセージが MQSeries キューに渡されたことを確証できますが、ブリッジはメッセージに関するいくらかの情報を (同期キューに) 保持して、アプリケーションが同じメッセージを重複して送信しないようにすることができます。ブリッジは Qos_Retry ビットが設定されている場合に限って、メッセージが重複して送信されないようにします。 confirmputMessage() は、ブリッジから (同期キューから) のメッセージのこのメモリーをフラッシュします。

次の手順では、4 つのメッセージが MQSeries キューに置かれます。

| | | |
|---|---|---|
| | 新規メッセージの作成 | |
| (1) | putMessage(Confirm=Yes) | - メッセージは MQ に送達されるが、何らかの注釈が同期キューに作成される |
| | メッセージ上に再試行ビットが設定される | |
| | putMessage(Confirm=Yes) | - メッセージの注釈がすでに同期キューにあると、抑止される |
| | putMessage(Conirm=Yes) | - メッセージの注釈がすでに同期キューにあると、抑止される |

| | | |
|---|---|---|
| (2) | putMessage(Confirm=No) | - 抑止されない。confirm=yes を設定した書き込みだけが同期キューを使って抑止される。メッセージは MQSeries キューに送達される。 |
| | メッセージから再試行ビットを除去する | |
| (3) | putMessage(Confirm=Yes) | - メッセージが MQ に送信されたが、再試行ビットが設定されておらず、その同期キューを調べなかった |
| | ConfirmputMessage() | - ブリッジがそのメッセージ・メモリーを消去する |
| | メッセージ上に再試行ビットが設定される | |
| (4) | putMessage() | - メッセージが送信される |

## 変換機能

変換機能は、MQSeries Everyplace メッセージを MQSeries メッセージに変換したり、MQSeries メッセージを MQSeries Everyplace メッセージに変換したりできる Java クラスです。変換機能は MQeBaseTransformer クラスから派生します。

変換機能は、MQSeries ブリッジ構成中にいくつかの方法で指定できます。

- デフォルト変換機能は各 MQSeries ブリッジごとに指定できます。
- 変換機能は各 MQSeries ブリッジ・キューごとに指定できます。
- 変換機能は各 MQSeries 伝送キュー・リスナーごとに指定できます。

それぞれの場合に、MQSeries ブリッジ構成が Java クラス名、または Java クラスに分解する MQSeries Everyplace 別名を予測します。生成された Java クラスは MQeBaseTransformer クラスから派生するはずです。

変換機能は、メッセージ変換のすべての局面を扱うので、エンド・ユーザーによって作成され、MQSeries 固有メッセージ・フォーマット、および MQSeries Everyplace 固有メッセージ・フォーマットの間の変換のメソッドを備えていなければなりません。つまり、MQSeries および MQSeries Everyplace の間を流れるメッセージに新しいフォーマットを作成する場合はいつでも、その新しいメッセージ・フォーマットに変換機能クラスを作成するか、変更する必要があります。

これらの変更は、さまざまな方法で扱うことができます。

- すべてのメッセージ・フォーマットを変換できる、巨大な変換機能を作成する。

  これは、ある変換機能が別の機能から継承し、その機能が別の機能から継承するというようにして、変換機能の連鎖を形成する、Java の継承モデルを使用して実現されるか、または 1 つの巨大な Java クラスとして実現されます。

  このアプローチの利点は次のとおりです。

  – MQSeries ブリッジに指定されるデフォルト変換機能を変更できます。これには、すべての操作に使用する変換機能を判別するために、構成のポイントを 1 つだけ必要とします。 (MQSeries-transmission-queue-listener および MQSeries ブリッジ・キュー定義で、変換機能名が指定されるところをブランク / ヌルのままにしてください。)

  – 非常に単純なアプローチです。

このアプローチの欠点は次のとおりです。

- アプリケーションのフォーマットの変更時、または新規フォーマットの作成時に、この大きな変換機能をすべての箇所で変更し、再展開する必要があります。
- システム中のすべてのメッセージ・フォーマットを理解する、1 つの変換機能を作成することは不可能である場合があります。

- それぞれがさまざまなグループのメッセージ・フォーマットを理解し変換できる、中サイズの一連の変換機能を作成する。

  各変換機能は、特定のアプリケーションを処理する役割を果たし、MQSeries Everyplace 経路指定は、各アプリケーションが MQSeries ブリッジ・キューのセット、および MQSeries 伝送キュー・リスナーを排他的に使用するように設定できます。 MQSeries ブリッジ・キューおよび伝送キュー・リスナー上の変換機能名は、その後アプリケーション固有に設定されます。

  このアプローチの利点は次のとおりです。

  - プログラマーはメッセージの経路指定される位置を完全に制御でき、正しい変換機能が使用されることを確認できます。
  - アプローチが単純です。
  - メッセージ・フォーマットを追加または変更する場合、変換機能は変更済みの、または新しいメッセージのフローの経路に沿って変更するだけですみます。

- システムのメッセージ・フォーマットごとに個別の変換機能を作成する。

  これには、非常に小さな変換機能のリストを使用する、高レベルの変換機能を作成することが必要です (examples.mqbridge.transformers.MQeListTransformer を参照)。この機能は、メッセージを使用できる変換機能が見つかるまで、それぞれを順番に呼び出します。

  各変換機能は、メッセージ・フォーマットを 1 つずつ理解しています。

  各メッセージ・フォーマットに注意を払い、さらに、小さな変換機能のそれぞれが変換するメッセージのフォーマットを固有に識別することを確認するため、変換機能にも注意する必要があります。メッセージのインスタンスが、複数の変換機能によって変換できるようにならないようにしてください。各変換機能は各メッセージを調べて、メッセージが、変換機能が処理するように設計されたフォーマットに適合しているか判別できなければなりません。

  さまざまなリストの変換機能が、MQSeries ブリッジ構成の異なるポイントで使用されることがあります。最も基本的なレベルでは、使用可能な小さな変換機能全体の巨大なリストで、リスト変換機能を作成し、これをデフォルトに設定します。最も複雑なレベルでは、変換機能の非常に小さなリストでリスト変換機能を作成し、 MQSeries ブリッジ・キューおよび MQSeries 伝送キュー・リスナーの変換機能パラメーターを設定します。

  リスト変換機能は、Java ソース・コード自体にあるハード・コーディングされたリテラル・ストリング定数、 JVM のシステム環境変数、基礎となるオペレーティング・システム環境、リスト変換機能クラスのロード時にロードされる ASCII データ・ファイルのいずれかから、または単に、ロード時にファイル・システムで使用可能な変換機能クラスを探すことによって、そのリストを獲得します。メ

ソッドの選択は、エンド・ユーザー・プログラマーに任されます。リスト変換機能のサンプルでは、Java ソース・コードの、ハード・コーディングされた変換機能リストのメソッドを使用します。

このアプローチの利点は次のとおりです。

– このアプローチは、よりオブジェクト指向であり、単一の小さな変換機能内で完全にカプセル化される、特定のメッセージ・フォーマットについて知ることができますが、リスト変換機能は、使用可能な変換機能を理解するだけです。
– 小さな変換機能を新しく追加しても、リスト変換機能を変更する必要はありません。たとえば、リスト変換機能がファイル・システムを探してどの変換機能が使用可能かを調べる場合、単にファイル・システムの正しい位置に変換機能を追加するだけで、その変換機能が使用されるようにできます。

• 上記のメソッドをすべて混合して使用する。

## examples.mqbridge.transformers.MQeListTransformer 変換機能クラスの例

このサンプル変換機能では、それ自体はメッセージのフォーマットを理解していなくても、メッセージの変換を実行するために、高レベルの変換機能クラスが非常に小さい変換機能のリストを使用する方法を示します。

ソース・ファイルは examples¥mqbridge¥transformers¥MQeListTransformer.java で、単純な MQSeries から MQSeries Everyplace への変換機能クラスです。

この変換機能は、実際には渡されるメッセージのフォーマットを理解していません。小さい変換機能の番号付きリストがあります。メッセージを変換する必要がある場合、このクラスは変換機能のリスト中を 1 つずつ処理し、各変換機能にメッセージを提示します。最初の変換機能が変換されたメッセージを正常に戻した結果は、このクラスのユーザーに戻されます。

このスタイルの変換機能は、それぞれの小さい変換機能が限られた数のメッセージ・フォーマットを理解している、小さい変換機能の集合と結び付けて使用できます。

このクラスは変換機能のリストを静的番号付きリスト (配列) に保持しますが、活動化メソッドが呼び出される場合はファイルからリストを簡単に読み取るか、または他のいくつかのメソッドを使ってリストを獲得できます (おそらく、これを実行するために活動化メソッドで渡されるユーザー定義のパラメーターを使用します)。

サンプルを使用するには、一連の小さい変換機能を作成し、それらのクラス名をサンプル・ファイルの最上部の静的リストに入れます。生成された (高レベル) 変換機能を、ブリッジ構成の必要な場所にコンパイルし、設定します。

### 変換機能と満了時間の考慮事項

MQSeries と MQSeries Everyplace の間で満了時間を変換する場合は、特別な注意が必要です。

MQSeries Everyplace 満了時間は、メッセージの有効期限が切れた後の明示的時間か、またはメッセージ作成時刻からメッセージが期限切れになるまでの時間の長さを 1 ミリ秒単位で表したデルタのどちらかとして指定されます。

MQSeries 単位は、1/10 秒です。

変換機能がこれらの満了時間を変換できない場合、メッセージの期限が切れ、メッセージは確実に消失します。

## MQSeries スタイル・メッセージ

MQeMQMsgObject は、MQSeries Everyplace で MQSeries スタイル・メッセージをサポートする、MQeMsgObject のサブクラスです。これは、通常、MQSeries ブリッジでデフォルトの変換機能を使用して MQSeries アプリケーションとメッセージを交換するのに使用されます。これは、標準 MQSeries メッセージを受け取る際に、MQeMQMsgObject を生成します。同様に、MQSeries Everyplace アプリケーションが MQeMQMsgObject を生成して MQSeries に送信すると、ブリッジにあるデフォルトの変換機能は、これを標準 MQSeries メッセージに変換する方法を認識します。

MQeMQMsgObject クラスがご使用の要件に適合しない場合には、ご使用のアプリケーションに適合する別のタイプのメッセージ・オブジェクトを使用するブリッジで、変換機能を作成することができます。

スペースを確保するために、このクラスを使用しないシステムからは、このクラスを除去することができます。

### MQSeries スタイル・メッセージの読み取り

アプリケーションがメッセージを受け取ると、以下のようにしてこのメッセージが MQeMQMsgObject クラスのものであるかどうかを検査できます。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
  MQeMsgObject msg = MyQM.getMessage(qmgr, queue, null, null, 0);
  if (msg instanceof MQeMQMsgObject)
  {
    MQeMQMsgObject mqeMsg = (MQeMQMsgObject) msg;
    ...
  }
```

メッセージがこのクラスのものである場合、メッセージ・オブジェクトで適切な取得メソッドを使用することにより、メッセージ・データと同様に、MQSeries メッセージ・ヘッダーからのすべての情報にアクセスできます。ヘッダー情報は、形式 `getxxx()` のメソッドを使用して取得できます。ここで、*xxx* は、ヘッダー・フィールドの名前です。整合性を守るために、ヘッダー・フィールドの名前とタイプは、MQSeries Java クライアントの名前とタイプに一致したものになります。アプリケーション・データは、`getData()` メソッドを使用して取得されます。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
  if (msg instanceof MQeMQMsgObject)
  {
    MQeMQMsgObject mqeMsg = (MQeMQMsgObject) msg;
    String replyQMgr = mqeMsg.getReplyToQueueManagerName();
    String replyQueue = mqeMsg.getReplyToQueueName();
    byte [] correlId = mqeMsg.getCorrelationId();
    String msgFormat = mqeMsg.getFormat();
     ...
    byte [] data = mqeMsg.getData();
     ...
  }
```

これで、データはアプリケーションによって処理されます。 MQeMQMsgObject は、データをバイト配列として戻し、アプリケーションは、バイト配列内でデータの構造を理解しなければなりません。さらに構造化されたフォーマットのデータが必要な場合、アプリケーション・データを理解し、これを必要なフォーマットに変換する変換機能を自分自身で作成することができます。

## MQSeries スタイル・メッセージの作成

デフォルトの変換機能が理解できる MQSeries スタイル・メッセージを作成するには、新しい MQeMQMsgObject を作成し、ヘッダー・フィールドおよびデータに必要な値を設定してから、通常の方法でメッセージを送信します。

新しいメッセージ・オブジェクトを作成するには、コンストラクターを呼び出します。これにはパラメーターはありません。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
try
{
  MQeMQMsgObject mqeMsg = new MQeMQMsgObject()
  ...
}
```

setxxx() という形式のメソッドを使用して、メッセージ内に MQSeries ヘッダー情報を設定します。ここで、*xxx* はヘッダー・フィールドの名前です。整合性を守るために、ヘッダー・フィールドの名前とタイプは、MQSeries Java クライアントの名前とタイプに一致したものになります。明示的に設定されないヘッダー・フィールドは、MQSeries デフォルト値を使用します。

アプリケーション・データは、setData() メソッドを使用して設定されます。

```
import com.ibm.mqe.mqemqmessage.MQeMQMsgObject;
  ...
try
{
  MQeMQMsgObject mqeMsg = new MQeMQMsgObject()
  mqeMsg.setPutApplicationName("myApp");
  mqeMsg.setFormat(...);
  mqeMsg.setData(...);
  MyQM.putMessage(qmgr, queue, mqeMsg, null, 0);
}
```

setData() に渡される前に、データは受信側アプリケーションが理解するバイト配列にフォーマットされる必要があります。

## 変換機能の作成

MQeMQMsgObject は、MQSeries ブリッジでデフォルトの変換機能によって使用されます。これを使用する代わりとしては、MQeMsgObject を使用する変換機能またはそのサブクラスを作成することができます。

MQSeries Everyplace アプリケーションでメッセージを既存の MQSeries アプリケーションと交換するようにしたい場合には、デフォルトの変換機能を使用するとコストを低く抑えられます。この方法の主な利点は、簡単であるということです。デフォルトの変換機能はすでに利用可能になっているため、アプリケーションがしなければならないことは、MQe アプリケーションにコードを組み込むことだけです。

この方法が不利になることもあります。たとえば、

**複数の MQSeries Everyplace アプリケーションがメッセージを理解する必要がある場合**
バイト配列のデータのフォーマットは、すべてのアプリケーションが理解しなければなりません。カスタマイズされた変換機能を使用すると、変換機能にあるコードしかデータ・フォーマットを理解できなくなります。

**MQSeries Everyplace アプリケーションのコード・サイズが重要な場合**
アプリケーションのコードのサイズを最小限に抑えておかなければならない場合には、すべてのデータ・フォーマット・コードを変換機能に書き込むことができます。こうすると、アプリケーションはこれをバイト配列にフォーマットし直さなくてもデータを送受信できることになります。また、クライアント・デバイスに MQeMQMsgObject クラスがなくてもかまわなくなります。

この他の考慮事項として、MQSeries ブリッジ上で変換機能に MQSeries Java クライアントが使用可能になっていると、データをバイト配列に圧縮および解凍できます。

## MQSeries ブリッジのルール

MQSeries ブリッジは以下のようなルール・クラスを使用します。これらのルールによって、ブリッジの動作を変えることができます。

**MQeLoadBridgeRule**
このルール・クラスは、サーバーの開始時にどのブリッジをロードするかを決定します。

**MQeUndeliveredMessageRule**
このルール・クラスは、MQSeries Everyplace に送信できない MQSeries メッセージの処理方法を決定します。

**MQeSyncQueuePurgerRule**
このルール・クラスは、MQSeries Everyplace から MQSeries への古い未確認メッセージに対する処置を決定します。

**MQeStartupRule**
このルール・クラスは、管理オブジェクトが最初にロードされた時点で、このオブジェクトを開始するかどうかを決定します。

これらのクラスについて、以下のセクションでさらに詳しく説明します。プログラマーは、これらのルール・クラスをサブクラス化し、MQSeries Everyplace の動作を変えるルールを作成して MQSeries Everyplace の構成を変更することによって、デフォルト・ルール・クラスの代わりに独自のルール・クラスを使用することができます。

### MQeLoadBridgeRule

このクラスは、サーバー開始時にどのブリッジをロードするかを定義します。サーバーが `MQeMQBridge.activate()` メソッドを使用する場合には、ブリッジ・ローダーが開始します。ブリッジ・ローダーはレジストリーのすべての項目を読み取り、このルール・クラスに従って、レジストリー内の各ブリッジ名ごとに、ブリッジをロードするかどうか判断します。基本的な MQeLoadBridgeRule クラスは、レジスト

リー内のすべてのブリッジをロードさせます。単一の MQSeries Everyplace キュー・マネージャーがレジストリーを使用している限り、このようなルールが適切です。

レジストリーが複数の MQSeries Everyplace キュー・マネージャーによって共用されている場合、同じブリッジ・オブジェクトを複数のキュー・マネージャーがロードしようとすることがありますが、これは無効です。すべてのブリッジ、キュー・マネージャーおよびキューへのアクセスは、最初に開始したサーバーに対して許可され、その後で開始したサーバーはすべてロックアウトされます。このため、MQeLoadBridgeRule のカスタマイズ・バージョンを作成することによって、各サーバーがロードするブリッジを選択しておくのが適切です。特定のブリッジをロードする必要のあるサーバーの名前とそのブリッジ名とが関連するような命名規則を使用すれば、カスタマイズ・ルールの作成が容易になります。

クラス **examples.mqbridge.rules.ExampleLoadBridgeRule** は、ブリッジ・オブジェクトに命名規則を適用する方法を示します。特に LoadBridgeRule と関連付けて使用することによって、サーバーがどのブリッジをロードするかを記述する方法を示します。

## MQeUndeliveredMessageRule

1 つのブリッジに対して複数の MQSeries 伝送キュー・リスナー・オブジェクトが定義され、それらが実行されて MQSeries 伝送キューから MQSeries Everyplace ネットワークに一連のメッセージを移動する場合があります。

ある MQSeries メッセージを MQSeries Everyplace に送信できない場合、伝送キュー・リスナーのスレッドは許可メソッドを呼び出して、リスナーの構成パラメーターに示された UndeliveredMessageRule クラスを調べます。このメソッドからの戻り値は、行うべき処置を示します。

- 結果が値 ″MQeUndeliveredMessageRule.STOP_LISTENER″ であれば、このメッセージは送信不能のため、リスナーは停止しなければなりません。メッセージは MQSeries システム内の伝送キューに残ります。
- 結果が値 ″MQeUndeliveredMessageRule.USE_MQ_REPORT_OPTIONS″ であれば、元の MQSeries メッセージの当初の「report (レポート)」フィールドの値に応じて、メッセージを廃棄または MQSeries Everyplace システムの送達不能キューに移動する必要があります。 MQSeries キュー・マネージャーの送達不能キューの名前は、伝送キュー・リスナーのブリッジ・オブジェクトの構成パラメーターが示しています。この値が戻され、しかもメッセージのレポート・オプションが MQRO_DISCARD である場合には、未配布メッセージは廃棄されます。
- 0 より大きい整数値が結果として戻された場合、その整数値は、リスナーが MQSeries から MQSeries Everyplace への転送操作を再試行するまでの待機時間を表します。

上のいずれにも該当しない値が戻された場合、またはルールが例外を報告した場合には、リスナーは結果 STOP_LISTENER が戻された場合と同じ動作をします。

クラス examples.mqbridge.rules.MQeUndeliveredMessageRule は、MQSeries Everyplace ブリッジ構成が使用するデフォルト・ルールの動作を示します。このクラスが呼び出されると、失敗が連続する場合に以下の動作が行われます。

- 最初の 1 分間は、再試行まで 5 秒間待機する。
- 次の 1 分間は、再試行まで 10 秒間待機する。
- 2 分経過後から 10 分経過までの間は、再試行まで 60 秒間待機する。
- 10 分経過しても再試行が失敗する場合には、STOP_LISTENER を適用する。

伝送キュー・リスナーの動作を調整するためのクラスのもう 1 つの例に、examples.mqbridge.rules.UndeliveredMQMessageToDLQRule があります。 `permit()` メソッドによって、常に値 MQeUndeliveredMessageRule.USE_MQ_REPORT_OPTIONS が戻されます。

## MQeSyncQueuePurgerRule

同期キューは MQSeries キュー・マネージャー上にローカルに定義されたキューで、 MQSeries ブリッジによって排他的に使用されます。このキューは、確実なメッセージ送信のために使用されます。 MQSeries あての MQSeries Everyplace メッセージに関して、未確認メッセージ 1 つにつき 1 個のレコードがキューに含まれます。不安定なシステムでは、時間の経過とともに未確認メッセージのレコードが同期キューに蓄積されて、ブリッジのパフォーマンスが低下します。

クライアント接続の同期キュー除去間隔パラメーターが指定する時間間隔ごとに、クライアント接続に定義された同期キュー除去ルール・クラスが呼び出されて、古い未確認メッセージ・レコードを処理します。提供されたメッセージを削除してもよい場合、このルールはブール値 true を戻し、メッセージを残す必要がある場合は false を戻します。また、管理者はこのルールを使用して、たとえば一定時間後にメッセージ受信が確認されない場合に、アラートを発行して適切な処置を実行することもできます。

詳しくは、examples.mqbridge.rules.MQeSyncQueuePurgerRules を参照してください。

**注:** 同期キューを使用して MQSeries 伝送キュー・リスナーの状態メッセージを保管している場合、これらのメッセージはこのルールによって影響を受けません。

## MQeStartupRule

ブリッジ、プロキシー、クライアント接続、またはリスナーのいずれかのオブジェクトがサーバー開始時にロードされるとき、管理対象の各オブジェクトに関してこのルールが参照されます。こうして、管理オブジェクトを開始するか、停止状態のままにするか、およびオブジェクトの子を同時に開始するかどうかが決定されます。

`MQeStartupRule.permit(...)` メソッドからの戻り値は、管理オブジェクトを開始するかどうかを示しています。可能な戻り値、およびそれぞれの効果は以下のとおりです。

- START_NOTHING - この管理オブジェクトを開始しない。これは、管理オブジェクトに「停止」管理メッセージを送るのと同じ効果があります。
- START_PARENTS_AND_ME - この管理オブジェクト、およびそのすべての親を開始する。これは、管理オブジェクトに対して、″affect-children″ フラグ値 false とともに「開始」管理メッセージを送るのと同じ効果があります。

- START_PARENTS_AND_ME_AND_CHILDREN - この管理オブジェクト、およびそのすべての親と子を開始する。これは、管理オブジェクトに対して、″affect-children″ フラグ値 true とともに「開始」管理メッセージを送るのと同じ効果があります。

戻り値はプログラム的に制御できるため、たとえば、接続先 MQSeries システムがアクティブであれば MQSeries 伝送キュー・リスナーだけを開始するようなインテリジェント・ルールを実装することができます。

デフォルト構成ですべての管理オブジェクトに適用される com.ibm.mqe.mqbridge.MQeStartupRule は、(ソース・コードが提供されている) クラス examples.mqbridge.rules.MQeStartupRule と類似しています。これら 2 つのクラスは、常に値 START_PARENTS_AND_ME を戻します。

## 各国語サポートの考慮事項

このセクションでは、図29 の図を使用して、MQSeries Everyplace クライアント・アプリケーションから MQSeries アプリケーションへのメッセージのフローを解説します。

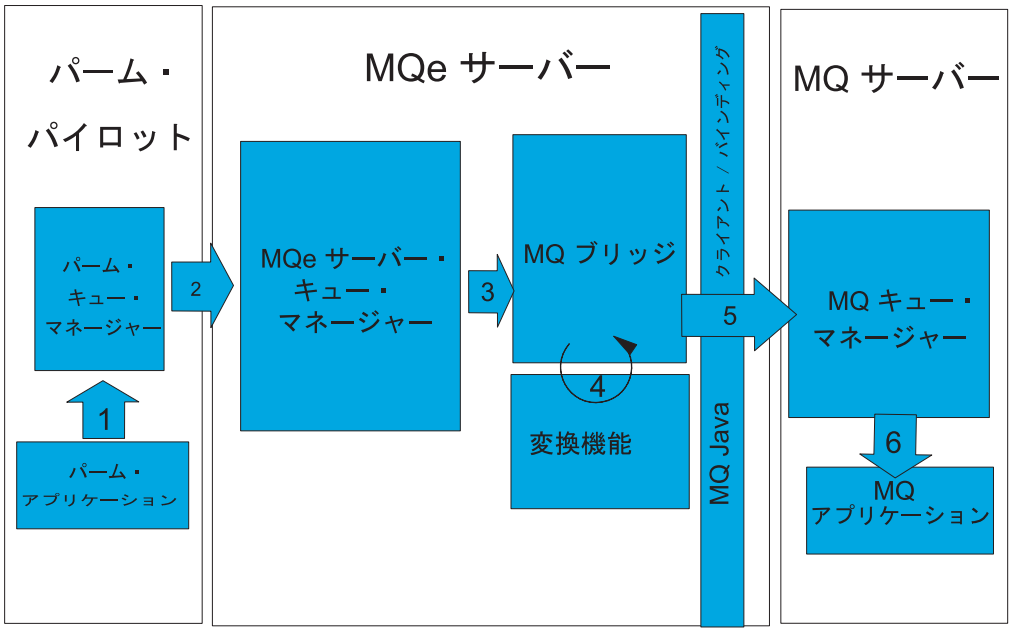


*図 29. MQSeries Everyplace から MQSeries へのメッセージ・フロー*

1. **クライアント・アプリケーション**
   a. クライアント・アプリケーションは、次のデータを含む MQSeries Everyplace メッセージ・オブジェクトを作成します。

      **ユニコード・フィールド**
      このストリングは、クライアント・マシン上で使用可能な該当するライブラリーを使用して生成されます (C/C++ を使用している場合)。

      **バイト・フィールド**
      このフィールドは翻訳してはなりません。

**ASCII フィールド**
このストリングには、ASCII 規格に準拠した、厳格に制限された有効文字が使用されます。有効文字は、すべての ASCII コード・ページ上で不変の文字だけです。

b. メッセージはパーム・キュー・マネージャーに書き込まれます。この書き込み中に、翻訳は行われません。

2. **クライアント・キュー・マネージャーは、サーバー・キュー・マネージャーに書き込みを行います。**

   このステップ中に、メッセージは翻訳されません。

3. **MQSeries Everyplace サーバーは、メッセージを MQSeries ブリッジ・キューに書き込みます。**

   このステップ中に、メッセージは翻訳されません。

4. **MQSeries ブリッジは MQSeries Everyplace メッセージをユーザー作成の変換機能に渡します。**

   変換機能は、情報を MQSeries メッセージにコピーします。変換機能はまた、MQSeries メッセージを作成します。 MQSeries Everyplace メッセージ内のユニコード・フィールドは、String value = MQemsg.GetUnicode(fieldname) を使用して検索され、MQmsg.writeChars(value) を使用して MQSeries メッセージにコピーされます。 MQSeries Everyplace メッセージ内のバイト・フィールドは、Byte value = MQemsg.getByte(fieldname) を使用して検索され、MQmsg.writeByte(value) を使用して MQSeries メッセージにコピーされます。 MQSeries Everyplace メッセージ内の ASCII フィールドは、プログラマーが MQSeries メッセージ内で作成したいのがユニコード値 (writeChars) なのか、それともコード・セット依存値 (writeString) なのかによって、MQmsg.writeChars(value) または MQmsg.writeString(value) のいずれかを使用して検索されます。 writeString() を使用する場合は、ストリングの文字セットも設定することができます (これはメッセージのメンバー変数です)。変換機能は、その結果作成された MQSeries メッセージを、呼び出し元の MQSeries ブリッジ・コードに戻します。

5. **MQSeries ブリッジは、MQSeries java クライアント / バインディング・クラスを使用して、メッセージを MQSeries に渡します。**

   MQSeries メッセージ内のユニコード値は、必要に応じて、ビッグ・エンディアンからリトル・エンディアンへ、またはその逆へ変換されます。 MQSeries メッセージ内のバイト値は、必要に応じて、ビッグ・エンディアンからリトル・エンディアンへ、またはその逆へ変換されます。 writeString() を使用して作成されたフィールドは、メッセージが MQSeries に書き込まれるときに、MQSeries java クライアント・コード内部の変換ルーチンを使用して変換されます。 ASCII データは、実行される文字セット変換にかかわりなく、ASCII データを残す必要があります。このステップ中に実行される変換は、メッセージのコード・ページ、送信 MQSeries java クライアント接続の CCSID、および受信 MQSeries サーバー接続チャネルの CCSID によって、異なる仕方で実行されます。

6. **メッセージは、MQSeries アプリケーションによって取得されます。**

   メッセージにユニコード・ストリングが含まれる場合、アプリケーションはそのストリングをユニコード・ストリングとして処理する必要があるか、またはそれを自分自身で (またはサポート・ライブラリーを使って) 他の何らかのフォーマット (UTF8 など) に変換する必要があります。メッセージにバイト・ストリングが含まれる場合、アプリケーションはそのバイトをそのまま使用することがで

きます (生データ)。メッセージにストリングが含まれる場合、それがメッセージから読み取られます。さらに、characterSet ヘッダー・フィールド内のコード・セット値に応じて、アプリケーションで必要とされる異なるデータ・フォーマット (ユニコードなど) に変換されることがあります。 Java クラスはこの機能を自動的に提供します。

### 結論

MQSeries Everyplace アプリケーションを使用しており、文字関連データを MQSeries Everyplace から MQSeries へ伝達したい場合、どのメソッドを使用するかは、伝達したいデータによっておおかた決まります。

- **ASCII 文字コード・ページの可変範囲の文字がデータに含まれる場合** (各種の ASCII コード・ページ間で変更するときに、コード・ポイントの絵文字が変更されます)、`putUnicode` (コード・ページ間の変換の対象にはなりません)、または `putArrayOfByte` (送信側のコード・ページと受信側のコード・ページとの間の変換を処理しなければならない場合) のいずれかを使用することができます。

  **注:** ASCII コード・ページの可変部分の文字が変換の対象となるときには、`putAscii()` を***使用しないでください***。
- **データに ASCII 文字コード・ページの不変範囲の文字だけが含まれる場合には、** `putUnicode` (コード・ページ間の変換の対象にはなりません)、または `putAscii` (すべてのデータが ASCII コード・ページの不変範囲内にあるときには、コード・ページ間の変換の対象になりません) を使用することができます。

## サンプル・ファイル

以下に示すサンプル・ファイルは、MQSeries ブリッジ機能をサポートする MQSeries Everyplace プログラムを作成および使用する方法を示すために提供されています。

**examples.mqbridge.awt.AwtMQBridgeServer クラス**

これは、基礎 examples.mqbridge.queuemanager.MQBridgeServer クラスへのグラフィカル・インターフェースの例です。

MQBridgeServer クラス・ソース・コードは、以下の指針に従いつつ、ブリッジ機能を MQSeries Everyplace サーバー・プログラムに追加する方法の例を示します。

ブリッジ使用可能サーバーを開始するには

1. 基本 MQSeries Everyplace キュー・マネージャーをインスタンス化して、実行を開始します。
2. 基本 MQSeries Everyplace キュー・マネージャーに渡したのと同じ .ini ファイルを渡して、com.ibm.mqe.mqbridge.MQeMQBridges オブジェクトをインスタンス化し、その `activate()` メソッドを使用します。

こうしてブリッジ機能は使用可能になります。

ブリッジ使用可能サーバーを停止するには

1. `MQeMQBridges.close()` メソッドを呼び出して、ブリッジ機能を使用不可にします。このようにすることにより、未完了のブリッジ操作すべてを完全に停止し、すべてのブリッジ機能をシャットダウンします。

2. MQeMQBridges オブジェクトをガーベッジ・コレクションされたものとして、MQeMQBridges オブジェクトへの参照をヌルにします。
3. 基本 MQSeries Everyplace キュー・マネージャーを停止およびクローズします。

**ExamplesAwtMQBridgeServer.bat**

このファイルは Awt サーバーを使用して MQBridgeServer を呼び出す方法、および AwtMQBridgeTrace モジュールの初期設定を制御する方法の例を提供します。

**ExamplesAwtMQBridgeServer.ini**

このファイルは、ブリッジ機能をサポートするキュー・マネージャー用の構成ファイルの例です。

## ブリッジの例

# 第7章 セキュリティー

このセクションでは、MQSeries Everyplace が提供するセキュリティー機能についての情報を説明します。様々なセキュリティーのレベルが、典型的な使用法のシナリオ、および使用法ガイダンスと共に説明されています。

## セキュリティー機能

MQSeries Everyplace には、統合された一群のセキュリティー機能が備えられており、メッセージ・データを、ローカルに保管するときにも転送するときにも、保護することができます。セキュリティーには、以下の 3 種類のカテゴリーがあります。

**ローカル・セキュリティー**
ローカル・セキュリティーは、MQSeries Everyplace メッセージがローカル・キュー・マネージャーに保留されている間、それを保護します。

**キュー・ベースのセキュリティー**
キュー・ベースのセキュリティーは、宛先キューが属性で定義されている限り、開始キュー・マネージャーと宛先キューの間で MQSeries Everyplace メッセージ・データを保護します。この保護は、宛先キューがローカルまたはリモート・キュー・マネージャーのどちらに属しているかには関係ありません。

**メッセージ・レベルのセキュリティー**
メッセージ・レベルのセキュリティーでは、発信側と受信側の MQSeries Everyplace アプリケーションの間で、メッセージ・データを保護します。

MQSeries Everyplace ローカルおよびメッセージ・レベルのセキュリティーは、アプリケーションで使用可能になっています。 MQSeries Everyplace キュー・ベースのセキュリティーは、内部サービスです。

3 つのカテゴリーはすべて、属性 (MQeAttribute またはそれより下のもの) を適用することにより、メッセージ・データを保護します。属性は、カテゴリーに応じて、明示的か暗黙的に適用されます。

すべての属性には、次のオブジェクトの一部またはすべてが含まれます。
- 認証機能
- 暗号機能
- 圧縮機能
- キー
- 宛先エンティティー名

これらのオブジェクトが使用される方法は、MQSeries Everyplace セキュリティーのカテゴリーに依存します。この後のセクションで、セキュリティーの各カテゴリーを詳しく説明します。

MQSeries Everyplace は、次のサービスを提供して、セキュリティーの補助も行います。

### 私用レジストリー・サービス

MQSeries Everyplace 私用レジストリーは、公開および私用オブジェクトを保管できるリポジトリーや、私用レジストリーへのアクセスが許可されたユーザーに制限されるように (ログイン) PIN 保護アクセスを提供します。また、エンティティーの秘密鍵を使用する機能、(ディジタル署名、および RSA 暗号化解除) が呼び出されるときに、それらが PrivateRegistry インスタンスをそのままにして私用認証がなくてもサポートされるように、付加的なサービスを提供します。

これらのサービスは、キュー・ベースのセキュリティー、および MQeTrustAttribute を使用するメッセージ・レベルのセキュリティーによって使用されます。

### 公開レジストリー・サービス

MQSeries Everyplace 公開レジストリーは、ミニ認証へ公的にアクセスできるリポジトリーを提供します。

これらのサービスは、キュー・ベースおよびメッセージ・レベルのセキュリティーによって使用されます。

### ミニ認証発行サービス

MQSeries Everyplace には、デフォルトのミニ*認証発行サービス* があります。これは、注意深く制御されるエンティティー名のセットに、ミニ認証を発行するように構成できます。

これらのサービスは、キュー・ベースおよびメッセージ・レベルのセキュリティーによって使用されます。

**注:** このサービスは、MQSeries Everyplace バージョン 1.0 の高セキュリティー・バージョンのみで使用可能です。

## ローカル・セキュリティー

ローカル・セキュリティーにより、MQSeries Everyplace メッセージ (MQeFields より下のもの) がローカル・キュー・マネージャーによって保持される場合に、その保護が容易になります。これを実現するためには、適切な認証機能、暗号機能、および圧縮機能を使って属性を作成し、(パスワードあるいはパスフレーズ・シードを入力することにより) 適切な鍵を設定して、そのキーを属性へ明示的に付加し、その属性を MQeMsgObject に付加します。属性の品質は、ローカルにメッセージ・データを保持するために適用されます。

選択された認証機能は、データへのアクセスが制御される方法を決定し、暗号機能の選択はデータの機密性を保護する暗号の強度を決定し、さらに圧縮機能の選択は記憶域におけるサイズの効果性を決定します。

MQSeries Everyplace には、ローカル・セキュリティーの使用を援助する MQeLocalSecure クラスが備えられていますが、どの場合にも、適切な属性を設定して、パスワードまたはパスフレーズの秘けつを提供することは、ローカル・セキュリティー・ユーザーの役割です。 MQeLocalSecure は、データを保護し、保管し、バックアップ記憶域から復元する機能を備えています。アプリケーションが MQeLocalSecure を使用しないでメッセージ (MQeFields オブジェクトより下のもの) に属性を付加する場合、ダンプを使用した後にデータを保護し、復元を使用する前

にデータを取り出すことも必要です。このアプローチの例として、バックアップ記憶域として保護トークンを使用するアプリケーションが考えられます。

MQSeries Everyplace プログラミング・リファレンス は、MQeLocalSecure の使用の単純な例を提供します。

## 使用法のシナリオ

さまざまな顧客サイトを扱う移動可能なエージェントが、ある顧客の機密データを偶発的に他の顧客と共用しないようにする場合のソリューションを考えてみましょう。ローカル・セキュリティー機能が、異なるキー、およびおそらく異なる暗号強度を使用して、指定されたマシン (たとえば PDA またはラップトップなど) で保留されている異なる顧客データを保護するため、単純なメソッドを提供します。

このシナリオを簡単に拡張すると、区域オフィスの MQSeries Everyplace サーバー・ノードにあるキュー (セキュア属性付き) から、「プル」されるキーを使って、保護されるローカル・データにアクセスします。 PDA またはラップトップのユーザーは、サーバー・キューにアクセスするためにそれ自体を認証し、ローカル・キー・データを「プル」する必要がありますが、使用されたキーは分かりません。

このアプローチを行う利点の 1 つは、顧客の特定のデータすべてにアクセスするため、監査証跡を容易に集計できるということです。

### セキュア機能の選択

MQeLocalSecure の使用時には、次の属性の選択項目が選択できます。

**認証機能**

例 NTAuthenticator または UserIdAuthenticator

**暗号機能**

対称暗号機能 MQeDESCryptor、MQe3DESCryptor、MQeRC4Cryptor、MQeRC6Cryptor または MQeMARSCryptor のいずれか

**圧縮機能**

MQeLZWCompressor、MQeRleCompressor または MQeGZIP 圧縮機能

**注:** 次のサービスは、MQSeries Everyplace バージョン 1.0 の高セキュリティー・バージョンのみで使用可能です。

- MQe3DESCryptor
- MQeRC4Cryptor
- MQeRC6Cryptor
- MQeMARSCryptor

### 選択基準

認証機能を使用するオプションは、未許可ユーザーによるローカル・データへのアクセスから保護するための、付加的な制御を提供する必要が生じたために準備されています。キー・パスワードまたはパスフレーズを提供すると、この秘けつを知っているユーザーに対してアクセスが自動的に制限されるので、認証機能を使用する必要がなくなる場合もあります。

暗号機能は、必要な保護の強度、つまり、データに不法なアクセスを行おうとして保護テキストを暗号的にハッキングするときに、ハッカーが直面する障害の程度に基づいて決定します。 128 ビットのキーを使用する対称暗号で保護されるデータは、もっと短いキーを使って暗号機能を使用する保護データより、ハッキングが難しくなります。しかし、暗号の強度に加えて、暗号機能の選択も他の様々な要因によって左右されます。例として、監査の承認を受けるため、トリプル DES を使用する必要がある金融業でのソリューションがあります。

圧縮機能を使用するオプションは、保護データのサイズを最適化する必要に応じて決定されます。しかし、圧縮機能の効率性は、データの内容に依存しています。 MQeRleCompressor は長さのエンコードを実行します。つまり、圧縮機能のルーチンが、繰り返されるバイトを圧縮したり展開したりします。したがって、何度も繰り返されるバイトでデータを圧縮または圧縮解除する際に効率的です。 MQeLZWCompressor は LZW スキームを使用します。 LZW アルゴリズムの最も単純な形式では、さまざまな語 (データ・パターン) が異なるコードに対して保管されている、ディクショナリー・データ構造を使用します。圧縮機能は、データに繰り返される語 (データ・パターン) が大量にある場合、最も効率的です。

## 使用法のガイド

1. 次のプログラム断片は、MQeLocalSecure を使用して MQeFields オブジェクトを保護します。

```
try
    {
.../* SIMPLE PROTECT FRAGMENT                                    */
.../* instantiate a DES cryptor                                  */
    MQeDESCryptor desC = new MQeDESCryptor( );
.../* instantiate an Attribute using the DES cryptor           */
    MQeAttribute desA  = new MQeAttribute( null, desC, null);
.../* instantiate a (helper) LocalSecure object               */
    MQeLocalSecure ls  = new MQeLocalSecure( );
.../* open LocalSecure obj identifying target file and directory*/
    ls.open( ".¥¥", "TestSecureData.txt" );
.../* use LocalSecure write to encode data and dump to target   */
    trace ( "i: test data in = " + "0123456789abcdef...");
    ls.write( asciiToByte( "0123456789abcdef..." ),
                                        desA, "It_is_a_secret" );
    ...
    }
  catch ( Exception e )
    {
     e.printStackTrace();...             /* show exception          */
    }
try
    {
.../* SIMPLE UNPROTECT FRAGMENT                                  */
.../* instantiate a DES cryptor                                   */
    MQeDESCryptor des2C  = new MQeDESCryptor( );
.../* instantiate an attribute using the DES cryptor             */
    MQeAttribute  des2A  = new MQeAttribute( null, des2C, null);
.../* instantiate a (a helper) LocalSecure object                */
    MQeLocalSecure ls2   = new MQeLocalSecure( );
.../* open LocalSecure obj identifying target file and directory */
    ls2.open( ".¥¥", "TestSecureData.txt" );
.../* use LocalSecure read to restore from target and decode data*/
    String outData       = byteToAscii( ls2.read( desA2,
                                          "It_is_a_secret" ) );
.../* show results....                                            */
    trace ( "i: test data out = " + outData);
    ...
    }
  catch ( Exception e )
    {
    e.printStackTrace();               /* show exception          */
    }
```

2. 次のプログラム断片は、MQeLocalSecure を使用しないで、ローカルに MQeLocalSecure を保護します。

```
try
    {
.../* SIMPLE PROTECT FRAGMENT                                    */
.../* instantiate a DES cryptor                                  */
    MQeDESCryptor desC = new MQeDESCryptor( );
.../* instantiate an Attribute using the DES cryptor           */
    MQeAttribute desA  = new MQeAttribute( null, desC, null);
.../* instantiate a base Key object                           */
    MQeKey mylocalkey  = new MQeKey( );
.../* set the base Key object local key                       */
    mylocalkey.setLocalKey( "It_is_a_secret" );
.../* attach the key to the attribute                          */
    desA.setKey( mylocalkey );
.../* activate the attribute                                   */
    desA.activateMaster( null, new MQeFields( ) );
```

```
    /* instantiate a Message object                                */
    MQeMsgObject myMsg = new MQeMsgObject( );
    /* attach the attribute to the message object                  */
    myMsg.setAttribute( desA );
    /* add some test message data                                  */
    myMsg.putAscii("testdata", "0123456789abcdef....");
    trace ("i: test data in = " + myMsg.getAscii("testdata") );
    /* encode the message                                          */
    byte[] protectedData = myMsg.dump();
    trace ("i: protected test data = byteToAscii(protectedData) );
    }
  catch ( Exception e )
    {
    e.printStackTrace();              /* show exception          */
    }

try
    {
.../* SIMPLE UNPROTECT FRAGMENT                                   */
.../* instantiate a DES cryptor                                   */
    MQeDESCryptor des2C  = new MQeDESCryptor( );
.../* instantiate an Attribute using the DES cryptor             */
    MQeAttribute  des2A  = new MQeAttribute( null, des2C, null);
.../* instantiate a base Key object                              */
    MQeKey mylocalkey2  = new MQeKey( );
.../* set the base Key object local key                          */
    mylocalkey2.setLocalKey( "It_is_a_secret" );
.../* attach the key to the attribute                            */
    des2A.setKey( mylocalkey2 );
    / instantiate a new msg object                               */
    MQeMsgObject myMsg2 = new MQeMsgObject( );
.../* activate the attribute                                     */
    des2A.activateMaster( null, new MQeFields( ) );
    /* attach the attribute to the message object                */
    myMsg2.setAttribute( des2A );
    /* decode the message data                                   */
    myMsg2.restore( protectedData );
    /* show the unprotected test data                            */
    trace ("i: test data out = " + myMsg2.getAscii("testdata") );
    }
  catch ( Exception e )
    {
    e.printStackTrace();              /* show exception          */
    }
```

## キュー・ベースのセキュリティー

キュー・ベースのセキュリティーは、宛先キューが属性で定義されている限り、開始キュー・マネージャーと宛先キューの間で MQSeries Everyplace メッセージ・データを保護します。この保護は、宛先キューがローカルまたはリモート・キュー・マネージャーのどちらに属しているかには関係ありません。

このセキュリティーの単純な例は、NTAuthenticator、MQe3DESCryptor、および MQeRleCompressor を持つ属性で定義される宛先キューです。そのような宛先キューが putMessage、getMessage、または browseMessages を使用する宛先キューを使ってアクセスされると (ローカルでもリモートでも)、キュー属性が自動的に適用されます。この例では、アクセスを開始するアプリケーションは、操作が許可される前に NTAuthenticator の要件を満たす必要があり、また操作が許可される場合、メッセージ・データは属性の MQe3DESCrryptor および MQeRleCompressor を使用して、自動的にエンコードまたはデコードされます。つまり、例の宛先キューがリモートにアクセスされると (たとえば putMessage を使用して)、キュー・ベースのセキュリ

ティーは、開始キュー・マネージャーとリモート・キュー・マネージャーの間の転送中、および宛先キューのバックアップ記憶域内の両方で、メッセージ・データがキュー属性の定義するレベルで自動的に保護されるようにします。

## 使用法のシナリオ

MQSeries Everyplace キュー・ベースのセキュリティーは、開始キュー・マネージャーと宛先キュー・マネージャーのキューの間で転送されているメッセージ・データの機密性を保護しなければならない、すべてのソリューションで使用できます。

典型的なシナリオとして、ソリューション・サービスがインターネットなどのオープン・ネットワーク経由で送達されることがあります。この場合、開始アプリケーションが PDA またはラップトップ常駐のキュー・マネージャーを使って、サーバー・キュー・マネージャー・アプリケーションが提供するサービスにアクセスします。

これは次のように実現されます。

1. 開始クライアント・キュー・マネージャー・アプリケーションが、MQSeries Everyplace メッセージで要求をカプセル化する
2. putMessage を使って、'XXX_service_request' キューを所有する、特定のリモート・サーバー・キュー・マネージャーにメッセージを確実に転送する
3. waitForMessage を使って、応答メッセージがローカル 'XXX_service_reply' キューに到達するのを待つ
4. サーバー・キュー・マネージャーが、'XXX_service_request' キューで、メッセージの聴取を行うようにセットアップする
5. メッセージ・イベントが発生すると、ローカル getMessage が実行され、サービス要求メッセージを入手する
6. 要求が処理される (たとえば、バックエンド・システムで CICS トランザクションを起動することにより)
7. 応答 (トランザクション結果) がメッセージにカプセル化される
8. putMessage を使用して、開始クライアント・キュー・マネージャーが所有する、リモート 'XXX_service_reply' キューに応答を戻す

この単純な例をサポートする 1 つの方法として、次のキューを定義します。

**開始クライアント・キュー・マネージャーが所有する (たとえば ClientQMgr)**

- TestClient_HomeServerQ
- XXX_service_reply

たくさんの選択項目がありますが、TestClient_HomeServerQ TimerInterval オプションをたとえば 5000 に設定すると、ポーリング間隔が 5 秒に設定され、クライアント・キュー・マネージャーがサーバー・キュー・マネージャーをポーリングするようにトリガーします。このポーリングは、クライアント・キュー・マネージャーに送信されている、サーバー・キュー・マネージャーの StoreAndForwardQ についてのメッセージを「プル」します。また、ClientQMgr アプリケーションを実行する前に、AddQueueManager オプションを使って ServerQMgr に参照を追加する必要があります。

**サーバー・キュー・マネージャーが所有する (たとえば ServerQMgr)**

- TestServer_StoreAndForwardQ

- XXX_service_request

このシナリオで使用するために TestServer_StoreAndForwardQ を定義するには、2 つのステップが必要です。

1. キューを作成する
2. ″ClientQMgr″ という名前を使って、setAction MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager を実行する

## セキュア機能の選択

キュー・ベースのセキュリティーを使用する際には、次の選択項目をすべて使用できます。

**認証機能**

NTAuthenticator または UserIdAuthenticator (または examples.attributes.LogonAuthenticator の下位にあるその他のもの)、あるいは MQeWTLSCertAuthenticator

**暗号機能**

MQeXORCryptor、または対象暗号機能 MQeDESCryptor、MQe3DESCryptor、 MQeRC4Cryptor、 MQeRC6Cryptor MQeMARSCryptor のいずれか

**圧縮機能**

QeLZWCompressor または MQeRleCompressor

## 選択基準

キュー・ベースのセキュリティーは、同期キューを使用するために設計されたソリューションに適しています。この場合、選択基準は実際に (同期) キュー属性の認証機能、暗号機能、および圧縮機能の選択と関係しています。

認証機能を使用するオプションは、未許可ユーザーによるローカル・データへのアクセスから保護するための、付加的な制御を提供する必要が生じたために準備されています。これは、キュー・データがローカルにアクセスされる場合もリモートにアクセスされる場合も等しく当てはまります。

LogonAuthenticator の下位にあるもの (NTAuthenticator または UserIdAuthenticator) を使用すると、属性が活動化されるとき (たとえばアプリケーションがキュー上でデータの putMessage、getMessage、または browseMessages を実行するとき)、操作が許可される前に認証機能の要件が満たされていなければなりません。キュー・ベースの 181ページの『使用法のシナリオ』では、XXX_service_request キューが NTAuthenticator を含む属性で定義される場合、サーバー XXX_service_request キューへのアクセス (たとえば putMessage 要求をクライアント・キュー・マネージャーからこのキューに送る場合) が、ターゲット・サーバーのドメインで有効な NT ユーザーとして定義される一連のユーザーに制限されます。 NTAuthenticator が例として提供され、より小さいユーザーのセットに対してより精密に細分化された制御を行える子孫を、簡単に作成できるようにします。

MQeWTLSCertAuthenticator を使用すると、この認証機能を使用する属性で保護されるキューへのすべてのリモート・アクセスが相互認証を完了してから、操作を実行することができます。交換されるミニ認証の相互認証は、受け取るミニ認証の妥当

性検査をする各参加プログラムから構成されます。この妥当性検査は、受け取ったミニ認証が、要求側の独自のミニ認証と同じミニ認証サーバーによって署名した有効な署名付きエンティティーであること、および日付が有効であること (つまり現在日付が開始日付より前であったり、終了日付より後であったりしないこと) を検査します。管理オプションによって、ソリューション作成者は、宛先キュー・マネージャーが独自の証明書 (独自のミニ認証および関連する秘密鍵を持つ独自の権限内の認証可能なエンティティー) を持つか、または所有するキュー・マネージャーの証明書を共用するかを選択することができます。キュー・ベースの 181ページの『使用法のシナリオ』では、XXX_service_request キューが MQeWTLSCertAuthenticator を含む属性で定義される場合、サーバー XXX_service_request キューへのアクセス (たとえば開始クライアント・キュー・マネージャー・アプリケーションがリモート putMessage を実行する場合) は、開始クライアント・キュー・マネージャー、および正常に相互認証された宛先 XXX_service_request キューの証明書に依存します。

暗号機能は、必要な保護の強度、つまり、不法なアクセスを行おうとして保護データを暗号的にハッキングするときに、ハッカーが直面する障害の程度に基づいて決定します。 128 ビットのキーを使用する対称暗号で保護されるデータは、もっと短いキーを使って暗号機能を使用する保護データより、ハッキングが難しくなります。しかし、暗号の強度に加えて、暗号機能の選択も他の様々な要因によって左右されます。この例として、監査の承認を受けるため、トリプル DES を使用する必要がある金融業でのソリューションがあります。

圧縮機能を使用するオプションは、保護データのサイズを最適化する必要に応じて決定されます。しかし、圧縮機能の効率性は、データの内容に依存しています。MQeRleCompressor は長さのエンコードを実行します。つまり、圧縮機能のルーチンが、繰り返されるバイトを圧縮したり展開したりします。したがって、何度も繰り返されるバイトでデータを圧縮または圧縮解除する際に効率的です。MQeLZWCompressor は LZW スキームを使用します。 LZW アルゴリズムの最も単純な形式では、さまざまな語 (データ・パターン) が異なるコードに対して保管されている、ディクショナリー・データ構造を使用します。圧縮機能は、データに繰り返される語 (データ・パターン) が大量にある場合に最も効率的です。

**注:** 次の暗号機能と認証機能は、MQSeries Everyplace バージョン 1.0 の高セキュリティー・バージョンのみで使用可能です。

- MQe3DESCryptor
- MQeRC4Cryptor
- MQeRC6Cryptor
- MQeMARSCryptor
- MQeWTLSCertAuthenticator

## 使用法のガイド

次のコード断片は、キュー・マネージャー・インスタンスを作成し、181ページの『使用法のシナリオ』で説明されているキュー・ベースのシナリオについて識別されるキューを定義する方法の例を示します。アプリケーションを開始するクライアント・キュー・マネージャー、およびサーバー・キュー・マネージャー AppRunList で開始されるアプリケーションの断片も提供されます。

## SimpleCreateQM を使用して ClientQMgr および ServerQMgr インスタンスを作成する

SimpleCreateQM は、私用レジストリーを持つキュー・マネージャー・インスタンスを作成するためにユーザーを援助します。クラスは、MQePrivateClient1.ini および MQePrivateServer1.ini の Registry セクションにあるパラメーターを使用します。

特定のインスタンスは、次のように作成されます。

1. クラスは、MQePrivateClient1.ini および MQePrivateServer1.ini の Registry セクションにある、私用レジストリー関連パラメーターを、デフォルトから必要な設定にリセットします。

   ```
   (ascii)LocalRegType=PrivateRegistry
   (ascii)DirName=.¥¥MQeNode_PrivateRegistry
   (ascii)PIN=12345678
     < change PIN from '12345678' to the PIN to be provided subsequently at
        queue manager  start-up time to enable the queue manager to access its
       own private registry >
   ```

   MQeWTLSCertAuthenticator が使用される場合のみ、3 つのキーワード (CertReqPIN、 KeyRingPassword および CAIPAddrPort) を含めてください。

   ```
   (ascii)CertReqPIN=12345678
     < change CertReqPIN from '12345678' to a new value that matches the value
        set  value defined by Mini Certertificate Server's Administrator when the
        queuemanager instance is defined >
   (ascii)KeyRingPassword=It_is_a_secret
     < change the KeyRingPassword  from 'It_is_a_secret'  to the password that
        to be subsequently provided at queuemanager start-up time to enable
        the queuemanager instance to access its protected private credentials
        within its Private Registry. >
   (ascii)CAIPAddrPort=9.20.X.YYY:8081
     < change this to the IP address and port of the solution's
         MiniCertificateServer
   ```

2. 最後の 3 つのキーワードが提供されると、自動登録が起動されるので、キュー・マネージャー・インスタンスを追加する前に、MiniCerificateServerGUI を開始する必要があります。さらに「Administration (管理)」モードを使ってキュー・マネージャー・インスタンス (ClientQMgr および ServerQMgr) を有効な認証可能エンティティーとして定義し、認証要求 PIN を前のステップの MQePrivateClient1.ini および MQePrivateServer1.ini ファイルにある、Registry セクションの CertReqPIN= 行で定義されたのと同じ値に設定することが必要です。
3. MiniCertificateServerGUI インスタンスを開始し、「Server (サーバー)」モードを選択します。
4. TestCreate プログラム (次のコード断片に示される) を実行して、キュー・マネージャー・インスタンスを作成します。

```
package test;
import com.ibm.mqe.*;
import examples.install.*;
public class TestCreate extends MQe
  {
  public void createQMs( )
    {
    /* start trace...                                                    */
    try{
```

```
      MQeTraceInterface trace =
       (MQeTraceInterface) MQe.loader.loadObject(
                                        "examples.awt.AwtMQeTrace" );
      trace.activate( "TestCreate...", null );
      }
    catch(Exception e) {e.printStackTrace(); }
    try{
      String INI_FileName   = ".¥MQePrivateClient1.ini";
      String QueueDir       = ".¥ClientQMgr¥Queues¥";
      SimpleCreateQM c_QMgr = new SimpleCreateQM();
      if ( c_QMgr.createQMgr(INI_FileName, QueueDir) )
        trace ( ">>>> ClientQMgr created OK...");
      else
        trace (">>>> error creating ClientQMgr...");
      INI_FileName          = ".¥MQePrivateServer1.ini";
      QueueDir              = ".¥ServerQMgr¥Queues¥";
      SimpleCreateQM s_QMgr = new SimpleCreateQM();
      if ( s_QMgr.createQMgr(INI_FileName, QueueDir) )
        trace ( ">>>> ServerQMgr created OK...");
      else
        trace (">>>> error creating ServerQMgr...");
      }
     catch (Exception e)
      {
      trace (">>>> SimpleCreateQM eception = "+ e.getMessage( ) );
      e.printStackTrace();
      }
    }
  public static void main(String args[])
    {
    TestCreate testc = new TestCreate( );
    testc.createQMs( );
    }
  }
```

## 前述のキュー・ベースのシナリオ用に識別されるキューの定義

キュー定義をキュー・マネージャー・インスタンスに追加するには、いくつかの方法があります。ここで説明されるメソッドは、キュー・マネージャー・インスタンスをローカルに開始し、関係のある管理メッセージを作成してキュー・マネージャーの独自の管理キューに送信することによって新しいキュー定義を追加し、その後 AdminReply キューで成功の確認を待機します。

### *ClientQMgr キュー - TestClient_HomeServerQ の追加:*

MQePrivateClient クラスを使用して ClientQMgr をローカルに開始し、異なるバージョン MQePrivateClient2.ini (これは、PIN、KeyRingPassword、および CertReqPIN のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加して、ポーリング時間間隔を設定します。

```
    {
    try{
      /* start ClientQMgr...                                          */
      String QMgrName       = "ClientQMgr";
      String QName          = "TestClient_HomeServerQ"
      MQeAttribute qattr    = new MQeAttribute(null,
                                        new MQe3DESCryptor, null);
      String FileDesc       = "MsgLog:.";
      MQePrivateClient newC = new MQePrivateClient(
                               ".//MQePrivateClient2.ini",
                               "12345678",         /* or new PIN       */
                               "It_is_a_secret",   /* or new KeyRingPwd*/
                                null);
```

```
MQeQueueManager newQM =  newC.queueManager;
/* create and use Admin msg to add HomeServerQ...                  */
MQeHomeServerQueueAdminMsg msg  =
  new MQeHomeServerQueueAdminMsg("ServerQMgr",
                                 "ServerTestQ_StoreAndForward");
MQeFields parms       = new MQeFields( );
parms.putLong( MQeHomeServerQueueAdmin.Queue_QTmerInterval, 5000 );
msg.setTargetQMgr( QMgrName );
msg.setName( QMgrName, QName );
msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
msg.putArrayOfByte( MQe.Msg_CorrelID,
         Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
MQeFields msgTest = new MQeFields( );
msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                       msg.getArrayOfByte( MQe.Msg_CorrelID ) );
parms.putAscii( msg.Queue_QMgrName, "ServerQMgr" );
parms.putAscii( msg.Queue_FileDesc, FileDesc );

if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
  {
  parms.putAscii( msg.Queue_Authenticator,
                                qattr.getAuthenticator( ).type( ) );
  if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
    {
    parms.putAscii( msg.Queue_AttrRule,
                                 "examples.rules.AttributeRule" );
    parms.putByte( msg.Queue_TargetRegistry,
                                          msg.Queue_RegistryQueue );
    }
  }
if ( qattr.getCryptor( ) != null )
  {
  parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
  if (! parms.contains( msg.Queue_AttrRule ) )
    parms.putAscii( msg.Queue_AttrRule,
                                 "examples.rules.AttributeRule" );
  }
if ( qattr.getCompressor( ) != null )
  parms.putAscii( msg.Queue_Compressor,
                               qattr.getCompressor( ).type( ) );
parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
msg.create( parms );
trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
/* use Admin msg to add HomeServerQ...                             */
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(">>> Waiting for a response to create Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                               "AdminReplyQ", msgTest, null, 0, 3000);
trace(">>> Admin Msg processed OK...");
/* process Admin msg response                 ...                  */
if ( respMsg == null )
  trace ( "i: create Queue failed, no response message received" );
else
  {
  if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
    trace( "i: create Queue added queue OK..." );
  else
    trace( "i: create Queue failed: " + respMsg.getReason( ) );
  }
newQM.close();
}
catch ( Exception e )
  {
```

```
      trace (">>>> add HomeServerQ exception = "+ e.getMessage( ) );
      e.printStackTrace();
      }
    }
```

### *ClientQMgr キュー - XXX_service_reply キューの追加:*

MQePrivateClient クラスを使用して ClientQMgr をローカルに開始し、異なるバージョン MQePrivateClient2.ini (これは、PIN、KeyRingPassword、および CertReqPIN のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加します。

```
{
    try{
      /* start ClientQMgr...                                              */
      String QMgrName       = "ClientQMgr";
      String QName          = "XXX_service_reply"
      MQeAttribute qattr    = new MQeAttribute(null,
                                        new MQe3DESCryptor, null);
      String FileDesc       = "MsgLog:.";
      MQePrivateClient newC = new MQePrivateClient(
                               ".//MQePrivateClient2.ini",
                               "12345678",         /* or new PIN       */
                               "It_is_a_secret",   /* or new KeyRingPwd*/
                                null);
      MQeQueueManager newQM =  newC.queueManager;
      /* create and use Admin msg to add XXX_service_reply queue       */
      MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
      MQeFields parms       = new MQeFields( );
      msg.setTargetQMgr( QMgrName );
      msg.setName( QMgrName, QName );
      msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
      msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
      msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
      msg.putArrayOfByte( MQe.Msg_CorrelID,
               Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
      MQeFields msgTest = new MQeFields( );
      msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                              msg.getArrayOfByte( MQe.Msg_CorrelID ) );
      parms.putAscii( msg.Queue_QMgrName, "ServerQMgr" );
      parms.putAscii( msg.Queue_FileDesc, FileDesc );
       if ( qattr.getAuthenticator( ) != null )
        {
        parms.putAscii( msg.Queue_Authenticator,
                                    qattr.getAuthenticator( ).type( ) );
        if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
          {
          parms.putAscii( msg.Queue_AttrRule,
                                       "examples.rules.AttributeRule" );
          parms.putByte( msg.Queue_TargetRegistry,
                                              msg.Queue_RegistryQueue );
          }
        }
      if ( qattr.getCryptor( ) != null )
        {
        parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
        if (! parms.contains( msg.Queue_AttrRule ) )
          parms.putAscii( msg.Queue_AttrRule,
                                       "examples.rules.AttributeRule" );
        }
      if ( qattr.getCompressor( ) != null )
        parms.putAscii( msg.Queue_Compressor,
                                   qattr.getCompressor( ).type( ) );
      parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
      msg.create( parms );
      trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
```

```
    /* use Admin msg to add queue     ...                              */
    newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
    MQeAdminMsg respMsg      = null;
    trace(">>> Waiting for a response to create Admin Msg...");
    respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                            "AdminReplyQ", msgTest, null, 0, 3000);
    trace(">>> Admin Msg processed OK...");
    /* process Admin msg response                ...                 */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e )
    {
    trace (" >>> add XXX_service_reply Q excep = "+ e.getMessage( ) );
    e.printStackTrace();
    }
 }
```

### *ServerQMgr キュー - TestServer_StoreAndForwardQ の追加:*

MQePrivateClient クラスを使用して ServerQMgr をローカルに開始し、異なるバージョン MQePrivateServer2.ini (これは、PIN、KeyRingPassword、および CertReqPIN のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加してから、リモート・キュー・マネージャー参照を追加します。

```
{
    try{
       /* start ServerQMgr, locally                                   */
       String QMgrName       = "ServerQMgr";
       String QName          = "TestServer_StoreAndForwardQ"
       MQeAttribute qattr    = new MQeAttribute(null,
                                          new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                               ".//MQePrivateServer2.ini",
                               "12345678",         /* or new PIN       */
                               "It_is_a_secret",   /* or new KeyRingPwd*/
                                null);
       MQeQueueManager newQM =  newC.queueManager;
       /* create and use Admin msg to add StoreAndForwardQ            */
       MQeStoreAndForwardQueueAdminMsg( ) msg  =
                                 new MQeStoreAndForwardQueueAdminMsg( );
       MQeFields parms       = new MQeFields( );
       msg.setTargetQMgr( QMgrName );
       msg.setName( QMgrName, QName );
       msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
       msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
       msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
       msg.putArrayOfByte( MQe.Msg_CorrelID,
                Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
       MQeFields msgTest = new MQeFields( );
       msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                              msg.getArrayOfByte( MQe.Msg_CorrelID ) );
       parms.putAscii( msg.Queue_QMgrName, QMgrName );
       parms.putAscii( msg.Queue_FileDesc, FileDesc );
       if ( qattr.getAuthenticator( ) != null )
```

```
  {
  parms.putAscii( msg.Queue_Authenticator,
                                qattr.getAuthenticator( ).type( ) );
  if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
    {
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
    parms.putByte( msg.Queue_TargetRegistry,
                                       msg.Queue_RegistryQueue );
    }
  }
if ( qattr.getCryptor( ) != null )
  {
  parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
  if (! parms.contains( msg.Queue_AttrRule ) )
    parms.putAscii( msg.Queue_AttrRule,
                                  "examples.rules.AttributeRule" );
  }
if ( qattr.getCompressor( ) != null )
  parms.putAscii( msg.Queue_Compressor,
                             qattr.getCompressor( ).type( ) );
parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
msg.create( parms );
trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
/* use Admin msg to add queue     ...                              */
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(">>> Waiting for a response to create Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                             "AdminReplyQ", msgTest, null, 0, 3000);
trace(">>> Admin Msg processed OK...");
/* process Admin msg response                 ...                 */
if ( respMsg == null )
  trace ( "i: create Queue failed, no response message received" );
else
  {
  if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
    trace( "i: create Queue added queue OK..." );
  else
    trace( "i: create Queue failed: " + respMsg.getReason( ) );
  }
/* use Admin msg to StoreAndForwardQ AddQueueManager reference  */
msg = new MQeStoreAndForwardQueueAdminMsg( );
msg.addQueueManager( "ClientQMgr" );
parms = new MQeFields( );
msg.setTargetQMgr( QMgrName );
msg.setName( QMgrName, QName );
msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
msg.putArrayOfByte( MQe.Msg_CorrelID,
        Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
MQeFields msgTest = new MQeFields( );
msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                       msg.getArrayOfByte( MQe.Msg_CorrelID ) );
parms.putAscii( msg.Queue_QMgrName, QMgrName );
parms.putAscii( msg.Queue_FileDesc, FileDesc );
msg.setAction(
       MQeStoreAndForwardQueueAdminMsg.Action_AddQueueManager );
trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
MQeAdminMsg respMsg      = null;
trace(" >>> Waiting for a response to update Admin Msg...");
respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                             "AdminReplyQ", msgTest, null, 0, 3000);
trace(">>> Admin Msg processed OK...");
/* process Admin msg response                 ...                 */
```

```
      if ( respMsg == null )
        trace ( "i: create Queue failed, no response message received" );
      else
        {
        if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
          trace( "i: create Queue added queue OK..." );
        else
          trace( "i: create Queue failed: " + respMsg.getReason( ) );
        }
      trace(" >>> StoreAndForwardQ AddQueueManager reference OK..." );
      newQM.close();
      }
  catch ( Exception e )
      {
      trace (" >>> add StoreAndForwardQ exception = "+ e.getMessage( ) );
      e.printStackTrace();
      }
```

### *ServerQMgr キュー - XXX_service_request キューの追加:*

MQePrivateClient クラスを使用して ServerQMgr をローカルに開始し、 (異なるバージョン MQePrivateServer2.ini (これは、PIN、KeyRingPassword、および CertReqPIN のハードコーディング値を意図的に保持しない) を使用して管理メッセージを作成し、さらにそれを使ってキューを追加します。

```
{
    try{
       /* start ServerQMgr...                                                  */
       String QMgrName       = "ServerQMgr";
       String QName          = "XXX_service_request"
       MQeAttribute qattr    = new MQeAttribute(null,
                                          new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                                ".//MQePrivateServer2.ini",
                                "12345678",         /* or new PIN       */
                                "It_is_a_secret",   /* or new KeyRingPwd*/
                                 null);
       MQeQueueManager newQM =  newC.queueManager;
       /* create and use Admin msg to add XXX_service_request queue     */
       MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
       MQeFields parms       = new MQeFields( );
       msg.setTargetQMgr( QMgrName );
       msg.setName( QMgrName, QName );
       msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
       msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
       msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
       msg.putArrayOfByte( MQe.Msg_CorrelID,
                Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
       MQeFields msgTest = new MQeFields( );
       msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                              msg.getArrayOfByte( MQe.Msg_CorrelID ) );
       parms.putAscii( msg.Queue_QMgrName, QMgrName );
       parms.putAscii( msg.Queue_FileDesc, FileDesc );


       if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
         {
         parms.putAscii( msg.Queue_Authenticator,
                                    qattr.getAuthenticator( ).type( ) );
         if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
           {
           parms.putAscii( msg.Queue_AttrRule,
                                      "examples.rules.AttributeRule" );
           parms.putByte( msg.Queue_TargetRegistry,
                                          msg.Queue_RegistryQueue );
```

```
          }
        }
      if ( qattr.getCryptor( ) != null )
        {
        parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
        if (! parms.contains( msg.Queue_AttrRule ) )
          parms.putAscii( msg.Queue_AttrRule,
                                    "examples.rules.AttributeRule" );
        }
      if ( qattr.getCompressor( ) != null )
        parms.putAscii( msg.Queue_Compressor,
                                  qattr.getCompressor( ).type( ) );
      parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
      msg.create( parms );
      trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
      /* use Admin msg to add XXX_service_request queue                */
      newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
      MQeAdminMsg respMsg      = null;
      trace(">>> Waiting for a response to create Admin Msg...");
      respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                                  "AdminReplyQ", msgTest, null, 0, 3000);
      trace(">>> Admin Msg processed OK...");
      /* process Admin msg response                  ...                  */
      if ( respMsg == null )
        trace ( "i: create Queue failed, no response message received" );
      else
        {
        if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
          trace( "i: create Queue added queue OK..." );
        else
          trace( "i: create Queue failed: " + respMsg.getReason( ) );
        }
      newQM.close();
      }
    catch ( Exception e )
      {
      trace (" >>> add XXX_service_request excep = "+ e.getMessage( ) );
      e.printStackTrace();
      }
   }
```

### *サーバー・キュー・マネージャー AppRunList が開始するアプリケーション:*

このセクションでは、MQePrivateServer2.ini の拡張例を提供し、ServerQMgr が開始するときに自動的に開始される AppRunList アプリケーションの追加方法を示します。また、TestService アプリケーションの例も提供します。

#### MQePrivateServer2.ini の例

```
MQePrivateServer2.ini - with AppRunList extension...
[Alias]
(ascii)EventLog=examples.log.LogToDiskFile
(ascii)Network=com.ibm.mqe.adapters.MqeTcpipHttpAdapter
(ascii)QueueManager=com.ibm.mqe.MqeQueueManager
(ascii)Trace=examples.awt.AwtMQeTrace
(ascii)MsgLog=com.ibm.mqe.adapters.MqeDiskFieldsAdapter
(ascii)FileRegistry=com.ibm.mqe.registry.MqeFileSession
(ascii)PrivateRegistry=com.ibm.mqe.registry.MqePrivateSession
(ascii)ChannelAttrRules=examples.rules.AttributeRule
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MqeSharedKey
[ChannelManager]
(int)MaxChannels=0
[Listener]
(ascii)Listen=Network::8081
(ascii)Network=Network:
```

```
(int)TimeInterval=300
[QueueManager]
(ascii)Name=ServerQMgr
(ascii)QueueStore=MsgLog:.¥MQeNode_PrivateRegistry
[Registry]
(ascii)LocalRegType=PrivateRegistry
(ascii)DirName=.¥¥MQeNode_PrivateRegistry
(ascii)PIN=not set
(ascii)CertReqPIN=not set
(ascii)KeyRingPassword=not set
(ascii)CAIPAddrPort=9.20.X.YYY:8081
[AppRunList]
(ascii)App1=test.TestService
```

### サーバー TestSevice アプリケーションの例

```
package test;
import com.ibm.mqe.*;
import com.ibm.mqe.attributes.*;
import java.util.*;
public class TestService extends MQe
   implements MQeRunListInterface, MQeMessageListenerInterface, Runnable
  {
  protected Thread applicationThread = null;
  protected MQeQueueManager thisQMgr = null;

  /* constructor                                                        */
  public TestService( ) throws Exception
    {
    }
  /* activate method                                                    */

  public Object activate( Object    owner,
                          Hashtable loadTable,
                          MQeFields setupData ) throws Exception
    {
    System.out.println(" TestService, activate, owner objref = " + owner);
    thisQMgr           = (MQeQueueManager)owner;  /* save QMgr objref   */
    applicationThread  = new Thread(
                  this, "applicationThread" ); /* create svr app thread */
    System.out.println(" TestService, activate no of active threads = " +
                                                 Thread.activeCount( ) );
    Thread t[] = new Thread[Thread.activeCount( )];
    int i = Thread.enumerate( t );
    for ( int j = 0; j < i; j++ )              /* look at svr threads   */
      System.out.println("TestService activate, active thread name = "
                                                      + t[j].getName( ) );
    applicationThread.start( );                /* start appl'n Thread.  */
    return this;
    }

  /* run method                                                         */
  public void run( )
    {
    System.out.println("TestService, Run...");
    /* add listener for XXX_service_request queue                      */
    try {
        thisQMgr.addMessageListener( this, "XXX_service_request",
                                                        new MQeFields( ) );
        }
    catch( Exception e)
        {
       e.printStackTrace();
        }
    }
```

```
/* MessageArrived event handler                                          */
/* MsgArrived event is generated when a message arrives on a queue   */
public void messageArrived( MQeMessageEvent msgEvent )
{
try {
    System.out.println(" TestService, msgEvent, messageArrived ");
    System.out.println(" TestService, msgEvent getQueueManagerName = " +
                                        msgEvent.getQueueManagerName() );
    System.out.println(" TestService, msgEvent getQueueName = " +
                                              msgEvent.getQueueName( ) );
    /* get XXX service request message                                   */
    MQeMsgObject reqmsg = thisQMgr.getMessage(
                            msgEvent.getQueueManagerName( ),
                            msgEvent.getQueueName( ),
                            msgEvent.getMsgFields( ),
                            null,
                            0);
    /* process service request here                                      */
    String reqdata   = reqmsg.getAscii("XXX_service_request_data");
    String replydata = reqdata + "_reply";
    /* build XXX_service reply message here                              */
    MQeMsgObject replymsg = new MQeMsgObject( );
    replymsg. putArrayOfByte( MQe.Msg_CorrelID,
                             reqmsg.getArrayOfByte(MQe.Msg_CorrelID ) );
    replymsg.putAscii("XXX_service_reply_data", replydata );
    System.out.println(" TestService, msgEvent putting service reply " +
                         "to ClientQMgr XXX_service_reply queue");
    /* put reply to ClientQMgr XXX_service reply queue                   */
    thisQMgr.putMessage( "ClientQMgr", "XXX_service_reply",
                                                    replymsg, null, 1 );
    }
catch ( Exception e )
    {
     e.printStackTrace();
    }
}
/* finalize method                                                       */
protected void finalize()
  {
  System.out.println("TestService, finalize...");
  applicationThread.stop( );
  applicationThread.destroy( );
  }
```

### *XXX_service_request を開始するクライアント・キュー・マネージャー・アプリケーション:*

181ページの『使用法のシナリオ』のキュー・ベースのセキュリティーのシナリオ例は、 MQeMsgObject の要求をカプセル化し、putMessage を使用することによって XXX_service_request メッセージを開始し、サーバー・キュー・マネージャーの XXX_service_request キューへ要求を確実に送達する、クライアント・キュー・マネージャー・キュー・アプリケーションについて説明します。その後、独自の XXX_service_reply キューで waitForReply を使って、サービス要求への応答を待ちます。

シナリオでは、サーバー上の TestService アプリケーションは、getMessage を使って XXX_service_request キューからサービス要求を入手することによりサービス要求を処理し、(たとえばバックエンド・トランザクションの起動によって) 要求を処理してから、応答 MQeMsgObject を構築します。さらにサーバー・キュー・マネージャー putMessage を使用して (リモート) 開始クライアント・キュー・マネージャーに応答を戻します。

サーバー・キュー・マネージャーは、メッセージを内部的に TestServer_StoreAndForwardQ に書き込みます。クライアント・キュー・マネージャーは、TestServer_StoreAndForwardQ からメッセージをプルし、それをその ClientTest_HomeServerQ で受け取ってから、意図した宛先 XXX_service_reply キューに書き込みます。

以下のクライアント・アプリケーションは、サービス要求の呼び出しと、結果の応答を処理する簡単な例を示しています。

```
package test;
import com.ibm.mqe.*;
import examples.queuemanager.*;
 public class UseTestService extends MQe
  {
  protected MQeQueueManager thisQMgr = null;
  /* serviceRequest method                                                */
  public void serviceRequest( )
    {
    /* start trace...                                                      */
    try{
       MQeTraceInterface trace =
        (MQeTraceInterface) MQe.loader.loadObject(
                                          "examples.awt.AwtMQeTrace" );
       trace.activate( "UseTestService...", null );
       }
    catch(Exception e) {e.printStackTrace();
    /* start and use Client queuemanager to put request & process reply */
    try {
        /* start Client queue manager                                     */
        MQePrivateClient newC = new MQePrivateClient(
                                       ".//MQePrivateClient2.ini",
                                       "12345678",
                                       "It_is_a_secret",
                                       null );
        MQeQueueManager newQM =  newC.queueManager;
        /* build svc request and use putMessage to put it to server     */
        MQeMsgObject msgreq   = new MQeMsgObject( );
        long thisReq_CorrelID = newQM.uniqueValue();
        msgreq.putArrayOfByte( MQe.Msg_CorrelID,
                                        longToByte( thisReq_CorrelID) );
        String reqdata        = "0123456789abcdef";
        msgreq.putArrayOfByte("XXX_service_request_data",
                                          asciiToByte(reqdata) );
        newQM.putMessage("ServerQMgr","XXX_service_reqest",msgreq,null,1);
        trace( " >>> request put to ClientQMgr,XXX_service_request q OK");
        /* field and process reply to service request                   */
        trace( " >>> waiting for reply message...");
        MQeFields msgreq_filter = new MQeFields();
        msgreq_filter.putArrayOfByte( MQe.Msg_CorrelID,
                                        longToByte( thisReq_CorrelID) );
        MQeMsgObject msgreply  = newQM.waitForMessage( newQM.getName( ),
                     "XXX_service_reply", msgreq_filter, null, 0, 3000 );
        trace(" >>> service request reply = " +
          byteToAscii(msgreply.getArrayOfByte("XXX_service_reply_data")));
        }
    catch(Exception e2) { e2.printStackTrace();}
    }
  }
  public static void main(String args[])
    {
    UseTestService testsvc = new UseTestService( );
    testsvc.serviceRequest();
    }
  }
```

## キュー・ベースのセキュリティーと自動登録のトリガー

キュー・マネージャーがリモート・キューまたは MQeWTLSCertAuthenticator を含む属性で定義されたローカル・キューにアクセスする場合、キュー・マネージャーおよびキューは認証可能なエンティティーであり、独自の証明書が必要です。

キュー・マネージャーの証明書は、自動登録を起動することによって作成されます。自動登録の起動の最も簡単な方法は、関係のあるキーワードを、キュー・マネージャーの作成時に使用される ini ファイルの Registry セクションに含めるという方法です。 ini ファイルの Registry セクションで必要なキーワードは次のとおりです。

```
(ascii)CertReqPIN=12345678
  < change CertReqPIN the default '12345678' to a new value that matches the value
     set  value defined by Mini Certertificate Server's Administrator when the
     QueueManager instance is defined >
(ascii)KeyRingPassword=It_is_a_secret
  < change the default KeyRingPassword  from 'It_is_a_secret'  to the password that
     is to be subsequently provided at QueueManager start-up time to enable
     the QueueManager instance to access its protected private credentials
     within its Private Registry. >
(ascii)CAIPAddrPort=9.20.X.YYY:8081
  < change this to the IP address and port of the solution's MiniCertificateServer
```

キューの証明書 (MQeWTLSCertAuthenticator を含む属性を持つ) も、自動登録のトリガーによって作成されます。これは、キューを追加する管理メッセージが次の場合に処理される際に、自動的に発生します。

- 所有するキュー・マネージャーがすでに自動登録済みであり、独自の証明書とソリューションのミニ認証サーバーにアクセスするのに必要なパラメーターを使用して開始している
- 所有するキュー・マネージャー名とキュー名がミニ認証サーバー・アドミニストレーターによって定義済みであり、ミニ認証要求 PIN が、所有するキュー・マネージャーを開始するのに使用される CertReqPIN 値と同じ値に設定されている
- ミニ認証サーバーが使用可能、開始済み、および「Server (サーバー)」モードである

(MQeWTLSCertAuthenticator を含む属性を持つ) キューを追加すると、キューは独自の証明書を持つか、または所有するキュー・マネージャーの証明書を共用できます。どちらになるかは、キュー作成管理メッセージの構成時に決定されます。次のコード断片は、関係のあるパラメーターとその意味を示します。

### *ServerQMgr キュー - ServerTestQWTLS2 の追加:*

次のコード断片には、次のことが適用されます。

- ミニ認証サーバー・アドミニストレーターが、認証要求 PIN =12345678 を指定して ServerQMgr+ServerTesTQWTLS2 を追加し、ミニ認証サーバーを「Server (サーバー)」モードで開始したことを前提とします。
- MQePrivateClient クラスを使用して ServerQMgr をローカルに開始し、異なるバージョン MQePrivateServer2.ini (これは、PIN、KeyRingPassword、および CertReqPIN のハードコーディング値を意図的に保持しない) を使用して) 管理メッセージを作成し、さらにそれを使って ServerTestQWTLS2 を追加します。

```
{
    try{
       /* start ServerQMgr...                                                  */
       String QMgrName       = "ServerQMgr";
       String QName          = "ServerTestQWTLS2"
       MQeAttribute qattr    = new MQeAttribute(
               new MQeWTLSCertAuthenticator(), new MQe3DESCryptor, null);
       String FileDesc       = "MsgLog:.";
       MQePrivateClient newC = new MQePrivateClient(
                               ".//MQePrivateServer2.ini",
                               "12345678",         /* or new PIN       */
                               "It_is_a_secret",   /* or new KeyRingPwd*/
                                null);
       MQeQueueManager newQM =  newC.queueManager;
       /* create and use Admin msg to add ServerTestQWTLS2 queue        */
       MQeQueueAdminMsg msg  = new MQeQueueAdminMsg( );
       MQeFields parms       = new MQeFields( );
       msg.setTargetQMgr( QMgrName );
       msg.setName( QMgrName, QName );
       msg.putInt( MQe.Msg_Style, MQe.Msg_Style_Request );
       msg.putAscii( MQe.Msg_ReplyToQ, "AdminReplyQ" );
       msg.putAscii( MQe.Msg_ReplyToQMgr, QMgrName );
       msg.putArrayOfByte( MQe.Msg_CorrelID,
                Long.toHexString( newQM.uniqueValue( ) ).getBytes( ) );
       MQeFields msgTest = new MQeFields( );
       msgTest.putArrayOfByte( MQe.Msg_CorrelID,
                              msg.getArrayOfByte( MQe.Msg_CorrelID ) );
       parms.putAscii( msg.Queue_QMgrName, QMgrName );
       parms.putAscii( msg.Queue_FileDesc, FileDesc );


       if ( qattr.getAuthenticator( ) != null ) /*add qattr auth details*/
         {
         parms.putAscii( msg.Queue_Authenticator,
                                    qattr.getAuthenticator( ).type( ) );
         if ( qattr.getAuthenticator( ).isRegistryRequired( ) )
           {
           parms.putAscii( msg.Queue_AttrRule,
                                      "examples.rules.AttributeRule" );
           /* for the Queue to have its own credentials                  */
           parms.putByte( msg.Queue_TargetRegistry,
                                           msg.Queue_RegistryQueue );
           /* for the Queue to share its host QMgr's cresdentials        */
        // parms.putByte( msg.Queue_TargetRegistry,
        //                                 msg.Queue_RegistryQMgr );
           }
         }
       if ( qattr.getCryptor( ) != null )
         {
         parms.putAscii( msg.Queue_Cryptor, qattr.getCryptor( ).type( ) );
         if (! parms.contains( msg.Queue_AttrRule ) )
           parms.putAscii( msg.Queue_AttrRule,
                                      "examples.rules.AttributeRule" );
         }
       if ( qattr.getCompressor( ) != null )
         parms.putAscii( msg.Queue_Compressor,
                                  qattr.getCompressor( ).type( ) );
       parms.putUnicode( msg.Queue_Description, "Q-based scenario Q");
       msg.create( parms );
       trace(">>> putting Admin Msg to QM/queue: "+QMgrName+"/AdminQ");
       /* use Admin msg to add ServerTestQWTLS2                         */
       newQM.putMessage(QMgrName, "AdminQ", msg, null, 0);
       MQeAdminMsg respMsg      = null;
       trace(">>> Waiting for a response to create Admin Msg...");
       respMsg = (MQeAdminMsg)newQM.waitForMessage( QMgrName,
                                  "AdminReplyQ", msgTest, null, 0, 3000);
       trace(">>> Admin Msg processed OK...");
```

```
    /* process Admin msg response                ...                */
    if ( respMsg == null )
      trace ( "i: create Queue failed, no response message received" );
    else
      {
      if ( respMsg.getRC () == MQeAdminMsg.RC_Success)
        trace( "i: create Queue added queue OK..." );
      else
        trace( "i: create Queue failed: " + respMsg.getReason( ) );
      }
    newQM.close();
    }
  catch ( Exception e ) { e.printStackTrace(); }
  }
```

### 私用レジストリーを持つキュー・マネージャーを開始するキュー・ベースのセキュリティー

キュー・マネージャーおよびそのキューが認証可能なエンティティーである場合、つまり独自の証明書を持っている場合、これらの証明書にアクセスするには、キュー・マネージャーの開始時に適切なパラメーターが必要です。

適切な ini ファイルの Registry セクションでこれらのパラメーターをハード・コーディングすることは、ソリューションの開発中に便利なメカニズムである一方、実動システムには不適切です。可能な場合はいつでも、これらのパラメーターを対話的に収集し、ファイルに保管しないでキュー・マネージャー・インスタンスを開始するために使用してください。

MQePrivateClient クラスを使用し、パラメーターを渡す (MQePrivateClient2.ini ファイルのキーワードでハード・コーディングするのではなく) MQSeries Everyplace クライアント・キュー・マネージャーが、例 187ページの『ClientQMgr キュー - XXX_service_reply キューの追加』で示されています。

## キュー・ベースのセキュリティー - チャネルの再利用

開始キュー・マネージャーと宛先キュー (同じリモート・キュー・マネージャーが所有) の間のデータを保護すると、開始キュー・マネージャーは 1 つ以上のチャネルをオープンし、保護のレベル (MQeAttribute) を、宛先キューに定義された保護のレベル (MQeAttribute) と一致するチャネルに適用します。複数のチャネルのオープンを最小限に抑えるために (たとえば宛先キューごとに 1 つ)、開始キュー・マネージャーは、選択されたチャネル属性ルールに従って、チャネルの再利用を試みます。

使用されるルールは、キュー・マネージャーの作成で使用された構成ファイル (たとえば、SimpleCreateQM が使用される前の MQePrivateClient.ini または MQePrivateServer.ini) の中の ″ChannelAttrRules″ キーワードの設定によって決まります 。デフォルトでは、以下の例を使用するように設定されます。

```
ChannelAttrRules=examples.rules.AttributeRule
```

チャネルを再利用する前に、開始キュー・マネージャーは、現行のチャネル属性が、指定された宛先キューに対するメッセージを保護するのに十分かどうかを判別します。これを行うために、現行の AttributeRule equals メソッドを使用します。このメソッドは、チャネル属性が宛先キュー属性と等しいか、またはそれ以上であるかを検査します。等しいかまたはそれ以上であればチャネルを再利用し、そうで

ない場合はチャネル属性をアップグレードすることによって、関係のある保護レベルにチャネルを動的にアップグレードしようとします。これが正常に実行されるとチャネルは再利用され、失敗すると必要な保護レベルで新しいチャネルをオープンして使用します。

アップグレード・メカニズムでは、現行の AttributeRule `permit` を使用して、チャネル属性のアップグレードが許可されるかどうかを判別します。examples.rules.AttributeRule `permit` メソッドにより、保護レベルを弱から強、または同等のレベルまでアップグレードできますが、その逆はできません。

チャネルの再利用を許可する前に、宛先キュー・マネージャーは、その現行の AttributeRule `equals` メソッドを使って、現行チャネル属性が宛先キューに適切な保護レベルを提供できるかどうかを判別します。

examples.attributes.AttributeRule が実際的なデフォルトを提供する一方、多くのソリューションに特有の、異なる動作が必要となる場合があります。 MQSeries Everyplace ベースのソリューション作成機能は、デフォルトの examples.attribute.AttributeRule を、必要な動作を定義するルールで拡張するか、または置換することによって、これを達成します。

ChannelAttrRules を設定しないで実行することも可能ですが、お勧めはできません。

## メッセージ・レベルのセキュリティー

メッセージ・レベルのセキュリティーでは、発信側と受信側の MQSeries Everyplace アプリケーション間で、メッセージ・データの保護を容易にします。メッセージ・レベルのセキュリティーは、アプリケーション層サービスです。発信側の MQSeries Everyplace アプリケーションでメッセージ・レベルの属性を作成し、 putMessage を使ってメッセージを宛先キューに入れるときに、その属性を提供する必要があります。受信側アプリケーションでは、getMessage を使用して、宛先キューからメッセージを取得するときにその属性を使えるように、適切で「一致する」メッセージ・レベルの属性をセットアップし、それを受信側のキュー・マネージャーに渡さなければなりません。

ローカル・セキュリティーの場合と同様に、メッセージ・レベルのセキュリティーでは、メッセージ (MQeFields オブジェクトの子孫) の属性のアプリケーションを活用します。発信側アプリケーションのキュー・マネージャーは、メッセージ・ダンプ・メソッドを使って putMessage を処理します。そして、(付加される) 属性の 'encodeData' メソッドでメッセージ・データを保護します。受信側アプリケーションのキュー・マネージャーは、メッセージの 'restore' メソッドを使ってアプリケーションの getMessage を処理します。そして、提供される属性の 'decodeData' メソッドで元のメッセージ・データを復元します。

### 使用法のシナリオ

メッセージ・レベルのセキュリティーは、通常、次の目的に役立ちます。

- 主に非同期キューを使用するために設計されるソリューション。
- アプリケーション・レベルのセキュリティーが重要なソリューション。つまり通常メッセージ・パスが、おそらく異なるプロトコルに接続される、複数のノード

を介するフローを含むソリューション。メッセージ・レベルのセキュリティーは、慣例としてアプリケーション・レベルで承認を管理します。つまり、他の層のセキュリティーは不必要になります。

典型的なシナリオは、複数のオープン・ネットワークを介して送達される、ソリューション・サービスです。たとえば、最初から非同期操作が予期される、モバイル・ネットワークやインターネットを介する場合です。このシナリオでは、おそらくメッセージ・データは、異なるセキュリティー機能を持つ可能性のある複数のリンクを介して流れることも考えられますが、そのセキュリティー機能は、ソリューションの所有者によって制御されたり承認されたりするとは限りません。この場合、ソリューションの所有者は大抵、メッセージ・データの機密性の承認を中間媒体に委任しないで、承認管理を直接管理し、制御するほうがよいと考えます。

MQSeries Everyplace メッセージ・レベルのセキュリティーは、発信側および受信側アプリケーションに直接制御されている方法でのメッセージ・データの強度の保護を可能にし、終端から終端、アプリケーションからアプリケーションへのメッセージ・データ転送全体にわたる機密性を確実にする機能を、ソリューション設計者に提供します。

## セキュア機能の選択

MQSeries Everyplace には、メッセージ・レベルのセキュリティーのために、次の 2 つの代替属性が備えられています。

### MQeMAttribute

これは企業間通信に適しています。ここでは、相互の信用がアプリケーション層で厳しく管理されているので、承認された第三者による介入は必要ありません。利用できる MQSeries Everyplace のすべての対称暗号機能および圧縮機能はどれでも使うことができます。ローカル・セキュリティーと同じように、属性のキーが putMessage および getMessage でパラメーターとして提供される前に、それを事前設定しておく必要があります。メッセージ・レベルの保護のために、簡単かつ強力なメソッドが備えられています。これにより、公開鍵 PKI のオーバーヘッドを要することなく、強力な暗号機能を使ってメッセージの機密性を保護することができます。

### MQeMTrustAttribute

**注:** このクラスは、MQSeries Everyplace バージョン 1.0 の高セキュリティー・バージョンのみで使用可能です。

ディジタル署名を使い、デフォルトの公開鍵 PKI を活用してディジタル・エンベロープ・スタイルの保護を提供する、さらに高機能なソリューションを備えています。これは、ISO9796 ディジタル署名 / 妥当性検査を使用し、受信側アプリケーションが、意図した送信側からメッセージが着信したことを証明できるようにします。提供された属性の暗号機能は、メッセージの機密性を保護します。 SHA1 ダイジェストはメッセージの整合性を保ち、RSA 暗号機能は、意図した受信側だけがメッセージを復元できるようにします。 MQeMAttribute の場合と同様に、利用できる MQSeries Everyplace のすべての対称暗号機能および圧縮機能はどれでも使うことができます。サイズを最適化するために、使用される認証は、WAP フォーラム WTLS 仕様で提案されている、WTLS 認証に基づいたミニ認証です。

MQSeries Everyplace のデフォルトの PKI インフラストラクチャーにより、認証 (署名の妥当性検査) に必要な情報は、相互に使用することができます。

一般的な MQeMTrustAtribute 保護メッセージは、次のような形式になります。

```
RSA-enc{SymKey}, SymKey-enc {Data, DataDigest, DataSignature}
```

ここで、

**RSA-enc:**
意図した受信側の公開鍵によって、そのミニ認証から RSA 暗号化されている

**SymKey:**
疑似乱数対称キーが生成されている

**SymKey-enc:**
SymKey で対称的に暗号化されている

**Data:** メッセージ・データ

**DataDigest:**
メッセージ・データのダイジェスト

**DigSignature:**
メッセージ・データの発信側のディジタル署名

### 選択基準

MQeMAttribute は、キー・シードの内容を管理するソリューションの所有者に完全に依存しています。キー・シードとは、データの機密性を保護するために使う対称キーを派生させるために使用されるものです。このキー・シードは、発信側と受信側の両方のアプリケーションに提供する必要があります。キー・シードは、PKI を必要としないメッセージ・データの強度を保護するために単純なメカニズムを提供する一方で、効率的な操作管理はこのキー・シードに明らかに依存しています。

MQeMTrustAttribute は、MQSeries Everyplace のデフォルト PKI の利点を活用し、メッセージ・レベルの保護のディジタル・エンベロープ・スタイルを提供します。これは流れるメッセージ・データの機密性を保護するだけでなく、その整合性を検査し、発信側は意図した受信側だけがデータにアクセスできることを確実にし、受信側はデータの発信元の妥当性検査を行うことを可能にし、こうして拒否の取り消しを効率的に管理します。

メッセージ・データの終端間機密性を保護するだけのソリューションであれば、おそらく MQeMAttrribute で十分ですが、1 対 1 (認証可能なエンティティーから認証可能なエンティティー) の転送およびメッセージ発信元の拒否の取り消しが重要なソリューションでは、MQeMTrustAttribute を選択すると良いでしょう。

## 使用法のガイド

次のコード断片は、MQeMAttribute および MQeMTrustAttribute を使って、メッセージを保護したり保護を解除したりする方法の例を示します。

## MAttribute を使用した MQSeries Everyplace メッセージ・レベルのセキュリティー

```
   /* SIMPLE PROTECT FRAGMENT                                    */
   MQeMsgObject msgObj = null;
   MQeMAttribute msgA  = null;
   long confirmId      = MQe.uniqueValue();
   try{
      trace(">>> putMessage to target Q using MQeMAttribute"
                        + " with 3DES Cryptor and key=It_is_a_secret");
      MQe3DESCryptor tdes  = new MQe3DESCryptor( );
      msgA                 = new MQeMAttribute( null, tdes, null );
      MQeKey localkey      = new MQeKey( );
      localkey.setLocalKey( "It_is_a_secret");
      msgA.setKey( localkey );
      msgObj               = new MQeMsgObject( );
      msgObj.putAscii("MsgData","0123456789abcdef....");
      newQM.putMessage( targetQMgrName, targetQName,
                                          msgObj, msgA, confirmId );
      trace(">>> MAttribute protected msg put OK...");
      }
   catch (Exception e)
      {
      trace(">>> on exception try resend exactly once....");
      msgObj.putBoolean( MQe.Msg_Resend, true );
      newQM..putMessage( targetQMgrName, targetQName,
                                           msgObj, null, confirmId );
      }

   /* SIMPLE UNPROTECT FRAGMENT                                  */
   {
   MQeMsgObject msgObj2 = null;
   MQeMAttribute msgA  = null;
   long confirmId      = MQe.uniqueValue();
   try{
      trace(">>> getMessage from target Q using MQeMAttribute" +
                        " with 3DES Cryptor and key=It_is_a_secret");
      msgA                 = new MQeMAttribute( null, null, null );
      MQeKey localkey      = new MQeKey( );
      localkey.setLocalKey( "It_is_a_secret");
      msgA.setKey( localkey );
      msgObj2              = newQM.getMessage( targetQMgrName,
                                 targetQName, null, msgA, confirmId );
      trace(">>> unprotected MsgData = "
                                     + msgObj2.getAscii(("MsgData") );
      }
   catch (Exception e)
      {                                 /* exception may have left  */
      newQM.undo( targetQMgrName,       /* message locked on queue  */
                targetQName, confirmId ); /* undo just in case        */
      e.printStackTrace( );             /* show exception reason    */
      }
...}
```

## MTustAttribute を使用した MQSeries Everyplace メッセージ・レベルのセキュリティー

```
   /* SIMPLE PROTECT FRAGMENT                                    */
   {
   MQeMsgObject msgObj      = null;
   MQeMTrustAttribute msgA  = null;
   long confirmId           = MQe.uniqueValue();
   try {
       trace(">>> putMessage from Bruce1 intended for Bruce8
          to target Q using MQeMTrustAttribute with MARSCryptor ");
       MQeMARSCryptor mars        = new MQeMARSCryptor( );
```

```
    msgA                      = new MQeMTrustAttribute(
                                          null, mars, null);
    String EntityName         = "Bruce1";
    String PIN                = "12345678";
    Object Passwd             = "It_is_a_secret";
    MQePrivateRegistry sendreg = new MQePrivateRegistry( );
    sendreg.activate( EntityName, ".//MQeNode_PrivateRegistry",
                                    PIN, Passwd, null, null );
    sendreg.setTargetRegistryName("Bruce8");
    msgA.setPrivateRegistry( sendreg );
    MQePublicRegistry pr      = new MQePublicRegistry( );
    pr.activate("MQeNode_PublicRegistry", ".//" );
    msgA.setPublicRegistry( pr );
    msgA.setHomeServer( MyHomeServer + ":8081" );
    msgObj               = new MQeMsgObject( );
    msgObj.putAscii("MsgData","0123456789abcdef....");
    newQM.putMessage( targetQMgrName, targetQName,
                                        msgObj, msgA, confirmId );
    trace(">>> MTrustAttribute protected msg put OK...");
    }
 catch (Exception e)
    {
    trace(">>> on exception try resend exactly once....");
    msgObj.putBoolean( MQe.Msg_Resend, true );
    newQM..putMessage( targetQMgrName, targetQName,
                                        msgObj, msgA, confirmId );
    }
 }

/* SIMPLE UNPROTECT FRAGMENT                                    */
{
MQeMsgObject msgObj2     = null;
MQeMTrustAttribute msgA  = null;
long confirmId           = MQe.uniqueValue();
try {
    trace(">>> getMessage from Bruce1 intended for Bruce8
       from target Q using MQeMTrustAttribute with MARSCryptor ");
    MQeMARSCryptor mars       = new MQeMARSCryptor( );
    msgA                      = new MQeMTrustAttribute(
                                          null, mars, null);
    String EntityName         = "Bruce8";
    String PIN                = "12345678";
    Object Passwd             = "It_is_a_secret";
    MQePrivateRegistry sendreg = new MQePrivateRegistry( );
    sendreg.activate( EntityName, ".//MQeNode_PrivateRegistry",
                                    PIN, Passwd, null, null );
    msgA.setPrivateRegistry( sendreg );
    MQePublicRegistry pr      = new MQePublicRegistry( );
    pr.activate("MQeNode_PublicRegistry", ".//" );
    msgA.setPublicRegistry( pr );
    msgA.setHomeServer( MyHomeServer + ":8081" );
    msgObj2              = newQM.getMessage( targetQMgrName,
                           targetQName, null, msgA, confirmId );
    trace(">>> MTrustAttribute protected msg =
                                     msgObj2.getAscii("MsgData") );
    }
 catch (Exception e)
    {                                 /* exception may have left  */
    newQM.undo( targetQMgrName,       /* message locked on queue  */
              targetQName, confirmId ); /* undo just in case        */
    e.printStackTrace( );             /* show exception reason    */
    }
 }
```

# 私用レジストリー・サービス

このセクションでは、MQSeries Everyplace が提供する私用レジストリー・サービスを説明します。

## 私用レジストリーと認証可能エンティティーの概念

相互認証に基づくミニ認証を使うキュー・ベースのセキュリティーと、ディジタル署名を使うメッセージ・レベルのセキュリティーでは、*認証可能*なエンティティーという概念が提示されています。相互認証というと、一般に 2 人のユーザー間での認証を思い浮かべますが、実際にはメッセージングにはユーザーという概念はありません。メッセージング・サービスの通常のユーザーはアプリケーションで、これらがユーザー概念を扱います。

MQSeries Everyplace は*認証の宛先* をユーザー (人) から*認証可能*なエンティティーに抽象化します。このことは、認証可能なエンティティーが人間である可能性を除外するものではありませんが、アプリケーションによって選択されて割り当てられることになります。

内部的には、MQSeries Everyplace はすべてのキュー・マネージャーを定義します。これらは、認証可能なエンティティーとして、発信側の可能性もありますし、ミニ認証に従属するサービスの宛先の可能性もあります。また、MQSeries Everyplace では、認証機能に基づいてミニ認証を使うよう定義されたキューを、認証可能なエンティティーとしても定義します。したがって、これらのサービスをサポートするキュー・マネージャーは、1 つの認証可能なエンティティー (キュー・マネージャーだけ) か、一群の認証可能なエンティティー (キュー・マネージャーと、証明書に基づく認証機能を使うそれぞれのキュー) を持つことができます。

MQSeries Everyplace は、構成可能なオプションを提供して、キュー・マネージャーとキューが、認証可能なエンティティーとして自動登録することを可能にします。MQSeries Everyplace 私用レジストリー・サービス (MQePrivateRegistry) は、MQSeries Everyplace アプリケーションが認証可能なエンティティーを自動登録し、結果の証明書を管理するためのサービスを提供します。

認証可能なエンティティーとして登録されたアプリケーションはすべて、MQeMTrustAttribute で保護されたメッセージ・レベルのサービスの発信側としても受信側としても使うことができます。

### 私用レジストリーと認証可能エンティティーの証明書

便利なことに、認証可能なエンティティーごとにそれぞれの証明書が必要とされています。これにより 2 つの問題点が生じます。 1 つ目は、証明書を入手するためにどのように登録を実行するかということ、そして 2 つ目は、その証明書を安全な方法でどこに管理するかということです。 MQSeries Everyplace 私用レジストリー・サービスは、これらの 2 つの問題を解決するのに役立ちます。これらのサービスを使えば、安全な方法でその証明書を作成する認証可能なエンティティーの自動登録を起動すると同時に、安全なリポジトリーを提供できます。

私用レジストリー (基本レジストリーの子孫) は、基本レジストリーに保護または暗号トークンの特徴の機能を追加します。たとえば、公開オブジェクト (ミニ認証) と私用オブジェクト (秘密鍵) 用の保護リポジトリーになります。また、許可ユーザー

に、私用オブジェクトへのアクセスを制限するメカニズムを提供します。さらに、私用オブジェクトが私用レジストリーから漏れないようにするサービス (たとえば、ディジタル署名、RSA 暗号化解除など) をサポートしています。また、共通インターフェースを提供することによって、基礎となるデバイス・サポートを見分けることができないようにします。

### 自動登録

MQSeries Everyplace には、自動登録をサポートするデフォルトのサービスがあります。これらのサービスは、認証可能なエンティティーが構成されると、自動的に起動します。たとえば、キュー・マネージャーの開始時、新しいキューの定義時、さらに MQSeries Everyplace アプリケーションが MQePrvateRegistry を直接使用して、認証可能なエンティティーを新しく作成する場合などです。登録が起動すると、新しい証明書が作成されて、認証可能なエンティティーの私用レジストリーに格納されます。自動登録のステップには、新しい RSA キーのペアを生成すること、秘密鍵を私用レジストリーで保護し、そこに保管すること、そして、デフォルトのミニ認証サーバーに対する新しい認証要求に公開鍵をパッケージすることが含まれます。ミニ認証サーバーが構成されて使用可能になっており、認証可能なエンティティーが (証明書を持つ権限を持つ) ミニ認証サーバーによって事前登録されている場合、ミニ認証サーバーはすでに所有しているミニ認証と共に、認証可能なエンティティーの新しいミニ認証を戻し、これらの認証は、エンティティーの新しい証明書として、保護された秘密鍵と一緒に認証可能なエンティティーの私用レジストリーに格納されます。

自動登録は、認証可能なエンティティーの証明書を設定する簡単なメカニズムですが、メッセージ・レベルの保護をサポートするために、エンティティーは、独自の証明書 (ディジタル署名を容易にする) と、意図した受信側の公開鍵 (ミニ認証) にアクセスする必要もあります。

## 使用法のシナリオ

MQSeries Everyplace の私用レジストリーの基本的な目的は、MQSeries Everyplace の認証可能なエンティティーの証明書に私用リポジトリーを提供することです。認証可能なエンティティーの証明書は、エンティティーのミニ認証 (エンティティーの公開鍵をカプセル化する) と、およびエンティティーの (鍵リングで保護された) 秘密鍵から成っています。

典型的な使用法のシナリオは、他の MQSeries Everyplace セキュリティー機能に関連付けて考慮する必要があります。

**MQeWTLSCertAuthenticator を使用するキュー・ベースのセキュリティー**

キュー・ベースのセキュリティーが使用されており、キュー属性が MQeWTLSCertAuthenticator で定義されている場合 (ミニ認証ベースの相互認証) は、関係する認証可能なエンティティーは MQSeries Everyplace に所有されています。そのようなキューでメッセージにアクセスするのに使用されるキュー・マネージャー、そのようなキューを所有するキュー・マネージャー、およびキュー自体は、すべて認証可能なエンティティーであり、独自の証明書を持つ必要があります。正しい構成オプションを使用し、MQSeries Everyplace ミニ認証発行サービスのインスタンスを使用することによって、キュー・マネージャーとキューの作成時に自動登録が起動され、新しい証明書を作成してエンティティーの独自のレジストリーに保管します。

**MQeMTrustAttribute を使用するメッセージ・レベルのセキュリティー**

メッセージ・レベルのセキュリティーを MQeMTrustAttribute で使用する場合はいつでも、MQeMTrustAttribute で保護されたメッセージの発信側と受信側は、独自の証明書を持っている必要のある、アプリケーションが所有する認証可能なエンティティーです。この場合、アプリケーションは MQePrivateRegistry のサービス (および MQSeries Everyplace ミニ認証発行サービスのインスタンス) を使って、自動登録を起動し、エンティティーの証明書を作成してエンティティーの独自の私用レジストリーに保管する必要があります。

### セキュア機能の選択

MQSeries Everyplace バージョン 1 では、認証可能なエンティティーの証明書の代替セキュア・リポジトリーをサポートしません。 MQeWTLSCertAuthenticator を使用するキュー・ベースのセキュリティー、または MQeMTrustAttribute を使用するメッセージ・レベルのセキュリティーが使用される場合、私用レジストリー・サービスを使う必要があります。

### 選択基準

私用レジストリーの選択基準は、キュー・ベースおよびメッセージ・レベルのセキュリティーの選択基準と同じです。

## 使用法のガイド

キュー・ベースのセキュリティーを使用する前に、MQSeries Everyplace が所有する認証可能なエンティティーには証明書がなければなりません。これは、正しい構成を完了して、キュー・マネージャーの自動登録を起動することによって行われます。それには次のステップが必要です。

1. MQSeries Everyplace ミニ認証発行サービスのインスタンスをセットアップし、開始します。
2. 「Administration (管理)」モードで、キュー・マネージャーの名前を有効な認証可能エンティティーとして追加し、さらにエンティティーの一時使用の認証要求 PIN を追加します。
3. 「Server (サーバー)」モードでミニ認証サーバーを開始します。
4. `MQePrivateClient1.ini` および `MQePrivateServer1.ini` を、「SimpleCreateQM を使用して ClientQMgr および ServerQMgr インスタンスを作成する」の説明に従って構成し、キュー・マネージャーが SimpleCreateQM を使って作成されるときに自動登録が起動するようにします。このセクションでは、`ini` ファイルの Registry セクションで必要なキーワード、およびエンティティーの一時使用の認証要求 PIN を使用する位置を説明します。

メッセージ・レベルのセキュリティーを使用して、MQeMTrustAttribute を使ってメッセージを保護する前に、アプリケーションは私用レジストリー・サービスを使って、発信側および受信側エンティティーが必ず証明書を持つようにする必要があります。それには次のステップが必要です。

1. MQSeries Everyplace ミニ認証発行サービスのインスタンスをセットアップし、開始します。
2. 「Administration (管理)」モードで、アプリケーション・エンティティーの名前を追加し、エンティティーを一時使用の認証要求 PIN に割り振ります。

3. 「Server (サーバー)」モードでミニ認証サーバーを開始します。
4. 以下のプログラム断片と同様のプログラムを使って、アプリケーション・エンティティーの自動登録を起動します。これによって、エンティティーの証明書が作成され、私用レジストリーに保管されます。

```
/* SIMPLE MQePrivateRegistry FRAGMENT                              */
try
   {
   /* setup PrivateRegistry parameters                             */
   String EntityName               = "Bruce";
   String EntityPIN                = "11111111";
   Object KeyRingPassword          = "It_is_a_secret";
   Object CertReqPIN               = "12345678";
   Object CAIPAddrPort             = "9.20.X.YYY:8081";
   /* instantiate and activate a Private Registry.                 */
   MQePrivateRegistry preg         = new MQePrivateRegistry( );
   preg.activate( EntityName,        /* entity name               */
                  ".//MQeNode_PrivateRegistry", /* directory root  */
                  EntityPIN,         /* private reg access PIN     */
                  KeyRingPassword,   /* private credential keyseed */
                  CertReqPIN,        /* on-time-use Cert Req PIN   */
                  CAIPAddrPort );    /* addr and port MiniCertSvr  */
   trace(">>> PrivateRegistry activated OK ...");
   }
catch (Exception e)
   {
   e.printStackTrace();
   }
```

# 公開レジストリー・サービス

このセクションでは、MQSeries Everyplace が提供する公開レジストリー・サービスを説明します。

MQSeries Everyplace には、MQSeries Everyplace ノード間で認証可能なエンティティーの*公開認証* (ミニ認証) の共用を容易にする、デフォルトのサービスがあります。これらのミニ認証へのアクセスは、メッセージ・レベルのセキュリティーの前提条件です。 MQSeries Everyplace 公開レジストリー (および基本レジストリーの子孫) は、ミニ認証へ公的にアクセスできるリポジトリーを提供します。これは、携帯電話の個人電話番号登録簿サービスに似ています。異なる点は、電話番号の代わりに、認証可能なエンティティーのミニ認証セットが使われる点です。 MQSeries Everyplace 公開レジストリーは純粋な受動サービスではありません。保持していないミニ認証を提供するようにアクセスするときに、公開レジストリーが有効なホーム・サーバーで構成されている場合には、その公開レジストリーは、ホーム・サーバーの公開レジストリーから、要求されたミニ認証を自動的に入手しようとします。また、ミニ認証を他の MQSeries Everyplace ノードの公開レジストリーと共用するためのメカニズムも備わっています。これらのサービスがまとまって、自動化されたインテリジェントなミニ認証複写サービスを構成します。これにより、必要なときに必要なミニ認証を使うことができます。

## 使用法のシナリオ

公開レジストリーの典型的なシナリオは、特定の MQSeries Everyplace ノードの公開レジストリーが、最も頻繁に必要とされるミニ認証が使用されるときにそれを保管して構築するように、これらのサービスを使用することです。

この単純な例は、MQSeries Everyplace クライアント・ノードが、MQSeries Everyplace ホーム・サーバーから、必要な他の認証可能なエンティティーのミニ認証を自動的に入手し、その後、それらを公開レジストリーに保管することです。

### セキュア機能の選択

異なる MQSeries Everyplace ノードの公開レジストリー間でミニ認証を共用し、それを入手するための公開レジストリーのアクティブな機能を使用するかどうかは、ソリューション作成者の選択に任されています。

このインテリジェントな複写の代替方法は、帯域外ユーティリティーに、必要なすべてのミニ認証を含む MQSeries Everyplace ノードの公開レジストリーを初期化させてから、それらを使用するすべてのセキュア・サービスを使用可能にすることです。

### 選択基準

MQSeries Everyplace ノードの公開レジストリーで使用可能なミニ認証セットの帯域外初期化には、ソリューションが主に非同期である場合に、公開レジストリーのアクティブな機能を使用するよりも利点の多いことがあります。一方、MQSeries Everyplace ノードのホーム・サーバーへの同期接続が難しい場合もあります。しかし、この接続が使用可能であると考えられる場合、公開レジストリーのアクティブなミニ認証複写サービスは、MQSeries Everyplace ノード公開レジストリーで最も役立つミニ認証セットを自動的に保持するのに役立つツールです。

## 使用法のガイド

```
/* SIMPLE MQePublicRegistry shareCertificate FRAGMENT                  */
try {
    String EntityName       = "Bruce";
    String EntityPIN        = "12345678";
    Object KeyRingPassword  = "It_is_a_secret";
    Object CertReqPIN       = "12345678";
    Object CAIPAddrPort     = "9.20.X.YYY:8081";
    /* auto-register Bruce1, Bruce2...Bruce8                            */
    int i                   = 1;
    for ( i = 1; i < 9; i++ )
      {
      EntityName = "Bruce" + (new Integer(i)).toString( );
      MQePrivateRegistry preg = new MQePrivateRegistry( );
      preg.activate( EntityName, ".¥¥MQeNode_PrivateRegistry" ,
      EntityPIN, KeyRingPassword, CertReqPIN, CAIPAddrPort);
      /* inst'ate and activate PublicReg & save MiniCert from PrivReg  */
      MQePublicRegistry pubreg = new MQePublicRegistry( );
      pubreg.activate(  "MQeNode_PublicRegistry", ".¥¥" );
      pubreg.putCertificate( EntityName,
                             preg.getCertificate( EntityName ) );
      /* before share of MiniCert                                      */
      pubreg.shareCertificate( EntityName,
              preg.getCertificate( EntityName ),  "9.20.X.YYY:8081" );
      preg.close();
      pubreg.close();
      }
    }
catch (Exception e)
    {
    e.printStackTrace( );
    }
```

**注:** 公開レジストリー・インスタンスは複数回活動化することができないため、上記の例では、公開レジストリーのアクセスの推奨例を示しています。つまり、MQePublicRegistry の新しいインスタンスを作成し、そのインスタンスを活動化してから、必要な操作を実行してインスタンスをクローズするという方法です。

# ミニ認証発行サービス

MQSeries Everyplace には、デフォルトのミニ*認証発行サービス* が含まれます。これは、私用レジストリー自動登録要求を満たすように構成できます。付属するツールを使うことにより、特定のソリューションで、ミニ認証発行サービスを設定して管理し、こうして注意深く制御された一群のエンティティー名にミニ認証を発行することができます。この発行サービスには、以下のような特性があります。

- 登録された認証可能なエンティティーのセットの管理
- ミニ認証の発行 (ミニ認証は、WAP WTLS ミニ認証に基づく)
- ミニ認証リポジトリーの管理

付属するツールを使うことにより、ミニ認証発行サービスの管理者は、エンティティー名と登録アドレスを登録し、一時使用の*認証要求 PIN* を定義することにより、エンティティーへのミニ認証発行を許可することができます。通常、これは要求側の認証性の妥当性検査をオフラインで実行した後に行われれます。認証要求 PIN は、意図したユーザーに通知できます (新しいキャッシュ・カードが発行されるときに、キャッシュ・カード PIN が通知されるのと同様です)。次に私用レジストリーのユーザー (たとえば、MQSeries Everyplace アプリケーションや MQSeries Everyplace キュー・マネージャー) を構成し、起動時にこの認証要求 PIN を提供するように設定することができます。私用レジストリーが自動登録を起動すると、ミニ認証発行サービスでは、生成される新しい認証要求を妥当性検査し、新しいミニ認証を発行してから、登録済みの認証要求 PIN をリセットして、それを再利用できないようにします。新しいミニ認証要求の自動登録はすべて、安全なチャネル経由で処理されます。

ミニ認証発行サービスによって発行されているミニ認証は、発行サービスのレジストリーに保管されます。ミニ認証が再発行されると (たとえば、有効期限切れなどで)、有効期限が切れたミニ認証は保存されます。

## ミニ認証発行サービス・サーバーのインスタンスの構成、開始、および終了

### MQSeries EveryplaceMiniCertificateServer.ini を使用する構成

`MQeMiniCertificateServer.ini` は構成ファイルの例です。この例を変更し、MQeMiniCertificateServer 起動時にそれを使用することによって、MQeMiniCertificateServer のインスタンスを作成できます。`MQeMiniCertificateServer.ini` には、Alias、ChannelManager、Listener および MiniCertSvrRegistry セクションが含まれます。 MQeMiniCertificateServer のインスタンスは、動作の自動構成の起動時に、これらのセクションの内容を使用します。

MQeMiniCertificateServer.ini は、ExamplesMQeServer.ini を拡張したものです。拡張については、ここで説明されています。その他のオプションについては、ExamplesMQeServer.ini の説明を参照してください。

**[Alias] セクションの拡張**

次の 2 つの必須キーワードが追加されます。

**MiniCertSvrRegistry**

この設定は、使用されるレジストリーのクラス名を識別します。

**MiniCertIssuanceManager**

この設定は、MQeMiniCertIssuanceInterface をインプリメントするクラスの名前を識別します。

**追加の [MiniCertServerRegistry] セクション**

このセクションには、次に 2 つのオプション・キーワードが含まれます。

**PIN** これは、有効な MQeMiniCertificateServer Administrator の PIN を識別します。これは、MQeMiniCertificateServer によって、その私用レジストリーへのアクセスを活動化し、入手するために使用されるものです。

**KeyRingPassword**

これは、MiniCertificateServer の私用レジストリーに保管される私用オブジェクトを保護するために使用される、パスワードまたはパスフレーズを識別します。

## MQeMiniCertificateServerGUI の開始

MQeMiniCertificateServerGUI.bat は、起動ファイルの例です。MQeMiniCertificateServer のインスタンスは、この例を変更して使用することによって開始できます。この例では次のコマンドを使用します。

```
java  com.ibm.mqe.server.MQeMiniCertificateServer <parameter1> <parameter2>
```

ここで、

<parameter1> = com.ibm.MQe.Server.MCSMessageBundle
(または MQeMiniCertificateServer メッセージの ListResourceBundle を変換したバージョン)

<parameter2> = Examples.Trace.MQeTraceResource
(または MQSeries Everyplace ベースのメッセージの ListResourceBundle を変換したバージョン)

## GUI を使用してミニ認証発行サービスを初めて開始する

MQeMiniCertificateServerGUI.bat を呼び出すと、次のような結果が表示されます。

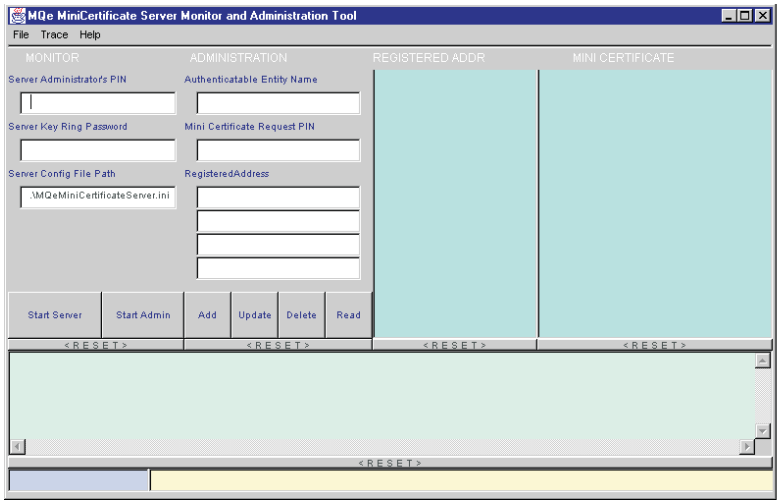


*図 30. ミニ認証サーバー GUI*

ミニ認証サーバーを初めて起動するためには、管理者は次のことを実行する必要があります。

1. 「ServerAdministrator's PIN (サーバー管理者の PIN)」入力フィールドに、ミニ認証サーバーのこのインスタンスへのアクセスに使用する PIN を入力します (ここでは、'12345678' になっています)。
2. 管理者がミニ認証サーバーのレジストリーの私用オブジェクトを保護するために使用するパスワードまたはパスフレーズを、「ServerKey Ring Password (サーバー・キー・リング・パスワード)」フィールドに入力します (ここでは、'It_is_a_secret' になっています)。
3. 起動構成ファイルのパスおよびファイル名を、「Server Config File Path (サーバー構成ファイル・パス)」フィールドに入力します (ここでは './MQeMiniCertificateServer.ini' になっています)。
4. 「Start Server (サーバーの開始)」ボタンをクリックします。

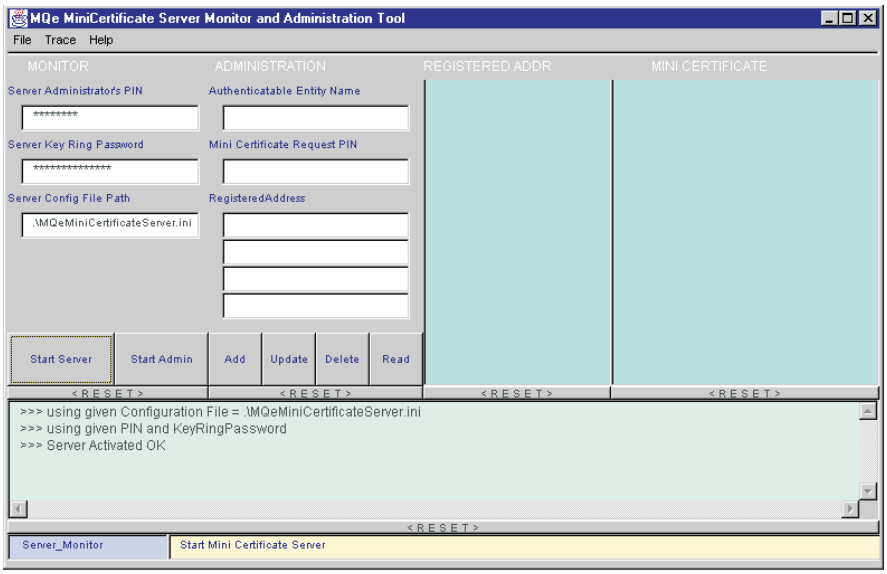


*図 31. 開始されたミニ認証サーバー*

**注:** GUI の下部の左側にある「Mode (モード)」標識は、サーバーが開始したことを 'Server_Monitor' と表示して示します。モード標識の右側の「Context (コンテキスト)」出力は、「Start Server (サーバーの開始)」ボタンのコンテキスト・ヘルプを示します。「Mode (モード)」および「Context (コンテキスト)」の上にある「Monitor (モニター)」出力は、有効なモニター出力の例です。

# 管理ツールの使用

## 管理モードの開始

管理ツールを使用するために、MQeMiniCertificateServerGUI を起動し、「Administration (管理)」モードを開始する必要があります。これは、`MQeMiniCertificateServerGUI.bat` を起動し、「Server Administrator's PIN (サーバー管理者の PIN)」、「ServerKey Ring Password (サーバー・キー・リング・パスワード)」、および「Server Config File Path (サーバー構成ファイル・パス)」入力フィールドに入力してから、「Start Admin (管理の開始)」ボタンを選択することによって行えます。このタスクからの視覚的フィードバックの例を以下に示します。

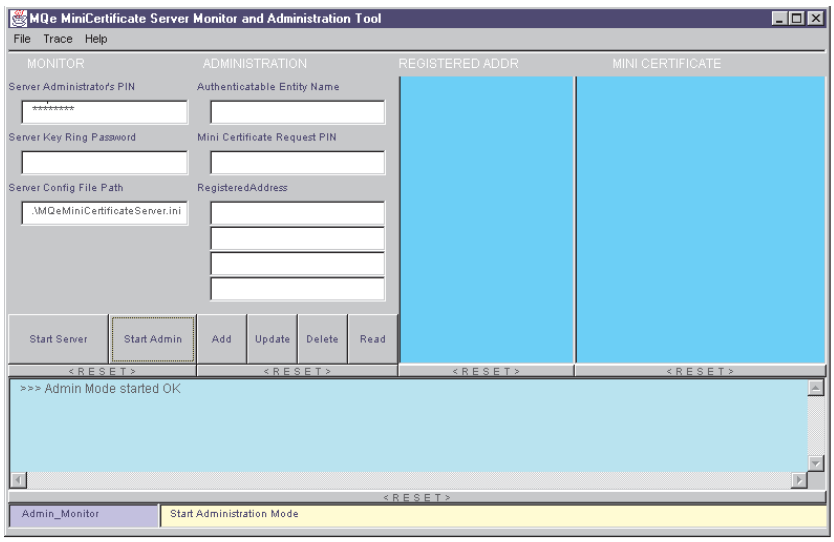


*図 32. ミニ認証サーバーの管理モード*

## 新しい認証可能なエンティティーを追加する

「Administration (管理)」モードを開始してあれば、新しい認証可能なエンティティーを追加するには、適切な入力フィールドにエンティティーの名前とアドレスを指定し、一時使用の認証要求 PIN を設定してから、「Add (追加)」ボタンをクリックします。このタスクからの視覚的フィードバックの例を以下に示します。

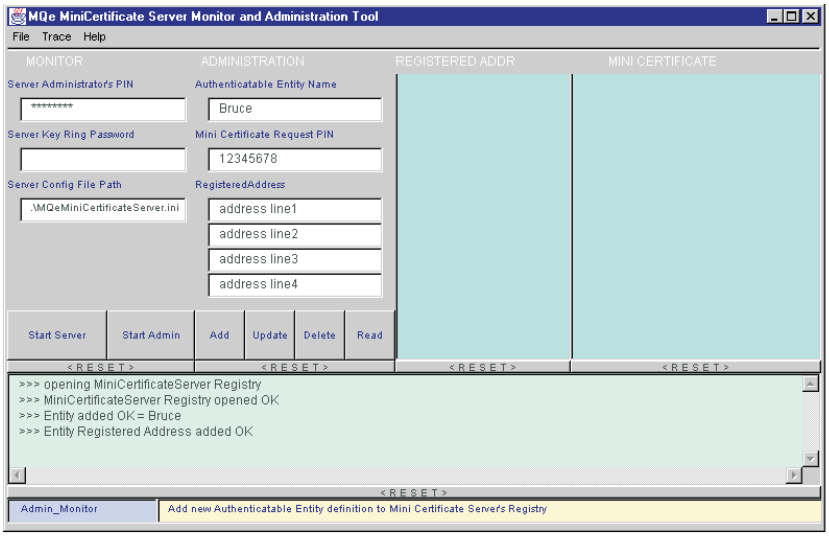


*図 33. 新しい認証可能なエンティティーを追加する*

## 認証可能なエンティティーを更新する

登録済みの認証可能なエンティティーの詳細を更新することは、エンティティーの追加と類似しています。「Administration (管理)」モードになっていれば、認証可能なエンティティーを更新した詳細が提供されます。これには、新しい認証要求 PIN

が含まれていることもあります (該当する場合)。その後、「Update (更新)」ボタンをクリックして更新します。このタスクからの視覚的フィードバックの例を以下に示します。

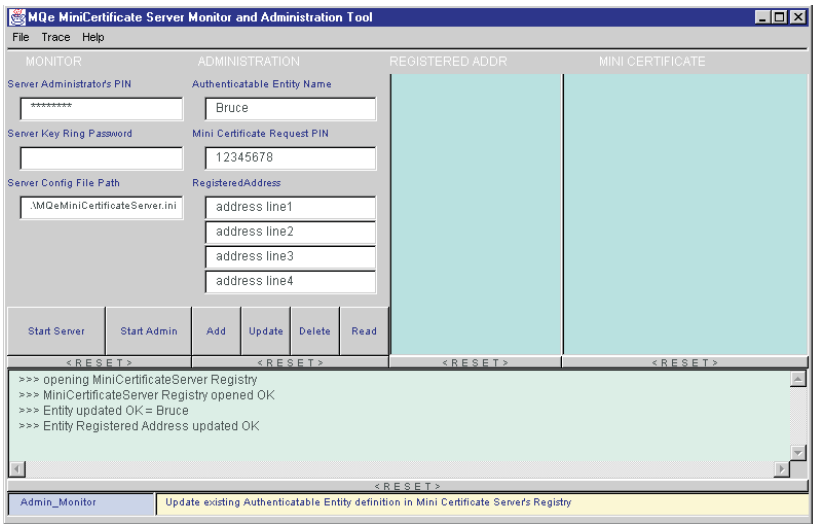


*図 34. 認証可能なエンティティーを更新する*

## 認証可能なエンティティーを削除する

登録済みの認証可能なエンティティーの詳細の削除は、認証可能なエンティティーの名前を入力フィールドに入力してから、「Delete (削除)」ボタンをクリックすることによって実行できます。

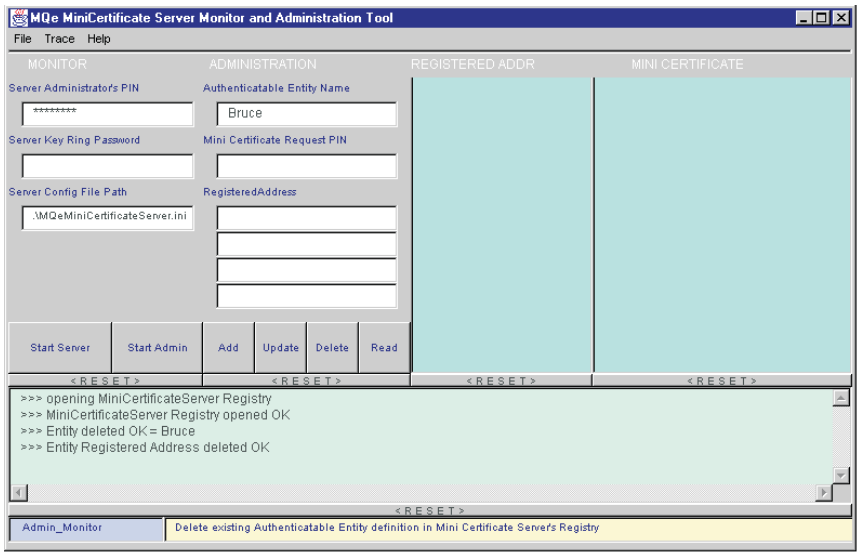


*図 35. 認証可能なエンティティーを削除する*

## 認証可能なエンティティーの詳細を読み取る

登録済みの認証可能なエンティティーの詳細を読み取るには、認証可能なエンティティーの名前を入力フィールドに入力してから、「Read (読み取り)」ボタンをクリックします。このタスクからの視覚的フィードバックの例を以下に示します。

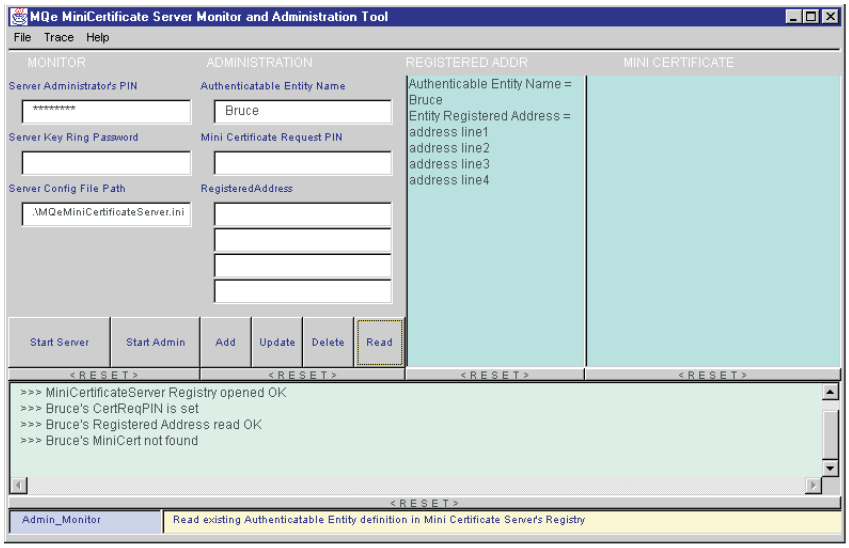


*図 36. 認証可能なエンティティーを読み取る*

これは、登録済みの認証可能なエンティティーの詳細を表示するメソッドを提供します。視覚的フィードバックは、登録済みアドレスとミニ認証、および一時使用の認証要求 PIN の状況 (使用可能な場合) を表示します。通常の使用では、認証可能なエンティティーは登録された後からミニ認証が発行される前までは、登録済みアドレスが表示され、認証要求 PIN の状況が設定されますが、ミニ認証状況は見つかりません。ミニ認証が発行された後は、登録済みアドレスと現行のミニ認証が表示され、要求 PIN 状況は設定されません。

## 「File (ファイル)」メニューの「Open (オープン)」オプションの使用

名前の入力を必要としない認証可能なエンティティーを選択するために、「Read (読み取り)」に加えて「Open (オープン)」オプションが提供されています。このオプションを使用するには、「Administration (管理)」モードで次のように実行します。

1. 「File (ファイル)」プルダウン・メニューから、「Open (オープン)」オプションを選択する。

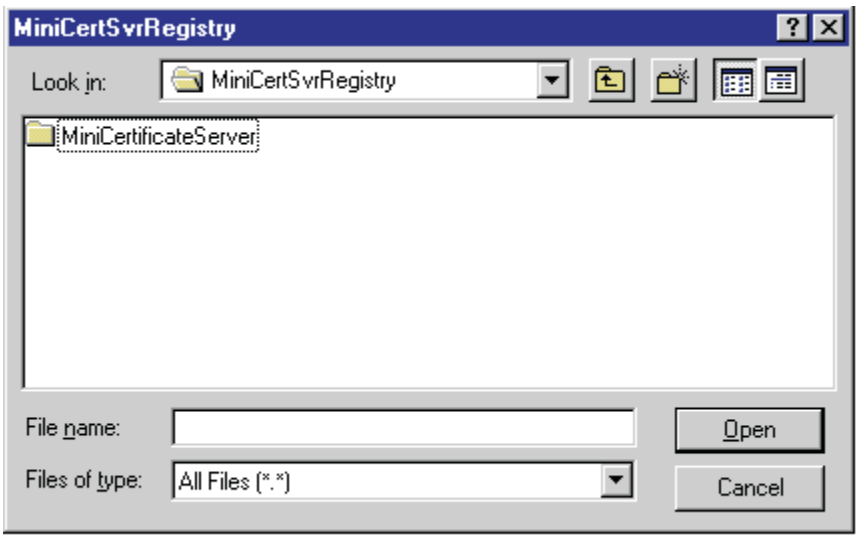


*図 37. MQSeries Everyplace の認証可能なエンティティーの詳細の表示*

2. 表示されたリストから、「EntityAddr」フォルダーを選択し、「Open (オープン)」ボタンをクリックする。

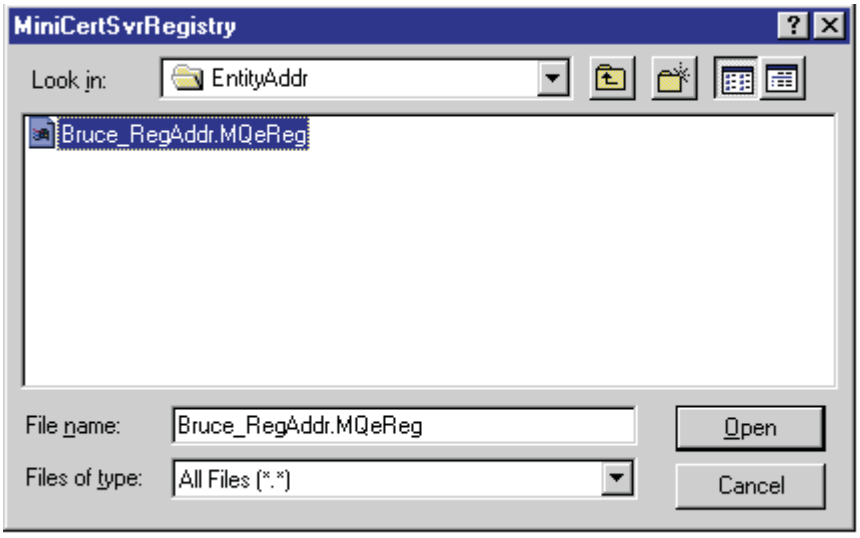


*図 38. MQSeries Everyplace の認証可能なエンティティーの詳細の表示*

3. 表示されたリストから、照会したいエンティティーの名前を選択して、「Open (オープン)」ボタンをクリックする。

エンティティーの詳細は、216ページの図39 で示されるように表示されます。

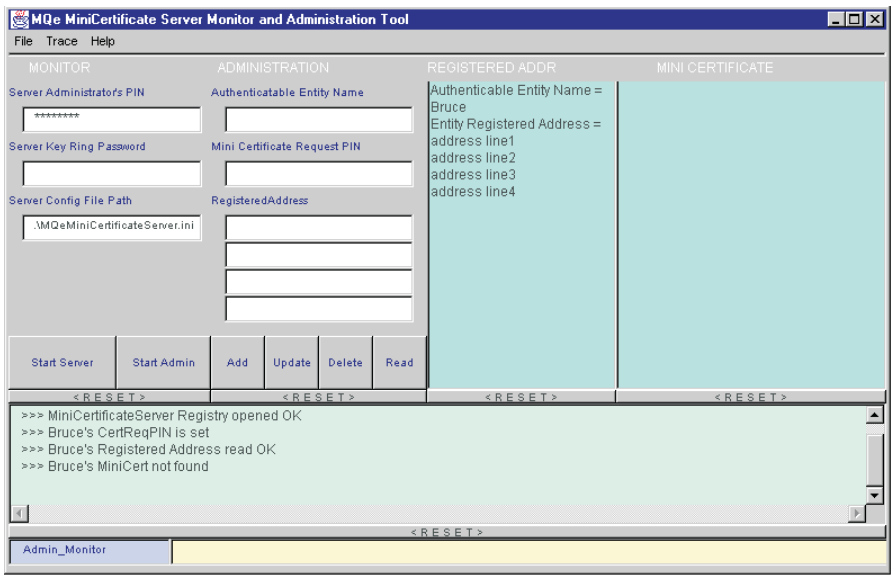


*図 39. MQSeries Everyplace の認証可能なエンティティーの詳細の表示*

# 操作

## 開始と停止

MQeMiniCertificateServerGUI のインスタンスの開始、および GUI を使用したサーバーの開始または「Administration (管理)」モードの開始については、209ページの『MQeMiniCertificateServerGUI の開始』および 211ページの『管理モードの開始』で説明されています。どちらの場合も、MQeMiniCertificateServerGUI インスタンスを終了するには、「File (ファイル)」プルダウン・メニューで「Exit (終了)」オプションを選択します。確認ダイアログで「yes (はい)」を選択し、ミニ認証サーバーのシャットダウンを完了します。

## モニターとログ記録

「Server_Monitor (サーバー・モニター)」モードまたは「Admin_Monitor (管理モニター)」モードでサーバーを実行する場合、重要なイベントがモニターされ、ビジュアル・フィードバックが「Monitor (モニター)」リスト・ボックスに、'>>> ' 接頭部付きで表示されます。

「Server_Monitor (サーバー・モニター)」モードと「Admin_Monitor (管理モニター)」モードの両方で、これらのイベントを指定されたファイルにログ記録するための追加のオプションが使用可能です。操作可能なソリューションでは、このオプションを使って監査証跡を提供することも可能です。このオプションをどちらかのモードで開始するには、「File (ファイル)」プルダウン・メニューから「Log (ログ)」オプションを選択します。このタスクの結果は、ファイル選択ダイアログ・ボックスに表示されます。

ログ・ファイル名 (後続のモニター・イベントが記録される) を選択するには、管理者は「File Name (ファイル名)」入力フィールドに表示される MQSeries Everyplace が生成したログ・ファイル名を受け入れるか (この例では '949679065895_MCSlog')、またはその名前を希望するログ・ファイル名で上書きしてから、「Save (保管)」ボタンをクリックします。

このタスクからの視覚的フィードバックの例は、図40 に示されています。

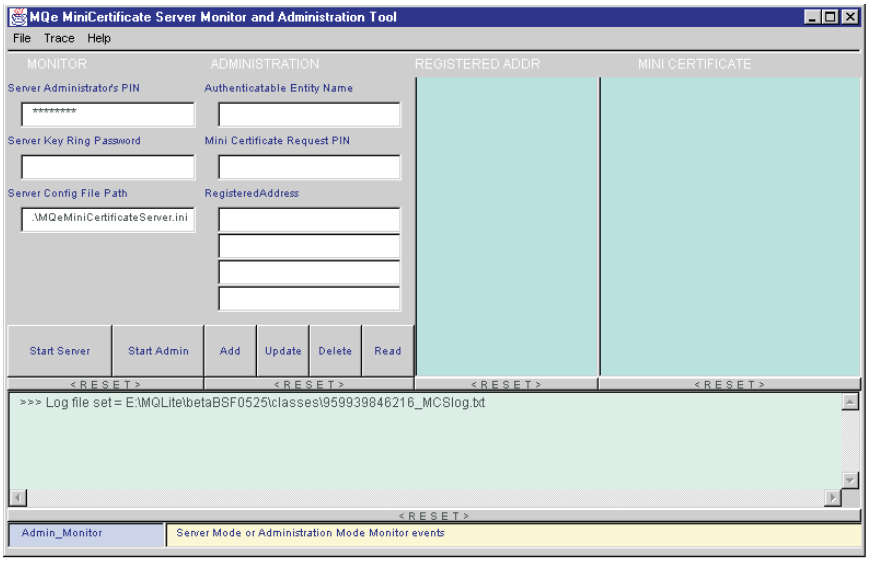


*図 40. ミニ認証サーバー・ログ・ファイル名の画面*

管理を使って「Bruce」という名前の認証可能なエンティティーを追加するために作成される、ログ・ファイルの例は以下のとおりです。

```
>>> Log file set = E:¥MQLite¥betaBSF0202¥Classes¥949682538438_MCSlog.txt
>>> Admin Mode started OK
>>> opening MiniCertificateServer Registry
>>> MiniCertificateServer Registry opened OK
>>> Entity added OK = Bruce
>>> Entity Registered Address added OK
```

# 第8章 MQSeries Everyplace でのトレース

このセクションでは、MQSeries Everyplace トレース・プログラムの使用とカスタマイズを助ける情報を提供しています。

MQSeries Everyplace は、単純で、かつ役立つトレース機能を提供します。この機能は、プログラムの実行中、または後でファイルに記録される実行の追跡を検査することによって、プログラムの実行の経路をたどるのに使用することができます。トレース・メッセージは実行中のコードから、トレース・ウィンドウに送信され、メッセージはそこに表示されます。

トレース機能はトレース専門であり、中断ポイントを設定して解放する機能など、デバッガーに備えられている一部の機能は含まれていません。

次の MQSeries Everyplace のトレース・クラスの例は、`examples.trace` サブディレクトリーにあります。

**MQeTrace**
このクラスは、トレース・メッセージを Java コンソールに表示する、単純なトレース・ハンドラーです。

**MQeTraceResource**
このクラスには、すべての MQSeries メッセージ用のテンプレートが含まれます。

**MQeTraceResourceGUI**
このクラスには、トレース・ウィンドウ制御用のすべての変換可能テキストが含まれます。

次の MQSeries Everyplace トレース・クラスは、`examples.awt` サブディレクトリーにあります。これらのクラスを使用して、トレース出力を表示するためのグラフィカル・ユーザー・インターフェースを作成することができます。

**AwtDialog**
このクラスは、単純なダイアログ・スタイルのウィンドウを作成し、処理します。

**AwtFormat**
このクラスは、ダイアログ・ウィンドウ内、またはアプリケーション・ウィンドウ・ツールキット・フレーム内で、さまざまなグラフィカル・ユーザー・インターフェース・コンポーネントを作成し、管理します。

**AwtFrame**
このクラスは、非常に単純なフレーム・スタイルのウィンドウを作成し、処理します。

これらのクラスは、実行中の MQSeries Everyplace 環境からトレースを処理し、表示するのに使用できます。トレースは通常、問題の診断を除いて、実稼働環境で使用されることはありません。どの形式であっても、トレースは MQSeries Everyplace のパフォーマンスに影響するためです。

# トレースの使用

アプリケーション・プログラムの実行をトレースするには、次の例が示すように、MQe.trace メソッドを使ってコードの適切な場所にステートメントを入れる必要があります。

```
...
/* */
trace( "We got here" );
...
```

実行時、この結果テキスト ″We got here″ が MQSeries Everyplace トレース・ウィンドウに表示されます。

## トレース・メッセージ・フォーマット

メッセージにはいくつかのタイプ (通知、警告、エラー、セキュリティー、およびデバッグ) があり、タイプは表5 で説明されているように、最初の文字で示されます。

*表5. トレース・メッセージのタイプ*

| 最初の文字 | 意味 |
|---|---|
| I または i | 通知 |
| W または w | 警告 |
| E または e | エラー |
| S または s | セキュリティー |
| D または d | デバッグ |

大文字の接頭部はアプリケーション・トレース・メッセージに使用され、小文字の接頭部はシステム・トレース・メッセージに使用されます。システム・トレース・メッセージは、通常 MQSeries Everyplace の内部からのみ生成されます。

メッセージは、メッセージのレベルを検査する MQSeries Everyplace トレース機能に送信され、必要であればそれをトレース・ウィンドウに出力します。認識可能な接頭部を持つトレース・メッセージは System.err に書き込まれ、その他のメッセージは System.out に書き込まれます。

examples.trace ディレクトリーの examples.trace.MQeTrace ファイルには、MQSeries Everyplace 内部ルーチンが発行するメッセージのさまざまなメッセージ・テンプレートが含まれます。メッセージの形式は次のとおりです。

```
/* common messages */
    {    "1",           "d;[00001]:Created"                    },
    {    "2",           "d;[00002]:Destroyed"                  },
    {    "3",           "d:[00003]:Close"                      },
    {    "4",           "w:[00004]:Warning:#0"                 },
    {    "5",           "e:[00005]:Error:#0"                   },
    {    "6",           "i:[00006]:Command:#0"                 },
    {    "7",           "i:[00007]:Waiting"                    },
    {    "8",           "i:[00008]:#0 input byte count=#1"     },
    ...,
```

ここで、先頭のストリングはメッセージ番号で、2 番目のストリングはメッセージ・テンプレートです。

examples.trace.MQeTraceResource には、メッセージ・ストリングが英語で入っています。その他の言語のバージョンも、このディレクトリーで提供されます。

テンプレートの形式は、次のとおりです。

- 220ページの表5 で説明されるメッセージ・タイプ
- 修飾子文字。この修飾子には次の意味があります。

*表 6. トレース・メッセージの修飾子*

| 修飾子 | 意味 |
|---|---|
| : | 変更は適用されない |
| ; | 作成 / 破棄オブジェクトのために予約済み |
| + | ログ・インターフェースを介してこのメッセージをログに記録する |
| ¬ | 無視 - このメッセージは表示しない |

- 形式 ’[nnnnn]:’ のメッセージ番号
- メッセージ・テキスト。これには ’#n’ という書式が挿入されています (’n’ は 0 ～ 9 の整数)。

このソース・ファイルを修正することによって、メッセージの種別を変更することができます。たとえば、警告からエラーに変更したり、修飾子文字を ’:’ から ’+’ に変更することによって、メッセージをイベント・ログにコピーしたりできます。

新しいトレース・メッセージは、addMessage または addMessageBundle 呼び出しを使って、実行時に追加できます。たとえば、1 つの新規メッセージを追加する方法は次のとおりです。

```
...
MQeTraceInterface MyTrace = MQe.GetTraceHandler();
myTrace.addMessage(" :[11111]:My Application - #0 = #1" );
...
trace( 11111, new String[] { "Magic word", "xyzzy" } );
...
```

## トレースの活動化

デフォルトでは活動状態でないトレースは、以下のコードで示すように、MQe.setTraceHandler を使って活動化できます。

```
...
  /* give the trace object to MQe */
  setTraceHandler( new myTraceHandler() );
  trace( "I:Starting..." );
  ...
```

MQSeries Everyplace ツールキットの一部として出荷されるトレース・ハンドラーのサンプルには、トレース活動化コードが含まれます。

# トレースのカスタマイズ

サンプル・ディレクトリーで提供されるトレース・クラスは、カスタム・トレース・ハンドラーとして使用できます。

## MQeTrace のサンプル

MQeTrace のサンプル・クラスは、デフォルトではトレース・メッセージを System.out および System.err、またはそのどちらかに出力する、単純なトレース機能を提供します。

トレース・ウィンドウを活動化するには、次のコードを指定します。

```
...
/* Start the example version of MQeTrace  */
new examples.trace.MQeTrace( "Trace", null ) );
...
trace( "I:Starting..." );
trace( 123456, "Insert" );
...
```

コンストラクターの 2 番目のパラメーターはトレース・メッセージに使用される言語で、ヌルが指定されると、デフォルト言語が使用されます。別の方法として、以下のように、メッセージの種別を変更する、異なるリソース・ファイルを指定することもできます。

```
...
/* Start the example version of MQeTrace  */
new examples.trace.MQeTrace( "Trace", "MyMessageResourceFile") );
...
trace( "I:Starting..." );
trace( 123456, "Insert" );
...
```

現在活動状態のトレース・ハンドラー・オブジェクトは、MQe.getTraceHandler メソッド呼び出しを発行することによって見つけることができます。この参照を使用して、トレースの動作 (書き込まれるトレースのタイプを選択または選択解除する) を変更できます。

```
...
  /* Start the example version of MQeTrace  */
  MQeTraceInterface trace = MQe.getTraceHandler( );
  if ( trace instanceof MQeTrace )
    {
    ((MQeTrace) trace).MsgInf   = true;
    ((MQeTrace) trace).MsgDebug = true;
    ((MQeTrace) trace).MsgTime  = true;
    }
  ...
  trace( "I:Starting..." );
  ...
```

変更できる MQeTrace の変数 (およびそのデフォルト) は次のとおりです。

```
public   boolean   MsgInf       = false;      /* Informaton msgs      */
public   boolean   MsgWarn      = true;       /* warning msgs         */
public   boolean   MsgErr       = true;       /* error msgs           */
public   boolean   MsgSecurity  = false;      /* Security  msgs       */
public   boolean   MsgSys       = true;       /* System modifier      */
public   boolean   MsgDebug     = false;      /* Debug modifier       */
public   boolean   MsgLog       = false;      /* Trace message to log */
public   boolean   MsgTime      = false;      /* add Time stamp       */
public   boolean   MsgPrefix    = false;      /* add object prefix    */
public   boolean   MsgThread    = false;      /* add Thread ID        */
```

詳細は、examples.trace ディレクトリーで、MQeTrace のソース・コードを調べることによって参照できます。

このトレースのサンプルを、より洗練されたトレース・プログラムの基礎として使用することもできますし、全く新しいトレースを作成することもできます。

アプリケーション・プログラムは、MQeTraceInterface をインプリメントし、MQe.setTraceHandler メソッド呼び出しを発行するだけで、通常の機能だけでなくトレース・ハンドラーにもなることができます。

## トレース用のグラフィカル・ユーザー・インターフェース

examples.trace ディレクトリーで提供される基本トレース機能は、アプリケーションと関連するコンソール・ウィンドウに、System.out および System.err に関するトレース・メッセージを表示するだけです。

examples.awt ディレクトリーには、Java AWT のサブセットを使用してグラフィカル・ユーザー・インターフェースをトレースに提供する別のトレース・ハンドラーもあります。これによってさまざまなトレース・オプションを動的に変更できます。

```
...
/* Start the example GUI version of MQeTrace */
new examples.awt.AwtMQeTrace( "My Trace title", null ) );
...
trace( "I:Starting..." );
```

このコードは、タイトル 'My Trace' でトレース・ウィンドウを開始し、通知メッセージ "I:Starting" を表示します。トレース・ウィンドウには、ユーザーがトレースのレベル、メッセージのフォーマット、および他のプロパティーを変更するためのプルダウン・メニューがあります (224ページの図41 に示されています)。トレースの実行には MQSeries Everyplace オブジェクトが必要であることに注意してください。上記の例では、コードが基本 MQSeries Everyplace クラスを拡張するクラスの一部であることが前提になっていました。それ自体は MQSeries Everyplace を拡張しないオブジェクトから、MQSeries Everyplace トレース・メッセージを出力することが可能です。この場合、MQSeries Everyplace オブジェクトを作成してから、このオブジェクトのメソッドを使って、トレースを指定することが必要です。たとえば、次のようにします。

```
...
/* create a MQe object */
MQe dbg = new MQe( );
dbg.Message( "D:We got here" );
...
```

MQSeries Everyplace トレースは、現行の Java 仮想計算機のスレッド上で実行される MQSeries Everyplace オブジェクトからのすべてのメッセージが、同じトレース機能によって処理され、同じトレース・ウィンドウに表示されるように、Java 仮想計算機全体で実行されます。これは、イベントが実際に発生する順番を示すので、大きな利点となり得ます。しかし、異なるスレッド上で発生する完全に独立したイベントを分離したい場合は、欠点になることもあります。

**注:** MQSeries Everyplace トレース・ウィンドウを終了しても、Java プログラムは終了しません。

## AWT トレース・ウィンドウのレイアウト例

トレースに関連するグラフィカル・ユーザー・インターフェース・コンポーネントで使用されるテキストを指定する、MyMessageResourceFileGUI ファイルが必要であることに注意してください。

examples.awt 中のトレース・プログラムの例は、図41 で示されるレイアウトでウィンドウを表示します。

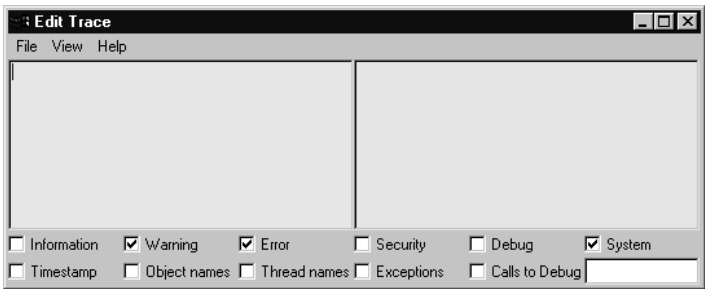


*図 41. トレース GUI ウィンドウのサンプル*

メニュー項目は次のとおりです。

- 「File (ファイル)」メニュー

  **Clear (消去)**
  トレース・ウィンドウを消去します。

  **Save As... (別名保管)**
  トレース・ウィンドウの内容をディスク・ファイルに保管します。

  **Trace to Log (ログへのトレース)**
  トレース・メッセージをイベント・ログにコピーします。

  **Trap I/O (トラップ入出力)**
  System.out および System.err への出力がウィンドウに表示されます。このオプションをチェックしないと、出力は Java コンソールに移動します。

  **Kill (強制終了)**
  トレースと所有するアプリケーションの両方を終了します。ウィンドウ・フレームの「Exit (終了)」ボタンをクリックすると、トレースだけが終了します。

- 「View (表示)」メニュー

  **View Options (表示オプション)**
  トレース・メッセージ表示オプションを表示します。

  **System.out**
  「System.out」ウィンドウを表示します。

  **System.err**
  「System.err」ウィンドウを表示します。

さまざまなトレース・メッセージの表示オプションは、*System.err.println* ウィンドウにトレース・メッセージが表示される方法、および表示されるトレース・メッセージを制御します。

**Information (通知)**
通知メッセージを表示します。

**Warning (警告)**
警告メッセージを表示します。

**Error (エラー)**
エラー・メッセージを表示します。

**Security (セキュリティー)**
セキュリティー・メッセージを表示します。

**Debug (デバッグ)**
デバッグ通知メッセージを表示します。

**System (システム)**
システム特性を持つメッセージが表示されるかどうかを制御します。これは、Information (通知)、Warning (警告)、Error (エラー)、Security (セキュリティー)、および Debug (デバッグ) スタイルのメッセージに影響を与えます。

**Timestamp (タイム・スタンプ)**
メッセージの先頭に現在のタイム・スタンプを付けます。

**Object names (オブジェクト名)**
メッセージの先頭にオブジェクト・タイプ、およびメッセージから発生するインスタンスを付けます。

**Thread names (スレッド名)**
メッセージの先頭に、現在実行中のスレッド名を付けます。

**Exceptions (例外)**
MQeException が出されるときにはスタック・トレースを表示します。

**Calls to Debug (デバッグへの呼び出し)**
アプリケーションまたは MQSeries Everyplace が MQe.Debug 呼び出しを発行する時にスタック・トレースを表示します。

'System.err.println and Trace message filter' は、出力内のものを一致させるのに使用されるストリングです。一致すると出力が表示され、一致しないと出力は表示されません。

この機能を使用して、特定のスレッドから選択的にメッセージを表示できます (「Thread name (スレッド名)」チェック・ボックスがチェックされていることが前提となっています)。

## トレース・オプションの設定

検査可能なコンポーネントのいずれかを事前検査する新しい GUI リソース・ファイルを作成することによって、AwtMQeTrace プログラムの始動時にさまざまなトレース・オプションを事前設定できます。たとえば次のとおりです。

```
public class MQeTraceResourceGUI extends java.util.ListResourceBundle
  {
  static final Object[][] contents = {
  /*    Check items can be pre-checked by replacing the blank with an "!"  */
          { "File",           "File"           },
          {   "Clear",        "Clear"          },
          {   "Save",         "Save As..."     },
          {   "Log",          " Trace to Log"  },    /* check item      */
          {   "Trap",         "!Trap I/O"      },    /* check item      */
          {   "Halt",         "Kill"           },
          { "View",           "View"           },
          {   "Options",      "!View Options"  },    /* check item      */
          {   "SystemOut",    "!System.out"    },    /* check item      */
          {   "SystemErr",    " System.err"    },    /* check item      */
          { "Help",           "Help"           },
          {   "About",        "About..."       },
  /* checkbox labels                                                     */
          { "Information",    " Information"   },    /* check item      */
          { "Warning",        "!Warning"       },    /* check item      */
          { "Error",          "!Error"         },    /* check item      */
          { "Debug",          " Debug"         },    /* check item      */
          { "Security",       " Security"      },    /* check item      */
          { "System",         "!System"        },    /* check item      */

          { "Timestamp",      " Timestamp"     },    /* check item      */
          { "Objects",        " Object names"  },    /* check item      */
          { "Threads",        " Thread names"  },    /* check item      */
          { "Exceptions",     " Exceptions"    },    /* check item      */
          { "CallStack",      " Calls to Debug" },   /* check item      */

  /* About dialog                                                        */
   { "AboutTitle",      "About MQe Trace" },
   { "AboutVersion",    "MQe version"     },
   { "AboutProduct",    "Product number 5639-I47"     },
   { "AboutCopyright",  "(C) Copyright IBM Corp. 1999 All Rights Reserved" },
   { "AboutCopyright2", "Licensed Materials - Property of IBM" },
   { "AboutTrace",      "Trace version"   },
   { "AboutComments",   " "               },
   { "OK",              "OK"              },

                                      };

  public Object[][] getContents( )

    {
    return( contents );
    }
```

**注:** トレース・オプションが MQeTrace で、次のコードで示されているように方針に基づいて変更される場合、「AwtMQeTrace」ウィンドウにある対応するコンポーネントは更新されません。

```
  ...
  /* Start the example version of MQeTrace  */
  MQeTraceInterface trace = MQe.getTraceHandler( );
  if ( trace instanceof MQeTrace )
    ((MQeTrace) trace).MsgDebug = true;
  ...
```

# 第9章 MQSeries Everyplace アダプター

このセクションでは、MQSeries Everyplace アダプターの作成について説明します。ここでは、2 つのアダプターのコーディングを説明します。 1 つは通信用のアダプター、そしてもう 1 方はメッセージ・ストア用のアダプターです。

## 簡単な通信アダプターの例

この例では、標準の Java クラスを使用して TCPIP を操作し、その先頭に独自のプロトコルを追加します。このプロトコルには、実データの後ろにあるデータ・パケットに 4 バイト長のデータから成るヘッダーがあります。このヘッダーよって、着信側には予想されるデータ量が知らされます。

この例は、MQSeries Everyplace で提供されるアダプターと置き換えることを意図したものではなく、むしろ通信アダプターの作成方法を簡単に紹介しているに過ぎません。実際のところ、エラーの処理、リカバリー、およびパラメーターの検査においては、もっと多くの注意を払う必要があります。使用する MQSeries Everyplace 構成によっては、提供されているアダプターで十分な場合もあります。

MQeAdapter の属性を継承して、新しいクラス・ファイルが構成されます。一部の変数は、このアダプターのインスタンス情報、すなわちホスト名、ポート番号、および出力ストリーム・オブジェクトを保持するように定義されています。

オブジェクトには MQeAdapter のコンストラクターが使用されるため、コンストラクターに付加的なコードを追加する必要はありません。

```
public class MyTcpipAdapter extends MQeAdapter
  {
  protected     String               host         = "";
  protected     int                  port         = 80;
  protected     Object               readLock     = new Object( );
  protected     ServerSocket         serversocket = null;
  protected     Socket               socket       = null;
  protected     BufferedInputStream  stream_in    = null;
  protected     BufferedOutputStream stream_out   = null;
  protected     Object               writeLock    = new Object( );
```

次に、activate メソッドがコード化されます。これは、ファイル記述子から、ターゲット・ネットワーク・アドレスの名前 (コネクターの場合) や聴取ポート (リスナーの場合) を取り出すメソッドです。 fileDesc パラメーターには、アダプターのクラス名または別名、およびそのアダプターに関するすべてのネットワーク・アドレス・データ (例、MyTcpipAdapter:127.0.0.1:80) が含まれます。 thisParam パラメーターには、管理によって接続が定義される際に設定されたすべてのデータが含まれます。なお、このパラメーターの値は、通常 ″?Channel″ のようになります。 thisOpt パラメーターには、管理によって設定されたアダプターのセットアップ・オプションが含まれます。たとえば、このアダプターが着信の接続に対して聴取を行うものである場合は、MQe_Adapter_LISTEN が含まれます。

```
  public void   activate( String        fileDesc,
                          Object        thisParam,
                          Object        thisOpt,
```

```
                      int           thisValue1,
                      int           thisValue2 ) throws Exception
  {
  super.activate( fileDesc,
                  thisParam,
                  thisOpt,
                  thisValue1,
                  thisValue2 );
  /* isolate the TCP/IP address - "MyTcpipAdapter:127.0.0.1:80"     */
  host = fileId.substring( fileId.indexOf( ':' ) + 1 );
  i    = host.indexOf( ':' );                /* find delimiter     */
  if ( i > -1 )                              /* find it ?          */
    {
    port = (new Integer( host.substring( i + 1 ) )).intValue( );
    host = host.substring( 0, i );
    }
  }
```

出力ストリームをクローズし、すべての残りのデータをストリーム・バッファーからフラッシュするためには、close メソッドを定義する必要があります。 close は、クライアント / サーバー間のセッションで何回も呼び出されますが、チャネルが完全にアダプターの使用を終えると、今度は close がオプション MQe_Adapter_FINAL を指定して MQSeries Everyplace を呼び出します。チャネルの存続期間ごとに 1 つのソケット接続を持つアダプターでは、オプション MQe_Adapter_FINAL を指定して呼び出しを設定し、ソケットを実際にクローズするために使用されるアダプターでは、バッファーをフラッシュするだけの別の呼び出しを行います。ただし、各要求ごとに新しいソケットが使用される場合は、MQSeries Everyplace への各呼び出しごとにソケットがクローズされ、次の open 呼び出しでまた新しいソケットが割り振られます。

```
public void    close(  Object  opt ) throws Exception
  {
  if ( stream_out   != null )                /* output stream ?    */
    {
    stream_out.flush();                      /* empty the  buffers */
    stream_out.close();                      /* close it           */
    stream_out = null;                       /* clear              */
    }
  if ( stream_in    != null )                /* input stream ?     */
    {
    stream_in.close();                       /* close it           */
    stream_in = null;                        /* clear              */
    }
  if ( socket       != null )                /* socket ?           */
    {
    socket.close();                          /* close it           */
    socket = null;                           /* clear              */
    }
  if ( serversocket != null )                /* serversocket ?     */
    {
    serversocket.close();                    /* close it           */
    serversocket = null;                     /* clear              */
    }
  host = "";
  port = 80;
  }
```

MQe_Adapter_ACCEPT 要求を処理して接続要求の着信を受け入れるためには、control メソッドをコード化する必要があります。これは、ソケットがリスナー (サーバー・ソケット) である場合にのみ可能です。聴取ソケットに指定されたすべてのオプション (MQe_Adapter_LISTEN を除く) は、受け入れの結果として作成され

たソケットにコピーされます。これを行うためには、MQe_Adapter_SETSOCKET という別の制御オプションを使用します。このオプションを使用して、インスタンス化されたばかりのアダプターにソケット・オブジェクトを渡すことができます。

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LISTEN      ) &&
         checkOption( opt, MQe.MQe_Adapter_ACCEPT ) )
      {
      /* CtrlObj - is a string representing the file descriptor of the */
      /*           MQeAdapter object to be returned e.g. "MyTcpip:"    */
      Socket  ClientSocket = serversocket.accept();   /* wait connect  */
      String  Destination  = (String) ctrlObj;        /* re-type object*/
      int i  = Destination.indexOf( ':' );
      if ( i < 0 )
        throw new MQeException( MQe.Except_Syntax,
                                "Syntax:" + Destination );
      /* remove the Listen option                                      */
      String NewOpt = (String) options;         /* re-type to string   */
      int j  = NewOpt.indexOf( MQe.MQe_Adapter_LISTEN );
      NewOpt = NewOpt.substring( 0, j ) +
               NewOpt.substring( j + MQe.MQe_Adapter_LISTEN.length( ) );
      MQeAdapter Adapter = MQe.newAdapter( Destination.substring( 0,i+1 ),
                                           parameter,
                                           NewOpt + MQe_Adapter_ACCEPT,
                                           -1,
                                           -1 );
      /* assign the new socket to this new adapater                    */
      Adapter.control( MQe.MQe_Adapter_SETSOCKET, ClientSocket );
      return( Adapter );
      }
    else
    if ( checkOption( opt, MQe.MQe_Adapter_SETSOCKET ) )
      {
      if ( stream_out != null )  stream_out.close();
      if ( stream_in  != null )  stream_in .close();
      if ( ctrlObj    != null )                 /* socket supplied ?   */
        {
        socket     = (Socket) ctrlObj;          /* save the socket     */
        stream_in  = new BufferedInputStream ( socket.getInputStream () );
        stream_out = new BufferedOutputStream( socket.getOutputStream() );
        }
    else
      return( super.control( opt, ctrlObj ) );
  }
```

open メソッドでは、聴取のソケットやコネクターのソケットを検査し、適当なソケット・オブジェクトを作成する必要があります。入力および出力ストリームの再初期設定を行うためには、control メソッドを使用し、これに新しいソケット・オブジェクトを渡します。 opt パラメーターは、MQe_Adapter_RESET に設定することができます。これは、直前の操作がすべて完了し、新しい任意の読み取りまたは書き込みによって新しい要求を構成することを意味します。

```
  public void open( Object opt ) throws Exception
    {
    if ( checkOption( MQe.MQe_Adapter_LISTEN ) )
      serversocket = new ServerSocket( port, 32 );
    else
      control( MQe.MQe_Adapter_SETSOCKET, new Socket( host, port ) );
    }
```

read メソッドでは、読み取り可能な最大レコード・サイズを指定するパラメーターを使用することができます。

この例では、内部ルーチンを呼び出してデータ・バイトの読み取りとエラー・リカバリー (それが適当な場合) を行い、次いで、読み取られる数バイトのデータについて正確な長さのバイト配列を返します。このソケットで 1 度に複数の読み取りが行われることのないよう、注意を払う必要があります。 opt パラメーターには、次の値を設定することができます。

**MQe_Adapter_CONTENT**
すべてのメッセージの内容を読み取る

**MQe_Adapter_HEADER**
すべてのヘッダー情報を読み取る

```
{ public byte[] read( Object opt, int recordSize ) throws Exception

    int Count = 0;                              /* number bytes read   */
    synchronized ( readLock )                   /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) )
        {
        byte lreclBytes[] = new byte[4];        /* for the data length */
        readBytes( lreclBytes, 0, 4 );          /* read the length     */
        int  recordSize = byteToInt( lreclBytes, 0, 4 );
        }
      if ( checkOption( opt, MQe.MQe_Adapter_CONTENT   ) )
        {
        byte Temp[] = new byte[recordSize];     /* allocate work array */
        Count = readBytes( Temp, 0, recordSize);/* read data           */
        }
      }
    if ( Count < Temp.length )                  /* read all length ?   */
      Temp    = MQe.sliceByteArray( Temp, 0, Count );
    return ( Temp );                            /* Return the data     */
    }
```

readByte メソッドは、ソケットから単一バイトのデータを読み取り、エラーがあった場合は指定された回数の再試行を試みたり、あるいは、読み取れる以上のデータが存在する場合はファイルの終わり例外を出すよう設計された内部ルーチンです。

```
  protected int readByte( ) throws Exception
    {
    int intChar    = -1;                        /* input characater    */
    int RetryValue =  3;                        /* error retry count   */
    int Retry = RetryValue + 1;                 /* reset retry count   */
    do{                                         /* possible retry      */
      try                                       /* catch io errors     */
        {
        intChar = stream_in.read();             /* read a character    */
        Retry   = 0;                            /* dont retry          */
        }
      catch ( IOException e )                   /* IO error occured    */
        {
        Retry   = Retry - 1;                    /* decrement           */
        if ( Retry == 0 )  throw e;             /* more attempts ?     */
        }
      } while ( Retry != 0 );                   /* more attempts ?     */
    if ( intChar == -1 )                        /* end of file ?       */
      throw new EOFException();                 /* ... yes, EOF        */
    return( intChar );                          /* return the byte     */
    }
```

readBytes メソッドは、ソケットから数バイトのデータを読み取り、エラーがあった場合は指定された回数の再試行を試みたり、あるいは、読み取れる以上のデータが存在する場合はファイルの終わり例外を出すよう設計された内部ルーチンです。

```
protected int readBytes( byte buffer[], int offset, int recordSize )
  throws Exception
  {
  int RetryValue = 3;
  int i = 0;                                    /* start index         */
  while ( i < recordSize )                      /* got it all in yet ? */
    {                                           /* ... no              */
    int NumBytes = 0;                           /* read count          */
    /* retry any errors based on the QoS Retry value                   */
    int Retry = RetryValue + 1;                 /* error retry count   */
    do{                                         /* possible retry      */
      try                                       /* catch io errors     */
        {
        NumBytes = stream_in.read( buffer, offset + i, recordSize - i );
        Retry    = 0;                           /* no retry            */
        }
      catch ( IOException e )                   /* IO error occured    */
        {
        Retry    = Retry - 1;                   /* decrement           */
        if ( Retry == 0 )  throw e;             /* more attempts ?     */
        }
      } while ( Retry != 0 );                   /* more attempts ?     */
    /* check for possible end of file                                  */
    if ( NumBytes < 0 )                         /* errors ?            */
      throw new EOFException( );                /* ... yes             */
    i = i + NumBytes;                           /* accumulate          */
    }    return ( i );                          /* Return the count    */
  }
```

readln メソッドは、0x0A の文字で終わるバイト・ストリングを読み取ります。このメソッドでは 0x0D 文字は無視されます。

```
  {
  synchronized ( readLock )                     /* only one at a time  */
    {
    /* ignore the 4 byte length                                        */
    byte lreclBytes[] = new byte[4];            /* for the data length */
    readBytes( lreclBytes, 0, 4 );              /* read the length     */

    int intChar    = -1;                        /* input characater    */
    StringBuffer Result = new StringBuffer( 256 );
    /* read Header from input stream                                   */
    while ( true )                              /* until "newline"     */
      {
      intChar = readByte( );                    /* read a single byte  */
      switch ( intChar )                        /* what character      */
        {                                       /*                     */
        case -1:                                /* ... no character    */
          throw new EOFException();             /* ... yes, EOF        */
        case 10:                                /* eod of line         */
          return( Result.toString() );          /* all done            */
        case 13:                                /* ignore              */
          break;
        default:                                /* real data           */
          Result.append( (char) intChar );      /* append to string    */
        }                                       /* end of line ?       */
      }
    }
  }
```

status メソッドは、アダプターに関する状況情報を返します。この例では、オプション MQe_Adapter_NETWORK に対してネットワークのタイプ (TCPIP) が返され、オプション MQe_Adapter_LOCALHOST に対して tcpip のローカル・ホスト・アドレスが返されます。

```
public String status( Object  opt  ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_NETWORK ) )
      return( "TCPIP" );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_LOCALHOST ) )
      return( InetAddress.getLocalHost( ).toString() );
    else
    return( super.status( opt ) );
    }
```

write メソッドは、データのブロックをソケットに書き込みます。ソケットに対する書き込みは、1 度に 1 つしか実行できないことを確認してください。この例で、このメソッドは内部ルーチン writeBytes を呼び出して実データを書き込み、任意の適当なエラー・リカバリーを実行します。

opt パラメーターには、次の値を設定することができます。

**MQe_Adapter_FLUSH**
バッファー内のすべてのデータをフラッシュする

**MQe_Adapter_HEADER**
すべてのヘッダー・レコードを書き込む

**MQe_Adapter_HEADERRSP**
すべてのヘッダー応答レコードを書き込む

```
  public void write( Object opt, int recordSize, byte data[] )
    throws Exception
    {
    synchronized ( writeLock )                  /* only one at a time  */
      {
      if ( checkOption( opt, MQe.MQe_Adapter_HEADER    ) ||
           checkOption( opt, MQe.MQe_Adapter_HEADERRSP )  )
        writeBytes( intToByte( recordSize ), 0, 4 );    /* write length*/
      writeBytes( data, 0, recordSize );        /* write the data      */
      if ( checkOption( opt, MQe.MQe_Adapter_FLUSH  ) )
        stream_out.flush( );                    /* make sure it is sent */
      }                                         /*                     */
    }
```

writeBytes は内部メソッドで、ソケットに対してバイト配列 (または部分的な配列) の書き込みを行い、エラーが発生した場合には簡単なエラー・リカバリーを試行します。

```
protected void writeBytes( byte buffer[], int offset, int recordSize )
    throws Exception
    {
    if ( buffer != null )                       /* any data ?          */
      {
      /* break the data up into manageable chuncks                     */
      int i = 0;                                /* Data index          */
      int j = recordSize;                       /* Data length         */
      int MaxSize    = 4096;                    /* small buffer        */
      int RetryValue = 3;                       /* error retry count   */
      do{                                       /* as long as data     */
        if ( j < MaxSize )                      /* smallbuffer ?       */
          MaxSize = j;
        int Retry = RetryValue + 1;             /* error retry count   */
        do{                                     /* possible retry      */
           try                                  /* catch io errors     */
            {
            stream_out.write( buffer, offset + i, MaxSize );
            Retry = 0;                          /* don't retry         */
```

```
          }
        catch ( IOException e )              /* IO error occured    */
          {
          Retry = Retry - 1;                 /* decrement           */
          if ( Retry == 0 )  throw e;        /* more attempts ?     */
          }
        } while ( Retry != 0 );              /* more attempts ?     */

      i = i + MaxSize;                       /* update index        */
      j = j - MaxSize;                       /* data left           */
      } while ( j > 0 );                     /* till all data sent  */
    }
  }
```

writeLn メソッドは、0x0A 文字や 0x0D 文字で終わる文字ストリングをソケットに書き込みます。

opt パラメーターには、次の値を設定することができます。

**MQe_Adapter_FLUSH**
バッファー内のすべてのデータをフラッシュする

**MQe_Adapter_HEADER**
すべてのヘッダー・レコードを書き込む

**MQe_Adapter_HEADERRSP**
すべてのヘッダー応答レコードを書き込む

```
public void writeln( Object opt, String data ) throws Exception
  {
  if ( data == null )                        /* any data ?          */
    data = "";
  write( opt, -1, MQe.asciiToByte( data + "¥r¥n" ) ); /* write data  */
  }
```

これで、(非常に簡単な例ではあるものの、) リスナーやコネクターとして開始された自身の他のコピーと通信を行う tcpip アダプターが完成しました。

## 簡単なメッセージ・ストア・アダプターの例

この例では、メッセージ・ストア用のインターフェースとして使用されるアダプターを作成します。このアダプターでは、標準の Java 入出力クラスを使用してストア内のファイルを操作します。

この例は、MQSeries Everyplace で提供されるアダプターと置き換えることを意図したものではなく、むしろメッセージ・ストア・アダプターの作成方法を簡単に紹介しているに過ぎません。

MQeAdapter の属性を継承して、新しいクラス・ファイルが構成されます。一部の変数は、ファイル / メッセージの名前やメッセージ・ストアの位置といった、このアダプターのインスタンス情報を保持するように定義されています。

オブジェクトには MQeAdapter のコンストラクターが使用されるため、コンストラクターに付加的なコードを追加する必要はありません。

```
public class MyMsgStoreAdapter extends    MQeAdapter
                               implements FilenameFilter
  {
  protected String  filter   = "";           /* file type filter    */
  protected String  fileName = "";           /* disk file name      */
```

```
  protected String  filePath = "";              /* drive and directory */
  protected boolean reading  = false;           /* open'd for reading  */
  protected boolean writing  = false;
```

このアダプターでは FilenameFilter を実装するため、次のメソッドをコード化する必要があります。これは、メッセージ・ストア内で特定のタイプのファイルを選択するために使用される、フィルター操作のメカニズムです。

```
  public boolean accept( File dir, String name )
    {
    return( name.endsWith( filter ) );
    }
```

次に、activate メソッドがコード化されます。これは、ファイル記述子から、すべてのメッセージの保持に使用するディレクトリーの名前を取り出すメソッドです。

このメソッド呼び出しの Object パラメーターには、属性オブジェクトを使用することができます。そのようにすると、これはメッセージ・ストア内のメッセージをエンコード / デコードするために使用される属性となります。

このアダプターでは、次のような Object オプションを使用できます。

- MQe_Adapter_READ
- MQe_Adapter_WRITE
- MQe_Adapter_UPDATE

他のすべてのオプションは無視されます。

```
public void   activate( String  fileDesc,
                        Object  param,
                        Object  options,
                        int     value1,
                        int     value2 ) throws Exception
    {
    super.activate( fileDesc, param, options, lrecl, noRec );
    filePath    = fileId.substring( fileId.indexOf( ':' ) + 1 );
    String Temp = filePath;                     /* copy the path data  */
    if ( filePath.endsWith( File.separator ) )  /* ending separator ?  */
      Temp      = Temp.substring( 0, Temp.length( ) -
                                  File.separator.length( ) );
    else
      filePath  = filePath + File.separator;    /* add separator       */
    File diskFile = new File( Temp );
    if ( ! diskFile.isDirectory( ) )            /* directory ?         */
      if ( ! diskFile.mkdirs( ) )               /* does mkDirs work ?  */
        throw new MQeException( MQe.Except_NotAllowed,
                                "mkdirs '" + filePath + "' failed" );
    filePath    = diskFile.getAbsolutePath( ) + File.separator;
    this.open( null );
    }
```

close メソッドでは、読み取りや書き込みが許されていません。

```
public void close( Object opt ) throws Exception
    {
    reading  = false;                           /* not open for reading*/
    writing  = false;                           /* not open for writing*/
    }
```

MQe_Adapter_LIST、つまり、ディレクトリー内のフィルターを満たすファイルをリストする要求を処理するためには、control メソッドをコード化する必要がありま

す。また、MQe_Adapter_FILTER、つまりフィルターを設定してファイルのリスト方法を制御させる要求を処理する場合にも、このメソッドのコード化が必要です。

```
public Object control( Object opt, Object ctrlObj ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_LIST   ) )
      return( new File( filePath ).list( this ) );
    else
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER ) )
      {
      filter = (String) ctrlObj;                /* set the filter      */
      return( null );                           /* nothing to return   */
      }
    else
    return( super.control( opt, ctrlObj ) );    /* try ancestor        */
    }
```

erase メソッドは、メッセージ・ストアからメッセージを除去する際に使用されます。

```
  public void erase( Object opt ) throws Exception
    {
    if ( opt instanceof String )                /* select file ?       */
      {
      String FN = (String) opt;                 /* re-type the option  */
      if ( FN.indexOf( File.separator ) > -1 )  /* directory ?         */
        throw new MQeException( MQe.Except_Syntax, "Not allowed" );
      if ( ! new File( filePath + FN ).delete( ) )
        throw new MQeException( MQe.Except_NotAllowed, "Erase failed" );
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

open メソッドは、メッセージの読み取りとメッセージの書き込みの両方を許可するブール値を設定します。

```
public void open( Object opt ) throws Exception
    {
    this.close( null );                         /* close any open file */
    fileName = null;                            /* clear the filename  */
    if ( opt instanceof String )                /* select new file ?   */
      fileName = (String) opt;                  /* retype the name     */
    reading  = checkOption( opt, MQe.MQe_Adapter_READ   ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    writing  = checkOption( opt, MQe.MQe_Adapter_WRITE  ) ||
               checkOption( opt, MQe.MQe_Adapter_UPDATE );
    }
```

readObject メソッドは、メッセージ・ストアからメッセージを読み取り、正しいタイプのオブジェクトを再作成します。また、activate 呼び出しに属性が指定されている場合は、データの復号と解凍も行います。これは、特別な機能であり、この機能によって要求は読み取りのパラメーターで指定された突き合わせの基準を満たすファイルを読み取り、最初に出現する突き合わせの基準を満たすメッセージを返します。

```
public Object readObject( Object opt ) throws Exception
    {
    if ( reading )
      {
      if ( opt instanceof MQeFields )
        {
        /* 1. list all files in the directory                          */
        /* 2. read each file in turn and restore as a Fields object    */
```

```
        /* 3. try an equality check - if equal then return that object */
        String List[] = new File( filePath ).list( this );
        MQeFields Fields = null;
        for ( int i = 0; i < List.length; i = i + 1 )
          try
            {
            fileName = List[i];                 /* remember the name   */
            open( fileName );                   /* try this file       */
            Fields = (MQeFields) readObject( null );
            if ( Fields.equals( (MQeFields) opt ) )     /* match ?     */
              return( Fields );
            }
          catch ( Exception e )                 /* error occured       */
            {
            }                                   /* ignore error        */
        throw new MQeException( Except_NotFound, "No match" );
        }
      /* read the bytes from disk                                       */
      File diskFile = new File( filePath + fileName );
      byte data[]   = new byte[(int) diskFile.length()];
      FileInputStream InputFile = new FileInputStream( diskFile );
      InputFile.read( data );                   /* read the file data  */
      InputFile.close( );                       /* finish with file    */
      /* possible Attribute decode of the data                          */
      if ( parameter instanceof MQeAttribute )  /* Attribute encoding ?*/
        data = ((MQeAttribute) parameter).decodeData( null,
                                                      data,
                                                      0,
                                                      data.length );
      MQeFields FieldsObject = MQeFields.reMake( data, null );
      return( FieldsObject );
      }
    else
      throw new MQeException( MQe.Except_NotSupported, "Not supported" );
    }
```

status メソッドは、アダプターに関する状況情報を返します。この例では、フィルターのタイプやファイル名が返されます。

```
public String status( Object  opt  ) throws Exception
    {
    if ( checkOption( opt, MQe.MQe_Adapter_FILTER   ) )
      return( filter   );
    if ( checkOption( opt, MQe.MQe_Adapter_FILENAME ) )
      return( fileName );
    return( super.status( opt ) );
    }
```

writeObject メソッドは、メッセージ・ストアにメッセージを書き込みます。また、activate 呼び出し員属性が指定されている場合は、メッセージ・オブジェクトの圧縮と暗号化も行います。

```
public void writeObject( Object  opt,
                         Object  data ) throws Exception
    {
    if ( writing && (data instanceof MQeFields) )
      {
      byte dump[] = ((MQeFields) data).dump( );         /* dump object */
      /* possible Attribute encode of the data                        */
      if ( parameter instanceof MQeAttribute )
        dump = ((MQeAttribute) parameter).encodeData( null,
                                                      dump,
                                                      0,
                                                      dump.length );
      /* write out the object bytes                                   */
      File diskFile = new File( filePath + fileName );
```

```
    FileOutputStream OutputFile = new FileOutputStream( diskFile );
    OutputFile.write( dump );                 /* write the data      */
    OutputFile.getFD().sync( );               /* synchronize disk    */
    OutputFile.close();                       /* finish with file    */
    }
  else
    throw new MQeException( MQe.Except_NotSupported, "Not supported" );
  }
```

これで、(非常に簡単な例ではありますが、) メッセージ・ストアに対してメッセージ・オブジェクトの読み取りおよび書き込みを行うメッセージ・ストア・アダプターが完成しました。

このアダプターは、たとえばデータベースや不揮発性メモリーにメッセージを保管させるなど、多種多様なコード化が可能です。

# 付録A. MQSeries Everyplace への保守の適用

保守の更新を適用する場合には、更新に関して出されている指示に従ってください。

保守の更新とその可用性についてのより詳細な一般情報は、http://www.software.ibm.com/ts/mqseries/ にある、MQSeries ファミリーの Web ページを参照してください。

# 付録B. 特記事項

本書において、日本では発表されていない IBM 製品 (機械およびプログラム)、プログラミングまたはサービスについて言及または説明する場合があります。しかし、このことは、弊社がこのような IBM 製品、プログラミングまたはサービスを、日本で発表する意図があることを必ずしも示すものではありません。本書で IBM ライセンス・プログラムまたは他の IBM 製品に言及している部分があっても、このことは当該プログラムまたは製品のみが使用可能であることを意味するものではありません。 IBM 製品、プログラム、またはサービスに代えて、 IBM の有効な知的所有権またはその他の法的に保護された権利を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM によって明示的に指定されたものを除き、他社の製品と組み合わせた場合の操作の評価と検証はお客様の責任で行っていただきます。

IBM は、本書で解説されている主題について特許権 (特許出願を含む)、商標権、または著作権を所有している場合があります。本書の提供は、これらの特許権、商標権、および著作権について、本書で明示されている場合を除き、実施権、使用権等を許諾することを意味するものではありません。実施権、使用権等の許諾については、下記の宛先に、書面にてご照会ください。

〒106-0032 東京都港区六本木 3 丁目 2-31
AP 事業所
IBM World Trade Asia Corporation
Intellectual Property Law & Licensing

**以下の保証は、国または地域の法律に沿わない場合は、適用されません。** IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

本書に対して、周期的に変更が行われ、これらの変更は、文書の次版に組み込まれます。 IBM は、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、 (i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire
England
SO21 2JN

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。これらの例は、すべての場合について完全にテストされたものではありません。 IBM はこれらのプログラムの信頼性、可用性、および機能について法律上の瑕疵担保責任を含むいかなる明示または暗示の保証責任も負いません。

## 商標

次のものは、IBM Corporation の米国およびその他の国における商標です。

IBM
MQSeries

Java およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

UNIX は、The Open Group がライセンスしている米国およびその他の国における登録商標です。

Windows、および Windows NT は Microsoft Corporation の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標または登録商標です。

# 用語集

この用語集は、本書で使用されている用語、および日常用いられる場合とは意味合いで使用されている語について説明します。場合によっては、1 つの用語にあてはまる定義が 1 つだけではないこともありますが、その定義は本書でその語が使用されるときの意味を表します。

探している用語が見つからない場合は、索引または *IBM Dictionary of Computing* (New York:. McGraw-Hill, 1994) を参照してください。

**アプリケーション・プログラミング・インターフェース (Application Programming Interface (API)).** アプリケーション・プログラミング・インターフェースは、作成するアプリケーションでプログラマーが使用することのできる関数と変数から成り立っている。

**非同期メッセージング (asynchronous messaging).** プログラム間で通信するときのメソッドの一つで、このメソッドを使用してプログラムはメッセージをメッセージ・キューに入ることができる。非同期メッセージングを使用すると、送信側のプログラムは送ったメッセージへの応答を待たずに自分の処理を継続する。*同期メッセージング (synchronous messaging)* と対比。

**認証プログラム (authenticator).** メッセージの送信側と受信側を検査するプログラム。

**ブリッジ (bridge).** MQSeries Everyplace と他のメッセージング・システム (MQSeries を含む) との間でメッセージを流せるようにする MQSeries Everyplace オブジェクト。

**チャネル (channel).** *ダイナミック・チャネル (dynamic channel)* および *MQI チャネル (MQI channel)* を参照。

**チャネル・マネージャー (channel manager).** エンドポイント間の論理多重並行通信パイプをサポートする MQSeries Everyplace オブジェクト。

**クラス (class).** クラスとは、データおよびそのデータを操作するメソッドのカプセル化コレクションである。クラスのインスタンスであるオブジェクトを作成するために、クラスをインスタンス化することもできる。

**クライアント (client).** MQSeries において、クライアントはローカル・ユーザー・アプリケーションにサーバー上でキューに入っているサービスへのアクセスを提供する実行時コンポーネントである。

**圧縮機能 (compressor).** 転送するデータのサイズが小さくなるよう、メッセージを圧縮するプログラム。

**暗号機能 (cryptor).** 転送中のセキュリティーを提供するため、メッセージを暗号化するプログラム。

**ダイナミック・チャネル (dynamic channel).** ダイナミック・チャネルは、MQSeries Everyplace デバイスを接続し、同期および非同期メッセージを転送し、および両方向で応答する。

**カプセル化 (encapsulation).** カプセル化は、オブジェクトのデータをプライベートまたは保護にしたり、プログラマーがメソッド呼び出しを通してのみオブジェクトのデータにアクセスおよび操作できるようにしたりするオブジェクト指向プログラミングの技法である。

**ゲートウェイ (gateway).** MQSeries Everyplace ゲートウェイ (またはサーバー) は、チャネル・マネージャーを含む MQSeries Everyplace コードを実行するコンピューターである。

**ハイパーテキスト・マークアップ言語 (Hypertext Markup Language (HTML)).** ワールド・ワイド・ウェブ (WWW) に表示する情報を定義するために用いられる言語。

**インスタンス (instance).** インスタンスとはオブジェクトである。クラスがインスタンス化されてオブジェクトが作成されたとき、そのオブジェクトはクラスのインスタンスであるという。

**インターフェース (interface).** インターフェースは、抽象メソッドだけを含みインスタンス変数を含まないクラスである。インターフェースは、メソッドの共通セットを提供するが、そのメソッドは多数の異なるクラスのサブクラスによってインプリメントすることができる。

**インターネット (Internet).** インターネットは、情報を共有する連携公衆ネットワークである。物理的には、インターネットは現在存在する公衆通信ネットワークすべての資源全体のサブセットを使用する。技術的に、インターネットは TCP/IP (転送制御プロトコル / インターネット・プロトコル) というプロトコル・セットを使用していることにより、連携公衆ネットワークとして識別される。

**Java 開発者キット (Java Developers Kit (JDK)).** Sun Microsystems 社によって Java 開発者向けに配布されているソフトウェア・パッケージ。その中には、Java インタープリター、Java クラス、および Java 開発ツールが含まれている。 Java 開発ツールにはコンパイラー、デバッガー、逆アセンブラー、アプレット・ビューアー、スタブ・ファイル生成プログラム、および文書生成プログラムがある。

**Java ネーミングおよびディレクトリー・サービス (Java Naming and Directory Service (JNDI)).** Java プログラム言語で指定される API。この API は Java プログラム言語で作成されたアプリケーションにネーミングおよびディレクトリー機能を提供する。

**Lightweight Directory Access Protocol (LDAP).** LDAP はディレクトリー・サービスへのアクセスに用いられるクライアント・サーバー・プロトコルである。

**ローカル・エリア・ネットワーク (Local area network (LAN)).** 限定された地域内でユーザーの構内に配置されたコンピューター・ネットワーク。

**メッセージ (message).** メッセージ・キューイング・アプリケーションにおいて、メッセージはプログラム間で送信される通信である。

**メッセージ・キュー (message queue).** 「キュー (queue)」を参照。

**メッセージ・キューイング (message queuing).** アプリケーションの各プログラムが、メッセージをキューに書き込んで他のプログラムと通信するプログラミング技法。

**メソッド (method).** メソッドとは、関数またはプロシージャーに対するオブジェクト指向プログラミングの用語である。

**MQI チャネル (MQI channel).** MQI チャネルは、MQSeries クライアントをサーバー・システムのキュー・マネージャーに接続し、MQI 呼び出しと応答を両方向に転送する。

**MQSeries.** MQSeries は、メッセージ・キューイング・サービスを提供する IBM ライセンス・プログラムのファミリーである。

**オブジェクト (object).** (1) Java において、オブジェクトはクラスのインスタンスである。クラスはあるもののグループをモデル化し、オブジェクトはそのグループの特定のメンバーをモデル化する。 (2) MQSeries において、オブジェクトはキュー・マネージャー、キュー、またはチャネルである。

**パッケージ (package).** Java においてパッケージは、一部の Java コードが特定のクラス・セットにアクセスできるようにする方法である。特定のパッケージの一部である Java コードは、そのパッケージのすべてのクラス、およびそのクラスのプライベートではないすべてのメソッドとフィールドにアクセスする。

**携帯情報端末 (personal digital addistant (PDA)).** ポケット・サイズのパーソナル・コンピューター。

**プライベート (private).** プライベート・フィールドはそれ自身のクラス以外からは見ることができない。

**保護 (protected).** 保護フィールドはそれ自身のクラス内、サブクラス内、またはクラスが含まれているパッケージ内からのみ見ることができる。

**パブリック (public).** パブリック・クラスまたはインターフェースはどこからでも見ることができる。パブリック・メソッドまたは変数は、そのクラスを見ることができるところからであれば見ることができる。

**キュー (queue).** キューは MQSeries オブジェクトである。メッセージ・キューイング・アプリケーションは、キューにメッセージを書き込んだり、またキューからメッセージを読み取ることができる。

**キュー・マネージャー (queue manager).** キュー・マネージャーはアプリケーションにメッセージ・キューイング・サービスを提供するシステム・プログラムである。

**サーバー (server).** (1) MQSeries Everyplace サーバーは、構成された MQSeries Everyplace チャネル・マネージャーを持つデバイスである。 (2) MQSeries サーバーは、リモート・ワークステーション上で実行するクライアント・アプリケーションにメッセージ・キューイング・サービスを提供するキュー・マネージャーである。 (3) より一般的には、サーバーとはクライアント / サーバーの 2 プログラム間情報フロー・モデルにおいて、情報の要求に応答するプログラムである。 (4) サーバー・プログラムが実行されるコンピューター。

**サーブレット (servlet).** Web サーバー上でのみ実行するように設計された Java プログラム。

**サブクラス (subclass).** サブクラスとは、別のクラスへ拡張するクラスである。サブクラスは、それ自身のスーパークラスのパブリックおよび保護のメソッドと変数とを継承する。

**スーパークラス (superclass).** スーパークラスは他の何らかのクラスによって拡張されたクラスである。スーパークラスのパブリックおよび保護のメソッドと変数とは、そのサブクラスで使用可能である。

**同期メッセージング (synchronous messaging).** プログラム間で通信するときのメソッドの一つで、このメソッドを使用してプログラムはメッセージをメッセージ・キューに入ることができる。同期メッセージングを使用すると、送信側プログラムはそのメッセージに対する応答を待ってから、自分自身の処理を再開する。*同期メッセージング (asynchronous messaging)* と対比。

**伝送制御プロトコル / インターネット・プロトコル (Transmission Control Protocol/Internet Protocol (TCP/IP)).** ローカル・エリア・ネットワークと広域ネットワークの両方に対して、対等通信接続機能をサポートする通信プロトコルのセット。

**Web.** 「ワールド・ワイド・ウェブ (World Wide Web)」を参照。

**Web ブラウザー (Web browser).** ワールド・ワイド・ウェブ上で配布される情報をフォーマットおよび表示するプログラム。

**ワールド・ワイド・ウェブ (World Wide Web (Web)).** ワールド・ワイド・ウェブは、プロトコルの共通セットに基づいたインターネット・サービスで、特別に構成されたサーバー・コンピューターが標準的な方法でインターネットを介して文書を配布できるようにする。

# 参照文献

関連資料:

- *MQSeries Everyplace 紹介*, (GC88-8653-00)
- *MQSeries Everyplace プログラミング・リファレンス*, (SC88-8655-00)
- *MQSeries An Introduction to Messaging and Queuing*, (GC33-0805-01)
- *MQSeries (Windows NT 版) インストールの手引き V5.1*, (GD88-7162-00)

# 索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

## [ア行]



## [カ行]

# ［サ行］



# ［タ行］



# ［ナ行］

## ［ハ行］



## ［マ行］



## ［ヤ行］



## ［ラ行］

## A



## E



## J



## M



## P



## R



## S



## W

Printed in Japan

SC88-8654-00

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12